UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

Suporte ao desenvolvimento e uso de frameworks e componentes

por

RICARDO PEREIRA E SILVA

Tese submetida à avaliação como requisito parcial
para a obtenção do grau de
Doutor em Ciência da Computação

Prof. Roberto Tom Price
Orientador

Porto Alegre, março de 2000

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Silva, Ricardo Pereira e
Suporte ao desenvolvimento e uso de frameworks e componentes / por Ricardo Pereira e Silva. - Porto Alegre: PPGC da UFRGS, 2000.
262f.:il.

Tese (doutorado) - Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2000. Orientador: Price, Roberto Tom.

1. Frameworks orientados a abjetos 2. Desenvolvimento baseado em componentes 3. Ambientes de desenvolvimento de software 4. Reuso de software 5. Projeto orientado a objetos. I. Price, Roberto Tom. II. Título.

Agradecimentos

A Deus, por permitir a conclusão, com êxito, de tão importante empreendimento.

Ao meu orientador, Professor Roberto Tom Price, por partilhar comigo seu tempo e sua competência. Fica aqui registrado que foi um prazer e uma honra trabalhar a seu lado nos últimos anos.

À minha esposa, Valdirene, e a meus filhos, João Ricardo e Henrique, que, em função de meu trabalho, mudaram a rotina de suas vidas e me apoiaram ao longo de todo o doutorado.

À Universidade Federal de Santa Catarina - em particular, ao Departamento de Informática e de Estatística - e à CAPES, pelo suporte à realização do doutorado.

Ao Instituto de Informática da Universidade Federal do Rio Grande Do Sul, pela acolhida. Em particular, aos professores e funcionários do Instituto, com quem convivi ao longo dos últimos anos - todos muito prestativos, corteses e eficientes.

Sumário

Lista de Abreviaturas

- ADS - ambiente de desenvolvimento de software
- CASE - computer aided Software Engineering (Engenharia de Software auxiliada por computador)
- ER - entidade-relacionamento
- LOTOS - Language of Temporal Ordering Specifications (Linguagem de Especificações Ordenadas no Tempo)
- OMT - Object Modelling Technique (Técnica de Modelagem de Objetos)
- OOA - object-oriented analysis (análise orientada a objetos)
- OOAD - object-oriented analysis and design (análise e projeto orientados a objetos)
- OOD - object-oriented design (projeto orientado a objetos)
- OOP - object-oriented programation (programação orientada a objetos)
- OOSE - Object-Oriented Software Engineering (Engenharia de Software Orientada a Objetos)
- SGBD - sistema de gerenciamento de base de dados
- UML - Unified Modelling Language (Linguagem de Modelagem Unificada)
- VDM - Viena Development Method (Método de Desenvolvimento de Viena)

Lista de Figuras

Lista de Tabelas

Resumo

Frameworks orientados a objetos e desenvolvimento baseado em componentes são abordagens que promovem reuso de projeto e código em um nível de granularidade elevado. Por outro lado, a ampla adoção destas abordagens é dificultada pela complexidade para desenvolver e para usar frameworks e componentes. Este trabalho apresenta o framework OCEAN e o Ambiente SEA, que são artefatos dirigidos às necessidades de frameworks e componentes. OCEAN e SEA suportam uma abordagem de desenvolvimento baseada em notação de alto nível e na automatização de ações de edição.

O framework OCEAN suporta a construção de ambientes de desenvolvimento de software, que manuseiam especificações de projeto. O framework possui a definição genérica de funcionalidades para produzir e alterar especificações. Também é capaz de suportar a produção de mecanismos funcionais a serem vinculados a um ambiente, para a manipulação de especificações

O ambiente SEA foi desenvolvido como uma extensão da estrutura do framework OCEAN, usando a totalidade dos recursos deste framework. SEA suporta o desenvolvimento e o uso de frameworks e componentes. O desenvolvimento de um framework, um componente ou uma aplicação neste ambiente consiste em construir a especificação de projeto do artefato, utilizando UML, e, de forma automatizada, verificar sua consistência e convertê-la em código.

Uma característica importante de SEA é que a informação referente à flexibilidade é registrada em especificações de projeto de frameworks. Outra característica importante é a capacidade de manter ligação semântica entre um artefato de software e o framework que o originou. SEA permite que partes de um artefato de software produzido a partir de um framework sejam incorporadas ao framework. O ambiente suporta a criação de estruturas de padrões de projeto e a sua inclusão em uma biblioteca de padrões. Suporta também a inserção semi-automática de padrões de projeto em especificações em desenvolvimento, a partir da importação de padrões previamente inseridos na biblioteca do ambiente. No ambiente, *cookbooks* ativos podem ser construídos ou usados para orientar o uso de frameworks. Um *cookbook* ativo é avaliado a partir da comparação de sua estrutura com a especificação do framework por ele tratado. Dado um framework, um *cookbook* ativo que o descreva e um artefato de software sendo produzido a partir do framework, partes da especificação do artefato podem ser geradas automaticamente através da ativação de *links* do *cookbook*.

O desenvolvimento de software baseado em componentes no ambiente SEA ocorre em três etapas. A primeira consiste na especificação de interface de componente, que usa rede de Petri para especificar dinâmica comportamental. A segunda etapa consiste na especificação de componente, usando UML. O ambiente suporta a construção automática de parte da especificação de um componente, a partir da importação de uma especificação de interface. Em uma terceira etapa, componentes com interfaces compatíveis são interligados, produzindo arquiteturas de componentes.

Os recursos providos por SEA possibilitam a aplicação simultânea das abordagens frameworks orientados a objetos e desenvolvimento baseado em componentes.

Palavras-chave: reuso de software, framework orientado a abjetos, desenvolvimento baseado em componentes, ambientes de desenvolvimento de software

TITLE: "SUPPORT FOR THE DEVELOPMENT AND USE OF FRAMEWORKS AND COMPONENTS"

Abstract

Object-oriented frameworks and component-oriented software development are approaches that promote design and code reuse at a higher granularity level. On the other hand, the widespread adoption of these aproaches is made difficult by the complexity to develop and to use frameworks and components. This work presents the framework OCEAN and the environment SEA, which are software artifacts oriented to framework and component needs. OCEAN and SEA support a development approach based on high-level notation and on edition automation.

The framework OCEAN supports the building of software development environments, which handle design specifications. The framework has the generic definition of the functionalities for generating and changing specifications. It is also able to produce functional devices to be attached to an environment for handling specifications.

The environment SEA was developed as an extension of the structure of the framework OCEAN, by thoroughly using its resources. SEA supports development and use of frameworks and components. The development of a framework, a component or an application in this environment consists of first building the artifact design specification using UML, and then checking its consistence and translating it into code, automatically.

An important feature of SEA is that the information regarding flexibility is recorded in framework design specifications. Another important characteristic is the ability to keep a semantic *link* between a software artifact and the framework from which it is produced. SEA allows that parts of a software artifact be incorporated into the generating framework. The environment supports the creation of *design pattern* structures and their inclusion in a pattern library. It also allows the semi-automatic insertion of *design patterns* in design specifications, by importing *patterns* from the environment library. Within the environment, active *cookbooks* can be built or employed for guiding framework use. An active *cookbook* is evaluated by comparing its structure with the structure of the handled framework. Given a framework, an active *cookbook* that describes it and an artifact being developed from the framework, parts of the artifact specification can be automatically generated by activating *cookbook links*.

In SEA, component-based software development is split in three phases. The first phase is the specification of component interface, which uses a Petri net to describe interface control flow. The second phase is the specification of the component, using UML. The environment supports automatic construction of part of the component specification by importing an interface specification. In the third phase, components with compatible interfaces are linked, resulting in component architectures.

The functionalities provided by SEA allow the simultaneous application of both approaches, object-oriented frameworks and component-oriented software development.

Keywords: software reuse, object-oriented frameworks, component-oriented software development, software development environments.

1 Introdução

No contexto da Engenharia de Software, diferentes abordagens buscam melhorar a qualidade dos artefatos de software[1], bem como diminuir o tempo e o esforço necessários para produzi-los. Frameworks são estruturas de classes que constituem implementações incompletas que, estendidas, permitem produzir diferentes artefatos de software. A grande vantagem desta abordagem é a promoção de reuso de código e projeto, que pode diminuir o tempo e o esforço exigidos na produção de software. Em contrapartida, é complexo desenvolver frameworks, bem como aprender a usá-los. A abordagem de frameworks pode se valer de padrões para a obtenção de estruturas de classes bem organizadas e mais aptas a modificações e extensões. O desenvolvimento orientado a componentes pretende organizar a estrutura de um software como uma interligação de artefatos de software independentes, os componentes. O reuso de componentes previamente desenvolvidos, como no caso dos frameworks, permite a redução de tempo e esforço para a obtenção de um software. Por outro lado, devido a deficiências das formas correntes de descrição de componentes, é complexo avaliar a adequação de um componente existente a uma situação específica. Também é complexo adaptar componentes quando estes não estejam exatamente adequados a necessidades específicas. Em Arquitetura de Software estudam-se formas de registrar a experiência de organização estrutural de software. O reuso desta experiência pode facilitar a definição da organização mais adequada dos componentes de uma aplicação, complementando a abordagem de desenvolvimento baseado em componentes. O presente trabalho se preocupa com a integração dessas abordagens, que têm em comum a reutilização, seja de projeto ou de código, isto é, o aproveitamento de experiência de desenvolvimento de software.

A Engenharia de Software atua na aplicação de abordagens sistemáticas ao desenvolvimento e manutenção de software. Os objetivos principais da Engenharia de Software são a melhora da qualidade do software e o aumento da produtividade da atividade de desenvolvimento de software [FAI 86]. A reutilização de software, em contraposição ao desenvolvimento de todas as partes de um sistema, é um fator que pode levar ao aumento da qualidade e da produtividade da atividade de desenvolvimento de software. Isto se baseia na perspectiva de que o reuso de artefatos de software já desenvolvidos e depurados, reduza o tempo de desenvolvimento, de testes e as possibilidades de introdução de erros na produção de novos artefatos. A reutilização de artefatos de software em larga escala é um dos argumentos a favor da abordagem de orientação a objetos. Em muitos casos porém, constitui uma perspectiva frustrada, pois a reutilização não é característica inerente da orientação a objetos, mas é obtida a partir do uso de técnicas que produzam software reutilizável [JOH 88] [PRE 94].

A produção de artefatos de software reutilizáveis se coloca como um requisito para que ocorra reutilização de software. Os princípios que norteiam a modularidade, como a criação de módulos com interfaces pequenas, explícitas e em pequena quantidade, e o uso do princípio da ocultação de informação [MEY 88], têm sido ao longo dos últimos anos o principal elemento de orientação para a produção de bibliotecas de artefatos reutilizáveis.

---

[1] A expressão *artefato de software* é usada aqui de forma genérica, não se referindo necessariamente a código (podendo abranger os produtos da análise e do projeto). Além disto, pode se referir a aplicação, framework ou componente.

A reutilização de artefatos de software pode abranger o nível de código, como também os de análise e projeto. A reutilização de código consiste na utilização direta de trechos de código já desenvolvidos. A reutilização de projeto consiste no reaproveitamento de concepções arquitetônicas de um artefato de software em outro artefato de software, não necessariamente com a utilização da mesma implementação.

A forma mais elementar de reutilização de artefatos de software é a reutilização de trechos de código de aplicações já desenvolvidas. Para sistematizar esta prática são criadas bibliotecas - como bibliotecas de funções em linguagem C, por exemplo. Um problema encontrado com o uso de bibliotecas de artefatos de software é a dificuldade de seleção de um artefato adequado à aplicação em desenvolvimento. Se o desenvolvedor concluir que é mais fácil produzir um novo artefato do que buscar um em uma biblioteca, não haverá reutilização [NEI 89].

Observa-se que a reutilização dos produtos das etapas de análise e projeto é mais significativa[2] que a reutilização de trechos de código [DEU 89]. Observa-se na literatura uma convicção de que uma abordagem voltada à reutilização só alcança resultados significativos se estiver voltada a um domínio específico [HEN 95] [SIM 95] [NEI 89] [ARA 91]. Neighbors afirma que a informação contida em produtos das etapas de análise e projeto é específica do domínio de aplicações tratado [NEI 89]. Verificam-se dois objetivos distintos, na reutilização no nível das especificações de análise e projeto:

- Produzir uma aplicação utilizando o projeto de outra aplicação. Isto é o que ocorre em um processo de Reengenharia, em que a especificação de uma aplicação é alterada, de modo a adaptar-se a novos requisitos, e é gerada uma nova aplicação.
- Produzir uma especificação adequada a um conjunto de aplicações (um domínio) e reutilizar esta especificação no desenvolvimento de diferentes aplicações. Esta é a abordagem dos frameworks orientados a objetos.

Um framework[3] é uma estrutura de classes interrelacionadas, que corresponde a uma implementação incompleta para um conjunto de aplicações de um domínio. Esta estrutura de classes deve ser adaptada para a geração de aplicações específicas [JOH 92].

A granularidade do artefato de software reutilizado é um fator relevante em termos de aumento de produtividade no desenvolvimento de software. Um programador que usa linguagem C, por exemplo se utiliza de bibliotecas de funções. Isto se classifica como reutilização de rotinas[4] [MEY 88]. A reutilização no nível de módulo corresponde a um nível de granularidade superior à reutilização de rotinas. A reutilização de classes em orientação a objetos, segundo Meyer, corresponde à reutilização de módulo. Quando uma classe é reutilizada, um conjunto de rotinas é reutilizado (métodos), bem como uma estrutura de dados (atributos). Uma classe que implementa uma estrutura de dados pilha, por exemplo, contém esta estrutura de dados e o conjunto de procedimentos para a sua manipulação. Reutilizar classes, portanto, tende a ser mais eficiente que reutilizar rotinas[5], e de mais alto nível de granularidade (uma classe de objetos tende a ser um artefato de software mais complexo que uma rotina isolada). Meyer em 1988 enfatizava a necessidade da disponibilidade de artefatos genéricos e afirmava que "as classes de

---

2 Em termos de diminuição do esforço e do tempo necessários para o desenvolvimento de uma aplicação, o que significa aumento de produtividade.

3 No presente trabalho a expressão *framework* se refere a *framework orientado a objetos*.

4 A reutilização de rotina pode ocorrer no nível de projeto quando, por exemplo, um programador se utiliza do projeto de uma rotina já implementada para gerar uma implementação mais eficiente. A reutilização pode se dar no nível de projeto ou código (ou ambos) independente de granularidade.

5 Em termos de produtividade, no desenvolvimento de software.

linguagens orientadas a objetos podem ser vistas como módulos de projeto bem como módulos de implementação" [MEY 88]. A reutilização de classes pode ocorrer sob dois enfoques diferentes:

- composição: consiste em usar as classes disponíveis em bibliotecas para originar os objetos da implementação;
- herança: consiste em aproveitar a concepção de classes disponíveis em bibliotecas, no desenvolvimento de outras, a partir de herança.

Uma característica comum à reutilização de rotinas e à reutilização de classes é que cabe a quem desenvolve a aplicação estabelecer a arquitetura da aplicação: a organização dos elementos que compõem a aplicação e o fluxo de controle entre eles. Além disto, há em comum o fato da reutilização ser de artefatos isolados, cabendo ao desenvolvedor estabelecer sua conexão ao sistema em desenvolvimento. Em um programa C o programador determina a chamada de funções; em um programa Smalltalk o programador procede a interligação das classes (desenvolvidas e reutilizadas).

A reutilização promovida pela abordagem de frameworks se situa num patamar de granularidade superior à reutilização de classes, por reusar classes interligadas, ao invés de isoladas. A reutilização de código e projeto em um nível de granularidade superior às outras abordagens citadas, confere aos frameworks o potencial de contribuir mais significativamente com o aumento de produtividade no desenvolvimento de software.

O desenvolvimento de software orientado a componentes é uma outra abordagem que promove o reuso de artefatos de alta granularidade, com a capacidade de promover reuso de projeto e código. Nesta abordagem uma aplicação é constituída a partir de um conjunto de módulos (componentes) interligados. Na medida em que parte das responsabilidades de uma aplicação seja delegada a um componente reutilizado, estar-se-á promovendo uma diminuição do esforço necessário para produzir esta aplicação, o que caracteriza aumento de produtividade - semelhante ao que ocorre com os frameworks.

Um componente é "uma unidade de composição com interfaces contratualmente especificadas e dependências de contexto explícitas" [SZY 96b]. Componentes podem ser desenvolvidos utilizando o paradigma de orientação a objetos, isto é, como estruturas de classes interrelacionadas, com visibilidade externa limitada [KRA 97], porém, no caso geral, componentes não dependem de tecnologia de implementação. Assim, qualquer artefato de software pode ser considerado um componente, desde que possua uma interface definida [SZY 97].

Observa-se em comum nas abordagens de desenvolvimento baseadas em frameworks e em componentes a possibilidade de reutilização de artefatos de software de alta granularidade, e a caracterização de reuso de projeto e código. A adoção destas abordagens no desenvolvimento de software - em conjunto ou isoladamente - pode diminuir significativamente o tempo para o desenvolvimento de novas aplicações, bem como a quantidade de código a desenvolver, na medida em que parte das responsabilidades das novas aplicações são atribuídas a artefatos de software reutilizados.

1.1 O Problema

Desenvolvimento baseado em frameworks e em componentes apresenta vantagens em relação a abordagens tradicionais de desenvolvimento de software, porém há obstáculos que dificultam a sua aplicação.

A abordagem de frameworks apresenta como principais problemas a complexidade de desenvolvimento e a complexidade de uso. Complexidade de desenvolvimento diz respeito à necessidade de produzir uma abstração de um domínio de aplicações adequada a diferentes aplicações, e além disto, dotada de flexibilidade. Complexidade de uso diz respeito ao esforço requerido para aprender a desenvolver aplicações a partir de um framework.

Metodologias de desenvolvimento de frameworks existentes, em sua maioria, estão voltadas a produzir código e não utilizam notação de alto nível, fazendo com que o resultado do processo de desenvolvimento seja o código do framework - um modelo de domínio expresso através de uma linguagem de programação [JOH 97]. Esta inexistência de mecanismos de descrição que registrem as decisões ao longo do processo de desenvolvimento torna mais complexo tanto este processo, como também o uso e a manutenção do framework resultante - em função da dificuldade de entender um software a partir de seu código. Uma solução possível para este problema seria a adoção das técnicas de modelagem das metodologias de análise e projeto orientado a objetos (metodologias OOAD). Por outro lado, as notações das metodologias OOAD existentes não são adequadas a frameworks, porque são incapazes de representar conceitos específicos desta abordagem.

A documentação mais comum de frameworks existentes consiste em descrição textual, código do framework e código de exemplos de aplicações desenvolvidas a partir do framework. Mecanismos auxiliares de descrição propostos pelos autores de metodologias abrangem fundamentalmente a questão de como utilizar os frameworks, como é o caso dos padrões de documentação de Johnson [JOH 92], dos metapadrões de Pree [PRE 94] e dos contratos de Helm [HEL 90]. Estas abordagens têm em comum o objetivo de destacar as partes da estrutura de classes de um framework que devem ser estendidas para a construção de aplicações, isto é, as partes flexíveis de um framework. Elas são incapazes de descrever completamente o projeto de frameworks, impossibilitado o entendimento global de um framework a partir de suas notações.

O desenvolvimento de um framework corresponde a uma evolução iterativa de sua estrutura de classes. Este refinamento cíclico também é motivado pela busca de adaptação da estrutura de classes, aos aspectos de generalidade e flexibilidade, e implica no manuseio de uma grande quantidade de informações. Assim, a adoção de mecanismos de descrição que apoiem o processo de desenvolvimento pode diminuir a dificuldade de desenvolver frameworks, bem como produzir um registro do projeto útil para a utilização e a manutenção de frameworks.

A abordagem de desenvolvimento orientado a componentes, semelhante à abordagem de frameworks, apresenta obstáculos relacionados ao desenvolvimento e uso de componentes. A dificuldade de desenvolvimento está associada à necessidade de produzir um artefato de software com uma clara distinção entre sua estrutura interna e sua interface, voltado a interagir com elementos externos, não necessariamente conhecidos quando do desenvolvimento de um componente. Esta necessidade de prever a comunicação com o meio externo exige que uma interface de componente seja precisamente especificada, estrutural e comportamentalmente, de modo a registrar as restrições na comunicação do componente com o meio externo. A necessidade de

desenvolvimento de interface e estrutura interna como entidades distintas torna o ciclo de vida de um componente diferente do ciclo de vida tradicional de uma aplicação. Uma mesma especificação de interface, por exemplo pode dar origem a um conjunto de componentes estrutural e comportamentalmente semelhantes, porém, funcionalmente diferentes. A distinção entre definição de interface e definição da estrutura interna necessita ser considerada por abordagens metodológicas voltadas à produção de componentes.

A dificuldade de uso está relacionada com a inflexibilidade dos componentes e com a dificuldade de entender um componente - que funcionalidades provê, como se comunica com o meio externo. É difícil que um componente esteja apto a ser amplamente reusado tal qual foi desenvolvido. Assim, geralmente componentes necessitam ser adaptados antes de serem utilizados [BOS 97]. A dificuldade de compatibilizar componentes originalmente incompatíveis constitui um problema da abordagem de desenvolvimento orientado a componentes, sugerindo a necessidade de mecanismos que facilitem a adaptação de componentes.

A falta de formas de especificar componentes amplamente aceitas, produz conseqüências tanto no desenvolvimento quanto no uso de componentes. Observa-se que diferentes abordagens de descrição de componentes apresentam diferentes tipos de deficiência. Mecanismos de descrição de interface existentes, em geral, são pobres para descrever componentes porque produzem apenas uma visão externa incapaz de descrever suas funcionalidades e como estes interagem [HON 97]. Abordagens formais produzem descrições de interoperabilidade de difícil compreensão, que não descrevem a funcionalidade dos componentes. Em geral, as abordagens de descrição não conseguem abranger o conjunto mínimo de informações necessário para compreender um componente: descrição de sua funcionalidade e descrição estrutural e comportamental de sua interface. Esta dificuldade em compreender componentes a partir dos mecanismos de descrição propostos dificulta seu reuso, bem como o desenvolvimento de mecanismos de armazenamento, busca e seleção de componentes [MIL 97]. Como no caso dos frameworks, a inexistência de mecanismos de descrição adequados a componentes que registrem as decisões ao longo do processo de desenvolvimento torna mais complexo este processo. Observa-se assim, na abordagem de componentes, a necessidade de definição de mecanismos de descrição que, assim como no caso dos frameworks, apoiem seu desenvolvimento, manutenção e uso.

As abordagens de frameworks e componentes têm em comum o potencial de promoção de reuso mas, por outro lado, apresentam obstáculos relacionados ao seu desenvolvimento e uso, que dificultam a ampla adoção destas abordagens. Estes obstáculos se sobrepõem se essas abordagens de desenvolvimento de software forem aplicadas conjuntamente.

1.2 A Tese

Considerando as vantagens das abordagens de desenvolvimento baseadas em frameworks e em componentes, bem como suas dificuldades, a hipótese básica que norteia esta tese é que um ambiente de desenvolvimento de software que suporte as necessidades específicas do desenvolvimento e do uso de frameworks e componentes constitui um auxílio importante para a exploração da potencialidade de reuso dessas abordagens. Um ambiente com esta finalidade deve apresentar as seguintes capacidades:

- suporte ao uso de notação de alto nível;

- registro da flexibilidade associada a um framework em sua especificação;
- suporte à produção de artefatos de software a partir do uso de um framework;
- suporte à alteração de um framework a partir de artefatos produzidos usando este framework;
- suporte à produção e à alteração de frameworks a partir de aplicações do domínio tratado;
- suporte ao desenvolvimento de componentes, considerando o desenvolvimento de interfaces e de estruturas internas de componentes como atividades distintas;
- suporte à construção de artefatos de software a partir da interconexão de componentes;
- suporte à aplicação conjunta das abordagens de frameworks e componentes, permitindo a utilização de frameworks para o desenvolvimento de componentes e vice-versa;
- suporte ao uso de padrões de projeto em especificações de artefatos de software;
- suporte ao uso de padrões arquitetônicos em especificações de artefatos de software;
- suporte a funções de edição semântica de especificações de projeto;
- geração de código a partir de especificações de projeto;
- suporte a procedimentos de Engenharia Reversa que permitam compor especificações a partir de código;
- suporte a procedimentos de transformação[6].

Visando produzir um suporte flexível e extensível para apoiar o desenvolvimento e o uso de frameworks e componentes, foi desenvolvido o framework OCEAN para suportar a construção de ambientes de desenvolvimento de software e, a partir da extensão de sua estrutura de classes, o ambiente SEA, para desenvolvimento e uso de frameworks e componentes.

O framework OCEAN fornece suporte para a construção de ambientes de desenvolvimento de software. Ambientes gerados sob este framework manuseiam *especificações de projeto*, que são documentos constituídos de outros documentos, os *modelos*. Modelos são documentos que agregam *conceitos*, que são as unidades de modelagem do dominio tratado[7]. A estrutura de uma especificação é definida pelos tipos de modelo e conceito tratados e também pelo registro das associações semânticas entre estes elementos. Documentos definidos sob o framework OCEAN têm sua estrutura baseada em Model-View-Controller (MVC). Assim, documentos possuem a definição de sua estrutura em separado de sua apresentação visual. Documentos podem ainda possuir *links* que apontem para outros documentos. Deste modo, um ambiente desenvolvido sob o framework OCEAN atua como um *browser*, que possibilita diferentes caminhos de navegação através dos documentos que compõem especificações. O framework possui a definição de mecanismos genéricos para produzir e alterar especificações, bem como

---

6 Funções de transformação podem se ater ou não às informações manipuladas pelo ambiente considerado. Um exemplo de transformação restrita aos limites de um ambiente consiste em alterar o tipo de uma especificação, o que pode corresponder, por exemplo, a transformar uma especificação de aplicação em uma especificação de framework. Um exemplo de transformação restrita aos limites de um ambiente, envolvendo os elementos que compõem uma especificação, é a conversão de uma associação binária em agregação. Funções de transformação que excedem os limites de um ambiente convertem especificações em estruturas de informação manipuláveis por mecanismos externos ao ambiente ou vice-versa. São exemplos deste tipo de transformação, a geração de código, a construção de especificação a partir de código (procedimento de Engenharia Reversa) e transformações envolvendo dados manipulados por outros ambientes, como a conversão de uma especificação em um arquivo de dados no formato de outro ambiente e a transformação inversa.

7 No caso de UML, são exemplos de conceito, classe, estado, caso de uso, mensagem etc.

para produzir diferentes mecanismos funcionais a serem vinculados a um ambiente, para a manipulação de especificações. As principais contribuições introduzidas pelo framework OCEAN, no aspecto de construção de ambientes de desenvolvimento de software, são:

- a definição de uma organização de classes que suporta a construção de ambientes que manipulem diferentes tipos de especificação e que disponham de diferentes conjuntos de funcionalidades;
- a produção de uma plataforma de experimentos de abordagens de modelagem e de funcionalidades relacionadas ao desenvolvimento de software.

A partir da extensão da estrutura de classes do framework OCEAN foi produzido o ambiente SEA, específico para o desenvolvimento e uso de frameworks e componentes. Neste ambiente, frameworks, componentes e aplicações são desenvolvidos como especificações de projeto baseadas no paradigma de orientação a objetos, isto é, como estruturas de classes. A interface para a construção de especificações são editores gráficos, através dos quais são construídos os diferentes modelos que compõem uma especificação. O conjunto de técnicas de modelagem suportado pelo ambiente SEA para a construção de especificações baseadas no paradigma de orientação a objetos corresponde a um subconjunto das técnicas de UML[8] [RAT 97], com modificações, para se adequarem às necessidades de frameworks e componentes.

O ambiente SEA dispõe de mecanismo de verificação de consistência de especificações. Dependendo do tipo de inconsistência encontrado, o ambiente pode atuar: impedindo que o erro seja produzido, indicando erros, corrigindo erros automática ou semi-automaticamente. Especificações cuja consistência tenha sido verificada podem ser traduzidas em código. O ambiente SEA possui um gerador de código para linguagem Smalltalk, que produz código de forma semi-automática, a partir de especificações de projeto.

Como os documentos sob o framework OCEAN são baseados em MVC, todo o processo de construção e modificação de especificações do ambiente SEA ocorre sobre a estrutura conceitual das especificações (correspondente a *model*), e se refletem em sua apresentação visual (correspondente a *view*). Utilizando este princípio, algumas funcionalidades de edição semântica foram incluídas no ambiente, como a possibilidade de fundir elementos de uma especificação, a cópia de elementos em uma área de transferência e a posterior colagem destes elementos em alguma outra parte da especificação (ou em outra especificação).

O ambiente permite manter ligação entre especificações distintas, como é necessário entre uma especificação de um artefato de software produzida a partir de um framework e e especificação deste framework. Com isto, classes de uma especificação de framework podem ser importadas para a especificação de uma aplicação produzida sob este framework como um elemento externo, possibilitando, por exemplo, expandir a cadeia hierárquica de herança. O ambiente suporta alteração simultânea de especificações distintas, como a migração de uma classe de uma especificação de aplicação desenvolvida sob um framework, para a especificação deste framework.

SEA suporta a criação de estruturas de padrões de projeto e a sua inclusão em uma biblioteca de padrões. Suporta também a inserção semi-automática de padrões de projeto em especificações de projeto em desenvolvimento no ambiente, a partir da importação das estruturas mantidas na biblioteca de padrões.

---

8 E mais uma técnica de modelagem não prevista em UML para a descrição dos algoritmos dos métodos, baseada no diagrama de ação [MAR 91].

O ambiente dispõe de mecanismo para a construção e uso de *cookbooks* ativos[9] de orientação ao uso de frameworks. Além de instruções textuais, um *cookbook* ativo do ambiente SEA, dispõe de *links* para navegação, que permitem a visualização de suas páginas, bem como de documentos de diferentes especificações. Apresenta também *links* ativos, responsáveis pela criação automática ou semi-automática de partes de uma especificação - dispensando o usuário do framework de parte das tarefas do desenvolvimento de especificações. O ambiente dispõe de mecanismo que avalia a estrutura de um *cookbook* ativo, confrontando-a com o framework cujo uso é orientado. Este mecanismo também verifica se todas as flexibilidades definidas na estrutura do framework estão contempladas no *cookbook*. Também é possível avaliar se uma especificação gerada sob um framework obedece às restrições impostas por este framework. Este procedimento de análise, em conjunto com a verificação de consistência de especificações suportada pelo ambiente, permite avaliar se uma especificação desenvolvida sob um framework está apta à geração de código.

No aspecto de suporte ao desenvolvimento e uso de frameworks, SEA introduz as seguintes contribuições:

- a definição de notação específica para especificação de projeto de frameworks, através da extensão da notação UML;
- uma abordagem de inserção semi-automática de padrões de projeto em especificações de frameworks;
- uma abordagem de avaliação de qualidade de *cookbooks* ativos, bem como de avaliação de qualidade de especificações produzidas a partir do uso de *cookbooks* ativos.

No ambiente SEA foi adotado o paradigma de orientação a objetos para o desenvolvimento de componentes. Assim, componentes são definidos como estruturas de classes, utilizando o suporte que também apóia o desenvolvimento e o uso de frameworks. No ambiente SEA o desenvolvimento orientado a componentes atravessa três fases distintas. A primeira fase consiste na definição de interfaces de componentes, cujas especificações são armazenadas em biblioteca. Uma segunda etapa consiste na definição de componentes, que reusam interfaces previamente produzidas. A especificação de interfaces em separado da especificação de componentes permite que uma mesma especificação de interface possa ser reusada por diferentes componentes, produzindo assim uma família de componentes estrutural e comportamentalmente semelhantes. Em uma terceira etapa, componentes com interfaces compatíveis são interligados, produzindo arquiteturas de componentes. O ambiente suporta a importação de uma especificação de interface para a estrutura de uma especificação de componente e suporta a importação de componentes desenvolvidos para serem usados na construção de especificações de outros artefatos de software.

Especificamente no aspecto de suporte ao desenvolvimento e uso de componentes, SEA introduz as seguintes contribuições:

- adoção do modelo rede de Petri para especificar comportamentalmente interfaces de componentes, bem como arquiteturas de componentes, obtidas pela interconexão de componentes através de suas interfaces;

---

[9] No contexto da abordagem de frameworks, um cookbook (livro de receitas) é um manual de utilização de um framework, composto por um conjunto de "receitas", que estabelecem os passos a serem seguidos para produzir uma aplicação ou parte de uma aplicação, a partir do framework. Um exemplo é o cookbook do ambiente VisualWorks [PAR 94]. Um cookbook ativo é capaz de executar ações de edição, além de conter instruções.

- adoção da abordagem de frameworks orientados a objetos para a construção de componentes flexíveis.
- proposição de um padrão de projeto para a construção de interfaces de componentes.

1.3 Organização do Texto da Tese

A presente tese é constituída por oito capítulos e um anexo, cujos conteúdos estão descritos a seguir.

O capítulo 2 apresenta as características da abordagem de frameworks orientados a objetos. A abordagem é comparada a outras abordagens de Análise de Domínio, para uma avaliação comparativa de seu potencial de reutilização.

A questão do desenvolvimento de frameworks é apresentada no capítulo 3. São apresentadas as características buscadas no desenvolvimento de um framework, é descrito o ciclo de vida de um framework e são descritas e comparadas metodologias de desenvolvimento de frameworks. Também são apresentados mecanismos disponíveis de apoio à construção de frameworks.

No capítulo 4 é tratada a questão da compreensão e uso de frameworks. São discutidos os requisitos para aprender a usar um framework e os mecanismos de suporte disponíveis.

O capítulo 5 apresenta a abordagem de desenvolvimento orientado a componentes, em que sistemas são contruídos a partir da composição de componentes. São discutidos aspectos de construção e adaptação de componentes. Discute-se como o uso conjunto das abordagens de frameworks e componentes pode aumentar o potencial de reutilização de cada abordagem.

No capítulo 6 é apresentado o ambiente SEA, enfatizando o conjunto de funcionalidades provido e como o ambiente suporta o desenvolvimento e uso de frameworks e componentes.

No capítulo 7 é apresentado o framework OCEAN. A ênfase desse capítulo é mostrar como a arquitetura do framework o torna capaz de suportar a construção de ambientes que manipulem diferentes tipos de especificação e que disponham de diferentes conjuntos de funcionalidades.

O capítulo 8 apresenta o sumário das contribuições do presente trabalho, discute limitações existentes e propõe futuros trabalhos e linhas de pesquisa utilizando os resultados do trabalho desenvolvido.

No anexo 1 é apresentada a estrutura semântica de uma especificação baseada no paradigma de orientação a objetos do ambiente SEA, através de uma gramática de atributos - em que as regras semânticas associadas às produções da gramática são definidas a partir de lógica de primeira ordem.

2 Frameworks Orientados a Objetos, uma Abordagem de Análise de Domínio

A abordagem de frameworks orientados a objetos utiliza o paradigma de orientação a objetos para produzir uma descrição de um domínio para ser reutilizada. Um framework é uma estrutura de classes interrelacionadas, que corresponde a uma implementação incompleta para um conjunto de aplicações de um domínio. Esta estrutura de classes deve ser adaptada para a geração de aplicações específicas[10].

A. Wirfs-Brock [WIA 91] propõe a seguinte definição para framework: "um esqueleto de implementação de uma aplicação ou de um subsistema de aplicação, em um domínio de problema particular. É composto de classes abstratas e concretas e provê um modelo de interação ou colaboração entre as instâncias de classes definidas pelo framework. Um framework é utilizado através de configuração ou conexão de classes concretas e derivação de novas classes concretas a partir das classes abstratas do framework". R. Wirfs-Brock [WIR 90] estalelece que um framework é "uma coleção de classes concretas e abstratas e as interfaces entre elas, e é o projeto de um subsistema". Jonhson ressalta: "não apenas classes, mas a forma como as instâncias das classes colaboram" [JOH 93].

A diferença fundamental entre um framework e a reutilização de classes de uma biblioteca, é que neste caso são usados artefatos de software isolados, cabendo ao desenvolvedor estabelecer sua interligação, e no caso do framework, é procedida a reutilização de um conjunto de classes inter-relacionadas - inter-relacionamento estabelecido no projeto do framework. As figuras abaixo ilustram esta diferença (a parte sombreada representa classes e associações que são reutilizadas).

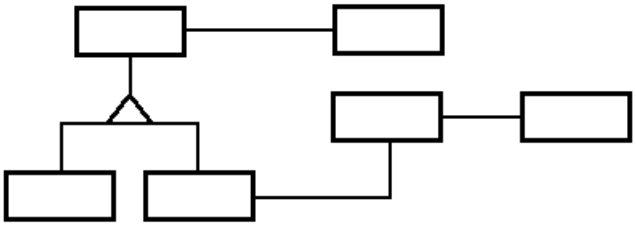

FIGURA 2.1 - Aplicação desenvolvida totalmente

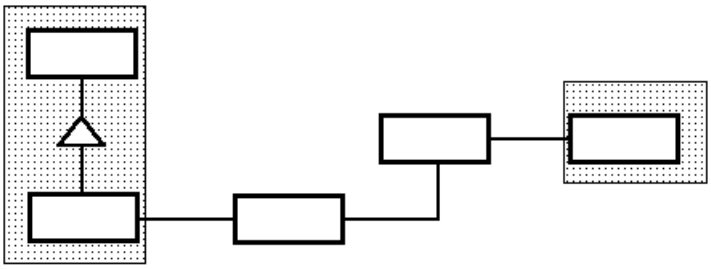

FIGURA 2.2 - Aplicação desenvolvida reutilizando classes de biblioteca

---

10 Um framework pode originar outros tipos de artefato de software, além de aplicações, como outros frameworks, por exemplo. Ao longo do presente trabalho, quando se afirma que um framework é voltado à produção de aplicações, está-se considerando que mesmo utilizando um framework para desenvolver outros tipos de artefato, o final da cadeia de desenvolvimento sempre resulta em aplicações.

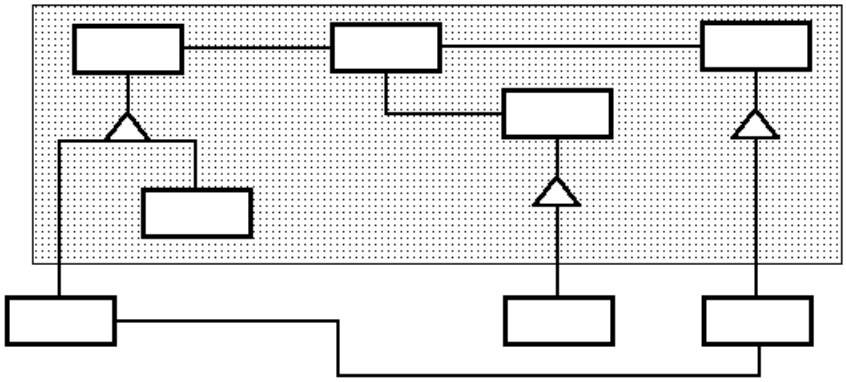

FIGURA 2.3 - Aplicação desenvolvida reutilizando um framework

Dois aspectos caracterizam um framework [TAL 95]:

- Os frameworks fornecem infraestrtura e projeto: frameworks portam infraestrutura de projeto disponibilizada ao desenvolvedor da aplicação, que reduz a quantidade de código a ser desenvolvida, testada e depurada. As interconexões preestabelecidas definem a arquitetura da aplicação, liberando o desenvolvedor desta responsabilidade. O código escrito pelo desenvolvedor visa estender ou particularizar o comportamento do framework, de forma a moldá-lo a uma necessidade específica.
- Os frameworks "chamam", não são "chamados": um papel do framework é fornecer o fluxo de controle da aplicação. Assim, em tempo de execução, as instâncias das classes desenvolvidas esperam ser chamadas pelas instâncias das classes do framework.

Um framework se destina a gerar diferentes aplicações para um domínio. Precisa, portanto, conter uma descrição dos conceitos deste domínio. As classes abstratas de um framework são os repositórios dos conceitos gerais do domínio de aplicação. No contexto de um framework, um método de uma classe abstrata pode ser deixado propositalmente incompleto para que sua definição seja acabada na geração de uma aplicação. Apenas atributos a serem utilizados por todas as aplicações de um domínio são incluídos em classes abstratas.

Os frameworks são estruturas de classes interrelacionadas, que permitem não apenas reutilização de classes, mas minimizam o esforço para o desenvolvimento de aplicações - por conterem o protocolo de controle da aplicação (a definição da arquitetura), liberando o desenvolvedor de software desta preocupação. Os frameworks invertem a ótica do reuso de classes, da abordagem *bottom-up*[11] para a abordagem *top-down*: o desenvolvimento inicia com o entendimento do sistema contido no projeto do framework, e segue no detalhamento das particularidades da aplicação específica, o que é definido pelo usuário do framework. Assim, a implementação de uma aplicação a partir do framework é feita pela adaptação de sua estrutura de classes, fazendo com que esta inclua as particularidades da aplicação [TAL 95].

Uma outra característica é o nível de granularidade de um framework. Frameworks podem agrupar diferentes quantidades de classes, sendo assim, mais ou menos complexos. Além disto, podem conter o projeto genérico completo para um domínio de aplicação, ou construções de alto nível que solucionam situações comuns em projetos. Deutsch admite framework de uma única classe (com esta nomenclatura) que consiste de "uma classe que fornece especificação e implementação parcial mas que

---

[11] Desenvolvimento bottom-upé o que ocorre, por exemplo, quando classes de objetos de uma biblioteca são interligadas, para a construção de uma aplicação.

necessita de subclasses ou parâmetros para completar a implementação". Frameworks com mais de uma classe são denominados frameworks de múltiplas classes [DEU 89].

Segundo a granularidade, Pree denomina frameworks de aplicação aqueles que constituem um projeto genérico para um domínio, e simplesmente frameworks aqueles que representam uma microarquitetura consistindo de poucos elementos, a ser usada como parte de um projeto [PRE 94]. Gamma adota uma classificação semelhante, porém com a denominação framework quando se trata do projeto genérico para um domínio, e *design pattern* (padrão de projeto) no caso de uma microarquitetura [GAM 94]. Ainda apresenta as seguintes diferenças entre frameworks e *design patterns*: *design patterns* são mais abstratos que frameworks, *design patterns* são menores (menos elementos arquitetônicos) que frameworks, *design patterns* são menos especializados que frameworks. No presente trabalho a expressão framework é usada independente de granularidade, porém, sempre supondo que a estrutura contenha mais de uma classe.

2.1 Aspectos da Geração de Aplicações a partir de um Framework

De acordo com a forma como deve ser utilizado, um framework pode se classificar como dirigido a arquitetura ou dirigido a dados. No primeiro caso a aplicação deve ser gerada a partir da criação de subclasses das classes do framework. No segundo caso, diferentes aplicações são produzidas a partir de diferentes combinações de objetos, instâncias das classes presentes no framework. Frameworks dirigidos a arquitetura são mais difíceis de usar, pois, a geração de subclasses exige um profundo conhecimento do projeto de um framework, bem como um certo esforço no desenvolvimento de código. Frameworks dirigidos a dados são mais fáceis de usar, porém são menos flexíveis. Uma outra abordagem seria uma combinação dos dois casos, onde o framework apresentaria uma base dirigida a arquitetura e uma camada dirigida a dados. Com isto, possibilita a geração de aplicações a partir da combinação de objetos, mas permite a geração de subclasses [TAL 94]. Johnson adota esta classificação de frameworks, porém usa a denominação caixa-branca para o framework dirigido a arquitetura e caixa-preta para o dirigido a dados [JOH 92]. A expressão caixa-cinza é utilizada para se referir à combinação dos dois casos.

Há portanto, duas maneiras de gerar aplicações a partir de um framework: a partir de combinações de instâncias das classes concretas do framework ou a partir da definição de classes - ou ainda a partir de combinações das duas possibilidades. A possibilidade de usar uma ou outra forma é uma característica particular de cada framework, definida em seu projeto. O projeto de um framework estabelece a flexibilidade provida, impondo restrições (de combinação de objetos ou de criação de novas classes) a serem consideradas pelos usuários. Um framework deve dispor de uma documentação específica para ensinar usuários a gerarem aplicações, estabelecendo estas restrições.

Na abordagem de uso de frameworks a partir da criação de classes fica imposto um conjunto de classes que deve ser definido pelo usuário (subclasses das classes de um framework). Para a criação de um editor de hipertexto a partir do framework ET++, por exemplo, o usuário necessita obrigatoriamente gerar, dentre outras, uma subclasse de *Application*, que conterá a sobreposição do método *DoMakeManager* (que dispara a execução da aplicação), e uma subclasse de *Document*, que conterá a definição do documento [PRE 94]. Também cabe ressaltar que além de um conjunto obrigatório de subclasses, o usuário não é totalmente livre para criar ou alterar funcionalidades a partir

da criação de classes, uma vez que o controle da aplicação é definido nas classes do framework e não nas classes do usuário. Por exemplo, um framework que gera aplicações para um domínio específico pode definir uma interface com o usuário padronizada (padrão Microsoft-Windows, ou algum outro) e não dar ao usuário a possibilidade de alterar esta característica.

Em termos ideais, um framework deve abranger todos os conceitos gerais de um domínio de aplicação, deixando apenas aspectos particulares para serem definidos nas aplicações específicas. No caso ideal, na geração de aplicações, o usuário do framework não precisa criar classes que não sejam subclasses de classes abstratas do framework. Se isto for alcançado, o framework terá conseguido de fato ser uma generalização do domínio modelado.

2.2 Um Exemplo: FraG, um Framework para Jogos de Tabuleiro

Como parte do trabalho de pesquisa aqui descrito, foi desenvolvido um pequeno framework para a geração de jogos de tabuleiro, FraG, para um exercício de investigação de aspectos referentes a desenvolvimento e utilização de frameworks. FraG é voltado a generalizar um conjunto de jogos que tem em comum o uso de tabuleiro, podendo também utilizar dados, peões e outros elementos, caracterizando um domínio de aplicações[12]. A atual versão foi implementada em Smalltalk, sob o framework MVC (do ambiente VisualWorks [PAR 94]), que dá suporte aos aspectos de geração de interfaces gráficas, bem como à interação do usuário com a aplicação através destas interfaces.

O desenvolvimento de FraG partiu da existência de um jogo elaborado em Smalltalk, o Jogo Corrida, ilustrado abaixo.

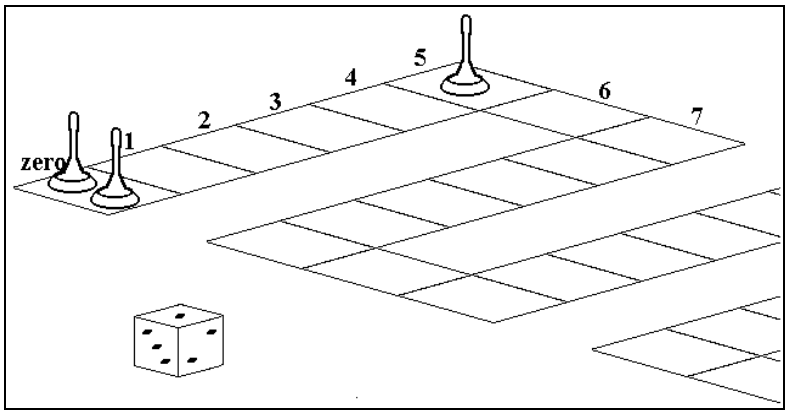

FIGURA 2.4 - Ilustração do Jogo Corrida

Para este jogo foi desenvolvida a estrutura de classes descrita no diagrama de classes[13] da figura 2.5.

---

12 Exemplos de jogos: Jogo da Velha, Dama, Ludo Real, Gamão, Banco Imobiliário, RPG etc.

13 Foram omitidas as relações de atributos e métodos, e informações relativas às associações (identificador, cardinalidade etc.). Classes sem superclasse explicitada, são subclasses de Model. Os nomes de algumas classes que aparecem neste diagrama e nos diagramas do framework FraG se referem a elementos do domínio de jogos de tabuleiro: Dice (dado), Board (tabuleiro), Player (jogador), Piece (peão), Position (casa).

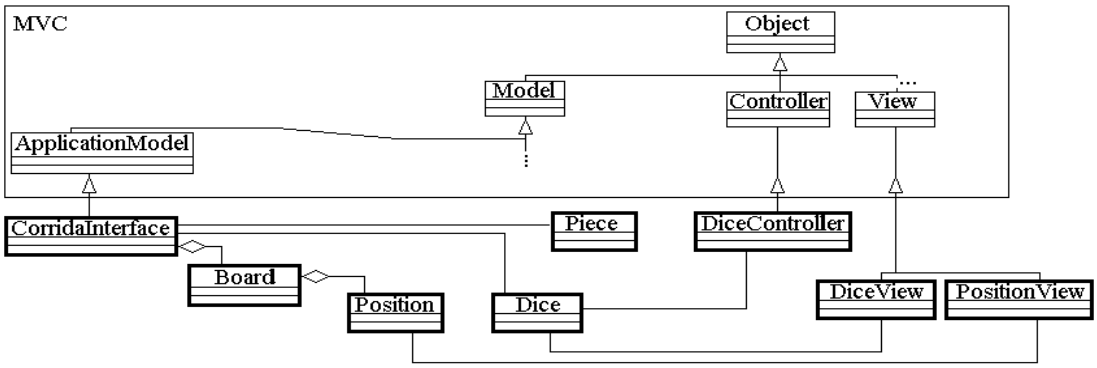

FIGURA 2.5 - Diagrama de classes do Jogo Corrida

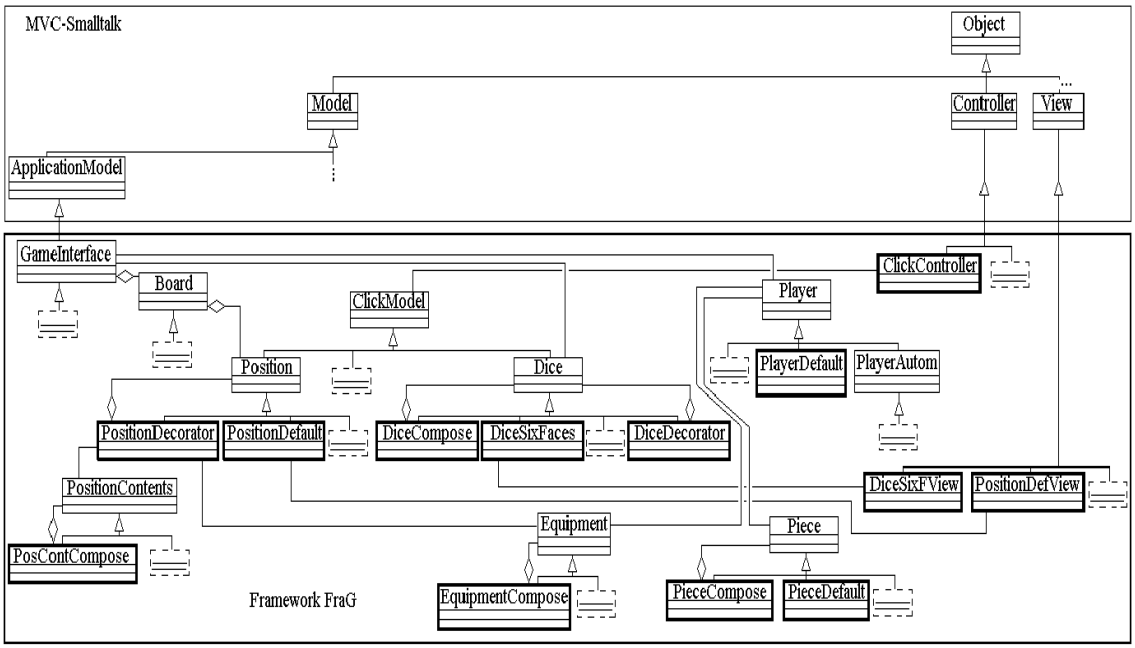

FIGURA 2.6 - Diagrama de classes do framework FraG

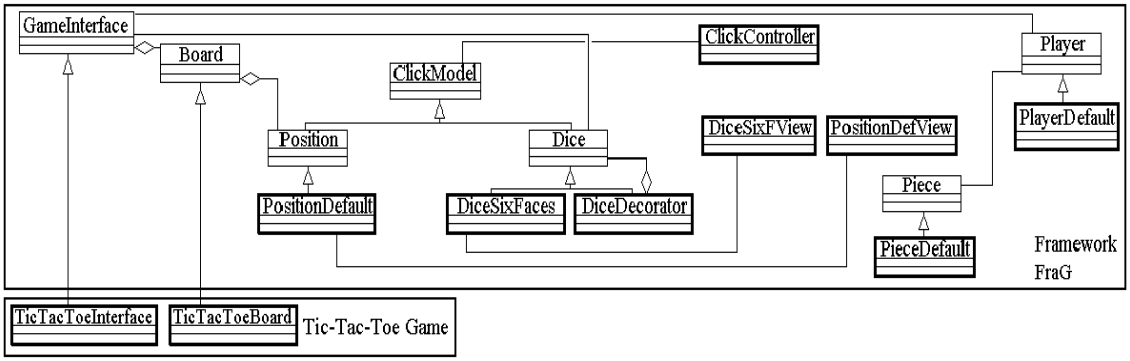

FIGURA 2.7 - Diagrama de classes do Jogo da Velha, desenvolvido sob o framework FraG

A partir da modificação desta estrutura de classes, levando em consideração as características de outros jogos de tabuleiro, foi obtida a estrutura de classes que generaliza o domínio tratado, isto é, o framework. A figura 2.6 apresenta um diagrama de classes que contém as principais classes do framework de jogos de tabuleiro. A figura 2.7 apresenta a estrutura de classes de um exemplo de aplicação desenvolvida sob o

framework - um Jogo da Velha (Tic-Tac-Toe Game), que demanda a criação de apenas duas classes. Nas figuras 2.5, 2.6 e 2.7 é utilizada a notação proposta em um trabalho anterior, que diferencia classes abstratas[14] (contorno com traço fino), classes concretas (contorno com traço gosso) e classes a serem criadas pelo usuário do framework na geração de aplicações (contorno tracejado)[15] [SIL 97]. Este exemplo é mencionado em outros capítulos do presente trabaho.

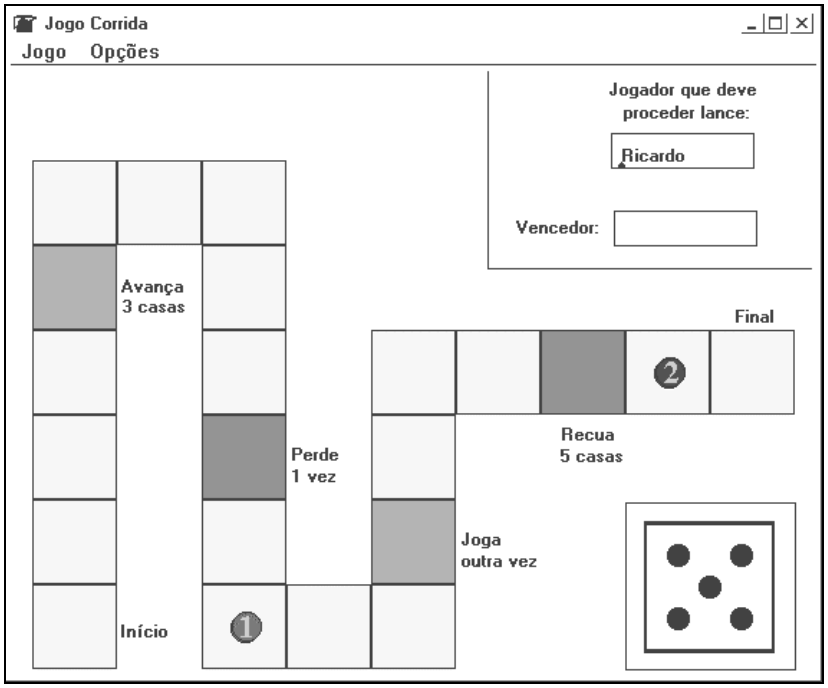

FIGURA 2.8 - Interface do Jogo Corrida desenvolvido sob o framework FraG

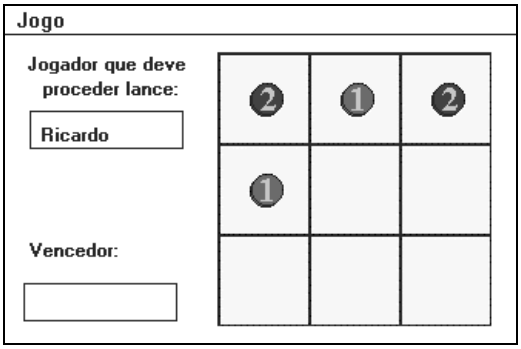

FIGURA 2.9 - Interface do Jogo da Velha (Tic-Tac-Toe) desenvolvido sob o framework FraG

---

14 Consideram-se classes abstradas as classes que possuem pelo menos um método abstrato (com a assinatura definida, mas sem implementação) [JOH 91].

15 Na figura 2.6 a noção de classe a acrescentar na utilização de um framework para a produção de aplicações é representada por uma figura de classe com contorno tracejado. Esta abordagem foi tratada em artigo apresentado no X Simpósio Brasileiro de Engenharia de Software [SIL 96]. No ambiente SEA esta representação foi substituída pela noção de redefinibilidade de classe, apresentada no capítulo 6. Como a especificação do framework FraG produzida no ambiente SEA fraciona o conjunto de classes do framework em mais de um diagrama de classes, e para possibilitar a apresentação do conjunto de classes do framework em um mesmo diagrama, optou-se por apresentar o diagrama na abordagem de representação anterior, que, porém, tem a mesma semântica da forma de representação adotada na atual versão do ambiente SEA (diagrama que faz parte de um artigo apresentado na 26th International Conference of the Argentine Computer Science and Operational Research Society [SIL 97]).

As figuras 2.8, 2.9 e 2.10 ilustram a interface de algumas aplicações (jogos) geradas a partir do framework FraG.

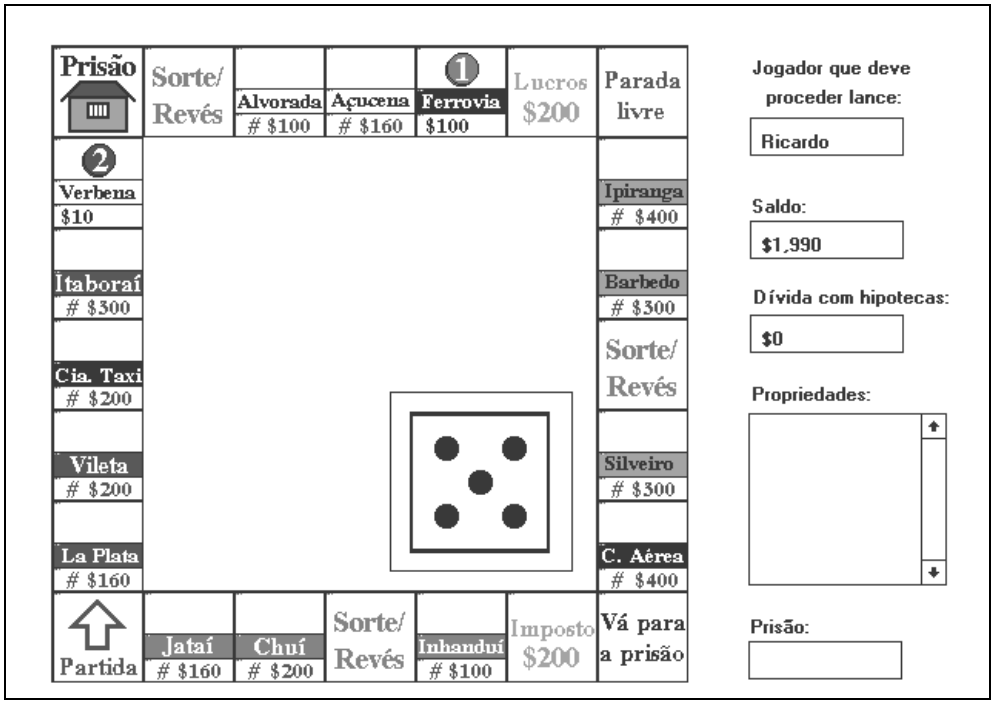

FIGURA 2.10 - Interface do Jogo Banco Imobiliário (Monopoly) desenvolvido sob o framework FraG

2.3 Análise de Domínio e a Abordagem de Frameworks Orientados a Objetos

A abordagem de frameworks orientados a objetos visa produzir uma descrição de um domínio, como uma estrutura de classes, a ser reusada no desenvolvimento de diferentes aplicações. Uma questão fundamental é a relevância desta abordagem em comparação a outras abordagens de análise de domínio.

A Análise de Domínio atua no desenvolvimento de descrições que generalizem uma família de aplicações com características comuns - um domínio de aplicações. A motivação para produzir uma especificação de um domínio de aplicação (modelo do domínio) é a meta de sua reutilização no desenvolvimento de aplicações para este domínio, com perspectiva de aumento de produtividade da atividade de desenvolvimento de software. Assim, o elemento motivador do surgimento da Análise de Domínio como área de pesquisa, é a busca da maximização da reutilização no desenvolvimento de software [ARA 91].

Arango define Análise de Domínio como o processo de identificação e organização de conhecimento a respeito de uma classe de problemas - um domínio de aplicações - para suportar a descrição e solução destes problemas [ARA 91].

A Análise de Domínio se baseia em duas premissas:

- Informações reutilizáveis são específicas de um domínio[16] [ARA 91]. Assim, a Análise de Domínio recomenda que o esforço para a produção de artefatos de software reutilizáveis se concentre em domínios específicos [HEN 95].
- Descrições de domínios apresentam mais estabilidade que os requisitos e especificações de aplicações específicas [COA 93] [ARA 91].

Para Neighbors, Análise de Domínio é uma tentativa de identificar os objetos, operações e relações que especialistas de um domínio percebem ser importantes para a descrição deste domínio [NEI 89].

Prieto-Diaz afirma que a Análise de Domínio é um passo fundamental na criação de artefatos de software reusáveis. Constata que elementos gerados pela Análise de Domínio são mais adequados à reutilização porque "capturam a funcionalidade essencial requerida por um domínio". A Análise de Domínio pode ser encarada como uma atividade que precede a análise de sistemas (definição da especificação de uma aplicação específica), e gera um produto (modelo do domínio) que é fonte de informação para a análise de sistemas [PRI 87]. Arango situa a Análise de Domínio em um meta-nível em relação ao desenvolvimento de uma aplicação - ao invés de desenvolver uma aplicação, é desenvolvida uma meta-aplicação, que será reusada no desenvolvimento de aplicações específicas [ARA 91].

O objetivo de um procedimento de Análise de Domínio é produzir subsídios para a reutilização de software. Estes subsídios correspondem a uma descrição do domínio contida em um modelo do domínio.

Uma descrição de domínio pode conter apenas uma descrição de características e funcionalidades do domínio, ou pode conter artefatos reutilizáveis. No primeiro caso, a descrição atua como uma fonte de informações do domínio a ser consultada em um desenvolvimento de aplicação. Fiorini propõe uma metodologia de Análise de Domínio com este objetivo [FIO 95]. No segundo caso, a descrição do domínio contém partes reutilizáveis no desenvolvimento de aplicações. Estas partes reutilizáveis podem corresponder a artefatos de análise e projeto, como proposto por Gomaa [GOM 94], ou incluir também código, como proposto por Neighbors, no paradigma Draco [NEI 89].

Arango propõe uma abordagem prática para a Análise de Domínio, baseada no paradigma de Engenharia do Conhecimento, que consiste nas seguintes etapas [ARA 91]:

- Identificar as fontes de conhecimento;
- Adquirir o conhecimento existente;
- Representar o conhecimento, através de um modelo do domínio.

Compatibilizando esta visão de Arango com a visão de Análise de Domínio de outros autores [PRI 87] [NEI 89] [SIM 91] [GOM 94], pode-se afirmar que a Análise de Domínio envolve duas etapas:

- Eliciação de conhecimento do domínio;
- Produção do modelo do domínio.

No presente capítulo a avaliação da abordagem de frameworks em relação a outras abordagens de análise de domínio se atém à segunda etapa, isto é, ao aspecto de produção de um modelo de domínio.

O processo de Análise de Domínio tem como resultado o modelo do domínio. As informações para a construção deste modelo são obtidas a partir de especialistas do

---

16 Nesta afirmação Arango se refere à reutilização que se reflete em aumento significativo de produtividade. Isto não descarta a possibilidade de reutilização de informações não-específicas a um domínio, mas classifica esta situação como de pouca contribuição ao aumento da produtividade.

domínio, de literatura técnica, de aplicações existentes, de pesquisas de mercado [ARA 91]. Abordagens metódicas de aquisição de conhecimento se baseiam em metodologias de Eliciação de Requisitos, como por exemplo a metodologia de Christel [CHR 92].

A construção do modelo do domínio das metodologias de Análise de Domínio pode se basear nas técnicas de modelagem de outras metodologias de desenvolvimento de software, com ou sem adaptações [ARA 91]. O processo de Análise de Domínio pode ser assistido por ferramentas de apoio às atividades, como editores e analisadores de especificações, *browsers*, ferramentas de Engenharia Reversa etc.

2.3.1 Metodologias de Análise de Domínio

A seguir são apresentadas três metodologias de Análise de Domínio [HEN 95] [GOM 94] [NEI 89] que estabelecem uma abordagem prática às tarefas da Análise de Domínio e que são voltadas à promoção de reutilização em algum nível (requisitos, especificações, código). É procedida uma análise de cada metodologia, considerando o nível de reutilização promovido e a flexibilidade do modelo do domínio para assimilar novas informações.

Metodologia de Henninger

A metodologia de Henninger propõe o uso de hipertexto para facilitar a consulta a projetos de aplicações previamente desenvolvidas, durante o desenvolvimento de uma nova aplicação [HEN 95]. Segundo a abordagem de Henninger, o conhecimento acumulado a respeito de um domínio de aplicações consiste em um conjunto de projetos de aplicações (incluindo arquivos de código fonte). Cada novo desenvolvimento de aplicação produz novas informações, que são agregadas ao repositório.

Um sistema de consulta baseado em hipertexto auxilia o desenvolvedor de uma aplicação a localizar os projetos de aplicações previamente desenvolvidas, que tem características semelhantes às da aplicação em desenvolvimento. A finalidade do sistema de consulta é ajudar a promover reutilização - no nível de especificações e de código - facilitando o acesso às informações existentes. A metodologia proposta atua nos aspectos de organização e localização de informações a respeito dos projetos existentes e em desenvolvimento.

O modelo do domínio para a metodologia de Henninger é composto de dois elementos:

- um conjunto de projetos de aplicações previamente desenvolvidas;
- um hipertexto que contém uma descrição sumária dos projetos, e mecanismos de indexação de informações.

A vantagem apontada pelo autor para esta abordagem, é a facilidade de montar o modelo do domínio, pois não ocorre um processo de modelagem do domínio propriamente dito, mas apenas a reunião de um conjunto de aplicações. A expansão do modelo do domínio é bastante simples, ou seja, basta incluir o projeto de cada nova aplicação no repositório (especificações de análise e projeto, código fonte), e incluir a descrição de cada aplicação no sistema de consulta.

O uso do hipertexto durante um desenvolvimento de aplicação leva à seleção de um ou mais projetos previamente desenvolvidos (ou partes específicas de projetos), que atacam problemas identificados no desenvolvimento. A decisão pelo reuso ou não (de parte das especificações ou mesmo de trechos de código) cabe ao desenvolvedor da aplicação - estando fora do escopo de atuação da metodologia de Henninger.

A metodologia de Henninger agrupa em um repositório um conjunto de aplicações previamente desenvolvidas. Para minimizar a dificuldade de manipulação do conjunto de projetos do repositório, define um hipertexto que conduz o processo de busca de projetos com características desejadas. O modelo de domínio produzido composto pelo conjunto de projetos e pelo hipertexto não produz uma generalização de um domínio, que é o produto normalmente esperado de um procedimento de Análise de Domínio [ARA 91] [PRI 87] [NEI 89]. A metodologia em si não promove reutilização; apenas facilita o acesso a informações potencialmente reutilizáveis.

Metodologia de Gomaa

A metodologia de Gomaa está voltada ao desenvolvimento de um ambiente para a produção de modelos de domínio e para a geração da especificação de aplicações [GOM 94]. O ambiente e as técnicas de modelagem usadas são independentes do domínio de aplicação tratado.

Gomaa adota a abordagem de construção de um modelo do domínio e o uso posterior deste modelo para a geração de aplicações. Modelo de domínio e aplicações são desenvolvidos segundo um "ciclo de vida evolucionário", ilustrado na figura 2.11.

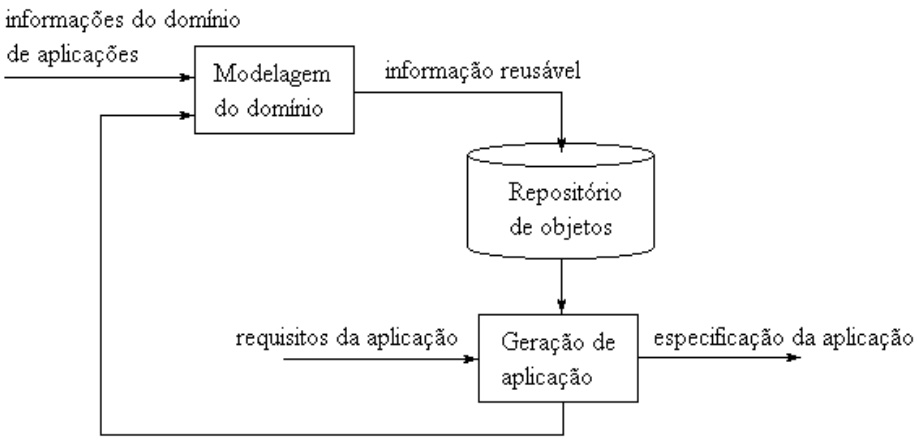

FIGURA 2.11 - Ciclo de vida evolucionário

O modelo do domínio consiste de um repositório de objetos (que são os artefatos reusáveis) composto a partir de um procedimento de Análise de Domínio. O procedimento de geração de uma especificação de aplicação consiste em uma composição de objetos do repositório. Durante o desenvolvimento de uma aplicação pode-se adquirir conhecimento do domínio que não esteja presente no modelo do domínio. Assim, o procedimento de desenvolvimento de aplicações pode produzir subsídios para uma evolução do modelo do domínio - efetuada a partir de um ciclo de Análise de Domínio. A expressão ciclo de vida evolucionário pretende, segundo Gomaa, eliminar a distinção entre as etapas de desenvolvimento e manutenção para o modelo do domínio. Assim, segundo esta abordagem, o modelo do domínio está permanentemente sujeito a alterações, na medida em que sejam adquiridos novos conhecimentos a respeito do domínio de aplicações.

O procedimento de Análise de Domínio da metodologia de Gomaa é baseado em metodologias de análise orientadas a objetos (OOA). Na descrição de sua metodologia, Gomaa não prevê procedimentos de captura de conhecimento.

O modelo do domínio é produzido no ambiente de desenvolvimento, em que fica inserido. Este ambiente contém um conjunto de ferramentas, responsáveis pela

verificação de consistência entre as diferentes visões do domínio e pelo mapeamento do modelo do domínio produzido, para uma representação interna (que é usada para a geração automática de especificações de aplicações).

O ambiente proposto por Gomaa suporta a geração automática de uma especificação de aplicação, a partir da definição dos requisitos da aplicação. Uma especificação de aplicação é expressa em termos das mesmas visões usadas na modelagem do domínio (agregação, herança, comunicação entre objetos, diagramas de transição de estados e relação das dependências característica/ objeto).

Gerar uma especificação de aplicação consiste em adaptar o modelo do domínio de acordo com as *características* desejadas para a aplicação. Especificar os requisitos da aplicação consiste em selecionar um conjunto de *características*, dentre as *características* presentes no modelo do domínio. A definição dos requisitos para o desenvolvimento de uma aplicação é conduzida por um sistema especialista, que define seqüências de tarefas, procede verificação de consistência entre características e objetos selecionados, e entre características interdependentes (quando uma característica selecionada depende de outras características, é verificado se foi feita a seleção de todo o conjunto).

Quando o processo de seleção de características é encerrado (o que significa que está concluída a definição de requisitos da aplicação), o ambiente constrói automaticamente a especificação da aplicação. Este procedimento consiste na seleção dos objetos que serão incluídos na especificação da aplicação.

A metodologia de Gomaa prevê a construção de um modelo do domínio geral para um conjunto de aplicações, e um procedimento de geração de especificações de aplicações a partir deste modelo do domínio. Em termos das informações contidas no modelo do domínio, e por conseguinte nas especificações de aplicações, pode-se classificar a reutilização promovida como ocorrendo no nível de especificações de análise. O formato de especificação adotado por Gomaa é semelhante ao formato das especificações de análise de metodologias OOAD, como Fusion [COL 94] e metodologia de Martin e Odell [MAR 95], no sentido de especificar um sistema como uma estrutura de objetos e um conjunto de funcionalidades. A especificação gerada pela metodologia de Gomaa não está fortemente ligada ao paradigma de orientação a objetos, na medida em que não define atributos e métodos (e com isso a semântica da relação de herança presente na especificação não induz necessariamente, à implementação a partir de linguagens orientadas a objetos). Assim, a especificação pode evoluir para um projeto orientado a objetos ou para um projeto definido a partir de outra abordagem.

A geração da especificação de uma aplicação é feita exclusivamente em termos da seleção de informações contidas no modelo do domínio. A metodologia não prevê a possibilidade da especificação da aplicação conter informações fornecidas pelo desenvolvedor e que não estejam presentes no modelo do domínio. Esta característica torna o processo de geração de aplicações inflexível em termos de inclusão de novos conhecimentos. A única forma prevista pela metodologia para a inclusão de conhecimento adquirido ao longo do desenvolvimento de uma aplicação, é a alteração do modelo do domínio e o procedimento de um novo desenvolvimento. Assim, pode-se considerar que a metodologia produz uma especificação de análise sujeita a refinamentos (se não se desejar proceder a alteração do modelo do domínio). O refinamento da especificação de análise e a produção da especificação de projeto estão fora do escopo de atuação de metodologia de Gomaa.

Paradigma Draco

Segundo a metodologia de Neighbors, conhecida como paradigma Draco de desenvolvimento de software, um domínio corresponde a um conjunto de objetos[17] e operações, e regras que estabelecem as combinações possíveis entre estes elementos [NEI 89]. Para um domínio é definida uma linguagem específica, cujas definições sintática e semântica estão em conformidade com estas regras de combinação de elementos (linguagem Draco).

Desenvolver um modelo de domínio no paradigma Draco corresponde a desenvolver uma linguagem para este domínio, e capacitar o ambiente de desenvolvimento (máquina Draco) a produzir um programa fonte a partir de uma especificação na linguagem do domínio.

Produzir uma aplicação segundo o paradigma Draco consiste em escrever uma especificação utilizando a linguagem do domínio da aplicação. Esta especificação é submetida à análise léxica e sintática, pode ser convertida em outras linguagens de domínio, e ser traduzida para um programa fonte, em uma linguagem de programação.

Modelo do domínio - A máquina Draco, que é o ambiente de desenvolvimento de software da metodologia de Neighbors, é o repositório de um conjunto de domínios. A metodologia estabelece uma classificação dos domínios em três níveis hierárquicos, em que considera o nível de abstração da linguagem de domínio. Uma linguagem de domínio tem sua semântica definida em termos de linguagens Draco previamente definidas no ambiente. Os três tipos de domínio estão relacionados a seguir.

- **Domínio executável**: corresponde a linguagens de programação. É o domínio de mais baixo nível de abstração.
- **Domínio de modelagem**: é o encapsulamento do conhecimento necessário para produzir uma funcionalidade de uma aplicação, como por exemplo interface interativa, gerenciamento de banco de dados etc. Um domínio de modelagem se encontra no nível de abstração intermediário entre os outros dois tipos de domínios. A linguagem de um domínio de modelagem pode ser definida em termos de outras linguagens de domínio de modelagem ou de linguagens de domínio executável (ou de uma combinação de linguagens dos dois tipos).
- **Domínio de aplicação**: corresponde ao acúmulo de conhecimento capaz de produzir aplicações completas. É o tipo de domínio de mais alto nível de abstração. Para Neighbors um domínio de aplicação é "uma espécie de cola que une domínios de modelagem". Esta união implica em restrições sobre os domínios de modelagem usados. O usuário de um domínio de aplicação só terá acesso à funcionalidade de um domínio de modelagem que interesse diretamente ao domínio de aplicação (restrição estabelecida na definição da linguagem do domínio de aplicação). Por exemplo, se um domínio de modelagem que porta a funcionalidade do gerenciamento de banco de dados for utilizado para a definição de uma linguagem de um domínio de aplicação, pode ser que nem toda a funcionalidade do domínio de modelagem seja disponibilizada ao usuário da linguagem do domínio de aplicação.

Processo de análise de domínio - O processo de análise de domínio na abordagem Draco pode ser voltado à inclusão de um dos três tipos de domínio ao ambiente. No caso de um domínio executável poderia corresponder, por exemplo, à capacitação do

---

17 A expressão *objeto* usada por Neighbors não está relacionada ao paradigma de orientação a objetos. Se refere à descrição dos elementos físicos ou conceituais que fazem parte do domínio.

ambiente para utilizar determinada linguagem de programação; no caso de um domínio de modelagem poderia corresponder a incluir por exemplo, novas facilidades para a geração de interfaces. Apenas a inclusão de um domínio de aplicação é que faz com que o ambiente esteja capacitado a produzir aplicações.

Uso do modelo do domínio - A produção de uma aplicação no ambiente Draco inicia com a produção de uma especificação para a aplicação, utilizando a linguagem de domínio de aplicação adequada. Uma vez produzida a especificação, ela é submetida às análises léxica e sintática. Após isto, a especificação (validada e na representação interna do ambiente) pode ser otimizada, analisada ou pode ser usada para a produção de uma implementação.

A produção da implementação consiste na translação da especificação para outra linguagem de domínio. Este processo pode se dar de forma interativa. Uma especificação produzida em uma linguagem de domínio de aplicação pode ser convertida para outra linguagem de domínio de aplicação ou para uma linguagem de domínio de modelagem[18]. A translação de linguagens ocorre de forma descendente[19], em termos de nível de abstração da linguagem de domínio, até a obtenção do código executável.

Análise - A abordagem de Neighbors promove reutilização de abstrações de alto nível contidas na definição das linguagens de domínio de aplicação, e, a partir da tradução semi-automática destas abstrações, promove reutilização de código. Assim, pode-se afirmar que ocorre reutilização nos níveis de projeto e código, o que é um mérito da abordagem.

Uma característica da metodologia é a sua ligação ao conceito de *rede de domínios*. Isto gera conseqüências positivas e negativas. Um aspecto positivo é que a modelagem de um domínio de aplicação pode ser bastante simplificada pela reutilização de outras descrições de domínio contidas no ambiente. A metodologia prevê a necessidade da existência de domínios de modelagem e de pelo menos um domínio executável, para a definição de um domínio de aplicação. Um aspecto negativo decorrente disto é que existe a necessidade de um certo esforço para a composição de um ambiente mínimo, capaz de gerar aplicações. Na medida em que a rede de domínios cresce com a inclusão de novos domínios, o ambiente passa a dispor de mais artefatos de software que podem ser reutilizados para a definição de novos domínios de aplicação. Por outro lado, a expansão da rede de domínios demanda mais esforço, em termos de conhecer os domínios existentes no ambiente (o que é imprescindível para que de fato ocorra reutilização).

Estudar um domínio corresponde a aprender a usar a sua linguagem. Conhecer um conjunto de domínios de modelagem para a geração de um domínio de aplicação implica em assimilar o respectivo conjunto de linguagens, o que pode demandar muito esforço, além de não caracterizar uma interação amigável com o usuário. Um aspecto negativo decorrente disto é que o uso do ambiente depende de experiência humana. Assim, pode ser necessário grande esforço para que usuários se capacitem ao uso do ambiente (tanto para a modelagem de domínios, quanto para a geração de aplicações).

---

18 Também é possível que diferentes partes de uma especificação sejam convertidas para linguagens diferentes, inclusive de níveis de abstração diferentes.

19 O que não descarta a possibilidade de uma etapa de translação para uma linguagem de domínio de mesmo nível de abstração.

2.3.2 Avaliação Comparativa da Abordagem de Frameworks Orientados a Objetos em relação a Metodologias de Análise de Domínio

Observam-se características comuns nas metodologias de Análise de Domínio apresentadas e na abordagem de frameworks. Assim, como ocorre no desenvolvimento de frameworks, as metodologias de Análise de Domínio descritas buscam a produção de uma abstração de um domínio de aplicação que permita reuso de informações[20] no desenvolvimento de aplicações específicas.

Um aspecto relevante a considerar para a comparação de enfoques de Análise de Domínio, é a abrangência em termos das etapas do ciclo de vida do desenvolvimento de software - tanto em termos da produção do modelo do domínio, como em termos da reutilização possibilitada pelo uso deste modelo. Quanto maior a abrangência de uma abordagem sobre as etapas do ciclo de vida, maior será seu potencial de contribuição à otimização do processo de produção de software, a partir da promoção de reuso. A abordagem de frameworks produz uma descrição de domínio na forma de uma estrutura de classes. Esta estrutura contém a arquitetura e a definição da estrutura de controle das aplicações a serem geradas a partir dela, o que promove reutilização de projeto. A implementação das classes do framework é reusada na geração de aplicações e com isto, verifica-se também reutilização de código. Assim, a reutilização promovida pela abordagem de frameworks abrange todo o ciclo de vida de uma aplicação, desde a etapa de análise até a implementação.

As três metodologias de Análise de Domínio apresentadas apresentam diferentes níveis de abrangência. A metodologia de Henninger produz um hipertexto que atua como indexador de características de aplicações existentes. Não chega a produzir um modelo de domínio que generalize aspectos do domínio. O uso do modelo proposto não promove diretamente reutilização - apenas facilita o acesso do desenvolvedor a informações existentes. A metodologia de Gomaa atua em um nível de abrangência mais amplo em termos de etapas do ciclo de vida do desenvolvimento de software. Produz um modelo que contém uma generalização de um domínio no nível de análise, e promove reutilização de especificações também no nível de análise (a especificação de uma aplicação é composta de partes do modelo do domínio). Das três metodologias apresentadas, a metodologia de Neighbors (paradigma Draco), é a que apresenta maior nível de abrangência nas etapas do ciclo de vida. O modelo do domínio contém especificações (definição das linguagens de domínio) e código - a reutilização também ocorre no nível de especificações (uso das abstrações definidas nas linguagens de domínio para especificar uma aplicação) e de código (tradução semi-automática).

Comparando as metodologias de Análise de Domínio apresentadas com a abordagem de frameworks, observa-se que os frameworks se situam no patamar mais elevado, em termos de promoção de reutilização, isto é, promovem reutilização de projeto e código, o que caracteriza uma vantagem desta abordagem. Dentre as metodologias de Análise de Domínio apresentadas, apenas a metodologia de Neighbors promove reutilização no nível de projeto e código. Assim, a comparação de abordagens de Análise de Domínio com a abordagem de frameworks orientados a objetos demonstra a viabilidade desta abordagem. Isto se baseia principalmente no potencial de reutilização de projeto e código, característico dos frameworks.

---

[20] Este reuso de informações pode se referir a reuso de código, especificações (análise ou projeto), ou simplesmente uma descrição que sirva como fonte de informações a respeito do domínio, sem disponibilizar código ou especificação.

3 Desenvolvimento de Frameworks Orientados a Objetos

Um framework é uma abstração de um domínio de aplicações, adequada a ser especializada em aplicações deste domínio. A principal característica buscada ao desenvolver um framework, é a generalidade em relação a conceitos e funcionalidades do domínio tratado. Além disso, é fundamental que a estrutura produzida seja flexível, isto é, apresente as características alterabilidade e extensibilidade[21].

A alterabilidade reflete a capacidade do framework de alterar suas funcionalidades, em função da necessidade de uma aplicação específica, o que é operacionalizado através de uma identificação adequada das partes da estrutura que diferem em aplicações distintas do mesmo domínio. Isto equivale a uma diferenciação entre os conceitos gerais do domínio, e os conceitos específicos das aplicações. Uma vez obtida esta diferenciação, é necessária a seleção de soluções de projeto adequadas para implementação dos conceitos gerais do domínio, de modo que a estrutura de classes resultante apresente a alterabilidade requerida.

A questão da extensibilidade se refere à manutenibilidade do framework. Um framework possui uma estrutura de classes mais complexa que a estrutura de uma aplicação do seu domínio. Esta estrutura é construída em ciclos iterativos. A evolução da estrutura do framework se estende por toda a sua vida útil, pois à medida em que é utilizado, novos recursos podem ser agregados. Assim, na definição de abstrações, deve-se ter a preocupação em prever futuras utilizações para o framework, inclusive a possibilidade de estender os limites do domínio tratado.

No processo de desenvolvimento de um framework, deve-se produzir uma estrutura de classes com a capacidade de adaptar-se a um conjunto de aplicações diferentes. Para construir um framework, é fundamental que se disponha de modelagens de um conjunto significativo de aplicações do domínio. Este conjunto pode se referir a aplicações previamente desenvolvidas, como ocorre na metodologia "Projeto Dirigido por Exemplo" [JOH 93], ou a aplicações que se deseja produzir a partir do framework. A ótica de diferentes aplicações é o que dá ao desenvolvedor a capacidade de diferenciar conceitos gerais de conceitos específicos.

Em termos práticos, dotar um framework de generalidade, alterabilidade e extensibilidade requer uma cuidadosa identificação das partes que devem ser mantidas flexíveis e a seleção de soluções de projeto de modo a produzir uma arquitetura bem estruturada. Isto passa pela observação de princípios de projeto orientado a objetos, como o uso de herança para reutilização de interfaces (ao invés do uso de herança para reutilização de código); reutilização de código através de composição de objetos; preocupação em promover polimorfismo, na definição das classes e métodos, de modo a possibilitar acoplamento dinâmico etc. [JOH 88]. No contexto de frameworks, o uso adequado de herança implica na concentração das generalidades do domínio em classes abstratas, no topo da hierarquia de classes. Isto promove uso adequado de herança, pois a principal finalidade destas classes abstratas é definir as interfaces a serem herdadas pelas classes concretas das aplicações.

---

21 Alterabilidade se refere à capacidade de alterar a funcionalidade presente, sem conseqüências imprevistas sobre o conjunto da estrutura; extensibilidade se refere à capacidade de ampliar a funcionalidade presente, sem conseqüências imprevistas sobre o conjunto da estrutura [FAY 96] [MEY 88]. A expressão flexibilidade é utilizada no presente trabalho para expressar a capacidade de um framework ser alterável e extensível.

Pode-se afirmar que o desenvolvimento de um framework é mais complexo que o desenvolvimento de aplicações específicas do mesmo domínio, devido:

- À necessidade de considerar os requisitos de um conjunto significativo de aplicações, de modo a dotar a estrutura de classes do framework de generalidade, em relação ao domínio tratado;
- À necessidade de ciclos de evolução voltados a dotar a estrutura de classes do framework de alterabilidade e extensibilidade.

A pesquisa sobre identificação de *padrões* fornece estruturas de projeto semiprontas que podem ser reutilizadas, contribuindo no desenvolvimento de frameworks, no sentido de produzir uma organização de classes bem estruturada. *Padrões* promovem o uso adequado de herança e o desenvolvimento de projetos em que o acoplamento entre classes é minimizado [PRE 94] [JOH 93].

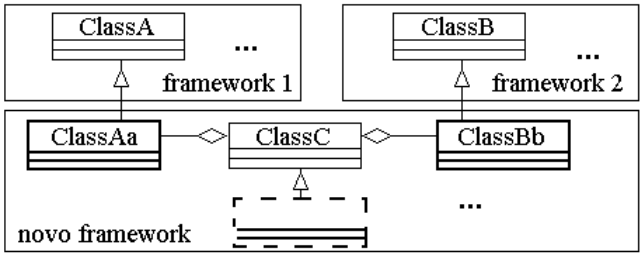

FIGURA 3.1 - Combinação de frameworks

O processo de desenvolvimento de frameworks pode envolver a tarefa de combinar frameworks para uso conjunto. O framework que necessite de uma interface gráfica interativa por exemplo, pode ser desenvolvido sob o o framework MVC. Neste caso o framework desenvolvido precisa respeitar as restrições de MVC. Uma outra situação seria a utilização de mais de um framework - por exemplo, MVC e um framework para sonorização, suportando um terceiro framework. Neste caso, além de respeito às restrições dos frameworks reutilizados, o framework desenvolvido teria a responsabilidade de compatibilizar os protocolos de controle dos outros dois frameworks reusados. A figura 3.1 ilustra o uso de composição de objetos para a combinação de objetos responsáveis pela lógica de controle.

A seguir, é tratado o ciclo de vida de um framework e apresentadas e avaliadas metodologias de desenvolvimento de frameworks. Discute-se o processo de desenvolvimento, o suporte provido pela abordagem de padrões, o registro das informações de projeto e o suporte ferramental disponível para auxílio ao desenvolvimento de frameworks.

3.1 Ciclo de Vida de Frameworks

O ciclo de vida de um framework difere do ciclo de vida de uma aplicação convencional porque um framework nunca é um artefato de software isolado mas sua existência está sempre relacionada à existência de outros artefatos - originadores do framework, originados a partir dele ou que exercem alguma influência na definição da estrutura de classes do framework. A figura 3.2 ilustra as várias fontes de informação que influem na definição da estrutura de um framework: artefatos de software existentes, artefatos de software produzidos a partir do framework e o conhecimento do desenvolvedor do framework (ou da equipe de desenvolvimento). As setas representam

o fluxo de informações que levam à produção da estrutura de classes do framework, bem como de aplicações sob o framework.

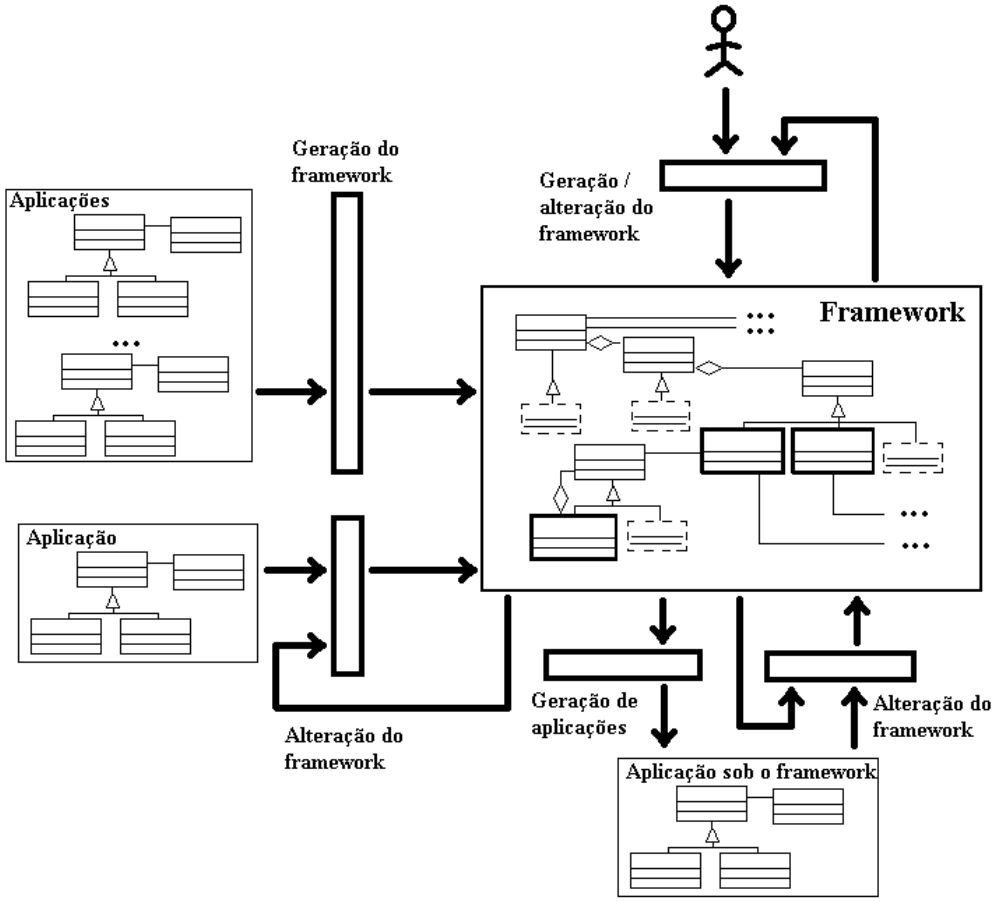

FIGURA 3.2 - Ciclo de vida de frameworks

A atuação do desenvolvedor - nenhuma abordagem de desenvolvimento de frameworks existente dispensa a figura do desenvolvedor. Ele é o reponsável por decidir que classes comporão a estrutura do framework, suas responsabilidades e a flexibilidade provida aos usuários do framework. O desenvolvedor atua não apenas na construção do framework, mas também na sua manutenção. A figura 3.3 destaca a atuação do desenvolvedor no ciclo de vida de um framework.

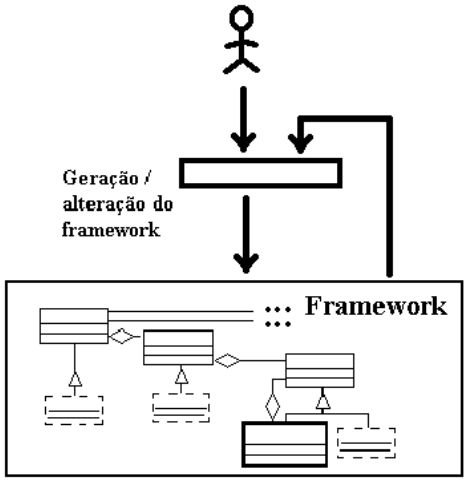

FIGURA 3.3 - Destaque da atuação do desenvolvedor no ciclo de vida de um framework

Aplicações do domínio tratado - um framework constitui um modelo de um domínio de aplicações. Assim, pode ser desenvolvido a partir de um conjunto de aplicações do domínio, que atuam como fontes de informação deste domínio. Esta influência de aplicações do domínio pode ocorrer no processo de desenvolvimento do framework, em que um framework é construído como uma generalização de diferentes estruturas, ou na fase de manutenção. Neste caso a alteração seria motivada pela obtenção de conhecimento do domínio tratado, não considerado (ou indisponível) durante o desenvolvimento do framework.

A figura 3.4 retrata um procedimento em que um conjunto de aplicações leva à produção de um framework e um procedimento em que uma aplicação leva à alteração da estrutura do framework. Da forma como está apresentado (sem a figura do desenvolvedor) subentende-se que sejam procedimentos automatizados. De fato, esta é a situação desejável: dispor de mecanismos de análise capazes de generalizar diferentes aplicações em uma estrutura de framework, bem como identificar partes de estrutura de aplicações que podem ser incluídas em um framework existente. Atualmente não existem mecanismos capazes de realizar tais tarefas e esta abordagem de captura automatizada foge do escopo do presente trabalho. Assim, esta influência de aplicações existentes prescinde da atuação do desenvolvedor, que é responsável por avaliar os aspectos gerais e os aspectos específicos de cada aplicação e de como estes aspectos serão ou não incluídos na estrutura de um framework - tanto na etapa de desenvolvimento, como na de manutenção.

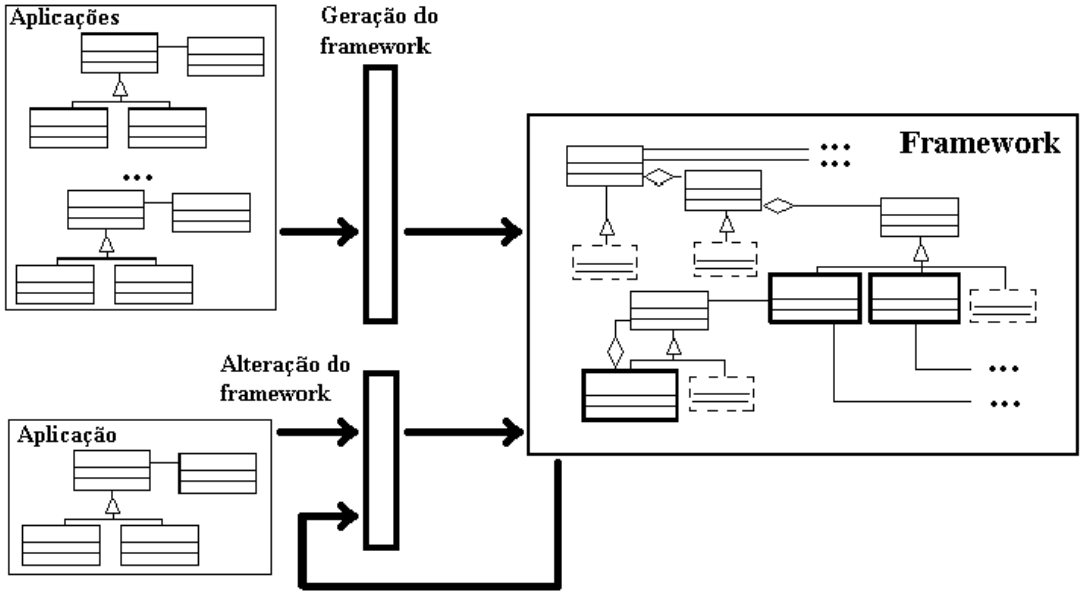

FIGURA 3.4 - Destaque da influência de aplicações existentes na geração e na alteração de um framework

Aplicações geradas sob o framework - a finalidade básica de um framework é ser reutilizado na produção de diferentes aplicações, minimizando o tempo e esforço requeridos para isto. A construção de um framework é sempre precedida por um procedimento de análise de domínio em que são buscadas informações do domínio tratado. Como uma abstração de uma realidade tratada, é inevitável que o framework seja incapaz de conter todas as informações do domínio. De fato, um framework consegue ser uma descrição aproximada do domínio, construída a partir das informações até então disponíveis.

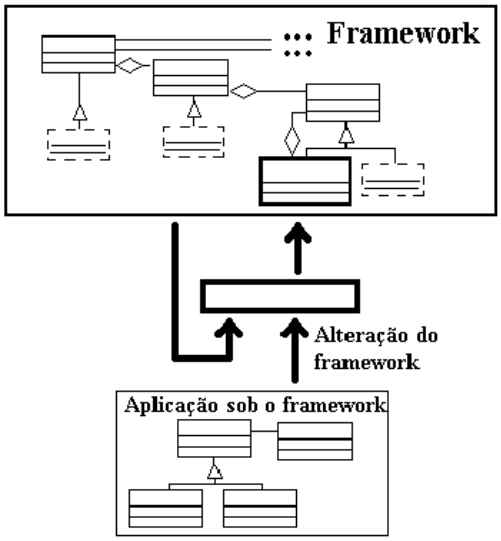

FIGURA 3.5 - Destaque da influência de aplicações geradas a partir de um framework na alteração deste framework

Idealmente a construção de aplicações sob frameworks consiste em completar ou alterar procedimentos e estruturas de dados presentes no framework. Sob esta ótica, uma aplicação gerada sob um framework não deveria incluir classes que não fossem subclasses de classes do framework. Porém, como um framework nunca é uma descrição completa de um domínio, é possível que a construção de aplicações sob um framework leve à obtenção de novos conhecimentos do domínio tratado, indisponíveis durante a construção do framework. Estas novas informações podem levar à necessidade de alterar o framework. A figura 3.5 ilustra a influência de aplicações geradas sob um framework na alteração de sua estrutura original. Conforme já mencionado, mecanismos de análise que procedam a análise de aplicações e a alteração do framework de forma automatizada são desejáveis, porém não estão disponíveis ainda e esta abordagem foge do escopo do presente trabalho. Assim, como no caso de aplicações não relacionadas diretamente ao framework, supõe-se que a análise e alteração do framework (e eventualmente, da aplicação que ocasionou esta alteração) prescinde a atuação do desenvolvedor.

3.2 Metodologias de Desenvolvimento de Frameworks

Atualmente existem algumas propostas de metodologias de desenvolvimento de frameworks. Consistem em propostas de procedimentos que abrangem a captura de informações de um domínio, a construção e o teste da estrutura de frameworks.

A seguir são descritas três metodologias voltadas ao desenvolvimento de frameworks: Projeto Dirigido por Exemplo (*Example-Driven Design*) [JOH 93], Projeto Dirigido por *Hot Spot*[22] (*Hot Spot Driven Design*) [PRE 94] e a metodologia de projeto da empresa Taligent [TAL 94]. Estas metodologias se caracterizam por estabelecer o processo de desenvolvimento de frameworks em linhas gerais, sem se ater à definição de técnicas de modelagem ou detalhar o processo.

---

[22] A expressão *hot-spot* foi mantida em Inglês e significa "parte flexível", significado este não obtido a partir de sua tradução literal.

3.2.1 Projeto Dirigido por Exemplo

Johnson estabelece que o desenvolvimento de um framework para um domínio de aplicação é decorrente de um processo de aprendizado a respeito deste domínio, que se processa concretamente a partir do desenvolvimento de aplicações ou do estudo de aplicações desenvolvidas. Porque as pessoas pensam de forma concreta, e não de forma abstrata, a abstração do domínio, que é o próprio framework, é obtida a partir da generalização de casos concretos - as aplicações. Abstrações são obtidas de uma forma *bottom-up*, a partir do exame de exemplos concretos: aspectos semelhantes de diferentes aplicações podem dar origem a classes abstratas que agrupam as semelhanças, cabendo às classes concretas do nível hierárquico inferior a especialização para satisfazer cada caso.

O processo de generalização ocorre a partir da busca de elementos que recebem nomes diferentes, mas são a mesma coisa; recorrendo à parametrização para eliminar diferenças; particionando elementos para tentar obter elementos similares; agrupando elementos similares em classes.

O processo de desenvolvimento segundo o Projeto Dirigido por Exemplo, atravessa as etapas de análise, projeto e teste. A forma tida como ideal para desenvolver um framework (por avaliar um número considerado adequado de exemplos) é:

1 - Análise do domínio:

- assimilar as abstrações já conhecidas;
- coletar exemplos de programas que poderiam ser desenvolvidos a partir do framework (no mínimo, quatro);
- avaliar a adequação de cada exemplo.

2 - Projetar uma hierarquia de classes que possa ser especializada para abranger os exemplos (um framework) - nesta etapa o autor recomenda a utilização de *design patterns*;

3 - Testar o framework usando-o para desenvolver os exemplos (implementar, testar e avaliar cada exemplo usado na primeira etapa, utilizando para isto, o framework desenvolvido).

Observe-se que o passo 2 corresponde à modelagem de uma estrutura de classes, semelhante ao que é feito pelas metodologias OOAD, porém partindo de subsídios de análise do domínio tratado, e visando não uma aplicação particular, mas um framework.

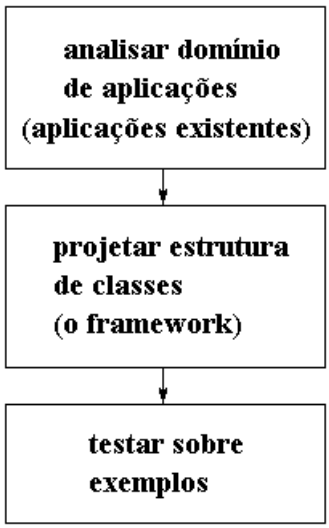

FIGURA 3.6 - As etapas do Projeto Dirigido por Exemplo para o desenvolvimento de um framework

O autor afirma que este caminho ideal nunca é seguido, por limitações de ordem econômica ou de tempo:

- a análise adequada de exemplos particulares demanda tempo e esforço, o que implica em custo;
- na medida em que os softwares - potenciais exemplos do domínio - funcionam, não há incentivo financeiro para convertê-los em um novo software (que seriam gerados a partir do framework);
- o esforço de análise acaba sendo mais voltado ao software tido como mais importante.

Em função da dificuldade prática de analisar um mínimo de quatro aplicações já desenvolvidas para então começar a desenvolver o framework, é estabelecido um procedimento tido como adequado (e economicamente viável) para o desenvolvimento de um framework: a partir de duas aplicações similares que se necessite desenvolver, proceder paralelamente o desenvolvimento do framework e das duas aplicações, procurando maximizar a troca de informações entre os três desenvolvimentos. Segundo o autor, para que este procedimento possa ser realizado, as equipes de desenvolvimento devem conter pessoas com experiência de aplicações do domínio tratado - como uma forma de capturar informações de outras aplicações.

Procedendo uma comparação preliminar entre esta primeira metodologia de desenvolvimento de frameworks e as metodologias OOAD, observa-se que o Projeto Dirigido por Exemplo carece de um conjunto de técnicas de modelagem e de um processo de desenvolvimento detalhado. Isto sugere que a metodologia poderia ser complementada por subsídios de metodologias OOAD.

3.2.2 Projeto Dirigido por *Hot Spot*

Uma aplicação orientada a objetos é completamente definida. Um framework, ao contrário possui partes propositalmente indefinidas - o que lhe dá a capacidade de ser flexível e se moldar a diferentes aplicações. Os *hot spots* são as partes do framework mantidas flexíveis. A essência da metodologia é identificar os *hot spots* na estrutura de classes de um domínio, e, a partir disto, construir o framework. Pree propõe a seqüência de procedimentos ilustrada na figura 3.7.

Na primeira etapa, semelhante ao que é feito no desenvolvimento de uma aplicação orientada a objetos, é procedida a identificação de classes. O desenvolvedor do framework, a partir de informações de especialistas do domínio, define uma estrutura de classes.

Na segunda etapa, também com o auxílio de especialistas do domínio, são identificados os *hot spots*. É preciso identificar que aspectos diferem de aplicação para aplicação, e definir o grau de flexibilidade que deve ser mantido em cada caso. Quando os aspectos que diferem entre aplicações são os dados, a conseqüência é que classes abstratas agruparão os dados (atributos) comuns e classes concretas, específicas para aplicações, agruparão os dados específicos. Quando os aspectos que diferem entre aplicações são as funções, a conseqüência é a presença nas classes além dos métodos base (métodos completamente definidos), de métodos abstratos (em classes abstratas) que definem apenas o protocolo do método, métodos *template*, que chamam outros métodos (*hook*) para completar seu processamento e métodos *hook,* que flexibilizam o comportamento dos métodos *template*. *Hot spots* são documentados com cartões *hot spot*. A figura 3.8 apresenta o formato proposto para estes cartões.

A terceira etapa, o projeto do framework (ou a reelaboração do projeto), ocorre após a identificação dos *hot spots*. Consiste em modificar a estrutura de classes

inicialmente definida, de modo a comportar a flexibilidade requerida. Nesta etapa se definirá o que o usuário do framework deve fazer para gerar uma aplicação - que subclasses deve definir, a que classes deve fornecer parâmetros para a criação de objetos, quais as combinações de objetos possíveis.

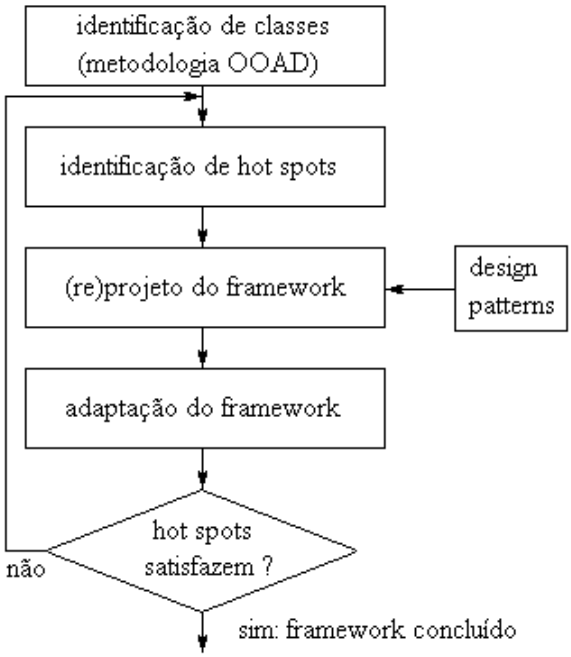

FIGURA 3.7 - As etapas do Projeto Dirigido por Hot Spot para o desenvolvimento de um framework

Nome do *hot spot*	☐ flexibilidade em tempo de execução
descrição geral da semântica	
comportamento do *hot spot* em pelo menos duas situações específicas	

FIGURA 3.8 - Formato de um cartão hot spot

A quarta etapa consiste num refinamento da estrutura do framework a partir de novas intervenções de especialistas do domínio. Se após isto o framework for avaliado como satisfatório, está concluída uma versão do framework (a questão é: os *hot spots* definidos dão ao framework o grau de flexibilidade necessário para gerar as aplicações requeridas ?), caso contrário, retorna-se à etapa de identificação de *hot spots*.

Segundo Pree, na etapa de projeto o desenvolvimento do framework é centrado em *design patterns*, cuja aplicação é fundamental para garantir flexibilidade ao framework.

3.2.3 Metodologia de Projeto da Empresa Taligent

A metodologia proposta pela empresa Taligent (empresa já extinta) difere das anteriores pelo conjunto de princípios que norteia o desenvolvimento de frameworks. Primeiramente, a visão de desenvolver um framework que cubra as características e necessidades de um domínio é substituída pela visão de produzir um conjunto de frameworks estruturalmente menores e mais simples, que usados conjuntamente, darão

origem às aplicações. A justificativa para isto é que "pequenos frameworks são mais flexíveis e podem ser reutilizados mais freqüentemente". Assim, a ênfase passa a ser o desenvolvimento de frameworks pequenos e direcionados a aspectos específicos do domínio.

Um outro aspecto é que uma das linhas mestras do processo de desenvolvimento é tornar o uso do framework o mais simples possível, através da minimização da quantidade de código que o usuário deve produzir, por meio de:

- disponibilidade de implementações (classes) concretas que possam ser usadas diretamente;
- minimização do número de classes que devem ser criadas;
- minimização do número de métodos que devem ser sobrepostos.

A metodologia propõe a seqüência de quatro passos, apresentados a seguir.

Identificar e caracterizar o domínio do problema:

- analisar o domínio e identificar os frameworks necessários (para o desenvolvimento de aplicações), o que pode incluir frameworks desenvolvidos e a desenvolver;
- examinar soluções existentes;
- identificar as abstrações principais;
- identificar o limite de responsabilidades do framework em desenvolvimento (considerando que se esteja no processo de desenvolvimento de um framework específico);
- validar estas informações com especialistas do domínio.

Definir a arquitetura e o projeto:

- refinar a estrutura de classes obtida no passo anterior, centrando atenção em como os usuários interagem com o framework:
  - ⇨ que classes o usuário instancia ?
  - ⇨ que classes o usuário produz (e que métodos sobrepõe) ?
  - ⇨ que métodos o usuário chama ?
- aperfeiçoar o projeto com o uso de *design patterns*;
- validar estas informações com especialistas do domínio.

Implementar o framework:

- implementar as classes principais;
- testar o framework (gerando aplicações);
- solicitar a terceiros o procedimento de teste
- iterar para refinar o projeto.

Desdobrar o framework:

- providenciar documentação na forma de diagramas, receitas (como usar o framework para desenvolver determinada aplicação) e exemplos de aplicações simples (incluindo o código da implementação, que complementa o framework);
- manter e atualizar o framework, seguindo as seguintes regras:
  - ⇨ corrigir erros imediatamente;
  - ⇨ adicionar novas características ocasionalmente;
  - ⇨ mudar interfaces tão infreqüentemente quanto possível ("é melhor adicionar novas classes que alterar a hierarquia de classes existente, bem como adicionar novos métodos que alterar ou remover métodos existentes").

3.2.4 Análise Comparativa das Metodologias de Desenvolvimento de Frameworks

Características comuns

As três metodologias apresentadas possuem uma série de características comuns. Uma primeira a destacar é a busca de informações do domínio de aplicação em aplicações já elaboradas - o Projeto Dirigido por Exemplo preconiza a busca das abstrações do domínio fundamentalmente a partir da utilização desta fonte. Cabe ressaltar que basear-se em aplicações existentes para construir um modelo do sistema durante o processo de análise, não é uma exclusividade das metodologias de desenvolvimento de frameworks. É um procedimento também adotado por metodologias OOAD, que são voltadas ao desenvolvimento de aplicações (pela metodologia de Coad e Yourdon, por exemplo [COA 92]).

Nenhuma das três metodologias de desenvolvimento de frameworks adota técnicas de modelagem para a descrição de frameworks. Um framework, assim como uma aplicação desenvolvida segundo o paradigma de orientação a objetos, carece de descrições estática e dinâmica tanto no nível de classes separadamente, como no nível do framework como um todo. Com isto, pode-se considerar que as metodologias OOAD podem fornecer subsídios para complementar as metodologias de desenvolvimento de frameworks.

Um outro ponto comum é a recomendação do uso de *patterns* (padrões). *Patterns* representam o registro de experiência em projeto. É unânime entre os autores das metodologias apresentadas que não utilizar *patterns* prejudica o desenvolvimento, tanto em termos de dispender mais tempo para chegar à solução de problemas de projeto, como em termos do risco de não ser produzida a solução mais adequada. Também é uma característica comum, que nenhuma das três metodologias estabelece especificamente, como utilizar *patterns*.

O teste de um framework necessariamente passa pelo desenvolvimento de aplicações. Todas as metodologias demandam este requisito para o encerramento do processo de desenvolvimento.

Características específicas

Os aspectos em que as metodologias de desenvolvimento de frameworks apresentadas diferem estão relacionados à ênfase a algum aspecto particular ao longo do desenvolvimento, e não a pontos conflitantes.

O que diferencia o Projeto Dirigido por Exemplo das demais metodologias é a ênfase à análise do maior número possível de aplicações já desenvolvidas. Idealmente, é do conjunto de aplicações que se origina a descrição do domínio (transposta para as classes abstratas), assim como é nas diferenças entre as aplicações que se identifica a flexibilidade necessária ao framework.

A característica principal do Projeto Dirigido por *Hot Spots* é a ênfase sobre os pontos de flexibilidade do framework, ou seja, os *hot spots*. Procedida a descrição do domínio, são buscados e documentados os *hot spots*; o desenvolvimento consiste em adaptar a estrutura de classes aos *hot spots* identificados. Se após um ciclo de desenvolvimento o framework for julgado inadequado, a atenção volta-se à redefinição dos *hot spots*.

Os aspectos que diferenciam a metodologia proposta pela Taligent das demais metodologias, são os princípios para o desenvolvimento de frameworks:

- substituição da visão de desenvolver um framework que cubra as características e necessidades de um domínio, pela visão de desenvolver um conjunto de frameworks

estruturalmente menores e mais simples que usados conjuntamente, darão origem às aplicações;
- preocupação ao longo do desenvolvimento em tornar o uso do framework o mais simples possível, através da minimização da quantidade de código que o usuário deve produzir.

Estas características das três metodologias não são mutuamente exclusivas:
- todas podem comportar a busca de informações do domínio em aplicações já desenvolvidas, como apregoado pelo Projeto Dirigido por Exemplo;
- todas podem comportar a ênfase à busca de *hot spots* na estrutura de classes, proposta na abordagem Projeto Dirigido por *Hot Spots*;
- todas podem comportar a ênfase em minimizar a quantidade de código que o usuário deve desenvolver, como proposto pela empresa Taligent;
- desenvolver frameworks extensos que cubram todas as necessidades das aplicações de um domínio ou desenvolver pequenos frameworks que devem ser usados em conjunto para a geração de aplicações, como propõe a abordagem da empresa Taligent, é uma decisão de projeto e pode ser adotada por qualquer metodologia.

Generalização

Generalizando o que é apresentado nas três propostas pode-se afirmar que as três metodologias de desenvolvimento de frameworks estabelecem que o desenvolvimento de um framework passa pelas seguintes etapas:
- Aquisição de conhecimento do domínio;
- Construção da estrutura de classes do framework (modelagem do framework);
- Implementação do framework;
- Avaliação do framework, através do desenvolvimento de aplicações;
- Refinamento do framework a partir da aquisição de novos conhecimentos do domínio.

Nenhuma das metodologias estabelece técnicas de modelagem capazes de conter a descrição de projeto de frameworks ou detalha o processo de desenvolvimento. Isto sugere a necessidade de buscar subsídios em outras áreas, como as metodologias OOAD, para a definição de um procedimento concreto de desenvolvimento.

3.3 Padrões no Desenvolvimento de Frameworks

Patterns constituem uma abordagem recente em termos de reutilização de projeto, no contexto do desenvolvimento de software orientado a objetos. A principal questão tratada é como proceder para registrar - para poder reutilizar em um desenvolvimento de software - a experiência de projeto adquirida em desenvolvimentos anteriores. Segundo vários autores [JOH 88], [BEC 94] e [PRE 94] a noção de *patterns*, bem como a origem da expressão *pattern* (em Português, padrão), está na obra do arquiteto Cristopher Alexander. Alexander desenvolveu uma linguagem padrão ("*pattern language"*) para permitir às pessoas projetarem suas próprias casas e comunidades (construção civil) [ALE 77]. A linguagem de Alexander permite iniciar uma descrição em uma escala de abstração elevada e utilizar o mesmo padrão de especificação (formato) para refinar a descrição. Apesar do uso da expressão "linguagem" o padrão de descrição de Alexander se baseia em texto estruturado, e não em modelos formais.

3.3.1 Catálogo de Padrões de Projeto

A partir da experiência adquirida na identificação de padrões de projeto (*design patterns*) sobre frameworks (principalmente sobre os frameworks ET++ e InterView), Gamma, Helm, Johnson e Vlissides produziram um catálogo com vinte e três padrões de projeto (*design patterns*) de uso geral [GAM 94]. Cada padrão de projeto representa uma solução para um problema freqüente de projeto, que visa auxiliar na escolha de alternativas de projeto que produzam software reutilizável, evitando alternativas que comprometam a reusabilidade.

Os padrões de projeto foram originados a partir da observação de que diferentes partes dos frameworks possuíam uma estrutura de classes semelhante. Isto demonstrou a existência de padrões de solução para problemas de projeto semelhantes, e que se repetiam à medida em que se procurava produzir uma estrutura flexível (para extensão ou adaptação).

Um padrão de projeto corresponde a uma microarquitetura, em que são definidas as classes envolvidas, suas responsabilidades e a forma como cooperam. O uso de um padrão de projeto consiste em incluir as classes da microarquitetura escolhida, na estrutura de classes do sistema em desenvolvimento (ou fazer com que classes já existentes assumam as responsabilidades correspondentes às classes da microarquitetura), de modo a incorporar ao sistema a funcionalidade desejada. Uma abordagem de uso metódico de padrões de projeto é proposta por Beck [BEC 94].

A descrição dos padrões de projeto adotada no catálogo é basicamente textual. Baseia-se nos *patterns* de Alexander, mas adota uma estrutura própria. No texto estruturado são incluídos diagramas de classes e, eventualmente, diagramas de seqüência para descrever a colaboração entre classes.

Através do enfoque de padrões de projeto, foi produzida uma classificação e descrição de soluções de problemas de projeto existentes. Estas soluções foram encontradas em sistemas existentes e identificadas como de uso geral. O catálogo tem o mérito de torná-las disponíveis a desenvolvedores de software menos experientes. A inserção dos padrões de Gamma tende a tornar os projetos mais flexíveis, incrementando a manuteniblidade e a reusabilidade.

3.3.2 Metapadrões

Metapadrões (*metapatterns*) [PRE 94] constituem uma abordagem complementar à abordagem dos padrões de projeto [GAM 94]. Os metapadrões se originaram da observação de aspectos comuns nos padrões de projeto. Pree observou e classificou alguns arranjos de funcionalidade sobre métodos, que se repetem em diferentes padrões de projeto - o que denominou metapadrões.

Os padrões de projeto de Gamma tratam da combinação de um conjunto de classes com divisão de responsabilidades, para a solução de um problema de projeto específico - como por exemplo, a questão do estado de um objeto ser alterado em função da mudança de estado de outro objeto, tratada pelo padrão de projeto *Observer*. Os metapadrões trabalham em um nível de abstração superior, em relação aos padrões de projeto. Definem como manter a flexibilidade das classes, encapsulando a funcionalidade sujeita a alteração em métodos *hook*, chamados por métodos *template*[23].

---

[23] Um método *template* é um método que ao ser executado invoca pelo menos um outro método - um método *hook*. O método *template* comporta a parte imutável de um procedimento. A flexibilização do procedimento ocorre através da troca do(s) método(s) *hook* invocado(s).

Segundo a proposta de Pree, os metapadrões se prestam principalmente a produzir a documentação de projeto de um framework cuja estrutura já esteja estável[24]. Neste caso, os metapadrões presentes no framework são destacados na documentação de projeto, ressaltando o que se decidiu manter flexível e de que modo, e o que se decidiu manter inflexível. A noção básica é que a inclusão de um método *template* define uma estrutura de algoritmo que se deseja manter estável. A flexibilidade permitida pelo projetista é concentrada na funcionalidade acessada através da chamada a métodos *hook* (pelo método *template*). A documentação de projeto destacando as características de flexibilidade usando metapadrões registra decisões de projeto.

A utilização de metapadrões em desenvolvimento de projeto, segundo proposto por Pree, está intimamente ligada à abordagem de projeto *hot-spot-driven design* [PRE 94]. Nesta abordagem são identificadas as partes do projeto que se deve manter flexíveis, que constituem os *hot spots*. Em contraste, as partes estáveis constituem os *frozen spots*. *Frozen spots* podem dar origem a métodos *template* - que tem comportamento flexível em função do uso de métodos *hook*. A identificação de estruturas de algoritmo estáveis (*frozen spots*) que podem ter partes variáveis (*hot spots*), define a necessidade de um metapadrão. A escolha do metapadrão depende da flexibilidade requerida (por exemplo, se há a necessidade de alteração de comportamento em tempo de execução), e de características específicas do projeto tratado.

Os metapadrões constituem uma proposta de padrão de solução para problemas de projeto distinta da proposta de Gamma. Metapadrões estabelecem como manter a flexibilidade de um procedimento, implementando-o através de métodos *template* e *hook*. O método *template* encapsula a parte estável do procedimento (a generalidade), enquanto que através do uso de diferentes métodos *hook*, podem-se estabelecer variações de comportamento para o procedimento.

3.3.3 Influência de Padrões no Desenvolvimento de Frameworks

Comparando as duas abordagens apresentadas, observam-se diferenças em abstração e granularidade. Em termos de nível de abstração, constata-se que os metapadrões apresentam soluções de projeto mais abstratas. Os padrões de projeto, apesar de constituirem soluções de projeto de uso geral, são direcionados à solução de determinados tipos de problema. Os metapadrões não são voltados a solucionar problemas de projeto específicos, mas definem estruturas de implementação que apresentam flexibilidade.

Os padrões de projeto apresentam estruturas de maior granularidade. São constituídos por estruturas de classes, com métodos e atributos definidos. Os metapadrões apenas relacionam dois métodos (um método *template* e um método *hook*), que podem ser de classes distintas ou de uma mesma classe. Cada par de métodos *template* e *hook* caracteriza uma estrutura de metapadrão.

A origem dos padrões de projeto e metapadrões está na busca de estruturas de projeto flexíveis, que são obtidas através da distribuição de uma responsabilidade entre um conjunto de classes (ao invés de concentrar sobre uma classe), procurando mantê-las o mais desacopladas quanto possível.

---

[24] Esta noção de estabilidade se refere ao fato de que um framework que foi usado para gerar poucas aplicações, possivelmente precisará ser alterado (à medida em que novas informações do domínio são obtidas, a partir do desenvolvimento de novas aplicações). Apenas um framework usado para gerar muitas aplicações de um domínio chega a uma estrutura menos sujeita a alterações [JOH 93] [GAM 94].

A distribuição de responsabilidade é observada nas diferentes abordagens de *patterns*. Os padrões de projeto do catálogo de Gamma [GAM 94] se utilizam deste expediente para a solução de situações de projeto específicas. Por exemplo:

- **Abstract factory**: a resposta à chamada de um método de criação de objeto consiste em chamar um método de criação de outro objeto: a possibilidade de troca do objeto chamado (com conseqüente troca de comportamento) confere flexibilidade à estrutura;
- ***Observer***: parte da responsabilidade de processamento decorrente da mudança de estado de um objeto é distribuída a observadores - o baixo acoplamento entre sujeito e observadores permite troca de objetos, com conseqüente troca de comportamento, conferindo flexibilidade à estrutura.

A estrutura de organização de métodos *template* e *hook* caracteriza o uso de distribuição de responsabilidade pela abordagem metapadrões. Quando um objeto executando um método *template* solicita a outro objeto a execução de um método *hook*, está transferindo parte da responsabilidade (pela funcionalidade associada à execução do método *template*) a este objeto. Os diferentes metapadrões classificam o nível de flexibilidade obtido, bem como a organização estrutural dos objetos envolvidos.

Assim, pode-se afirmar que um mérito da abordagem de *patterns* (padrões de projeto e metapadrões) reside na identificação, classificação e descrição de soluções existentes, possibilitando o reuso destas soluções.

Os padrões de projeto [GAM 94] consistem em soluções semi-prontas para usar no desenvolvimento de projetos (para auxiliar na definição da estrutura de classes do framework como proposto por Beck [BEC 94], ou para refinar a estrutura de classes ao longo do projeto como proposto por Gamma [GAM 94]). Usar padrões de projeto demanda conhecê-los - não apenas os do catálogo, que segundo a proposta, são de uso geral, mas também padrões voltados à solução de problemas de domínios específicos, à medida em que se tornam disponíveis.

Metapadrões [PRE 94] podem ser usados sempre que se identificar a necessidade de manter a implementação flexível. Podem complementar a abordagem de padrões de projeto [GAM 94], a partir da flexibilização da implementação de padrões de projeto selecionados, e eventualmente, como originadores de padrões de projeto. Assim, como no caso dos padrões de projeto, o uso de metapadrões pode ocorrer tanto nas fases preliminares de definição da estrutura de classes, como no refinamento da estrutura.

3.4 As Etapas do Processo de Construção da Estrutura de Classes de Frameworks

A partir do exercício de uso das metodologias de desenvolvimento de frameworks, em particular na construção do framework de jogos de tabuleiro, FraG, apresentado no capítulo anterior, buscou-se detalhar a etapa de construção da estrutura de classes de frameworks. Este detalhamento foi buscado como complemento às metodologias de desenvolvimento de frameworks propostas e é descrito a seguir.

A construção de um framework passa por um conjunto de atividades, não-seqüenciais, que se repetem até a obtenção de uma estrutura de classes que satisfaça os requisitos de generalidade, alterabilidade e extensibilidade. Para maior clareza, pode-se organizar estas atividades que compõem o processo de desenvolvimento da estrutura de classes de um framework nas seguintes etapas:

- etapa de generalização;
- etapa de flexibilização;
- etapa de aplicação de metapadrões;
- etapa de aplicação de padrões de projeto;
- etapa de aplicação de princípios práticos de orientação a objetos.

A seguir são apresentadas cada uma destas etapas, com a preocupação de ilustrá-las através de situações de projeto enfrentadas ao longo do desenvolvimento do framework FraG.

3.4.1 Etapa de Generalização

A etapa de generalização consiste na identificação de estruturas idênticas nas aplicações analisadas, e na fatoração de estruturas em que se identificam semelhanças, de modo a obter uma estrutura de classes que generalize o domínio tratado. Um primeiro aspecto na busca de generalidade da estrutura do framework, é a identificação das classes que representam elementos e conceitos do domínio tratado. Isto é obtido a partir da confrontação de estruturas de classes de diferentes aplicações (implementadas ou não). Nesta etapa também são buscados elementos prontos que podem ser reutilizados, como outros frameworks e componentes. No caso do framework FraG, foi utilizado o framework MVC (Model View Controller, implementado em Smalltalk). No início do seu desenvolvimento havia disponível a implementação de um Jogo (Corrida) em Smalltalk, e o respectivo projeto. O confronto da estrutura de classes do Jogo Corrida, com estruturas de classes desenvolvidas para outros jogos[25], levaram:

- ao reconhecimento das classes *Board*, *Position* e *Dice*, como gerais para o domínio;
- à necessidade de definir uma nova classe, *Player*, originada da fatoração da classe *Piece* (do Jogo Corrida) em duas classes: *Piece* e *Player*. No Jogo Corrida a figura do jogador se confunde com a figura de seu peão. Em jogos com vários peões por jogador esta abstração não é adequada.

3.4.2 Etapa de Flexibilização

Nesta etapa o objetivo é identificar o que deve ser mantido flexível na estrutura de classes que generaliza o domínio, de modo que a estrutura possa ser especializada, gerando diferentes aplicações. É um processo de localização de *hot spots* na estrutura de classes e de escolha de soluções de projeto para modelá-los. A identificação de *hot spots* ocorre quando se determinam situações de processamento comuns às aplicações do domínio (os *frozen spots*), porém, com variações de uma aplicação específica para outra. Considerando o exemplo do framework FraG, a ação do usuário sobre os elementos do jogo para o procedimento de lances, é uma situação de processamento comum às aplicações do domínio. Durante um lance de uma partida, a ação do usuário pode se dar sobre dados (lançamento), casas (colocar, remover peões) ou outros elementos que venham a ser definidos - considerando inclusive a possibilidade de ocorrência de várias ações a cada lance. No jogo inicialmente implementado (Corrida), a única ação do jogador era o lançamento do dado - e por isso a presença da classe *DiceController* na estrutura de classes deste jogo[26] (vide figura 3.9). Em um Jogo da Velha, por exemplo, a ação do usuário ocorre sobre as casas - o que demandaria a associação de objetos

---

25 Desenvolvidas apenas modelagens, sem que fossem produzidas as respectivas implementações.

26 Cada elemento gráfico que deve responder à ação do usuário, necessita de um controlador associado (instância de subclasse da classe Controller). Esta restrição se deve ao uso do framework MVC para suportar a interface gráfica dos jogos.

controladores às casas. Generalizando o tratamento das ações do usuário no framework, foram definidas as classes *ClickModel* (abstrata) e *ClickController*, subclasses de *Model* e *Controller*, respectivamente. *ClickController* é responsável por verificar a ação do *mouse*, e *ClickModel*, por conter o protocolo de resposta a esta ação. Este protocolo é herdado pelas classes *Dice* e *Position*, do framework - o que ilustra uma situação de uso adequado de herança para reutilização de interfaces. As figuras 3.9 e 3.10 ilustram esta evolução da estrutura de classes. Observe-se que esta solução é polimórfica: instâncias de *ClickController* podem interagir com objetos de classes diferentes.

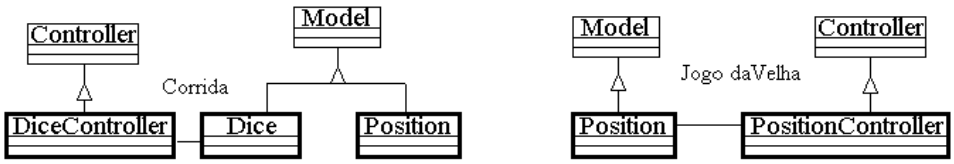

FIGURA 3.9 - Detalhe das estruturas de classes do Jogo Corrida e do Jogo da Velha

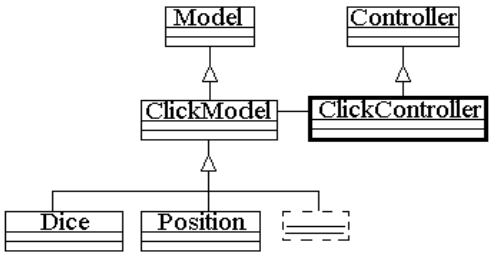

FIGURA 3.10 - A generalização do controle de ações do usuário, no framework FraG

3.4.3 Etapa de Aplicação de Metapadrões

Uma vez identificado um *hot spot*, deve-se comparar o requisito de flexibilização por ele imposto com os casos tratados por padrões existentes. Se um padrão é julgado adequado para a solução do problema, ele pode ser incorporado à estrutura de classes do framework. No caso específico dos metapadrões, o seu uso consiste em transformar um procedimento geral em um método *template*, cujo comportamento é flexibilizado através da dependência de métodos *hook*, que podem ter diferentes implementações. No caso do framework FraG, observa-se que inicializar um jogo é um procedimento comum às aplicações do domínio, porém cada aplicação tem aspectos particulares. Um método *template* pode ser usado para generalizar a estrutura do algoritmo de inicialização, deixando as particularidades de cada caso sob a responsabilidade de métodos *hook*. Isto caracteriza uma situação em que é possível usar um metapadrão. A figura 3.11 ilustra quatro ocorrências do metapadrão *unificação*, envolvendo o método "*initialize*" da classe abstrata *GameInterface* do framework FraG. Este método é responsável por promover a instanciação e inicialização dos objetos, necessários no início da execução da aplicação. No caso do domínio tratado, o processo de inicialização de um jogo segue o seguinte algoritmo:

- produzir um tabuleiro específico (com a geração de casas, jogadores, peões e dados compatíveis com este tabuleiro);
- proceder a inicialização de atributos dos objetos;
- definir a ordem de procedimento de lances dos jogadores e
- exibir estado inicial da partida e aguardar ação do usuário (procedimento de lance).

Cada um dos passos que compõem o procedimento acima, que define o método "*initialize*" (um método *template*), é implementado na forma de um método *hook*

(respectivamente, "*generateBoard*", "*initializeObjects*", "*first:*" e "*enableStart*"), como ilustrado na figura 3.11.

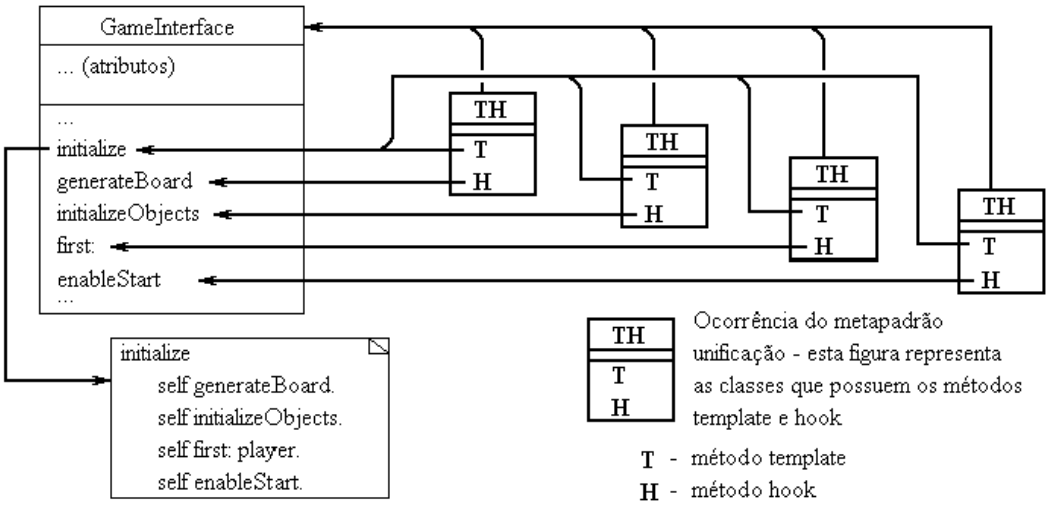

FIGURA 3.11 - Destaque ao uso do metapadrão unificação

3.4.4 Etapa de Aplicação de Padrões de Projeto

A incorporação de padrões de projeto consiste em incluir as classes de um padrão selecionado na estrutura de classes em desenvolvimento (ou fazer com que classes já existentes assumam as responsabilidades correspondentes às classes do padrão de projeto). A implementação do método "*generateBoard*" do framework FraG (citado no item anterior) adota o padrão de projeto "*Abstract Factory*". Em conseqüência disto, a única responsabilidade do método, que deve ser definido na subclasse concreta de *GameInterface* (específica de cada aplicação), é chamar o método construtor da classe correspondente ao tabuleiro da aplicação (subclasse concreta de *Board*). A figura 3.12 ilustra a aplicação do padrão de projeto "*Abstract Factory*" no framework. Esta solução produz acoplamento apenas da subclasse de *GameInterface* a uma subclasse de *Board*, e não o contrário. Assim, um tabuleiro pode ser reutilizado com diferentes interfaces.

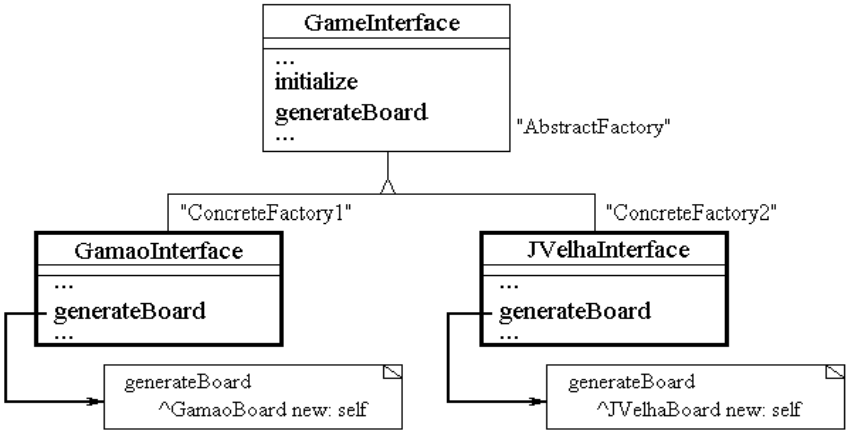

FIGURA 3.12 - Destaque ao uso do padrão de projeto "Abstract Factory"

O padrão de projeto "*Observer*" foi usado para fazer com que o objeto tabuleiro[27] seja afetado pela ação do usário: o objeto tabuleiro é o observador, e as

[27] Instância de subclasse concreta de *Board*.

instâncias de subclasses de *ClickModel*, os sujeitos. Quando um dos sujeitos é submetido a uma ação do usuário, o observador é notificado, devendo produzir uma resposta adequada. A aplicação do padrão de projeto "*Observer*" na definição da estrutura do framework desenvolvido possibilitou que fosse deixada para o usuário do framework apenas a tarefa de implementar os métodos que produzem a resposta do tabuleiro - todo o procedimento de interação com o *mouse* é reutilizado. Além disto, há um baixo acoplamento entre observador e sujeitos. As classes de observadores não guardam referência dos sujeitos, e a referência do observador mantida pelos sujeitos independe da classe do observador - assim, casas e dados podem ser reutilizados em diferentes tabuleiros. A figura 3.13 ilustra o uso do padrão de projeto "*Observer*" no framework FraG.

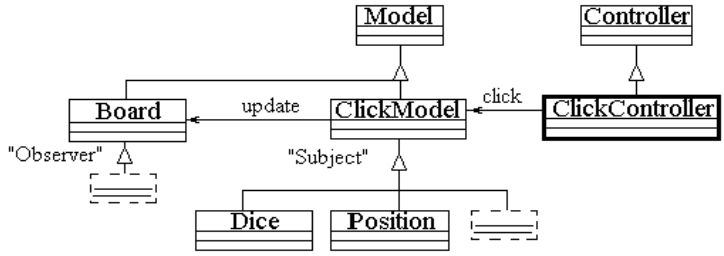

FIGURA 3.13 - Destaque ao uso do padrão de projeto "Observer"

Adotou-se o padrão de projeto "*Decorator*" para estender a funcionalidade da classe *Position*, permitindo a uma casa executar ações sobre os peões que a ocupam[28]. A ação específica é responsabilidade de um objeto *conteúdo*, associado à casa através do objeto decorador[29]. Com isto, as subclasses de *Position* tornam-se menos específicas - e, portando, mais adequadas à reutilização - pois, as mesmas casas podem ser usadas com ou sem ações associadas. O uso do padrão "*Decorator*" associado à classe *Position* está descrito na figura 3.14. Neste caso, o uso do padrão permitiu que a funcionalidade de *Position* fosse estendida através de composição de objetos.

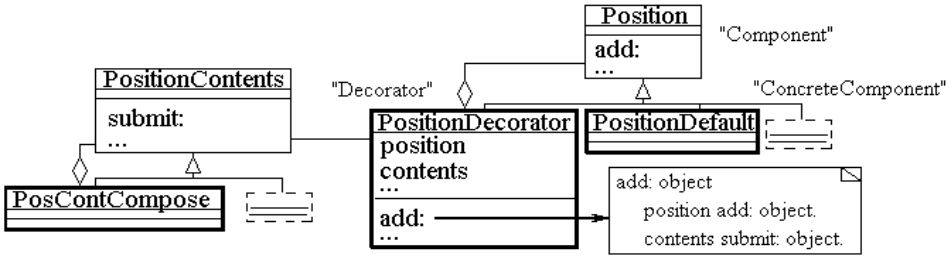

FIGURA 3.14 - Destaque ao uso do padrão de projeto "Decorator"

A figura 3.15 apresenta o uso do padrão de projeto "*Decorator*" associado à classe *Dice* para bloquear a ação de um dado de seis faces (para impedir seu relançamento). Isto exigiu o refinamento da estrutura inicialmente definida. Uma alternativa possível seria o uso de herança: gerar uma subclasse de *DiceSixFaces* com esta funcionalidade. Esta solução foi considerada uso inadequado de herança: a

---

28 Isto é dispensável em jogos como Jogo da Velha ou Damas, mas é necessário em jogos como Banco Imobiliário e RPG.

29 O objeto conteúdo é uma instância de uma subclasse concreta de PositionContents e é responsável por um tipo de ação específica, e o objeto decorador é uma instância da classe PositionDecorator.

funcionalidade de bloqueio estaria sendo forçada sobre uma classe para reutilizar sua implementação. O uso do padrão "*Decorator*" possibilitou bloquear dados a partir de composição de objetos, sem alteração da classe abstrata *Dice*, ou das suas subclasses concretas, que implementam dados específicos.

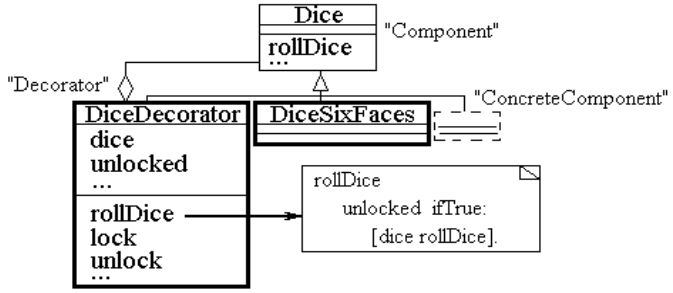

FIGURA 3.15 - Destaque ao uso do padrão de projeto "Decorator"

3.4.5 Etapa de Aplicação de Princípios Práticos de Orientação a Objetos

Para a obtenção de um framework com uma estrutura de classes flexível é necessário seguir princípios de projeto orientado a objetos[30], como o uso de herança para reutilização de interfaces, ao invés do uso de herança para reutilização de código (a delegação é mais adequada que o uso indevido de herança); reutilização de código através de composição de objetos; preocupação em promover polimorfismo, na definição das classes e métodos, de modo a possibilitar acoplamento dinâmico, etc. [JOH 88] [MEY 88].

O uso do padrão de projeto "*Decorator*" para possibilitar o bloqueio de um dado no framework FraG (descrito no item anterior), ilustra uma situação em que a delegação se mostrou uma alternativa de projeto mais adequada que herança.

3.5 Suporte das Metodologias OOAD ao Desenvolvimento de Frameworks

A indefinição de mecanismos de descrição do projeto de frameworks é uma das características das metodologias de desenvolvimento de frameworks, conforme anteriormente citado. As metodologias de desenvolvimento de frameworks existentes estão, em sua maioria, voltadas a produzir código e não utilizam notação de alto nível, fazendo com que o resultado do processo de desenvolvimento seja o próprio código do framework - um modelo de domínio expresso através de uma linguagem de programação [JOH 97]. Como forma de contornar o problema da dificuldade de entender frameworks apenas a partir de código, autores de metodologias propuseram mecanismos de descrição que abrangem fundamentalmente a questão de como utilizar os frameworks. Este é o caso exemplo, dos padrões de documentação propostos por Johnson [JOH 92], autor da metodologia Projeto Dirigido por Exemplo, como também é o caso da notação de metapadrões proposta por Pree [PRE 94], autor da metodologia Projeto Dirigido por *Hot Spot*.

---

30 O que não é exclusivo do contexto de frameworks, mas também se aplica ao desenvolvimento de aplicações.

Esta cultura de desenvolvimento de software sem preocupação com a sua documentação está arraigada não apenas na comunidade de frameworks, mas também na comunidade de desenvolvimento de software em geral. Referências a isto podem ser encontradas na literatura de Engenharia de Software. Por exemplo, em 1981 Commer enunciava uma regra enfática: "documente um programa ou jogue-o fora" [COM 81]. Em 1986 Parnas afirmou que a "maioria dos programadores encarava documentação como um mal necessário, que era produzida após a elaboração dos programas, apenas por imposição de algum burocrata" [PAD 86]. Afirmou ainda que a solução para este obstáculo estava em buscar mecanismos de documentação que fossem de encontro às necessidades tanto dos desenvolvedores de programas, quanto daqueles que posteriormente manuseiem estes programas.

Johnson classifica o tipo de documentação necessária para frameworks conforme o tipo de público a que é dirigida. Um primeiro público corresponde às pessoas que estão tentando decidir que framework usar. Para este público a documentação deve descrever o que pode ser feito a partir do framework. Um segundo público são as pessoas que desejam aprender a desenvolver aplicações típicas a partir do framework. Para este público Johnson recomenda uma documentação que ensine da forma mais sumária possível (e sem entrar em detalhes do projeto de frameworks), como feito pelo *cookbook* do ambiente VisualWorks [PAR 94]. O terceiro público corresponde às pessoas que necessitam conhecer em detalhes o projeto do framework. Isto inclui usuários responsáveis pelo desenvolvimento de aplicações mais complexas (cujas necessidades não são atendidas pelo tipo de documentação voltada ao desenvolvimento de aplicações elementares), bem como pela alteração do framework (manutenção) [JOH 94]. No presente trabalho tem-se a perspectiva de adotar mecanismos de descrição do projeto de frameworks capazes de atender às necessidades deste terceiro público e, como preconizado por Parnas, sejam mecanismos de auxílio ao processo de desenvolvimento de frameworks, ao invés de transtorno adicional ao esforço de produção de frameworks.

Como já discutido, o desenvolvimento de frameworks é um processo de evolução iterativa de uma estrutura de classes. A estrutura resultante deve apresentar as características de generalidade e flexibilidade. A complexidade deste processo sugere a necessidade de mecanismos que registrem os resultados das várias etapas do processo de desenvolvimento, bem como registrem explicitamente decisões de projeto, como a identificação das partes flexíveis de um framework. A adoção de mecanismos adequados de descrição de projeto possibilita o registro, classificação e organização de acesso ao conjunto de informações referentes ao projeto de um framework. Isto pode facilitar o desenvolvimento de frameworks, orientando os ciclos de evolução, a etapa de manutenção (o refinamento do framework, à medida em que se obtém novos conhecimentos do domínio), bem como para utilização do framework (entender o framework a partir do estudo de seu projeto). Como frameworks são estruturas baseadas no paradigma de orientação a objetos, é natural que uma alternativa para descrever o seu projeto seja o suporte denotativo provido pelas metodologias OOAD.

A seguir são discutidos os requisitos de uma especificação baseada no paradigma de orientação a objetos e de uma notação para projeto de frameworks.

3.5.1 As Visões de um Sistema

Uma modelagem de sistema construída segundo a abordagem de orientação a objetos - assim como sob outras abordagens - deve ter a capacidade de descrever este sistema sob uma ótica estática e sob uma ótica dinâmica.

Espera-se da modelagem estática, a descrição dos elementos de um sistema. Em um sistema orientado a objetos espera-se que a modelagem estática seja eficiente em descrever as classes, seus atributos e relacionamentos. Em contrapartida, a elaboração da modelagem dinâmica de um sistema orientado a objetos deve levar à identificação e descrição da colaboração entre objetos e dos métodos das classes.

A descrição da estrutura estática neste contexto, é feita por um diagrama de classes (com esta ou outra denominação), que contém classes e associações entre classes (podendo conter ou não, atributos, métodos, subsistemas, aspectos de modelagem dinâmica incluídos etc.). Nas várias metodologias o diagrama de classes é basicamente uma variação do diagrama de Entidade-Relacionamento (ER) [CHE 76], onde as classes correspondem às entidades e as associações entre classes, aos relacionamentos. Em função dos acréscimos sintáticos, uma técnica de modelagem pode ser mais expressiva do que outra, para destacar determinados detalhes da implementação, como a diferenciação entre associação estática e dinâmica adotada em OOSE [JAC 92] e ausente em OMT [RUM 94], por exemplo.

O objetivo da modelagem dinâmica sob a abordagem de orientação a objetos, é descrever a colaboração entre objetos, bem como identificar e detalhar (para posterior implementação) os métodos das classes. Nas metodologias variam tanto as técnicas de modelagem usadas, como a heurística associada à tarefa de identificar e descrever métodos. De um modo geral, as metodologias usam diagrama de transição de estado (com variações sintáticas e acréscimos diversos), diagramas de seqüência e outras técnicas de modelagem.

Além da questão da ênfase da modelagem ser estática ou dinâmica, um outro aspecto a considerar é o foco da modelagem, ou seja, se volta a atenção ao sistema como um todo, ou a uma classe isoladamente. Em uma aplicação orientada a objetos, a implementação consiste em um conjunto de classes (que possuem atributos e métodos). Implementar um sistema é, portanto, implementar as classes que o constituem. Porém, para compreender um sistema, normalmente não basta observar cada uma de suas classes, mas também é necessário dispor de alguma descrição da cooperação entre as classes. As técnicas de modelagem usadas pelas metodologias também variam em termos da capacidade de descrever o sistema como um todo ou as suas partes (classes).

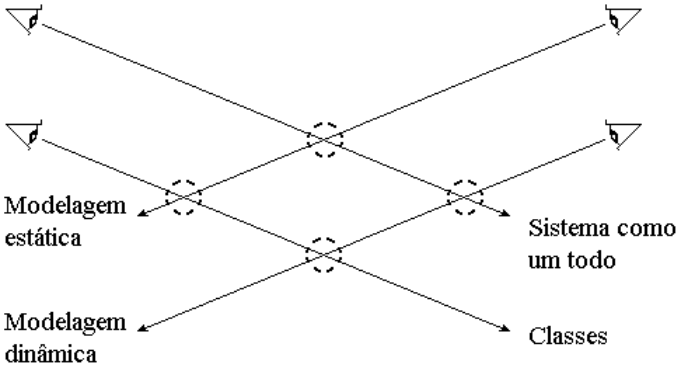

FIGURA 3.16 - Visões de uma especificação de sistema

A descrição isolada de cada classe é a descrição mais próxima da implementação: isto inclui uma descrição completa de cada classe, incluindo os atributos (tipos, estruturas de dados, restrições etc.), os métodos (detalhamento do algoritmo associado ao método, explicitação da dependência entre métodos etc.) e a interface, a parte do objeto externamente visível.

A descrição de um sistema orientado a objetos como um todo, é a descrição da cooperação entre as classes que compõem o sistema. No desenvolvimento de um sistema, esta visão é fundamental para a distribuição de responsabilidades entre classes desenvolvidas (o que resulta na definição dos métodos destas classes) e classes reutilizadas, como também para a integração dessas classes. Na manutenção, auxilia na localização de que classes devem ser alteradas para modificar ou incrementar a funcionalidade do sistema. Na figura 3.16 são ilustradas as diferentes visões de uma especificação produzida a partir do paradigma de orientação a objetos.

3.5.2 Aplicabilidade das Notações das Metodologias OOAD na Documentação de Frameworks

Framewoks, como são estruturas de classes, podem, a princípio, ser descritos a partir da notação de metodologias OOAD. Assim, uma primeira questão é a escolha de uma notação adequada. Observa-se desuniformidade no aspecto de expressividade na notação das metodologias em geral. Avaliando seis metodologias OOAD selecionadas para comparação[31], a saber, metodologias OOAD de Coad e Yourdon [COA 92], [COA 93], OMT [RUM 94], OOSE [JAC 92], Fusion [COL 94], de Martin e Odell [MAR 95] (sendo esta última apenas de análise) e UML[32] [RAT 97], pode-se considerar menos expressivos os recursos denotativos das metodologias de Coad e Yourdon e de Martin e Odell. As demais metodologias apresentam vantagens e desvantagens quando comparadas entre si. UML, como uma tentativa de unificação de um conjunto de metodologias, que inclui OMT e OOSE, deveria se mostrar mais espressiva que estas. De fato, isto ocorre quando avaliados os recursos das técnicas de modelagem absorvidas destas metodologias, quase sempre estendidos em relação às propostas originais. Porém, UML apresenta deficiências, como a incapacidade de descrever o algoritmo dos métodos - que também ocorre em OMT, mas não em OOSE. Assim, exceto por esta incapacidade e pela incapacidade de estabelecer restrições de seqüenciamento de situações de processamento (casos de uso), recurso presente na metodologia Fusion, pode-se considerar que UML contém a proposta de notação mais expressiva.

Um problema das notações em geral, é a falta de ligação semântica entre as diferentes técnicas de modelagem de modo a garantir a consistência da especificação como um todo, para que esta não seja apenas um conjunto de modelos isolados. Por exemplo, as metodologias que trabalham com a descrição de situações de processamento de um sistema não estabelecem o mecanismo de refinamento destas situações de processamento que permita um aprofundamento gradual do nível de abstração desde este nível até a descrição dos algoritmos dos métodos. Assim, têm-se descrições em diferentes níveis de abstração, porém, sem uma interligação semântica. Um outro problema é a indefinição semântica de elementos conceituais modelados. Por exemplo, nas metodologias que adotam diagramas de transição de estados, os estados são especificados apenas através de um nome. Não existe uma associação entre os estados definidos e os atributos (e respectivos valores) que definem cada estado. Assim, pode-se afirmar que a semântica dos estados não é definida na notação (apesar da descrição das

---

31 A escolha destas metodologias foi feita com base em um conjunto de publicações que procedem comparações entre metodologias e destacam estas como estando entre as que apresentam mais características desejáveis em relação a outras [COL 97] [FOW 94], [MON 92], [PAS 94], [PER 95].

32 UML consiste em uma notação, sem proposta de processo de desenvolvimento. Apesar disto, como a questão ora tratada é notação de especificações baseadas em orientação a objetos, foi incluída como uma metodologia OOAD.

metodologias em geral, transmitir a noção de que o estado de um objeto é definido pelos valores assumidos por seus atributos). Observa-se assim, que as lacunas semânticas são problemas encontrados nas metodologias e constituem um obstáculo a procedimentos automatizados de verificação de consistência de especificações, bem como à geração automática de código.

As metodologias OOAD tradicionais estão voltadas a produzir uma aplicação específica, sem se preocupar com o desenvolvimento ou o uso de artefatos de software reutilizáveis. Metodologias OOAD mais recentes, como Catalysis [DSO 97] e RSEB [JAC 97], incluem a noção de reuso de artefatos de software na construção de uma aplicação. No contexto do desenvolvimento de aplicações específicas, como tratado pelas metodologias OOAD em geral, não existe a noção de estrutura de classes a ser completada, ou a definição de partes da estrutura que podem ser alteradas. No desenvolvimento do projeto de um framework é estabelecida a flexibilidade da estrutura para a geração de aplicações e as partes de sua estrutura que podem (ou devem) ser estendidas ou modificadas. Isto é decisão de projeto e deve fazer parte da especificação, pois é uma informação essencial para o uso, bem como para a manutenção do framework desenvolvido. Assim, um obstáculo para descrever o projeto de frameworks a partir das notações das metodologias OOAD é a sua incapacidade em descrever aspectos específicos da abordagem de frameworks.

O destaque dos padrões de projeto ou metapadrões pode ser usado como complemento, para registrar as partes flexíveis da estrutura do framework. Porém, apenas o registro isolado de cada ocorrência só consegue definir uma visão isolada de cada parte flexível, sem definir restrições em torno do processo de alteração da estrutura de um framework na construção de aplicações. Na descrição de um framework há a necessidade de diferenciar entre a possibilidade e a obrigatoriedade de estender partes de sua estrutura. A notação de projeto de frameworks deve ter a capacidade de descrever estas características. Por exemplo, no projeto do framework para jogos de tabuleiro, FraG, apresentado no capítulo 2, foi definido que toda aplicação deve possuir uma subclasse concreta de *GameInterface* e uma de *Board*. A definição da interface com o usuário é feita em uma subclasse de *GameInterface* e a resposta às ações do usuário, isto é, a dinâmica do jogo, é definida em uma subclasse de *Board*. As classes *GameInterface* e *Board* são abstratas e não possuem subclasses concretas na estrutura do framework. Assim, é obrigatória a criação de subclasses destas classes no desenvolvimento de qualquer aplicação. Toda aplicação também deve apresentar pelo menos uma subclasse concreta de *Player*. O framework FraG possui uma subclasse concreta desta classe. Neste caso, é possível (e não obrigatório) criar outras subclasses de *Player*. O framework dispõe de implementações concretas para posições (repositórios de peões) e para dados (de seis faces) e respectivas visões (segundo a abordagem MVC). Se uma aplicação a ser construída necessitar de dados ou posições, podem ser reusadas as classes concretas do framework que definem dados e posições ou podem ser criadas novas classes. Observando as janelas de três aplicações desenvolvidas sob o framework FraG, apresentadas na seção 2.2, verifica-se por exemplo, que a do Jogo da Velha não possui dado. Assim, nem toda aplicação utiliza dado e posição, porém, caso uma aplicação específica necessite, é possível reusar as classes concretas do framework ou definir novos tipos de dado (criar subclasses concretas de *Dice*) ou de posição (sublasses de *Position*) ou novas visões para para estes elementos.

O exemplo das restrições em torno da extensão da estrutura do framework FraG acima descrito, ilustra o caso geral dos frameworks: nem toda parte flexível da estrutura de um framework requer uma atuação por parte do desenvolvedor de aplicações, seja

pela disponibilidade de implementação adequada no framework ou porque determinada parte da estrutura do framework não exerce influência na aplicação a ser desenvolvida.

Verifica-se que é possível usar notações de metodologias OOAD para a descrição de frameworks. Porém, o uso deste tipo de notação se confronta com um conjunto de obstáculos. Primeiro, a necessidade de escolher uma notação expressiva; segundo, a necessidade de contornar as lacunas semânticas da notação escolhida; terceiro buscar solução adequada para a necessidade de explicitar as decisões de projeto referentes à forma de usar o framework - a explicitação das partes alteráveis, bem como das restrições em torno das alterações.

Considerando a tendência de UML tornar-se um padrão de notação de estruturas baseadas em orientação a objetos [COL 97], bem como a comparação das notações das várias metodologias, conclui-se pela adequação da adoção da notação de UML para a descrição de frameworks, sendo necessário porém, estender a notação com vista a atender às necessidades específicas de frameworks. No capítulo 6 são descritas as extensões propostas à notação de UML, adotadas no ambiente SEA.

3.6 Suporte Ferramental Disponível para o Desenvolvimento de Frameworks

Os ambientes de desenvolvimento de software (ADS) constituem o caminho natural no desenvolvimento de metodologias e ferramentas. São compostos a partir da integração de ferramentas, de modo a facilitar a construção de software.

Na década de setenta era comum que programas em cartões perfurados fossem compilados em *mainframes* durante a noite para, no dia seguinte, após uma execução, verificar se o programa "funcionou". Os ambientes de desenvolvimento de programas, como o Turbo Pascal, constituíram uma evolução a esta situação. Nestes ambientes se edita, compila, depura e executa programas de uma forma mais amigável, que executar cada tarefa a partir de programas isolados.

De um ambiente de desenvolvimento de software se exige mais do que apoio à implementação, como ocorre com os ambientes de desenvolvimento de programas. Espera-se apoio às demais etapas do ciclo de vida do software, a partir de facilidades como editores dirigidos por sintaxe - inclusive gráficos - para apoio à análise, projeto e implementação; mecanismos de validação dos produtos de cada etapa; facilidades para prototipação; apoio à implementação na linguagem alvo. Este tipo de ambiente de desenvolvimento de software, que apóia a evolução do software ao longo das etapas do ciclo de vida, também recebe a denominação genérica de CASE (Computer-Aided Software Engineering) ou ferramenta CASE.

Os ambientes de desenvolvimento de software constituem um suporte importante para o desenvolvimento de software, na medida em que a disponibilidade de ferramentas integradas facilita o registro e o acesso ao conjunto de informações geradas ao longo do desenvolvimento, minimiza a introdução de erros durante o desenvolvimento e se o ambiente dispuser de mecanismos de verificação de consistência de especificação, possibilita a validação do produto de cada fase do ciclo de vida. Este conjunto de características faz do uso de ambientes de desenvolvimento de software um fator de incremento da produtividade e da qualidade no desenvolvimento de software .

Especificamente no escopo da abordagem de orientação a objetos, a carência de ferramentas para apoio ao desenvolvimento de software orientado a objetos foi apontada como um dos fatores que retardaram a adoção desta abordagem na produção comercial

de software. Em 1990 durante o evento OOPSLA/ECOOP, Yourdon, um dos autores de métodos orientados a objetos, afirmou que a passagem (da abordagem estruturada) para a orientação a objetos levaria pelo menos cinco anos em função da inexistência de ferramentas de apoio ao desenvolvimento de software [DEC 91].

Atualmente não existem ferramentas voltadas especificamente ao desenvolvimento de frameworks. Desenvolvedores que não utilizam notação de alto nível dispõem dos ambientes de desenvolvimento de programas para produzir o código de seus frameworks, como o ambiente VisualWorks, para a linguagem Smalltalk [PAR 94]. O framework Luthier [CAM 97] é um exemplo de framework desenvolvido neste ambiente, cuja documentação de projeto consiste de uma descrição das colaborações entre as classes do framework através de contratos [HEL 90].

Existem ainda ferramentas e ambientes de desenvolvimento de software, com características e capacidades diversas, voltadas ao desenvolvimento de aplicações orientadas a objetos. Estes ambientes[33] se baseiam em metodologias OOAD e constituem um suporte ferramental para a aplicação dessas metodologias. Possibilitam desenvolver especificações de análise e projeto e, eventualmente, apresentam outras funcionalidades. Ambientes baseados em metodologias OOAD podem ser usados no desenvolvimento de especificações de frameworks, porém, com as mesmas restrições descritas em relação às metodologias OOAD. A questão é avaliar que benefícios podem ser obtidos a partir do uso destes ambientes no desenvolvimento de frameworks. A seguir são discutidas as características de diferentes ambientes atualmente disponíveis e avaliada a sua potencial contribuição ao desenvolvimento de frameworks.

Não estão sendo considerados na presente avaliação os ambientes de apoio ao desenvolvimento de especificações baseadas em métodos formais. A razão para isto é que diante do antagonismo entre a possibilidade de produzir especificações validáveis, possibilitada por abordagens formais, e a compreensibilidade promovida pelas técnicas de modelagem gráficas, optou-se pela última [FOW 94]. Uma razão para esta escolha foi a busca de mecanismos de descrição que também pudessem auxiliar o processo mental de criação de frameworks - e para isso, a compreensibilidade, em diferentes níveis de abstração, se tornou um requisito básico para estes mecanismos. Com isto, não se questiona a importância dos métodos de descrição formal. Apenas optou-se por não utilizá-los para a especificação de frameworks, no presente trabalho.

3.6.1 Características de Ambientes Existentes

Ambientes apóiam a geração de especificações (modelo lógico) compostas a partir da aplicação de técnicas de modelagem, voltadas a descrever determinados aspectos de projeto - como estrutura de classes, cooperação entre objetos, evolução de estados de um objeto etc - de um modo geral, baseados na notação de uma ou mais metodologias OOAD.

Um primeiro critério de classificação de ambientes corresponde à abrangência sobre o ciclo de vida de um software. Existem neste caso os ambientes que apóiam o desenvolvimento de especificações de análise, os que apóiam projeto e geração de

---

[33] O restante do capítulo se atém a ambientes de desenvolvimento de software voltados à abordagem de orientação a objetos (ADS OO), e a expressão ambiente é usada indistintamente para ambientes ou ferramentas voltados a esta abordagem - sem distinguir se há de fato integração de mais de uma ferramenta para a composição de um ambiente.

código e os que apóiam todo o ciclo de vida (estes também conhecidos como Integrated CASE ou I-CASE).

Um segundo critério considera a flexibilidade do ambiente. Os ambientes convencionais (também chamados ferramentas CASE) suportam o desenvolvimento de especificações a partir da notação de uma ou mais metodologias OOAD. Os ambientes flexíveis (ou meta-CASES's) são suportes para a geração de ambientes convencionais. Em geral, correspondem a ambientes parametrizáveis, em que novas notações podem ser introduzidas. A flexibilidade funcional para manipulação de especificações (além da flexibilidade notacional, que é uma característica básica deste tipo de mecanismo) depende da implementação de cada ambiente específico.

Quanto aos recursos dos ambientes, espera-se que disponham de processos automatizados para as várias etapas do ciclo de vida de um software, com vista a melhorar a qualidade e aumentar a produtividade do software produzido. Consideram-se desejáveis para estes ambientes as características abaixo descritas [OLI 96] [MAR 95].

Disponibilidade de interfaces gráficas para a edição e visualização dos modelos que compõem uma especificação - conforme já abordado, considera-se necessário o uso de técnicas de modelagem gráficas pela sua capacidade de facilitar a compreensão de especificações de software [FOW 94], o que pode tornar o desenvolvimento de software uma tarefa menos árdua.

Adoção de repositório centralizado para os dados do software em desenvolvimento - a adoção de um repositório que concentre todas as informações de uma especificação e que seja compartilhável entre diferentes mecanismos, que suporte funcionalidades como verificação de consistência e geração de código. Martin descreve a importância e o papel dos repositórios [MAR 95]:

> *"Um repositório é o cerne de um ambiente CASE. Ele armazena todas as informações sobre sistemas, projeto e código de uma maneira basicamente não-redundante. Ele é usado por todos os desenvolvedores. Usando ferramentas poderosas, os desenvolvedores utilizam as informações do repositório e criam novas informações que, por sua vez, são também armazenadas nele. O repositório e seus sistemas coordenadores verificam e correlacionam todas as informações armazenadas para garantir sua coerência e integridade. O repositório deve conter classes reusáveis que possibilitem aos desenvolvedores construírem sistemas rapidamente.*
>
> *Uma vez que o repositório é o cerne do ambiente CASE, ele também deve ser o cerne da metodologia empregada para a construção de sistemas. Os desenvolvedores criam projetos com a ajuda de informações do repositório. Eles constroem progressivamente seus projetos no repositório e geram código a partir deles.*
>
> *É importante entender que um repositório CASE é mais que um dicionário.*
>
> *Um dicionário contém nomes e descrições de itens de dados, processos, variáveis e assim por diante.*
>
> *Um repositório contém uma completa representação codificada dos objetos usados em planejamento, análise, projeto e geração de código. Os métodos orientados a objetos são usados para garantir a coerência e a integridade deste conhecimento por meio do processamento baseado em regras. O repositório armazena o significado representado nos diagramas CASE e reforça a*

> *coerência dentro dessa representação. Ele, dessa forma, entende o desenho, ao passo que um simples dicionário, não. Particularmente importante é que o repositório impulsiona um gerador de códigos. O desenvolvedor não precisa codificar linguagens com uma sintaxe arcaica, como, por exemplo, COBOL ou C. Atualmente os melhores geradores de código em uso, gerando COBOL ou C, produzem cem por cento do código para o sistema de produção.*
>
> *(...)*
>
> *O repositório armazena muitos tipos de objeto. A fim de manuseá-los corretamente, o repositório exige seu próprio conjunto de classes e métodos associados. Esses métodos podem ser expressos como coleções de regras ou podem ser algoritmos que processam as informações armazenadas no repositório. Dessa forma, ele tanto armazena informações sobre o desenvolvimento como ajuda a controlar sua precisão e validade.*
>
> *Qualquer diagrama em uma tela CASE é uma faceta de um conjunto mais amplo de conhecimento que pode residir no repositório. Este normalmente contém bem mais detalhes do que aquilo que está contido no diagrama. Qualquer parte desse detalhe pode ser exibida em outras janelas usando-se um mouse.*
>
> *O conhecimento que vem dos diagramas CASE deve ser coordenado para garantir que se encaixe de maneira lógica e consistente. A IBM denomina esta capacidade de serviços de repositório; a KnowledgeWare Inc., de coordenador do conhecimento; e nós nos referimos a ela como coordenador de repositórios. Um bom coordenador de repositórios é altamente complexo - capaz de manusear repositórios com milhares de regras."*

Mecanismos de verificação de consistência de especificações - sua função é garantir consistência semântica de especificações, localizando construções incorretas, ambigüidades e estruturas incompletas, preferencialmente promovendo correção automática ou dirigindo o processo de correção. Para sistemas complexos esta funcionalidade é particularmente importante, pois, pela grande quantidade de informação envolvida, a verificação de consistência pode tornar-se uma tarefa árdua, monótona ou até mesmo humanamente impossível de ser executada sem suporte automatizado.

Suporte à prototipação - consiste em suportar o desenvolvimento de protótipos da interface do sistema sob desenvolvimento e da simulação de sua operação, para verificação do atendimento dos requisitos durante a etapa de projetos.

Mecanismos de referência cruzada - consiste na capacidade de comparar informações contidas em diferentes contextos, como por exemplo, poder verificar as referências nos vários modelos de uma especificação, a um método de uma classe. *Browsers* de especificação podem ser disponibilizados em um ambiente para análises diversas.

Geração de código - é a capacidade de expressar o conteúdo do repositório em linguagem de programação. Um ambiente pode ter a capacidade de produzir código para diferentes linguagens de programação, a partir de uma mesma especificação de projeto. Para que um ambiente consiga produzir código executável (ou compilável) sem necessidade de atuação do usuário, é necessário que todas as informações necessárias para a produção do código estejam contidas na especificação. Assim, não se pode esperar, por exemplo, que uma especificação elaborada a partir na notação de UML, que não contempla a descrição do algoritmo dos métodos, possa ir além da definição da

assinatura de métodos no processo de geração de código, como é verificado, por exemplo, no ambiente Rose (*Rational Software Corporation*).

Mellor descreve três possibilidades de geração de código a partir de especificações baseadas em UML: geração de código estrutural, comportamental e traduzível [MEL 99]. A geração de código estrutural consiste em produzir esqueletos de programas a partir das especificações. O código produzido é depois completado, de forma não-automatizada. Um problema comum é que após esta etapa o código resultante não seja equivalente ao modelo lógico que o originou. Para suprir esta deficiência podem ser aplicados procedimentos de Engenharia Reversa para reconstruir os modelos a partir do código. Na geração de código comportamental o código produzido de forma não-automatizada é incorporado ao modelo lógico. O processo de tradução combina a informação dos modelos gráficos com o código escrito a mão, para produzir o código do sistema. A geração de código traduzível conta com a predefinição de estruturas de código. O processo de geração de código consiste em substituir elementos do modelo lógico pelas estruturas predefinidas, compondo assim código executável. Mellor apresenta um exemplo em que aplica esta abordagem para, de fato, gerar código a partir de *statecharts*. Em função da necessidade de ter previamente desenvolvidos todos os trechos de código que comporão o código do sistema, esta abordagem tende a ser menos flexível que as anteriores.

Se a notação utilizada para produzir especificações baseadas no paradigma de orientação a objetos tiver capacidade de descrever o algoritmo dos métodos das classes, é possível gerar código de forma completamente automatizada.

Geração de documentação - assim como na geração de código, consiste em traduzir o conteúdo do repositório em algum formato específico, para a geração de relatórios descrevendo o projeto.

Suporte à reutilização - consiste na capacidade de produzir, armazenar, recuperar, importar e utilizar artefatos de software reutilizáveis. Isto abrange reuso de código (como classes de uma biblioteca) assim como reuso de projeto (como padrões de projeto).

Oliveira procedeu um levantamento do estado da arte dos ambientes de desenvolvimento de software comerciais voltados à abordagem de orientação a objetos, em que foram avaliados trinta e três produtos. Na sua descrição dos ambientes avaliados não se encontra nenhum caso que satisfaça o conjunto de requisitos acima. Verificou que "algumas ferramentas permitem apenas a substituição do papel e do lápis pelo computador (...), ou seja, não passam de simples ferramentas para a construção de desenhos" [OLI 96].

Durante o desenvolvimento desta pesquisa foram avaliados alguns ambientes de desenvolvimento, cujas informações foram obtidas através da Internet. Em alguns casos foi obtida uma versão *shareware* para avaliação em uso. Outros ambientes foram avaliados apenas a partir das características descritas pelos fabricantes e também a partir do conhecimento das metodologias OOAD suportadas[34]. Nesta busca, além dos

[34] Atualmente é relativamente fácil obter informações sobre ambientes comerciais, uma vez que, por uma questão mercadológica, um grande número de fabricantes mantém páginas na Internet. Alguns endereços disponíveis, com links para vários fabricantes (acessados pela última vez em novembro de 1999):
http://www.rhein-neckar.de/~cetus/oo_ooa_ood_tools.html

produtos que são meros editores gráficos, encontrou-se ambientes com características importantes, como:

- **geração de código** - a situação mais comum é a geração do conjunto de classes, com seus atributos, e métodos mais elementares, como método construtor, destrutor e métodos de acesso aos atributos. Exemplos: WClass (Microgold Software) [MIC 94], Rose (Rational Software Corporation). O ambiente ObjChart (IBM) [IBM 96] permite a geração de outros métodos (relacionados na especificação), porém de forma semi-automática, exigindo a intervenção do usuário.
- **restrições semânticas entre diferentes diagramas**. Exemplo: no ambiente Rose a modificação da assinatura de um método se reflete em todos os diagramas em que este método esteja representado;
- **verificação de consistência do modelo**. Exemplo: o ambiente Rose não permite *looping* de herança (em tempo de edição); atualiza em tempo de edição o conjunto de diagramas, sempre que um método (ou classe) é alterado ou excluído.
- **execução do modelo**. Exemplo: ObjChart (IBM) permite seguir o fluxo de envio de mensagens nos diagramas produzidos, de modo semelhante à depuração de um programa.

Os ADS OO são construídos para suportar a notação de uma ou mais metodologias OOAD. Com isto, uma limitação freqüente é a incapacidade de usar técnicas de modelagem diferentes daquelas previamente estabelecidas. Outra limitação é a impossibilidade de estender a funcionalidade dos ambientes.

Como alternativa a estas limitações, existem os ambientes flexíveis (meta-CASE's). Alguns exemplos de meta-CASE's comerciais são ObjectMaker (Mark V Systems Ltd.) e Graphical Designer (Advanced Software Technologies Inc.) [UNI 96].

Além de ambientes voltados ao desenvolvimento de software (produzir uma especificação e refiná-la em direção à implementação, o que também é chamado de *Forward Engineering*), existem também ambientes de Engenharia Reversa. Estes ambientes produzem uma especificação a partir de um código existente. São exemplos destes ambientes Luthier (para Linguagem Smalltalk, procede análise dinâmica de aplicações) [CAM 97] e ADvance (para Linguagem Smalltalk, procede análise estática de aplicações) [ICC 97]. Nos dois exemplos é produzida uma modelagem gráfica no nível de abstração da implementação (classes com métodos e atributos, mensagens para chamada de métodos etc). O uso deste tipo de mecanismo pode ser útil para levantar a descrição de aplicações existentes para a geração ou alteração de frameworks.

3.6.2 Uso de Ambientes Existentes no Desenvolvimento de Frameworks

Durante o desenvolvimento do framework FraG, apresentado no capítulo 2, que antecedeu o desenvolvimento do ambiente SEA, foi usado o ambiente Rose para o registro da especificação de projeto. Como vantagem do uso, pode-se citar a conveniência das técnicas de modelagem como mecanismos de condução do processo de criação: o relacionamento das situações de processamento através de diagrama de casos de uso, o refinamento dos casos de uso através de diagramas de seqüência - quando foram definidos os principais métodos de cada classe. Como desvantagem, observou-se que a desvinculação entre a especificação e o código demandava um esforço

---

http://www.qucis.queensu.ca/Software-Engineering/case.html
http://www.jyu.fi/~kelly/meta/caselist.html

considerável para manter o registro do projeto atualizado (diagramas de classes, diagramas de seqüência etc), uma tarefa adicional, com o mérito de manter documentação do projeto, porém que não agilizava o processo de desenvolvimento, muito pelo contrário. Além disto, a especificação obtida não era capaz de descrever adequadamento o framework, em função das limitações da notação disponível para representar informações sobre frameworks como a impossibilidade de destacar as partes flexíveis do framework e a impossibilidade de vincular a especificação de uma aplicação à especificação do framework, sob o qual foi desenvolvida.

No levantamento das características de ambientes existentes (como também no levantamento feito por Oliveira [OLI 96]) não foi localizado nenhum ambiente que apresentasse o conjunto características desejáveis. Esta porém não é a questão central, pois mesmo havendo, o uso de um ambiente voltado ao desenvolvimento de aplicações se mostra inadequado ao desenvolvimento de frameworks, pois neste contexto necessita-se, por exemplo de:

- tratamento conjunto de especificações distintas: de aplicações desenvolvidas a partir de frameworks e dos frameworks em que estas se baseiam;
- capacidade de especificar a flexibilidade de um framework;
- capacidade de verificar se uma aplicação baseada em um framework obedece às restrições impostas no projeto deste framework.

A dificuldade de usar um ambiente convencional para frameworks reside na sua inflexibilidade notacional e funcional, que impede sua adaptação a estas necessidades.

Um ambiente com o conjunto de características descrito, adaptado às características dos frameworks, que suporte o processo de desenvolvimento e que seja capaz de gerar código (dispensando o esforço para produzir código, bem como para manter consistência entre projeto e código), se coloca como um mecanismo importante para facilitar o desenvolvimento de frameworks. Um requisito que precede o desenvolvimento de um ambiente assim é a definição de uma notação capaz de representar adequadamente todas as características dos frameworks (como discutido na seção anterior, estender a notação de UML é uma alternativa adequada para isto). Uma alternativa possível para a obtenção de um ambiente de desenvolvimento de frameworks é adaptar um ambiente flexível (meta CASE) para atender às necessidades desta abordagem. Uma outra alternativa é desenvolver um novo ambiente, opção adotada no trabalho de pesquisa ora descrito, em função da perspectiva de obtenção de uma plataforma de suporte a experimentações acadêmicas em torno de abordagens de desenvolvimento de software.

4 Compreensão e Uso de Frameworks Orientados a Objetos

Utilizar um framework consiste em produzir aplicações a partir deste framework. A motivação para produzir aplicações a partir de frameworks é a perspectiva de aumento de produtividade e qualidade, em função da reutilização promovida. Para produzir uma aplicação sob um framework deve-se estender sua estrutura, de modo a cumprir os requisitos da aplicação a desenvolver.

Um obstáculo à adoção da abordagem para o desenvolvimento de software é a dificuldade para aprender a desenvolver aplicações sob frameworks. A questão é que se um desenvolvedor de aplicações necessitar empreender um esforço maior para aprender a usar um framework que para desenvolver uma aplicação do início, a utilidade do framework torna-se questionável.

Torna-se um imperativo do processo de desenvolvimento de frameworks a produção de subsídios para minimizar o esforço requerido para compreender como usar os frameworks produzidos. Johnson classifica diferentes níveis de compreensão de um framework: saber que tipos de aplicações podem ser desenvolvidas a partir do framework, ser capaz de desenvolver aplicações elementares sob o framework e conhecer em detalhes o projeto do framework. Para capacitar ao desenvolvimento de aplicações elementares recomenda o desenvolvimento de documentação no estilo *cookbook*, que não entre em detalhes de projeto [JOH 94]. A utilização de um framework a partir de instruções passo a passo mostra-se limitada quanto à gama de aplicações que podem ser desenvolvidas: se um requisito de uma aplicação não é tratado no conjunto de instruções, o mecanismo não consegue auxiliar o desenvolvimento. Para suprir esta deficiência, usuários de frameworks, invariavelmente, são forçados a buscar conhecimento acerca do projeto do framework. Assim, é importante que a descrição de um framework fornecida a um usuário o habilite a produzir aplicações com o menor esforço possível, mas que também forneça meios de aprofundar a compreensão do projeto do framework, a fim de habilitá-lo a produzir aplicações mais complexas, sem que lhe seja exigido um esforço excessivo - o fator de comparação é que o esforço dispendido para produzir uma aplicação sob um framework não deve ser superior ao necessário para produzir uma aplicação equivalente, sem o framework.

A seguir são analisadas abordagens propostas para auxiliar o processo de aprendizagem a respeito de um framework. Também é feito um levantamento do suporte ferramental disponível, para o auxilio à utilização de frameworks.

4.1 Indivíduos Envolvidos com o Desenvolvimento e o Uso de Frameworks

O desenvolvimento tradicional de aplicações envolve dois tipos de indivíduo: desenvolvedor de aplicação e usuário de aplicação (nos dois casos isto pode corresponder a grupos de indivíduos, com diferentes funções). Desenvolvedores devem levantar os requisitos de uma aplicação, desenvolvê-la (o que inclui a documentação que ensina a usar a aplicação, como manuais de usuário) e entregá-la aos usuários. Usuários interagem com uma aplicação apenas através de sua interface. A figura 4.1 apresenta os indivíduos envolvidos neste caso.

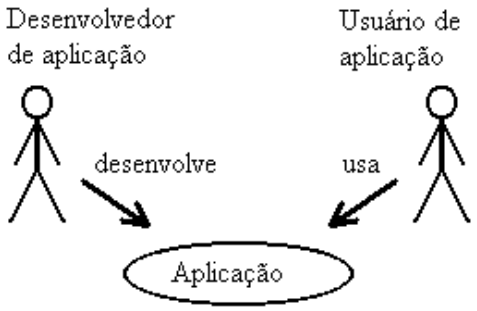

FIGURA 4.1 - Elementos do desenvolvimento tradicional de aplicações

O desenvolvimento de frameworks introduz outro indivíduo, além de desenvolvedor e usuário de aplicação: o desenvolvedor de framework. No contexto dos frameworks, o papel do usuário de aplicação é o mesmo descrito acima. O papel do desenvolvedor de aplicações difere do caso anterior pela inserção do framework no processo de desenvolvimento de aplicações. Com isto, o desenvolvedor de aplicações é um usuário de um framework, que deve estender e adaptar a estrutura deste framework para a produção de aplicações. Ele tem as mesmas funções do caso anterior: obter os requisitos da aplicação, desenvolvê-la usando o framework, (o que em geral, não dispensa completamente do desenvolvedor de aplicações a necessidade de produzir código) e desenvolver a documentação da aplicação. O novo papel criado no contexto dos frameworks, o desenvolvedor de framework, tem a responsabilidade de produzir frameworks e algum modo de ensinar como usá-los para produzir aplicações. A figura 4.2 apresenta os indivíduos envolvidos neste último caso.

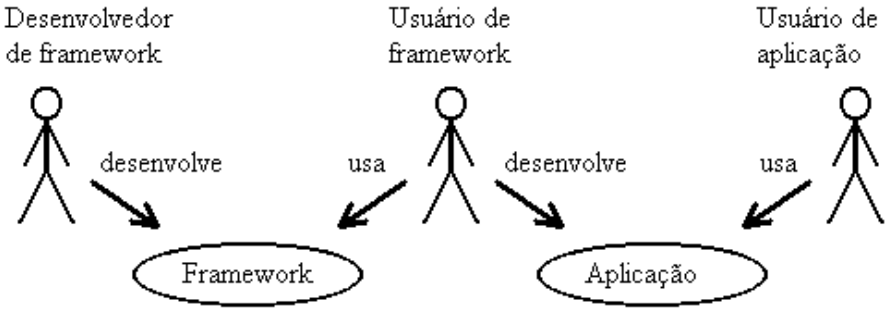

FIGURA 4.2 - Elementos do desenvolvimento de aplicações baseado em frameworks

4.2 As Questões-Chave para Usar um Framework

Usuários de frameworks desenvolvidos segundo a abordagem caixa-preta precisam apenas conhecer que objetos estão disponíveis e as regras para combiná-los. No caso dos frameworks caixa-branca e caixa-cinza, necessitam saber quais classes devem ser geradas, quais classes podem ser geradas, e precisam diferenciar os dois casos. Também necessitam conhecer quais as responsabilidades e as funcionalidades das classes a serem geradas. Assim, no caso geral, as três questões-chave a seguir devem ser respondidas[35]:

---

35 Na abordagem caixa-preta os usuários de um framework sabem previamente que não serão geradas classes e nem métodos.

- **Questão 1 - *Quais classes?***: as classes concretas de uma aplicação podem ser criadas ou reutilizadas do framework. Então, que classes devem ser desenvolvidas pelo usuário e que classes concretas do framework podem ser reutilizadas ?

- **Questão 2 - *Quais métodos?***: Um método de uma classe abstrata pode ser classificado como [JOH 91]:
  - abstrato: um método que tem apenas a sua assinatura definida na classe abstrata (o corpo deve ser definido nas subclasses);
  - *template*: é definido em termos de outros métodos (métodos *hook*);
  - base: é completamente definido.

Na execução de um método *template* ocorre a chamada de pelo menos um método *hook*, que pode ser um método abstrato, base ou *template* [PRE 94]. Ao produzir uma aplicação:

- métodos *abstratos* precisam ser definidos;
- métodos *template* fornecem uma estrutura de algoritmo definida, mas permitem flexibilidade através dos métodos *hook* chamados. A avaliação da necessidade ou conveniência de produção de métodos deve ser estendida para os métodos *hook*.
- métodos *base* de acordo com o projeto do framework, não precisam ser alterados (mas é possível sobrepô-los).

Assim, ao gerar uma classe concreta para uma aplicação (considerando o caso de subclasse de classe abstrata do framework), que métodos devem ser definidos e que métodos herdados podem ser reutilizados ?

- **Questão 3 - *O que os métodos fazem?***: os métodos cuja estrutura deve ser definida pelo usuário do framework, produzem o comportamento específico da aplicação sob desenvolvimento. Assim, definidas as classes e os respectivos métodos a desenvolver, quais as responsabilidades destes métodos, em que situações de processamento eles atuam e como eles implementam a cooperação entre diferentes objetos?

4.3 Aprendizado de Como usar Frameworks a partir de Documentação

A resposta para as questões anteriores está contida no projeto de um framework, pois, a definição das classes e métodos que devem ser desenvolvidos (bem como as classes e métodos cuja criação é opcional) é resultado de decisões de projeto, tomadas no desenvolvimento do framework. Assim, aprender a usar um framework corresponde a buscar as respostas das questões-chave na estrutura do framework, usando como fontes de informação, documentação, código fonte[36] ou ambos.

Aprender a usar um framework vai além de adquirir a capacidade de produzir aplicações. Também implica em conhecer que recursos são disponibilizados, para evitar esforços desnecessários de desenvolvimento. Um usuário que utiliza um framework sem conhecê-lo razoavelmente, pode dispender esforço desenvolvendo o que poderia ser reutilizado. Ao submeter um grupo de usuários ao desenvolvimento de editores gráficos utilizando o framework HotDraw, Campo verificou no trabalho de parte dos usuários a criação desnecessária de classes, bem como a alta proporção de classes criadas que não

---

36 Pode-se dispor de documentação e código fonte do framework, e de aplicações produzidas a partir deste framework.

eram subclasses de classes do framework, o que foi ocasionado pelo desconhecimento da estrutura do framework usado [CAM 97].

Pode-se considerar que existem dois tipos de descrição de frameworks: descrição do projeto e descrição de como usar. A seguir são analisadas diferentes propostas de documentação baseadas nestes enfoques.

4.3.1 Informação de Projeto de Frameworks

As abordagens de descrição de estruturas de projeto de frameworks são originadas de metodologias de desenvolvimento de frameworks, que não usam técnicas de modelagem. Assim, a documentação de projeto é normalmente baseada em descrição textual, referências ao código fonte e uso de dispositivos que descrevem aspectos isolados do projeto de framework - quase sempre para destacar partes flexíveis da estrutura, como por exemplo, o destaque de padrões de projeto que o framework apresenta.

Pree propôs o destaque dos metapadrões usados em um framework, através de um diagrama semelhante a um diagrama de classes das metodologias OOAD, que incluem as classes desenvolvidas e uma representação icônica do metapadrão usado, destacando os métodos *template* e *hook* [PRE 94]. A figura 4.3, apresenta o padrão de projeto *Publisher/Subscriber* (*Observer*, segundo a nomenclatura de Gamma [GAM 94]) descrito através desta notação. A classe *Publisher* é a classe que possui o método *template Notify()*, e *Subscriber*, a classe que possui o método *hook Update()*, invocado pelo método *template Notify()*. Considerando que diferentes implementações de *Update()* podem ser produzidas nas subclasses de *Subscriber*, e que instâncias destas subclasses podem ser conectadas dinamicamente a instâncias de *Publisher*, observa-se que diferentes comportamentos podem ser observados a partir da invocação de *Notify()*. A representação registra, portanto, a flexibilidade da estrutura.

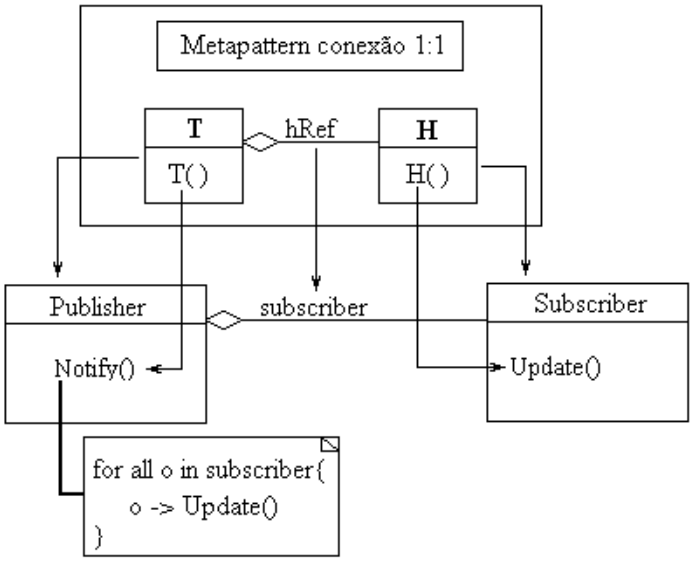

FIGURA 4.3 - Padrão de projeto Publisher/Subscriber descrito através de notação de metapadrões

A vantagem desta abordagem é a capacidade de destacar os pontos da estrutura de projeto de um framework que foram mantidos flexíveis - onde deve começar a busca das respostas das questões-chave. Esta notação no entanto, não descreve toda a estrutura do framework e nem ensina a usá-lo para gerar aplicações.

Contratos [HEL 90] descrevem a cooperação entre grupos de objetos através de uma linguagem formal, que representa:

- os participantes - instâncias ou conjuntos de instâncias que tomam parte no contrato;
- as obrigações contratuais - responsabilidades dos participantes em sua interação.

A figura 4.4 ilustra a descrição do padrão de projeto *Publisher/Subscriber* baseada em contratos [PRE 94]. Verifica-se a definição dos dois tipos de participante: *Publisher* e *Subscriber*. *Publisher* possui os métodos *Notify()*, que invoca *Update()* de todo assinante (*subscriber*) referenciado, *AttachSubscriber(s: Subscriber)*, para incluir assinantes no conjunto dos referenciados (*subscribers*) e *DettachSubscriber(s: Subscriber)*, para excluí-los. *Subscriber* possui o método *Update()*. O estabelecimento do contrato ocorre pela invocação do método *AttachSubscriber(s: Subscriber)*, a partir da qual um assinante passa a ser referenciado por um publicador (*publisher*).

```
contract PublisherSubscriber
        Publisher supports [
                Notify() =>
                        < || s: s ∈ Subscribers: s -> Update() >
                AttachSubscriber(s: Subscriber)     =>
                        {s ∈ Subscribers}
                DettachSubscriber(s: Subscriber)    =>
                        {s ∉ Subscribers}
        ]
        Subscribers: Set(Subscriber) where each Subscriber supports [
                Update()            =>
        ]
        contract invariant
                -
        contract establisment
                < || s: s ∈ Subscribers:
                        <Publisher -> AttachSubscriber(s) >          >
end contract
```

FIGURA 4.4 - Padrão de projeto Publisher/Subscriber descrito através de notação de contratos

Contratos descrevem aspectos de cooperação entre objetos contidos no projeto de um framework, podendo ser usados, como ocorre na abordagem de Pree, para destacar as partes flexíveis da estrutura de um framework. Modelam a interação entre objetos de uma maneira formal, e, como contratos podem ser reusados como base ou parte de outros contratos, permitem descrever comportamentos complexos como composições de comportamentos simples. A principal desvantagem dos contratos é que produzem uma descrição extensa e de baixo nível de abstração. Isto torna difícil entender o projeto de um framework a partir desta abordagem. Além disto, o mecanismo não é capaz de descrever todos os detalhes de um projeto.

4.3.2 Informação de "Como Usar" um Framework

Documentação direcionada a ensinar como usar um framework normalmente descreve os passos a serem seguidos por um usuário para gerar aplicações. Este tipo de documentação dá pouca ênfase a aspectos de projeto - seu principal objetivo é descrever da forma mais breve possível, como produzir aplicações. Em geral, apresentam formato textual e incluem alguma outra forma de descrição, como exemplos usando código fonte.

Cookbooks, como utilizado para a descrição do framework MVC [PAR 94], são conjuntos de "receitas" textuais que estabelecem como usar um framework para resolver problemas no desenvolvimento de aplicações. Sua principal vantagem é a capacidade de responder às questões-chave diretamente e minimizar o tempo gasto para produzir aplicações. O principal problema da abordagem é que cobre uma faixa limitada de tipos de aplicação: se as necessidades de um usuário do framework não se ajustam ao conjunto de problemas tratados em um *cookbook*, isto é, se um usuário necessita produzir uma aplicação não-prevista, o auxílio do *cookbook* pode ser ínfimo. Outro problema com a abordagem, é que quase sempre o processo de entendimento de um framework envolve atividades de baixo nível: uma vez respondidas as questões chaves (o que fazer), para aprender como fazer, um usuário freqüentemente necessita buscar outras fontes de informação.

Padrões de documentação propostos por Johnson e usados para descrever como usar o framework HotDraw [JOH 92], constituem uma alternativa de formato aos *cookbooks*. Os padrões de documentação são receitas textuais estruturadas que descrevem como fazer algo usando recursos de um framework, e que contém referências que apontam outros padrões de documentação - semelhante aos *links* de um hiperdocumento. Sua principal vantagem em relação ao *cookbooks* convencionais é que a ligação entre receitas torna mais fácil a navegação através da documentação, a partir de um caminho definido de acordo com as necessidades do usuário do framework. As desvantagens desta abordagem são as mesmas dos *cookbooks*. Em particular, analisando os padrões de documentação que descrevem o framework HotDraw, fica clara a necessidade de interpretação de código para transpor a lacuna entre saber o que fazer e saber como fazer[37].

4.4 Aprendizado de Como Usar Frameworks a partir de Código Fonte

O modo mais elementar de auxílio ao desenvolvimento de aplicações baseadas em frameworks é a disponibilização de código fonte aos usuários do framework. Isto pode incluir o código do framework tratado - de todo o framework ou apenas de parte dele - e o código de aplicações desenvolvidas sob o framework. Entender o projeto de um framework dispondo apenas de código é o único modo possível quando não há documentação disponível. Nenhuma das três questões-chave anteriormente mencionadas é respondida, sem algum esforço. É uma atividade de baixo nível e, de um modo geral, é a maneira mais árdua de aprender a usar um framework.

Compreender o projeto de um framework a partir de seu código é uma atividade de Engenharia Reversa, uma área de pesquisa voltada ao exame e entendimento de artefatos de software existentes. O produto da atividade de Engenharia Reversa é registrado em um formato utilizável, isto é, resulta em documentação de software. Esta atividade cobre as deficiências da documentação de artefatos de software existentes, seja porque seu desenvolvimento não incluiu a produção de descrição de forma adequada, ou por desatualização da documentação existente em função de manutenções, executadas sem o cuidado de alterar a documentação original [WES 93].

---

37 O framework HotDraw pode ser obtido em ftp://st.cs.uiuc.edu, juntamente com publicações que tratam este framework - inclusive o artigo de Johnson com a descrição do framework usando padrões de documentação [JOH 92]. Um usuário que busque aprender a usar HotDraw, como foi feito no trabalho de pesquisa ora descrito, necessariamente é forçado a interpretar seu código, pela insuficiência de informações de projeto na documentação disponibilizada.

A Engenharia Reversa (em termos de software de computadores) é definida como:

> *"O processo de análise de um software para identificar os elementos do sistema e seus interrelacionamentos, e para criar representações do sistema em outra forma ou em um nível de abstração mais elevado." [CHI 90].*

A Engenharia Reversa engloba três atividades distintas: análise de código, redocumentação e recuperação de projeto. A análise de código é uma atividade de baixo nível de abstração que visa buscar a compreensão do código de uma aplicação. Produz subsídios para o processo de redocumentação.

A redocumentação está em um nível de abstração imediatamente superior à análise de código e visa organizar as informações obtidas através desta atividade. Produz a descrição da estrutura da aplicação - constituição e organização dos módulos de implementação. Pode gerar documentação no código fonte, documentação da organização física da aplicação (descrição dos módulos de implementação) e documentação para utilização da aplicação (como manuais de utilização).

A recuperação de projeto, por sua vez é a atividade de nível de abstração mais elevado, que produz a especificação de projeto da aplicação, isto é, produz a descrição da estrutura lógica da aplicação. Ao contrário da redocumentação, a recuperação de projeto produz uma especificação usando diferentes níveis de abstração.

Observa-se que as metodologias de Engenharia Reversa dependem principalmente de técnicas baseadas em heurística [PEN 95] [WES 93] [LEI 92] [JAC 91]. O entendimento de sistemas não-orientados a objetos é feito a partir da busca de entendimento dos procedimentos e de sua integração (o que inclui o entendimento das estruturas de dados manipuladas). No caso de aplicações orientadas a objetos, o encapsulamento de métodos e atributos garante uma melhor organização das informações da aplicação, porém, a interpretação de métodos e a cooperação entre classes consiste em uma atividade de interpretação de código, semelhante ao que é feito em sistemas não-orientados a objetos.

A atividade de análise de código é dependente da linguagem de programação usada na aplicação. Assim, a recuperação de projeto de uma aplicação não-documentada é altamente dependente da atuação de especialistas na linguagem, para a etapa de análise de código. A análise de código admite ferramentas como analisadores de referências cruzadas (*cross-analyser*), que fazem apenas uma parte do trabalho de análise. Um analisador de referências cruzadas é um programa relativamente simples de produzir, a partir de geradores de analisadores léxicos e sintáticos (como os programas Lex e Yacc, por exemplo). O mecanismo de registro de referências pode ser embutido em um programa interpretador, na forma de ações semânticas que, em um processo de inspeção do código fonte, registram todas as chamadas de procedimento existentes no código de cada procedimento. A análise de referência cruzada ocorre por exemplo em metodologias propostas por Leite [LEI 92] e por Penteado [PEN 95].

O conjunto de informações sobre um framework obtido a partir de um procedimento de Engenharia Reversa é o ponto de partida para a obtenção das respostas às questões-chave. A complexidade envolvida e o esforço demandado para a compreensão podem tornar inviável o uso de um framework. Assim, para garantir a aplicabilidade, é necessário que ao longo do desenvolvimento de um framework se tenha a preocupação de produzir documentação, bem como de atualizá-la nas manutenções.

4.5 Ferramentas de Auxílio ao Uso de Frameworks

O uso de frameworks para a geração de aplicações pode ser suportado por ferramentas de diferentes tipos. Um primeiro tipo corresponde às ferramentas de análise, que automatizam etapas de um procedimento de Engenharia Reversa e que podem ser usadas para ajudar a entender um framework, a partir da interpretação do seu código e do código de aplicações geradas sob ele. Um segundo grupo corresponde às ferramentas desenvolvidas para auxiliar a geração de aplicações (ou de parte de aplicações) construídas em função de características de frameworks específicos. Existem também ferramentas baseadas em *cookbooks*. Estas ferramentas disponibilizam o *cookbook* de um framework e podem dispor de outras funcionalidades. Um último grupo corresponde a ambientes visuais de suporte ao desenvolvimento de aplicações sob frameworks caixa-preta. A seguir são descritos exemplos destas ferramentas.

4.5.1 Ferramentas de Análise

No capítulo anterior foi citada a utilidade de ferramentas de Engenharia Reversa no desenvolvimento de frameworks. Naquela situação a recuperação de projeto de aplicações de um domínio se voltava a fornecer informações do domínio tratado, para a definição da estrutura de classes de um framework, ou para sua manutenção. No apoio ao uso de frameworks, as ferramentas de análise ajudam a entender o projeto de um framework, na busca das respostas das questões-chave, isto é, conhecer o que fazer para produzir aplicações.

Existem diversos ambientes e ferramentas capazes de produzir descrições de artefatos de software, a partir de seu código fonte. Um exemplo é o ambiente ADvance [ICC 97]. Este ambiente procede análise da estrutura de código Smalltalk e produz descrições através de técnicas de modelagem gráficas de metodologias OOAD. A análise procedida considera o código fonte, o que caracteriza análise estática.

Mecanismos de análise dinâmica procedem a análise de artefatos de software em tempo de execução. Meta-Explorer é uma ferramenta de análise dinâmica de aplicações desenvolvidas sob frameworks, voltada a compreensão de frameworks [CAM 97]. Esta ferramenta foi desenvolvida sob o framework Luthier. Meta-Explorer produz descrição do comportamento dinâmico de uma aplicação através do uso de meta-objetos, que observam o fluxo de mensagens entre os objetos de uma aplicação.

Estes dois exemplos de ferramenta produzem uma modelagem no nível de abstração da implementação, subsídio para o entendimento de um framework a partir de seu código.

4.5.2 Ferramentas para Frameworks Específicos

Ferramentas desenvolvidas para frameworks específicos constituem um outro tipo de ferramenta para assistir o desenvolvimento de aplicações baseadas em frameworks. A ferramenta de *canvas* de VisualWorks [PAR 94] e o construtor de ferramentas de HotDraw [BRA 95] constituem dois exemplos destas ferramentas. A ferramenta de *canvas* de VisualWorks permite a construção de interfaces gráficas através da seleção de elementos visuais em um *palette* e sua alocação em um *canvas*. O produto resultante é uma descrição de interface em forma declarativa, a ser interpretada por um objeto construtor de interfaces, em uma aplicação baseada em MVC. O construtor de ferramentas de HotDraw é um *browser* que facilita a tarefa de associação de *figures*, *commands* e *readers* (instâncias de classes do framework) para definir a estrutura de

ferramentas em editores gráficos. Estas ferramentas auxiliam apenas o uso de seus respectivos frameworks.

4.5.3 Ferramentas Baseadas em *Cookbook*

Os *cookbooks* on-line dos frameworks MVC [PAR 94] e ET++ [PRE 94] são exemplos de ferramentas baseadas na abordagem *cookbook*. O *cookbook* de MVC apresenta uma seqüência de receitas que podem ser selecionadas em um índice ou visualizadas na ordem de apresentação. Inclui descrição textual e pequenos exemplos em Smalltalk. O *cookbook* do framework ET++ é baseado na abordagem de hiperdocumentos, o que torna possível definir caminhos de navegação de acordo com as necessidades do usuário. Inclui partes da especificação de projeto usando a notação de metapadrões, além de descrição textual e exemplos em C++. Estas ferramentas apresentam a vantagem da abordagem *cookbook*: indicação direta da seqüência de passos a seguir, para solucionar um problema de projeto. Em contrapartida, apresentam a desvantagem dos *cookbooks*: deixam os usuários sem auxílio quando suas necessidades não se ajustam às situações previstas.

Meusel produziu um *cookbook* para o framework HotDraw [MEU 97] combinando as abordagens padrão de documentação e hiperdocumento. Além dos padrões de documentação, a ferramenta inclui tutoriais (descrição passo a passo do desenvolvimento de exemplos de aplicação) e padrões de projeto, que descrevem aspectos específicos do projeto do framework. Em relação a outros *cookbooks*, esta ferramenta contribui principalmente com a associação da noção de hiperdocumento aos padrões de documentação de Johnson (a ferramenta do framework ET++ também é baseada em hiperdocumento).

A abordagem *cookbook* ativo proposta por Pree e implementada em um protótipo, vai além da capacidade de descrição das ferramentas baseadas em *cookbook*, sendo apto a produzir partes da implementação da aplicação, automaticamente [PRE 95]. De acordo com esta abordagem uma ferramenta de *cookbook* além de ser usada para aprender a usar um framework, também pode auxiliar no desenvolvimento do código da aplicação.

As ferramentas acima descritas disponibilizam aos usuários dois tipos de informação: o conteúdo do *cookbook* e o código fonte referenciado (de framework, de aplicações ou ambos). Dúvidas do usuário não solucionadas a partir do conteúdo de um *cookbook* devem ser respondidas através de interpretação de código - atividade não apoiada por estas ferramentas. Nautilus é uma ferramenta para a produção de *cookbooks* ativos que descrevam os passos para produzir aplicações usando frameworks, que permite incluir ações automáticas de criação do código da aplicação (em Smalltalk) [ZAN 98]. Além disto, Nautilus está acoplado à ferramenta de análise dinâmica de aplicações, Meta-Explorer [CAM 97]. A integração de Nautilus com Meta-Explorer contribui no processo de aprendizado de um framework, fornecendo auxílio à interpretação de código, como complemento às instruções do *cookbook*.

4.5.4 Ambientes de Desenvolvimento de Aplicações Baseados em Linguagens Visuais

Linguagens visuais constituem uma abordagem utilizável para o desenvolvimento de aplicações a partir de frameworks caixa-preta. Durham produziu um protótipo de ferramenta, inicialmente desenvolvendo um framework caixa-branca. A definição de

classes concretas que poderiam ser combinadas, levou à obtenção de um framework caixa-preta. A etapa final foi o desenvolvimento de uma linguagem visual para a definição de aplicações e de uma ferramenta para a utilização dessa linguagem [DUR 96]. A disponibilização de uma linguagem visual facilita o uso do framework. Por outro lado, esta abordagem apresenta duas desvantagens: primeiro, a faixa limitada de aplicações que se pode produzir a partir de um framework caixa-preta; segundo, a tarefa de desenvolvimento de um framework caixa-preta e de uma interface para a produção de aplicações baseada em linguagem visual requer muito mais esforço do que o desenvolvimento de framework caixa-branca ou caixa-cinza, para o mesmo domínio.

A ferramenta de *canvas* de VisualWorks [PAR 94] também utiliza uma abordagem de programação baseada em linguagem visual, porém suporta apenas a produção de parte de uma aplicação.

4.5.5 Análise das Ferramentas

As ferramentas descritas podem ser divididas em três grupos, segundo o seu princípio construtivo: ferramentas de análise, ferramentas para frameworks específicos e ferramentas baseadas em um princípio geral. Ferramentas de análise são úteis para apoiar a atividade de interpretação de código (do framework ou de aplicações) quando há a necessidade de fazê-lo. Ferramentas para frameworks específicos, o que inclui a ferramenta de *canvas* de VisualWorks e o construtor de ferramentas de HotDraw, ilustram que alguns frameworks podem usar ferramentas específicas, que não são aplicáveis para outros frameworks.

O último grupo inclui ferramentas baseadas em *cookbooks* e o framework de Durham. Este caso ilustra a abordagem de desenvolvimento mais amigável a usuários: frameworks caixa-preta com linguagem visual para geração de aplicações.

O princípio construtivo das ferramentas baseadas em *cookbooks* pode ser usado para produzir ferramentas equivalentes para diferentes frameworks. As ferramentas de *cookbook* mencionadas têm como características comuns, que devem ser construídas por alguém que conheça profundamente o projeto do framework tratado, que descrevem os passos para o desenvolvimento de aplicações baseadas em frameworks e que suas limitações devem ser resolvidas através de interpretação de código. À exceção de Nautilus, nenhuma das outras ferramentas suporta interpretação de código. Outra característica comum das ferramentas de *cookbook* é que elas tratam apenas dois tipos de informação: o conteúdo do *cookbook* (majoritariamente composto de descrição textual, incluindo em alguns casos exemplos usando código fonte e diagramas para descrever aspectos específicos de projeto) e código fonte.

4.6 Análise das Abordagens de Suporte ao Uso de Frameworks

A questão do uso de frameworks está ligada à necessidade de conhecer a resposta às questões-chave, isto é, conhecer o que precisa ser feito para produzir aplicações. Existem basicamente três abordagens de informar isto aos usuários de frameworks: disponibilização de projeto, de código ou de instruções de como usar framework (os *cookbooks*).

A disponibilização de projeto é dificultada pela não-adoção de mecanismos de descrição de alto nível, característica comum nas metodologias de desenvolvimento de frameworks. De fato, as abordagens de descrição ora propostas disponibilizam detalhes

de projeto de partes isoladas da estrutura de um framework, com a capacidade de destacar aspectos úteis para a produção de aplicações, mas não de descrever toda a estrutura de um framework. Atuam como complementos a outros mecanismos de descrição.

A abordagem de *cookbooks* apresenta a vantagem de orientar os passos para a produção de aplicações de uma forma direta, porém *cookbooks* são limitados em sua capacidade de orientar a produção de aplicações. Outro problema é que são descrições textuais, informais e, portanto, não-validáveis. Além disto são produzidos de maneira artesanal, baseados nos conhecimentos de especialistas dos frameworks descritos, sem contar com mecanismos que orientem sua elaboração ou avaliem sua qualidade.

A respeito da descrição de frameworks, Johnson, autor da metodologia Projeto Dirigido por Exemplo [JOH 93], afirma que "frameworks eliminam a necessidade de uma nova notação de projeto através do uso de uma linguagem de programação orientada a objetos como notação de projeto" [JOH 97]. Frameworks, em geral, são distribuídos com manuais, que contém basicamente descrição textual e outros mecanismos de descrição como os aqui mencionados. Como este tipo de descrição invariavelmente deixa lacunas de informação, usuários são obrigados a proceder interpretação de código. A conseqüência da adoção deste tipo de procedimento é que os usuários de frameworks assim concebidos, invariavelmente têm a interpretação de código como o único recurso para suprir deficiências de descrição.

Neste panorama em que a descrição de frameworks é parcial, torna-se necessária a disponibilização de código dos frameworks (todo o código ou apenas parte) e de aplicações baseadas nestes. A interpretação de código pode ser suportada por mecanismos de análise, mas ainda assim é uma tarefa árdua e, em função do esforço requerido, pode desestimular o uso de um framework.

A ação de ferramentas no uso de frameworks é basicamente de apoio às abordagens descritas: *cookbooks* e interpretação de código. A exceção são as ferramentas de frameworks específicos, que são construídas em função de requisitos específicos, e os ambientes de programação visual, que constituem a abordagem mais amigável para a produção de aplicações, porém, com as limitações características de um framework caixa-preta e a custa de esforço elevado. Estas limitações impedem que este tipo de mecanismo possa ser considerado como o ideal para os frameworks em geral.

No conjunto de abordagens descrito, pela sua capacidade de tornar usuários aptos ao uso de um framework mais rapidamente e com menor esforço que outras abordagens, os *cookbooks* caracterizam a forma de apoio ao uso de frameworks mais adequada, principalmente se contarem com suporte ferramental capaz de automatizar etapas do processo de construção de aplicações, como é o caso dos *cookbooks* ativos. Esta abordagem demanda mecanismos que orientem a elaboração de *cookbooks*, que avaliem sua qualidade e que facilitem a definição das ações dos *cookbooks* na geração de aplicações. Devido à impossibilidade de prever todos os requisitos de aplicações a serem geradas por um framework, os *cookbooks* tem uma capacidade limitada de orientação. Com isto, usuários de *cookbooks* normalmente necessitam de outras fontes de informação. Verifica-se que a regra geral de não-adoção de mecanismos de descrição de alto nível para a especificação de frameworks traz como conseqüência a necessidade de interpretação de código para entender um framework. A adoção de mecanismos de descrição de alto nível capazes de descrever completamente um framework, como discutido no capítulo anterior, pode constituir uma alternativa à interpretação de código, com a vantagem de demandar menor esforço para a compreensão de um framework.

5 Desenvolvimento Orientado a Componentes

A meta de construir aplicações a partir da composição de artefatos de software tem mais de três décadas. Em 1968 durante a "crise do software", na Conferência de Engenharia de Software da OTAN, McIlroy fala da necessidade da indústria de software produzir famílias de componentes reusáveis. Segundo seu ponto de vista, desenvolvedores de software deveriam poder escolher componentes ajustados às suas necessidades específicas e estes componentes seriam usados em seus sistemas como artefatos caixa-preta [MCI 68].

Em 1976 DeRemer propõe um paradigma de desenvolvimento de software em que um sistema é desenvolvido como um conjunto de módulos produzidos separadamente e posteriormente interligados [DER 76]. Este paradigma gerou trabalhos de pesquisa, como o Ambiente ADES, com uma linguagem específica para a produção de módulos e outra para compor aplicações através da interligação de módulos [FRA 88]. No entanto, a dependência do uso de linguagens e plataformas específicas, como no caso do ambiente ADES, limitou a aplicabilidade desta abordagem.

A abordagem de desenvolvimento orientado a componentes, semelhante à proposta de DeRemer, determina que uma aplicação seja constituída a partir de um conjunto de módulos (componentes) interligados. No evento WCOP 96 definiu-se componente como "uma unidade de composição com interfaces contratualmente especificadas e dependências de contexto explícitas. Componentes podem ser duplicados e estar sujeitos a composições com terceiros" [SZY 96b]. Esta visão foi refinada no WCOP 97: "o que torna alguma coisa um componente não é uma aplicação específica e nem uma tecnologia de implementação específica. Assim, qualquer dispositivo de software pode ser considerado um componente, desde que possua uma interface definida. Esta interface deve ser uma coleção de pontos de acesso a serviços, cada um com uma semântica estabelecida" [SZY 97].

Krajnc vê o desenvolvimento orientado a componentes como uma evolução natural da abordagem de orientação a objetos, em que um componente corresponde a um conjunto de classes interrelacionadas, com visibilidade externa limitada [KRA 97].

Kozaczynski classifica a disponibilidade de mecanismos de interconexão de componentes como um dos principais estímulos à abordagem de desenvolvimento baseado em componentes [KOZ 98]. CORBA é um destes mecanismos. Permite interconectar componentes independentemente da linguagem, plataforma de execução e localização física dos componentes [SEE 98].

Obstáculos associados à abordagem de componentes estão relacionados à busca e seleção de componentes para reuso. A dificuldade de localização de componentes está associada à inexistência de padrões de repositório e mecanismos de busca que permitam a potenciais usuários selecionar componentes que supram suas necessidades [MIL 97]. A dificuldade de seleção está associada às deficiências apresentadas pelos mecanismos de descrição de componentes em especificar o que componentes fazem e como interagem.

Geralmente componentes necessitam ser adaptados antes de serem utilizados [BOS 97]. A dificuldade de compatibilizar componentes originalmente incompatíveis constitui outro obstáculo da abordagem de desenvolvimento baseado em componentes.

O presente capítulo trata a abordagem de desenvolvimento orientado a componentes, com ênfase aos aspectos de especificação e adaptação de componentes. A abordagem é avaliada sob a ótica de sua adequabilidade como suporte à definição de arquiteturas de software. A partir da noção de sistemas de software como componentes

interligados, avalia-se como a adoção conjunta das abordagens de frameworks e componentes pode contribuir com o aumento de qualidade e produtividade no desenvolvimento de software.

5.1 Componentes, Interfaces, Canais de Comunicação

Sob a ótica do desenvolvimento orientado a componentes, um artefato de software é composto pela conexão de um conjunto de componentes. O que diferencia um componente de um artefato de software qualquer é que um componente possui uma interface. A interface corresponde a "uma coleção de pontos de acesso a serviços, cada um com uma semântica estabelecida" [SZY 97].

Uma função de uma biblioteca de funções para linguagem C corresponde ao exemplo mais simples de componente. A definição da interface corresponde à assinatura do procedimento chamado. Um programa C comunica-se com este tipo de componente invocando a função por seu nome, passando parâmetros que obedeçam à tipagem definida para a função e atribuindo o retorno (caso haja retorno) a uma variável de tipo compatível. Esta é a forma mais elementar de comunicação com um componente: unidirecional e com apenas um tipo de procedimento.

Um componente pode corresponder, por exemplo, a um tipo abstrato de dados, como um objeto desenvolvido a partir de uma linguagem de programação orientada a objetos. Neste caso, o componente admite invocação de um conjunto de procedimentos. Este caso ainda caracteriza comunicação unidirecional: os serviços do componente são invocados pelo meio externo.

Na visão do desenvolvimento orientado a componentes, comunicação unidirecional é um caso particular, pois a interação de um componente com o meio externo pode ser bidirecional, ou seja, em geral o componente pode fornecer serviços mas também requerê-los.

É possível que um componente precise interagir com mais de um componente simultaneamente. Neste caso a interface deve dispor de mecanismos de acesso bidirecionais e que possibilitem a conexão de um componente a mais de um componente. Cada um destes pontos de acesso é chamado de canal de comunicação. Assim, no caso geral, um componente possui uma interface, composta de um ou mais canais de comunicação, através dos quais o componente se comunica com o meio externo. A figura 5.1 ilustra um artefato de software constituído pela interligação de um conjunto de componentes, através dos canais de comunicação de suas interfaces.

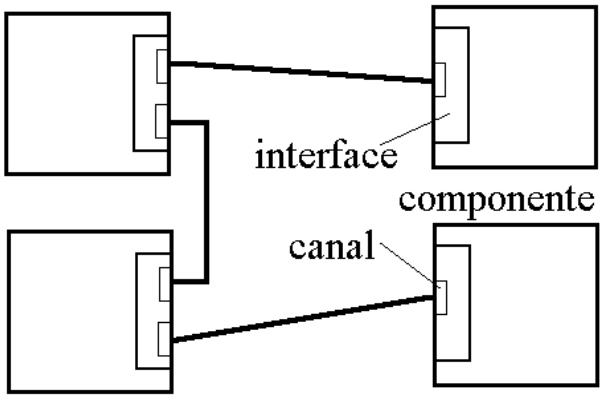

FIGURA 5.1 - Interligação de componentes através de seus canais de comunicação

5.2 Mecanismos de Conexão de Componentes

Uma questão fundamental da abordagem de desenvolvimento baseado em componentes é a maneira de proceder a conexão de componentes. Considerando um ambiente de desenvolvimento em que todos os componentes tenham sido implementados em uma mesma linguagem de programação e possuam a mesma localização física, mecanismos da própria linguagem, como apontadores (de C, Smalltalk etc.) podem ser usados para que um componente mantenha referência de outro, e chamadas de procedimento dão acesso aos serviços dos componentes.

O aumento da atenção ao desenvolvimento orientado a componentes ocorrido nos últimos anos se deve, porém, à disponibilidade de mecanismos de interconexão, como CORBA, que permitem a operação conjunta de componentes independentemente de sua linguagem, plataforma de execução e localização física. As unidades de conexão em CORBA são os objetos CORBA, que possuem uma interface pública definida através de uma linguagem de definição de interfaces (IDL). A disponibilidade de IDL's para a definição de interface para componentes desenvolvidos em diferentes linguagens de programação, como Java, C++ e Smalltalk, possibilita a interação entre componentes desenvolvidos em diferentes linguagens. O Object Request Broker (ORB) implementa o meio de comunicação entre componentes, tornando transparente a localização destes [SEE 98].

Além de CORBA, podem ser citados como padrões "*de fato*" de interconexão SOM, COM e JavaBeans. System Object *Model* (SOM) da IBM, baseado em CORBA e Component Object *Model* (COM) da Microsoft usam IDL's e possibilitam conexão com independência de linguagem, plataforma de execução e localização física dos componentes [PFI 96]. JavaBeans, da Sun Microsystems, em que os componentes são programados em Java, permite independência de plataforma de execução e localização física dos componentes [HED 98]. Os mecanismos de interconexão, apesar de solucionarem a questão da heterogeneidade entre elementos que precisam interagir, provocam queda de desempenho, exigindo para sua adoção, que este aspecto seja avaliado [KRA 97b].

Para que dois componentes possam ser conectados é necessário que suas interfaces sejam compatíveis, isto é, que independentemente de homogeneidade de linguagem em que estão implementados, de localização física e de plataforma de execução, os serviços requeridos por um, estejam disponíveis no outro. A descrição de componentes deve permitir verificar esta compatibilidade, senão é impossível verificar se dois componentes podem ser conectados. Mecanismos de descrição de componentes são tratados a seguir.

5.3 Mecanismos de Descrição de Componentes

A forma usual de descrever um componente consiste na descrição de sua interface[38]. Porém, os mecanismos de descrição de interface existentes, em geral, são pobres para descrever componentes porque produzem apenas uma visão externa incapaz

---

[38] Componentes podem se comunicar através de diferentes mecanismos, como chamada de procedimentos, compartilhamento de memória, *broadcast*, fluxos de dados. No restante do capítulo só há referência a chamada de procedimento, por ser a forma adotada no trabalho realizado - uma situação particular.

de descrever suas funcionalidades e como estes interagem [HON 97]. A descrição de um componente pode abranger três aspectos distintos: estrutural, comportamemtal e funcional. A descrição estrutural corresponde à relação das assinaturas de métodos da interface. A descrição comportamental também se atém à interface e especifica restrições na ordem de invocação de métodos. A descrição funcional vai além da interface e visa descrever o que o componente faz, pois, não é possível saber o que os métodos fazem apenas conhecendo sua assinatura e restrições de ordem de invocação.

Um dos fatores que dificultam o reuso de componentes é a inexistência de padrões de descrição de componentes capazes de abranger todos esses aspectos.

5.3.1 Descrição da Interface de Componentes

A interação de um componente com o meio externo pode ser bidirecional. Assim, uma descrição de interface que apresente o conjunto de métodos fornecidos por um componente não é suficiente para entender como este interage. Ólafsson afirma que além da interface provida, isto é, o conjunto de métodos do componente que podem ser invocados, uma descrição de interface de componente também deve estabelecer a interface requerida, isto é, os métodos que o componente invoca [OLA 96].

Murer estabelece que certas informações de interoperabilidade são comumente publicadas em documentação adicional, porque as linguagens correntes de descrição de interfaces não estão equipadas com a capacidade de descrever interoperabilidade de componentes. Ele propõe três níveis de informação de interoperabilidade [MUR 96]:

- Nível de interface: uso de linguagem semelhante a uma IDL para descrever como componentes podem ser agrupados, estruturalmente;
- Nível originador: que tipos e versões de componentes podem trabalhar juntos (restrições do fabricante);
- Nível semântico: uma descrição completa da funcionalidade do componente.

As propostas mais recentes ressaltam que a descrição de uma interface não pode ser feita exclusivamente através da relação de assinaturas de métodos fornecidos e requeridos, mas que também deve incluir a dinâmica da interação. *Reuse contract* [LUC 97] é um mecanismo de descrição de interface e de interligação de interfaces para um conjunto de componentes participantes de uma composição e que colaboram entre si. A notação estabelece os participantes que desempenham um papel em um *reuse contract*, suas interfaces, suas relações de conhecimento (componentes referenciados) e a estrutura de interação entre conhecidos. *Reuse contracts* documentam dependências interoperacionais entre um conjunto de componentes. A figura 5.2 apresenta um exemplo de *reuse contract*, usando notação gráfica.

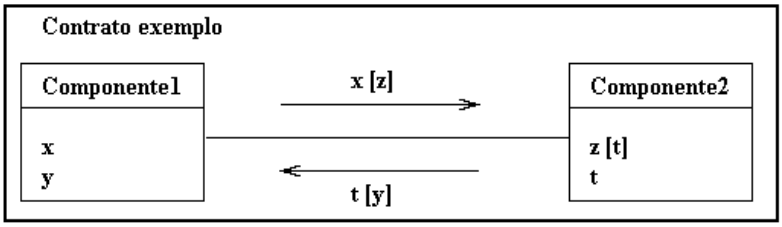

FIGURA 5.2 - Exemplo de conexão de componentes especificada através de reuse contract

Um *reuse contract* é definido como um conjunto de descrições de participantes (no exemplo, *Componente1* e *Componente2*), onde cada descrição de participante consiste de um nome único, uma cláusula de conhecimento, que estabelece o conjunto de componentes conhecidos de um participante (no caso do exemplo, a linha de conexão

entre os dois componentes representa conhecimento mútuo entre eles) e uma descrição de interface. Uma descrição de interface é um conjunto de assinaturas de métodos consistindo de um nome único e uma cláusula de especialização. Uma cláusula de especialização é um conjunto de nomes de componentes conhecidos, cada um com um conjunto de nomes de operações associadas a eles (no exemplo, a execução do procedimento *x* de *Componente1* pode acarretar na execução do procedimento *z* de *Componente2*).

Várias das propostas de descrição de interface de componentes carecem de formalismo, o que é necessário para descrever precisamente a interação entre componentes. Formalismos algébricos têm sido usados para descrever protocolos de comunicação e sistemas distribuídos e, segundo Canal, são adequados à abordagem de componentes por permitirem a avaliação de propriedades, como equivalência, inexistência de *deadlock* e outras. Canal propõe o uso de *Lambda Calculus* para a especificação de arquiteturas de componentes. Por se tratar de uma notação de baixo nível, torna-se difícil aplicá-la para descrever sistemas complexos. Uma solução para isto seria a incorporação de *Lambda Calculus* a uma linguagem de descrição de arquiteturas de componentes [CAN 97].

De um modo geral, as propostas existentes de descrição de como componentes interagem são incompletas. Dos casos acima apresentados, Murer identifica a necessidade de descrição funcional "completa" dos componentes, mas não deixa claro como fazê-lo. Lucas, com os *reuse contracts*, se atém aos aspectos estrutural e comportamental dos componentes e propõe um modelo com notação gráfica que facilita o entendimento da interação, porém sua informalidade acarreta deficiências, como a impossibilidade de verificação de propriedades como *deadlock*, o que é crítico. Canal, que se atém ao aspecto comportamental da conexão de componentes, propõe o uso de Álgebra. A notação de baixo nível usada é reconhecida pelo proponente como um obstáculo à adoção da abordagem.

5.3.2 A Necessidade de Descrever a Funcionalidade de Componentes

Dois componentes a serem interligados são estruturalmente compatíveis se o conjunto de métodos invocados através da interface de um está disponível na interface do outro. A conexão destes componentes porém, pode produzir resultados imprevistos. Seja o exemplo da figura 5.3, usando notação de *Reuse Contracts*, com dois componentes estruturalmente compatíveis interligados. Segundo a abordagem *Reuse Contracts* esta seria uma conexão válida, pois:

- durante a execução de *x*, *C1* pode invocar *y* de *C2* e *C2* dispõe deste método e
- durante a execução de *y*, *C2* pode invocar *x* de *C1* e *C1* dispõe deste método.

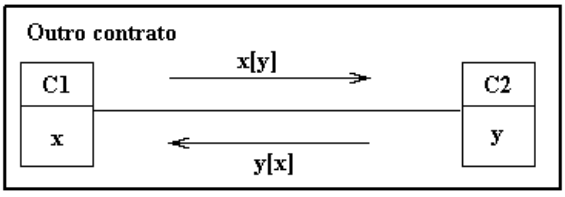

FIGURA 5.3 - Exemplo de conexão de componentes especificada através de reuse contract

Porém, se a implementação destes métodos nos componentes for tal que *C1* aguarde a invocação de *x* antes de invocar *y*, e *C2* aguarde a invocação de *y* antes de

invocar *x*, estará caracterizado um *deadlock*. Assim, estes componentes seriam estruturalmente compatíveis mas comportamentalmente incompatíveis.

Além desta limitação, através de reuse contracts também não é possível especificar restrições associadas à ordem de invocação de métodos. Considere-se uma situação hipotética em que um componente apresente um conjunto de métodos fornecidos e que um destes métodos seja um método de inicialização do componente, que deva ser invocado antes de todos os demais. Esta situação não pode ser especificada através de reuse contracts. Assim, observa-se que esta técnica consegue descrever a relação de métodos fornecidos e requeridos, as dependências entre estes métodos, a topologia de uma conexão de componentes, mas é limitada no aspecto de descrição do fluxo de controle (isto é, a descrição comportamental dos componentes e da conexão de componentes).

Garantir que dois componentes sejam estrutural e comportamentalmente compatíveis porém, também não assegura operação conjunta sem imprevistos. O caso da explosão da aeronave no vôo 501 do projeto Ariane-5 da Agência Espacial Européia (ESA), ilustra esta situação. Segundos após o lançamento, devido a uma falha no sistema de referência inercial, o sistema de auto-destruição foi ativado. Segundo o relatório da comissão que analisou o ocorrido [LIO 96], foi verificado que:

- um componente de software do sistema de referência inercial da aeronave ficou inoperante (primeiro do sistema *back-up* e em seguida do sistema ativo) em função de uma variável associada à velocidade horizontal ter assumido valor acima do limite previsto;
- estava planejado que o referido sistema de referência inercial seria testado através de simulação, porém, ao longo do projeto considerou-se este procedimento desnecessário, uma vez que o sistema (*hardware* e *software*) havia sido usado com êxito no projeto Ariane-4, que antecedeu o projeto Ariane-5;
- tal decisão foi a causa do problema, pois desconsiderou o fato de Ariane-5 assumir uma velocidade horizontal cinco vezes superior à velocidade de Ariane-4 nos segundos posteriores ao lançamento, causa do valor elevado assumido pela variável responsável pela pane.

O componente reusado era estrutural e comportamentalmente compatível com o sistema a que foi conectado, porém o desconhecimento de aspectos fundamentais da operação deste componente levou a uma situação de reuso indevido, gerando danos irreparáveis. Com isto, verifica-se que além de compreender a estrutura e o comportamento da interface de um componente, a compreensão do que o componente faz exatamente, como opera, é imprescindível para que se possa julgá-lo adequado ao reuso. Esta é a importância do uso de mecanismos de descrição de componentes capazes de descrevê-los estrutural, comportamental e funcionalmente. No capítulo 6 é descrita a solução adotada no presente trabalho para componentes e sistemas resultantes da conexão de componentes, levando em conta esse requisito.

5.4 Adaptação de Componentes

Dificilmente componentes são reusáveis tal qual foram desenvolvidos. Normalmente precisam ser adaptados para se moldarem aos requisitos do sistema a que serão acoplados. Duas abordagens têm sido usadas para adaptar um componente: alteração e empacotamento (*wrapping*) [BOS 97]. O empacotamento, ao invés de modificar o componente, cria uma visão externa diferente para ele. Outra alternativa para

interconectar componentes originalmente incompatíveis consiste em criar um componente intermediário que intermedie a comunicação. Este componente intermediário é genericamente chamado cola (*glue*) [SZY 97].

5.4.1 Empacotamento de Componentes

O empacotamento consiste em produzir uma visão externa para um componente, isto é, uma interface, diferente de sua interface original com vista a adaptá-lo a requisitos específicos. A figura 5.4 ilustra esta situação. Apresenta dois componentes, *Componente1* e *Componente2*, com interfaces incompatíveis, isto é, não podem ser conectados para operação conjunta. Na parte inferior da figura o componente *Componente2* é inserido em uma estrutura de empacotamento com uma interface diferente da interface definida neste componente. O resultado é que a interface da estrutura de empacotamento é compatível com a interface do componente *Componente1*, possibilitando a interligação de suas interfaces. Com isto, obtém-se a situação inicialmente impossível: a operação conjunta dos componentes *Componente1* e *Componente2*, com interfaces incompatíveis.

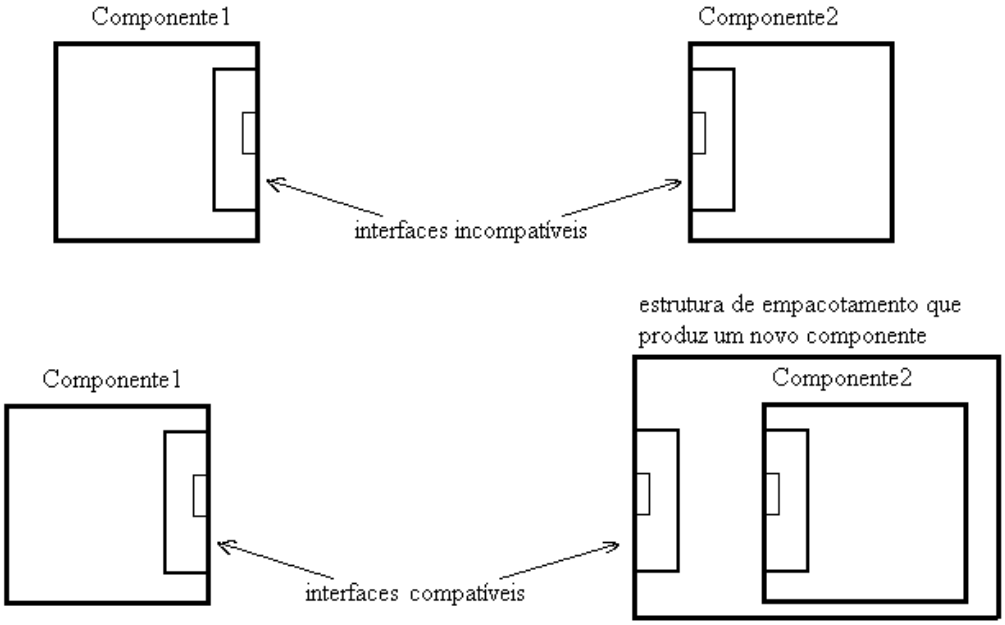

FIGURA 5.4 - Adaptação de componente através de empacotamento (wrapping)

O recurso do empacotamento pode ser usado com diferentes finalidades. Uma primeira situação é que a única incompatibilidade seja a estrutura de interface, isto é, componentes comportamental e funcionalmente compatíveis com interfaces não conectáveis. Isto pode ser causado por assinaturas de métodos diferentes em um ambiente homogêneo (diferenças em nome de método, ordem de parâmetros, previsão de retorno), ou por heterogeneidade dos componentes (que pode incluir diferença de linguagem de programação, de plataforma de execução e de localização física). No caso de diferenças sintáticas nas assinaturas de métodos (invocados por um componente e supridos por outro) a estrutura de empacotamento tem uma finalidade semelhante à do padrão de projeto *Adapter* [GAM 94]. No caso de apenas problemas de heterogeneidade, a estrutura de empacotamento se assemelha ao padrão de projeto

Proxy [GAM 94], isto é, a estrutura torna transparentes as incompatibilidades referentes a diferenças de linguagem de programação, plataforma de execução ou localização física.

Uma segunda finalidade de uma estrutura de empacotamento é alterar a funcionalidade original do componente empacotado. Neste caso, à semelhança do padrão de projeto *Decorator* [GAM 94], a estrutura poderia incluir novas funcionalidades, como também alterar o tratamento original a determinadas invocações de métodos.

5.4.2 Colagem de Componentes

A colagem de componentes (*glueing*) trata o mesmo problema do empacotamento, isto é, viabilizar a operação conjunta de componentes originalmente incompatíveis. Como no caso anterior, esta incompatibilidade pode estar associada a sintaxe das interfaces, heterogeneidade ou necessidade de extensões ou alterações funcionais. A diferença neste caso é que o tratamento dado ao problema é a inclusão de um novo elemento, a cola (*glue*), entre os componentes incompatíveis, possibilitando sua operação conjunta. A figura 5.5 ilustra a compatibilização de componentes através de colagem.

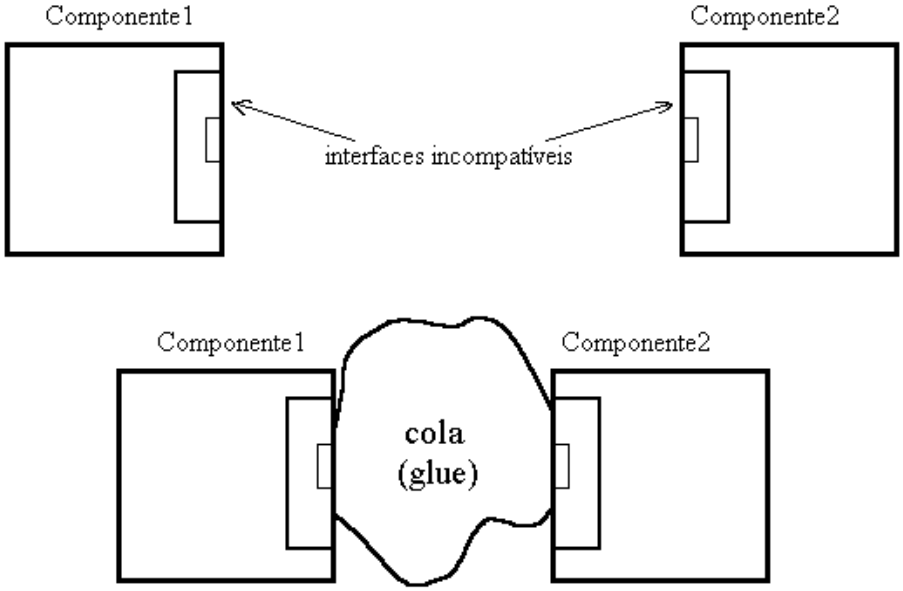

FIGURA 5.5 - Adaptação de componente através de colagem (glueing)

O elemento cola da figura 5.5 nada mais é senão um terceiro componente, cuja interface possibilita sua conexão aos componentes *Componente1* e *Componente2* e cuja funcionalidade consiste em compatibilizar a operação conjunta destes componentes. A figura 5.6 descreve a mesma situação de colagem descrita na figura 5.5, porém representando a cola como um terceiro componente, ao invés de representá-la como um elemento abstrato.

Seja a situação de colagem representada na figura 5.6 e considere-se que *Componente1* seja implementado em Smalltalk e *Componente2*, em Java. Com isto, o componente cola deve solucionar a heterogeneidade entre os outros dois componentes (que também pode envolver localização física e plataforma de execução), bem como eventuais incompatibilidades sintáticas e funcionais. Suponha-se que a questão da heterogeneidade seja resolvida com o uso de CORBA. Neste caso o componente cola mascararia o tratamento da heterogeneidade. A figura 5.7 ilustra uma visão mais refinada do componente cola, considerando esta situação.

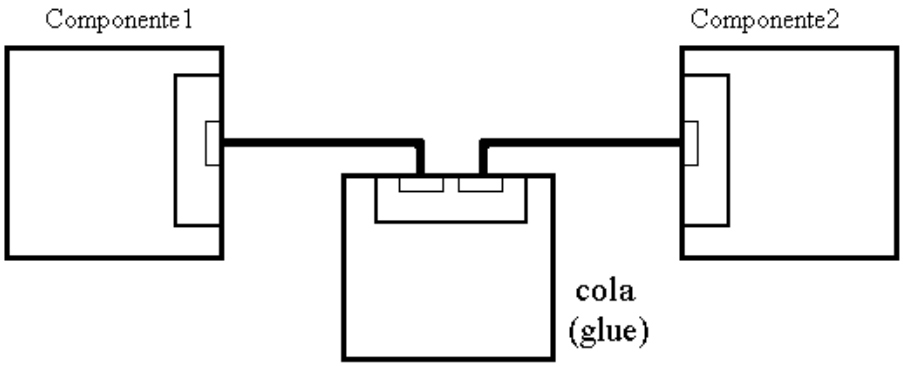

FIGURA 5.6 - A cola como um terceiro componente

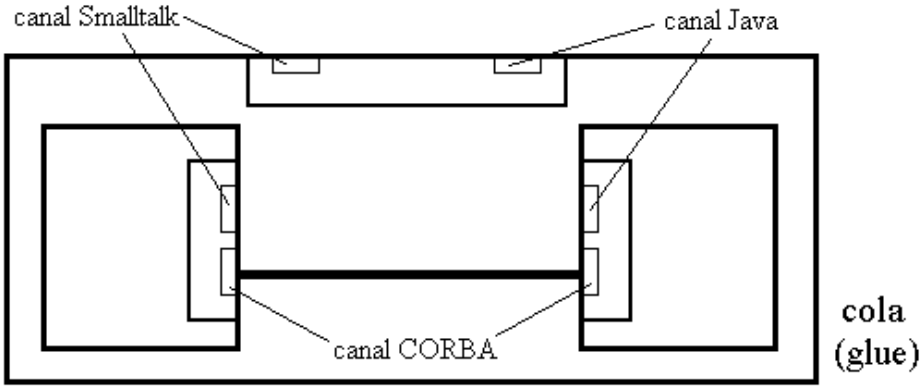

FIGURA 5.7 - Refinamento do componente cola

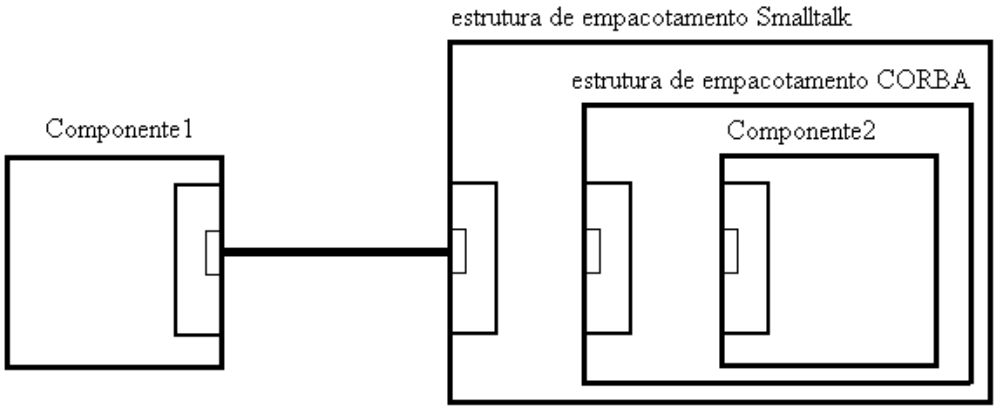

FIGURA 5.8 - Compatibilização de componentes através de empacotamentos sucessivos

O componente cola parcialmente refinado na figura 5.7 transparece que em sua constituição interna possui os elementos responsáveis pela conexão baseada em CORBA. Uma outra alternativa para conectar os componentes *Componente1* e *Componente2* do exemplo tratado seria o empacotamento. Neste caso um dos componentes estaria embutido em uma estrutura de empacotamento que mascararia a conexão CORBA. A figura 5.8 ilustra esta situação, em que o componente *Componente2* está embutido em uma estrutura de empacotamento que viabiliza sua conexão ao componente *Componente1*. Comparando os dois tratamentos dados ao mesmo problema, isto é, empacotamento e colagem para viabilizar a operação conjunta de dois componentes, observa-se que sob a ótica de cada um dos componentes originais, as soluções são idênticas. *Componente1* nos dois casos está conectado a um componente

implementado em Smalltalk, com interface compatível à sua. Para o componente *Componente1* as estruturas que excedem o próprio componente nas figuras 5.6 e 5.8 são equivalentes. De modo semelhante, *Componente2* está conectado a um componente implementado em Java, com interface compatível à sua. Pode-se dizer assim, que a colagem e o empacotamento são soluções semelhantes para um mesmo problema. As duas alternativas têm em comum o mascaramento da interface original de componentes.

Um problema observado na prática de colagem de componentes é que freqüentemente a cola fica escondida em uma estrutura de programação complexa, que torna obscuro seu propósito. Um melhor entendimento da semântica da colagem de componentes pode ser obtido pelo uso de abstrações para modelar a cola no nível de projeto, isto é, modelá-la como um componente, como descrito acima. Alguns trabalhos desenvolvidos tratam esta questão. Alencar propõe um modelo para especificar colagem de componentes baseado em pontos de vista (*viewpoints*). O relacionamento de cola é isolado e definido separadamente dos componentes [ALP 98]. A definição da cola consiste na criação de objetos ponto de vista, que são observadores de componentes existentes, sendo que:

- mais de um ponto de vista pode ser associado a um componente;
- o estado de um ponto de vista se altera quando ocorre a alteração do estado do componente observado;
- um ponto de vista atua como interface entre o componente e o meio com que ele interage, permitindo alteração e adaptação da interface original, bem como extensão de funcionalidades
- pontos de vista podem ser criados, destruídos, acoplados ou desacoplados dinamicamente.

Assmann propõe que o uso de componentes seja baseado em visões abstratas destes (semelhante à abordagem de Alencar). Para desacoplar usos de componentes de suas definições, componentes não fazem referência direta a outros componentes, mas a visões destes. Software é construído por composição. Operadores de composição são usados para combinar múltiplas visões de componentes. O modelo proposto é chamado *software cocktail mixer*. Para construir componentes complexos a partir de componentes mais simples, o modelo usa operadores de composição de definição. Este operador combina duas definições de componentes e produz uma nova [ASS 97].

Welch propõe uma estrutura de colagem baseada em reflexão. O componente de colagem, por sua capacidade de capturar as mensagens trocadas entre os componentes originais, pode interferir na comunicação, alterando a funcionalidade original dos componentes ou incluindo novas funcionalidades. Welch destaca a importância disto para a inclusão de características não funcionais à conexão, como tolerância à falta e aumento de segurança na comunicação. A idéia básica é a utilização de protocolos de meta-objetos para definir o comportamento que se deseja adicionar [WEL 98].

5.4.3 Arcabouços de Componente

A abordagem de alteração de componentes pode ser uma tarefa árdua, caso não haja uma descrição do componente que permita entender como modificá-lo. Caso somente se disponha do código fonte, é preciso proceder interpretação de código para entender o componente e descobrir o que deve ser modificado. Isto corresponde a uma atividade de Engenharia Reversa, em que é procedida uma recuperação de projeto para possibilitar a posterior alteração do componente original. Em situações que exijam demasiado esforço, a modificação de um componente pode tornar-se impraticável.

A alteração de um componente pode se tornar uma tarefa menos árdua se o componente tiver sido desenvolvido para ser alterado. Este é o caso dos arcabouços de componente (*component frameworks*). Arcabouço de componente é um componente que prevê o acoplamento de outros componentes e que é modificado a partir da troca dos componentes a ele acoplados [WEC 96] [HED 98]. A figura 5.9 ilustra genericamente um arcabouço de componente. Parte dos pontos de conexão de um arcabouço de componente são reservados à conexão dos componentes que completarão sua estrutura. Um arcabouço de componente só se torna um componente operacional, mediante o acoplamento destes componentes.

Parte dos pontos de conexão produzem a visão externamente acessível do arcabouço de componente. A capacidade de obter componentes diferentes a partir da abordagem de arcabouços de componentes reside na possibilidade de trocar os componentes que completam a estrutura, produzindo componentes resultantes diferentes.

Cabe salientar que apesar do uso comum da expressão *framework*, arcabouços de componente, ou *component frameworks*, e frameworks orientados a objetos (ou simplesmente frameworks, como vêm sendo tratados no presente trabalho) são temas distintos. Arcabouços de componentes, como componentes em geral, independem de tecnologia de implementação, enquanto os frameworks orientados a objetos estão ligados ao paradigma de orientação a objetos.

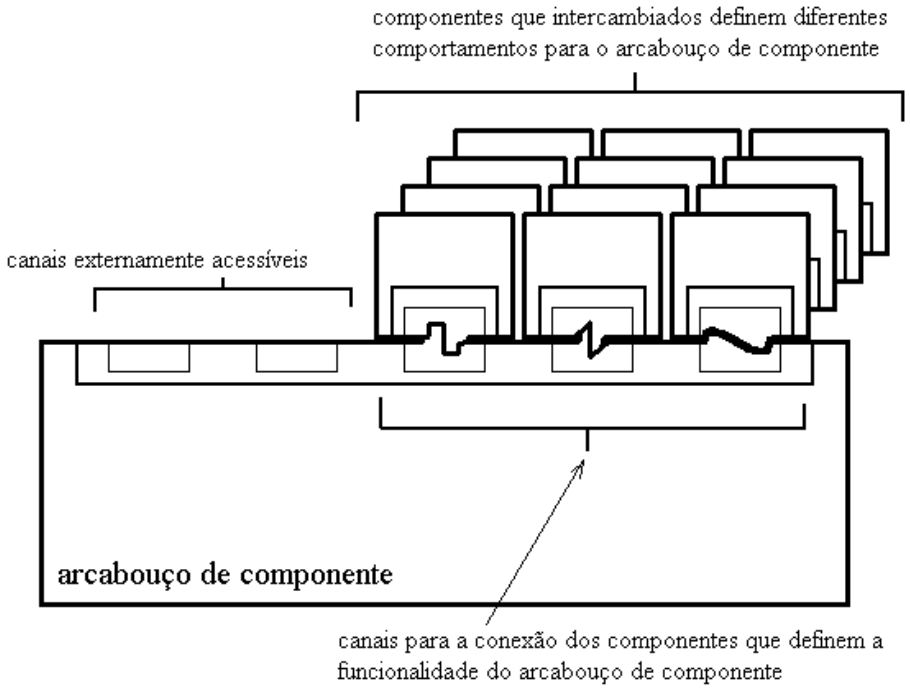

FIGURA 5.9 - Arcabouço de componente (component framework)

5.5 Arquitetura de Software

A visão sobre um software pode se ater a diferentes níveis de abstração. De fato, diferentes técnicas de modelagem produzem descrições de um software em diferentes níveis de sua escala de abstração. A arquitetura representa o mais elevado grau de abstração. Define um software em termos de um conjunto de artefatos e de conexões entre esses artefatos.

Na medida em que o tamanho e a complexidade dos sistemas de software têm aumentado, torna-se crucial definir adequadamente a organização desses sistemas em subsistemas. A busca de formas adequadas de organização, além de produzir sistemas de software de mais alta qualidade, pode permitir o reuso de suas partes constituintes em outros sistemas (na medida em que essas partes tenham funções e interfaces bem definidas) e facilitar a manutenção do software. A manutenção é facilitada pela organização do software, que facilita a sua compreensão e também possibilita a substituição de módulos constituintes, na medida em que seus limites estejam bem definidos.

Segundo Shaw, a Arquitetura de Software surgiu como uma evolução natural das abstrações de projeto, na busca de novas formas de construir sistemas de software maiores e mais complexos [SHA 96]. A Arquitetura de Software busca formas de descrever organizações de sistemas de software existentes, com vista a promover o reuso de experiência de projeto, possibilitando a escolha da forma de organização mais adequada para um sistema de software a desenvolver.

Buschmann classifica as descrições de arquitetura de software como padrões arquitetônicos, padrões de construção de software, assim como os padrões de projeto [GAM 94], porém de nível de abstração superior [BUS 96].

Uma estrutura de classes, que constitui uma aplicação, um framework ou um componente, embute uma definição arquitetônica. No caso de uma aplicação produzida a partir de um único framework, que constitua um modelo de domínio, a arquitetura da aplicação está definida na estrutura do framework. Campo prevê que um framework que define uma arquitetura para um domínio de aplicações pode ser subdividido em subframeworks, como ocorre com o framework Luthier, por ele desenvolvido, que corresponderiam aos componentes de software que constituem a arquitetura [CAM 97].

A origem da Arquitetura de Software, assim como a origem da abordagem padrões de projeto, está na verificação de que determinadas estruturas de software são adequadas para determinados tipos de problema de projeto. Por exemplo, o estilo arquitetônico *pipeline*, em que fluxos de dados são transportados linearmente através de um conjunto de componentes de software, tem sido usado para a construção de compiladores. Cada componente construtivo de um compilador recebe um conjunto de dados, procede transformações nestes dados e disponibiliza o resultado. A experiência de construção de software produziu várias estruturas organizacionais de software que são aqui tratadas como estilos arquitetônicos. A seguir são exemplificados alguns destes estilos.

5.5.1 Estilos Arquitetônicos

Uma arquitetura de software é definida a partir de um conjunto de artefatos de software e da forma como estes artefatos são interligados. Assim, um estilo arquitetônico define uma topologia de conexão específica, isto é, uma forma de interconectar os artefatos de uma arquitetura, o que restringe as possibibildades de interação entre os artefatos da topologia. Os principais elementos de uma arquitetura de software são os artefatos que constituem esta arquitetura. O que caracteriza um estilo arquitetônico, porém, é a forma como os artefatos estão conectados e como interagem. A seguir, são informalmente apresentados alguns estilos arquitetônicos comumente referenciados na literatura [SHA 96] [BUS 96].

Tubos e Filtros (Pipes and Filters)

A figura 5.10 ilustra o estilo arquitetônico tubos e filtros. Os elementos que executam processamento são os filtros, que são artefatos de software com entradas por onde recebem fluxos de dados, e saídas, por onde disponibilizam fuxos de dados resultantes de sua atuação sobre os fluxos de entrada. Tubos interligam entradas e saídas de filtros, estabelecendo uma topologia. A forma de interação característica deste estilo arquitetônico é o fluxo de dados unidirecional.

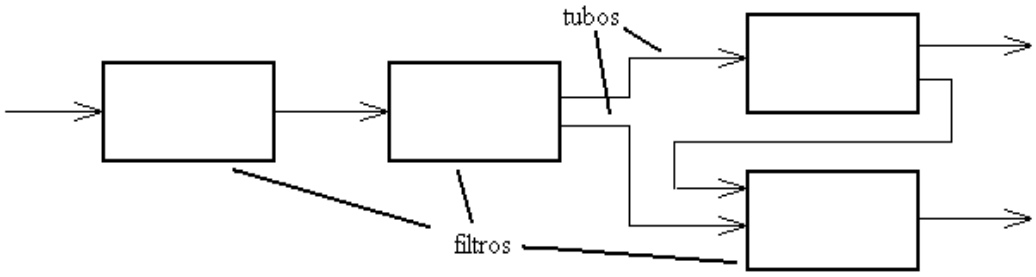

FIGURA 5.10 - Tubos e filtros (pipes and filters)

Pipeline é um caso particular de tubos e filtros em que ocorre um fluxo de dados linear ao longo dos filtros. Um compilador, como mencionado, é uma aplicação típica do estilo *pipeline*.

Sistemas em Camadas (Layered Systems)

Uma arquitetura em camadas se caracteriza pela separação dos seus artefatos de software constituintes em partes distintas (camadas) e pela restrição de conexão imposta por esta divisão. Neste estilo os artefatos de uma camada só podem ser conectados a artefatos de camadas adjacentes e cada camada pode ter uma ou duas camadas adjacentes. A forma de interação entre artefatos conectados pode ser baseada em chamadas de procedimento ou em protocolos de comunicação mais complexos.

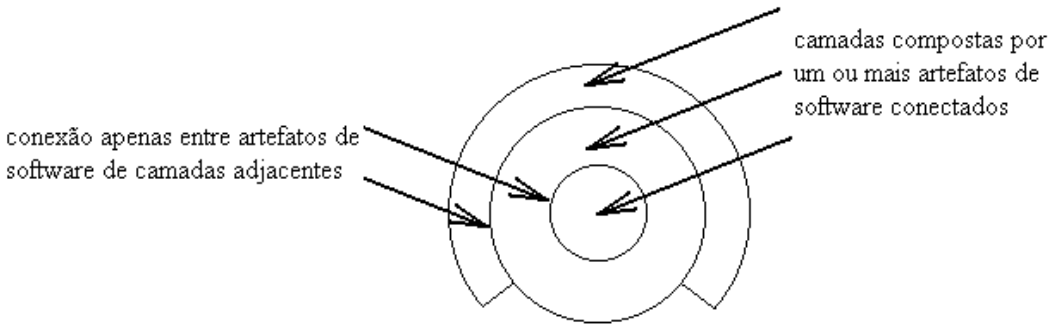

FIGURA 5.11 - Sistemas em camadas (layered systems)

Sistemas Baseados em Invocação Implícita (Implicit Invocation Systems)

Em sistemas baseados em chamada de procedimento há a definição explícita do artefato a que é dirigida a comunicação. No estilo invocação implícita a comunicação é baseada na divulgação (*broadcast*) de eventos. Artefatos podem estar ou não habilitados para reagir à ocorrência de um evento. A reação consiste na execução de algum procedimento. Quando um artefato divulga um evento, os artefatos habilitados a reagirem àquele evento executam seus respectivos procedimentos. O artefato originador do evento não invoca procedimentos de outros artefatos diretamente, mas a ocorrência do evento acarreta em invocações - daí o nome de invocação implícita.

Sistemas Baseados em Repositório

A figura 5.12 ilustra o estilo arquitetônico baseado em repositório. Neste estilo subsistemas independentes compartilham um repositório de dados. Os subsistemas não interagem diretamente entre si. A forma de interação possível entre subsistemas consiste na manipulação das informações compartilhadas através do repositório.

A interação entre subsistemas e repositório pode ocorrer através de compartilhamento de área de memória. Em aplicações mais complexas o repositório pode ser um banco de dados ou alguma estrutura mais complexa.

Uma vantagem dos sistemas baseados em repositório é sua flexibilidade. Na medida em que os subsistemas que compartilham um repositório não têm dependências entre si, torna-se mais fácil alterar o sistema através de substituição, remoção, inclusão ou alteração de subsistemas sem que os demais sejam afetados.

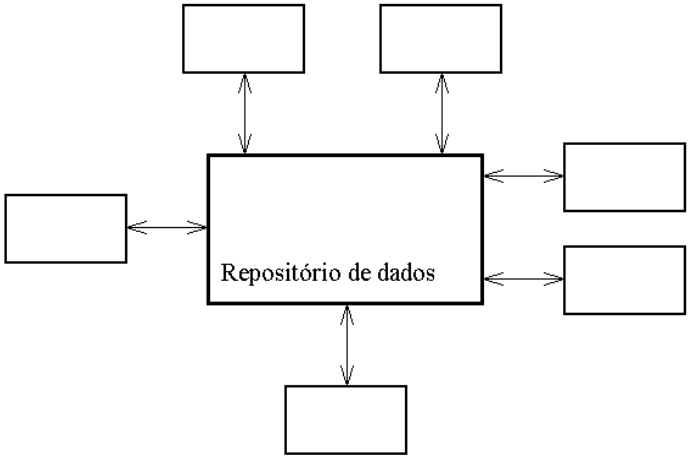

FIGURA 5.12 - Sistema baseado em repositório

Model-View-Controller

Model-view-controller (MVC) é um estilo de desenvolvimento de aplicações familiar a desenvolvedores que utilizam Smalltalk e está disponibilizado na forma de um framework em alguns ambientes de programação desta linguagem, como VisualWorks [PAR 94]. Neste estilo uma aplicação de software é dividida em três partes: *model*, que corresponde à estrutura de armazenamento e tratamento dos dados da aplicação, *view*, que corresponde à apresentação visual destes dados e *controller*, responsável pela gerência dos dispositivos de entrada de dados. Esta arquitetura é adotada em sistemas que utilizam interfaces gráficas.

A figura 5.13 ilusta uma situação de aplicação do estilo MVC [GAM 94]. Um mesmo conjunto de dados (*model*) pode ser visualmente apresentado de diferentes formas (*view*), como ilustrado na parte superior da figura. A separação da estrutura da informação de sua representação visual permite uma maior flexibilidade do sistema (como a troca de apresentação visual sem alteração da estrutura de um elemento), bem como estimula o reuso, devido à clara separação entre as partes constituintes do sistema, que podem ser reusadas conjunta ou separadamente. A separação do tratamento da entrada de dados por parte do usuário (*controller*) permite que também esta parte do sistema seja alterável (por exemplo, para uma aplicação utilizar um *joystick* ao invés de um *mouse*) sem que as outras partes sejam afetadas.

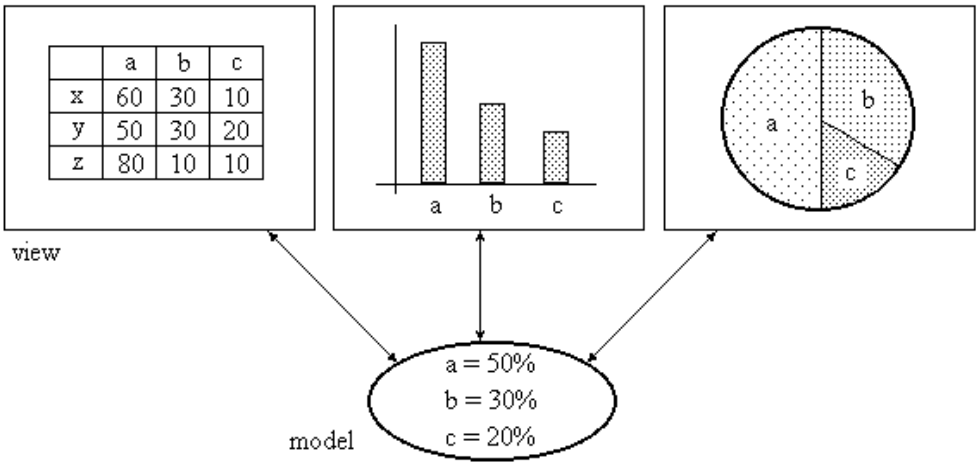

FIGURA 5.13 - Diferentes visões para um mesmo conjunto de dados

5.5.2 Descrição de Arquitetura de Software

A busca de meios de descrever arquiteturas de software constitui uma área de conhecimento com muitas questões a serem solucionadas. A visão do sistema como um todo, em um nível de abstração elevado, é importante para auxiliar o processo de compreensão do ser humano. Isto tanto pode contribuir para o entendimento de software existente, como também no auxílio ao processo de desenvolvimento. Por exemplo, saber a priori que o framework HotDraw foi desenvolvido baseado no estilo MVC, é uma informação importante no processo de interpretação de seu código. A separação das classes do framework como subclasses de *Model*, *View* ou *Controller* (classes do framework MVC) e a ciência do papel destas três classes no estilo MVC constituem um passo inicial significativo no processo busca de entendimento do framework.

Por outro lado, uma definição arquitetônica adequada pode levar ao êxito de um desenvolvimento de software, economizando esforço em ciclos de tentativa e fracasso. O conhecimento da adequação da arquitetura *pipeline* para o desenvolvimento de compiladores, por exemplo, economiza esforço para o desenvolvimento de uma aplicação deste tipo. Quando um desenvolvedor se vale deste tipo de informação está reusando experiência de projeto.

Assim, um aspecto importante de descrever arquitetura de software, como ocorre na abordagem de padrões, é a perspectiva de reuso de projeto. Buschmann produziu um catálogo em que, sob o rótulo de *patterns*, inclui descrições de estilos arquitetônicos, de padrões de projeto e de idiomas, como sendo padrões em três níveis de abstração distintos, com vista a promover reuso de experiência de projeto [BUS 96].

A forma mais freqüente de descrever arquiteturas está exemplificada na seção anterior, em que foram descritos alguns estilos arquitetônicos: descrição textual e diagramas com semântica imprecisa. Esta forma de descrição possui os inconvenientes da informalidade, isto é, as descrições podem ser incompletas e mesmo dúbias. Shaw identifica um conjunto de requisitos para os mecanismos voltados à descrição de arquiteturas de software - composição, abstração, reusabilidade, configuração, heterogeneidade, análise - apresentados a seguir [SHA 96].

> ***Composição*** *- deve ser possível descrever um sistema como uma composição de módulos e conexões independentes.*

Abstração *- deve ser possível descrever módulos e suas interações na arquitetura de software, de modo a definir com precisão, de forma clara e explícita, seus papéis abstratos em um sistema.*
Reusabilidade *- deve ser possível reusar concepções de módulos, conectores e padrões arquitetônicos em diferentes descrições arquitetônicas, para diferentes tipos de sistema.*
Configuração *- descrições arquitetônicas devem permitir entender e alterar a estrutura arquitetônica de um sistema sem a necessidade de examinar individualmente cada um dos elementos do sistema. Devem suportar também reconfiguração dinâmica.*
Heterogeneidade *- deve ser possível combinar descrições arquitetônicas distintas e heterogêneas.*
Análise *- deve ser possível proceder análise de descrições arquitetônicas.*

Identificam-se diferentes mecanismos existentes que permitem descrever a organização de sistemas. Um primeiro tipo de mecanismo corresponde a linguagens. Linguagens produzidas para diferentes finalidades permitem descrever a organização de um sistema em subsistemas, porém, em geral, são pobres para expressar a adoção de um determinado estilo arquitetônico. Um primeiro tipo de linguagem corresponde às linguagens de programação que prevêem a declaração de módulos e a organização de um sistema como uma composição de módulos. Um exemplo é a linguagem Ada, em que os módulos são definidos como *packages*.

Um segundo tipo são as linguagens voltadas a interligação de módulos, em que o aspecto *programming-in-the-large* é separado da construção de cada módulo. O ambiente ADES [FRA 88], para que foram desenvolvidas duas linguagens (uma para programação de módulos e outra para interligação de módulos) exemplifica esta situação.

Um terceiro tipo são as linguagens de descrição de interface, IDL's, que permitem descrever a interface de artefatos de software, possibilitando separar o aspecto de interligação de módulos de sua constituição interna - possibilitando inclusive, a interação de módulos heterogêneos, o que tem sido classificado como um dos principais estímulos à abordagem de desenvolvimento orientado a componentes.

Estes diferentes tipos de linguagem têm em comum a capacidade de descrever um sistema como um conjunto de módulos interligados. Não conseguem porém, exprimir claramente a adoção de um estilo arquitetônico. Suponha-se a intenção de construir um sistema adotando o estilo arquitetônico tubos e filtros. A visualização do sistema como um conjunto de módulos interligados não é suficiente para identificar a semântica da forma de interação implícita na adoção deste estilo arquitetônico.

Garlan identifica que a dificuldade de descrever adequadamente estilos arquitetônicos a partir de mecanismos de descrição existentes está na falta de capacidade de associar uma semântica específica aos elementos de conexão [GAR 98]. Afirma que os conectores, ao invés de serem tratados como elementos secundários de uma descrição, devem ser tratados como entidades de projeto com o mesmo grau de importância dos módulos que são interligados e como tal, devem ser adequadamente especificados.

A adoção de técnicas de descrição formal permite descrever módulos, conectores e combinações destes elementos com uma semântica precisa. Shaw demonstra isto produzindo especificações formais para o estilo arquitetônico tubos e filtros [SHA 96]. Nestas descrições a semântica da interação entre módulos é precisa porque os tubos são

formalmente definidos como entidades da especificação. Um problema observado nesta abordagem é que uma situação relativamente simples (tubos, filtros e as regras para sua interligação) acarreta uma especificação complexa.

5.5.3 Arquitetura de Software e Desenvolvimento Orientado a Componentes

A abordagem de desenvolvimento orientado a componentes pode servir como base para o desenvolvimento de sistemas baseados em estilos arquitetônicos específicos, homogêneos ou não. A figura 5.14 ilustra uma topologia de componentes organizada como um sistema em camadas: só há interligação entre componentes de camadas adjacentes.

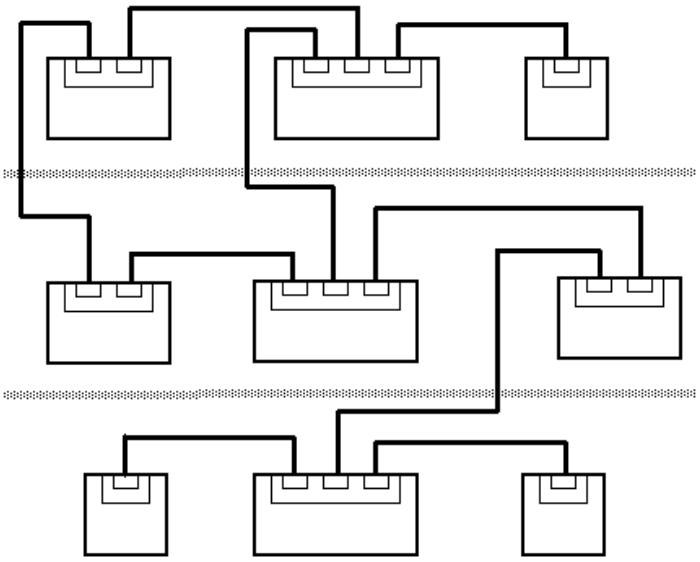

FIGURA 5.14 - Um sistema em camadas obtido a partir de uma topologia de componentes

A abordagem de desenvolvimento orientado a componentes tem sido estimulada pela perpectiva de reuso de artefatos de software complexos. A busca de formas de descrição de componentes visa torná-los compreensíveis a um maior número de desenvolvedores de software, aumentando assim seu potencial de reuso. O obstáculo da heterogeneidade é tratado por mecanismos de interconexão capazes de interligar componentes independente de linguagem de programação, localização física e plataforma de execução. A incompatibilidade entre componentes vem sendo atacada através da busca de mecanismos capazes de promover a operação conjunta de componentes originalmente incompatíveis.

Neste contexto, a Arquitetura de Software pode contribuir com o desenvolvimento orientado a componentes informando as possíveis formas de combinar componentes, as vantagens e desvantagens destas formas de combinação, bem como as situações de projeto a que são adequadas. Semelhante ao que ocorre com os padrões de projeto, subsídios para a definição da arquitetura de um sistema sob desenvolvimento constituem reutilização de projeto, porém em um nível de abstração mais elevado e que ocorre no início do processo de desenvolvimento de software.

O problema existente na abordagem de desenvolvimento orientado a componentes, de busca de meios de descrição de componentes, se estende também para a busca de meios de descrição de arquiteturas de componentes.

A questão de construir arquiteturas com explicitação semântica da forma de interação pode ser tratada a partir da definição de componentes de conexão. O uso de elementos de conexão que implementam protocolos de meta-objetos ao invés de simples apontadores, como proposto por Welch [WEL 98], ilustra esta possibilidade.

5.6 Utilização Conjunta das Abordagens de Frameworks e Componentes para a Produção de Artefatos de Software Reutilizáveis

Componentes podem ser estruturas de software bastante complexas. Assim, o reuso de componentes, como ocorre com a abordagem de frameworks, implica no reuso de projeto (materializado na estrutura do componente reusado), e de código (o componente em si, é uma estrutura de código). Uma das premissas do presente trabalho é que as abordagens de frameworks e componentes, ao invés de conflitantes, podem ser usadas conjuntamente, reforçando o potencial de promoção de reuso de cada uma delas.

Uma forma particular de utilizar a abordagem de desenvolvimento orientado a componentes é desenvolver componentes a partir do paradigma de orientação a objetos. Neste caso componentes seriam construídos como estruturas de classes e poderiam ser interligados com recursos de linguagem (apontadores) no caso de componentes homogêneos ou através de mecanismos como CORBA, no caso de componentes heterogêneos.

A adaptação de componentes é um problema identificado na abordagem de componentes. Alternativas de adaptação que fazem uso de mecanismos externos aos componentes, como ocorre com o empacotamento e a colagem, evitam proceder modificações internas nos componentes, principalmente pela dificuldade de fazê-lo. Se um componente for projetado prevendo futuras modificações, e se houver suporte para modificá-lo, é possível diminuir consideravelmente a complexidade da alteração da estrutura de componentes. Visando a criação de componentes flexíveis, adotou-se no presente trabalho a abordagem de frameworks para o desenvolvimento de componentes flexíveis.

Um framework desenvolvido para produzir componentes corresponde a uma estrutura de classes que apresenta partes mantidas flexíveis, para possibilitar sua adaptação a diferentes requisitos. Um framework pode corresponder a uma implementação inacabada de componente ou pode conter uma implementação *default*. Neste caso haveria a disponibilidade de um componente sem necessidade de adaptação da estrutura do framework. Nos dois casos a estrutura pode ser adaptada para a obtenção de componentes específicos.

A vantagem da adoção desta abordagem está na perspectiva de obter um conjunto de componentes distintos, a partir de um esforço menor que o necessário para desenvolver cada um isoladamente. Isto é possibilitado pelo reuso de projeto e código promovido pela abordagem de frameworks. As dificuldades citadas do desenvolvimento e uso de frameworks podem ser atenuadas por um suporte de desenvolvimento e uso de frameworks, como o suporte fornecido pelo ambiente SEA.

A diminuição do esforço necessário para produzir novos componentes promovida pela abordagem de frameworks viabiliza uma alternativa às abordagens ora propostas para tratar incompatibilidade de componentes: criar um novo componente reutilizando o framework e até mesmo, reutilizando parte da especificação de projeto de componentes existentes.

Além de suporte ao desenvolvimento de componentes flexíveis, o uso conjunto de frameworks e componentes pode produzir outros benefícios. Uma outra situação que pode ser suportada por este uso conjunto é a viabilização de uso de mais de um framework simultaneamente. Combinar as restrições de diferentes frameworks para operação conjunta é uma tarefa complexa. Mesmo no caso de frameworks desenvolvidos para uso conjunto, é complexo compreender como eles se relacionam, como interagem e como ocorre a distribuição de responsabilidades. Se frameworks forem desenvolvidos como componentes, isto é, contendo uma definição de interface, torna-se mais simples organizar a interação entre as diferentes estruturas.

Verifica-se que a adoção conjunta das abordagens de frameworks e componentes pode reforçar o potencial de reuso de cada uma destas abordagens. Identificam-se obstáculos em cada uma das abordagens que podem ser atacados através de suporte ferramental. A adoção do paradigma de orientação a objetos para o desenvolvimento de componentes, além de facilitar sua utilização em conjunto com frameworks, permite que um mesmo suporte ferramental possa suportar as duas abordagens - desde que supra as necessidades específicas de cada uma. No próximo capítulo é descrito o suporte provido pelo ambiente SEA para desenvolvimento e uso de frameworks e componentes.

6 O Ambiente SEA

O ambiente SEA foi desenvolvido como uma extensão da estrutura de classes do framework OCEAN e é voltado ao desenvolvimento e uso de artefatos de software reutilizáveis. O ambiente é descrito no presente capítulo, sob a ótica de sua interação com usuários, isto é, quais as estruturas de especificação tratáveis no ambiente e que funcionalidades o ambiente dispõe para manipular especificações. No próximo capítulo é descrita a estrutura do framework OCEAN, em que se trata a extensão da sua estrutura para a construção de outros ambientes que manipulem diferentes tipos de especificação, bem como apresentem diferentes funcionalidades.

SEA provê suporte para o desenvolvimento de software, possibilitando a utilização integrada das abordagens de desenvolvimento orientado a componentes - para suportar a construção de artefatos de software como arquiteturas de componentes - e desenvolvimento baseado em frameworks - incluindo o uso de padrões de projeto. Estas abordagens têm em comum a ênfase em promover reuso, tanto de implementação, como de projeto. SEA suporta o desenvolvimento de frameworks, componentes e aplicações como estruturas baseadas no paradigma de orientação a objetos. Possibilita que estes tipos de artefato sejam construídos como arquiteturas de componentes, bem como a partir da extensão da estrutura de frameworks.

No ambiente SEA, o desenvolvimento de frameworks, componentes e aplicações consiste em construir a especificação de projeto destes artefatos, utilizando UML [RAT 97] (com modificações, como descrito mais adiante) e convertendo esta especificação em código, de forma automatizada. A seguir, são descritas as funcionalidades providas pelo ambiente para o desenvolvimento de artefatos de software, com ênfase aos aspectos que diferenciam o ambiente SEA de outros ambientes existentes.

Os protótipos do framework OCEAN e do ambiente SEA foram implementados em Smalltalk, sob o ambiente VisualWorks, reutilizando classes da biblioteca de classes deste ambiente. No desenvolvimento destes protótipos foi usado o framework HotDraw [BRA 95] (implementado em Smalltalk) para suporte ao desenvolvimento de editores gráficos.

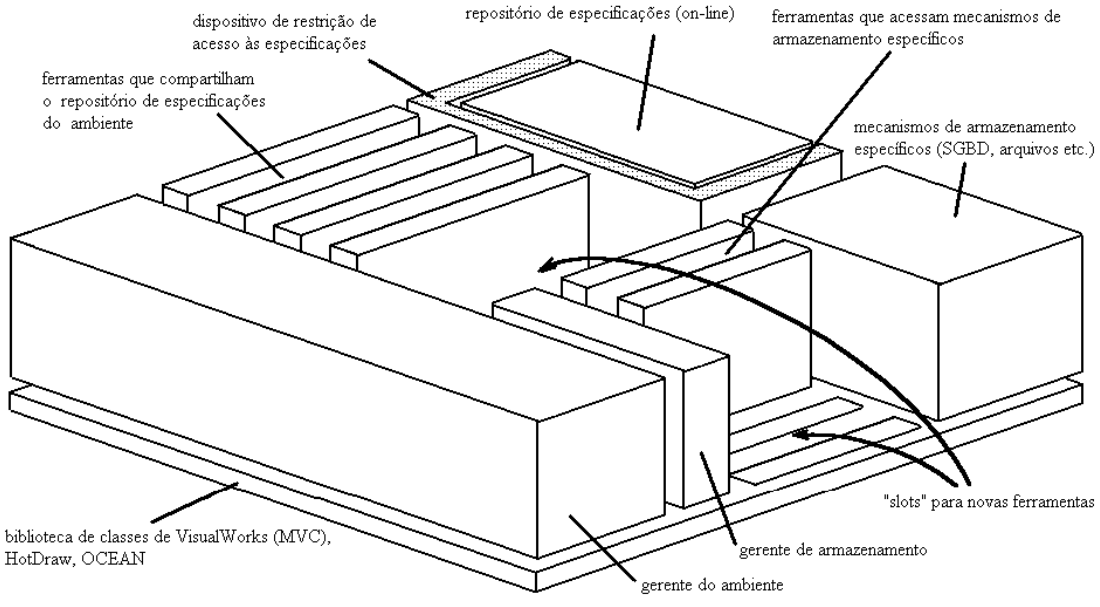

FIGURA 6.1 - Estrutura do ambiente SEA

A figura 6.1 mostra a estrutura do ambiente SEA. Baseia-se na arquitetura *toaster* [SHA 96]. O ambiente possui um repositório de especificações compartilhado por diferentes ferramentas. O gerente do ambiente provê a interface com o usuário e possibilita a manipulação das especificações contidas no repositório do ambiente por meio de ferramentas. Através de um gerente de armazenamento, especificações contidas no repositório de especificações do ambiente podem ser gravadas em dispositivos de armazenamento ou especificações contidas nestes dispositivos podem ser carregadas no repositório do ambiente.

A seguir, discutem-se os requisitos para um ambiente voltado ao desenvolvimento e uso de frameworks e componentes, o que complementa a discussão dos requisitos dos ambientes de desenvolvimento de software procedida no capítulo 3. Após, são descritas as características do ambiente SEA, que foi desenvolvido considerando o conjunto de requisitos apresentado. Inicialmente, são apresentadas as características do ambiente, comuns a outros ambientes, como a capacidade de criar e modificar especificações de projeto. A seguir, são apresentadas as funcionalidade que diferenciam SEA de outros ambientes, destacando-se a capacidade de inserção automática de padrões de projeto, a capacidade de construir de forma automática partes de artefatos desenvolvidos sob frameworks e a capacidade de construir de forma automática a interface de um componente, a partir de uma especificação de interface.

6.1 Requisitos para Ambientes de Apoio ao Desenvolvimento e Uso de Frameworks e Componentes

Um ambiente que suporte as abordagens *frameworks orientados a objetos* e *desenvolvimento baseado em componentes* deve suprir os requisitos específicos de cada uma destas abordagens e, ao mesmo tempo, possibilitar sua aplicação conjunta. A figura 6.2 ilustra algumas situações de desenvolvimento envolvendo frameworks e componentes. Um artefato de software que reusa um framework é representado no lado esquerdo e um artefato de software que reutiliza um componente, no lado direito. A figura destaca que frameworks e componentes, por sua vez, podem ser artefatos que reutilizem outros frameworks, outros componentes e padrões de projeto. Assim, no caso geral, artefatos de software (frameworks, componentes e aplicações) devem poder ser produzidos reutilizando frameworks, componentes e padrões de projeto, simultaneamente ou não. Várias combinações envolvendo reuso podem ser definidas, além daquelas ilustradas na figura.

Além dos requisitos para ambientes de desenvolvimento de software em geral, discutidos no capítulo 3 (item 3.6.1), um ambiente que suporte o desenvolvimento e o uso de frameworks e componentes deve considerar os requisitos abaixo relacionados.

- **Registro da informação referente à flexibilidade de um framework em sua especificação**: a flexibilidade conferida a um framework é o resultado de decisões de projeto e como tal, deve estar registrada na descrição do framework. Esta informação é útil para aprender a usar frameworks, bem como para alterá-los;
- **Suporte à produção de artefatos de software a partir do uso de um framework**: um ambiente que suporte o uso de frameworks deve possibilitar que um artefato de software seja produzido estendendo a estrutura de um framework. Além disto, para a geração de artefatos de software a partir de um framework, um ambiente pode dispor de recursos para direcionar as ações dos usuários do framework, reduzir o esforço

para entender o projeto do framework, liberar o usuário de atividades de baixo nível e verificar a obediência a restrições estabelecidas no projeto do framework;

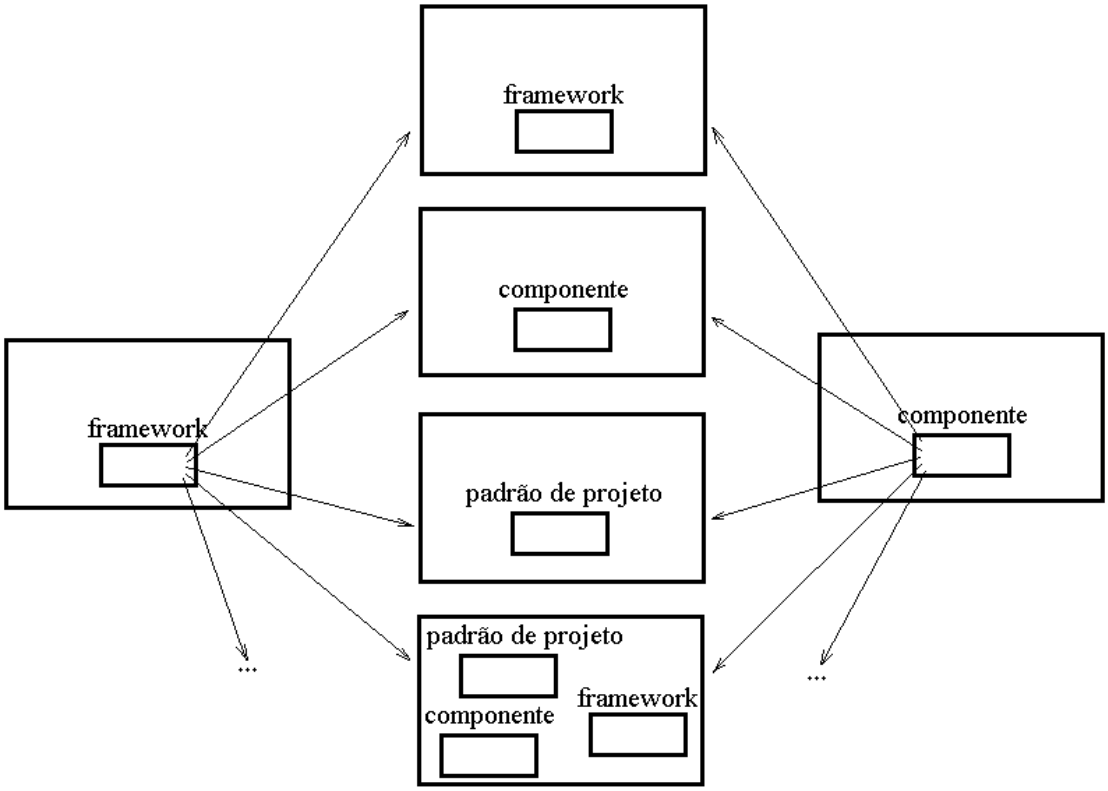

FIGURA 6.2 - Integração de abordagens de desenvolvimento de software

- **Suporte à alteração de um framework a partir de artefatos produzidos usando este framework**: deve ser possível fazer com que um framework incorpore partes de artefatos produzidos a partir dele. Este recurso permite que o framework assimile informações do domínio tratado, obtidas ao longo de seu uso;
- **Suporte à produção e à alteração de frameworks a partir de aplicações do domínio tratado**: de forma semelhante ao requisito anterior, a produção e a manutenção de um framework podem se valer de estruturas de artefatos de software do domínio tratado, desvinculados do framework. Um ambiente deve suportar o reuso de estruturas de artefatos existentes para produzir e alterar frameworks ;
- **Suporte ao desenvolvimento de componentes, considerando o desenvolvimento de interfaces e de estruturas internas de componentes como atividades distintas**: o suporte ao desenvolvimento baseado em componentes deve prever a especificação de interfaces de componentes em separado. Com isto, uma mesma especificação de interface pode ser usada por mais de um componente, incrementando o reuso;
- **Suporte à construção de artefatos de software a partir da interconexão de componentes**: deve ser possível especificar um artefato de software como uma arquitetura de componentes interconectados através de seus canais de comunicação;
- **Suporte à aplicação conjunta das abordagens de frameworks e componentes**: deve ser possível desenvolver artefatos de software reutilizando frameworks e componentes simultaneamente, permitindo por exemplo, a utilização de frameworks para o desenvolvimento de componentes e vice-versa;

- **Suporte ao uso de padrões em especificações de artefatos de software**: um ambiente voltado ao reuso deve suportar o uso de padrões na construção de artefatos de software, promovendo desta forma reuso de experiência de projeto;
- **Suporte ao uso de padrões arquitetônicos em especificações de artefatos de software**: deve ser possível especificar o estilo arquitetônico de um artefato desenvolvido como uma interconexão de componentes, promovendo outra forma de reuso de experiência de projeto .

6.2 A Estrutura de Especificações no Ambiente SEA

O protótipo do ambiente SEA ora desenvolvido suporta três tipos de especificação:

- Especificação OO[39], que pode descrever framework, aplicação, componente ou componente flexível;
- Especificação de interface de componente;
- *Cookbook* ativo, para orientar o uso de um framework previamente desenvolvido.

A figura 6.3 apresenta as superclasses abstratas do framework OCEAN que definem a estrutura das especificações manipuláveis por ambientes desenvolvidos sob este framework[40]. Esta estrutura de classes suporta a criação de estruturas de especificação distintas, através da definição de subclasses concretas. Segundo a abordagem adotada no framework OCEAN, uma especificação agrega elementos de especificação, que podem ser *conceitos*[41] ou *modelos*[42]. Um elemento de especificação pode estar associado a outros elementos de especificação. Produzir uma estrutura de especificação consiste em definir um conjunto de tipos de modelo (subclasses concretas de *ConceptualModel*), um conjunto de tipos de conceito (subclasses concretas de *Concept*) e uma rede de associações entre os elementos destes conjuntos. A estrutura de um tipo de modelo é definida em termos dos tipos de conceito que o modelo referencia. Uma especificação OO do ambiente SEA, por exemplo, utiliza diagrama de classes (um tipo de modelo), que referencia classes, mas não pode referenciar estados (isto é, instâncias destes tipos de conceito). Esta descrição da estrutura de uma especificação é refinada a seguir, a partir de uma situação concreta, a estrutura de uma especificação OO do ambiente SEA.

---

39 No presente trabalho a expressão *especificação OO* é utilizada para designar uma especificação de artefato de software, baseada no paradigma de orientação a objetos e produzida a partir do uso de notação de alto nível - UML, no caso desta tese.

40 O diagrama da figura 6.3, assim como os demais diagramas do presente capítulo foram produzidos no ambiente SEA.

41 Neste trabalho a expressão *conceito* é usada para designar as unidades de informação do domínio de modelagem tratado. No caso da modelagem baseada no paradigma de orientação a objetos como tratado no ambiente SEA, constituem tipos de conceito por exemplo, classe, atributo, mensagem, caso de uso etc. No ambiente SEA existe uma subclasse concreta de *Concept* para cada tipo de conceito tratado. Modelos referenciam instâncias destas classes.

42 Um modelo de uma especificação, como um diagrama de classes por exemplo, possui uma estrutura de informações (as classes, atributos, métodos etc.) e uma ou mais representações visuais desta estrutura (representação gráfica, textual etc.). A expressão *modelo conceitual* e a expressão *modelo* se referem à estrutura de informações. No ambiente SEA existe uma subclasse concreta de *ConceptualModel* associada a cada tipo de modelo. Um modelo de uma especificação é uma instância de uma dessas subclasses.

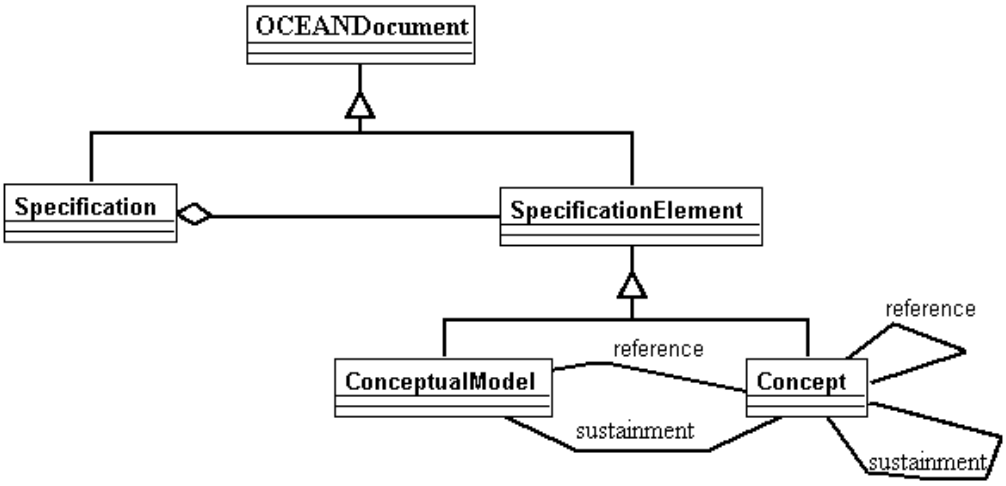

FIGURA 6.3 - Superclasses do framework OCEAN que definem a estrutura de uma especificação

6.3 Especificação OO no Ambiente SEA

O ambiente SEA utiliza UML [RAT 97] para produzir especificações OO. Uma especificação que consiste em uma estrutura de classes pode corresponder à especificação de projeto de um framework, de uma aplicação, de um componente ou de um componente flexível.

Um framework, conforme descrito no capítulo 2, é uma estrutura de classes em cujo projeto são previstas partes flexíveis, isto é, partes da estrutura que podem ser estendidas ou alteradas. Notações de metodologias OOAD em geral, incluída a notação de UML, não são totalmente adequadas à modelagem de frameworks por não possibilitarem a modelagem desta flexibilidade e nem a ligação semântica entre especificações distintas - o que é necessário para registrar a ligação entre a especificação de um artefato de software e a especificação do framework que a originou.

No ambiente SEA são utilizadas cinco das oito técnicas de modelagem propostas em UML e uma técnica adicional para descrever o algoritmo dos métodos, o que não é previsto em UML. SEA utiliza as seguintes técnicas:

- diagrama de casos de uso - para a modelagem das situações de processamento em que o sistema modelado deve atuar;
- diagrama de atividades - proposto em UML como uma alternativa ao diagrama de transição de estados, no ambiente SEA o diagrama de atividades é usado em complemento ao diagrama de casos de uso, com as atividades correspondendo a casos de uso, possibilitando o estabelecimento de restrições de ordem destes;
- diagrama de classes - classes e seus relacionamentos;
- diagrama de seqüência - refinam as situações de processamento, isto é, os casos de uso. Diagramas de seqüência também podem refinar diagramas de seqüência, possibilitando a estruturação da descrição dos casos de uso;
- diagrama de transição de estados - são associados às classes para a descrição da evolução de estado de suas instâncias. Os estados são definidos em termos dos valores assumidos pelos atributos da classe modelada. A transição de estados ocorre mediante a invocação de método da classe que pode ou não acarretar alteração de valor de atributos.
- diagrama de corpo de método (não previsto em UML) - semelhante ao diagrama de ação de Martin [MAR 91] e a outros diagramas de descrição de algoritmos como

fluxograma e diagrama de Nassi-Shneiderman, utilizados em outras metodologias de análise e projeto de software [PRR 82], [MAR 95]. É voltado à descrição dos algoritmos dos métodos das classes.

Por razões de implementação, nem todos os elementos sintáticos previstos nas técnicas de UML foram incorporados aos editores na atual versão do ambiente SEA, resultando em uma diminuição da expressividade original[43]. Por outro lado, foram introduzidas extensões para:

- representar conceitos do domínio de frameworks, não representáveis nas técnicas de UML;
- agregar recursos de modelagem a UML, como a possibilidade de estabelecimento de restrições na ordem dos casos de uso e a associação de semântica aos estados;
- possibilitar a descrição do algoritmo dos métodos na especificação de projeto;
- expressar a ligação semântica entre elementos de uma especificação, como um caso de uso e um diagrama de seqüência que o refine, e entre especificações distintas, como entre a especificação de um framework e a especificação de um artefato de software desenvolvida sob essa especificação de framework;
- possibilitar que especificações sejam tratadas como hiperdocumentos, de modo que os elementos de uma especificação possam ter *links* associados que apontem outros documentos, isto é, especificações ou elementos de especificação.

A seguir, são discutidas as influências da abordagem de frameworks sobre o processo de modelagem e são descritas as técnicas de modelagem para a descrição de especificações OO do ambiente SEA. Após, é descrita a organização das informações inseridas através da edição de diferentes tipos de modelos, na estrutura de uma especificação OO.

6.3.1 Influência da Abordagem de Frameworks no Processo de Modelagem

A abordagem de frameworks insere um conjunto de requisitos de modelagem não atendidos por metodologias OOAD em geral, o que foi um dos elementos de motivação para a produção de um novo ambiente de desenvolvimento de software.

Um primeiro aspecto relacionado ao processo de desenvolvimento de especificações de frameworks, corresponde à necessidade de explicitar na documentação de projeto as partes flexíveis de um framework. A flexibilidade de um framework reside nas classes que podem ou devem ser redefinidas através da criação de subclasses e nos métodos abstratos herdados que devem ser sobrepostos ou nos métodos *hook* que podem ser substituídos[44]. Conforme discutido no início deste capítulo, a definição da flexibilidade de um framework é parte da atividade de desenvolvimento e deve, portanto, ser explicitada para evitar esforço desnecessário de futuros usuários do framework. No ambiente SEA são introduzidas as propriedades redefinibilidade e essencialidade, abaixo descritas, associadas às classes, para registro explícito da flexibilidade de um framework.

Redefinibilidade de classe - a redefinibilidade de cada classe registrada no projeto de um framework estabelece se as classes podem ou não originar subclasses nos artefatos

---

43 A inclusão do conjunto de recursos denotacionais de UML está relacionada como um dos objetivos para futuras alterações do protótipo.

44 Substituir um método *hook* corresponde a trocar um objeto que o implementa (referenciado pelo objeto que implementa o método *template* e que invoca o método *hook* em questão) por outro objeto, instância de outra classe que apresente uma implementação diferente para este método *hook* (polimorfismo).

desenvolvidos sob o framework. Classificar uma classe de um framework como redefinível corresponde a registrar que a construção de artefatos de software sob um framework pode incluir a criação de subclasses desta classe - o que é o registro explícito de uma decisão de projeto. Uma classe de uma especificação de framework pode ser redefinível ou não. Apenas é prevista a criação de subclasses de classes redefiníveis.

Essencialidade de classe - uma classe de uma especificação de framework pode ser essencial ou não. Apenas classes redefiníveis podem ser classificadas como essenciais. Classificar uma classe de um framework como essencial corresponde a registrar que todo artefato de software produzido a partir deste framework deve utilizar esta classe (ou uma subclasse desta). No caso do framework FraG, exemplo apresentado no item 2.2, toda aplicação produzida sob este framework de jogos deve apresentar uma subclasse de *Board*, que é uma classe abstrata, redefinível e essencial.

Além das propriedades redefinibilidade e essencialidade, as classes também são classificadas como concretas ou abstratas. O ambiente também prevê a explicitação da classificação dos métodos como *base*, *template* ou *abstrato*, o que, conforme discutido no item 4.3, consiste em registro da flexibilidade associada a métodos.

Uma segunda influência inserida pela abordagem de frameworks ao processo de modelagem está relacionada ao uso de um framework existente para originar novos artefatos de software (frameworks, aplicações, componentes). Consiste na necessidade de estabelecimento de uma ligação semântica entre mais de uma especificação: entre a especificação de um artefato e a especificação do framework sob a qual foi produzida. No ambiente SEA adotou-se que uma especificação pode apontar uma ou mais "especificações-mãe". O framework FraG, por exemplo, é especificação-mãe dos jogos (aplicações) desenvolvidos sob ele. A semântica desta relação é que uma especificação desenvolvida sob uma especificação de framework corresponde a uma extensão desta, sendo assim tratada no processo de verificação de consistência. Uma especificação desenvolvida sob um framework não pode, por exemplo, possuir uma classe com nome coincidente com o de uma classe do framework.

6.3.2 As Técnicas de Modelagem do Ambiente SEA para Especificações OO

As técnicas de modelagem de UML são de domínio público e suas descrições são acessíveis na página www da empresa dos autores, *Rational* (www.rational.com), que comercializa o ambiente *Rational Rose* para modelagem baseada em UML - bem como em várias publicações recentes. Assim, a presente descrição das técnicas de modelagem usadas no ambiente SEA se atém às características que as diferenciam das técnicas de UML. Na descrição a seguir, são apresentados modelos da especificação do framework FraG para exemplificar o uso das técnicas.

Diagrama de Classes (Class Diagram)

Os conceitos representáveis no diagrama de classes do ambiente SEA são classe, com seus atributos e métodos, associação binária, agregação e herança. Todos estes conceitos podem ser classificados como externos. Um conceito externo corresponde a um conceito criado fora do contexto da especificação. Uma classe externa, por exemplo, denota uma classe referenciada pela especificação, que pode inclusive gerar subclasses.

Uma restrição estabelecida é que se uma classe for externa, todos os seus atributos e métodos também são externos e se uma classe não for externa, nenhum de seus atributos e métodos pode ser externo (exceto atributos e métodos herdados). Com

isto, classes externas só podem ter características alteradas através da criação de subclasses.

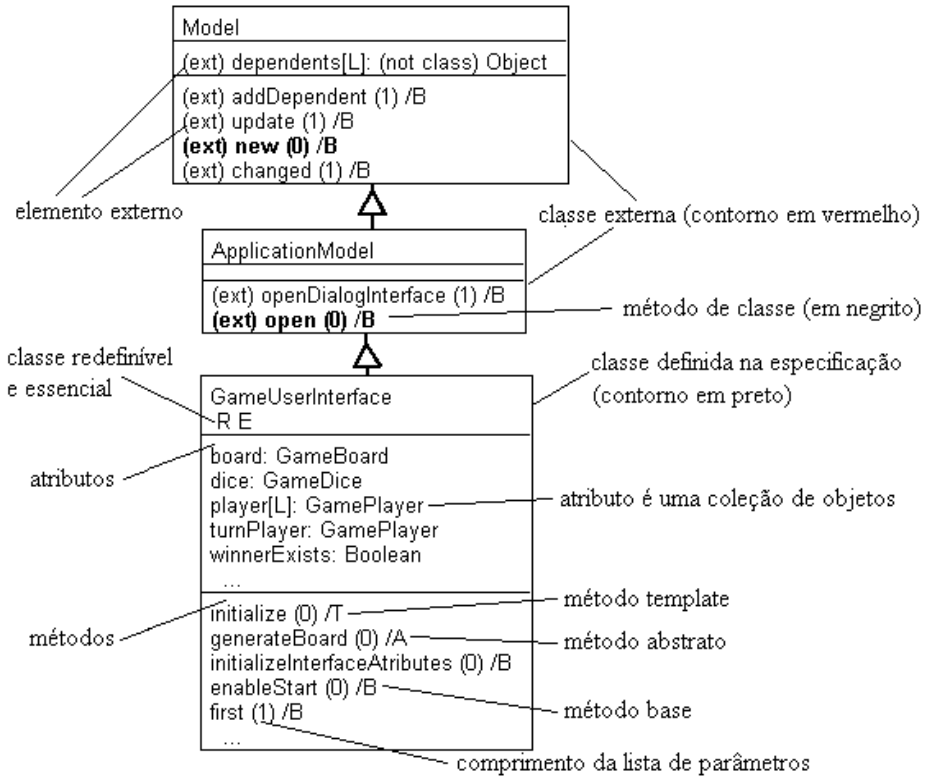

FIGURA 6.4 - Parte de um diagrama de classes do framework FraG

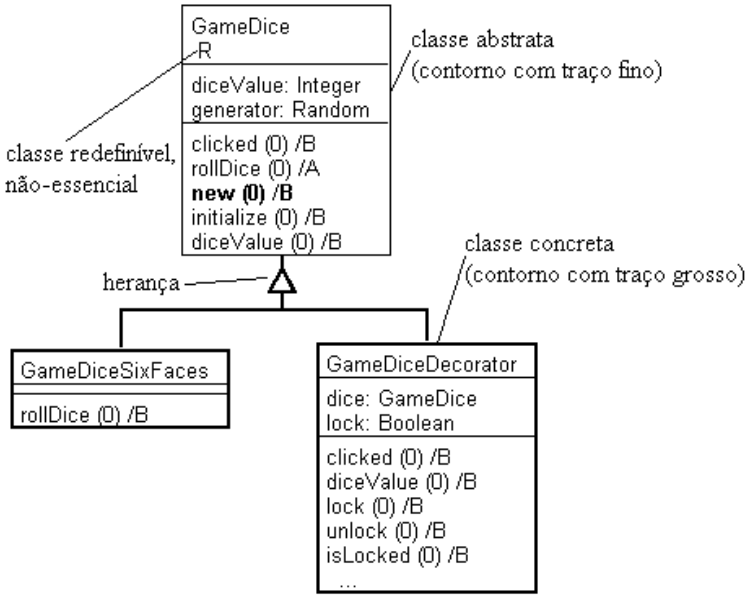

FIGURA 6.5 - Parte de um diagrama de classes do framework FraG

Uma classe externa pode referenciar ou não a especificação em que foi criada. A referência ocorre quando uma classe de um framework está presente em uma especificação de artefato de software produzida sob este framework - neste caso, a especificação do framework é referenciada pela classe classificada como externa. A figura 6.4 apresenta parte de um diagrama de classes do framework FraG. As classes

Model e *ApplicationModel* são classes externas, relacionadas por herança (herança externa), originadas do framework MVC (FraG foi desenvolvido sob MVC).

A figura 6.4 também ilustra a classificação de redefinibilidade e essencialidade associada às classes: a letra R sob o nome da classe denota classe redefinível (a ausência da letra significa classe não-redefinível) e a letra E denota classe essencial (da mesma forma, ausência da letra significa classe não-essencial). A associação da letra A, B ou T a um método representa que sua classificação é, respectivamente, *abstrato*, *base* ou *template*.

Na figura 6.5 observa-se a diferenciação entre classe abstrata (contorno com traço fino) e classe concreta (contorno com traço grosso).

Diagrama de Casos de Uso (Use Case Diagram)

O diagrama de casos de uso é composto por casos de uso, atores e relações que interligam atores e casos de uso. A figura 6.6 apresenta o diagrama de casos de uso do framework FraG.

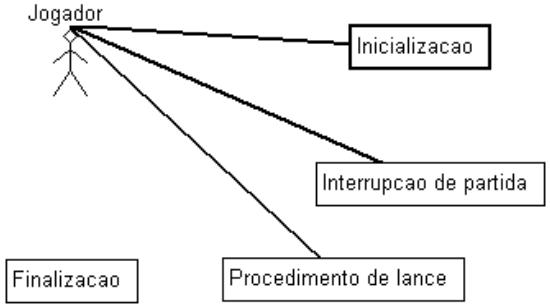

FIGURA 6.6 - Diagrama de casos de uso do framework FraG

Diagrama de Atividades (Activity Diagram)

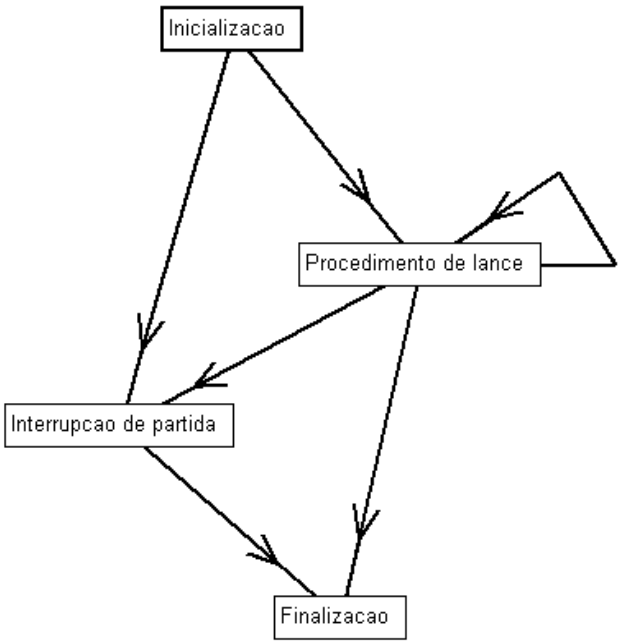

FIGURA 6.7 - Diagrama de atividades do framework FraG

O diagrama de atividades é composto por casos de uso (que correspondem às atividades) e transições entre casos de uso. A figura 6.7 apresenta o diagrama de

atividades do framework FraG. No ambiente SEA o conjunto de atividades do diagrama é o conjunto de casos de uso definido no diagrama de casos de uso. Um dos casos de uso deve ser classificado como inicial (representado com contorno grosso) - corresponde à primeira situação de processamento do sistema modelado. Todos os demais casos de uso devem ser alcançáveis a partir do caso de uso inicial (através das transições). Esta abordagem de uso do diagrama de atividades no ambiente SEA permite estabelecer restrições quanto às seqüências de situações de processamento. No exemplo da figura 6.7 observa-se que a situação de processamento "*Procedimento de lance*" só pode ocorrer após a situação de processamento "*Inicializacao*".

Diagrama de Transição de Estados (State Transition Diagram)

Um diagrama de transição de estados é associado a uma classe e cada estado é definido em termos de atributos da classe e de valores assumidos por estes atributos. Nos estados do diagrama apresentado na figura 6.8 pode-se observar na parte de baixo das figuras que representam os estados, os atributos que os definem[45].

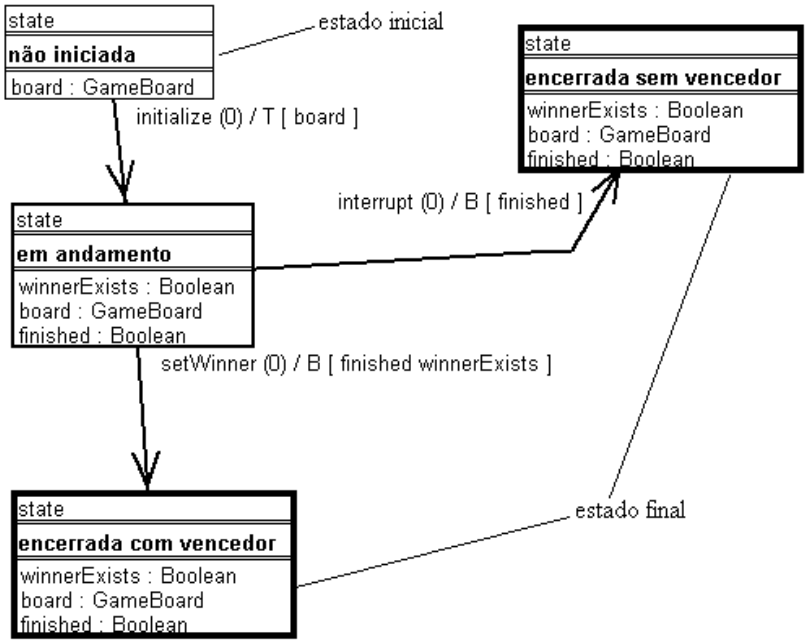

FIGURA 6.8 - Diagrama de transição de estados da classe GameUserInteface do framework FraG

Um método da classe modelada é associado a cada transição de estado. Observa-se também sobre as transições, a relação dos atributos alterados.

No diagrama de transição de estados do ambiente SEA não são representáveis estados paralelos, como ocorre em UML. Também não é possível expandir um estado em subestados em um mesmo diagrama. No ambiente SEA o refinamento de um estado pode ser feito em um diagrama separado.

Diagrama de Seqüência (Sequence Diagram)

O diagrama de seqüência é composto por mensagens e elementos que trocam mensagens. Uma mensagem é enviada a um objeto, que corresponde a uma instância de uma das classes da especificação, e invoca um método da classe desse objeto. O emissor

---

45 Os valores associados aos atributos não aparecem no diagrama. Para observar esta informação no ambiente SEA é preciso selecionar o estado e abrir uma janela de edição de estados.

de uma mensagem pode ser um objeto, um ator (do diagrama de casos de uso) ou nenhum emissor. Neste último caso, uma mensagem sem emissor - tratada no presente trabalho como *mensagem originadora* - deve necessariamente ser a primeira mensagem de um diagrama de seqüência e corresponde ao refinamento da execução do método referenciado por esta mensagem - isto é, refina a execução de um método invocado em outro diagrama de seqüência. Um exemplo de diagrama com *mensagem originadora* pode ser observado na figura 6.11, onde é refinada a execução do método "*clicked*", o último método invocado no diagrama da figura 6.10.

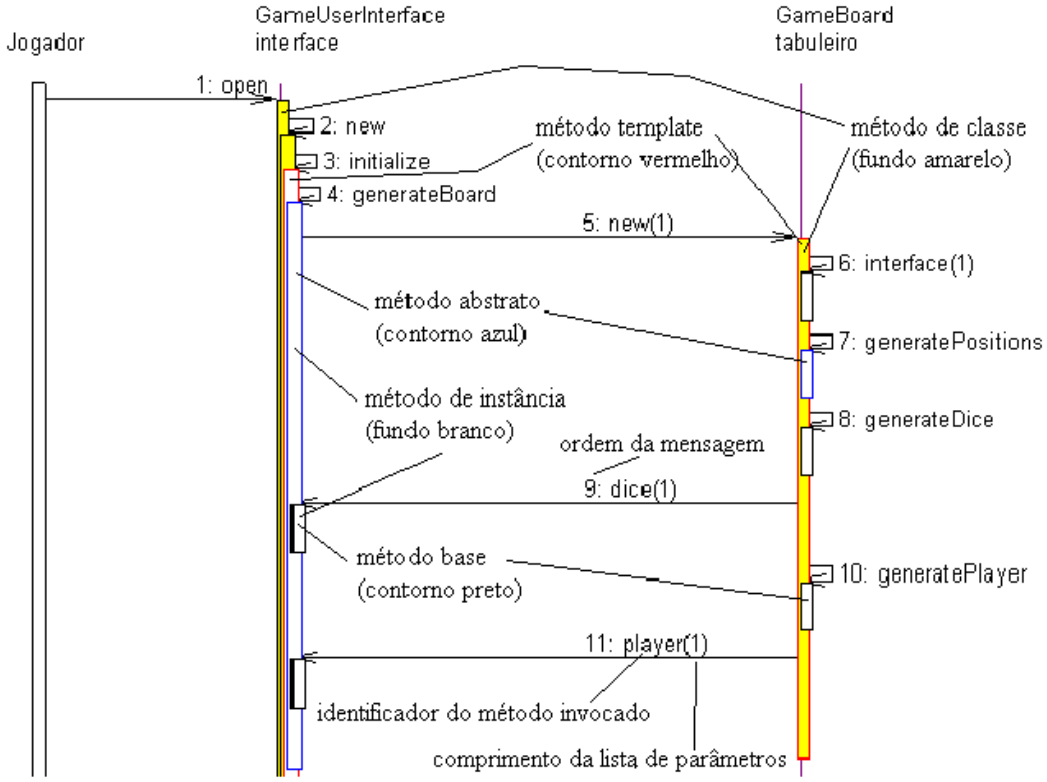

FIGURA 6.9 - Parte do diagrama de seqüência "inicialização" do framework FraG, com destaque para classificação dos métodos referenciados

Um ator como emissor denota influência externa ocasionando a execução de um método. Na primeira mensagem da figura 6.10 ocorre a invocação do método *controlActivity* da classe *GameClickModel*, por influência do ator *Jogador*. Esta representação abstrai o acionamento do botão esquerdo do mouse sobre um elemento visual, que resulta na atuação do controlador, procedimento reusado do framework MVC, como descrito no item 3.4.2.

Uma mensagem pode ter um condicionador de ocorrência associado. No ambiente SEA são previstos condicionadores do tipo *if*, *while*, *repeat* e *nTimes*. Um condicionador *if* denota que o envio da mensagem depende de uma condição (um predicado). Os condicionadores *while* e *repeat* permitem representar repetição do envio de mensagem, condicionada a um predicado. *nTimes* representa que o envio da mensagem ocorre n vezes, onde *n* é definido por uma função. As mensagens 4, 6 e 8 da figura 6.11 apresentam condicionadores *if* associados.

A classificação dos métodos e classes inserida no diagrama de classes possui uma representação visual correspondente no diagrama de seqüência, conforme ressaltado nas figuras 6.9, 6.10 e 6.11.

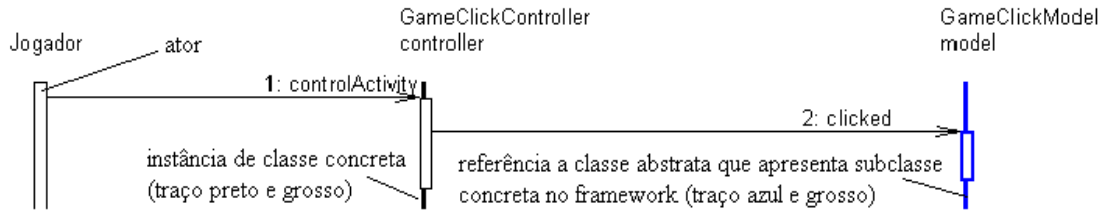

FIGURA 6.10 - Diagrama de seqüência "procedimento de lance" do framework FraG

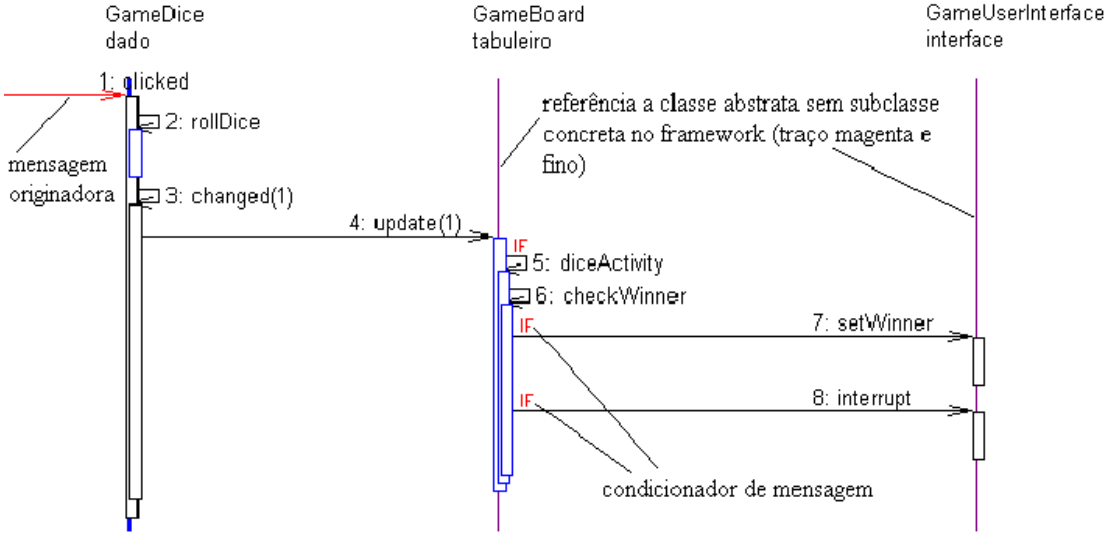

FIGURA 6.11 - Diagrama de seqüência "diceActivity" do framework FraG

Diagrama de Corpo de Método[46]

No diagrama de corpo de método do ambiente SEA estão previstos treze tipos de comandos para a descrição dos algoritmos dos métodos. Estes comandos são conversíveis para comandos equivalentes em diferentes linguagens de programação orientadas a objetos, como Smalltalk e Java. Os comandos previstos para o diagrama de corpo de método são os seguintes:

Variables - para declaração de variáveis temporárias associadas ao método descrito;
Assignment - para atribuição de atributo, variável temporária, parâmetro, classe ou constante a um atributo da classe que possui o método modelado ou a uma variável temporária deste método
Return - para retornar um objeto;
Comment - para conter comentários;
If - invólucro de outros comandos que condiciona a execução destes a um predicado;
IfElse - semelhante ao If, porém com cláusula Else;
While - invólucro de repetição em que a condição é testada antes de cada execução;
Repeat - invólucro de repetição em que a condição é testada após cada execução;
nTimes - invólucro de repetição em que a a quantidade de repetições é definida por uma função;

---

46 O diagrama de corpo de método não possui um diagrama equivalente em UML, não cabendo assim a explicitação de sua denominação em Inglês, como feito com as outras técnicas, em que é referenciada a denominação original. No anexo 1 em que é descrita formalmente a estrutura de uma especificação OO do ambiente SEA, o diagrama de corpo de método é tratado como *method body diagram*, em Inglês para manter uniformidade com as demais expressões dessa descrição.

Message - comando de envio de mensagem a objeto
, com a opção de atribuição de objeto retornado a atributo ou variável temporária. Suporta a atribuição de valor aos parâmetros do método referenciado;
Task - comporta um string que pode ser editado e que no processo de geração de código é copiado sem alteração, para o código fonte. Estabelece um grau de liberdade que permite ao desenvolvedor escrever trechos de código diretamente, se assim desejar, ao invés de compor um algoritmo ou parte de um algoritmo, a partir dos demais comandos. O conteúdo do comando task, como não é alterado no processo de geração de código, deve ser escrito na linguagem de programação alvo - com isto, a opção de usar o comando task em uma especificação limita a possibilidade de geração de código a uma única linguagem alvo;
ExternalStateTransitionDiagram - equivalente a um comentário, é destinado a conter os atributos alterados a partir da execução do método modelado, segundo descrito nos diagramas de transição de estados da especificação. Este comando foi definido para registrar a referência ao método descrito nos diagramas de transição de estados da especificação;
ExternalSequenceDiagram - equivalente a um comentário, é destinado a conter comandos Message produzidos partir da execução do método modelado, segundo descrito nos diagramas de seqüência da especificação[47]. Este comando foi definido para registrar a referência ao método descrito nesses diagramas.

A figura 6.12 apresenta o diagrama de corpo de método do método *controlActivity* da classe *ClickController* do framework FraG[48].

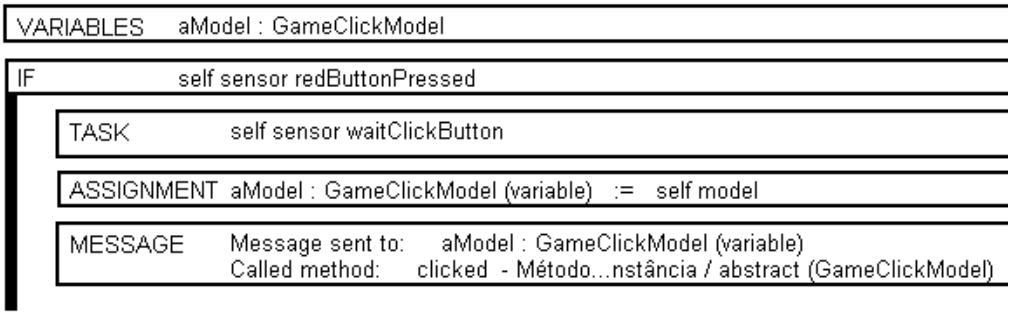

FIGURA 6.12 - Diagrama de corpo de método do método controlActivity da classe ClickController do framework FraG

6.3.3 Mecanismos de Interligação de Elementos de Especificação do Framework OCEAN

Na abordagem de estruturação de especificações adotada no framework OCEAN são utilizados mecanismos que procedem a ligação entre elementos de especificação (conceitos e modelos) definidos em diferentes contextos de modelagem. Por exemplo, para estabelecer que um caso de uso, presente em um diagrama de casos de uso, é refinado através de um diagrama de seqüência, ou que a existência de um diagrama de transição de estados está condicionada à existência da classe por ele modelada, é necessário o uso de mecanismos de ligação de elementos de especificação. O framework

---

47 Como é descrito mais adiante, o ambiente SEA dispõe de mecanismos para compor estes dois últimos comandos automaticamente, a partir da varredura dos diagramas de estado e de seqüência da especificação.

48 Detalhes que não são visíveis diretamente no diagrama podem ser observados selecionando um comando e abrindo sua janela de edição.

OCEAN dispõe de três mecanismos para estabelecer ligações entre elementos de especificação, que são usados no ambiente SEA: *links*, associações de *sustentação* e associações de *referência*.

O framework OCEAN suporta a associação de *links* a elementos de especificação para apontar outros documentos (uma especificação ou um elemento de especificação[49]). Um *link* corresponde a um tipo de conceito (subclasse de *Concept*) que pode ser associado a qualquer elemento de especificação. Os *links* têm duas finalidades nos ambientes gerados sob o framework OCEAN: possibilitar a criação de caminhos de navegação nas especificações produzidas e estabelecer ligações semânticas entre elementos de uma especificação. Um exemplo de uso de *link* para a criação de caminhos de navegação é o *link* associado a um método que aponta o diagrama de corpo de método que descreve seu algoritmo (criado automaticamente pelo ambiente, na criação do modelo). É possível criar manualmente *links* voltados à definição de caminhos de navegação, como por exemplo, um *link* associado a uma classe que aponte o diagrama de transição de estados de sua superclasse, que se julgue adequado referenciar durante a elaboração de uma especificação.

O uso de *links* com conotação semântica é previsto na estrutura semântica de uma especificação. Um exemplo de estabelecimento de ligação semântica através de *link* é a ligação entre um caso de uso e o diagrama de seqüência que o refine. Neste caso, na elaboração de uma especificação OO há a necessidade de estabelecer qual diagrama de seqüência refina cada caso de uso (vide a estrutura semântica de especificações OO, no anexo 1). Os *links* semânticos permitem estabelecer uma ligação semântica entre elementos de especificação, mantendo baixo acoplamento entre as classes envolvidas. No caso da estrutura semântica de especificações OO no ambiente SEA, casos de uso são refinados através de diagramas de seqüência. Na criação de outro ambiente, pode-se estabelecer o refinamento dos casos de uso através de outro tipo de modelo, sem que nada precise ser alterado na implementação dos modelos e conceitos relacionados a casos de uso. Um *link* estabelece uma associação que não afeta diretamente a estrutura dos elementos envolvidos (o elemento que possui o *link* e o elemento apontado pelo *link*), isto é, a remoção de um dos elementos da especificação não acarreta a alteração da estrutura do outro - podendo acarretar, por outro lado, que uma especificação se torne incompleta, como, por exemplo, a remoção de um diagrama de seqüência acarretar que a especificação não apresente o modelo que refina um caso de uso.

Uma associação *de sustentação* é um outro recurso do framework OCEAN para o estabelecimento de ligação semântica entre dois elementos de especificação distintos, em que um elemento assume o papel de *elemento sustentador* e o outro, de *elemento sustentado*. A semântica desta associação é que a permanência de um elemento sustentado em uma especificação depende da permanência do elemento sustentador. Existe uma associação de sustentação, por exemplo, entre um diagrama de transição de estados e a classe por ele modelada: a classe é o elemento sustentador e se for removida da especificação, o diagrama de transição de estados, que é o elemento sustentado, também o será. Uma vantagem da associação de sustentação definida no framework OCEAN é a diminuição do acoplamento entre diferentes tipos de elemento de especificação. É o caso, por exemplo, da estrutura de uma classe no ambiente SEA (a subclasse de *Concept* que define o conceito classe), que não prevê referência a atributos e métodos. Atributos e métodos são associados às classes através de associações de

---

49 Pode ser apontado um elemento de especificação de uma especificação distinta da que contém o link.

sustentação. Com isto, a mesma implementação de classe pode ser usada para UML, que prevê que classes apresentem atributos e métodos e para o modelo de objetos de OOSE, que não prevê métodos associados às classes [JAC 92].

A associação *de referência* entre elementos de especificação estabelece que parte da definição de um elemento de especificação, o *elemento referenciador*, depende da referência a outro elemento, o *elemento referenciado*. É o caso, por exemplo, da referência de uma mensagem (de um diagrama de seqüência) a um método. Neste caso, se o método referenciado for removido da especificação, a mensagem que o referencia não será removida, porém sua estrutura ficará incompleta, pois não estará referenciando um método.

As associações de *sustentação* e *referência* entre elementos de especificação (de uma mesma especificação), descritas como associações binárias no diagrama da figura 6.3, são registradas na estrutura de uma especificação através de tabelas - a tabela de sustentação e a tabela de referência.

6.3.4 A Estrutura Semântica de uma Especificação OO no Ambiente SEA

Especificação OO é um dos tipos de especificação tratados no ambiente SEA, que pode ser usada para produzir especificações de projeto de frameworks, aplicações ou componentes, a partir das técnicas de modelagem apresentadas. Conforme mencionado, uma estrutura de especificação é definida a partir de um conjunto de tipos de modelo, de um conjunto de tipos de conceito e de uma rede de associações entre os elementos desses conjuntos. No anexo 1 é apresentada a estrutura semântica de uma especificação OO do ambiente SEA, através de uma gramática de atributos. Na descrição a seguir são ressaltados os aspectos que caracterizam esta estrutura de especificação.

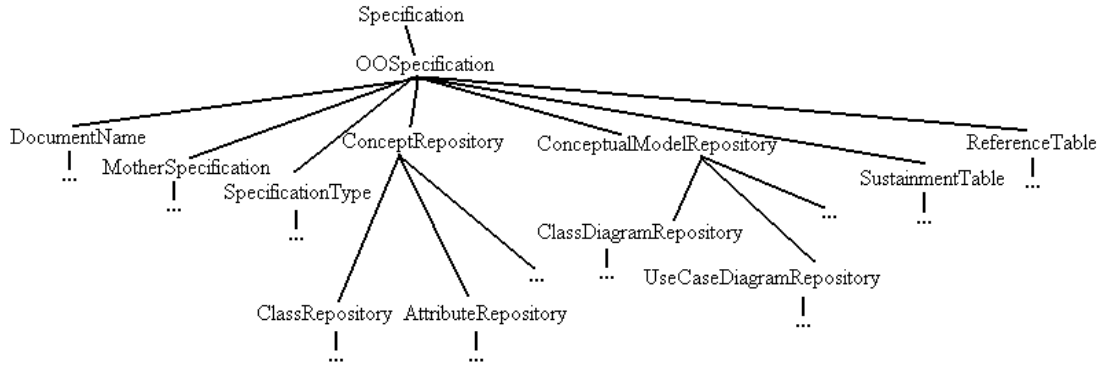

FIGURA 6.13 - Parte da árvore de derivação de uma especificação OO

A figura 6.13 apresenta parte de uma árvore de derivação originada da gramática do anexo 1 e que ilustra os principais elementos de uma especificação OO. Uma primeira característica da estrutura proposta é a presença de um repositório de conceitos (*ConceptRepository*), voltado ao armazenamento de todos os conceitos de uma especificação (isto é, instâncias de subclasses de *Concept*). Este repositório é composto por um conjunto de repositórios, um para cada tipo de conceito tratado pela especificação. A estrutura também possui um repositório de modelos (*ConceptualModelRepository*) que contém um repositório para cada tipo de modelo tratado pela especificação - no caso da especificação OO do ambiente SEA, seis

repositórios. Observam-se também na ilustração as tabelas de sustentação e referência, anteriormente descritas.

Nesta estrutura foi adotado o armazenamento centralizado de conceitos. Com isto, todos os conceitos (instâncias) produzidos na construção de modelos são armazenados no repositório de conceitos, podendo ser referenciados por um ou mais modelos (ou por outros conceitos). Assim, uma classe presente em mais de um diagrama de classes e que seja referenciada como a classe relacionada a um objeto de um diagrama de seqüência, é um único elemento da especificação. Quaisquer alterações sobre esta classe a partir de ações de edição (como troca de nome, por exemplo) produzem efeito sobre todos os elementos de especificação que a referenciam.

Um tipo de conceito é definido como uma estrutura de informações, que podem corresponder a referências a outros conceitos ou informações exclusivas do conceito tratado. Seja o conceito mensagem, presente em diagramas de seqüência. Este conceito (denominado *Message*, subclasse concreta de *Concept*) é definido no ambiente SEA a partir da seguinte estrutura (vide figura 6.11 e anexo 1):

- *ElementLinks* - conjunto dos *links* associados à mensagem;
- *ConceptualModelReferers* - conjunto de modelos que referenciam a mensagem;
- *OriginElement* - ator, referência externa[50] ou objeto que origina a mensagem;
- *OriginMessage* - mensagem de onde é originada a mensagem descrita (nem toda mensagem apresenta esta informação);
- *TargetElement* - objeto a que é enviada a mensagem;
- *TargetMethod* - método invocado pela mensagem;
- *MessageOrder* - ordem da mensagem (inteiro, maior que zero);
- *MessageWrapper* - condicionador de ocorrência da mensagem (nem toda mensagem apresenta esta informação).

Observe-se que neste caso todos os atributos, exceto *MessageOrder*, se referem a outros elementos de especificação.

Tipos de modelos são caracterizados pelos tipos de conceitos referenciados. Modelos mantém referências a conceitos que definem sua estrutura. A especificação OO do ambiente SEA utiliza seis tipos de modelos, que mantêm referência a instâncias dos tipos de conceito abaixo relacionados[51] (vide detalhes no anexo 1).

- *ClassDiagram - Class, BinaryRelationship, Aggregation, Inheritance*;
- *UseCaseDiagram - UseCase, UseCaseActor, UseCaseRelationship*;
- *ActivityDiagram - UseCase, Transition*;
- *SequenceDiagram - Message, Object, UseCaseActor, ExternalReference*;
- *StateTransitionDiagram - State, Transition*;
- *MethodBodyDiagram - Statement*.

Conforme mencionado no item 6.3.3, um *link* pode ou não assumir uma conotação semântica. É parte da estrutura de uma especificação a definição de todos os pares de elementos de especificação que devem ser relacionados através de *links* semânticos (isto é, *links* que estabelecem ligação semântica). Na especificação OO do ambiente SEA, por exemplo, todo caso de uso deve apontar através de um *link* semântico, o diagrama de seqüência que o refina. Todo diagrama de seqüência deve ser

---

50 Referência externa é o tipo de conceito associado a *OriginElement* para caracterizar mensagem originadora, isto é, o refinamento de uma mensagem invocada em outro diagrama (vide mensagem 1 da figura 6.11).

51 Na presente descrição foi usada denominação de conceitos e modelos em Inglês, como ocorre no anexo 1, bem como na implementação do ambiente SEA (denominação das classes correspondentes aos tipos de conceitos e modelos)

apontado por *link* semântico associado a caso de uso ou a mensagem. Neste caso, significa que o diagrama refina a execução do método invocado pela mensagem, recurso de modelagem que permite estruturar o refinamento dos casos de uso.

As tabelas de sustentação e referência também estabelecem ligações semânticas entre elementos de especificação. Para descobrir quais os métodos de uma classe por exemplo, buscam-se na tabela de sustentação os métodos que têm a referida classe como elemento sustentador. A seguir estão relacionados os tipos de elemento da especificação OO do ambiente SEA, seus elementos sustentadores da tabela de sustentação e seus elementos referenciadores da tabela de referência (vide detalhes no anexo 1[52]).

TABELA 6.1 - Associações estabelecidas pelas tabelas de sustentação e referência

elemento de especificação	elemento(s) sustentador(es)[53]	elemento(s) referenciador(es)
Class	–	BinaryRelationship, Aggregation, Inheritance, Attribute, Method, Type, Object, PointerStatement, DoublePointerStatement, MessageStatement
BinaryRelationship	Class	–
Aggregation	Class	–
Inheritance	Class	Class
Attribute	Class	State, PointerStatement, DoublePointerStatement, MessageStatement
Method	Class	Message, Transition, MessageStatement
MethodParameter	Method	PointerStatement, DoublePointerStatement, MessageStatement
MethodTemporaryVariable	Method, MethodBodyDiagram	PointerStatement, DoublePointerStatement, MessageStatement
Type	Class*	Attribute, Method, MethodParameter, MethodTemporaryVariable
Object	Class	–
Message	Message*, Object, ExternalReference*, UseCaseActor*	–
ExternalReference	–	–
MessageWrapper	Message	–
UseCase	–	–

52 No anexo 1 estas relações estabelecidas pelas tabelas estão representadas nos predicados, que compõem as regras semânticas associadas às produções da gramática. Nessas regras é estabelecida a necessidade de existência de pares pertencentes às tabelas que incluam os tipos de elemento relacionados.

53 É possível que os tipos de elemento de especificação marcados com asterisco (*) não sejam elementos sustentadores de todas as instâncias do tipo de elemento que aparece na tabela como elemento sustentado. Por exemplo, uma mensagem pode ter ou não uma outra mensagem como elemento sustentador: se uma mensagem for caracterizada como *mensagem originadora* (vide estrutura do diagrama de seqüência no item 6.2.1) não terá outra mensagem como elemento sustentador, caso contrário, terá.

TABELA 6.1 - Associações estabelecidas pelas tabelas de sustentação e referência

UseCaseActor	–	Class
UseCaseRelationship	UseCase, UseCaseActor	–
State	Class, State*	–
Transition	State ou UseCase	–
MultiplePointerStatement	Method,	–
PointerStatement	Method, CompositeStatement*, DoubleCompositeStatement*	–
DoublePointerStatement	Method, CompositeStatement*, DoubleCompositeStatement*	–
MessageStatement	Method, CompositeStatement*, DoubleCompositeStatement*	–
TextualStatement	Method, CompositeStatement*, DoubleCompositeStatement*	–
CompositeStatement	Method, CompositeStatement*, DoubleCompositeStatement*	CompositeStatement, DoubleCompositeStatement, PointerStatement, DoublePointerStatement, MessageStatement, TextualStatement
DoubleCompositeStatement	Method, CompositeStatement*, DoubleCompositeStatement*	CompositeStatement, DoubleCompositeStatement, PointerStatement, DoublePointerStatement, MessageStatement, TextualStatement
Link	qualquer elemento de especificação a que estiver associado	–
UseCaseDiagram	–	–
ActivityDiagram	–	–
ClassDiagram	–	–
SequenceDiagram	–	–
StateTransitionDiagram	Class, State*	–
MethodBodyDiagram	Method	–

6.4 Flexibilidade Funcional do Framework OCEAN

Além de suportar a definição de diferentes estruturas de especificação, o framework OCEAN também é flexível no aspecto de definição da funcionalidade dos ambientes desenvolvidos sob ele, isto é, as funcionalidades disponibilizadas para atuar sobre as especificações manipuladas pelos ambientes. A seguir, discutem-se as formas através das quais os ambientes produzidos sob o framework OCEAN reusam funcionalidade; no capítulo 7 discute-se como o framework OCEAN implementa esta funcionalidade e como a estrutura do framework pode ser estendida para a inclusão de novas funcionalidades.

- **Funcionalidades supridas pelo framework OCEAN**: correspondem às funcionalidades gerais, aplicáveis a diferentes ambientes, definidas no framework para serem reusadas. Algumas destas funcionalidades são completamente definidas no framework e outras precisam ser completadas em subclasses concretas das classes redefiníveis de OCEAN. Por exemplo, as funcionalidades de navegação são completamente definidas no framework OCEAN e são acessíveis através da janela do

ambiente (botões e menu) e as funcionalidades referentes à criação de conceitos e modelos que são genericamente definidas no framework, devem ser completadas na definição de tipos de modelos e de conceitos e são acessadas através de editores ou de outras ferramentas;

- **Funcionalidades supridas através de editores de modelos**: editores permitem criar, alterar e remover conceitos em modelos de uma especificação. Esta capacidade está restrita aos tipos de conceito manipuláveis pelo editor, ou seja, os tipos de conceito que definem o tipo de modelo tratado. Estas funcionalidades se utilizam de funcionalidades de edição supridas pelo framework OCEAN;
- **Funcionalidades supridas através de ferramentas**: para ações de edição, análise ou transformação sobre conceitos, modelos ou especificações e, diferentemente dos editores de modelos ou das janelas de edição de conceitos, não têm sua atuação restrita a um conceito, um modelo e nem mesmo a uma especificação. Como ocorre com os editores, as funcionalidades supridas por ferramentas reutilizam funcionalidades de edição supridas pelo framework OCEAN.

As ferramentas constituem o meio de inserção de funcionalidade mais flexível dentre as possibilidades oferecidas pelo framework OCEAN, porque a inclusão de novas ferramentas em um ambiente não exige a alteração do restante da estrutura de um ambiente. O framework estabelece uma definição genérica de ferramenta (implementada pela classe *OCEANTool*), a ser especializada para a construção de ferramentas específicas[54]. Na construção de um ambiente é definido o conjunto de ferramentas utilizáveis por este ambiente (dentre as ferramentas disponíveis no framework ou desenvolvidas para o ambiente). Estas ferramentas passam a ser acessíveis através do menu, podendo ser selecionadas durante a operação do ambiente. Ferramentas também podem ser desenvolvidas para execução de tarefa auxiliar durante a atuação de outra ferramenta, ao invés de serem selecionáveis através do menu. Esta abordagem de divisão de uma responsabilidade atribuída a uma ferramenta selecionada ser dividida em subtarefas que são distribuídas entre diferentes ferramentas é utilizada em algumas ferramentas usadas pelo ambiente SEA, como por exemplo as ferramentas de correção de erro, invocadas pelas ferramentas de análise, mencionadas mais adiante.

Ferramentas podem ser classificadas como:

- **Ferramentas de edição**: Uma ferramenta de edição tem a capacidade de ler e alterar uma especificação. Nesta categoria estão incluídos os editores de modelos. Além destes, outras ferramentas de edição podem fazer parte de um ambiente, como a ferramenta de inserção de estruturas de padrões em especificações, do ambiente SEA.
- **Ferramentas de análise**: Uma ferramenta de análise tem a capacidade de ler uma especificação, sem alterá-la. Como as estruturas de especificação definíveis sob o framework OCEAN possuem semântica definida, é possível produzir ferramentas de verificação de consistência (como as que fazem parte do ambiente SEA), de avaliação de qualidade da especificação (métricas) etc. Ferramentas de análise produzem, portanto, descrições de características de uma especificação.
- **Ferramentas de transformação**: Ferramentas de transformação fazem a ligação entre as especificações do ambiente e elementos externos. Têm a capacidade de importar ou exportar especificações. A ferramenta de geração de código Smalltalk

---

54 No protótipo atualmente desenvolvido do framework OCEAN / ambiente SEA existem mais de quarenta subclasses de *OCEANTool*, voltadas a suprir funcionalidades diversas, como algumas das ferramentas descritas nos itens seguintes.

do ambiente SEA, por exemplo, produz o código correspondente a uma especificação. Ferramentas de Engenharia Reversa podem ser incluídas no ambiente para produzir especificações (ou partes de especificações) a partir de código.

Nos itens seguintes são apresentadas as funcionalidades disponibilizadas no ambiente SEA, sob a ótica do uso do ambiente para manipulação de especificações. No capítulo 7, em que o framework OCEAN é descrito, discute-se como o framework suporta estas funcionalidades.

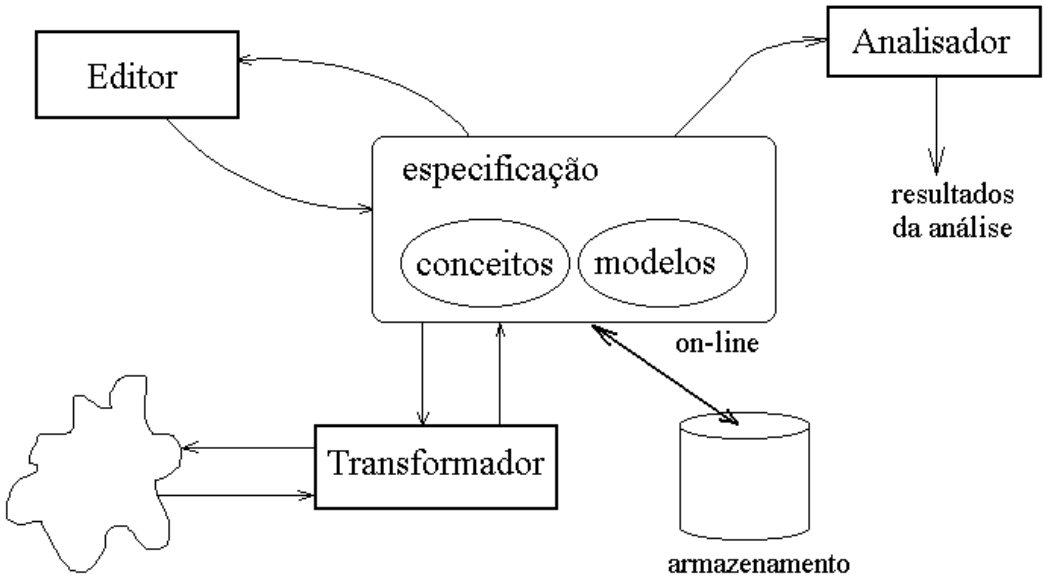

FIGURA 6.14 - Tipos de ferramenta de ambiente sob o framework OCEAN

6.5 Suporte à Produção de Especificações OO

Especificações OO no ambiente SEA podem ser construídas para especificar frameworks, aplicações ou componentes (flexíveis ou não). Estas especificações podem também ser desenvolvidas sob especificações de frameworks previamente desenvolvidas, ou reusar componentes. A presente seção se atém às funcionalidades específicas para a construção de especificações OO, aplicáveis a todas as situações acima mencionadas. Desenvolvimento de especificações a partir de especificações de frameworks e desenvolvimento e uso de especificações de componentes são tratados nas seções posteriores.

6.5.1 Criação de Especificação

A figura 6.15 apresenta a interface do ambiente SEA, sem especificação carregada (janela maior). A parte inferior da janela (a área branca da figura) pode ser ocupada por um editor gráfico para edição de modelo, por uma janela de edição de conceito ou por uma ferramenta.

A janela menor da figura 6.15, sobreposta à janela do ambiente, é carregada quando selecionada a opção de criação de especificação. Observe-se que ao criar uma nova especificação no ambiente SEA deve-se optar por criar uma especificação OO, uma especificação de interface de componente, uma especificação de arquitetura de

componentes[55] ou uma especificação de *cookbook* ativo (para orientar o uso de um framework).

Quando uma especificação é criada, é apresentada na janela do ambiente com a relação dos tipos de modelos suportados, conforme ilustrado na figura 6.16, em que foi criada uma especificação OO com o nome "*Example specification*". A seleção de um dos tipos e o acionamento de "*Go to selection*" (botão) carrega a relação de modelos do tipo selecionado, contidos na especificação.

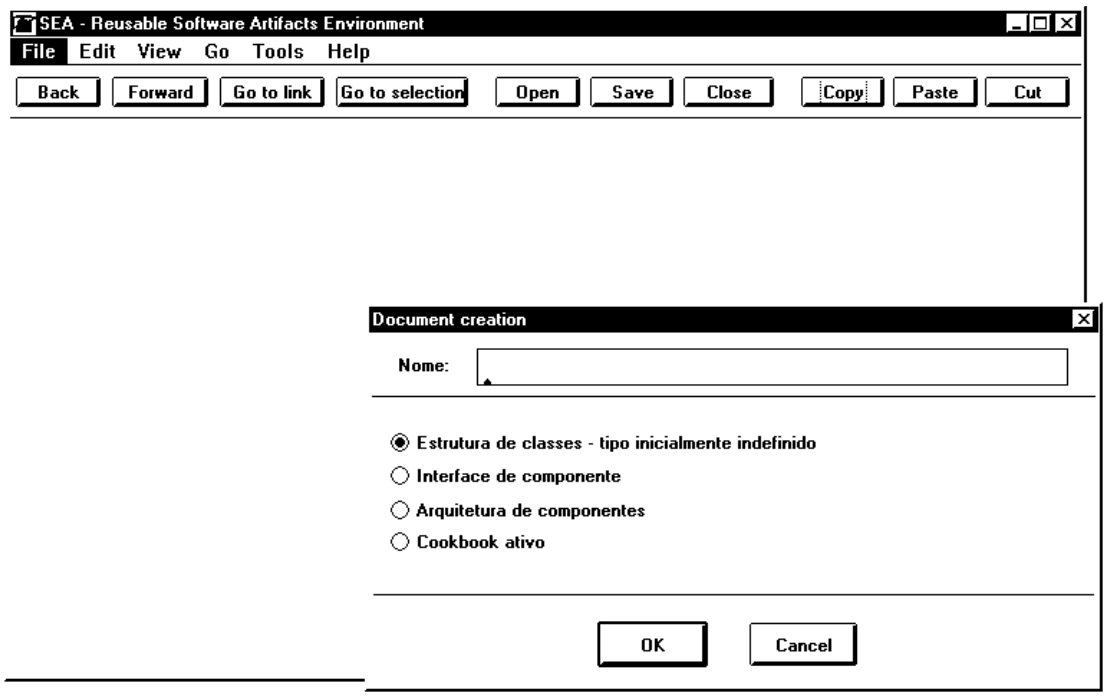

FIGURA 6.15 - Interface do ambiente SEA

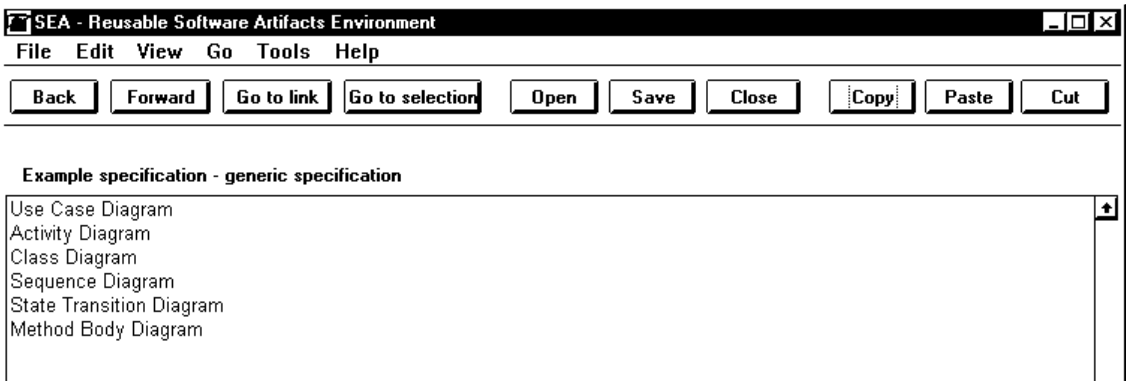

FIGURA 6.16 - Relação dos tipos de modelo de uma especificação

6.5.2 Criação e Edição de Modelos

Quando um modelo é criado, é carregado no ambiente o editor correspondente. Na figura 6.17 estão apresentados os seis editores gráficos para a construção dos

---

55 No protótipo do ambiente SEA atualmente desenvolvido esta opção não está implementada, estando prevista como um dos trabalhos futuros. Conforme descrito no item 6.7.3, arquiteturas de componentes são construídas atualmente no ambiente SEA através de especificações OO.

modelos previstos para uma especificação OO. Os editores gráficos do ambiente SEA são especializações das estruturas de editores do framework HotDraw. Um editor gráfico ocupa a parte de baixo da janela do ambiente e apresenta uma barra de ferramenta à esquerda, através da qual se tem acesso às funcionalidades exclusivas de cada editor, como as ferramentas de criação de conceito (em cada editor só é possível criar conceitos dos tipos previstos para o tipo de modelo que está sendo editado). Os seis editores apresentados na figura 6.17, com destaque para suas barras de ferramentas, são, da esquerda para a direita, editor de diagrama de casos de uso, editor de diagrama de atividades, editor de diagrama de classes, editor de diagrama de seqüência, editor de diagrama de transição de estados e editor de diagrama de corpo de método. Nas barras de ferramentas, além das ferramentas de criação de conceitos, observam-se ferramenta de seleção (a de cima em todos os editores), ferramenta de edição de *links* associados ao modelo (a de baixo em todos os editores), ferramenta de fusão de conceitos (acima da ferramenta de edição de *links*) e ferramenta de transferência de conceitos (acima da ferramenta de fusão de conceitos)

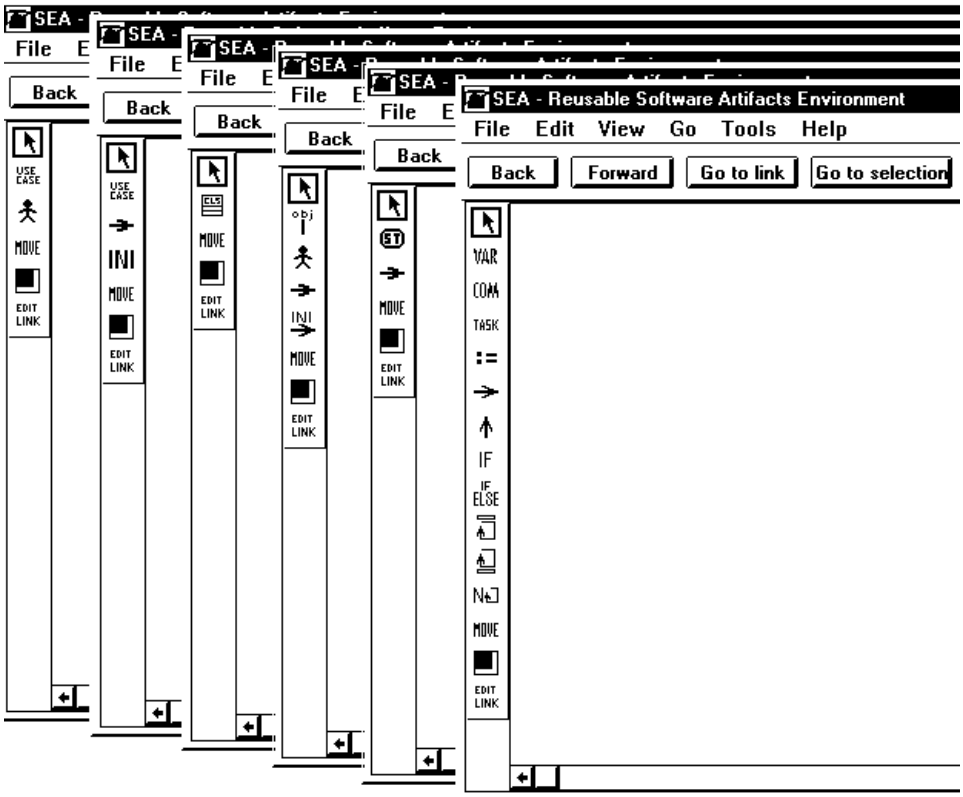

FIGURA 6.17 - Editores gráficos do ambiente SEA para especificação OO

A criação de um modelo corresponde à inclusão de uma instância da respectiva classe no repositório de modelos da especificação. A representação visual deste modelo corresponde a um diagrama gráfico (instância de subclasse concreta de *Drawing*, classe do framework HotDraw) associado ao modelo. As figuras que compõem um diagrama gráfico associado a um modelo são representações gráficas dos conceitos que compõem o modelo (subclasses concretas de *Figure*, classe do framework HotDraw).

As ações mais elementares de edição de um modelo são a criação e remoção de conceito. A criação de conceito na edição de modelo é procedida através da ferramenta correspondente, selecionada na barra de ferramenta do editor. A criação de uma classe

em um diagrama de classes, por exemplo, é procedida selecionando a ferramenta de criação de classes e acionando o mouse sobre o diagrama gráfico. Conceitos que estabelecem uma associação entre outros conceitos, como herança no diagrama de classes ou mensagem no diagrama de seqüência, são criados através de um procedimento de "*arraste e liberação*" com o mouse entre as figuras associadas aos conceitos envolvidos, isto é, desde a figura de um dos conceitos relacionados até a figura associada ao outro conceito envolvido.

A criação de um conceito no ambiente SEA é sempre uma ação semântica seguida ou não de uma ação de edição gráfica. A ação de criação de um conceito consiste em incluir uma instância do tipo de conceito criado no repositório de conceitos da especificação tratada. Somente se a ação de criação for bem sucedida é que a representação gráfica do conceito criado é produzida, isto é, uma figura a ser inserida no diagrama gráfico associado ao modelo sob edição. Quando uma ferramenta de criação de conceito é ativada ocorre um procedimento de verificação de viabilidade de criação do conceito. Na edição de um diagrama de classes, por exemplo, a tentativa de criação de ciclo de herança, o que é semanticamente inconsistente, é sempre bloqueada - e, em conseqüência, não é criada no diagrama gráfico associado ao modelo a figura representando a herança que se tentou criar.

A ação de uma ferramenta de criação de conceito nem sempre é a criação efetiva de um novo conceito na especificação, mas pode resultar no uso de um conceito já existente na especificação. Um exemplo disto se verifica na edição do diagrama de seqüência. No ambiente SEA foi definido que atores são inseridos na especificação apenas através da edição de diagramas de casos de uso. A ação da ferramenta de criação de ator no diagrama de seqüência não é a criação de um novo ator, mas a seleção de um ator do repositório de conceitos da especificação, inserido na edição de um diagrama de casos de uso. O ator existente passa a ser referenciado também pelo diagrama de seqüência e no diagrama gráfico associado a este modelo é criada uma figura associada ao ator (graficamente diferente da representação de atores usada no diagrama de casos de uso).

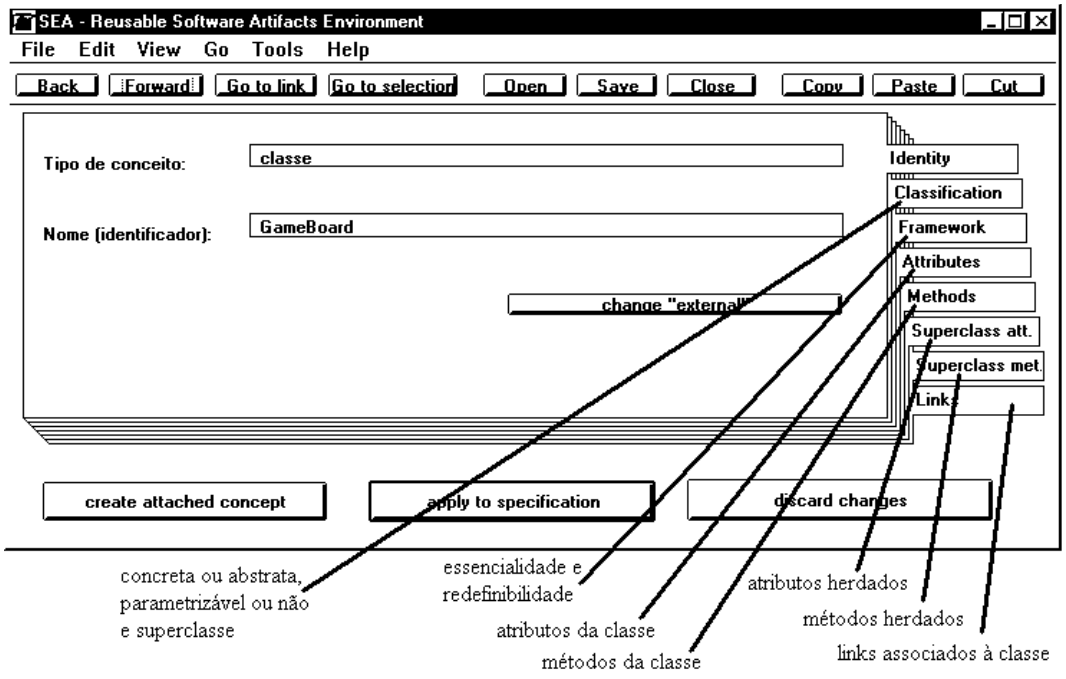

FIGURA 6.18 - Janela de edição de classe do ambiente SEA

A ação de remoção de conceito pode ocorrer no contexto de um modelo ou no de uma especificação. Um exemplo de remoção de conceito restrita ao contexto de um modelo é o de uma mesma classe contida em mais de um diagrama de classes ser removida de um deles, através da ação "cut" (acessível no menu e na barra de ferramentas). A outra possibilidade é a opção de remoção da especificação (acessível no menu) que acarreta a remoção do conceito da especificação. No caso da remoção de uma classe de uma especificação, além da classe ser removida de todos os diagramas de classe que a referenciam, também são removidos da especificação todos os elementos de especificação que têm a classe removida como elemento sustentador (associações registradas na tabela de sustentação da especificação) e os elementos de especificação que têm esses elementos removidos como elementos sustentadores, e assim sucessivamente. Por exemplo, a remoção de uma classe acarreta a remoção dos métodos associados a esta classe e, em conseqüência, a remoção de todos os parâmetros associados a esses métodos.

Nem todos os conceitos tratados por uma especificação são manipuláveis diretamente através da edição de modelos. É o caso, por exemplo, do conceito parâmetro. A criação de um parâmetro associado a um método não pode ser feita através da edição de um modelo, mas através da edição do conceito *método*. No ambiente SEA existe uma janela de edição associada a cada tipo de conceito, em que se pode alterar características específicas do conceito (como alterar o nome de um método), criar ou remover conceitos associados ao conceito sob edição (como métodos associados a uma classe ou parâmetros associados a um método). A figura 6.18 apresenta a janela de edição de classes, composta por um livro que, além da página apresentada na figura (*identity*), possui um conjunto de páginas com as demais informações de uma classe.

6.5.3 Alteração das Relações Semânticas entre Conceitos a partir de Procedimentos de Transferência e Fusão

Em uma especificação existem conceitos em que parte de sua estrutura corresponde a referência a outros conceitos - relações semânticas registradas nas tabelas de sustentação e referência, segundo a abordagem de definição de estruturas de especificação adotada no framework OCEAN. A transferência e a fusão de conceitos são ações de edição semântica disponibilizadas no ambiente SEA, restritas ao contexto de um modelo, que alteram as relações semânticas entre conceitos.

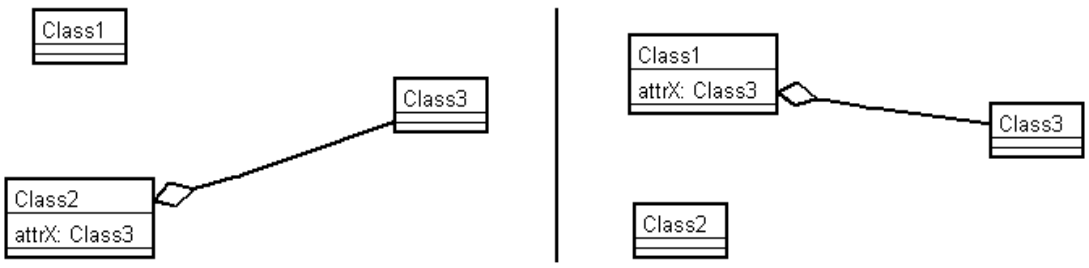

FIGURA 6.19 - Transferência de conceitos em um diagrama de classes

A transferência consiste em trocar um conceito relacionado ao conceito sob edição, por outro. A figura 6.19 ilustra duas ações de transferência procedidas sobre um diagrama de classes, envolvendo um atributo e um relacionamento de agregação. Na situação inicial o atributo *attrX* é atributo da classe *Class2* e a agregação relaciona as

classes *Class2* e *Class3*. Após as ações de transferência, o atributo *attrX* passa a ser atributo da classe *Class1* e a agregação passa a relacionar as classes *Class1* e *Class3*. Como pode ser observado na tabela 6.1 (item 6.3.4), classe é elemento sustentador de atributo e de agregação. Assim, o que ocorre na ação de transferência de conceito é a troca de um elemento sustentador do conceito alterado. No exemplo, o elemento sustentador do atributo deixou de ser a classe *Class2* e passou a ser a classe *Class1*. Da mesma forma, um dos elementos sustentadores da agregação deixou de ser a classe *Class2* e passou a ser a classe *Class1*.

A ocorrência da alteração provocada por uma transferência de conceito é propagada (padrão *Observer*) a todos os elementos de especificação que têm o conceito alterado como elemento referenciado (tabela de referência), como elemento sustentador ou como elemento sustentado (tabela de sustentação). Assim, por exemplo, um estado do diagrama de transição de estados da classe *Class2* que fizesse referência ao atributo transferido seria notificado e esta referência deixaria de existir (a referência seria mantida se, por exemplo, o atributo fosse transferido para uma superclasse da classe *Class2*).

No caso geral, a ação de transferência, que também pode ser procedida nos outros tipos de modelos, acarreta a troca de um elemento sustentador do conceito alterado. A ação não tem efeito sobre conceitos que não possuem elemento sustentador, caso do conceito classe, por exemplo. No diagrama de seqüência a ação de transferência pode ser usada para alterar ordem ou destinatário de mensagens; nos diagramas de transição de estados e de atividades, para alterar os extremos de uma transição; no diagrama de corpo de método, para alterar a posição relativa dos comandos e no diagrama de casos de uso para alterar os extremos dos relacionamentos entre atores e casos de uso.

A ação de fusão de conceitos é uma ação semântica que também altera as relações estabelecidas nas tabelas de referência e sustentação. Envolve dois conceitos de mesmo tipo que após a fusão resultarão em um único, combinando as características dos dois conceitos fundidos. A figura 6.20 ilustra a fusão de duas classes em um diagrama de classes. A ação de fundir a classe *Class1* com a classe *Class2* produz a estrutura de classes apresentada na figura, em que a classe *Class1* deixa de existir e os métodos das duas classes fundidas, assim como as associações que envolvem estas classes, passam a estar associadas à classe *Class2*.

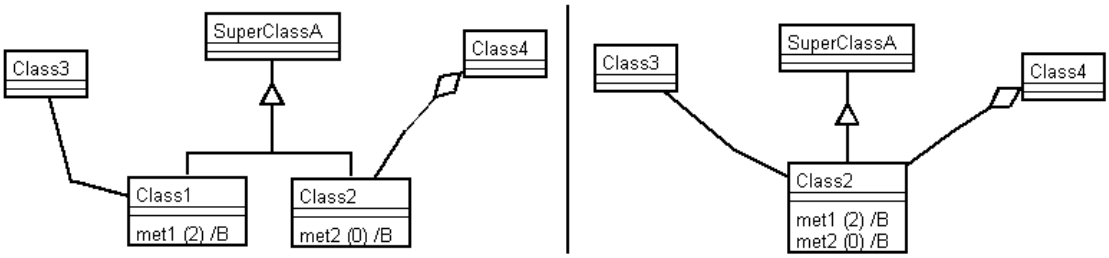

FIGURA 6.20 - Fusão de classes em um diagrama de classes

A ação de fusão também pode ser procedida envolvendo outros tipos conceitos tratados nos demais tipos de modelos. Sempre envolve dois conceitos de mesmo tipo, sendo um deles removido da especificação. Todos os elementos de especificação que têm o conceito removido como elemento sustentador passam a ter o outro conceito envolvido na fusão como elemento sustentador. De forma semelhante, todos os elementos de especificação que têm o conceito removido como elemento referenciado passam a referenciar o outro conceito envolvido na fusão.

As ações de transferência e fusão de conceitos estão apoiadas na estrutura semântica de especificação definida no framework OCEAN e sempre ocorrem sobre a estrutura conceitual da especificação, produzindo efeitos nas representações gráficas de modelos e conceitos. Observe-se que a transferência envolvendo o atributo *attrX* ilustrada na figura 6.19 não altera a estrutura da classe *Class2* (pois no ambiente SEA, classe não referencia atributo, como anteriormente mencionado), porém a representação visual desta classe é alterada (pois a representação visual de classe do diagrama de classes inclui a representação de atributos).

6.5.4 Apoio das Funcionalidades de Cópia e Colagem para Reuso de Estruturas de Conceito

Cópia e colagem é um recurso de edição semântica do ambiente SEA que usa a área de transferência do ambiente. A cópia consiste em fazer a área de transferência referenciar um ou mais conceitos selecionados. Na ação de colagem de um conceito referenciado pela área de transferência, que pode ocorrer em um modelo da mesma especificação que contém o conceito ou de outra especificação, ocorre uma das seguintes situações:

- criação de um novo conceito na especificação sob edição, igual ao conceito referenciado, que passa a ser referenciado pelo modelo em que está ocorrendo a colagem;
- o modelo em que está ocorrendo a colagem passa a referenciar um conceito já existente na especificação (isto é, o próprio conceito copiado na área de transferência).

Nos dois casos é criada no diagrama gráfico associado ao modelo sob edição, uma figura associada ao conceito colado.

A figura 6.21 ilustra o resultado de um procedimento de cópia e colagem. O atributo *attrX* associado à classe *Class2* é selecionado e copiado para a área de transferência (comando *copy* do ambiente). Após isto, a classe *Class1* é selecionada e é procedida a colagem (comando *paste*). O resultado é a criação de um novo atributo associado à classe *Class1*, igual ao atributo associado à classe *Class2*.

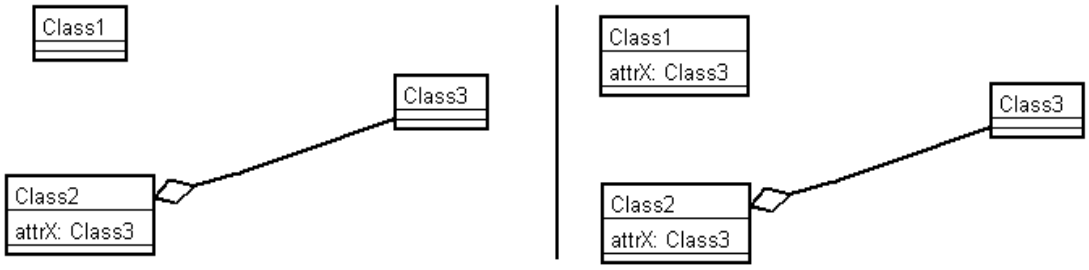

FIGURA 6.21 - Cópia e colagem de atributo em um diagrama de classes

Um atributo é um conceito que deve ter uma classe como elemento sustentador. Assim como ocorre com atributo, na colagem de qualquer conceito que deva ser sustentado por outro conceito há a necessidade de identificação do conceito que assumirá o papel de elemento sustentador do conceito a ser colado. No exemplo da figura 6.21, a classe *Class1* é selecionada antes do procedimento de colagem.

Se no procedimento de colagem de um conceito que demande um elemento sustentador, não houver um conceito selecionado, é procedida uma busca de elemento

sustentador. No caso da colagem de atributo busca-se dentre as classes referenciadas pelo modelo sob edição, uma com o mesmo nome da classe associada ao conceito a ser colado. A colagem não é procedida se não for identificado elemento sustentador.

Quando um conceito é copiado na área de transferência do ambiente, também são copiados os conceitos da especificação que o têm como único elemento sustentador e que são representáveis no tipo de modelo que o referencia. Esta característica do procedimento de cópia faz com que uma classe seja copiada para a área de transferência juntamente com todos os atributos, métodos e tipos sustentados por ela, e juntamente com os parâmetros sustentados pelos métodos copiados. Na figura 6.4, que apresenta um diagrama de classes da especificação do framework FraG, observam-se as classes *Model* e *ApplicationModel*, que foram copiadas da especificação do framework MVC e coladas no diagrama da especificação de FraG. Como a especificação do framework MVC é especificação-mãe da especificação do framework FraG, nesta especificação as classes coladas (assim como seus atributos e métodos) são classificadas como externas.

As funcionalidades de cópia e colagem, além de serem acessíveis para a edição de especificações, também são usadas por ferramentas do ambiente cuja ação envolva a inclusão de conceitos em modelos de uma especificação - como ocorre, por exemplo, com a inserção de padrões de projeto.

6.5.5 Criação Automática de Métodos de Acesso a Atributos

A criação de métodos de leitura e de escrita para os atributos das classes de uma especificação é uma tarefa monótona e, na medida em que aumenta o número de atributos, exige um certo esforço de desenvolvimento. O ambiente SEA possui duas ferramentas para criação automática de métodos de acesso a atributos, capazes de criar toda a estrutura de um método, isto é, assinatura e corpo.

A *ferramenta de criação automática de métodos de acesso a atributos* cria um método de leitura e outro de escrita para cada atributo de uma classe selecionada. A figura 6.22 ilustra o efeito da ação da ferramenta sobre uma classe. A figura 6.23 apresenta os diagramas de corpo de método dos métodos de acesso a um dos atributos.

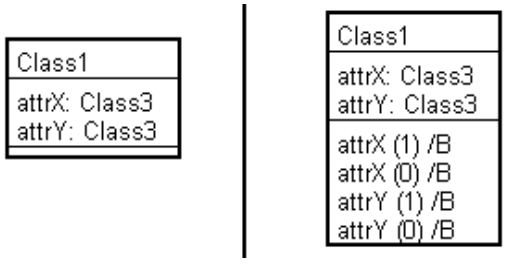

FIGURA 6.22 - Criação automática de métodos de acesso a atributos

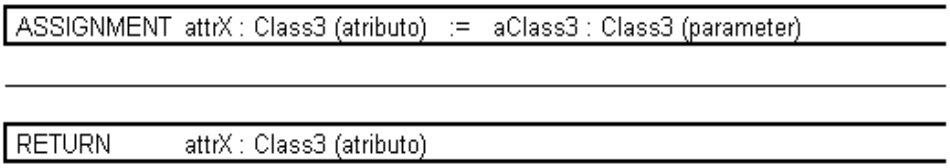

FIGURA 6.23 - Diagramas de corpo de método de acesso ao atributo attrX (de escrita acima e de leitura abaixo)

Para cada atributo tratado pela ferramenta, são criados e associados à classe selecionada um método de leitura com o mesmo nome do atributo tratado e um método de escrita com o mesmo nome do atributo e com um parâmetro com o mesmo tipo do atributo. Esta inclusão de conceitos ocorre através de procedimentos de cópia e colagem, conforme descrito no item anterior. Além disto, para cada método é criado um diagrama de corpo de método associado. No diagrama do método de leitura é inserido um comando *return* que retorna o atributo tratado; no diagrama do método de escrita é inserido um comando *assignment* que atribui o parâmetro ao atributo tratado. Estas inserções de comandos nos diagramas de corpo de método também correspondem a procedimentos de cópia e colagem.

A outra ferramenta, a *ferramenta de criação de métodos de acesso a conjuntos de atributos*, atua de forma semelhante porém permite que sejam selecionados os atributos tratados, que seja criado apenas método de leitura ou escrita e que se defina o nome do método criado. A figura 6.24 apresenta esta ferramenta carregada na janela do ambiente SEA.

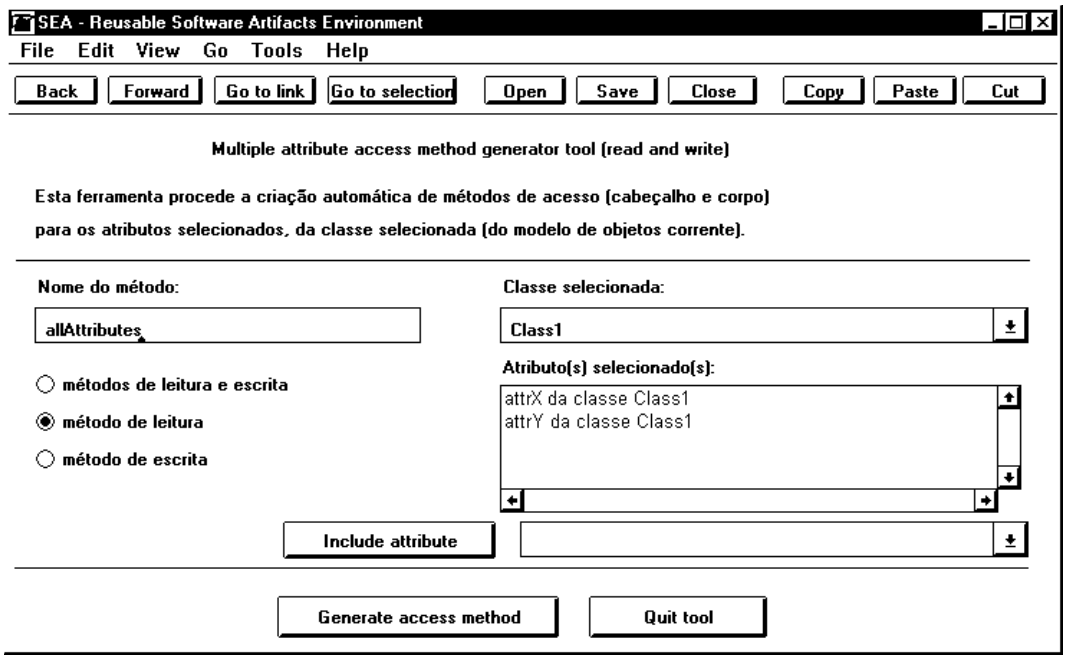

FIGURA 6.24 - Ferramenta de criação de métodos de acesso a conjuntos de atributos

O processo de criação dos métodos - assinatura e corpo - realizado pelas duas ferramentas é completamente automatizado. As únicas intervenções do usuário são seleção da classe para a primeira ferramenta e fornecimento das informações previstas na figura 6.24 para a segunda ferramenta.

6.5.6 Suporte Automatizado à Composição de Diagrama de Corpo de Método, a partir da Análise de Diagramas de Seqüência e de Transição de Estados

A modelagem dinâmica a partir das técnicas de modelagem disponibilizadas no ambiente SEA para construir especificações OO prevê a descrição no nível de abstração mais elevado utilizando casos de uso, para a enumeração das situações de processamento a que o sistema modelado pode ser submetido. No nível de abstração imediatamente inferior, os casos de uso são refinados a partir de diagramas de seqüência, que podem ser

refinados por outros diagramas de seqüência. Nos diagramas de seqüência as situações de processamento são descritas como interações entre objetos, a partir de troca de mensagens. O nível de abstração mais baixo corresponde à descrição dos algoritmos dos métodos das classes. A descrição do algoritmo de um método deve ser compatível com o conjunto de diagramas de seqüência de uma especificação, isto é, o algoritmo deve possibilitar todos os envios de mensagem previstos no conjunto de diagramas de seqüência da especificação, a partir da execução do método. Este requisito de consistência exige que durante a construção dos diagramas de corpo de método seja considerada a referência ao método descrito nos diagramas de seqüência da especificação - o que pode ser uma tarefa árdua, na medida em que uma especificação apresente uma grande quantidade de diagramas de seqüência.

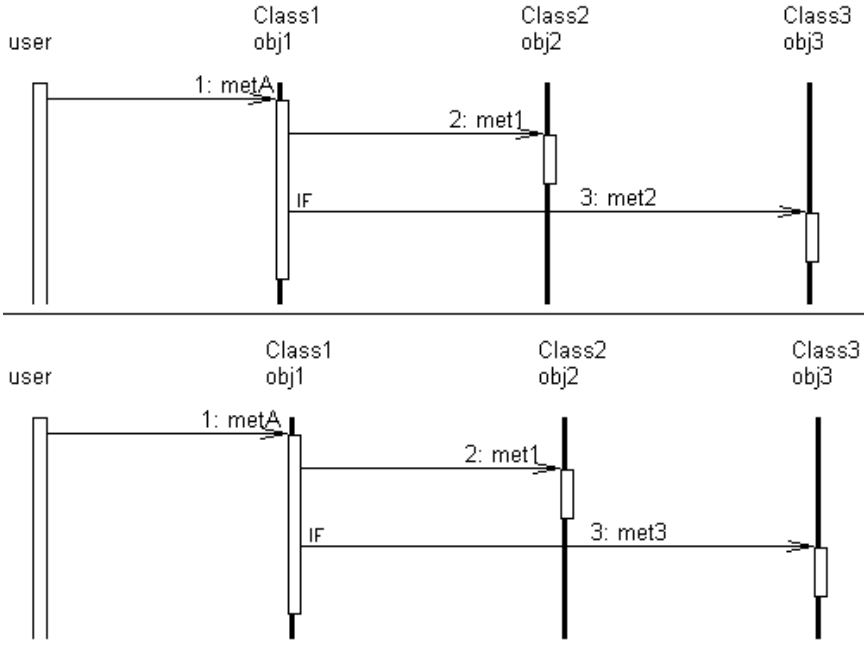

FIGURA 6.25 - Diagramas de seqüência de uma especificação hipotética

A outra técnica para modelagem dinâmica disponível, o diagrama de transição de estados, permite descrever a evolução de estados das instâncias de uma classe, isto é, a alteração dos atributos, em função da invocação dos métodos da classe. Esta descrição também deve ser levada em conta na construção dos diagramas de corpo de método. Quando se verifica em um diagrama de transição de estados que a execução de um método de uma classe resulta na alteração do valor de um atributo desta classe, significa:

- que durante a execução do método o atributo é alterado ou
- que durante a execução do método é invocado outro método que altera o atributo ou
- que durante a execução é iniciada uma seqüência de invocação de métodos que resulta na alteração do atributo.

Assim, as referências a um método nos diagramas de transição de estados da especificação devem ser consideradas durante a construção do diagrama de corpo de método associado a este método.

O ambiente SEA possui a ferramenta de geração de comandos *external* (*ExternalStateTransitionDiagram* e *ExternalSequenceDiagram*), de apoio à construção de diagramas de corpo de método, que faz a varredura dos diagramas de seqüência e dos diagramas de transição de estados. Esta ferramenta produz automaticamente comandos

external no diagrama de corpo de método sob edição, que registram a participação do método que está sendo modelado naqueles diagramas e que podem ser usados para a construção dos diagramas de corpo de método.

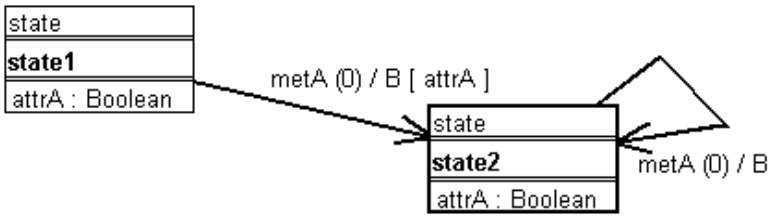

FIGURA 6.26 - Diagrama de transição de estados de uma das classes de uma especificação hipotética

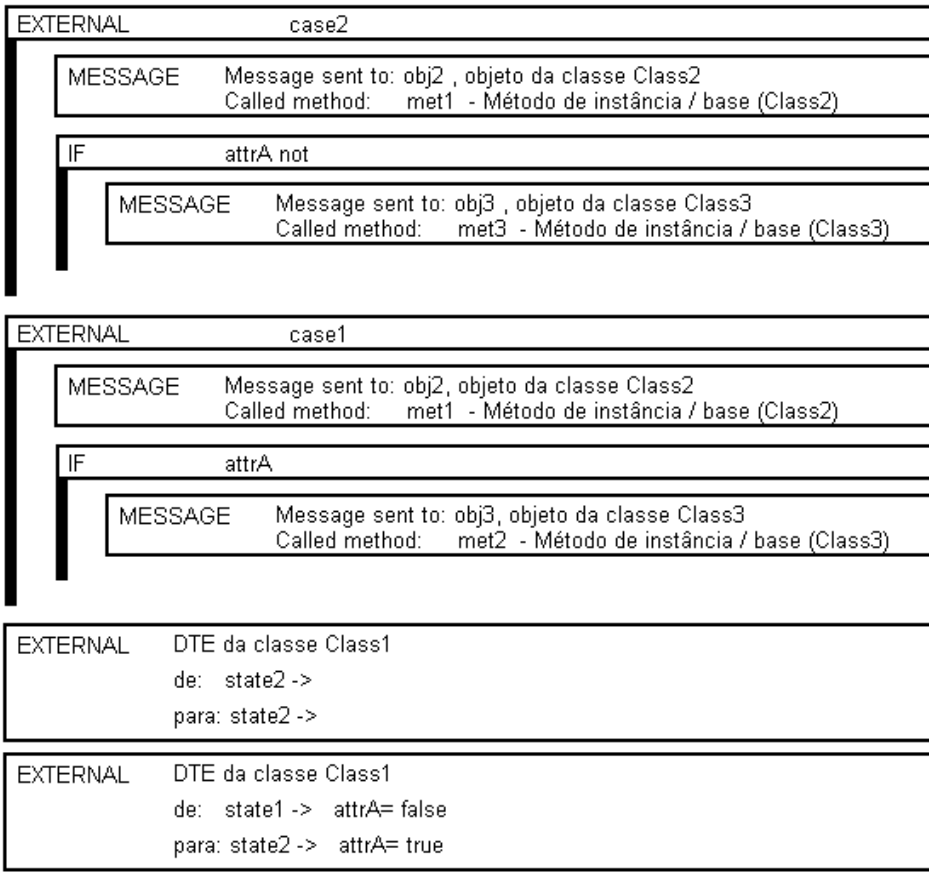

FIGURA 6.27 - Comandos external inseridos automaticamente pela ferramenta de construção de corpo de métodos do ambiente SEA no diagrama de corpo de método do método metA

A figura 6.25 apresenta dois diagramas de seqüência de uma especificação hipotética e a figura 6.26 apresenta o diagrama de transição de estados da classe *Class1*, da mesma especificação. A partir da análise destes diagramas é gerado pela ferramenta de geração de comandos external o conjunto de comandos apresentado na figura 6.27. No primeiro comando *ExternalSequenceDiagram*, obtido a partir da análise do segundo diagrama de seqüência da figura 6.25, observam-se dois comandos *message*, estando o segundo embutido em um comando *if* - os dois comandos *message* referenciam os métodos *met1* e *met3*, nesta ordem, assim como as mensagens do diagrama analisado. Nos dois últimos comandos *ExternalStateTransitionDiagram*, obtidos a partir da análise do diagrama de transição de estados da figura 6.26, observa-se que na execução do

método modelado, *metA*, é possível que o valor do atributo *attrA* mude de *false* para *true*.

A estrutura gerada pela ferramenta - apresentada na figura 6.27 - é usada para a construção do diagrama de corpo de método do método modelado. Comandos contidos nos comandos *ExternalSequenceDiagram* podem ser transferidos e os comandos external descartados. A figura 6.28 apresenta um diagrama de corpo de método para o método metA, consistente com os diagramas das figuras 6.25 e 6.26.

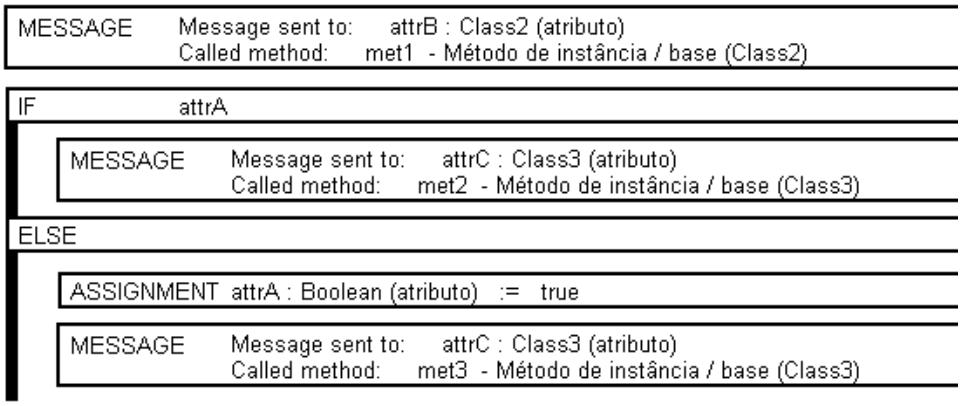

FIGURA 6.28 - Diagrama de corpo de método do método metA

6.5.7 Suporte Automatizado para a Alteração de Frameworks, através de Transferência de Trechos de Especificação

Um framework é uma abstração de um domínio que pode demandar modificações, na medida em que novos conhecimentos do domínio modelado são adquiridos. Ao longo do desenvolvimento de artefatos de software a partir da extensão da estrutura do framework, pode-se verificar que características do domínio não são previstas pelo framework. Assim, o desenvolvimento de artefatos de software sob um framework pode constituir um meio de aquisição de novos conhecimentos do domínio modelado pelo framework, que pode indicar a necessidade de modificar o framework usado. Por isso, um framework só pode ser considerado estável após suportar o desenvolvimento de diferentes tipos de aplicações [PRE 94]. No experimento de desenvolvimento e uso do framework FraG, por exemplo, observou-se a necessidade de modificar o framework ao longo do desenvolvimento de novos jogos, para capacitá-lo a ser uma abstração geral, aplicável a todos os jogos tratados.

Seja *X* a especificação de um framework e *Y*, a especificação de um artefato de software produzido sob o framework *X*. Uma funcionalidade requerida para o tratamento de frameworks em um ambiente consiste na capacidade de transferir classes da especificação *Y* para a especificação do framework *X*. As classes de um framework representam as informações gerais do domínio modelado, reusáveis por diferentes artefatos desenvolvidos sob o framework. Quando é observada a necessidade de transferir uma classe de um artefato de software desenvolvido sob um framework para este framework, está-se considerando que a classe corresponde a um conceito geral do domínio (não tratado quando o framework foi desenvolvido) e que pode ser reusado em outros artefatos de software.

A transferência de uma classe de uma especificação para outra (no caso, para a especificação de um framework) envolve a transferência de conceitos associados, como atributos, métodos, parâmetros associados aos métodos, e de modelos, como diagramas

de transição de estados associados à classe transferida e diagramas de corpo de método associados aos métodos transferidos, bem como envolve o restabelecimento das associações de sustentação e referência inicialmente existentes. O ambiente SEA possui uma ferramenta de transferência de classe para framework que automatiza este procedimento, cuja principal vantagem é evitar a introdução de erros no procedimento de transferência, que envolve um grande número de ações de edição.

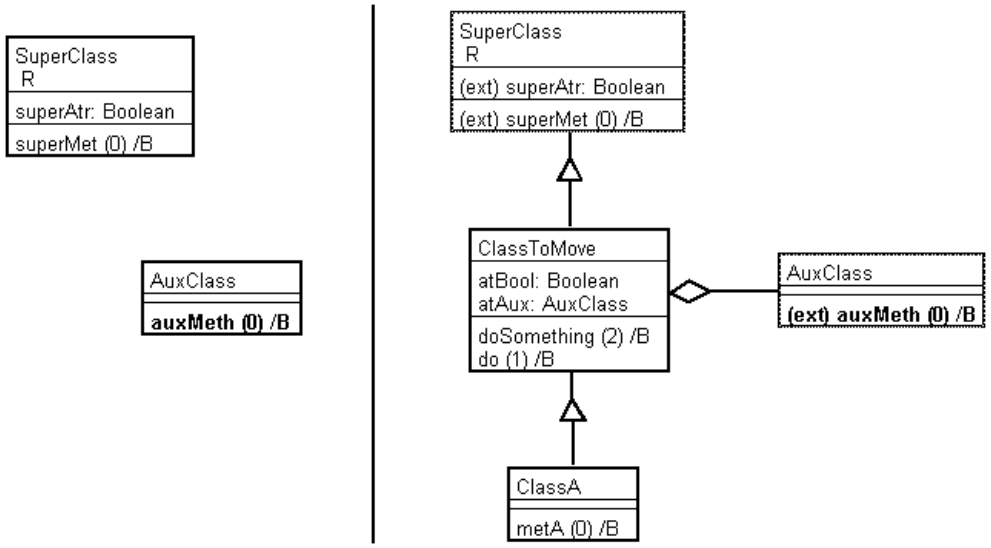

FIGURA 6.29 - Classes de um framework e de uma aplicação sob este framework

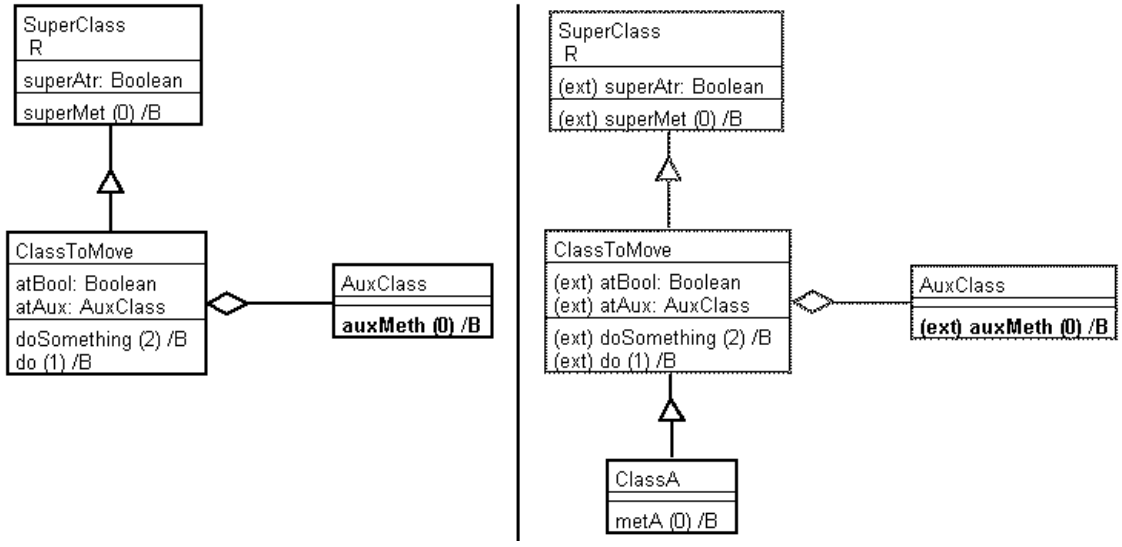

FIGURA 6.30 - Resultado da transferência de uma classe de uma aplicação para um framework

As figuras 6.29 e 6.30 ilustram um procedimento de transferência de uma classe originalmente pertencente à especificação de uma aplicação para sua especificação-mãe, isto é, para a especificação de um framework. Observa-se na especificação da aplicação da figura 6.29 (à direita) a presença de duas classes externas, *SuperClass* e *AuxClass*, definidas na especificação do framework (à esquerda). A figura 6.30 apresenta os mesmos modelos, porém alterados pela transferência da classe *ClassToMove* da especificação da aplicação para a especificação do framework. A classe transferida passa a fazer parte da especificação do framework e passa a ser classificada como externa na

especificação da aplicação - o mesmo ocorrendo com conceitos associados a esta classe. Além da alteração visível nos diagramas de classes, a transferência da classe *ClassToMove* provoca os seguintes efeitos:

- os diagramas de transição de estados associados à classe (o diagrama de transição de estados da classe e os diagramas que o refinem), assim com todos os conceitos por eles referenciados, são excluídos da especificação da aplicação e incluídos na especificação do framework.
- os diagramas de corpo de método associados aos métodos transferidos para a especificação do framework (isto é, os métodos associados à classe transferida), bem como os conceitos por eles referenciados, são excluídos da especificação da aplicação e incluídos na especificação do framework.

O procedimento de transferência é precedido por um teste de viabilidade que condiciona a transferência aos seguintes requisitos:

- a classe a ser transferida e os conceitos a ela associados não podem referenciar classes definidas na especificação da aplicação (a especificação da aplicação conhece a especificação do framework que a originou, mas a recíproca não é verdadeira);
- a transferência da classe não pode inserir inconsistências na especificação do framework, bem como não pode torná-la incopleta. Os mecanismos de análise de consistência do ambiente são utilizados para verificar a presença de inconsistências ou partes incompletas nas estruturas a serem transferidas, antes do procedimento de transferência.

6.5.8 Inserção Semi-Automática de Padrões de Projeto em Especificações OO

Conforme já apresentado, padrões de projeto são estruturas de classes que correspondem a soluções para problemas de projeto, a serem inseridas na estrutura de classes de artefatos de software em desenvolvimento, de modo que estes artefatos incorporem as soluções propostas nos padrões. No ambiente SEA o desenvolvimento de artefatos de software sempre corresponde ao desenvolvimento de especificação de projeto para posterior geração de código. Assim, a inserção de padrões de projeto corresponde a incorporar as estruturas previstas na definição dos padrões a especificações sob edição. SEA possui uma ferramenta de inserção semi-automática de padrões que permite selecionar uma especificação de padrão de projeto na biblioteca de padrões do ambiente e inseri-la em uma especificação em desenvolvimento.

O suporte ao tratamento de padrões do ambiente SEA automatiza etapas do processo de inserção de padrões de projeto em um artefato de software em desenvolvimento - o que além diminuir o esforço de desenvolvimento demandado, diminui a possibilidade de introdução de erros durante o processo - e permite a usuários navegar através de especificações de padrões de projeto - podendo assim, também auxiliar no entendimento da finalidade e da estrutura de padrões de projeto. Os padrões de projeto contidos na biblioteca do ambiente correspondem a padrões do catálogo de Gamma [GAM 94]. SEA suporta o desenvolvimento de novas especificações de padrões de projeto e sua inclusão na biblioteca do ambiente.

No ambiente SEA uma especificação de padrão é construída a partir de quatro técnicas de modelagem: diagrama de classes, diagrama de transição de estados, diagrama de seqüência e diagrama de corpo de método. Em relação ao conjunto de técnicas de modelagem para especificação de aplicações, frameworks e componentes, foram excluídas as técnicas diagrama de casos de uso e diagrama de atividades. Assim, foi definido que especificações não incluem a definição de casos de uso. A razão disto é que

um padrão de projeto não é voltado a definir novas situações de processamento para um artefato de software, mas a solucionar problemas de projeto na modelagem das situações de processamento identificadas. Cabe salientar que uma especificação de padrão não é a especificação de um artefato de software acabado, mas um trecho de especificação a ser inserido em uma especificação de aplicação, framework ou componente.

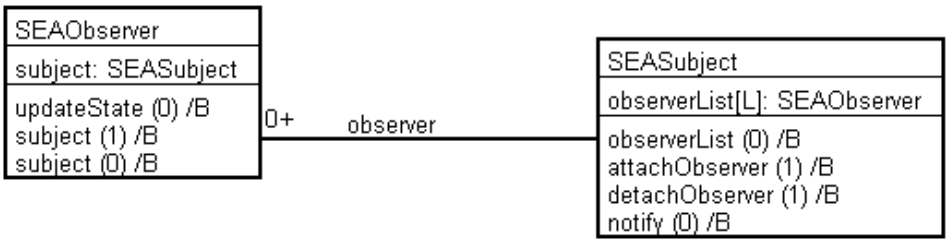

FIGURA 6.31 - Diagrama de classes do padrão de projeto Observer

As figuras 6.31, 6.32 e 6.33 apresentam modelos da especificação do padrão de projeto *Observer*, contido na biblioteca de padrões do ambiente SEA. Na figura 6.31 observam-se as duas classes definidas no padrão, *SEAObserver* e *SEASubject*. A figura 6.32 apresenta o único diagrama de seqüência da especificação do padrão, que descreve a dinâmica estabelecida no padrão: a execução de um método do sujeito (indefinido na especificação do padrão) faz com que o sujeito invoque seu método *notify*; a execução deste método faz com que seja invocado o método *updateState* do observador. A figura 6.33 apresenta o diagrama de corpo de método do método *notify*.

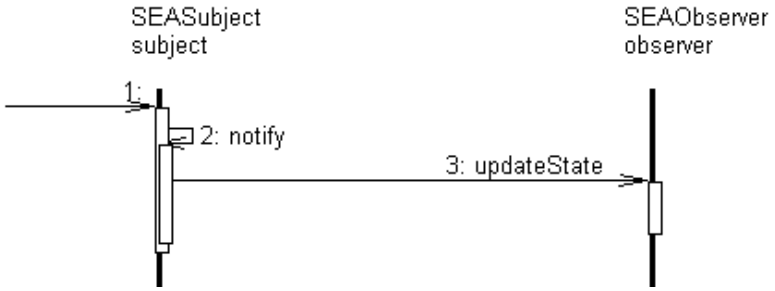

FIGURA 6.32 - Diagrama de seqüência do padrão de projeto Observer

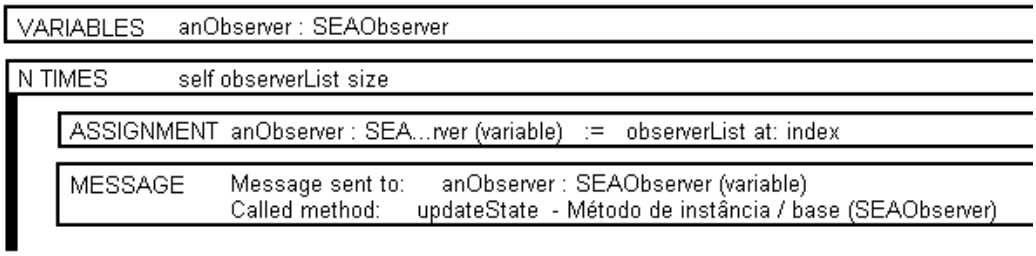

FIGURA 6.33 - Diagrama de corpo de método do método notify da classe SEASubject

O procedimento de inserção de padrões de projeto em uma especificação é ilustrado a seguir, utilizando o padrão *Singleton*. Este padrão possui apenas uma classe e sua finalidade é garantir que esta classe apresente apenas uma instância. Carregada a ferramenta de inserção de padrões, a primeira etapa consiste em selecionar um padrão da biblioteca - *Singleton*, neste caso. A figura 6.34 apresenta a interface da ferramenta com este padrão selecionado.

A segunda etapa consiste em associar as classes do padrão de projeto às classes da especificação em que o padrão vai ser inserido, isto é, definir que classes da especificação assumirão as responsabilidades atribuídas às classes do padrão de projeto. Neste caso, há duas opções possíveis para cada classe do padrão: associar a classe a uma classe existente ou criar uma nova classe na especificação. A figura 6.35 apresenta um diagrama de classes de uma especificação hipotética e a figura 6.36 apresenta a opção de associação da classe *Singleton* do padrão selecionado à classe Class1 dessa especificação.

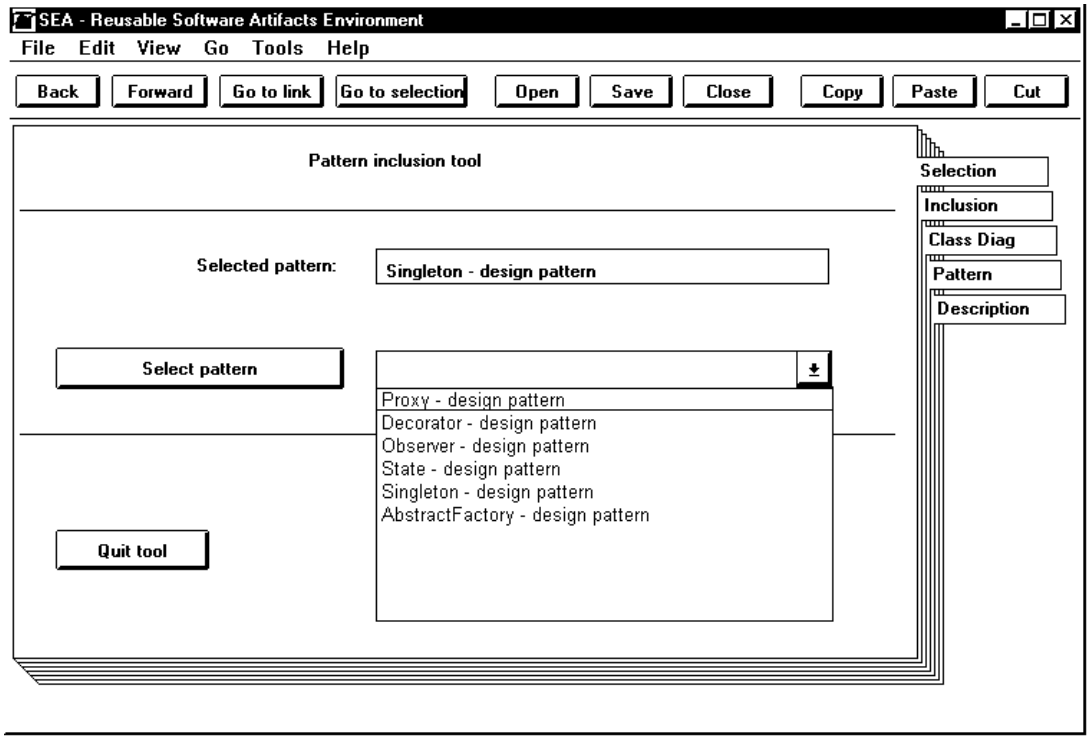

FIGURA 6.34 - Interface da ferramenta de inserção de padrões do ambiente SEA

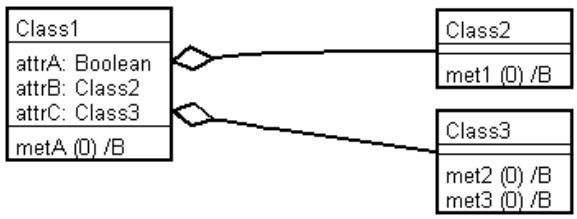

FIGURA 6.35 - Diagrama de classes de uma especificação hipotética

A etapa seguinte, procedida de forma automática, corresponde à inserção do padrão selecionado na especificação tratada, considerando as opções de associação de classes definidas na etapa anterior. Antes de proceder esta inclusão (opção *include pattern*) é possível testar a possibilidade de inclusão (opção *test inclusion*). No caso da opção de proceder a inclusão, a análise da possibilidade de inserção do padrão também é procedida e a inclusão só ocorre se a análise não indicar impedimento. A figura 6.37 apresenta a classe *Singleton* do padrão de mesmo nome e o diagrama de classes da figura 6.35, porém modificado pela inserção do padrão de projeto *Singleton*. Além da inserção de classes, métodos e atributos (e conceitos relacionados, como tipos e parâmetros), a

ação da ferramenta de inserção de padrões também insere na especificação em edição os diagramas de seqüência, de transição de estados e de corpo de método contidos na especificação do padrão. No caso do padrão *Singleton* do exemplo, são inseridos os diagramas de corpo de método dos métodos *new* e *initialize*.

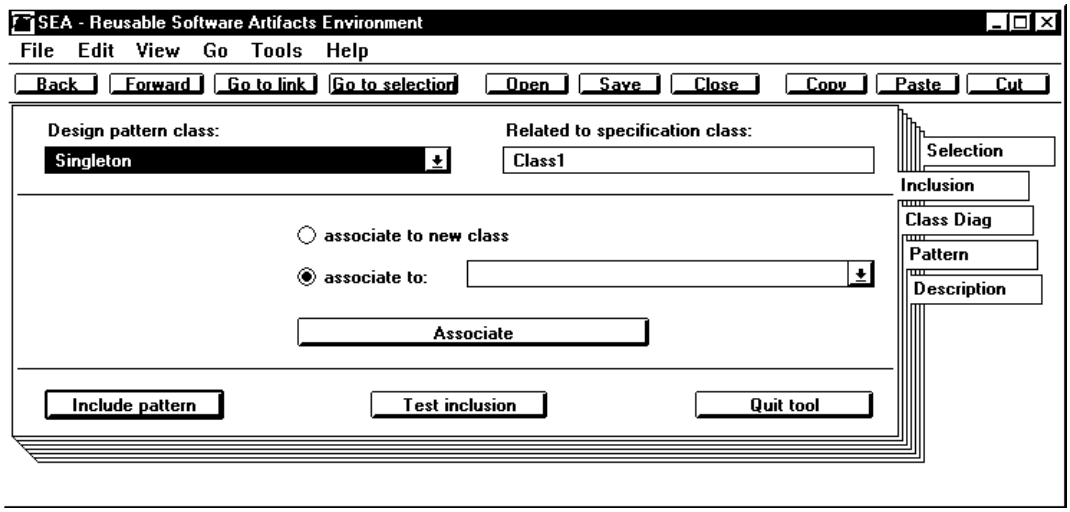

FIGURA 6.36 - Associação de classe de padrão de projeto à classe da especificação tratada

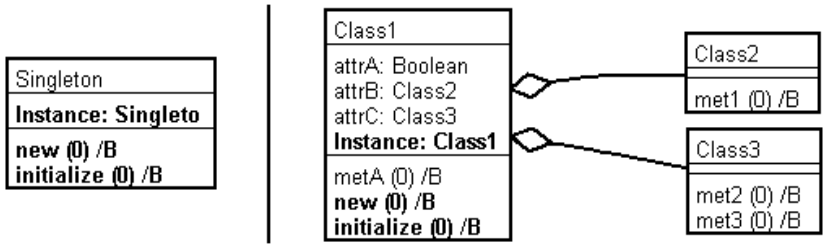

FIGURA 6.37 - Classe Singleton e diagrama de classes após inserção do padrão de projeto Singleton

A ferramenta de teste de inserção *default* do ambiente SEA busca situações de conflito, como por exemplo coincidência de nome entre atributos da classe do padrão e da classe da especificação associada (incluídos os atributos das suas superclasses e subclasses). Um padrão pode usar esta ferramenta ou pode demandar uma ferramenta de teste específica. O padrão *Singleton*, por exemplo, possui uma ferramenta de teste específica que acrescenta a necessidade de haver uma associação entre a classe *Singleton* e uma classe da especificação (impedindo a opção de criação de nova classe na inserção desse padrão)[56].

O padrão *Singleton*, composto por um diagrama de classes com a classe *Singleton* e dois diagramas de corpo de método, insere um procedimento completo na especificação sob edição, isto é, se o padrão for inserido em uma especificação consistente, após a introdução do padrão a especificação continuará semanticamente

---

56 Na implementação do protótipo, a ferramenta de teste de inserção *default* está implementada na classe *PatternTestTool* e a ferramenta de teste associada ao padrão *Singleton*, *PatternTestSingleton*, é uma subclasse de *PatternTestTool* que adiciona um teste: verifica se ocorreu associação entre a classe *Singleton* e uma classe da especificação sob edição. Ferramentas de teste específicas, dos diferentes padrões de projeto, são subclasses de *PatternTestTool*. Cabe salientar que nem todo padrão demanda uma ferramenta de teste específica, como é o caso do padrão de projeto *Observer*, por exemplo - que usa a ferramenta de teste de inserção *default*.

consistente. Manter a consistência da especificação afetada não é o caso geral do procedimento de inserção de padrões. Isto não quer dizer que a inserção vá introduzir erros na especificação sob edição (isto é impedido pela verificação de consistência procedida antes de incluir uma especificação na biblioteca de padrões e pela avaliação de viabilidade da inserção), mas que podem ser introduzidas estruturas incompletas a serem tratadas ao longo do desenvolvimento da especificação. Um exemplo de introdução de estrutura incompleta pode ser observado na introdução do padrão de projeto *Observer* em uma especificação.

Na inserção do padrão de projeto *Observer*, cuja especificação está parcialmente apresentada nas figuras 6.31, 6.32 e 6.33, são introduzidas as seguintes estruturas a serem completadas:

- O método *updateState*, associado à classe *SEAObserver*, não possui diagrama de corpo de método. Este método é o método invocado nos observadores quando ocorre alguma alteração de estado do sujeito que deva ser notificada. A razão para isto é que o padrão introduz a lógica da comunicação entre sujeito e observadores, mas o tratamento da situação depende de cada caso em que o padrão é utilizado. Assim, a definição do algoritmo do tratamento, que corresponde ao método *updateState*, está fora do escopo do padrão *Observer* - o algoritmo deve ser definido em cada especificação em que o padrão for introduzido.
- O diagrama de seqüência que descreve o procedimento de notificação (apresentado na figura 6.32) não possui um *link* semântico que o aponte e nem tem definido o método associado à mensagem de ordem 1. O método *notify* é o método da classe *SEASubject* invocado pelo próprio sujeito na ocorrência de uma alteração de estado que deva ser notificada aos observadores. Semelhante à situação anterior, está fora do escopo do padrão definir as situações que demandam notificação - em cada especificação em que o padrão for introduzido deve ser definido que método invoca *notify*.

Nos dois padrões de projeto mencionados nos exemplos anteriores (*Singleton* e *Observer*), definidas as associações entre classes do padrão e classes da especificação sob edição e verificada a possibilidade de inserção do padrão, o procedimento de inserção consiste em reproduzir a estrutura do padrão na especificação tratada e fundir cada par de classes associadas (uma do padrão e outra da especificação) - no caso da opção de criação de nova classe na especificação não há a necessidade de fusão. A reprodução da estrutura do padrão na especificação consiste em:

- copiar e colar todas as classes e associações entre classes do padrão no diagrama de classes sob edição;
- reproduzir os demais modelos do padrão (diagramas de transição de estados associados às classes, diagramas de corpo de método associados aos métodos e diagramas de seqüência).

A inserção de alguns padrões demanda procedimentos adicionais, além dos acima descritos, como ocorre por exemplo, com o padrão de projeto *Proxy*. A figura 6.38 apresenta o diagrama de classes deste padrão. Na inserção do padrão *Proxy*, a classe *SEASubject* deve ser associada à classe para a qual se deseja criar um representante (*proxy*) e a classe *SEAProxy*, a uma classe sem métodos e sem atributos, subclasse da classe associada a *SEASubject*, ou a nenhuma classe (esta condição é testada pela ferramenta de teste de inserção associada ao padrão). A figura 6.39 apresenta a janela de edição de uma classe, *Class3*, que apresenta dois métodos, *met2* e *met3*. A figura 6.40 apresenta o diagrama de classes resultante da inserção do padrão *Proxy*, com a classe *Class3* associada à classe *SEASubject* do padrão. A figura 6.41 apresenta os diagramas

de corpo de método dos métodos *met2* e *met3* criados na classe *SEAProxy* no processo de inserção do padrão e que sobrepõem os métodos homônimos da classe *Class3*.

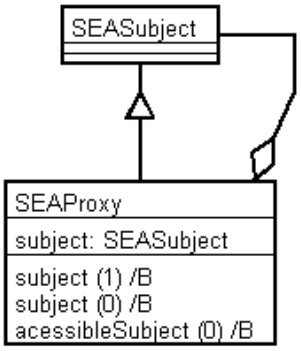

FIGURA 6.38 - Diagrama de classes do padrão de projeto Proxy

Comparando os diagramas das figuras 6.38 e 6.40 observa-se que a classe *SEAProxy* foi copiada na especificação em que o padrão foi inserido (sem fusão) e foram associados a esta classe dois métodos que sobrepõem os métodos definidos na superclasse. Analisando os diagramas de corpo de método destes dois métodos verifica-se que o modelo correspondente ao método *met2*, um método que não produz retorno, corresponde a invocar método homônimo do atributo *subject*. O algoritmo do método *met3*, que retorna um objeto, também invoca método homônimo do atributo *subject*, porém, retorna ao emissor o objeto retornado pela execução do método no objeto referenciado pelo atributo. Esta sobreposição dos métodos de uma classe para a qual se cria um representante (inclusive dos métodos herdados) na inserção do padrão de projeto *Proxy* é feita por uma ferramenta de adaptação da estrutura do padrão associada ao padrão. A ferramenta atua após o procedimento de colagem e fusão de classes e reprodução dos demais modelos do padrão. No caso do padrão *Proxy*, a ação da ferramenta consiste em sobrepor os métodos da classe representada, definindo algoritmos semelhantes aos apresentados na figura 6.41.

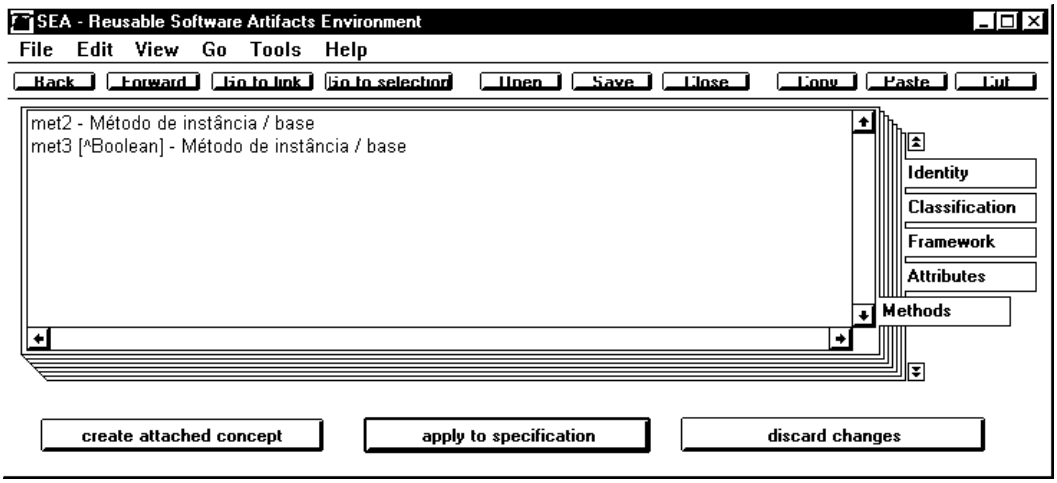

FIGURA 6.39 - Janela de edição de uma classe, Class3, com destaque para sua lista de métodos

O ambiente possui uma ferramenta de adaptação de estrutura de padrão *default*, que não executa qualquer ação após a colagem e fusão de classes e reprodução de modelos. Esta ferramenta é associada a padrões que não demandam ações adicionais, como *Singleton* e Observer. Um padrão que demande uma ação específica, como no

caso do padrão Proxy, deve ter associada uma ferramenta de adaptação de estrutura de padrão própria[57], com ações de edição específicas.

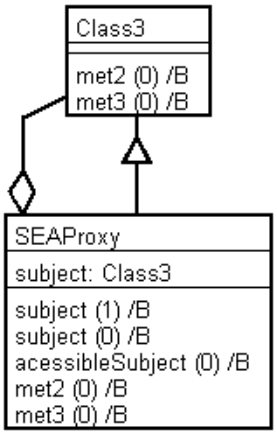

FIGURA 6.40 - Inserção do padrão de projeto Proxy, com a classe SEASubject associada à classe Class3

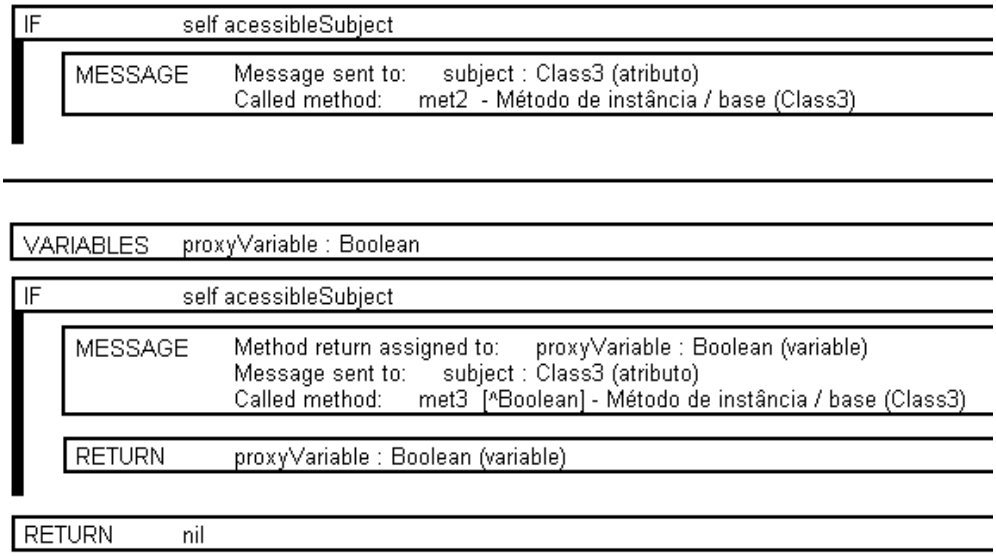

FIGURA 6.41 - Diagramas de corpo de método dos métodos met2 e met3, criados na classe SEAProxy

6.5.9 Consistência de Especificações OO no Ambiente SEA

Uma especificação OO do ambiente SEA apresenta semântica definida, conforme descrito no anexo 1, em que é apresentada a gramática de atributos para este tipo de especificação. Esta característica possibilita a verificação de consistência de especificações OO (especificações de aplicações, frameworks ou componentes), bem como a conversão de uma especificação em outras linguagens - particularmente importante, a possibilidade de conversão em linguagens de programação. Esta

---

57 Na implementação do protótipo a ferramenta de adaptação de estrutura de padrão *default* está implementada na classe *PatternAdapterTool* e a ferramenta de adaptação associada ao padrão *Proxy*, *ProxyAdapterSingleton*, é uma subclasse de *PatternAdapterTool*. Ferramentas de adaptação específicas, dos diferentes padrões de projeto, são subclasses de *PatternAdapterTool*. Como ocorre com as ferramentas de teste de implantação de padrão, nem todo padrão demanda uma ferramenta de adaptação específica, como é o caso do padrão de projeto *Observer*, por exemplo.

possibilidade de obtenção de estrutura de especificação com semântica definida é conferida pela estrutura genérica de especificação definida no framework OCEAN, apresentada no item 6.2, e não é exclusiva de especificações OO.

A geração de código no ambiente SEA só é possível a partir de especificações consistentes. Uma especificação é consistente se todos os requisitos estabelecidos na sua gramática são verificados, isto é, se todos os elementos da estrutura são reconhecidos (modelos e conceitos com estruturas completas) e se o predicado da especificação, avaliado a partir do conjunto de regras semânticas associadas às produções, assumir o valor *true*.

A consistência de uma especificação OO é garantida pelo seguinte conjunto de recursos do ambiente:

- **ações de edição bloqueadas**: toda ação de edição que envolva criação ou modificação de elemento de especificação é submetida a avaliação de conflito, que pode impedir sua consumação[58]. Alguns exemplos de ações bloqueadas são ciclo de herança, criação de classes com coincidência de nome e criação de atributo com coincidência de nome na classe e na cadeia hierárquica de herança de que a classe participa.
- **ações de edição impossibilitadas**: certas ações não são possíveis de realizar em função das interfaces de edição do ambiente SEA. Não é possível definir uma transição de estados sem que os estados envolvidos pertençam à especificação, por exemplo, porque para criar uma transição em um diagrama de transição de estados é necessário selecionar os dois estados envolvidos no conjunto dos estados existentes. Assim, o requisito dos estados envolvidos em uma transição de estados pertencerem à especificação sempre se verifica (isto se verifica também na remoção de um estado, já que isto provoca a remoção das transições em que ele esteja envolvido, em função da associação de sustentação que existe entre estes elementos). De forma semelhante, só é possível referenciar classes existentes. Isto influencia a criação de métodos, atributos, agregações, associações binárias, heranças, diagramas de transição de estados, bem como a alteração de vários conceitos, como objeto de diagrama de seqüência, comandos de diagrama de corpo de método e outros.
- **ferramentas de análise**: o ambiente apresenta sete ferramentas de análise para verificação de consistência de especificações OO - uma ferramenta de análise para cada um dos seis tipos de modelo e uma ferramenta de análise do conjunto da especificação. As ferramentas de análise específicas para cada tipo de modelo se atêm à avaliação dos tipos de conceito tratados no contexto do modelo e à avaliação dos inter-relacionamentos entre os conceitos referenciados - por exemplo, se foi procedida a definição do método referenciado por cada mensagem, na análise de um diagrama de seqüência. O analisador da especificação, além de invocar a atuação dos analisadores de modelos, verifica aspectos que extrapolam os limites dos modelos,

---

58 A avaliação de conflito se restringe a avaliar o elemento de especificação tratado comparando-o com os outros elementos de mesmo tipo, contidos na especificação. Assim, a tentativa de produzir um ciclo de herança é impedida pela comparação da ocorrência de herança que se tenta incluir na especificação com as demais já incluídas. Por outro lado, a criação de uma herança relacionando duas classes que tenham atributos homônimos não é impedida (o que possibilita a inserção uma inconsistência na especificação tratada). Inconsistências deste tipo, isto é, que envolvam tipos de elementos de especificação diferentes do tipo do elemento tratado, não são avaliadas em tempo de edição - são porém, constatadas no procedimento de avaliação de consistência da especificação. A avaliação de consistência do conjunto da especificação em tempo de edição não foi adotada para não prejudicar o desempenho do ambiente.

como a verificação da existência de diagrama de corpo de método associado a cada método concreto e que não seja classificado como externo, análise da existência e da consistência da interligação entre casos de uso e diagramas de seqüência e outros.

As ferramentas de análise podem produzir dois tipos de ação: descrição e correção semi-automática dos erros encontrados. No procedimento de análise é gerado um relatório com todos os erros encontrados (que podem corresponder a estruturas incompletas ou inconsistentes). A correção de erros é procedida por ferramentas para tratamento de erros específicos, invocadas pelas ferramentas de análise quando são verificados os erros por elas tratados. O Ambiente SEA possui ferramentas deste tipo para tratamento de alguns tipos de erro, como incompatibilidade entre o conjunto de mensagens de um diagrama de seqüência e o conjunto de comandos *message* de um diagrama de corpo de método. Quando uma ferramenta de tratamento de erro invocada pelo ambiente propõe uma ou mais soluções para um problema encontrado, é possível optar pelo não-procedimento da correção pela ferramenta.

No caso da verificação de incompatibilidade entre o conjunto de mensagens de um diagrama de seqüência e o conjunto de comandos *message* de um diagrama de corpo de método, por exemplo, a ferramenta de tratamento do erro interage com o usuário para a escolha de uma das soluções possíveis para um dos seguintes problemas:

- falta de mensagem em diagrama de seqüência (considerando o conjunto de comandos *message* em diagrama de corpo de método) - criação de mensagem no diagrama de seqüência ou remoção do comando *message* responsável pela inconsistência do diagrama de corpo de método;
- falta de comando *message* em diagrama de corpo de método (considerando o conjunto de mensagens em diagrama de seqüência) - criação de comando *message* no diagrama de corpo de método ou remoção da mensagem responsável pela inconsistência do diagrama de seqüência;
- condicionador de mensagem de diagrama de seqüência sem comando equivalente no diagrama de corpo de método - criação de comando *if*, *while*, *repeat*, ou *nTimes* (de acordo com o tipo de condicionador de mensagem identificado no diagrama de seqüência) e transferência de comando *message* para esse comando.

6.5.10 Geração de Código no Ambiente SEA

O ambiente SEA dispõe de uma ferramenta capaz de gerar código Smalltalk executável, a partir de especificações consistentes de frameworks, aplicações ou componentes. A ferramenta de geração de código vem sendo testada com exemplos de especificação, inclusive a especificação do framework FraG[59] - o código gerado neste caso não necessitou de qualquer alteração para execução.

A geração de código consiste em um procedimento de conversão de formato para um conjunto de informações. No caso da ferramenta do ambiente SEA, a descrição de

---

[59] O desenvolvimento do framework FraG, como mencionado nos capítulos anteriores, precedeu o desenvolvimento do ambiente SEA - constituiu um experimento de desenvolvimento de um framework e posterior uso deste framework para apoio ao desenvolvimento de aplicações, para avaliação de recursos de desenvolvimento disponíveis e levantamento de requisitos para suporte ao desenvolvimento e uso de frameworks (contribuindo significativamente para o desenvolvimento do ambiente SEA). Uma vez disponíveis os recursos ora descritos do ambiente SEA, como um dos experimentos de uso do ambiente, foi produzida uma especificação de projeto para o framework FraG, a partir da qual foi produzida pela ferramenta de geração de código, a implementação acima citada (testada em conjunto com o código das aplicações desenvolvidas sob a primeira implementação do FraG).

uma estrutura de classes é convertida do formato de especificação do ambiente para o de uma linguagem de programação orientada a objetos, Smalltalk, no caso. O requisito fundamental para que o ambiente produza código que não necessite ser complementado é que todas as informações necessárias à geração de código executável estejam contidas na especificação. Como verificado por Mellor quando analisa as possibilidades de geração de código a partir de especificações produzidas em UML (UML sem modificações, neste caso), a inexistência da definição dos algoritmos dos métodos em especificações em UML impossibilita a geração de código executável (ou compilável) diretamente a partir da conversão das especificações - demandando a inclusão posterior do corpo dos métodos [MEL 99]. No caso da geração de código a partir de uma especificação UML, como feito por alguns ambientes existentes, o que ocorre é uma tradução dos diagramas de classes, isto é, são gerados *esqueletos* das classes, com atributos tipados e assinaturas dos métodos.

No caso do procedimento de tradução da ferramenta de geração de código do ambiente SEA, o código é composto a partir das informações contidas nos diagramas de classes e nos diagramas de corpo de método. Os demais diagramas que modelam a dinâmica do sistema, cooperam para a definição do algoritmo dos métodos expressos nos diagramas de corpo de método - casos de uso são refinados em diagramas de seqüência, evolução de estados de instâncias é descrita nos diagramas de transição de estados; diagramas de seqüência e de transição de estados são analisados para a construção dos diagramas de corpo de método (no ambiente SEA, de forma automatizada).

É possível converter uma especificação para outras linguagens de programação a partir da criação de novas ferramentas de geração de código, com outras linguagens alvo. Há porém, duas restrições em relação à estrutura de especificação atualmente implementada. A primeira é que a opção de uso de comando *task*, em pelo menos um diagrama de corpo de método da especificação, restringe a geração de código a apenas uma linguagem de programação. Além disso, a análise do conteúdo do comando *task*, tratado pelo ambiente como um *string*, é protelada para a compilação - assim, o ambiente não garante a sua consistência sintática ou semântica. A segunda restrição envolve os comandos de diagrama de corpo de método que embutem outros comandos. Como as expressões associadas a estes comandos (predicados ou funções) são tratadas como strings, como na situação anterior, o ambiente não garante a sua consistência sintática ou semântica.

6.6 Suporte ao Uso de Frameworks no Ambiente SEA

Um dos problemas associados à abordagem de frameworks é a complexidade de uso. *Cookbooks* ativos, como já discutido, são mecanismos que estabelecem os passos do desenvolvimento de um artefato de software a partir de um framework, e que podem automatizar tarefas do processo de desenvolvimento. *Cookbooks*, por outro lado, são limitados pela incapacidade de descrever todas as possibilidades de criação de artefatos de software a partir de um framework - o que força a busca de outras fontes de informação para entender o projeto do framework usado e poder solucionar problemas de desenvolvimento. No ambiente SEA foi adotada a abordagem de *cookbook* ativo, a partir do estabelecimento dos seguintes requisitos para um mecanismo de suporte ao uso de frameworks:

- **direcionar as ações dos usuários de frameworks** - o mecanismo de auxílio ao desenvolvimento de artefatos de software a partir de um framework (aplicações, componentes ou frameworks) deve dirigir as ações dos usuários. O direcionamento de passos diminui o esforço para descobrir o que deve ser feito. Um hiperdocumento constitui um dispositivo adequado de condução das atividades do usuário de um framework, porque permite que diferentes caminhos de desenvolvimento sejam seguidos, de acordo com as necessidades específicas do artefato de software em desenvolvimento.
- **reduzir o esforço para entender o projeto do framework** - quando as necessidades do usuário não puderem ser satisfeitas pelo conteúdo do hiperdocumento de direcionamento de navegação, o mecanismo de suporte deve permitir acesso a descrições de projeto de alto nível. Análise de código fonte não deve ser a única alternativa para suprir as deficiências do hiperdocumento.
- **liberar o usuário de atividades de baixo nível** - ações obrigatórias, como a criação de alguns elementos de especificação, podem ser semi-automatizadas. Por exemplo, se o projeto de um framework estabelece a necessidade de criação de uma subclasse de uma de suas classes, o mecanismo de apoio deve perguntar o nome da nova classe e incluí-la na especificação de projeto em desenvolvimento. Além disto, a tradução de especificações de projeto para código fonte pode ser feita por gerador de código.
- **verificar a satisfação dos requisitos para a geração de aplicações a partir de um framework** - o projeto de um framework estabelece as restrições para o desenvolvimento de artefatos de software sob ele, como a necessidade de produzir subclasses de algumas de suas classes abstratas, ou a sobreposição de métodos abstratos destas classes. Um mecanismo de suporte ao uso de frameworks deve verificar o cumprimento desses requisitos, como parte da tarefa de direcionamento das atividades do usuário.

O suporte ao uso de frameworks do ambiente SEA, que segue os requisitos acima, difere das abordagens existentes, principalmente pela ênfase ao uso de especificações de alto nível, ao invés do manuseio de código, o que pode reduzir o esforço necessário para usar um framework, bem como tornar esta atividade mais amigável.

6.6.1 Produção de *Cookbook* Ativo Vinculado a uma Especificação de Framework

Cookbook ativo no ambiente SEA é um tipo de especificação (diferente do tipo usado para modelar estruturas de classes) que prevê um único tipo de modelo: página de hiperdocumento[60]. Este tipo de modelo pode referenciar quatro tipos de conceito[61] abaixo relacionados.

- *Link* de navegação - *link* que aponta documento, que pode ser página do hiperdocumento ou qualquer outro tipo de documento definido sob o framework OCEAN, como especificação e elemento de especificação. Estes *links* dão suporte à navegação, que não precisa se ater às páginas do hiperdocumento tratado.

---

60 O tipo de especificação *cookbook ativo* é implementado pela classe *SEAPreceptorHyperdocument*, subclasse de *Specification*, assim como *OOSpecification*. O tipo de modelo *página de hiperdocumento* corresponde na implementação do protótipo, à classe *HyperdocumentPage*, subclasse concreta de *ConceptualModel*.

61 Na implementação do protótipo, correspondem a subclasses de *Concept*.

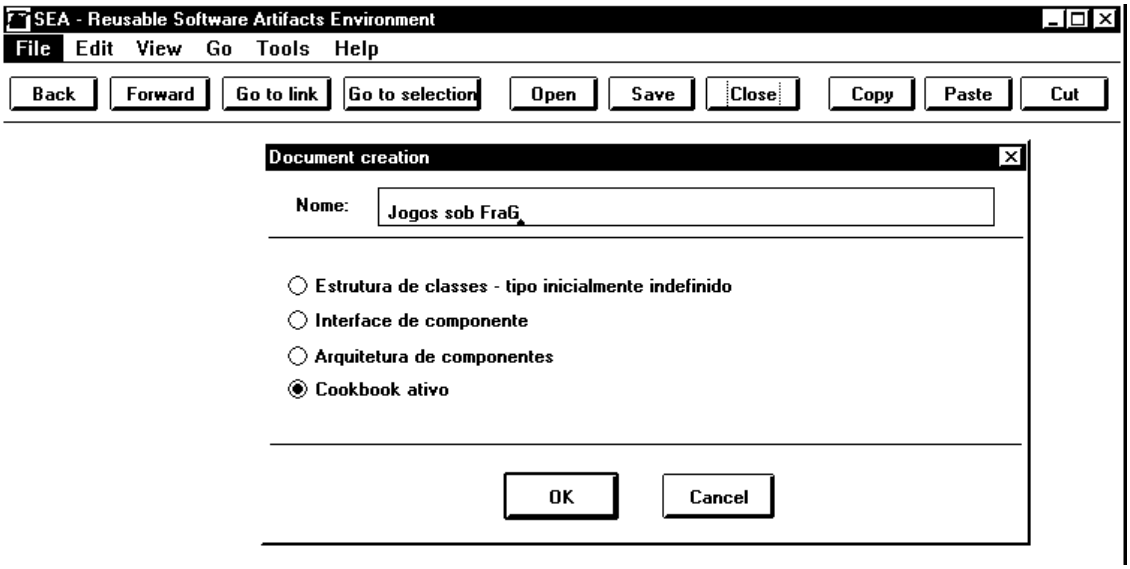

FIGURA 6.42 - Criação de cookbook ativo no ambiente SEA

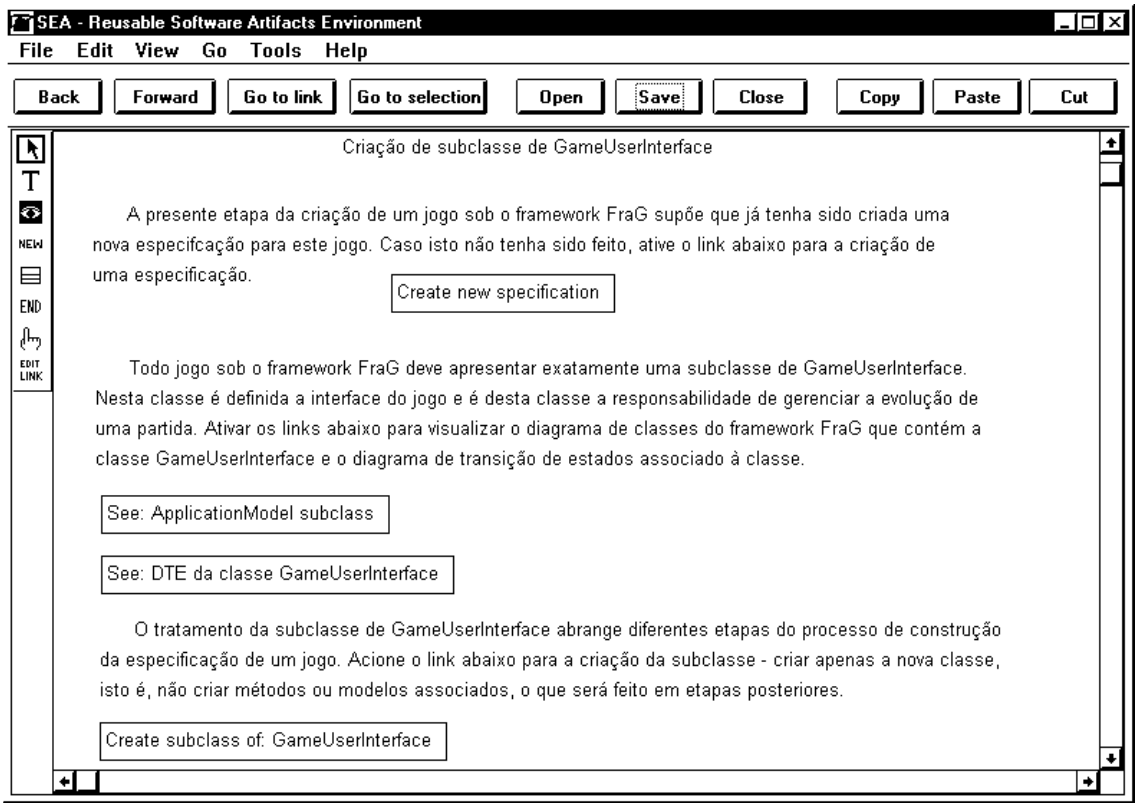

FIGURA 6.43 - Editor de página de hiperdocumento do ambiente SEA, com uma página do cookbook de orientação ao uso do framework FraG

- *Link* ativo de criação de especificação - quando acionado, provoca a criação de uma nova especificação sob o framework associado ao *cookbook*, que passa a ser referenciada pelo *cookbook* (como a especificação que está sendo construída).
- *Link* ativo de tratamento de classe redefinível - associado a uma classe redefinível do framework tratado pelo *cookbook*, é ativado para a criação de subclasse da classe referenciada ou de conceito ou modelo associado a uma subclasse.

- *Link* ativo de conclusão de especificação - para ser acionado quando se conclui o uso do *cookbook* para produzir uma especificação, ativa as últimas ações automáticas de criação de especificação sob um framework, através do uso de *cookbook*.

O ambiente possui um editor de página de hiperdocumento para suportar a criação das páginas de um *cookbook* ativo para um framework. Como ocorre com os outros tipos de modelo desenvolvidos sob o framework OCEAN, uma página de hiperdocumento (modelo) possui uma representação visual associada. Nesta representação visual, além da representação visual associada aos *links* (que são conceitos), são incluídas as instruções textuais que fazem parte de um *cookbook*. A figura 6.42 apresenta a janela de criação de especificação do ambiente SEA, selecionada a opção de criação de *cookbook* ativo. A figura 6.43 apresenta o editor de página de hiperdocumento do ambiente SEA, com uma página do *cookbook* de orientação ao uso do framework FraG. Nesta página observam-se quatro *links* (de cima para baixo): *link* ativo de criação de especificação, dois *links* de navegação e *link* ativo de tratamento de classe redefinível.

O desenvolvimento e o uso de *cookbooks* ativos no ambiente SEA é apoiado por duas ferramentas: o *analisador SEA-Preceptor* e o *construtor SEA-Preceptor*. A primeira é voltada à análise de especificações de frameworks, para levantar os requisitos para um *cookbook* ativo que o descreva; de especificações de artefatos de software desenvolvidos sob um framework, para verificar se estão de acordo com as restrições impostas pelo framework; de *cookbooks* ativos, para avaliar se abrangem todas as possibilidades de criação de elementos de especificação previstas no framework descrito. O construtor SEA-Preceptor é responsável por todas as ações automáticas de criação de especificações ou de elementos de especificação provocadas pelo acionamento dos *links* ativos de um *cookbook*.

O *analisador SEA-Preceptor*, no levantamento de requisitos para a construção de um *cookbook* ativo de apoio ao uso de um framework, busca os requisitos para a geração de artefatos de software sob este framework, em termos de que classes e que métodos devem ou podem ser definidos. Estas informações são buscadas na especificação do framework tratado. Classes de uma especificação de framework do ambiente SEA, apresentam as seguintes informações:

- abstrata ou concreta;
- redefinível ou não (segundo definido no projeto de um framework, só devem ser criadas subclasses das classes redefiníveis);
- essencial ou não (uma classe essencial de um framework deve estar presente em todas as aplicações geradas a partir dele - isto se refere à própria classe ou a subclasses).

O *analisador SEA-Preceptor* busca e relaciona as seguintes situações, que exigem a ação do usuário:

- classes de um framework que são abstratas, redefiníveis e essenciais, e que não apresentam subclasses na especificação do framework, devem originar subclasses - e seus métodos abstratos herdados precisam ser sobrepostos;
- classes de um framework que são abstratas, redefiníveis e essenciais, e que apresentam pelo menos uma subclasse na especificação do framework, não exigem a criação de subclasses - se o usuário decidir pela criação de uma subclasse de algumas destas classes, todos os seus métodos abstratos herdados devem ser sobrepostos;
- classes de um framework que são abstratas, redefiníveis e não-essenciais, e que apresentam ou não subclasses na especificação do framework, também não exigem a

criação de subclasses - a opção pela criação de subclasse exige a sobreposição dos métodos abstratos herdados.

O *analisador SEA-Preceptor*, quando analisa os requisitos para a criação de *cookbook* ativo, também relaciona todos os métodos *template* da especificação de framework analisada e os métodos invocados por estes métodos template (métodos *hook*).

Cada situação acima descrita de criação obrigatória ou não de classe deve corresponder um *link ativo de tratamento de classe redefinível* no hiperdocumento, cujo acionamento provoca as ações de criação anteriormente descritas. No conjunto de páginas que compõe um *cookbook* ativo deve haver pelo menos um *link* ativo para:

- criação de especificação;
- tratamento de cada classe redefinível;
- conclusão de especificação.

O *analisador SEA-Preceptor* avalia *cookbooks* ativos para verificar se esse requisito de inclusão de *links* ativos é cumprido. A avaliação corresponde a uma comparação entre a estrutura de um *cookbook* e a especificação do framework por ele descrita. A avaliação de um *cookbook* ativo também verifica se todas as suas páginas são acessíveis através de *links* de navegação, a partir da página raiz (denominada *main page*).

Um *cookbook* para orientação ao uso de um framework é tradicionalmente produzido de forma artesanal, por especialistas no uso do framework, baseado na experiência destes especialistas, e que não tem como ser avaliado - a não ser a verificação de suas limitações à medida em que é utilizado para apoiar o uso de um framework. As estruturas de especificação definidas no ambiente SEA para suportar a construção de especificações de frameworks e de *cookbooks* ativos introduzem duas inovações à abordagem de *cookbooks*. Primeiro, permitem a introdução da possibilidade de levantar as características de um framework a serem consideradas na construção de um *cookbook* ativo. Segundo, introduzem a possibilidade de avaliar a qualidade do *cookbook* ativo produzido para um framework.

Uma outra aplicação para um hiperdocumento do ambiente SEA, é produzir tutoriais para auxiliar a compreensão do projeto de um artefato de software. Neste caso, o hiperdocumento não apresentaria *links* ativos, não podendo ser caracterizado como *cookbook* ativo. A possibilidade de combinar descrição textual e *links* de navegação (capazes de promover a visualização de modelos e conceitos de uma especificação) em conjunto com os mecanismos de navegação do ambiente SEA, permite produzir hiperdocumentos voltados a descrever aspectos de projeto de uma especificação. A abordagem de intercalação de descrição textual e modelos é utilizada no próximo capítulo para a descrição do framework OCEAN.

6.6.2 Utilização de *Cookbook* Ativo para a Produção de Especificações sob um Framework

Desenvolver um artefato de software sob um framework no ambiente SEA, consiste em criar uma especificação OO, tendo a especificação do framework como especificação-mãe. É possível fazer isto usando os recursos de criação e edição de especificações do ambiente SEA, sem a utilização da orientação de *cookbook* ativo.

O desenvolvimento de um artefato de software sob um framework apoiado por um *cookbook* ativo também utiliza os recursos de edição de especificações do ambiente SEA, porém, parte do desenvolvimento é automatizada pela ação do *construtor SEA*-

Preceptor, ferramenta do ambiente que opera associada ao *cookbook*. Criar uma especificação sob um framework, a partir da orientação de um *cookbook* ativo, consiste em carregar o *cookbook* no ambiente e seguir as instruções apresentadas. A partir da primeira página, um usuário de framework seguirá um caminho de navegação adequado às suas necessidades. A ativação de *links* ativos produz as ações de edição descritas. Por exemplo, ativando o *link* ativo de tratamento de classe associado à classe GameUserInterface contido na figura 6.43 e seguindo a recomendação de não criar métodos ou modelos associados, ocorrerá a inclusão da classe GameUserInterface da especificação do framework na especificação em construção (classificada como classe externa) e a criação da subclasse *TicTacToeInterface* (o nome da classe a ser criada é solicitado ao usuário), conforme ilustrado na figura 6.44. A ativação de outro *link* ativo de tratamento de classe associado à classe *GameBoard* do framework FraG, desta vez permitindo a criação de métodos e modelos associados, produz a subclasse com a sobreposição dos métodos abstratos herdados (conforme apresentado na figura 6.44), a criação de um diagrama de corpo de método vazio[62] associado a cada método criado, a criação opcional de um diagrama de transição de estados vazio associado à classe e a criação opcional de um diagrama de seqüência por método criado, em que o método aparece referenciado por uma mensagem originadora (para descrever a execução do método). Todos estes elementos de especificação são criados de forma automática pela ferramenta *construtor SEA-Preceptor* - no caso dos elementos cuja criação é opcional, o usuário é consultado. A figura 6.45 apresenta o diagrama de seqüência criado para refinar a execução do método *positionActivity* da classe *TicTacToeBoard*.

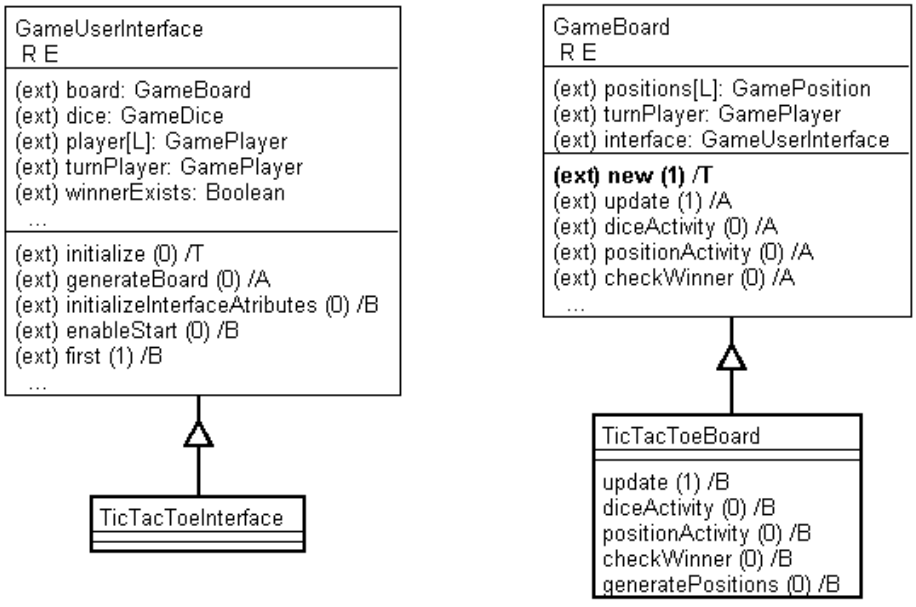

FIGURA 6.44 - Partes de especificação criadas pela ação de um cookbook ativo

Especificações produzidas no ambiente SEA são tratadas como hiperdocumentos, isto é, todos os modelos e conceitos podem incluir *links*, para apontar outras partes da especificação. Assim, quando um *link* de um *cookbook* produz a carga de um modelo - para descrever algum aspecto de projeto, por exemplo - um caminho não previsto no *cookbook* ativo pode ser seguido pelo usuário do framework, a partir da ativação de *links* contidos neste modelo. Ativações do botão de retorno do ambiente

62 Um modelo vazio é um modelo que não referencia conceitos. Todo modelo é criado vazio.

SEA (*back*) permitem a volta ao hiperdocumento. Esta possibilidade de navegação através de especificações de projeto durante o desenvolvimento de novas especificações ajuda a preencher as lacunas de informação do *cookbook* ativo e constitui uma forma de compreender o projeto do framework, alternativa à análise de código.

Quando se considera concluído o desenvolvimento da especificação da aplicação, procede-se uma primeira avaliação, submetendo a especificação desenvolvida ao *analisador SEA-Preceptor*. Esta ferramenta avalia se foram criadas na especificação as classes cuja criação é obrigatória, e se foram sobrepostos todos os métodos abstratos herdados pelas classes criadas. A consistência do conjunto da especificação é verificada pelo *analisador de consistência de especificação* do ambiente SEA.

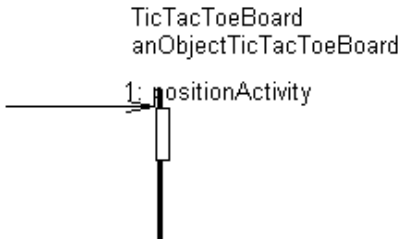

FIGURA 6.45 - Diagrama de seqüência para a descrição das interações produzidas pela execução do método positionActivity da classe TicTacToeBoard

No desenvolvimento de especificações sob um framework, o uso de um *cookbook* ativo não dispensa o usuário de todo o desenvolvimento. Por exemplo, o diagrama de seqüência criado para refinar a execução do método *positionActivity*, apresentado na figura 6.45, deve ser completado, de acordo com as características do jogo em desenvolvimento (*TicTacToe*). Da mesma forma, outros modelos criados vazios devem ser completados ao longo do desenvolvimento. Estas atividades não são tratadas pelo *cookbook* ativo.

Sumário das ações executadas a partir do acionamento de *links* ativos:

- ***Link* ativo de criação de especificação**
  - ⇨ criação de especificação;
  - ⇨ reprodução dos diagramas de casos de uso e de atividades da especificação do framework na especificação criada;
  - ⇨ criação de um diagrama de classes, vazio, na especificação criada (com o nome *main object model*, em que serão incluídas as classes criadas);
- ***Link* ativo de tratamento de classe redefinível**
  - ⇨ criação de subclasse da classe referenciada (ou apenas seleção de subclasse existente, caso haja);
  - ⇨ criação opcional de diagrama de transição de estados, vazio, associado à classe tratada (classe recém-criada ou classe existente selecionada);
  - ⇨ sobreposição na classe tratada, de cada método abstrato herdado, com um método base (homônimo);
  - ⇨ para cada método criado, criação de um diagrama de corpo de método, vazio, associado;
  - ⇨ para cada método criado, criação opcional de um diagrama de seqüência referenciando o método na mensagem originadora ;
- ***Link* ativo de conclusão de especificação**
  - ⇨ submissão da especificação tratada ao analisador SEA-Preceptor (demais ações são condicionadas à obediência da especificação às restrições do framework tratado);

- ⇨ reprodução dos diagramas de seqüência da especificação do framework, na especificação criada;
- ⇨ desvinculação da especificação tratada, do *cookbook* ativo.

6.7 Suporte ao Desenvolvimento e Uso Componentes

A abordagem de desenvolvimento e uso de componentes aqui apresentada é utilizada no ambiente SEA, em que o paradigma de orientação a objetos é adotado para desenvolver frameworks, componentes e aplicações. Assim, componentes são definidos através de especificações OO. O que caracteriza uma estrutura de classes de componente é que esta reutiliza uma interface, selecionada em uma biblioteca de interfaces de componentes. Uma interface aparece na especificação do componente como parte de sua estrutura de classes, porém possui uma especificação individual que descreve sua estrutura e sua dinâmica comportamental.

A especificação de interfaces em separado da especificação de componentes permite que uma mesma especificação de interface possa ser reusada em vários componentes, produzindo assim, uma família de componentes estrutural e comportamentalmente equivalentes.

A seguir, é descrita a abordagem de especificar interfaces de componentes desenvolvida no trabalho de pesquisa ora descrito, e implantada no ambiente SEA. É descrito também como estas especificações de interface são reusadas no desenvolvimento de especificações de componentes, e como componentes são usados para produzir outros artefatos de software.

6.7.1 Especificação de Interface de Componente

A especificação estrutural de uma interface de componente relaciona os métodos fornecidos, os métodos requeridos e a associação destes a cada canal da interface. A figura 6.46 exemplifica uma situação hipotética de uma interface com três canais, cinco métodos requeridos e cinco métodos fornecidos. Nem todos os métodos fornecidos precisam estar acessíveis em todos os canais (a marca na tabela define a acessibilidade). O mesmo ocorre com os métodos requeridos, possibilitando que a responsabilidade de implementar os métodos requeridos por um componente esteja distribuída entre um conjunto de componentes estrutural, comportamental e funcionalmente diferentes.

A estrutura de uma interface no ambiente SEA é produzida relacionando assinaturas de métodos e canais, e definindo a ligação entre estes, conforme o exemplo da figura 6.46.

		Métodos requeridos					Métodos fornecidos				
		mA	**mB**	**mC**	**mD**	**mE**	**mF**	**mG**	**mH**	**mI**	**mJ**
Canais	**c1**	✓		✓	✓		✓	✓	✓		
	c2	✓	✓				✓			✓	✓
	c3		✓		✓	✓	✓		✓	✓	✓

FIGURA 6.46 - Estrutura de relacionamento de canais e métodos de uma especificação hipotética de interface de componente

A questão a ser tratada na descrição comportamental da interface de um componente é se há ou não restrições associadas à ordem de invocação de métodos. A

inexistência de restrições significa que qualquer método fornecido ou requerido pode ser invocado a qualquer instante do tempo de existência de um componente. Se existirem porém restrições, como a necessidade de invocar determinado método antes de outro, estas devem fazer parte da modelagem da interface. Nas propostas de mecanismos de descrição comportamental apresentadas no capítulo anterior, observam-se duas tendências opostas. De um lado mecanismos como *reuse contracts* [LUC 97], de fácil compreensão, porém com expressividade limitada, conforme ilustrado no exemplo da figura 5.3, em que se demonstrou que a incompatibilidade comportamental de dois componentes não é representável com este mecanismo. De outro, mecanismos formais, como a proposta de descrição do comportamento da interface através de *Lambda Calculus* [CAN 97], porém de difícil compreensão.

Isto ilustra um dilema na escolha da técnica de modelagem adequada à modelagem comportamental de interfaces. Uma técnica de modelagem formal permite produzir modelos sem ambigüidades e que podem ser validados formalmente. A questão da compreensibilidade se reflete na maior ou menor facilidade de entendimento dos modelos produzidos. Os dois aspectos são importantes, porém, em geral, são antagônicos [FOW 94].

		Métodos requeridos		Métodos fornecidos
		metX	**metY**	**metZ**
Canais	**Ca**	✓	✓	✓
	Cb	✓		✓

FIGURA 6.47 - Estrutura de canais e métodos de uma interface de componente

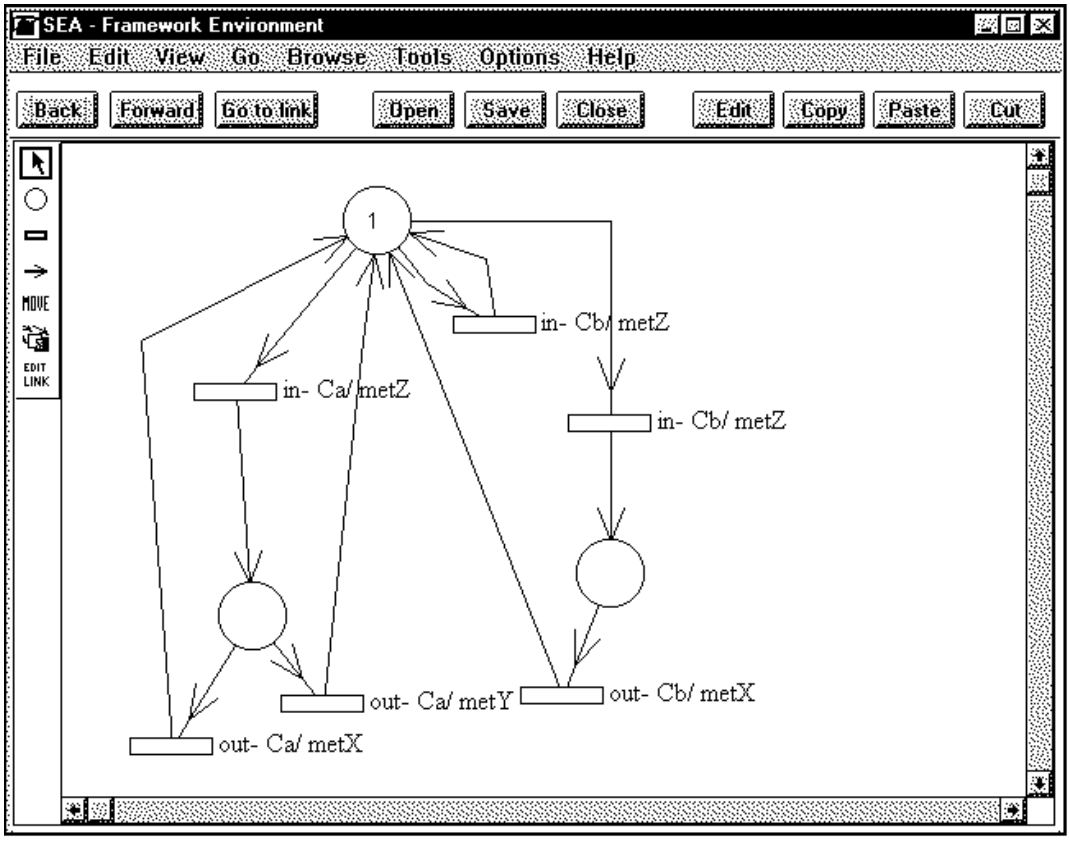

FIGURA 6.48 - Descrição comportamental de interface de componente usando rede de Petri

Considerando a adequação de uma técnica de modelagem gráfica em relação ao aspecto da compreensibilidade, e a necessidade de formalismo que permita avaliar uma composição de componentes em relação a um conjunto de propriedades, adotou-se o uso de rede de Petri [PET 62] para a modelagem comportamental da interface de componentes. Este modelo é baseado em formalismo algébrico, o que permite a análise de propriedades de uma composição de componentes. Também possui uma notação gráfica correspondente, que o torna de mais fácil compreensão que outros formalismos.

No ambiente SEA está sendo usada a rede de Petri ordinária, que é composta por lugares, transições, arcos que interligam lugares e transições, e uma marcação inicial, caracterizada por uma quantidade de fichas em cada lugar da rede. A única extensão em relação à rede de Petri ordinária é a necessidade de associar cada par *(canal, método)* definido na estrutura da interface a uma ou mais transições da rede. A figura 6.48 ilustra a descrição comportamental de uma interface hipotética usando este modelo. A figura 6.47 contém a descrição da estrutura desta interface.

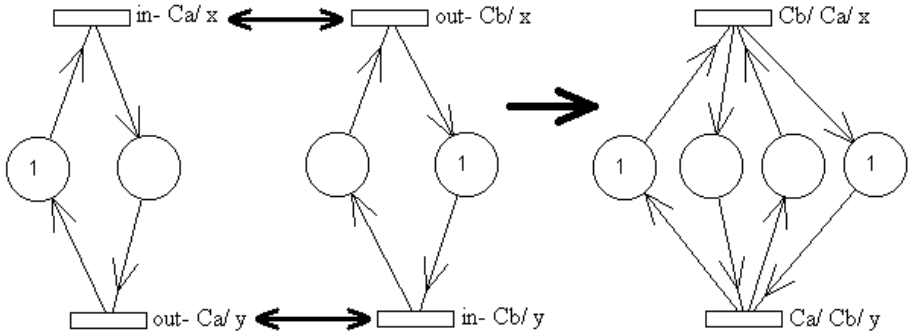

FIGURA 6.49 - Descrição comportamental de uma arquitetura de componentes

O comportamento de um conjunto de componentes interligados é obtido a partir da união das redes de Petri que descrevem suas interfaces. No processo de união das redes, lugares, arcos e marcação inicial das redes originais são mantidos e pares de transições correspondentes são fundidos em uma transição. Duas transições de redes diferentes são correspondentes se estiverem associadas ao mesmo método (requerido por uma interface e fornecido pela outra) e a canais interligados na composição da arquitetura de componentes. A figura 6.49 ilustra a rede que descreve a interligação de dois componentes estruturalmente compatíveis, mas comportamentalmente incompatíveis, exemplo descrito no capítulo anterior. As duas redes de Petri à esquerda descrevem a interface dos dois componentes e a rede à direita descreve o comportamento da arquitetura resultante da conexão destes. Os componentes estão interligados a partir dos canais únicos de suas interfaces (Ca e Cb). Observe-se que a marcação inicial da rede resultante não habilita o disparo de nenhuma transição, caracterizando um *deadlock*, como descrito textualmente no capítulo anterior.

Quando um par *(canal, método)* está associado a N transições em uma das redes, e o par *(canal, método)* correspondente da outra rede (mesmo método, canais interligados) está associado a uma transição, a rede resultante apresentará N transições associadas a estes dois pares. A figura 6.50 ilustra a situação em que em uma das redes o par correspondente ao método x está associado a uma transição e na outra, a duas. A rede resultante apresenta duas transições para x. No caso geral, em que em uma das redes um par *(canal, método)* está associado a N transições e na outra o par correspondente está associado a M transições, a rede resultante apresentará NxM transições associadas a estes pares.

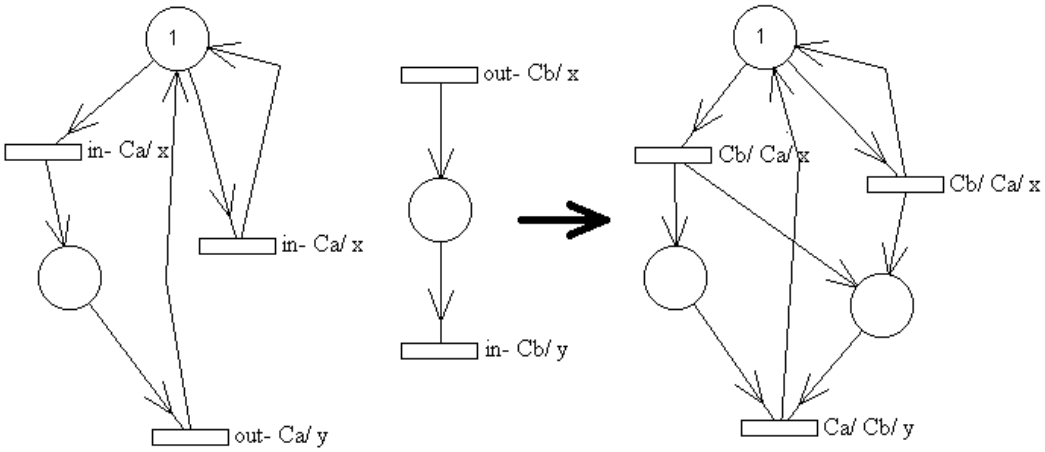

FIGURA 6.50 - Descrição comportamental de uma arquitetura de componentes

Os lugares da rede de Petri representam pré ou pós-condições das transições a que estão interligados. Assim, uma transição sem pré-condição (que representa um método sem restrições para ser invocado) não possui lugar de entrada. De modo análogo, uma transição que não gera pós-condição não possui lugar de saída. O exemplo da figura 6.50 apresenta transições com estas características.

A adoção de redes de Petri para descrever o comportamento de interfaces de componentes, além da vantagem da compreensibilidade, permite analisar propriedades de interfaces individuais e arquiteturas de componentes, como a possibilidade de avaliar a existência de *deadlock* ou a possibilidade de verificar se existem transições que nunca são disparadas (ou seja, métodos que nunca são invocados), a partir da análise de invariantes.

6.7.2 Especificação de Componente

No ambiente SEA um componente é definido como uma especificação OO, semelhantemente ao que ocorre no desenvolvimento de frameworks e aplicações. O que caracteriza a especificação de um componente é que parte da especificação corresponde à descrição de uma interface, isto é, um conjunto de classes responsável pela comunicação do componente com o meio externo. A figura 6.51 apresenta a estrutura de classes correspondente à implantação da especificação de interface descrita nas figuras 6.47 e 6.48 (*SpecificInterface*) em uma especificação de componente (*SpecificComponent*), no ambiente SEA. Todas as classes, exceto *SpecificComponent*, são classificadas como externas. Nesta estrutura é reusado o padrão de projeto para interfaces de componentes composto pelas três classes do nível mais elevado de herança: *SEAComponent*, *SEAInterfaceChannel* e *SEAOutputMailbox*. No ambiente SEA, a presença destas três classes em uma especificação (classificadas como classes externas) e de uma subclasse concreta de *SEAComponent* caracteriza a especificação de um componente. As classes que definem o padrão de interface de componentes são descritas a seguir.

- ***SEAComponent*** - define o protocolo da parte externamente acessível de um componente. Uma subclasse concreta desta classe corresponde à fachada de um componente específico e provê os meios de comunicação entre os elementos internos de um componente e o meio externo (utiliza-se neste caso, o padrão de projeto *Facade*). Uma instância de subclasse desta classe agrega uma ou mais instâncias de subclasses de *SEAInterfaceChannel*.

- ***SEAInterfaceChannel*** - corresponde à modelagem de um canal de interface de componente. Tem uma estrutura de dados (que corresponde ao atributo *referenceStructure*) para referenciar os objetos da estrutura interna que implementam os métodos fornecidos pelo componente, acessíveis no canal. Cada subclasse concreta desta classe implementa o conjunto de métodos fornecidos pelo componente através do canal. O algoritmo de um método fornecido em uma subclasse de *SEAInterfaceChannel* corresponde a um repasse da invocação do método ao objeto interno referenciado para implementá-lo, semelhante ao algoritmo dos métodos de um *proxy* (item 6.5.8). Uma instância de subclasse desta classe agrega exatamente uma instância de subclasse de *SEAOutputMailbox* (atributo o*utputMailbox*).
- ***SEAOutputMailbox*** - fornece o meio de acesso aos métodos requeridos por um componente através de um canal de interface. Mantém referência do elemento externo de que são invocados os métodos requeridos (atributo *externalReference*). Cada subclasse concreta desta classe implementa o conjunto de métodos requeridos pelo componente através do canal. Semelhante aos algoritmos dos métodos fornecidos, definidos nas subclasses de *SEAInterfaceChannel*, os algoritmos dos métodos requeridos, definidos nas subclasses de *SEAOutputMailbox*, correspondem a repasses da invocação, porém, neste caso, ao elemento externo referenciado. Um canal de um componente utiliza uma instância de subclasse *SEAInterfaceChannel*, em que estão implementados os métodos fornecidos pelo componente através do canal, e uma instância de subclasse *SEAOutputMailbox*, em que estão implementados os métodos requeridos. A necessidade de objetos distintos para implementação dos métodos requeridos e fornecidos se deve à possibilidade de existirem métodos fornecidos e requeridos com assinaturas idênticas - a implementação do canal em um único objeto inviabilizaria esta possibilidade.

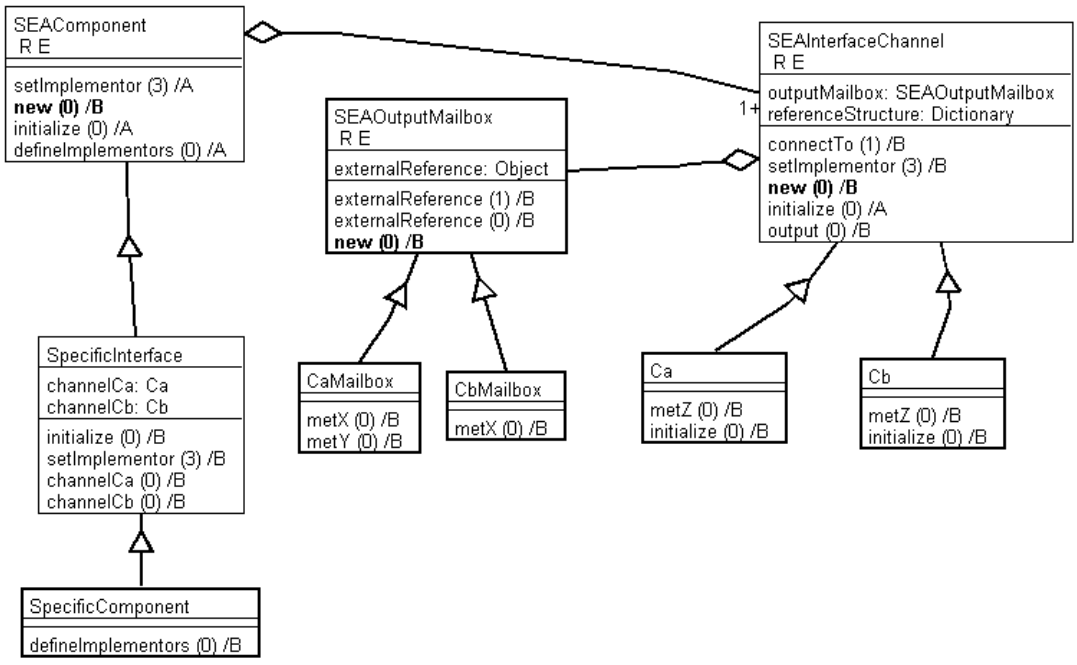

FIGURA 6.51 - Estrutura de classes correspondente à implantação da especificação de interface descrita nas figuras 6.47 e 6.48 (SpecificInterface) em uma especificação de componente (SpecificComponent)

Na construção da especificação do componente *SpecificComponent*, exemplo ilustrado na figura 6.51, todas as classes referentes à especificação da interface (presentes nesta figura) são inseridas automaticamente pela *ferramenta de importação de interface* do ambiente SEA, a partir da seleção de uma especificação de interface previamente inserida na biblioteca do ambiente - especificada da forma descrita na seção anterior. No procedimento de inserção de interface executado pela *ferramenta de importação de interface*, o diagrama de corpo de método do método *defineImplementors* da classe *SpecificComponent* (a classe é criada com o mesmo nome da especificação do componente) é inserido vazio. Isto porque a definição dos objetos internos que implementam os métodos fornecidos pelo componente é parte da construção da especificação do componente. Na figura 6.52, em que é descrita a instanciação de um componente, observa-se a ação do método *defineImplementors*.

A *ferramenta de conversão de especificação de interface* do ambiente é capaz de produzir a especificação de um framework (uma estrutura de classes) a partir de uma especificação de interface (construída como descrito no item 6.7.1). A conversão da especificação de interface descrita nas figuras 6.47 e 6.48 produz a estrutura de classes de um framework - que reusa o padrão de interfaces - que contém as classes presentes na figura 6.51 (exceto a classe *SpecificComponent*), com as classes *SEAComponent*, *SEAInterfaceChannel* e *SEAOutputMailbox*, classificadas como externas[63]. A ferramenta produz uma especificação OO consistente, que inclui os algoritmos dos métodos e que pode ser convertida em linguagem de programação. O framework correspondente à conversão de uma especificação de interface é a estrutura inserida automaticamente pela *ferramenta de importação de interface* em uma especificação de componente - com as classes do framework classificadas como externas - como ilustrado na figura 6.51.

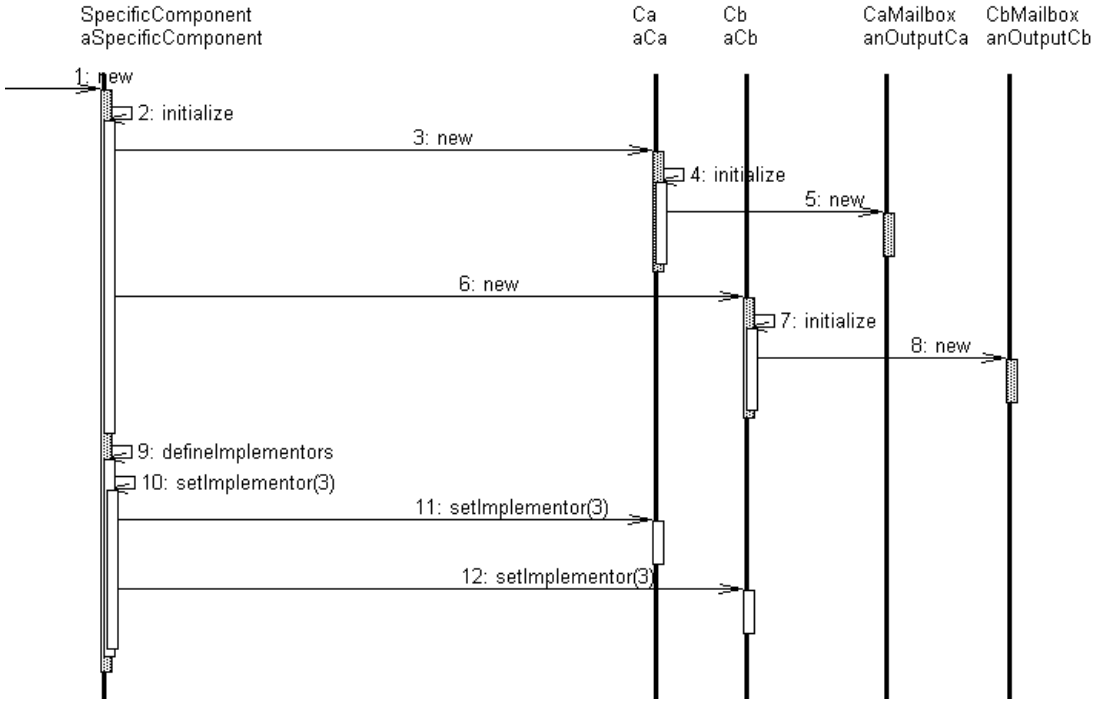

FIGURA 6.52 - Instanciação de um componente

63 A subclasse abstrata de SEAComponent produzida na conversão de uma especificação de interface recebe o mesmo nome da especificação de interface.

Na utilização da *ferramenta de importação de interface* do ambiente, pode-se optar pela inclusão ou não da rede de Petri que faz parte da especificação de uma interface. Uma rede de Petri é implementada como uma classe (*InterfacePetriNet*) e uma mesma instância desta classe é referenciada por todas as instâncias de subclasses de *SEAInterfaceChannel* e *SEAOutputMailbox* (um atributo adicional que não aparece na figura 6.51, que corresponde à estrutura de interface sem rede de Petri) que comunicam à rede a execução de método requerido ou fornecido. Na execução de um método requerido por um componente - em uma instância de subclasse de *SEAOutputMailbox* - é enviada notificação para instância de *InterfacePetriNet*, que retorna *true* se a transição correspondente estiver habilitada e *false*, caso contrário. Apenas no caso de transição habilitada é que haverá envio de mensagem ao objeto externo referenciado pela instância de subclasse de *SEAOutputMailbox*. Com isto, garante-se que um componente não produzirá invocações indevidas de métodos requeridos. Nada é garantido em relação aos métodos fornecidos, porém, se um artefato de software for construído como uma arquitetura de componentes e se for garantido que nenhum dos componentes invocará método de outro desobedecendo as restrições estabelecidas na rede resultante para o conjunto, estar-se-á garantindo consistência - se nenhum componente do conjunto origina invocação indevida, por conseguinte, nenhum componente recebe invocação indevida.

Uma limitação da atual implementação do ambiente SEA é a indisponibilidade de um mecanismo de análise de rede de Petri - necessário para a análise da rede de uma especificação de interface e para a análise da rede resultante de uma arquitetura de componentes. Também o procedimento de unificação das redes de uma arquitetura ainda não está automatizado.

O ambiente SEA suporta o uso da abordagem de frameworks para a construção de componentes flexíveis. Um framework desenvolvido para produzir componentes corresponde a uma estrutura de classes que apresenta uma interface e partes mantidas flexíveis, para possibilitar sua adaptação a diferentes requisitos, possibilitando a produção de componentes diferentes. Um framework pode corresponder a uma implementação inacabada de componente ou pode conter uma implementação *default*. Neste caso, haveria a disponibilidade de um componente sem necessidade de adaptação da estrutura do framework.

A vantagem da adoção desta abordagem está na possibilidade de obter componentes diferentes, a partir de um esforço menor que o necessário para desenvolver cada um isoladamente. Quando se reutiliza um componente flexível, construído como um framework, para produzir um componente específico, parte da estrutura deste componente é reutilizada, reduzindo o esforço necessário para a construção do componente. Isto é possibilitado pelo reuso de projeto e código promovido pela abordagem de frameworks. As dificuldades associadas ao desenvolvimento e uso de frameworks, discutidas nos capítulos 3 e 4, são atenuadas pelo suporte fornecido pelo ambiente SEA.

Com a diminuição do esforço necessário para produzir novos componentes promovida pela abordagem dos frameworks, viabiliza-se uma alternativa às abordagens de adaptação de componentes. Ao invés de adaptar um componente existente, pode-se criar um novo componente, reutilizando um framework construído para suportar o desenvolvimento de componentes e até mesmo reutilizando parte da especificação de componentes existentes. Cabe salientar que mesmo um framework que não tenha sido desenvolvido para suportar o desenvolvimento de componentes e por isto não apresente uma interface, como o framework Frag, pode suportar a produção de componentes.

Neste caso, parte do procedimento de extensão da estrutura do framework corresponde à inserção de uma interface no artefato desenvolvido.

6.7.3 Uso de Componentes

Na atual implementação do ambiente SEA, componentes contidos na biblioteca de especificações do ambiente podem ser importados para especificações OO (de aplicações, frameworks ou componentes). A inserção de componentes é feita pela *ferramenta de importação de componentes* do ambiente, que insere apenas a classe *fachada* do componente (*SpecificComponent*, no caso do exemplo da figura 6.51), classificada como classe externa. Cada instância desta classe corresponde a um componente, cuja estrutura interna não é visível à especificação que o utiliza. A interação de um componente com o meio externo inclui a conexão de seus canais a elementos externos (método *connectTo* de *SEAInterfaceChannel*), o recebimento de invocações de métodos fornecidos através dos canais do componente, e a execução dos métodos requeridos, invocados pelo componente.

Uma especificação de componente flexível (que apresenta característica de framework e de componente) deve ser usada para a produção de uma especificação de componente, pois uma especificação de framework flexível não é importável para uma especificação OO.

A partir da instanciação e interligação de um conjunto de componentes é possível produzir artefatos de software como arquiteturas de componentes, baseados em estilos arquitetônicos específicos, como os apresentados no capítulo anterior. Para isto, podem ser produzidos componentes com a função de conectores, isto é, voltados a estabelecer a conexão entre outros componentes, como um conector para o estilo *pipeline*, por exemplo. A estrutura de especificação voltada à modelagem de arquiteturas de componentes (mencionada no item 6.5.1, em que é discutido o procedimento de criação de especificações), ainda não incluída no ambiente SEA, visa modelar artefatos de software como conexões de componentes, permitindo a explicitação de estilos arquitetônicos. O framework OCEAN, apresentado no capítulo seguinte, provê a infra-estrutura necessária para incluir esta característica no ambiente SEA

7 O Framework OCEAN

O framework OCEAN é voltado à produção de ambientes de desenvolvimento de software. Possibilita o desenvolvimento de ambientes que manipulem diferentes estruturas de especificação, bem como apresentem diferentes funcionalidades. O framework OCEAN foi utilizado para o desenvolvimento do ambiente SEA, descrito no capítulo anterior, que suporta o desenvolvimento e uso de frameworks e componentes.

O desenvolvimento do framework OCEAN foi motivado pela necessidade de produzir um ambiente flexível de suporte ao desenvolvimento e uso de frameworks e componentes. Por se tratar de um ambiente acadêmico para experimentações em torno do desenvolvimento e uso de frameworks e componentes, a flexibilidade foi estabelecida como um requisito fundamental para permitir ao ambiente moldar-se a necessidades constatadas ao longo de seu uso. Neste contexto, a solução natural foi desenvolver um framework para suportar a construção de ambientes, ao invés de desenvolver diretamente um ambiente. O confronto dos requisitos estabelecidos para suporte ao desenvolvimento e uso de frameworks e componentes com as características de ambientes existentes permitiu identificar os requisitos de flexibilidade para um framework voltado ao desenvolvimento de ambientes. Assim, OCEAN consegue generalizar as características de ambientes de desenvolvimento de software e pode suportar o desenvolvimento de ambientes em geral - não apenas ambientes relacionados a frameworks e componentes.

De um modo geral, um framework é um artefato de software que, diferente de uma aplicação, não precisa corresponder a um artefato executável. Sua finalidade é prover uma estrutura extensível, que generalize um domínio de aplicações, a ser utilizada no desenvolvimento de diferentes aplicações. Um framework, todavia, pode incluir aplicações completas em sua estrutura. Este é o caso do framework HotDraw [BRA 95], utilizado na elaboração do framework OCEAN, para suportar o desenvolvimento de editores gráficos. A classe *DrawingEditor* de HotDraw, cujas subclasses correspondem a editores específicos, é concreta e implementa um editor, disponibilizado com o framework. O framework MVC [PAR 94] é voltado a suportar o desenvolvimento de interfaces e foi utilizado no desenvolvimento de OCEAN. MVC, ao contrário de HotDraw, não inclui aplicações acabadas[64].

A única classe específica de um ambiente desenvolvido sob o framework OCEAN é uma subclasse concreta de *EnvironmentManager* (que é uma classe abstrata). Assim, das duzentas e sessenta classes que constituem o protótipo OCEAN-SEA, apenas uma classe, a classe *Sea*, subclasse de *EnvironmentManager*, é exclusiva do ambiente SEA - todas as demais podem ser reutilizadas no desenvolvimento de outros ambientes. Um ambiente específico é produzido a partir da criação de uma subclasse concreta de *EnvironmentManager*[65]. Em uma subclasse desta classe são definidas as seguintes características de um ambiente específico:

- quais tipos de especificação são tratados pelo ambiente (cada tipo de especificação define um conjunto de tipos de modelo e cada tipo de modelo define um conjunto de tipos de conceito);

---

64 Exceto pequenas aplicações, voltadas a demonstrar características de partes do framework.

65 Além de classes concretas - subclasses de classes abstratas do framework OCEAN - relacionadas a especificações, conceitos, modelos e respectivos mecanismos de visualização e edição, que não estejam previstos no framework.

- qual o mecanismo de visualização e edição associado a cada modelo e conceito tratado pelo ambiente;
- qual o mecanismo de armazenamento de especificações utilizado pelo ambiente;
- quais as ferramentas utilizáveis para manipular especificações.

No presente capítulo o framework OCEAN é apresentado com ênfase na descrição do suporte ao desenvolvimento de diferentes ambientes. Isto é, como a arquitetura do framework prevê a definição de novos tipos de documento[66], bem como de estruturas funcionais que manipulem documentos. A descrição apresentada neste capítulo segue o estilo de tutorial, descrito no capítulo anterior (item 6.6.1), que intercala descrição textual e modelos (da especificação de OCEAN). Não tem o objetivo de apresentar a especificação de projeto do framework OCEAN, o que seria demasiado extenso, mas ressaltar as partes de sua estrutura que lhe conferem flexibilidade para suportar o desenvolvimento de ambientes diferentes. Inicialmente, é descrita a parte da estrutura do framework que suporta a definição de diferentes tipos de documento, em complemento à descrição da estrutura de especificações apresentada no capítulo anterior. São também apresentadas as estruturas de visualização e edição dos documentos que compõem uma especificação. A seguir, são descritas as funcionalidades de edição básicas supridas pelo framework, e é discutido como elas devem ser completadas ou modificadas para a definição de novos tipos de documento ou de novos mecanismos de edição. Ao final, discute-se como produzir ferramentas complexas de manipulação de documentos, a partir de composição de funcionalidades.

7.1 Suporte à Criação de Estruturas de Documentos

Ambientes de desenvolvimento de software constituem um tipo de aplicação de manipulação de documentos. Especificações de projeto de artefatos de software são documentos estruturados, que descrevem visões distintas de projeto. Metodologias específicas utilizam diferentes abordagens de desenvolvimento, caracterizadas por conjuntos de técnicas de modelagem para a produção de especificações de projeto, formais ou não, e baseadas em paradigmas de desenvolvimento específicos. O framework OCEAN suporta a definição de diferentes estruturas de documentos. No ambiente SEA são manipulados documentos com diferentes estruturas - *cookbook* ativo, especificação estrutural e comportamental de interfaces de componentes e especificação OO. Para suprir a flexibilidade necessária para suportar diferentes conjuntos de técnicas de modelagem, parte da estrutura do framework OCEAN corresponde à definição genérica de uma estrutura de documento, a ser estendida para a produção de estruturas específicas, como aquelas tratadas pelo ambiente SEA.

A seguir, é descrito como o framework OCEAN suporta a definição de diferentes tipos de documentos, como também as funcionalidades genéricas previstas para a visualização e o armazenamento de documentos.

---

66 Um ambiente desenvolvido sob o framework OCEAN manipula especificações, que são compostas por modelos e conceitos, como descrito no capítulo anterior. Neste capítulo a expressão documento é usada para referência a especificação, conceito, modelo ou qualquer outro tipo de estrutura de informações definível sob o framework OCEAN.

7.1.1 Estrutura de Documentos sob o Framework OCEAN

A figura 7.1 apresenta a estrutura hierárquica de herança das classes abstratas do framework OCEAN que constitui o suporte à definição de documentos[67]. Este mesmo conjunto de classes foi apresentado no capítulo anterior, onde se destacou que uma especificação, isto é, uma instância de subclasse de *Specification*, agrega elementos de especificação, instâncias de subclasses de *Concept* e *ConceptualModel*, e que uma especificação registra associações de sustentação e referência entre pares de elementos de especificação (vide item 6.2). A figura 7.2 inclui as classes *SpecificationElementHolder* e *RelationshipTable*, classes de projeto[68] definidas para organizar as associações entre uma instância de especificação e as instâncias de elementos de especificação que a compõem. Na figura 7.3 observa-se o procedimento de instanciação de uma especificação definido no framework OCEAN. Quando é criada uma instância de especificação, é criado um conjunto de instâncias de *SpecificationElementHolder* para o repositório de modelos e um outro conjunto para o repositório de conceitos - uma instância de *SpecificationElementHolder* para cada tipo de modelo e para cada tipo de conceito tratado pela especificação[69]. Também são produzidas duas instâncias de *RelationshipTable*, uma correspondente à tabela de sustentação e outra, à tabela de referência.

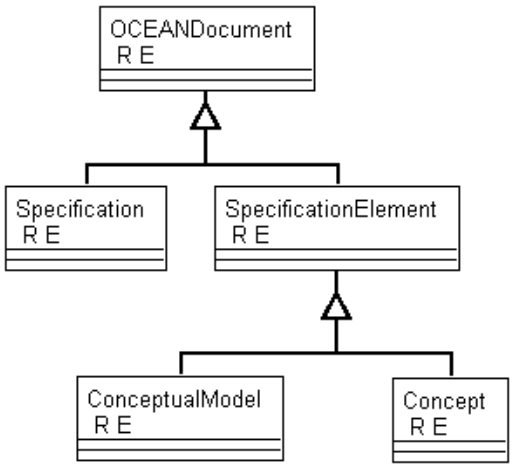

FIGURA 7.1 - Hierarquia de herança das classes abstratas que definem a estrutura de um documento sob o framework OCEAN

O procedimento apresentado na figura 7.3, que é reusado por diferentes tipos de especificação, produz a estrutura de armazenamento de informações de uma especificação descrita no capítulo anterior, isto é, o repositório de conceitos, o repositório de modelos, a tabela de sustentação e a tabela de referência. Esta estrutura também pode ser observada no anexo 1, em que é descrita formalmente a estrutura de

---

67 O diagrama de classes da figura 7.1, assim como os demais diagramas de classes apresentados no presente capítulo, omite a relação de atributos e métodos das classes. Métodos das classes que atuam nos procedimentos descritos aparecem nos diagramas de seqüência apresentados.

68 A classificação *classe de projeto*, aplicada às classes *SpecificationElementHolder* e *RelationshipTable*, designa uma classe definida como solução para problema de projeto, o que a diferencia das classes do domínio da aplicação, como as classes presentes na figura 7.1. Classes do domínio representam elementos ou informações do domínio modelado [JAC 92] [COA 92].

69 Um tipo de modelo corresponde a uma subclasse concreta de *ConceptualModel* e um tipo de conceito, a uma subclasse concreta de *Concept*.

especificação OO. A definição de um tipo de especificação consiste basicamente em definir quais os tipos de modelo tratados. A definição de um tipo de modelo inclui os tipos de conceitos tratados. Assim, o conjunto de tipos de conceito tratados por uma especificação é a união dos conjuntos de tipos de conceito tratados pelos tipos de modelo previstos para a especificação. A interdependência entre modelos de diferentes tipos é estabelecida pelos seguintes mecanismos:

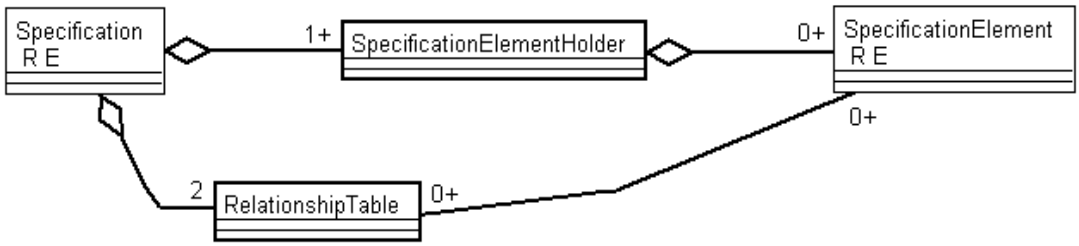

FIGURA 7.2 - Organização dos elementos de uma especificação

- uso de conceitos comuns (instâncias) pelos modelos, como um ator presente em um diagrama de casos de uso e em um diagrama de seqüência, por exemplo;
- estabelecimento de ligações, com ou sem conotação semântica, através de *links*, como a ligação entre um caso de uso e um diagrama de seqüência produzido para refiná-lo;
- requisitos estabelecidos para o conjunto da especificação, associados ao mecanismo responsável por verificar a sua consistência, como a verificação da existência de um diagrama de corpo de método associado a cada método template ou base, que não seja externo, de uma especificação OO - requisito avaliado pela ferramenta de verificação de consistência de especificações OO do ambiente SEA.

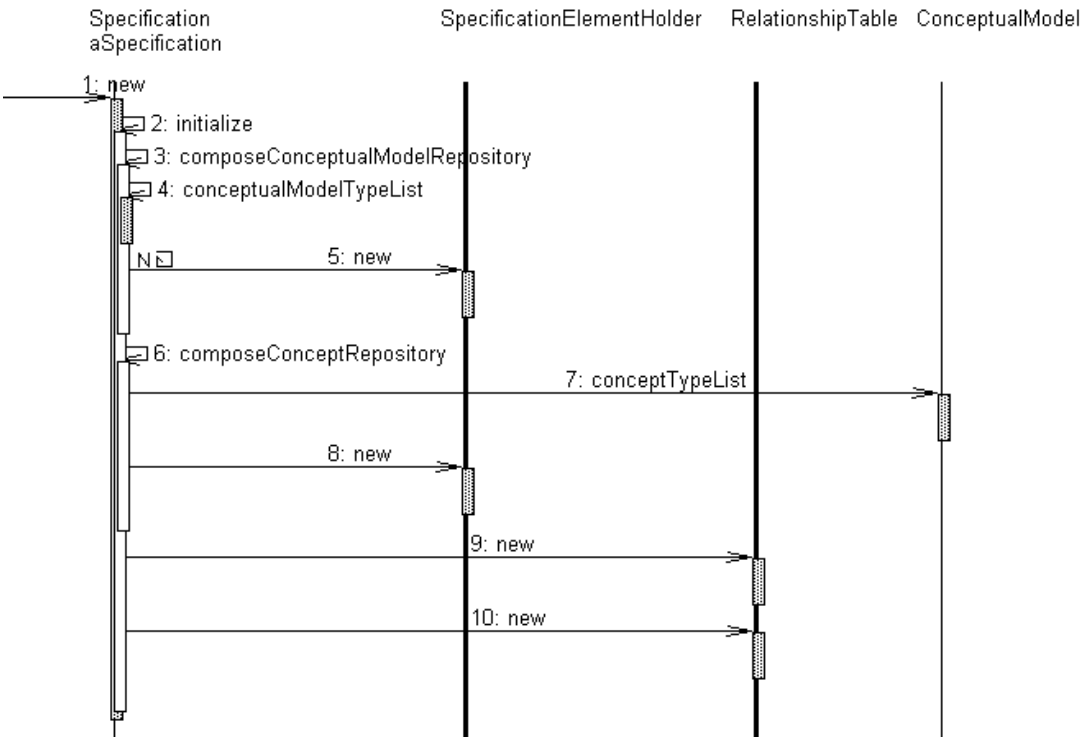

FIGURA 7.3 - Procedimento de instanciação de uma especificação

7.1.2 Os Mecanismos de Visualização de Elementos de Especificação

Um gerente de ambiente, isto é, uma instância de subclasse de *EnvironmentManager*, agrega um gerente de referências (instância de *ReferenceManager* ou de uma subclasse desta classe), que relaciona cada elemento de especificação tratado pelas especificações manipuláveis em um ambiente a um mecanismo de visualização do elemento. Na classe *ReferenceManager* do framework OCEAN é estabelecido um relacionamento *default* entre todos os elementos de especificação definidos no framework e os mecanismos de visualização respectivos, que também fazem parte da estrutura do framework. A criação de uma subclasse de *ReferenceManager* possibilita a alteração dos relacionamentos *default* ou a inclusão de novos relacionamentos entre modelos ou conceitos não-previstos no framework e respectivos mecanismos de visualização. O gerente de referência do framework OCEAN possibilita o desacoplamento entre elementos de especificação e mecanismos de visualização, possibilitando o reuso isolado de cada uma destas estruturas, bem como a associação de novos mecanismos de visualização a tipos de elemento de especificação existentes.

A classe *SpecificationElementInterface*, apresentada na figura 7.4, define a estrutura genérica dos mecanismos de visualização de elementos de especificação. A classe *ConceptInterface* corresponde à estrutura genérica dos mecanismos de visualização de conceitos e a classe *ConceptualModelInterface* corresponde à estrutura genérica dos mecanismos de visualização de modelos. Uma instância de subclasse concreta de *ConceptualModelInterface* agrega um editor gráfico. No protótipo do framework OCEAN, os editores gráficos desenvolvidos para os tipos de modelo tratados foram desenvolvidos estendendo o framework HotDraw (a classe *DrawingEditor* da figura 7.4 pertence ao framework HotDraw). Através da criação de subclasses concretas de *SpecificationElementInterface* podem ser definidos novos mecanismos de visualização de conceitos ou de modelos - que podem diferir dos mecanismos existentes, inclusive deixando de utilizar os recursos do framework HotDraw para a visualização de modelos ou deixando de utilizar os recursos do framework MVC para a visualização de conceitos. Dispositivos funcionais (ferramentas, botões etc.) podem ser associados aos mecanismos de visualização de elementos de especificação.

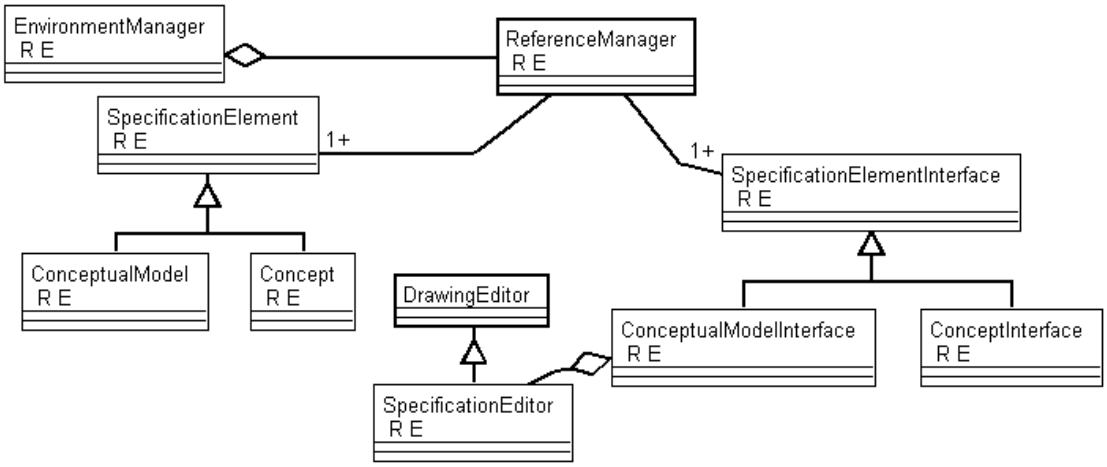

FIGURA 7.4 - Estrutura de edição de elemento de especificação do framework OCEAN

7.1.3 O Suporte à Navegação

O framework OCEAN provê um mecanismo de memorização dos documentos visualizados implementado na classe *Path*. Uma instância de *Path* agrega células de referência, instâncias de *MemoryCell*, que referenciam documentos visualizados ao longo do uso de um ambiente (vide figura 7.5). Esta estrutura suporta funções como *back* e *forward*, que alteram o índice do documento corrente, exibido na janela do ambiente. A figura 7.6 apresenta a dinâmica do método *updateWindow*, de *EnvironmentManager*, invocado após um procedimento de alteração do índice do documento corrente. A exibição de um documento consiste em:

- Fazer com que o documento seja referenciado por uma célula de referência;
- Fazer o índice desta célula ser o índice corrente (este procedimento e o anterior antecedem a invocação de *updateWindow*);
- Solicitar ao gerente de referências o mecanismo de visualização do documento;
- Carregar o documento, inserido no respectivo mecanismo de visualização, na janela do ambiente.

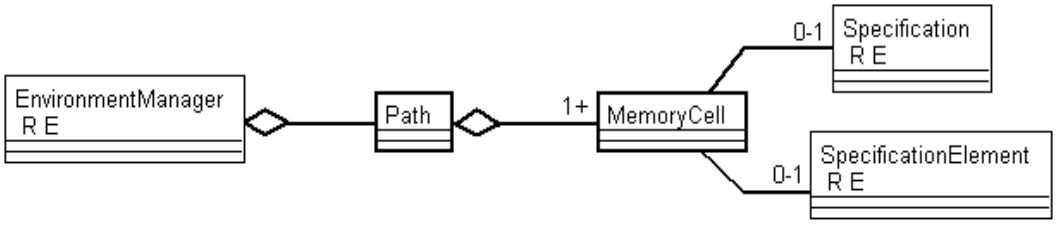

FIGURA 7.5 - Estrutura de suporte à navegação

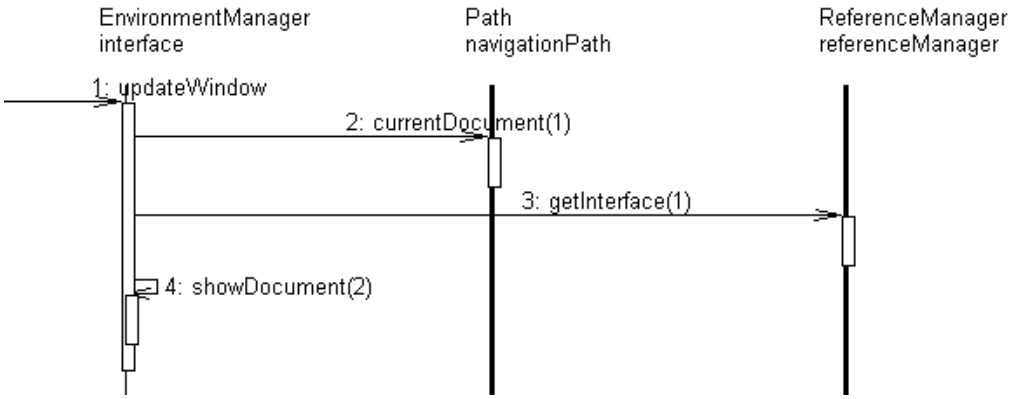

FIGURA 7.6 - Procedimento de exibição de um documento na interface de um ambiente

7.1.4 O Mecanismo de Armazenamento de Especificações

O framework OCEAN provê flexibilidade na definição do mecanismo de armazenamento de especificações de um ambiente. O protocolo de armazenamento e recuperação de especificações está definido na classe *StorageManager* (vide figura 7.7). O protótipo do framework OCEAN possui uma subclasse desta classe, *BOSSStorageManager*, utilizado pelo ambiente SEA, que possibilita o armazenamento de especificações em arquivo. A definição de um dispositivo de armazenamento diferente consiste em definir uma subclasse de *StorageManager*. Isto pode ser feito, por exemplo, para acoplar um sistema de gerenciamento de banco de dados a um ambiente, para o armazenamento de especificações. As figuras 7.8 e 7.9 apresentam, respectivamente, os

procedimentos de armazenamento e recuperação de especificações definidos no framework OCEAN.

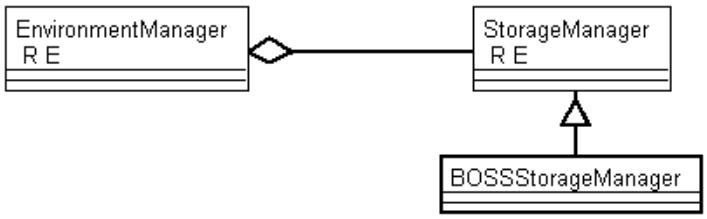

FIGURA 7.7 - Estrutura de suporte ao armazenamento e à navegação

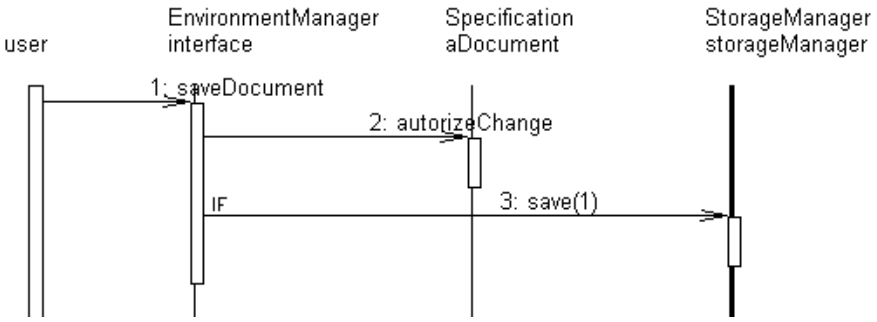

FIGURA 7.8 - Procedimento de armazenamento de uma especificação

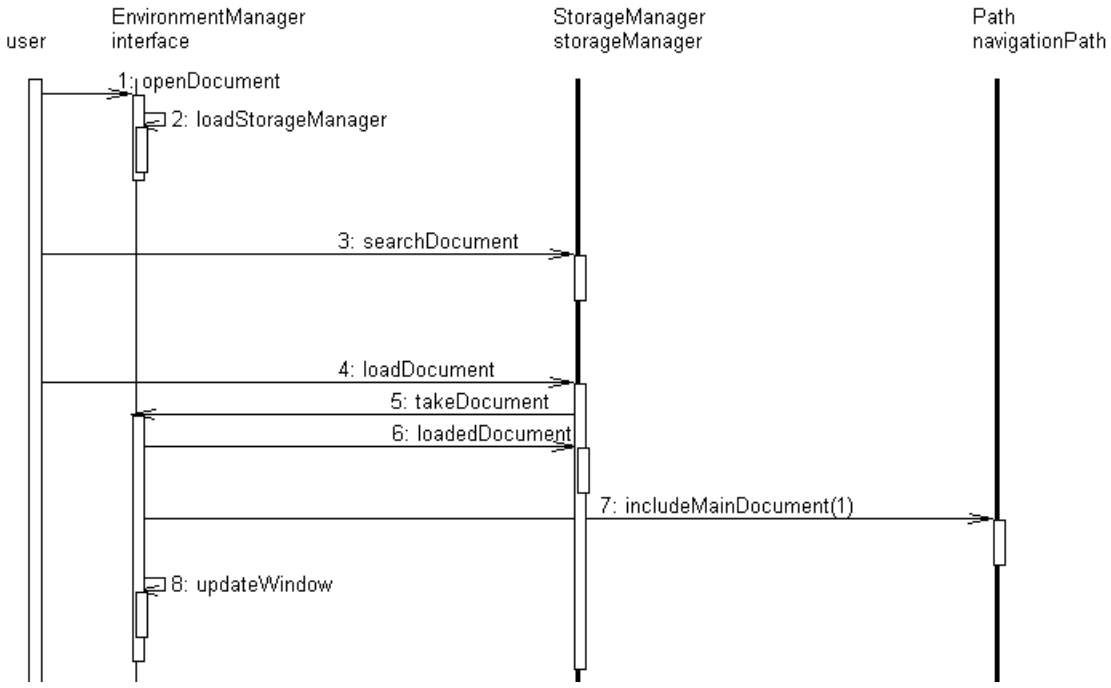

FIGURA 7.9 - Procedimento de recuperação de uma especificação armazenada

Para salvar uma especificação basta selecionar a opção *save* no menu ou botão da janela do ambiente - que acessa a funcionalidade definida na classe *EnvironmentManager* (vide figura 7.8). Para carregar uma especificação, ativa-se a opção *load* (menu ou botão da janela do ambiente), que carrega a janela do gerente de armazenamento[70] na janela do ambiente. A partir disto, a interação ocorre através do gerente de armazenamento, para selecionar uma especificação e carregá-la no ambiente, como descrito no diagrama de

70 Gerente de armazenamento é uma instância de subclasse de *EnvironmentManager*.

seqüência da figura 7.9. Na medida em que as subclasses de *StorageManager* mantenham fidelidade ao protocolo de armazenamento e recuperação de especificações, o mecanismo específico de armazenamento de especificações usado por um ambiente só é referenciado pela subclasse de *StorageManager* deste ambiente e a troca de um dispositivo de armazenamento por outro, está restrita à definição de uma nova subclasse de *StorageManager*, o que não afeta o restante da estrutura de um ambiente.

7.2 Suporte à Edição Semântica de Especificações

O processo de desenvolvimento de uma especificação consiste em criar, modificar ou remover elementos de especificação, isto é, conceitos e modelos. A modificação de um modelo consiste basicamente na inclusão e remoção de conceitos, enquanto a modificação de um conceito consiste em alterar o conjunto de informações que define sua estrutura - o que também pode envolver inclusão e remoção de conceitos envolvidos em associações de sustentação ou referência. Um ambiente que suporta edição semântica de especificações, o que é o caso de um ambiente produzido sob o framework OCEAN, procede as ações de edição sobre a estrutura de informações da especificação, modificações que se refletem nas representações visuais dos elementos afetados. A ênfase à edição semântica possibilita o estabelecimento de restrições às ações de edição, como o impedimento de estabelecer um relacionamento de herança entre elementos que não sejam classes, que garantem a manutenção da estrutura semântica de uma especificação.

A seguir é descrito como o framework OCEAN provê suporte aos procedimentos básicos de edição semântica de especificações. Em OCEAN, parte da responsabilidade das ações de edição é delegada aos elementos de especificação. Assim, a descrição da dinâmica de edição enfatiza os requisitos funcionais dos elementos de especificação, que precisam ser considerados na criação de novos tipos de elemento. A descrição a seguir, complementa a descrição das funcionalidades procedida no capítulo anterior, enfatizando como o framework OCEAN as suporta.

7.2.1 Criação de Modelos e Conceitos

Na figura 7.10 é apresentado um diagrama de seqüência que descreve o procedimento de criação de um elemento de especificação em uma especificação. Trata-se de uma das possíveis situações de criação, em que o método de criação invocado apresenta quatro parâmetros: a classe do elemento a ser criado, seu identificador (*string*) e dois elementos sustentadores (previamente inseridos na especificação). Existem outros métodos de criação de especificação que prevêem diferentes quantidades de elementos sustentadores - zero, inclusive.

No diagrama da figura 7.10 observam-se dois aspectos a serem considerados na criação de subclasses concretas de SpecificationElement, para descrever um tipo de conceito ou de modelo: o método construtor[71] do elemento de especificação e os métodos que procedem testes.

---

[71] Método construtor é um método que produz instâncias da classe a que pertence. A denominação construtor o diferencia de métodos de acesso, que procedem a leitura dos atributos de um objeto e dos métodos modificadores, que alteram os atributos de um objeto [MEY 97].

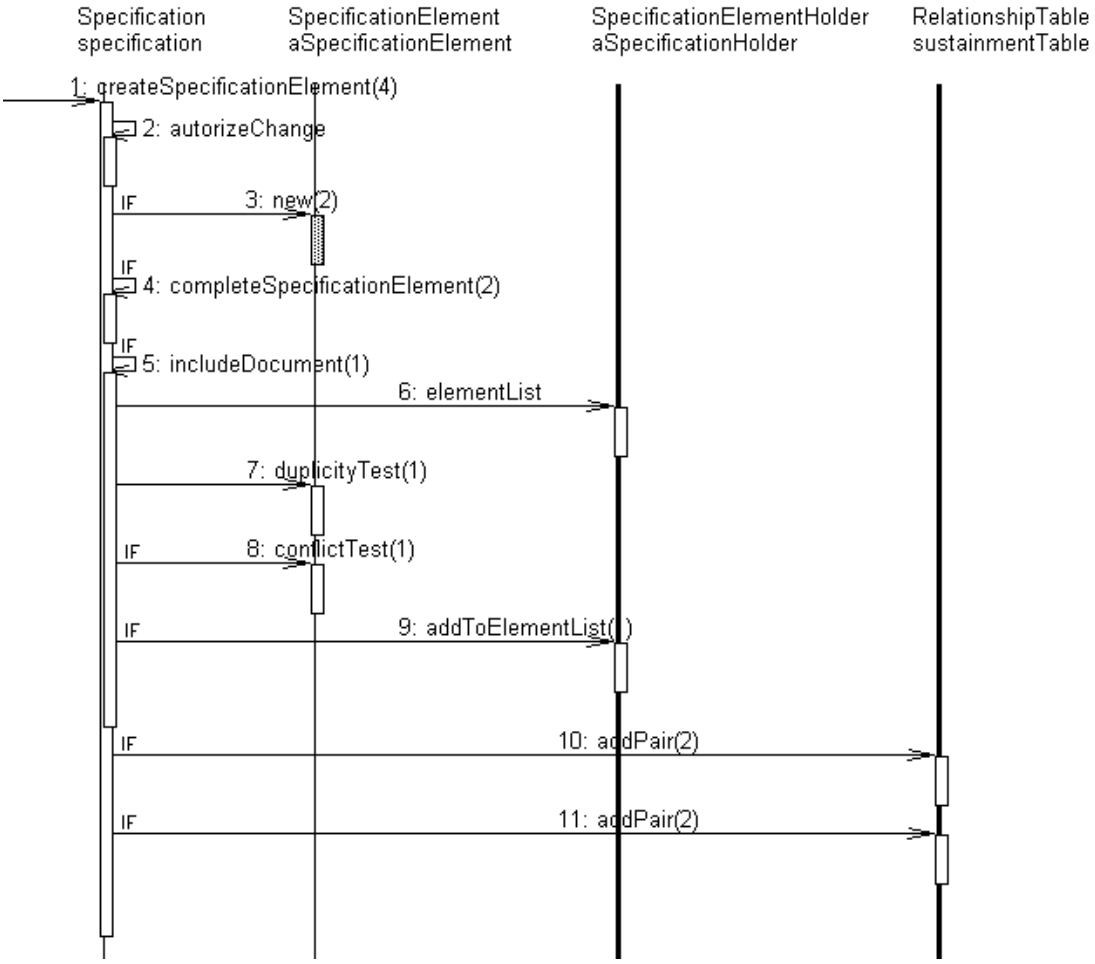

FIGURA 7.10 - Criação de elemento de especificação

O método construtor de uma classe que descreve um elemento de especificação deve apresentar parâmetros associados aos elementos sustentadores previstos para o tipo de elemento de especificação modelado[72]. O método construtor de herança (classe *Inheritance* de OCEAN, subclasse de *Concept*), por exemplo, tem como parâmetros as duas classes envolvidas, que são referenciadas pelos atributos *subclass* e *superclass* da instância criada.

No procedimento de criação de um elemento de especificação são feitos dois testes, teste de duplicidade e teste de conflito, e a criação só é efetuada se estes testes não indicarem impedimento (vide mensagens 7 e 8 da figura 7.10). Estes testes são implementados nas classes que descrevem elementos de especificação e o parâmetro é o conjunto de elementos do tipo testado, contidos na especificação tratada. O teste de duplicidade verifica se o elemento de especificação que está sendo criado apresenta uma coincidência de características (atributos com mesmo valor) com algum outro elemento de mesmo tipo presente na especificação, que não possibilite diferenciá-los - o que caracterizaria elementos duplicados. Exemplos de duplicidade seriam a criação de uma

72 Uma classe de elemento de especificação pode ter mais de um método construtor, com diferentes quantidades de parâmetros. A classe *Message* de OCEAN, por exemplo, que descreve mensagem de diagrama de seqüência, pode ser instanciada informando o elemento origem e o elemento destino ou apenas o elemento destino - dois métodos construtores, um com dois parâmetros e o outro com apenas um.

classe com o mesmo nome de uma classe existente ou a criação de um relacionamento de herança envolvendo as mesmas classes, com os mesmos papéis (subclasse e superclasse).

O teste de conflito verifica se o elemento criado produz uma inconsistência semântica na especificação, comparando-o com os elementos do mesmo tipo presentes na especificação tratada. Um exemplo de conflito seria a criação de uma herança produzindo um ciclo de herança.

No framework OCEAN estes testes apresentam uma implementação *default* na classe *Concept*, que retorna *false* para os dois testes (isto é, não-ocorrência de duplicidade ou conflito). Na criação de uma subclasse de *Concept* para descrever um tipo de elemento de especificação estes métodos devem ser sobrepostos, na medida em que haja a possibilidade de caracterizar duplicidade ou conflito para o tipo de elemento tratado. Na classe *Message* do framework OCEAN, por exemplo, os métodos de teste não são sobrepostos.

Na figura 7.10 observa-se que a primeira ação do objeto *specification*, quando invocado o método de criação de especificação, é invocar seu método *autorizeChange* (mensagem ordem 2). Este método faz parte do protocolo de controle acesso a especificações, que pode ser utilizado para restringir a possibilidade de alterar uma especificação - o que é necessário na construção de ambientes com múltiplos usuários, com diferentes níveis de permissão de acesso, assim como para proceder controle de versões (para bloquear a possibilidade de alterar uma versão de uma especificação).

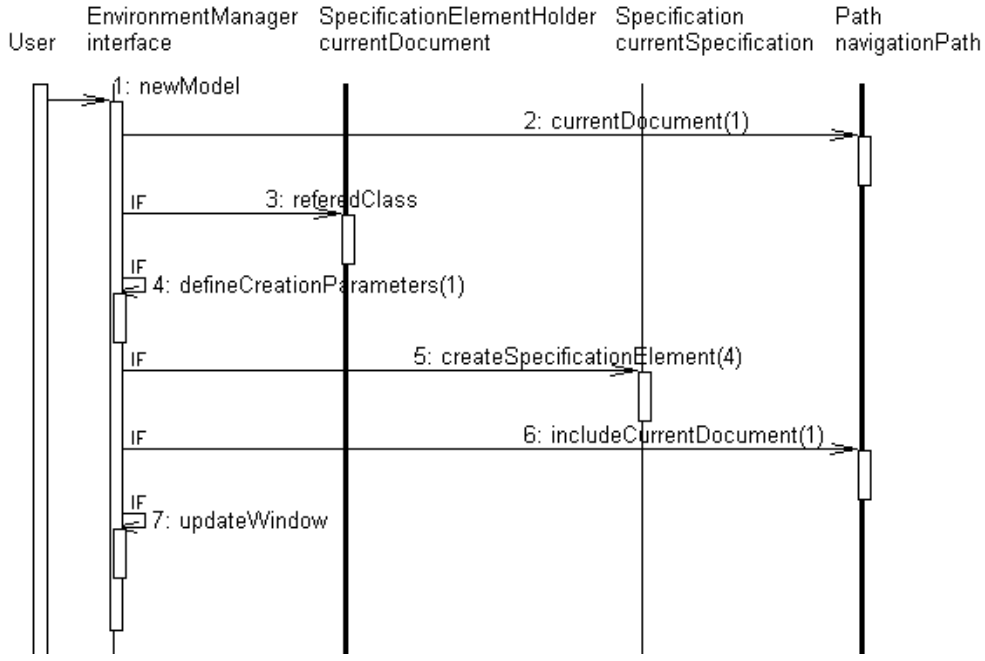

FIGURA 7.11 - Criação de modelo

A figura 7.11 apresenta um procedimento de criação de modelo. Este procedimento é invocado a partir da interface do ambiente e reusa o procedimento de criação de elemento de especificação descrito na figura 7.10. Para a criação de um modelo de um determinado tipo, o repositório daquele tipo de modelo (instância de *SpecificationElementHolder*, parte do repositório de modelos da especificação tratada) deve estar sendo apresentado na janela do ambiente (documento corrente). Uma vez criado e inserido na especificação, o novo modelo é inserido no registro de navegação do ambiente como documento corrente (vide item 7.1.3) e sua representação visual é

apresentada na janela do ambiente, inserida no editor gráfico associado. A figura 7.12 apresenta os elementos visuais associados a conceitos e modelos definidos no protótipo do framework OCEAN. Um modelo (instância de subclasse de *ConceptualModel*) agrega uma instância de subclasse de *SpecificationDrawing*, que agrega instâncias de subclasses de *Figure*, cada instância associada a um conceito (corresponde à representação visual de um conceito no diagrama gráfico associado a um modelo). *Drawing*, *Figure*, *LineFigure*, *CompositeFigure* e *TextFigure* são classes do framework HotDraw.

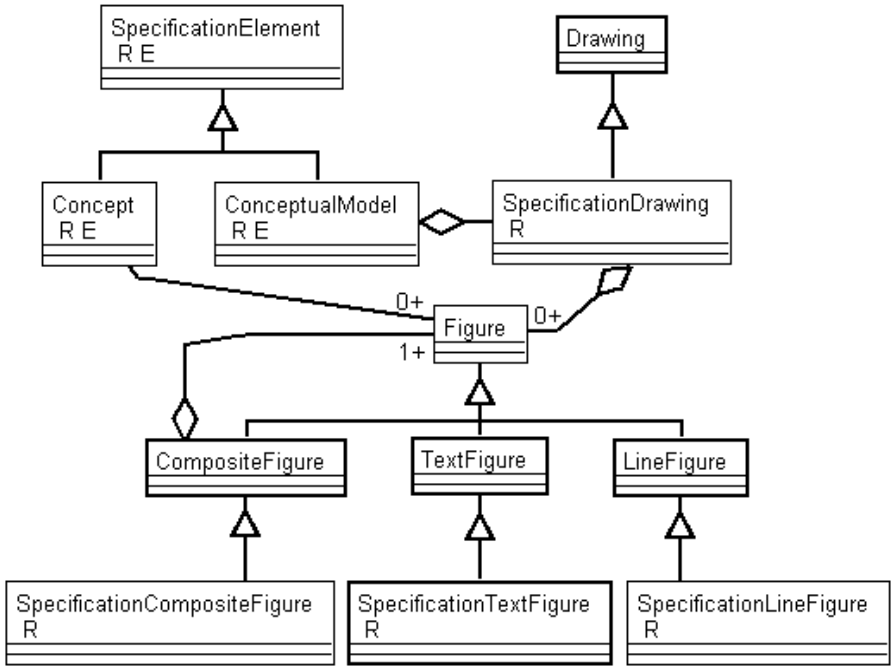

FIGURA 7.12 - Elementos visuais originados de HotDraw associados a conceitos e modelos

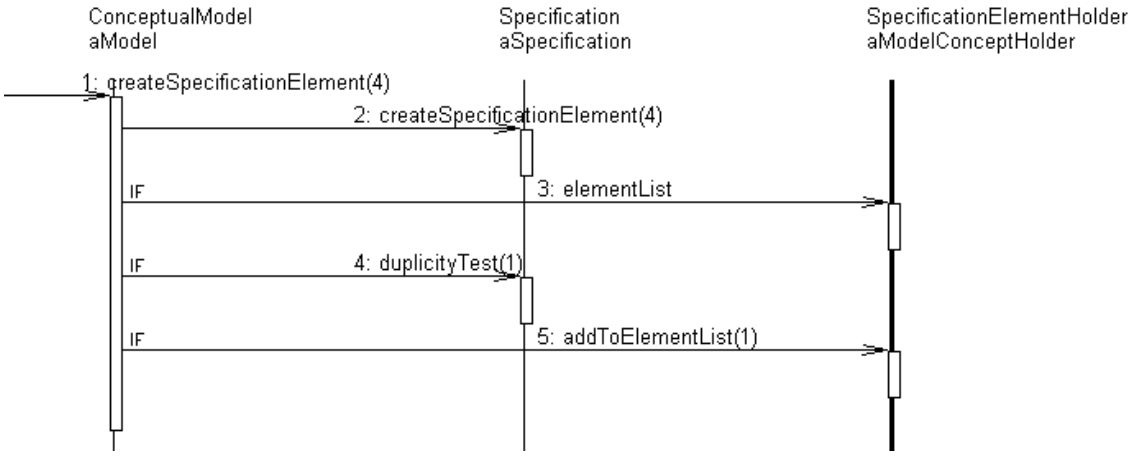

FIGURA 7.13 - Criação de conceito em modelo

A criação de um conceito durante a edição de um modelo, como a criação de uma classe durante a edição de um diagrama de classes, por exemplo, ocorre pela ação de uma ferramenta do editor gráfico (no capítulo anterior, no item 6.5.2, foram relacionados os editores e respectivas ferramentas para a construção de modelos de especificações OO). A figura 7.13 descreve o método de criação de conceito da classe *ConceptualModel*. Este método é invocado na atuação das ferramentas de criação de

conceito dos editores gráficos. A invocação de criação é repassada para a especificação (procedimento descrito na figura 7.10). O conceito retornado desta invocação é submetido a teste de duplicidade em relação ao conjunto de conceitos do mesmo tipo referenciados pelo modelo, antes de ser incluído na relação de conceitos referenciados. Quando este procedimento é invocado por uma ferramenta de criação de conceito de editor gráfico, caso o procedimento de criação seja bem sucedido, após a criação do conceito é criada a figura associada a ele e incluída no diagrama gráfico associado ao modelo sob edição.

7.2.2 Remoção de Conceitos e Modelos

Quando um modelo é removido, ele deixa de existir na especificação sob edição. A remoção de um conceito pode fazer o conceito deixar de existir na especificação ou simplesmente deixar de ser referenciado por um modelo. A figura 7.14 descreve o procedimento de remoção de um conceito na edição de um modelo, que pode corresponder a um dos dois casos. Se o único modelo que referencia o conceito for o modelo de que o conceito está sendo removido, a invocação da remoção é repassada para a especificação (neste caso a mensagem 6 é enviada e as mensagens de 7 a 11, não). Se o conceito for referenciado por mais de um modelo, a remoção fica restrita ao contexto do modelo sob edição (neste caso a mensagem 6 não é enviada e as demais, sim). Quando um conceito é removido de um modelo, todos os conceitos sustentados pelo conceito removido também são removidos do modelo sob edição (mensagem 9).

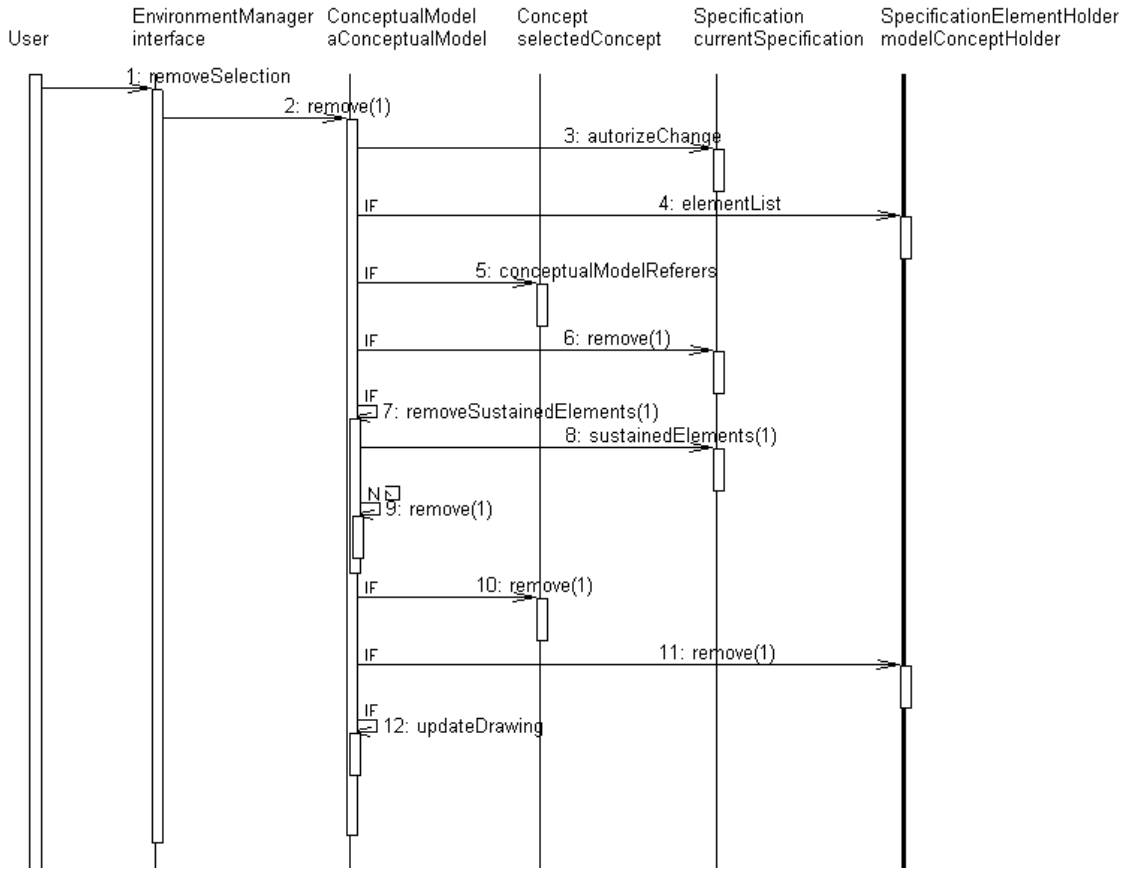

FIGURA 7.14 - Remoção de conceito na edição de modelo

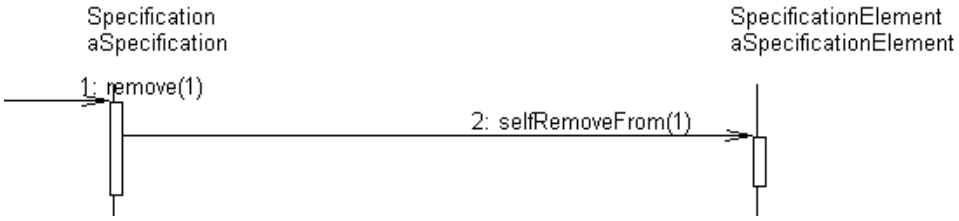

FIGURA 7.15 - Remoção de elemento de especificação de uma especificação

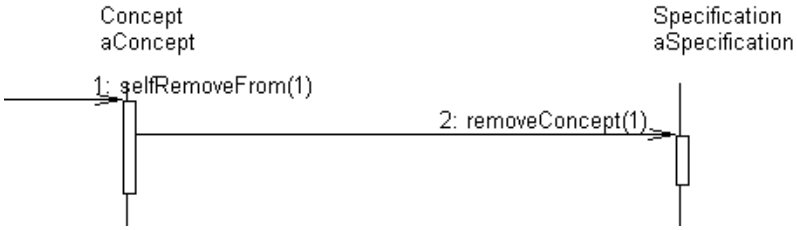

FIGURA 7.16 - Refinamento de selfRemoveFrom(aSpecificationElement) em Concept

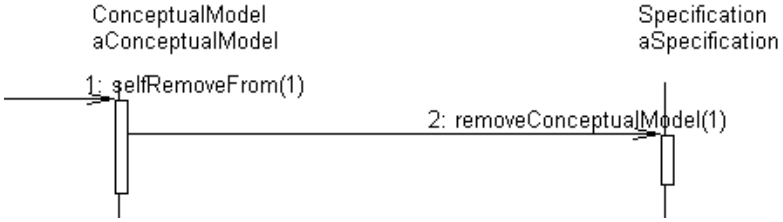

FIGURA 7.17 - Refinamento de selfRemoveFrom(aSpecificationElement) em ConceptualModel

A remoção de elementos de especificação de uma especificação é descrita no diagrama da figura 7.15, refinado pelos diagramas das figuras 7.16 e 7.17, que são refinados, respectivamente pelos diagramas das figuras 7.18 e 7.19. A remoção de conceito de especificação está descrita no diagrama da figura 7.18. Neste diagrama observa-se que quando um conceito é removido de uma especificação:

- todos os conceitos que têm o conceito removido como elemento referenciado são notificados da remoção (mensagem 5) e devem atualizar sua estrutura. Cada subclasse de *Concept* deve estabelecer o tratamento a ser dado pelo tipo de conceito descrito;
- todos os elementos de especificação que têm o conceito removido como elemento sustentador são removidos da especificação (mensagem 8);
- todos os elementos de especificação que têm o conceito removido como elemento sustentado são notificados da remoção (mensagem 11). Cada subclasse de *SpecificationElement* deve estabelecer o tratamento a ser dado ao tipo de elemento de especificação descrito, e se há necessidade de algum tratamento;
- o conceito é removido de todos os modelos que o referenciam (método *restrictedRemotion(concept)*, que simplesmente remove o conceito do modelo, sem propagação de efeito - diferente, portanto, do procedimento descrito na figura 7.14);
- todas as referências ao conceito são removidas das tabelas de sustentação e de referência (mensagens 13 e 14);
- o conceito é removido do repositório de conceitos da especificação (mensagem 15).

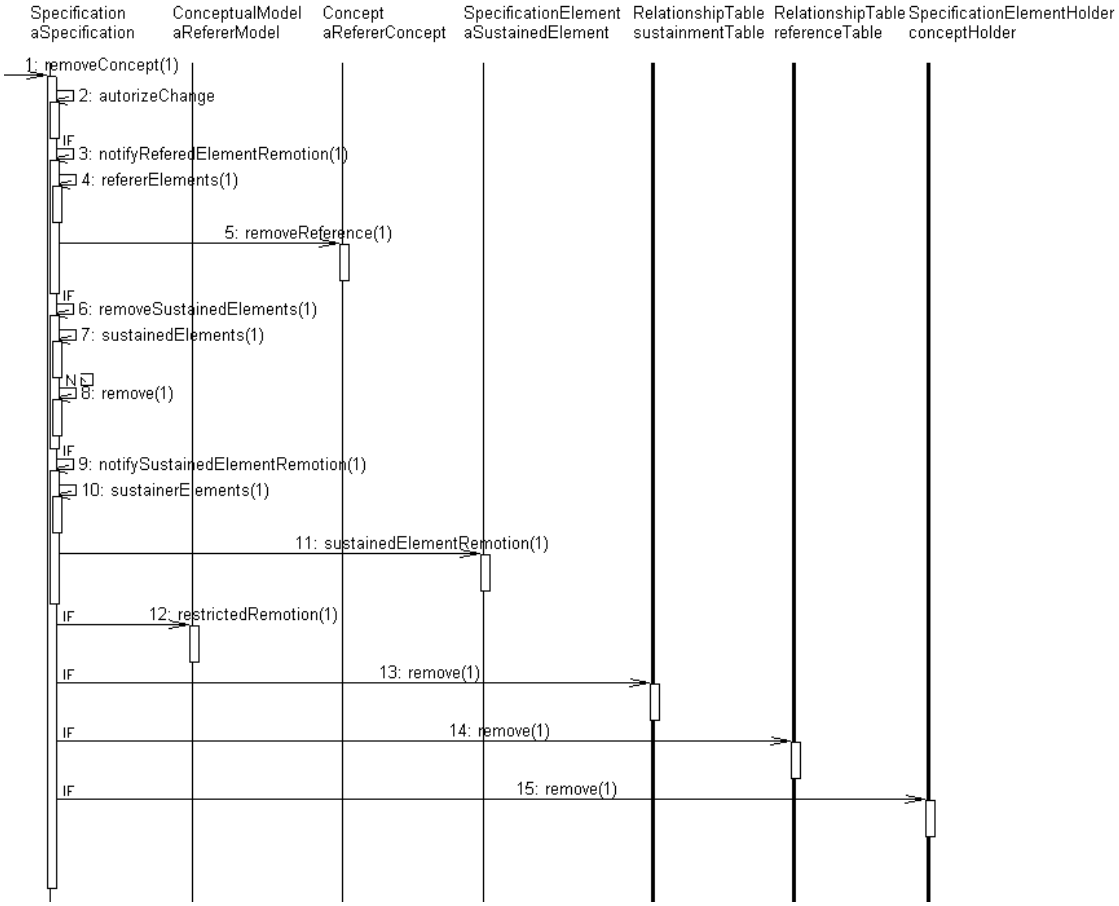

FIGURA 7.18 - Remoção de conceito de especificação

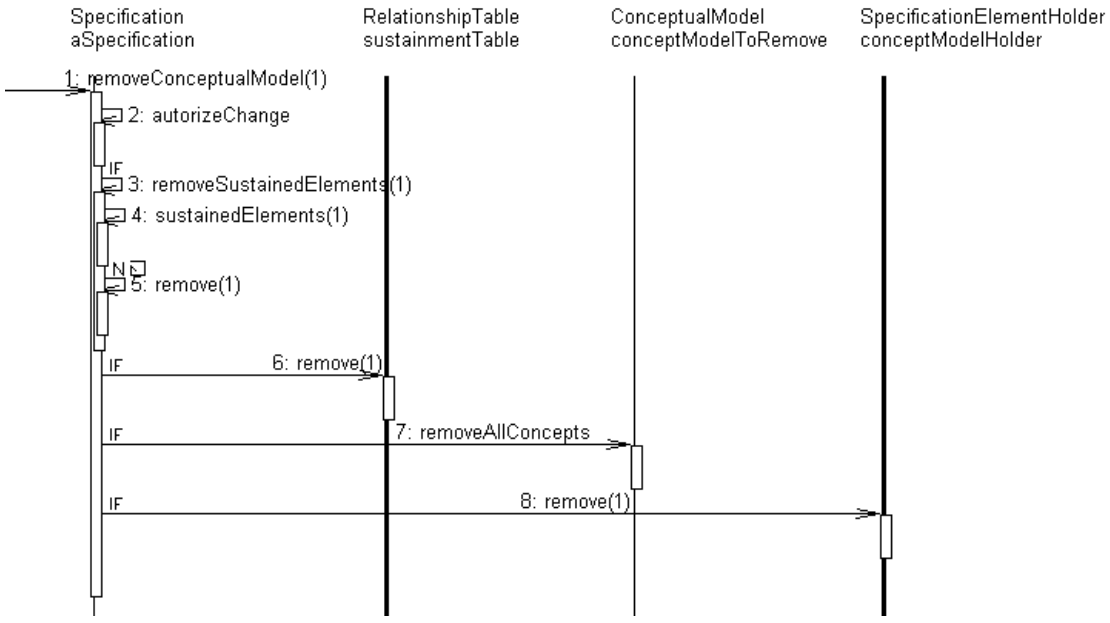

FIGURA 7.19 - Remoção de modelo de especificação

A figura 7.19 descreve a remoção de um modelo de uma especificação. Quando um modelo é removido de uma especificação:

- todos os elementos de especificação que têm o modelo removido como elemento sustentador são removidos da especificação (mensagem 5);

- todas as referências ao modelo são removidas da tabelas de sustentação (mensagem 6);
- o modelo é esvaziado, isto é, todos os conceitos referenciados são removidos do modelo (mensagem 7);
- o modelo é removido do repositório de modelos da especificação (mensagem 8).

7.2.3 Alteração de Conceitos

A alteração de um conceito é procedida em uma interface de edição específica para cada tipo de conceito tratado. Como descrito anteriormente, cada conceito manipulado em um ambiente deve estar associado a uma interface de edição, definida como uma subclasse de *ConceptInterface*. Quando um conceito é visualizado, a interface de edição referencia uma cópia do conceito manipulado, que pode ser submetida a ações de edição. A assimilação das alterações pelo conceito pertencente à especificação ocorre através da invocação do método *absorbChanges* da classe *ConceptInterface*, cuja execução é descrita no diagrama de seqüência da figura 7.20.

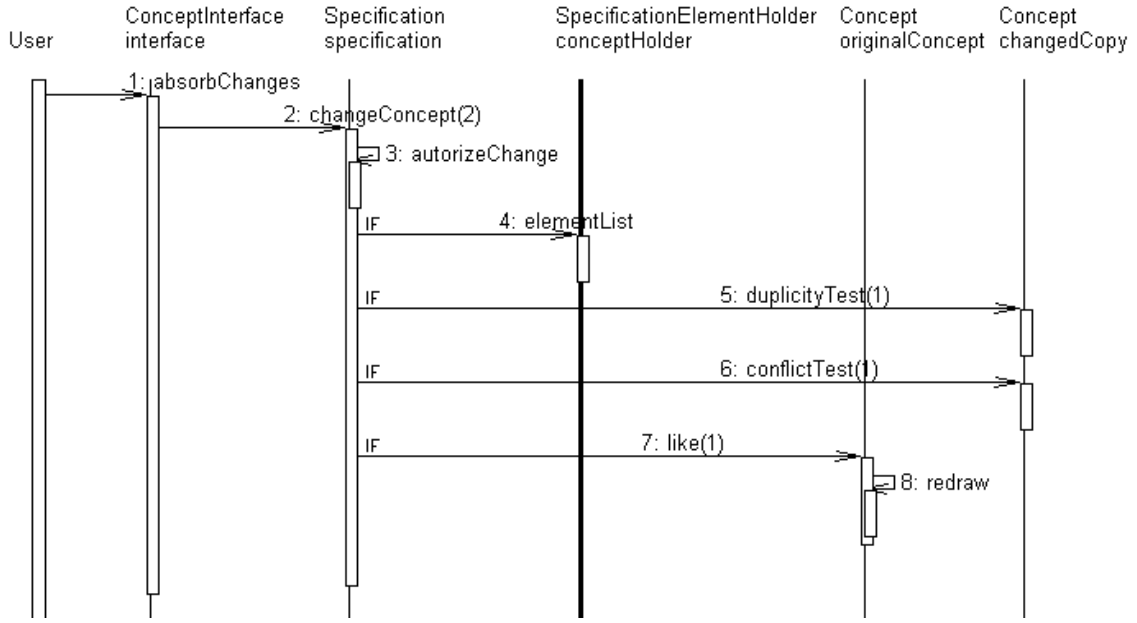

FIGURA 7.20 - Absorção das alterações por um conceito

Neste diagrama observa-se um procedimento de testes semelhante ao que ocorre na criação de um conceito, em que o conceito alterado (isto é, a cópia do conceito original) é submetido aos testes de duplicidade e conflito. Se os testes não acusarem problemas, é invocado o método *like(concept)* do conceito original, tendo a cópia alterada como parâmetro (mensagem 7). A sobreposição deste método é obrigatória em todas as subclasses concretas de *Concept*. Sua responsabilidade é alterar um conceito, baseando-se no conceito de mesmo tipo, passado na invocação do método.

O método redraw, referenciado pela última mensagem do diagrama da figura 7.20, é invocado para notificação da alteração (baseada no padrão de projeto *Observer*):

- às representações visuais associadas ao conceito alterado;
- aos elementos sustentados pelo conceito alterado;
- aos elementos sustentadores do conceito alterado;
- aos elementos que referenciam o conceito alterado.

7.2.4 Transferência e Fusão de Conceitos

Transferência e fusão de conceitos são funcionalidades de edição semântica supridas pelo framework OCEAN, utilizáveis em diferentes ambientes, aplicáveis a diferentes tipos de conceito. Estas funcionalidades estão baseadas na abordagem de organização de informações de especificações introduzida no framework OCEAN e alteram as relações semânticas entre elementos de especificação. A transferência de conceito consiste na troca do elemento sustentador de um conceito, como apresentado no capítulo anterior, em que foi descrito o uso de transferência no ambiente SEA.

A fusão de dois conceitos de mesmo tipo faz com que um dos conceitos seja removido da especificação e transfere associações de sustentação e referência do conceito removido para o segundo conceito envolvido na fusão.

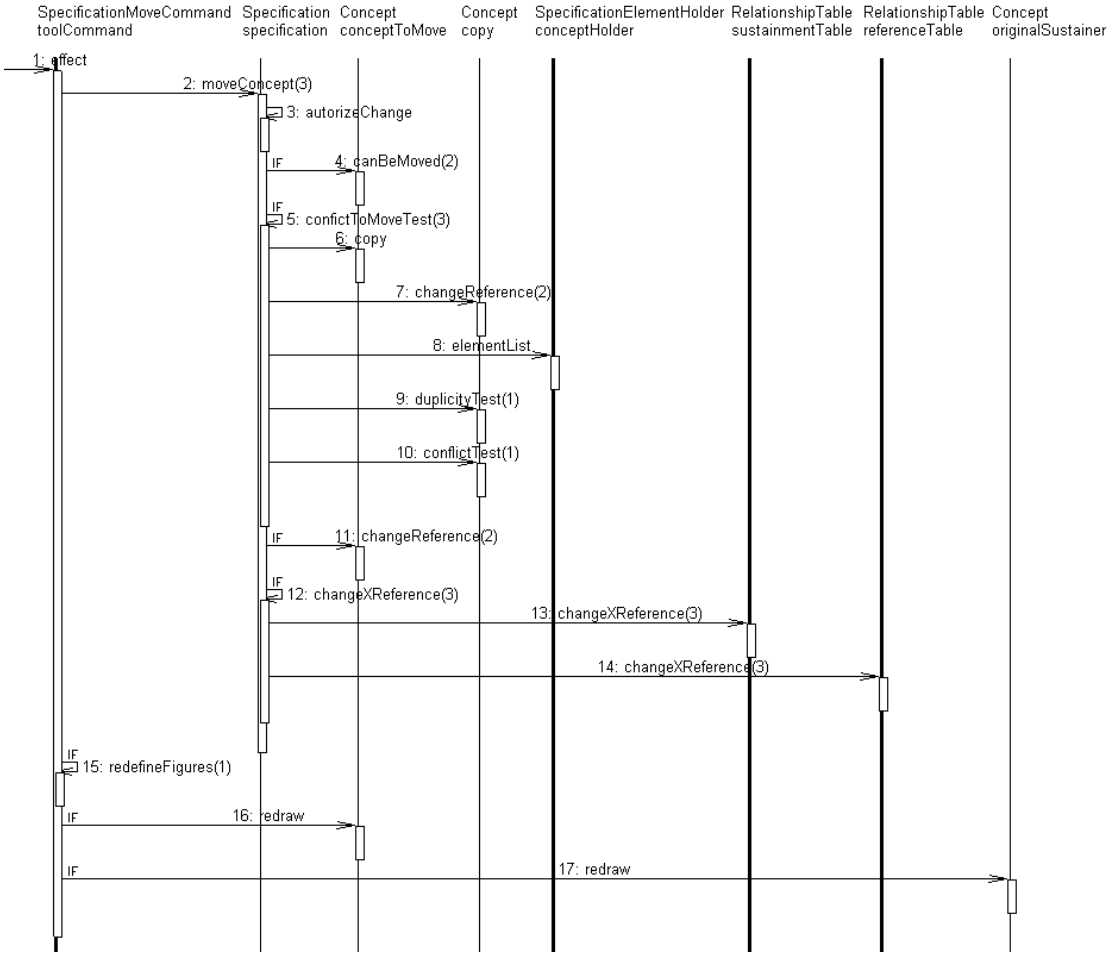

FIGURA 7.21 - Transferência de conceito

Na figura 7.21 o procedimento de transferência de conceito suportado pelo framework OCEAN é descrito através de um diagrama de seqüência. A transferência de conceito pode corresponder, por exemplo, à transferência de um atributo (conceito a transferir) de uma classe (elemento sustentador original) para outra classe (novo elemento sustentador). O procedimento é iniciado pela ação de uma ferramenta associada ao editor gráfico de um modelo que, ao arrastar o conceito a transferir do elemento sustentador original para o novo elemento sustentador, invoca o método *effect* da classe *SpecificationMoveCommand* do framework OCEAN, subclasse da classe *Command*, do framework HotDraw.

A primeira ação do método *effect*, invocado pela mensagem 1 da figura 7.21, é invocar o método *moveConcept* da classe *Specification*, tendo como argumentos o conceito a transferir, o elemento sustentador original e o novo elemento sustentador. Na execução deste método, isto é, quando uma especificação (instância de subclasse de *Specification*) procede a transferência de um conceito, esta especificação:

- através do protocolo de controle acesso a especificações, verifica se a especificação pode ser alterada (mensagem 3);
- verifica se o conceito admite a alteração (mensagem 4). O método *canBeMoved* de *Concept*, cujos parâmetros são a origem e o destino, deve ser sobreposto em todas as subclasses concretas de *Concept*. É responsabilidade de cada tipo de conceito decidir se pode ser transferido de um conceito para outro. Um atributo contido em uma especificação, por exemplo, só pode ser transferido de uma classe para outra classe.
- verifica se a transferência é possível, considerando a estrutura atual da especificação. Para isto é produzida uma cópia do conceito a transferir, a cópia é alterada (mensagem 7) e é testada a possibilidade de incluí-la na especificação - procedimento de teste semelhante ao que é realizado na criação de elemento de especificação, descrito no diagrama da figura 7.10. O método *changeReference*, com os mesmos parâmetros do método *canBeMoved*, utilizado para alterar a cópia do conceito e posteriormente o próprio conceito, procede a alteração da estrutura do conceito em função dos parâmetros de transferência. Um atributo, por exemplo, referencia a classe que o sustenta. Na execução de *changeReference* ele passa a referenciar a classe para a qual está sendo transferido. Este método deve ser sobreposto em cada subclasse concreta de *Concept*.
- após o conjunto de verificações, se constatado que a transferência é realizável, a estrutura do conceito é alterada (método *changeReference*, mensagem 11) e as relações semânticas registradas nas tabelas de sustentação e referência são alteradas - a associação entre o elemento sustentador original do conceito transferido deixa de existir e é criada uma associação entre conceito transferido e o novo elemento sustentador.

Após efetuada a transferência, as figuras que representam o conceito transferido são redefinidas e é invocado o método *redraw* do conceito transferido e do elemento sustentador original, para propagação da alteração ao restante da especificação.

O procedimento de fusão de dois conceitos está descrito no diagrama de seqüência da figura 7.22. Este procedimento pode corresponder, por exemplo, à fusão de duas classes de uma especificação. Semelhante à transferência de conceito, a fusão é iniciada pela ação de uma ferramenta de fusão de um editor gráfico de modelo, que invoca o método *effect* da classe *SpecificationMergeCommand* de OCEAN (subclasse de *Command*, de HotDraw). Este método invoca o método *mergeConcept* da classe *Specification*, que procede a fusão - os parâmetros deste método são o conceito a fundir, o conceito em que este conceito será fundido e o conceito origem do conceito a fundir[73]. Após executada a fusão, a figura do conceito envolvido que persiste após o procedimento é redefinida (mensagem 22) e é invocado o método redraw deste conceito, para propagação da alteração (mensagem 23). O procedimento de fusão além de poder alterar a estrutura dos dois conceitos envolvidos (um dos quais é removido da especificação), também, altera as relações semânticas registradas nas tabelas de

---

[73] Conceito origem é um conceito que é elemento sustentador do conceito considerado. É responsabilidade de cada tipo de conceito retornar seu conceito origem, considerada a possibilidade de notificar que não apresenta conceito origem.

sustentação e referência da especificação tratada. A figura 7.23 descreve o efeito do procedimento de fusão sobre as associações de sustentação e referência que envolvem o conceito a fundir (a representação diagramática não corresponde a um modelo gerado no ambiente SEA).

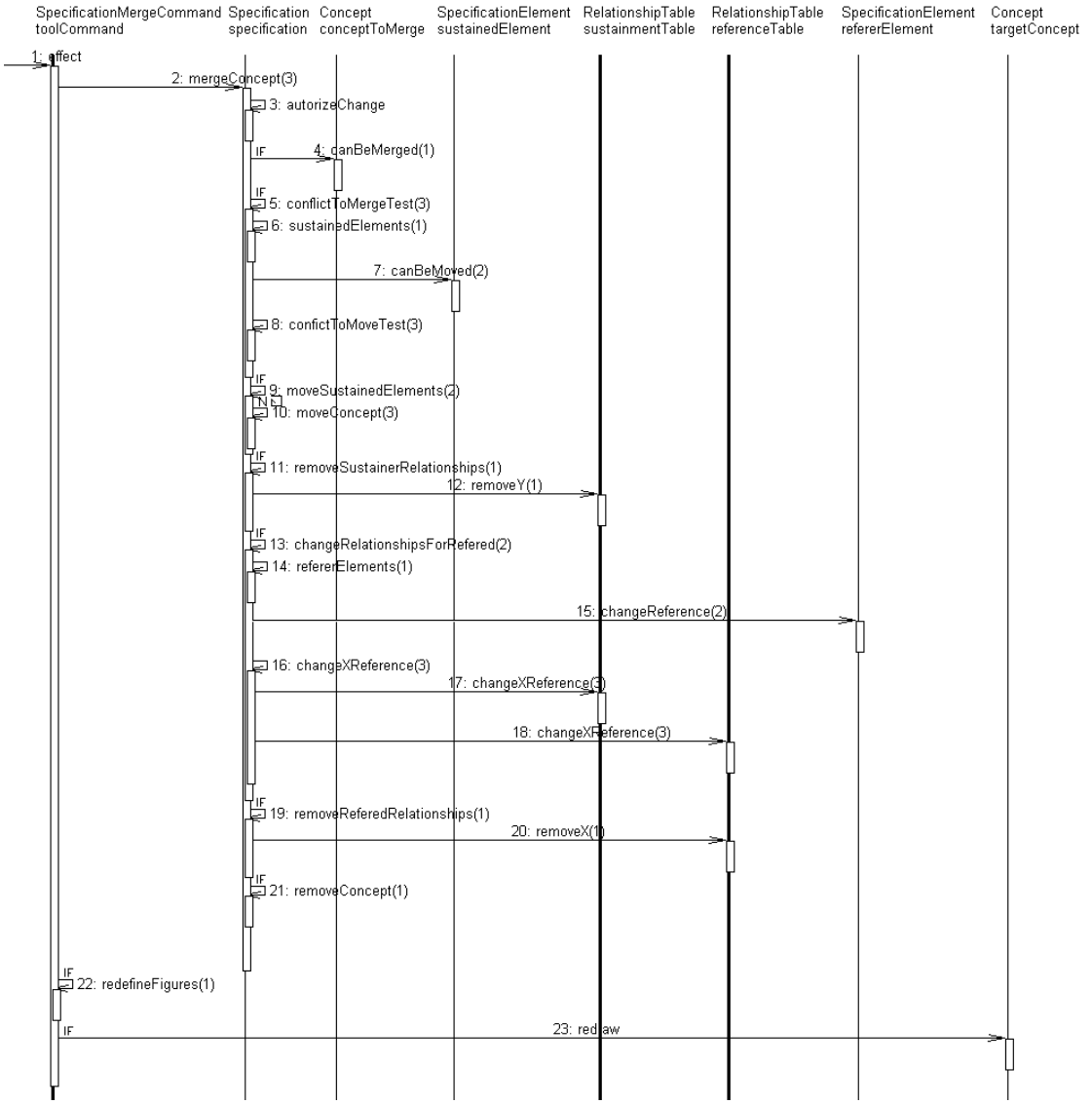

FIGURA 7.22 - Fusão de conceitos

Na execução do método *mergeConcept*, isto é, quando uma especificação (instância de subclasse de *Specification*) procede a fusão de um conceito em outro, esta especificação:

- verifica se a especificação pode ser alterada (mensagem 3);
- verifica se o conceito a fundir admite a alteração (mensagem 4). O método *canBeMerged* de *Concept*, cujo parâmetro é o conceito em que ocorrerá a fusão, deve ser sobreposto em todas as subclasses concretas de *Concept*. É responsabilidade de cada tipo de conceito decidir se pode ser fundido em outro conceito. Além de só ser possível fundir conceitos de mesmo tipo (não faz sentido fundir um atributo e um método, por exemplo), podem ser estabelecidos outros requisitos como a necessidade de terem o mesmo elemento sustentador (no framework OCEAN, apenas atributos da mesma classe podem ser fundidos, por

exemplo) ou requisitos específicos de cada tipo de conceito (como a necessidade de superclasse comum para a fusão de classes, por exemplo);

- verifica se a transferência é possível (mensagem 7), invocando o método *noConflictToMergeTest*, com os mesmos parâmetros do método *mergeConcept*, que, basicamente, verifica se todos os elementos sustentados pelo conceito a fundir podem ser transferidos para o outro conceito envolvido na fusão.
- após o conjunto de verificações, se constatado que a fusão é realizável, ocorrem as alterações das associações de sustentação e referência:
  - ⇨ os conceitos sustentados pelo conceito a fundir são transferidos para o outro conceito envolvido na fusão (mensagem 9), o que corresponde ao procedimento de transferência de conceitos anteriormente descrito. Com isto, as associações em que o conceito a fundir corresponde ao elemento sustentador são substituídas por outras em que o outro conceito envolvido na fusão passa a ser o elemento sustentador (na figura 7.23 corresponde à associação entre os conceitos A e D, que é substituída por uma associação entre os conceitos B e D);
  - ⇨ as associações em que o conceito a fundir corresponde ao elemento sustentado são removidas da tabela de sustentação da especificação (vide mensagem 11 do diagrama da figura 7.22 e a associação entre os conceitos A e C da figura 7.23)
  - ⇨ as associações em que o conceito a fundir corresponde ao elemento referenciado são substituídas por outras em que o outro conceito envolvido na fusão passa a ser o elemento referenciado (mensagem 16 do diagrama de figura 7.22, que na figura 7.23 corresponde à substituição da associação entre os conceitos A e E por uma associação entre os conceitos B e E);
  - ⇨ as associações em que o conceito a fundir corresponde ao elemento referenciado são removidas da tabela de referência da especificação (vide mensagem 19 do diagrama da figura 7.22 e associação entre os conceitos A e F da figura 7.23);
- Após a alteração das associações de sustentação e referência, o conceito a fundir, que já não faz parte de nenhuma associação, é removido da especificação (mensagem 21 do diagrama de figura 7.22).

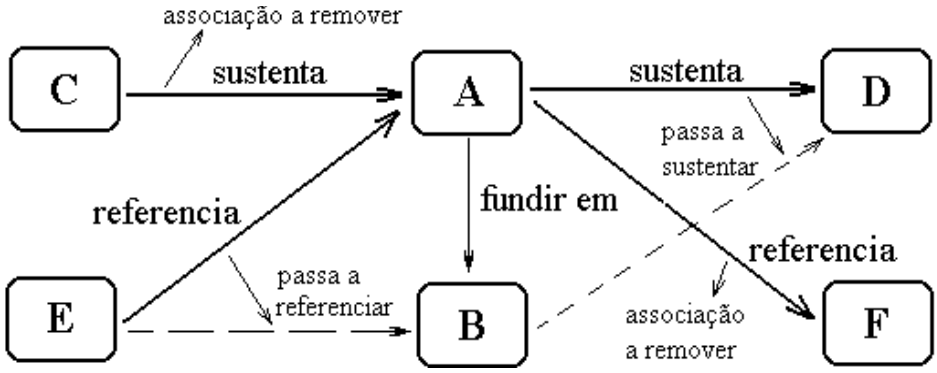

FIGURA 7.23 - Conseqüências da fusão do conceito A no conceito B

7.2.5 Cópia e Colagem de Conceitos

Cópia e colagem de conceitos são funcionalidades complementares que permitem reusar a estrutura de conceitos existentes para a criação de novos conceitos. Um método associado a uma classe, por exemplo, pode ser copiado na área de transferência de um ambiente e colado em outra classe - o que corresponde à criação de um novo método com outra classe como elemento sustentador. No ambiente SEA, por exemplo, uma

classe pertencente a uma especificação pode ser copiada e colada em um diagrama de classes da mesma ou de outra especificação.

O procedimento de cópia utiliza como área de transferência o mesmo mecanismo utilizado para registrar o caminho de navegação em um ambiente (uma instância da classe *MemoryCell* - vide figura 7.5). Apesar de ser utilizada a expressão *cópia*, o procedimento de cópia de um conceito suportado pelo framework OCEAN consiste em fazer com que a área de transferência referencie o conceito. Além do conceito, a área de transferência também passa a referenciar a especificação a que o conceito pertence. A área de transferência pode referenciar um conjunto de conceitos, desde que de uma mesma especificação.

A colagem de conceitos referenciados pela área de transferência está descrita nos diagramas das figuras 7.24 e 7.25. O diagrama da figura 7.25 refina a execução do método *paste*, da classe *ConceptualModel*, invocado no diagrama da figura 7.24 (nas mensagens 10 e 12). O diagrama da figura 7.24 descreve o procedimento de colagem iniciado por iniciativa do usuário, que seleciona a função na interface do ambiente. No procedimento de colagem correspondente à execução do método *paste* da classe *EnvironmentManager*, ocorre:

- levantamento da especificação e do modelo correntes (isto é, o modelo carregado na janela do ambiente e a especificação a que ele pertence). O procedimento de colagem só ocorre se o documento corrente for um modelo;
- levantamento dos documentos apontados pela área de transferência, isto é, especificação e conceito (ou conjunto de conceitos);
- verificação se a especificação corrente pode ser alterada (mensagem 6);
- verificação se o modelo em que será feita a colagem admite a alteração (mensagem 7). O método *canPaste* de *ConceptualModel*, cujo parâmetro é o conceito ou conjunto de conceitos a colar, deve ser sobreposto em todas as subclasses concretas de *ConceptualModel*. É responsabilidade de cada tipo de modelo decidir se admite a colagem de um conceito. Um diagrama de classes, por exemplo, admite a colagem de uma classe ou de um método, mas não admite a colagem de um estado;
- ordenamento do conjunto de conceitos a colar, o que só é necessário caso a área de transferência aponte um conjunto de conceitos (mensagem 8). É responsabilidade de cada tipo de modelo definir a ordem adequada de colagem de um conjunto de conceitos. Ao colar uma classe e um atributo associado a esta classe em um diagrama de classes, por exemplo, é preciso que a colagem da classe ocorra primeiro;
- definição do conceito que efetivamente será colado (mensagem 9). Também é responsabilidade de cada tipo de modelo definir o conceito que será usado na colagem, considerando o conceito apontado pela área de transferência, a especificação a que ele pertence, a especificação corrente e o conceito selecionado no modelo, caso algum conceito tenha sido selecionado (informações passadas como parâmetros). A implementação default do método *defineConceptToPaste* contida na classe *ConceptualModel*, por exemplo, corresponde a invocar o método *cloneToPaste* do conceito tratado. Este método produz uma cópia do conceito apontado, cujos atributos que apontam conceitos ou apontam conceitos pertencentes à especificação corrente ou nenhum conceito.
- após a definição do conceito a colar, que pode ser o próprio conceito apontado pela área de transferência ou uma cópia produzida como descrito acima, é invocado o método *paste* da classe *ConceptualModel*, passando como parâmetros o conceito a colar, a especificação corrente e o conceito selecionado no modelo, caso algum conceito tenha sido selecionado (procedimento refinado no diagrama da figura 7.25).

Este procedimento, precedido pelo procedimento de definição do conceito a colar é invocado para cada conceito apontado pela área de transferência. O caso descrito no diagrama da figura 7.24 supõe a colagem de dois conceitos.

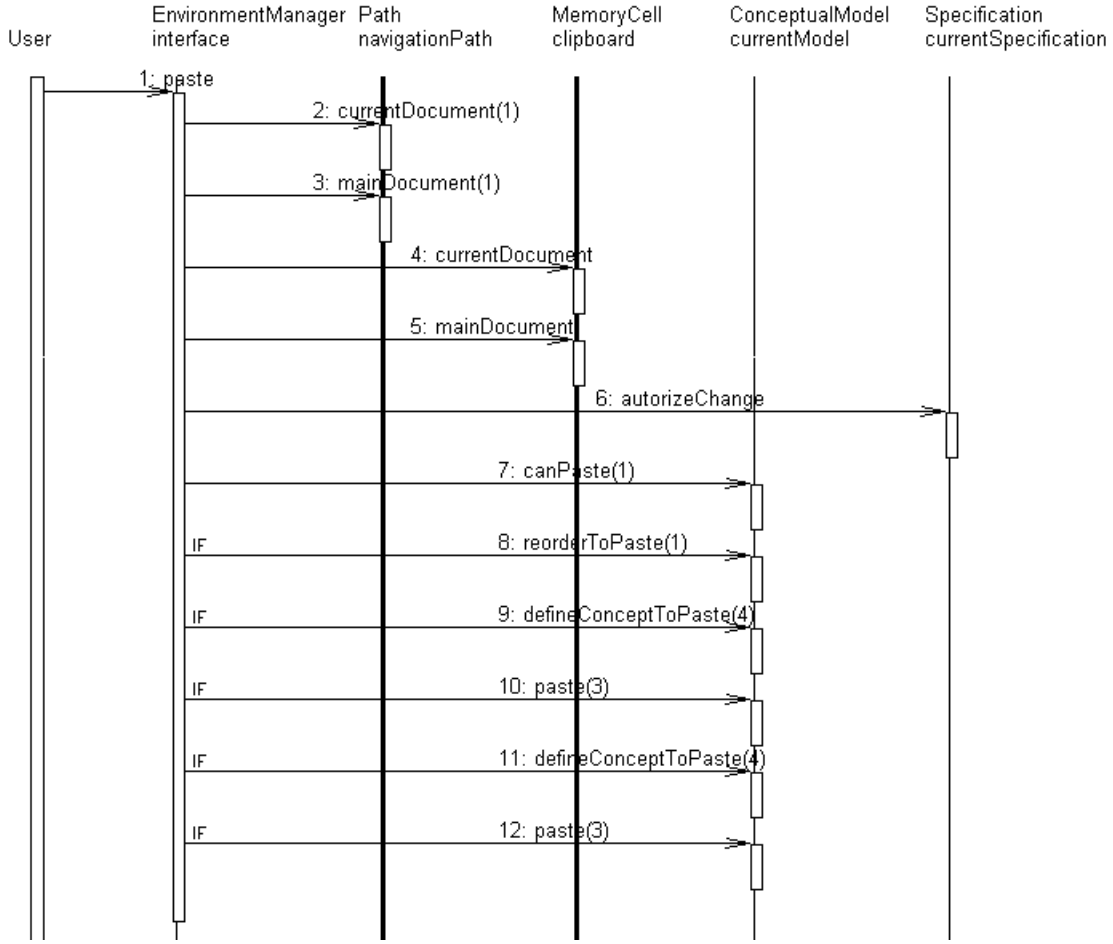

FIGURA 7.24 - Colagem de conceitos

Na colagem de um conceito procedida por um modelo, descrita no diagrama de seqüência da figura 7.25, ocorre:

- verificação se o conceito a colar já faz parte da especificação corrente (mensagem 2);
- caso o conceito não esteja incluído na especificação, ele é incluído - o mesmo procedimento de inclusão que ocorre na criação de um elemento de especificação (mensagem 3);
- caso o conceito tenha sido incluído na especificação, como descrito no passo anterior, é invocado o método *defineRelationships* para a especificação estabelecer associações de sustentação (em que o conceito inserido é o elemento sustentado) e de referência (em que o conceito inserido é o elemento referenciador);
- é invocado o método *paste* de *ConceptualModel*, com o conceito a colar como parâmetro - um método diferente daquele invocado na mensagem 1. Neste procedimento o conceito é incluído no modelo (semelhante ao procedimento de inclusão que ocorre na criação de um conceito em um modelo), é criada uma figura para o conceito colado (responsabilidade delegada ao diagrama associado ao modelo) e é enviado o método redraw ao conceito colado, para propagação da alteração.

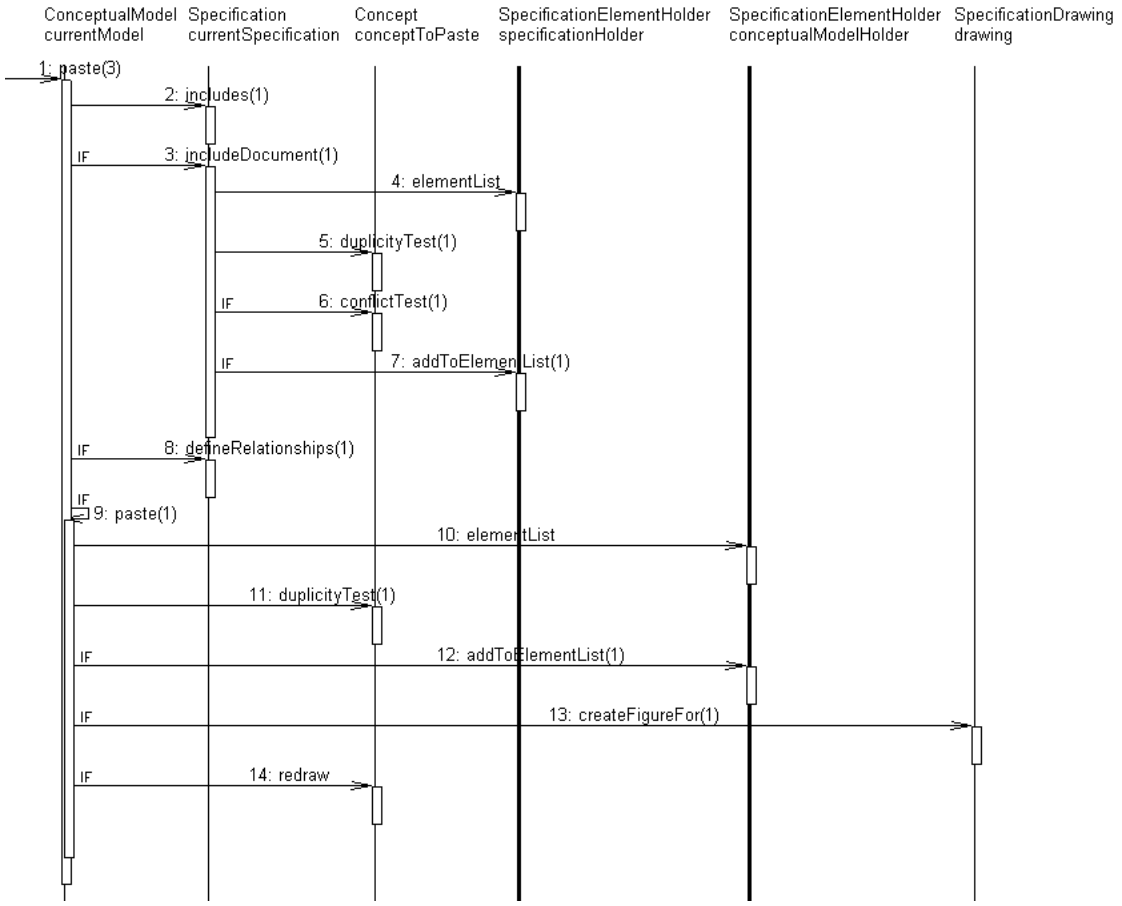

FIGURA 7.25 - Refinamento do método paste (concept, specification, selectedConcept) da classe ConceptualModel

7.3 Suporte à Composição de Ações de Edição Complexas através de Ferramentas

Nos itens anteriores deste capítulo foi apresentada a abordagem de organização de especificações do framework OCEAN, responsável pela flexibilidade para a definição de diferentes estruturas de especificação, e foi descrito como OCEAN suporta um conjunto de funcionalidades de edição semântica. A seguir, é descrito como o framework OCEAN suporta flexibilidade para inclusão de estruturas funcionais, denominadas genericamente de *ferramentas*, que reutilizam os procedimentos de edição semântica supridos pelo framework.

A figura 7.26 apresenta o conjunto das classes relacionadas a ferramentas em um ambiente. Todo ambiente específico (definido a partir de uma subclasse de *EnvironmentManager*) agrega um gerente de ferramentas (instância da classe *ToolManager* ou de uma subclasse). Na inicialização de um ambiente é definido o conjunto de ferramentas utilizáveis. Ferramentas são construídas como subclasses de *OCEANTool* e são associadas a um ou mais tipos de especificação (para que foram construídas) e podem estar associadas a um ou mais tipos de modelo (se forem específicas para atuar sobre determinados tipos de modelo). As ferramentas de inserção de padrões de projeto, de importação de interfaces de componentes e de verificação de consistência, apresentadas no capítulo anterior, são exemplos de ferramentas desenvolvidas sob a estrutura de suporte do framework OCEAN.

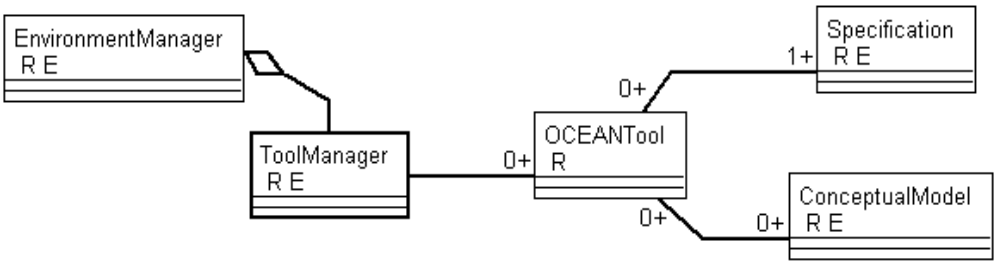

FIGURA 7.26 - Estrutura de suporte à produção de ferramentas

A seguir, é descrita a operação de um conjunto de exemplos de ferramentas que ilustram como estruturas funcionais complexas podem ser obtidas a partir do suporte funcional provido pelo framework OCEAN.

7.3.1 Ações da Ferramenta de Inserção de Padrões de Projeto do Framework OCEAN

A ferramenta de inserção de padrões, pertencente à estrutura do framework OCEAN e utilizada no ambiente SEA, constitui um exemplo de ferramenta desenvolvida a partir do suporte de desenvolvimento do framework OCEAN. *PatternTool*, a classe em que estão definidas as funcionalidades da ferramenta, é subclasse de *OCEANTool*, referencia *OOSpecification* (o tipo de especificação em que pode atuar) e *ClassDiagram* (o tipo de modelo em que sua ação pode ser invocada[74]) e na inicialização do ambiente SEA, é incluída no conjunto de ferramentas disponíveis, referenciado pelo gerente de ferramentas do ambiente. Como descrito no capítulo anterior, esta ferramenta reproduz a estrutura de um padrão de projeto (definida como uma especificação) em uma especificação OO em desenvolvimento. Além da reprodução dos elementos de especificação pertencentes à estrutura do padrão de projeto, a ação da ferramenta de inserção de padrões pode envolver fusão de classes e ações de edição adicionais.

O diagrama de seqüência apresentado na figura 7.27 descreve o procedimento de inserção de um padrão. O método *includeSelectedPattern* da classe *PatternTool* é invocado após a seleção de um dos padrões da biblioteca de padrões disponibilizada e após o procedimento de associação das classes do padrão selecionado às classes da especificação em desenvolvimento (isto é, para cada classe do padrão de projeto, se a classe corresponderá a uma classe presente na especificação tratada ou a uma nova classe). No procedimento de associação de classes é composta uma estrutura de dados que registra a associação e que é passada como parâmetro nas mensagens 4 e 9 do diagrama. Na execução do método *includeSelectedPattern*:

- é obtida do gerenciador de referência de padrões a ferramenta de teste de inserção associada ao padrão selecionado (mensagem 2);

---

74 A associação da ferramenta ao tipo de modelo *diagrama de classes* (correspondente à classe ClassDiagram) estabelece que a ferramenta só pode ser carregada na janela do ambiente quando o documento corrente for um diagrama de classes. O diagrama de classes carregado é o modelo em que ocorre a colagem das classes do padrão durante o procedimento de inserção. Apenas a necessidade de definir o modelo em que incluir as classes originou a necessidade de associar a ferramenta a um tipo de modelo, o que não significa que sua ação esteja restrita a este modelo.

- é obtida do gerenciador de referência de padrões a ferramenta de adaptação associada ao padrão selecionado (mensagem 3). Esta ferramenta atua após os procedimentos de inserção dos elementos de especificação e de fusão de conceitos[75];
- é verificada a viabilidade da inserção do padrão selecionado na especificação em desenvolvimento, considerada a associação definida entre classes do padrão e da especificação (mensagem 4);
- é obtida a ferramenta responsável por reproduzir a estrutura de um padrão selecionado em uma especificação (mensagem 5);
- esta ferramenta é acionada para reproduzir os elementos de especificação selecionado na especificação em desenvolvimento (mensagem 6);
- conceitos *tipo* relacionados (sustentados por classes relacionadas) são fundidos (mensagem 7);
- conceitos *classe* relacionados são fundidos (mensagem 8);
- a ferramenta de adaptação é acionada para concluir a inserção do padrão (mensagem 9).

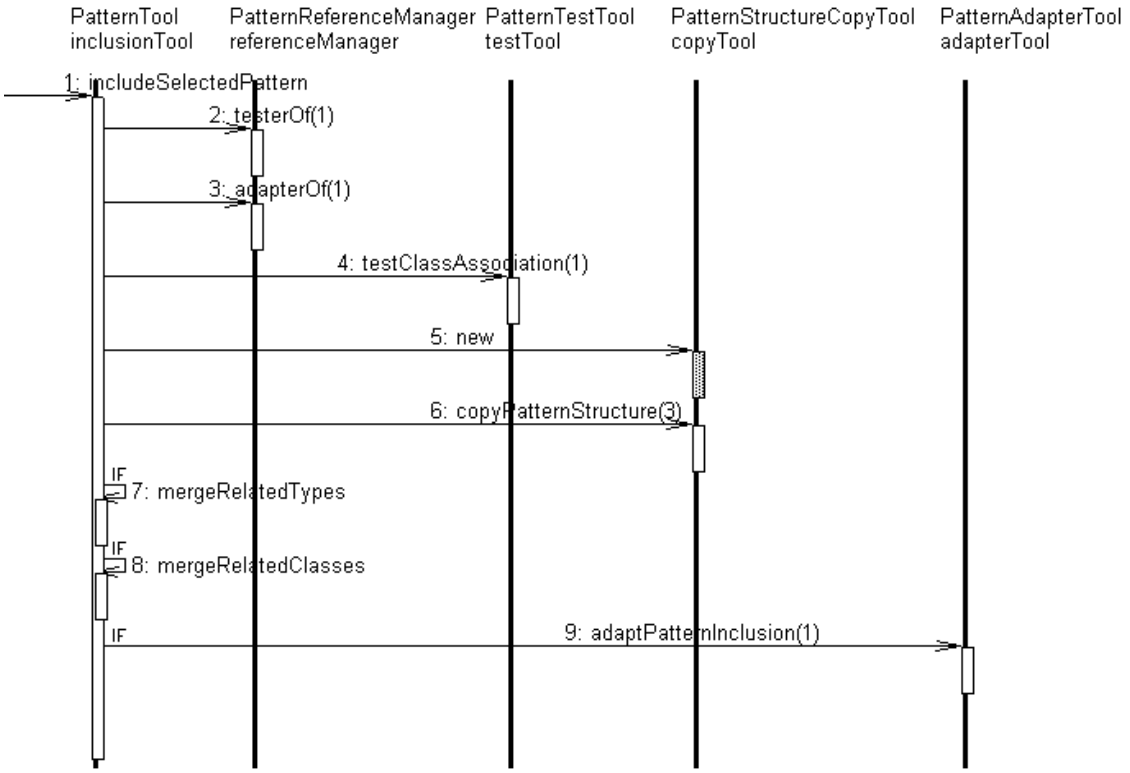

FIGURA 7.27 - Ação da ferramenta de inserção de padrões de projeto

A figura 7.28 destaca um método da classe *ClassCopyTool* que é invocado durante a atuação da ferramenta que procede a reprodução de estrutura de padrões, *PatternStructureCopyTool* (durante a execução do método referido na mensagem 6 do diagrama de seqüência da figura 7.27). É apresentada a implementação do método

---

75 No capítulo anterior, na descrição do uso da ferramenta de inserção de padrões no ambiente SEA (item 6.4.8), foi descrita a necessidade desta funcionalidade e um exemplo de adaptação foi apresentado utilizando o padrão de projeto *Proxy*.

contida no framework OCEAN, utilizando sintaxe Smalltalk[76]. O método procede a cópia de uma classe da especificação do padrão (*originSpecification*) na especificação em desenvolvimento (*targetSpecification*) e invoca os métodos que copiam os diagramas de corpo de método associados aos métodos da classe tratada (linha 11, na figura 7.28) e os diagramas de transição de estados associados à classe (linha 12). Observa-se no método de cópia de classe o uso das funcionalidades de cópia e colagem supridas pelo framework OCEAN. A classe do padrão é copiada na área de transferência do ambiente em operação (funcionalidade de cópia é usada nas linhas 4 e 8) e colada na especificação que está sendo editada (linha 10).

Classe: ClassCopyTool

```
copyClass: aClass ofModel: anOriginClassDiagram from: originSpecification
toModel: aTargetClassDiagram from: targetSpecification
1.  | envManager auxList |
2.  envManager := SingleConnector manager.
3.  ((aClass inheritance) isNil)
4.  ifTrue:  [envManager copy: aClass from: originSpecification containedIn: anOriginClassDiagram]
5.  ifFalse: [auxList := List new.
6.            auxList add: aClass.
7.            auxList add: (aClass inheritance).
8.            envManager copy: auxList from: originSpecification containedIn: anOriginClassDiagram].
9.  envManager updateWindow.
10. envManager paste.
11. self copyMethodBodyDiagramsOf: aClass from: originSpecification to: targetSpecification.
12. self copyStateTransitionDiagramsOf: aClass from: originSpecification to: targetSpecification.
```

FIGURA 7.28 - Código de método da classe ClassCopyTool

Classe: PatternTool

```
mergeRelatedClasses

•••
1.  (targetSpecification notNil) ifTrue: [
2.   auxList do: [:aKey |
3.      originClass := targetSpecification conceptOfClass: OCEANClass withName: aKey.
4.      targetClass := targetSpecification conceptOfClass: OCEANClass withName:
                                                  (self correlationDictionary at: aKey).
5.      ((originClass notNil) & (targetClass notNil))
6.          ifTrue:        [self ajustInheritanceOf: originClass toPasteWith: targetClass
                                                              in: targetSpecification.
7.       (targetSpecification mergeConcept: originClass from: nil in: targetClass)
•••
```

FIGURA 7.29 - Código de método da classe PatternTool

A figura 7.29 apresenta um trecho do código do método *mergeRelatedClasses* da classe *PatternTool*, invocado na mensagem 8 do diagrama de seqüência da figura 7.27. Este método procede a fusão das classes que foram coladas na especificação sob edição (copiadas do padrão selecionado) nas classes a elas associadas. No trecho destacado na

---

[76] A numeração das linhas não é parte do código. Foi acrescida no código da figura 6.28 e nas figuras posteriores do presente capítulo para facilitar a referência a detalhes de implementação ao longo do texto.

figura observa-se que os pares de classes a fundir são identificados (linhas 3 e 4) e é ativada a funcionalidade de fusão disponibilizada no framework OCEAN (linha 7).

Classe: ProxyAdapterTool

```
fulfillVoidMBD: aMBD in: aSpecification
1.  | aStatement containerStatement newContainerStatement subjectAttribute subjectCalledMethod |
        "Criação do comando composite (IF) que testa a disponibilidade de subject"
2.  containerStatement := CompositeStatement new.
3.  containerStatement header: 'IF'.
4.  containerStatement method: (aMBD method).
5.  containerStatement container: nil.
6.  containerStatement order: 1.
7.  containerStatement auxOrder: nil.
8.  containerStatement stringSentence: 'self acessibleSubject'.
9.  aMBD paste: containerStatement in: aSpecification with: nil.
10. (aMBD concepts: MBDStatement) do: [:aMBDStatement |
11.     (((     ((aMBDStatement header) = (containerStatement header))
12.     &       ((aMBDStatement order) = (containerStatement order))      )
13.     &       ((aMBDStatement auxOrder) = (containerStatement auxOrder))      )
14.     &       ((aMBDStatement container) = (containerStatement container))      )
15.                     ifTrue: [newContainerStatement := aMBDStatement]].

        "Criação de comando message embutido no comando composite recém-criado"
16. aStatement := MessageStatement new.
17. aStatement header: 'MESSAGE'.
18. aStatement method: (aMBD method).
19. aStatement container: newContainerStatement.
20. aStatement order: 1.
21. aStatement auxOrder: 1.
22. aStatement referedObject: nil.
23. (aSpecification concepts: OCEANAttribute  sustainedBy: (aMBD method class)) do: [:anAttribute |
24. ((anAttribute name) = 'subject') ifTrue: [subjectAttribute := anAttribute]].
25. aStatement secondReferedObject: subjectAttribute.
26. (self acessibleMethodListOf: (aMBD method class superclass)) do: [:aMethod |
27.     (aMethod equivalentToOtherClassMethod: (aMBD method)) ifTrue: [
28.         subjectCalledMethod := aMethod]].
29. aStatement calledMethod: subjectCalledMethod.
30. aStatement parameterAssignmentDictionary: Dictionary new.
31. (subjectCalledMethod parameterList) do: [:calledMethPar |
32.     ((aMBD method) parameterList) do: [:describedMethPar |
33.         ((calledMethPar name) = (describedMethPar name)) ifTrue: [
34.             (aStatement parameterAssignmentDictionary) at: calledMethPar put: describedMethPar] ] ].
35. aMBD paste: aStatement in: aSpecification with: nil
```

FIGURA 7.30 - Código de método da classe ProxyAdapterTool

A figura 7.30 apresenta o código do método *fulfillVoidMBD: aMBD in: aSpecification*, da classe *ProxyAdapterTool*. Esta classe corresponde à ferramenta de adaptação associada ao padrão de projeto *Proxy*, que atua apenas no procedimento de inserção deste padrão. O método apresentado na figura 7.30 é invocado durante a atuação da ferramenta de adaptação (mensagem 9 do diagrama da figura 7.27) e tem a funcionalidade de preencher diagramas de corpo de método do *proxy* (métodos sobrepostos que não produzem retorno) - vide figura 6.41, do capítulo anterior, que apresenta um diagrama de corpo de método produzido a partir da atuação deste método. A ação do método consiste em produzir dois comandos, *If* e *Message*, o segundo

embutido no primeiro, e inseri-los em um diagrama de corpo de método. No código apresentado na figura 7.30 observa-se que estes comandos são instanciados (o primeiro na linha 2 e o segundo na linha 16), têm seus atributos preenchidos, referenciando apenas conceitos contidos na especificação em que serão introduzidos e são colados no diagrama de corpo de método. Observa-se o procedimento de colagem (diretamente no modelo) nas linhas 9 e 35.

O procedimento de inserção de padrões de projeto suportado pelo framework OCEAN, acima descrito, corresponde a uma funcionalidade complexa, composta pelas ações de edição semântica supridas pelo framework. A atuação da ferramenta de inserção de padrões, que envolve a atuação de um conjunto de ferramentas, consiste em automatizar a seqüência de ações de edição para obtenção do resultado desejado - no caso, a inserção de um padrão de projeto. O conjunto de procedimentos descritos nas figuras 7.28, 7.29 e 7.30, que correspondem a uma fração do conjunto de procedimentos de edição que ocorrem na atuação de ferramenta de inserção de padrões, mostram a utilização das funcionalidades de cópia, colagem e fusão para a inserção de padrões. As outras ferramentas que apoiam a construção de especificações, descritas no capítulo anterior, também são baseadas na composição de ações de edição. O exemplo da ferramenta de inserção de padrões ilustra como o framework OCEAN suporta o desenvolvimento de mecanismos funcionais complexos, isto é, como estes mecanismos são inseridos na estrutura de um ambiente e como são construídos, reutilizando funcionalidades providas pelo framework.

7.3.2 Ações de Ferramentas de Edição, Análise e Transformação

Um ambiente de desenvolvimento de software demanda funcionalidades além daquelas voltadas à construção de especificações - como pode ser classificada a ferramenta de inserção de padrões, acima descrita. A seguir são apresentados exemplos de procedimentos de manipulação de especificações que ocorrem em outros tipos de ferramenta, que fazem parte da estrutura do framework OCEAN.

Classe: AggregationRepairer

changeConcept

```
1.  | auxRelationship |
2.  (SingleConnector specification autorizeChange) ifTrue: [
3.    ((self solutionAspect value) = #substitute) ifTrue: [
4.      auxRelationship := BinaryRelationship new: (self handledConcept wholeClass)
                                                  and: (self handledConcept partClass).
5.      auxRelationship name: 'it_was_aggregation'.
6.      auxRelationship multiplicity2: (self handledConcept multiplicity).
7.      (self handledConcept conceptualModelReferers at: 1) paste: auxRelationship
                                          in: (SingleConnector specification) with: nil.].
8.    SingleConnector specification remove: (self handledConcept)]
```

FIGURA 7.31 - Código de método da classe AggregationRepairer

A figura 7.31 descreve o método *changeConcept*, da classe *AggregationRepairer*, que executa uma das seguintes ações: remoção de uma agregação (referida no código como *handledConcept*) ou conversão da agregação em associação binária. A segunda opção corresponde a uma funcionalidade de transformação de elemento de especificação e consiste em criar uma associação binária com os mesmos

extremos da agregação original (linha 4), inseri-la na especificação tratada, através de colagem (linha 7), e remover a agregação (linha 8). Funcionalidades de transformação são suportadas pelas funcionalidades de edição semântica do framework OCEAN e podem compor ferramentas de transformação mais complexas, que possibilitem a conversão de elementos de especificação em elementos de outros tipos.

A ferramenta de geração de código Smalltalk a partir de especificações OO ilustra uma outra modalidade de ferramenta, ferramentas que convertem as informações manipuladas por um ambiente em um formato a ser utilizado por algum elemento externo ao ambiente. As figuras 7.32 e 7.33 apresentam métodos da classe *SmalltalkCodeGenerator*, que implementa a ferramenta de geração de código. Verifica-se que a ação destes métodos consiste em obter informações da especificação tratada, compor *strings* com as informações obtidas e inserir estes *strings* em um arquivo *ASC II*. Ferramentas para a produção de código em outras linguagens podem ser desenvolvidas usando o mesmo princípio, bastando proceder a composição de *strings* obedecendo a sintaxe da linguagem de programação selecionada.

Classe: SmalltalkCodeGenerator

```
generateHeaderTo: aMethod of: aSpecification
1.  |headerString|
2.  headerString := aMethod name.
3.  ((aMethod parameterList size) > 0)
4.    ifTrue: [headerString := headerString,': ', ((aMethod parameterList at: 1) name).
5.            ((aMethod parameterList size) > 1)
6.            ifTrue: [2 to: (aMethod parameterList size) do: [:index |
7.                    headerString := headerString,' with: ', ((aMethod parameterList at: index) name)] ] ].
8.  self outputFileManager blankLines: 2.
9.  self outputFileManager appendString: headerString.
```

FIGURA 7.32 - Código de método da classe SmalltalkCodeGenerator

Classe: SmalltalkCodeGenerator

```
generateAssignmentCodeFor: aStatement
1.  | codeString |
2.  codeString := aStatement referedObject name.
3.  codeString := codeString, ' := '.
4.  (aStatement secondReferedObject isKindOf: String)
5.      ifTrue: [codeString := (codeString, (aStatement secondReferedObject), '.')]
6.      ifFalse: [codeString := (codeString, (aStatement secondReferedObject name), '.')].
7.  self outputFileManager newLine.
8.  self outputFileManager appendString: codeString
```

FIGURA 7.33 - Código de método da classe SmalltalkCodeGenerator

Ferramentas de Engenharia Reversa que inspecionem código e produzam uma especificação (ou trechos de especificação) fazem o caminho inverso das ferramentas de geração de código. Atualmente não existe na estrutura do framework OCEAN ferramentas com esta finalidade. Como OCEAN suporta a criação de especificações e a composição de especificações a partir da criação e inclusão de elementos de especificação, uma ferramenta de Engenharia Reversa para um ambiente sob o framework OCEAN demanda basicamente a construção do mecanismo de análise de

código. A construção de especificações a partir das informações obtidas na análise pode reutilizar as funcionalidades de edição semântica supridas pelo framework.

A figura 7.34 apresenta um trecho de código de um método da classe *OOSpecificationAnalyser*, que corresponde à ferramenta de verificação de consistência de especificações OO. A responsabilidade do método apresentado é verificar se na especificação analisada existe um diagrama de corpo de método associado a cada método concreto, não-externo, da especificação. Este tipo de procedimento consiste basicamente em obter informações da especificação, proceder uma análise e produzir um relatório com o resultado da análise. No caso apresentado:

- são obtidos os métodos da especificação (linha 2);
- são obtidos os métodos associados aos diagramas de corpo de método da especificação (linhas 5 a 7);
- são levantados os métodos da especificação que são concretos, que não são classificados como externos e que não estejam contidos na relação de métodos associados a diagramas de corpo de método (linhas 8 a 10);
- se houver algum método nesta condição, é registrada a situação de erro (linhas 11 e 12).

Classe: OOSpecificationAnalyser

```
MethodBodyDiagramExistenceTest: aSpecification
1.  | specificationMethodList MBDMethodList undescribedMethodList |
2.  specificationMethodList := aSpecification concepts: OCEANMethod.
3.  MBDMethodList := List new.
4.  undescribedMethodList := List new.
5.  (aSpecification models: MethodBodyDiagram) do: [:aMBD |
6.      (MBDMethodList includes: (aMBD method))
7.          ifFalse: [MBDMethodList add: (aMBD method)] ].
8.  specificationMethodList do: [:aMethod |
9.      (  ((MBDMethodList includes: aMethod) not)
        &  ( ((aMethod classification) ~= #abstract) & ((aMethod external) not) )  )
10.         ifTrue: [undescribedMethodList add: aMethod] ].
11. ((undescribedMethodList size) > 0)
12. ifTrue: [self analysisError: true.
            •••
```

FIGURA 7.34 - Código de método da classe OOSpecificationAnalyser

Além das ferramentas de análise que compõem a estrutura do framework OCEAN, utilizadas pelo ambiente SEA (e descritas no capítulo anterior), outras ferramentas deste tipo podem ser produzidas a partir do suporte provido pelo framework OCEAN, como ferramentas para a produção de métricas, voltadas à avaliação de qualidade das especificações produzidas por um ambiente.

Os exemplos acima apresentados ilustram diferentes tipos de ferramentas que podem ser produzidas a partir do framework OCEAN. Esta flexibilidade para a produção de ferramentas, o baixo acoplamento mantido entre as classes que implementam ferramentas, ambientes, especificações e elementos de especificação, juntamente com a flexibilidade para a definição de diferentes tipos de especificação, constituem as principais características do framework OCEAN.

8 Conclusão

Nos capítulos anteriores apresentou-se a abordagem adotada no presente trabalho para suportar o desenvolvimento e o uso de frameworks e componentes. O framework OCEAN constitui um suporte flexível e extensível para a construção de ambientes de desenvolvimento de software. É caracterizado pela capacidade de suportar a construção de ambientes que manipulem diferentes tipos de especificação e que disponham de diferentes conjuntos de funcionalidades.

O ambiente SEA foi desenvolvido utilizando a totalidade dos recursos atualmente oferecidos pelo framework OCEAN. Suporta o desenvolvimento e o uso de frameworks e componentes, contando com funcionalidades que permitem automatizar tarefas destas atividades, destacando-se a capacidade de inserção automática de padrões de projeto, a capacidade de construir de forma automática partes de artefatos desenvolvidos sob frameworks e a capacidade de construir de forma automática a interface de um componente, a partir de uma especificação de interface.

A seguir, são apresentadas as principais contribuições obtidas com o trabalho desenvolvido, suas limitações e discutem-se futuras extensões aos protótipos desenvolvidos, bem como futuros caminhos de pesquisa que podem ser seguidos utilizando os resultados do presente trabalho.

8.1 Resumo das Contribuições

O trabalho de pesquisa desenvolvido produziu um conjunto de resultados que engloba os objetivos estabelecidos na Proposta de Tese[77], aprovada em 1997, e inclui um conjunto de soluções para problemas em torno do desenvolvimento orientado a componentes, tema não previsto na referida proposta. As principais contribuições introduzidas pela presente tese, nos diferentes aspectos tratados, são:

- Foi proposta uma arquitetura para frameworks de ambientes de desenvolvimento de software. Esta arquitetura se caracteriza pela flexibilidade para a definição de estruturas de especificação e de mecanismos funcionais. Através da arquitetura do framework OCEAN é introduzida uma abordagem de construção de estruturas de especificação baseada no armazenamento centralizado dos elementos que constituem uma especificação e no registro das relações semânticas entre esses elementos, aplicável a diferentes abordagens de modelagem;
- É introduzida uma notação de projeto voltada às características dos frameworks, através da extensão da notação UML, que possibilita produzir especificações de projeto com o registro das partes flexíveis de um framework - decisão de projeto que não é possível expressar através de uma linguagem de programação. Esta notação foi utilizada no ambiente SEA. O tema modelagem de frameworks foi abordado em artigos apresentados no X Simpósio Brasileiro de Engenharia de Software [SIL 96] e na 26th International Conference of the Argentine Computer Science and Operational Research Society [SIL 97];

---

77 A defesa de proposta de tese é uma etapa intermediária do programa de Doutorado em Ciência da Computação da Universadade Federal do Rio Grande do Sul. A proposta que antecedeu a elaboração da presente tese foi aprovada em setembro de 1997 por uma banca composta pelos doutores Ana Maria de Alencar Price, Carlos Alberto Heuser e Paulo Cesar Masiero, além do orientador do trabalho, o Professor Roberto Tom Price.

- O uso de padrões de projeto é recomendado pela possibilidade de reuso de experiência de projeto que leva à obtenção de estruturas de classes bem organizadas e mais aptas à manutenção. A prática do uso de padrões, porém, em geral, depende basicamente do conhecimento dos padrões pelos desenvolvedores de software. O framework OCEAN provê um mecanismo voltado suportar a aplicação de padrões de projeto, utilizado pelo ambiente SEA. No aspecto de utilização de padrões de projeto, a tese introduz uma abordagem de inserção de padrões em especificações de artefatos de software que possibilita a semi-automatização deste procedimento. O tema uso de padrões no desenvolvimento de frameworks foi abordado no artigo apresentado no Workshop Iberoamericano de Engenharia de Requisitos e Ambientes de Software [SIL 98];
- Nas abordagens atuais de desenvolvimento de *cookbooks* para auxílio ao uso de frameworks, *cookbooks* são produzidos de forma artesanal, baseando-se somente na experiência dos desenvolvedores, no uso do framework descrito. O ambiente SEA suporta o desenvolvimento de *cookbooks* ativos e o uso de *cookbooks* ativos para a construção de artefatos de software a partir de frameworks. A ação de *links* ativos no uso de um *cookbook* resulta na criação automática de partes de um artefato de software produzido sob o framework tratado pelo *cookbook*. No aspecto de construção de *cookbooks* ativos, a tese introduz uma abordagem de avaliação da qualidade de um *cookbook* ativo produzido para orientar o uso de um framework. A estrutura de especificação proposta no presente trabalho possibilita a comparação de um *cookbook* com a especificação de projeto do framework por ele descrita, característica não encontrada em outras abordagens. O tema desenvolvimento e uso de *cookbooks* ativos para apoio ao uso de frameworks foi abordado no artigo apresentado na XVIII International Conference of the Chilean Computer Science Society [SIL 98b];
- É introduzida a possibilidade de avaliar se um artefato de software produzido sob um framework apresenta o conjunto mínimo de classes e métodos imposto pelo projeto do framework usado. Isto é viabilizado pelo registro da informação referente à flexibilidade de um framework em sua especificação de projeto, e pela comparação de estruturas de especificação. Com isto, introduz-se um critério de avaliação de qualidade de especificações produzidas a partir do uso de *cookbooks* ativos, no nível de projeto, característica não observada em abordagens de uso de frameworks;
- A adoção do modelo rede de Petri para especificar comportamentalmente interfaces de componentes, bem como arquiteturas de componentes - obtidas pela conexão de componentes através de suas interfaces - introduz uma abordagem de avaliação da dinâmica comportamental de componentes e de arquiteturas de componentes;
- A tese introduz uma abordagem de desenvolvimento de componentes flexíveis baseada na construção destes componentes como frameworks orientados a objetos. Esta abordagem trata a questão de incompatibilidade de componentes e, em função da diminuição do esforço para a obtenção de novos componentes através do reuso de um framework, viabiliza a opção de criação de novos componentes, ao invés da adaptação de componentes existentes;
- A tese propõe um *padrão de projeto* (utilizando a nomenclatura de Gamma [GAM94]) para a construção de interfaces de componentes. O padrão proposto trata o problema da definição de um conjunto de canais de comunicação bidirecional para um componente, o problema da geração da estrutura interna na instanciação de um componente e estabelece uma solução de projeto reusável em diferentes abordagens de desenvolvimento de componentes. As soluções propostas no presente trabalho

para problemas relacionados ao desenvolvimento orientado a componentes foram descritas no artigo apresentado no XIII Simpósio Brasileiro de Engenharia de Software [SIL 99];

- A concepção de ambientes de desenvolvimento de software é uma área de pesquisa que demanda recursos, principalmente de suporte de desenvolvimento com flexibilidade para possibilitar a experimentação de diferentes abordagens tanto de modelagem, quanto de funcionalidades relacionadas ao desenvolvimento de software. Neste sentido, o framework OCEAN constitui uma plataforma de experimentos de ambientes e ferramentas de suporte à produção de software e representa uma contribuição para o desenvolvimento de projetos nessa área.

8.2 Limitações

O ambiente SEA apresenta um conjunto de limitações em relação aos requisitos para um ambiente de apoio ao desenvolvimento e uso de frameworks e componentes, discutidos no capítulo 6. As limitações são herdadas de OCEAN, pela forma como certos recursos estão definidos neste framework ou pela ausência de recursos necessários ao ambiente. A seguir são discutidas as limitações do ambiente SEA e os possíveis meios de superá-las.

8.2.1 Falta de Eficiência

O ambiente SEA apresenta problemas de falta de eficiência que são acentuados na medida em que aumenta a quantidade de elementos que compõem uma especificação que esteja sendo manipulada. Este problema de falta de eficiência tem o conjunto de causas e possíveis soluções abaixo descritos.

- No atual protótipo do framework OCEAN, objetos visuais associados a uma especificação são tratados como objetos persistentes - especificação agrega modelos e cada modelo agrega um diagrama gráfico HotDraw, que agrega figuras que representam conceitos. Com isto, alterações procedidas em uma especificação que demandem a redefinição de muitos elementos visuais, inclusive de elementos visuais que não estejam sendo exibidos, podem implicar em um custo de processamento elevado. Uma solução para este problema seria a criação de objetos visuais por demanda (na medida em que precisem ser exibidos), ao invés de tratá-los como objetos persistentes. Esta solução tem o custo de exigir a criação do diagrama visual associado a um modelo (e as figuras que o compõem) cada vez que o modelo é exibido, porém pode diminuir significativamente o custo de processamento envolvido na propagação de alterações em especificações extensas.
- O mecanismo de propagação de alterações definido no framework OCEAN, baseado no padrão de projeto *Observer*, divulga a alteração de um elemento de especificação aos elementos de especificação semanticamente associados, sem classificar o tipo de alteração ocorrida. Com isto, um elemento de especificação notificado executa um procedimento de atualização, que envolve a atualização de sua representação visual e a propagação da ocorrência de alteração a outros elementos de especificação semanticamente associados a ele. Como não é identificado o tipo de alteração ocorrida, um elemento de especificação notificado de uma alteração não tem meios de avaliar se há ou não necessidade de executar um procedimento de atualização. Isto pode acarretar a execução de procedimentos de atualização desnecessários,

inclusive a redefinição desnecessária de objetos visuais (neste caso, com um custo de processamento elevado). Este problema se agrava na medida em que aumenta a quantidade de elementos de uma especificação. Uma solução possível para este problema seria identificar o tipo de alteração procedida, possibilitando que um elemento de especificação notificado de uma alteração avalie a necessidade de executar um procedimento de atualização e de propagar a ocorrência da alteração aos elementos de especificação a ele associados.

- O uso do framework HotDraw acarreta a produção de objetos visuais *grandes* (em relação ao espaço de memória necessário para armazená-los). Na medida em que uma especificação apresente grande quantidade de elementos, o arquivo que armazena esta especificação torna-se extenso, acarretando um custo em termos de eficiência para sua manipulação, bem como para seu armazenamento. Uma solução para este problema seria usar um mecanismo alternativo ao framework HotDraw para suportar a construção de editores gráficos. Neste caso, outra abordagem de edição que possibilite a produção de objetos visuais *menores* que os produzidos pelo framework HotDraw poderia reduzir o problema da manipulação de objetos visuais *grandes*.
- O uso da linguagem Smalltalk, que é uma linguagem interpretada, para a implementação de OCEAN e SEA faz com que o ambiente seja menos eficiente do que se fosse implementado em outra linguagem de programação, como C++. A implementação do ambiente em C++ diminuiria o problema de falta de eficiência, porém impediria o reuso dos frameworks MVC e HotDraw, implementados em Smalltalk.

8.2.2 Ausência de Recursos

O ambiente SEA apresenta as seguintes limitações associadas à ausência de recursos no framework OCEAN, incluídas na relação de alterações previstas para a evolução do ambiente:

- inexistência de um mecanismo de análise de redes de Petri, necessário para apoiar a avaliação de consistência de especificações de interfaces de componentes e de arquiteturas de componentes;
- incapacidade de utilização do conjunto dos recursos denotacionais definidos em UML.

8.2.3 Falta de Avaliação do Ganho de Produtividade com o Uso do Ambiente SEA

Ainda não foi feita uma avaliação experimental do ganho de produtividade obtido com o uso do ambiente SEA. Pretende-se promover experimentações envolvendo grupos de usuários que seriam submetidos ao desenvolvimento de sistemas com níveis de complexidade equivalentes, utilizando o ambiente SEA, algum outro ambiente comercial e apenas suporte à programação. A avaliação do tempo de desenvolvimento e dos produtos desenvolvidos nas três situações deve produzir uma avaliação qualitativa e quantitativa dos benefícios que podem ser obtidos a partir do uso do ambiente SEA no desenvolvimento e uso de frameworks e componentes.

8.3 Trabalhos Futuros

O framework OCEAN constitui um suporte para a construção de ambientes de desenvolvimento de software que deve ser estendido em futuros esforços de pesquisa. A evolução do ambiente SEA e o desenvolvimento de novos ambientes e novas ferramentas são os aspectos a serem tratados em futuras extensões do framework OCEAN. A seguir são descritos trabalhos e linhas de pesquisa que podem ser suportados pelo trabalho ora desenvolvido.

8.3.1 Extensão do Conjunto de Recursos do Framework OCEAN, para Utilização no Ambiente SEA

O protótipo de ambiente atualmente desenvolvido é um resultado parcial, em relação ao conjunto de requisitos estabelecidos para um ambiente de apoio ao desenvolvimento e uso de frameworks e componentes[78]. Assim, a primeira prioridade em termos de trabalhos futuros é a evolução do ambiente SEA, visando a satisfação do conjunto de requisitos estabelecido. Isto significa a evolução dos recursos de modelagem e funcionais do framework OCEAN, a serem utilizados pelo ambiente, abaixo relacionados.

- **Avaliação da necessidade de adoção de técnicas de modelagem adicionais, e complementação sintática das técnicas de modelagem atualmente utilizadas**: um dos caminhos de evolução do ambiente SEA envolve a avaliação do conjunto de técnicas de modelagem utilizado para produzir especificações. Os aspectos de modelagem abaixo relacionados devem ser tratados.
  - ⇨ Inclusão de toda a expressividade das técnicas de modelagem de UML utilizadas, eliminando as simplificações de notação adotadas na atual implementação;
  - ⇨ Avaliação da adoção de técnicas de modelagem adicionais para a descrição de aspectos que não são satisfatoriamente descritos com os recursos de modelagem ora disponíveis, como a modelagem de interfaces e a modelagem da interação de um artefato de software com o meio externo.
- **Inclusão de mecanismo de análise de redes de Petri**: para eliminar uma limitação atual, anteriormente discutida;
- **Inclusão da capacidade de modelar arquiteturas de componentes, com a possibilidade de reusar padrões arquitetônicos e associar semântica aos elementos de conexão de componentes**: isto corresponde a uma nova estrutura de especificação que reuse componentes previamente desenvolvidos e em que é estabelecida a interconexão dos componentes. Um requisito importante a ser considerado é a inclusão da capacidade de reconfiguração dinâmica de arquiteturas de componentes.
- **Inclusão de uma ferramenta de edição voltada a proceder conversão automática de elementos de uma especificação**: mecanismos de edição de especificações que automatizem procedimentos complexos são importantes porque diminuem a necessidade de esforço, bem como a possibilidade de inserção de erros

---

78 Conforme discutido no capítulo 6, a concepção de modelagem de artefatos de software adotada no presente trabalho prevê que um artefato de software possa ser modelado como uma especificação OO ou como uma especificação de arquitetura de componentes (sendo que esta última possibilidade de modelagem ainda não é tratada pelo ambiente SEA). Uma especificação OO, que pode modelar um framework, um componente ou uma aplicação, pode reusar em sua estrutura frameworks, componentes e padrões de projeto.

durante a construção de especificações. Um exemplo de mecanismo funcional deste tipo, disponível no ambiente SEA, é a ferramenta de geração automática de métodos de acesso a atributos. Uma outra ferramenta de edição a acrescentar consiste em uma ferramenta que produza transformação de elementos de especificação em elementos de tipos diferentes. Alguns exemplos de situações atendidas por este mecanismo seriam:

- ⇨ a conversão de uma associação binária em agregação (ou vice-versa);
- ⇨ a conversão de uma associação binária em duas, inserindo uma nova classe entre as classes relacionadas pela associação original;
- ⇨ a inserção de um estado intermediário em uma transição de estados;
- ⇨ conversão de tipo de um comando de embutimento[79] (de diagrama de corpo de método).

- **Inclusão de novos geradores de código**: para possibilitar a tradução de especificações de projeto para outras linguagens de programação, além de Smalltalk.
- **Inclusão de ferramentas de Engenharia Reversa**: para possibilitar a importação de artefatos de software descritos através de código de linguagem de programação.
- **Inclusão de ferramentas de transformação**: deve ser avaliada a possibilidade de produzir ferramentas de conversão que produzam transformações envolvendo dados manipulados por outros ambientes, como a conversão de uma especificação do ambiente SEA em um arquivo de dados no formato de outro ambiente de desenvolvimento de software, e a transformação inversa. A importância deste tipo de mecanismo consiste na possibilidade de ultrapassar as fronteiras de um ambiente, permitindo reutilizar artefatos definidos em outros ambientes, ou produzir especificações para serem manuseadas em outros ambientes.
- **Inclusão de ferramentas para levantamento de métricas, para a avaliação da qualidade de software no nível de projeto**: atualmente o framework OCEAN tem desenvolvidas ferramentas de análise para a avaliação de consistência semântica de especificações e modelos. Um aspecto ainda não tratado, porém, é a exploração da capacidade do framework OCEAN de dar origem a ferramentas de análise, para produzir ferramentas que avaliem a qualidade de um software. Ferramentas podem produzir avaliação de qualidade baseadas em critérios propostos para especificações OO, como avaliar a largura e a profundidade de hierarquias de herança, o número de métodos e atributos em cada classe, a quantidade de comandos nos algoritmos dos métodos e outros. A adoção deste tipo de procedimento possibilitaria localizar os pontos críticos de um artefato de software, em termos de um conjunto de critérios de qualidade adotado, durante a etapa de projeto.

8.3.2 Criação de Novos Ambientes e Ferramentas

O framework OCEAN pode ser utilizado para apoiar a construção de outros tipos de ambiente, com requisitos diferentes daqueles considerados no desenvolvimento do ambiente SEA. Pode-se mencionar os seguintes exemplos que abordam características não tratadas no desenvolvimento do ambiente SEA, e que podem nortear futuros esforços de pesquisa:

- ambientes multi-usuário;
- ambientes para operação distribuída;

---

79 Comandos de diagrama de corpo de método que embutem outros comandos: If, IfElse, While, Repeat, nTimes, ExternalSequenceDiagram.

- ambientes que suportem o gerenciamento do processo de desenvolvimento de software e o controle de versões;
- ambientes para desenvolvimento de software com características específicas, como software concorrente ou software para execução em tempo real;
- ambientes para suportar etapas específicas do ciclo de vida de um software, como um ambiente para suportar Análise de Requisitos.

As experimentações de funcionalidades em torno do desenvolvimento de software podem se ater ao desenvolvimento de mecanismos funcionais com finalidades específicas, a serem incluídos em um ambiente desenvolvido sob o framework OCEAN. Neste aspecto, OCEAN pode suportar o desenvolvimento de ferramentas voltadas à edição, análise ou transformação de especificações, devendo ser usado com esta finalidade em futuros trabalhos de pesquisa.

8.4 Considerações Finais

Ao longo das últimas décadas a evolução dos requisitos dos artefatos de software vem causando o aumento de sua complexidade. Por outro lado, o mercado de desenvolvimento de software cada vez mais exige a redução do tempo empregado no desenvolvimento. O tratamento do paradoxo de produzir artefatos de software mais complexos em menos tempo, passa necessariamente pela reutilização. Apenas reutilizando recursos previamente produzidos é possível produzir software mais complexo em menos tempo. A gama dos recursos que podem ser reusados abrange desde recursos abstratos, como a experiência de desenvolvedores, até recursos palpáveis, como estruturas de código pré-elaboradas.

Neste panorama, as abordagens desenvolvimento orientado a componentes, frameworks orientados a objetos, Arquitetura de Software e padrões têm em comum a meta de sistematizar o reuso de experiência de projeto e assim, reduzir a necessidade de esforço no desenvolvimento de software. Frameworks e componentes, além disso, promovem reuso de código. Atualmente a carência de suporte automatizado dificulta a adoção dessas abordagens para o desenvolvimento de software, fazendo com que sua aplicação dependa da experiência de desenvolvedores de software em utilizá-las.

O presente trabalho foi motivado pelo reconhecimento das vantagens e das dificuldades em torno da aplicação desse conjunto de abordagens de desenvolvimento de software preocupadas com a promoção de reuso. Neste sentido, produziu um estudo a respeito das dificuldades de aplicar o conjunto de abordagens citado e das possíveis formas de contornar estas dificuldades, preocupando-se inclusive com a utilização simultânea do conjunto de abordagens. O framework OCEAN e o ambiente SEA materializam um conjunto de abordagens propostas para facilitar o desenvolvimento e o uso de frameworks e componentes e constituem uma contribuição para a reflexão dos problemas em torno da reutilização de software.

O aprofundamento da pesquisa introduzida por esta tese poderá ajudar a tornar a aplicação de abordagens voltadas à promoção de reuso uma prática amplamente difundida, o que pode contribuir para o aumento da produtividade e da qualidade no desenvolvimento de software.

Anexo 1 Estrutura Semântica de uma Especificação Baseada no Paradigma de Orientação a Objetos

A estrutura semântica de especificações baseadas no paradigma de orientação a objetos produzidas no ambiente SEA é descrita formalmente a seguir, a partir de uma gramática de atributos. As regras semânticas associadas às produções da gramática foram definidas a partir de lógica de primeira ordem.

As regras semânticas apresentam apenas atributos sintetizados. *Class•var* denota uma ocorrência (instância) do tipo de conceito *Class*. *Class•pred* denota o valor do predicado associado a uma ocorrência do tipo de conceito *Class* (*true ou false*).

Para uma especificação ser considerada consistente é necessário que:

- sua estrutura, definida nas produções da gramática, seja reconhecida;
- seu predicado assuma o valor *true*.

A avaliação da estrutura da especificação, assim como do valor do predicado, se estende ao longo da árvore de derivação associada a uma ocorrência de especificação.

A escolha deste mecanismo de descrição deveu-se à possibilidade de produzir uma especificação formal que, apesar de longa, é relativamente simples de ser consultada, pois a gramática está organizada em forma de tabela, com os elementos que compõem uma especificação definidos como um símbolos não-terminais no lado esquerdo das produções. Mesmo as regras semânticas associadas às produções, por vezes longas, são constituídas por um conjunto de predicados ligados por operadores *AND*, ou seja, cada regra semântica é, de fato, uma composição de regras. É apresentado um exemplo, a seguir, para ilustrar a expressividade da gramática de atributos utilizada.

Seja a produção 251 que define um atributo sem origem externa definida, abaixo reproduzida.

> ***Attribute*** $\rightarrow$ *'startAttribute' . DocumentName . ElementLinks . External . Type . ObjectIsSet . Class . MetaAttribute . 'endAttribute'.*

A produção acima estabelece que a estrutura de um conceito *attribute* (uma instância) apresenta:

- um nome,
- um conjunto de *links*,
- a definição de ser externo ou não (*true ou false*),
- um tipo,
- a definição se o atributo consiste em um conjunto (*true ou false*) e
- a referência à classe a que o atributo está ligado.

A regra semântica associada à produção é reproduzida a seguir.

$$Attribute•pred :=$$
$$Pred11 \wedge DocumentName•pred \wedge ElementLinks•pred \wedge External•pred \wedge Type•pred \wedge ObjectIsSet•pred \wedge Class•pred \wedge MetaAttribute•pred$$

$$Pred11 :=$$
$$(\forall Attribute1•var \in ConceptRepository•var . \forall SpecificationElementPair•var \in SustainmentTable•var . (((Attribute•var \neq Attribute1•var) \wedge (Attribute•DocumentName•var = Attribute1•$$

$$DocumentName \bullet var) \wedge (SpecificationElementPair \bullet SpecificationElement2 \bullet var = Attribute1 \bullet var))$$
$$\Rightarrow$$
$$((SpecificationElementPair \bullet SpecificationElement1 \bullet var \neq Attribute \bullet Class \bullet var) \wedge$$
$$(SpecificationElementPair \bullet SpecificationElement1 \bullet var \notin superclassesOf(Attribute \bullet Class \bullet var))\ )$$
$$\wedge$$
$$(\exists\ SpecificationElementPair \bullet var \in SustainmentTable \bullet var\ .\ ((SpecificationElementPair \bullet$$
$$SpecificationElement1 \bullet var = Attribute \bullet Class \bullet var) \wedge (SpecificationElementPair \bullet$$
$$SpecificationElement2 \bullet var = Attribute \bullet var))\ )$$
$$\wedge$$
$$(\exists\ SpecificationElementPair \bullet var \in ReferenceTable \bullet var\ .\ ((SpecificationElementPair \bullet$$
$$SpecificationElement1 \bullet var = Attribute \bullet Type \bullet var) \wedge (SpecificationElementPair \bullet$$
$$SpecificationElement2 \bullet var = Attribute \bullet var))\ )$$
$$\wedge$$
$$(Attribute \bullet External \bullet var = Attribute \bullet Class \bullet External \bullet var)$$
$$\wedge$$
$$(\forall\ Link \bullet var \in Attribute \bullet ElementLinks \bullet var\ .\ \exists\ SpecificationElementPair \bullet var \in SustainmentTable \bullet$$
$$var\ .\ ((SpecificationElementPair \bullet SpecificationElement1 \bullet var = Attribute \bullet var) \wedge$$
$$(SpecificationElementPair \bullet SpecificationElement2 \bullet var = Link \bullet var))\ )$$

Esta regra semântica estabelece que para um atribulo ser considerado semanticamente consistente (isto é, o atributo associado ao predicado assumir valor *true*), além da necessidade dos elementos que compõem sua estrutura serem consistentes, é necessário que:

- não exista outro atributo com o mesmo nome na classe que contém este atributo (referenciada pelo atributo) e nem nas suas superclasses;
- exista um par na tabela de sustentação estabelecendo a classe referenciada pelo atributo como seu elemento sustentador;
- exista um par na tabela de referência estabelecendo o tipo referenciado pelo atributo como seu elemento referenciado;
- a condição *external* do atributo seja igual à da classe por ele referenciada;
- exista um par na tabela de sustentação para cada *link* referenciado pelo atributo estabelecendo cada *link* como elemento sustentado pelo atributo.

Na gramática apresentada a seguir, $X \bullet var$ denota uma ocorrência de X; $X \bullet Y \bullet var$ denota uma ocorrência de Y, presente na estrutura de uma ocorrência de X. Na análise da estrutura de uma ocorrência de X, $Y \bullet var$ denota uma ocorrência de Y, presente na estrutura de uma ocorrência de X. Caso a estrutura de X não apresente ocorrência de Y, denota a ocorrência de Y imediatamente superior à ocorrência de X considerada, na árvore de derivação da especificação.

	Produção		
nº	**lado esquerdo →**	**lado direito**	**regra semântica**
	Specification		
1	**Specification**	OOSpecification	Specification•pred := OOSpecification•pred
2		HyperdocSpecification	Specification•pred := HyperdocSpecification•pred
3		ComponentInterfaceSpecification	Specification•pred := ComponentInterfaceSpecification•pred
4		ComponentArchitectureSpecification	Specification•pred := ComponentArchitectureSpecification•pred
5	**OOSpecification**	'startSpecification' . DocumentName . SpecificationType . ConceptRepository . ConceptualModelRepository . SustainmentTable . ReferenceTable. 'endSpecification'	

OOSpecification•pred :=
Pred01 ∧ DocumentName•pred ∧ SpecificationType•pred ∧ ConceptRepository•pred ∧ ConceptualModelRepository•pred∧ SustainmentTable•pred ∧ ReferenceTable•pred

Pred01 =
((SpecificationType = 'application') ⇒
((∀ Class•var ∈ ConceptRepository•var . ((Class•Redefinable•var = TRUE) ⇒ ((Class•External•var = TRUE) ∧ (∃ Class1•var ∈ ConceptRepository•var . ((Class1•var ≠ Class•var) ∧ (Class1•Redefinable•var = FALSE) ∧ (Class•var ∈ superclassesOf(Class1•var)[1]))))))
∧ (∀ Class•var ∈ ConceptRepository•var . ((Class•Essential•var = TRUE) ⇒ ((Class•External•var = TRUE) ∧ (Class•Redefinable•var = TRUE) ∧ (∃ Class1•var ∈ ConceptRepository•var . ((Class1•var ≠ Class•var) ∧ (Class1•Redefinable•var = FALSE) ∧ (Class•var ∈ superclassesOf(Class1•var)))))))
∧ (∀ Class•var ∈ ConceptRepository•var . ((Class•Abstract•var = TRUE) ⇒ (∃ Class1•var ∈ ConceptRepository•var . ((Class1•var ≠ Class•var) ∧ (Class1•Abstract•var = FALSE) ∧ (Class•var ∈ superclassesOf(Class1•var)))))
∧ (∀ Class•var ∈ ConceptRepository•var . (Class•DocumentName•var ≠ 'SEAComponent'))))
∧
((SpecificationType = 'framework') ⇒
((∀ Class•var ∈ ConceptRepository•var . ((Class•Essential•var = TRUE) ⇒ (Class•Redefinable•var = TRUE)))
∧ (∀ Class•var ∈ ConceptRepository•var . (Class•DocumentName•var ≠ 'SEAComponent'))))
∧
((SpecificationType = 'component') ⇒
((∀ Class•var ∈ ConceptRepository•var . ((Class•Redefinable•var = TRUE) ⇒ ((Class•External•var = TRUE) ∧ (∃ Class1•var ∈ ConceptRepository•var . ((Class1•var ≠ Class•var) ∧ (Class1•Redefinable•var = FALSE) ∧ (Class•var ∈ superclassesOf(Class1•var)))))))
∧ (∀ Class•var ∈ ConceptRepository•var . ((Class•Essential•var = TRUE) ⇒ ((Class•External•var = TRUE) ∧ (Class•Redefinable•var = TRUE) ∧ (∃ Class1•var ∈ ConceptRepository•var . ((Class1•var ≠ Class•var) ∧ (Class1•Redefinable•var = FALSE) ∧ (Class•var ∈ superclassesOf(Class1•var)))))))
∧ (∀ Class•var ∈ ConceptRepository•var . ((Class•Abstract•var = TRUE) ⇒ (∃ Class1•var ∈ ConceptRepository•var . ((Class1•var ≠ Class•var) ∧ (Class1•Abstract•var = FALSE) ∧ (Class•var ∈ superclassesOf(Class1•var)))))
∧ (∃ Class•var ∈ ConceptRepository•var . ((Class•DocumentName•var = 'SEAComponent') ∧ (Class•Abstract•var = TRUE) ∧ (Class•External•var = TRUE) ∧ (∃ Class1•var ∈ ConceptRepository•var . ((Class1•var ≠ Class•var) ∧ (Class•External•var = FALSE) ∧ (Class1•Abstract•var = FALSE) ∧ (Class•var ∈ superclassesOf(Class1•var))))))
∧ (∃ Class•var ∈ ConceptRepository•var . ((Class•DocumentName•var = 'SEAOutputMailbox') ∧ (Class•Abstract•var = TRUE) ∧ (Class•External•var = TRUE)))

---

[1] "superclassesOf(Class•var)" é uma função que retorna o conjunto de todas as superclasses de uma ocorrência de Class (Class•var, no caso), contidas no repositório de conceitos (ConceptSetRepository•var). Caso a ocorrência de Class não possua superclasse, a função retorna um conjunto vazio.

∧ (∃ Class•var ∈ ConceptRepository•var . ((Class•DocumentName•var = 'SEAInterfaceChannel') ∧ (Class•Abstract•var = TRUE) ∧ (Class•External•var = TRUE)))))
∧
((SpecificationType = 'flexibleComponent') ⇒
((∀ Class•var ∈ ConceptRepository•var . ((Class•Essential•var = TRUE) ⇒ (Class•Redefinable•var = TRUE)))
∧ (∃ Class•var ∈ ConceptRepository•var . ((Class•DocumentName•var = 'SEAComponent') ∧ (Class•Abstract•var = TRUE) ∧ (Class•External•var = TRUE) ∧ (∃ Class1•var ∈ ConceptRepository•var . ((Class1•var ≠ Class•var) ∧ (Class•External•var = FALSE) ∧ (Class1•Abstract•var = FALSE) ∧ (Class•var ∈ superclassesOf(Class1•var))))))
∧ (∃ Class•var ∈ ConceptRepository•var . ((Class•DocumentName•var = 'SEAOutputMailbox') ∧ (Class•Abstract•var = TRUE) ∧ (Class•External•var = TRUE)))
∧ (∃ Class•var ∈ ConceptRepository•var . ((Class•DocumentName•var = 'SEAInterfaceChannel') ∧ (Class•Abstract•var = TRUE) ∧ (Class•External•var = TRUE)))))

6	**OOSpecification**	'startSpecification' . DocumentName . MotherSpecification . SpecificationType . ConceptRepository . ConceptualModelRepository . SustainmentTable . ReferenceTable. 'endSpecification'

OOSpecification •pred :=
Pred01 ∧ DocumentName•pred ∧ MotherSpecification•pred ∧ SpecificationType•pred ∧ ConceptRepository•pred ∧ ConceptualModelRepository•pred∧ SustainmentTable•pred ∧ ReferenceTable•pred

7	DocumentName	String (terminal)	DocumentName•pred := TRUE
8	MotherSpecification	'StartMother' .'endMother'	MotherSpecification •pred := TRUE
9		'StartMother' . OOSpecification . 'endMother'	MotherSpecification •pred := OOSpecification•pred
10		'StartMother' . OOSpecification . OOSpecificationSet . 'endMother'	MotherSpecification •pred := OOSpecification•pred ∧ OOSpecificationSet• pred
11	OOSpecificationSet	OOSpecification	OOSpecificationSet•pred := OOSpecification•pred
12		OOSpecification . OOSpecificationSet'	OOSpecificationSet•pred := OOSpecification•pred ∧ OOSpecificationSet'• pred
13	SpecificationType	'application'	SpecificationType •pred := TRUE
14		'framework'	SpecificationType •pred := TRUE
15		'component'	SpecificationType •pred := TRUE
16		'flexibleComponent'	SpecificationType •pred := TRUE

ConceptRepository

17	**ConceptRepository**	ClassRepository . BinaryRelationshipRepository . AggregationRepository . InheritanceRepository . AttributeRepository . MethodRepository . MethodParameterRepository . MethodTemporaryVariableRepository . TypeRepository . ObjectRepository . MessageRepository . ExternalReferenceRepository . MessageWrapperRepository . UseCaseRepository . UseCaseActorRepository . UseCaseRelationshipRepository . StateRepository . TransitionRepository . StatementRepository . LinkRepository	ConceptRepository•pred := ClassRepository•pred ∧ BinaryRelationshipRepository•pred ∧ AggregationRepository•pred ∧ InheritanceRepository•pred ∧ AttributeRepository•pred MethodRepository •pred ∧ MethodParameterRepository•pred ∧ MethodTemporaryVariableRepository•pred ∧ TypeRepository•pred ∧ ObjectRepository•pred ∧ MessageRepository•pred ∧ ExternalReferenceRepository•pred ∧ MessageWrapperRepository•pred ∧ UseCaseRepository•pred ∧ UseCaseActorRepository•pred ∧ UseCaseRelationshipRepository•pred ∧ StateRepository•pred ∧ TransitionRepository•pred ∧ StatementRepository•pred ∧ LinkRepository• pred
18	**ClassRepository**	'startRepositoryClass' . Class . 'endRepositoryClass'	ClassRepository•pred := Class•pred
19		'startRepositoryClass' . Class . ClassSet . 'endRepositoryClass'	ClassRepository•pred := Class•pred ∧ ClassSet•pred
20	ClassSet	Class	ClassSet•pred := Class•pred
21		Class . ClassSet'	ClassSet•pred := Class•pred ∧ ClassSet'•pred
22	**BinaryRelationshipRepository**	'startRepositoryBinaryRelationship' . 'endRepositoryBinaryRelationship'	BinaryRelationshipRepository•pred := TRUE
23		'startRepositoryBinaryRelationship' . BinaryRelationship . 'endRepositoryBinaryRelationship'	BinaryRelationshipRepository•pred := BinaryRelationship•pred
24		'startRepositoryBinaryRelationship' . BinaryRelationship . BinaryRelationshipSet . 'endRepositoryBinaryRelationship'	BinaryRelationshipRepository•pred := BinaryRelationship•pred ∧ BinaryRelationshipSet•pred
25	BinaryRelationshipSet	BinaryRelationship	BinaryRelationshipSet•pred := BinaryRelationship•pred
26		BinaryRelationship . BinaryRelationshipSet'	BinaryRelationshipSet•pred := BinaryRelationship•pred ∧ BinaryRelationshipSet'•pred
27	**AggregationRepository**	'startRepositoryAggregation' . 'endRepositoryAggregation'	AggregationRepository•pred := TRUE
28		'startRepositoryAggregation' . Aggregation . 'endRepositoryAggregation'	AggregationRepository•pred := Aggregation•pred
20		'startRepositoryAggregation' . Aggregation . AggregationSet . 'endRepositoryAggregation'	AggregationRepository•pred := Aggregation•pred ∧ AggregationSet•pred
30	AggregationSet	Aggregation	AggregationSet•pred := Aggregation•pred
31		Aggregation . AggregationSet'	AggregationSet•pred := Aggregation•pred ∧ AggregationSet'•pred
32	**InheritanceRepository**	'startRepositoryInheritance' . 'endRepositoryInheritance'	InheritanceRepository•pred := TRUE

33		'startRepositoryInheritance' . Inheritance . 'endRepositoryInheritance'	InheritanceRepository•pred := Inheritance•pred
34		'startRepositoryInheritance' . Inheritance . InheritanceSet . 'endRepositoryInheritance'	InheritanceRepository•pred := Inheritance•pred ∧ InheritanceSet•pred
35	InheritanceSet	Inheritance	InheritanceSet•pred := Inheritance•pred
36		Inheritance . InheritanceSet'	InheritanceSet•pred := Inheritance•pred ∧ InheritanceSet'•pred
37	**AttributeRepository**	'startRepositoryAttribute' . 'endRepositoryAttribute'	AttributeRepository•pred := TRUE
38		'startRepositoryAttribute' . Attribute . 'endRepositoryAttribute'	AttributeRepository•pred := Attribute•pred
39		'startRepositoryAttribute' . Attribute . AttributeSet . 'endRepositoryAttribute'	AttributeRepository•pred := Attribute•pred ∧ AttributeSet•pred
40	AttributeSet	Attribute	AttributeSet•pred := Attribute•pred
41		Attribute . AttributeSet'	AttributeSet•pred := Attribute•pred ∧ AttributeSet'•pred
42	**MethodRepository**	'startRepositoryMethod' . 'endRepositoryMethod'	MethodRepository•pred := TRUE
43		'startRepositoryMethod' . Method . 'endRepositoryMethod'	MethodRepository•pred := Method•pred
44		'startRepositoryMethod' . Method . MethodSet . 'endRepositoryMethod'	MethodRepository•pred := Method•pred ∧ MethodSet•pred
45	MethodSet	Method	MethodSet•pred := Method•pred
46		Method . MethodSet	MethodSet•pred := Method•pred ∧ MethodSet'•pred
47	**MethodParameterRepository**	'startRepositoryMethodParameter' . 'endRepositoryMethodParameter'	MethodParameterRepository•pred := TRUE
48		'startRepositoryMethodParameter' . MethodParameter . 'endRepositoryMethodParameter'	MethodParameterRepository•pred := MethodParameter•pred
49		'startRepositoryMethodParameter' . MethodParameter . MethodParameterSet . 'endRepositoryMethodParameter'	MethodParameterRepository•pred := MethodParameter•pred ∧ MethodParameterSet•pred
50	MethodParameterSet	MethodParameter	MethodParameterSet•pred := MethodParameter•pred
51		MethodParameter . MethodParameterSet'	MethodParameterSet•pred := MethodParameter•pred ∧ MethodParameterSet'•pred
52	**MethodTemporaryVariableRepository**	'startRepositoryMethodTemporaryVariable' . 'endRepositoryMethodTemporaryVariable'	MethodTemporaryVariableRepository•pred := TRUE
53		'startRepositoryMethodTemporaryVariable' . MethodTemporaryVariable . 'endRepositoryMethodTemporaryVariable'	MethodTemporaryVariableRepository•pred := MethodTemporaryVariable• pred

54		'startRepositoryMethodTemporaryVariable' . MethodTemporaryVariable . MethodTemporaryVariableSet . 'endRepositoryMethodTemporaryVariable'	MethodTemporaryVariableRepository•pred := MethodTemporaryVariable• pred ∧ MethodTemporaryVariableSet•pred
55	MethodTemporaryVariableSet	MethodTemporaryVariable	MethodTemporaryVariableSet•pred := MethodTemporaryVariable•pred
56		MethodTemporaryVariable . MethodTemporaryVariableSet'	MethodTemporaryVariableSet•pred := MethodTemporaryVariable•pred ∧ MethodTemporaryVariableSet'•pred
57	**TypeRepository**	'startRepositoryType' . 'endRepositoryType'	TypeRepository•pred := TRUE
58		'startRepositoryType' . Type . 'endRepositoryType'	TypeRepository•pred := Type•pred
59		'startRepositoryType' . Type . TypeSet . 'endRepositoryType'	TypeRepository•pred := Type•pred ∧ TypeSet•pred
60	TypeSet	Type	TypeSet•pred := Type•pred
61		Type . TypeSet'	TypeSet•pred := Type•pred ∧ TypeSet'•pred
62	**ObjectRepository**	'startRepositoryObject' . 'endRepositoryObject'	ObjectRepository•pred := TRUE
64		'startRepositoryObject' . Object . 'endRepositoryObject'	ObjectRepository•pred := Object•pred
64		'startRepositoryObject' . Object . ObjectSet . 'endRepositoryObject'	ObjectRepository•pred := Object•pred ∧ ObjectSet•pred
65	ObjectSet	Object	ObjectSet•pred := Object•pred
66		Object . ObjectSet'	ObjectSet•pred := Object•pred ∧ ObjectSet'•pred
67	**MessageRepository**	'startRepositoryMessage' . 'endRepositoryMessage'	MessageRepository•pred := TRUE
68		'startRepositoryMessage' . Message . 'endRepositoryMessage'	MessageRepository•pred := Message•pred
69		'startRepositoryMessage' . Message . MessageSet . 'endRepositoryMessage'	MessageRepository•pred := Message•pred ∧ MessageSet•pred
70	MessageSet	Message	MessageSet•pred := Message•pred
71		Message . MessageSet'	MessageSet•pred := Message•pred ∧ MessageSet'•pred
72	**ExternalReferenceRepository**	'startRepositoryExternalReference' . 'endRepositoryExternalReference'	ExternalReferenceRepository•pred := TRUE
73		'startRepositoryExternalReference' . ExternalReference . 'endRepositoryExternalReference'	ExternalReferenceRepository•pred := ExternalReference•pred
74		'startRepositoryExternalReference' . ExternalReference . ExternalReferenceSet . 'endRepositoryExternalReference'	ExternalReferenceRepository•pred := ExternalReference•pred ∧ ExternalReferenceSet•pred
75	ExternalReferenceSet	ExternalReference	ExternalReferenceSet•pred := ExternalReference•pred

76		ExternalReference . ExternalReferenceSet'	ExternalReferenceSet•pred := ExternalReference•pred ∧ ExternalReferenceSet'•pred
77	**MessageWrapperRepository**	'startRepositoryMessageWrapper' . 'endRepositoryMessageWrapper'	MessageWrapperRepository•pred := TRUE
78		'startRepositoryMessageWrapper' . MessageWrapper . 'endRepositoryMessageWrapper'	MessageWrapperRepository•pred := MessageWrapper•pred
79		'startRepositoryMessageWrapper' . MessageWrapper . MessageWrapperSet . 'endRepositoryMessageWrapper'	MessageWrapperRepository•pred := MessageWrapper•pred ∧ MessageWrapperSet•pred
80	MessageWrapperSet	MessageWrapper	MessageWrapperSet•pred := MessageWrapper•pred
81		MessageWrapper . MessageWrapperSet'	MessageWrapperSet•pred := MessageWrapper•pred ∧ MessageWrapperSet'•pred
82	**UseCaseRepository**	'startRepositoryUseCase' . 'endRepositoryUseCase'	UseCaseRepository•pred := TRUE
83		'startRepositoryUseCase' . UseCase . 'endRepositoryUseCase'	UseCaseRepository•pred := UseCase•pred
84		'startRepositoryUseCase' . UseCase . UseCaseSet . 'endRepositoryUseCase'	UseCaseRepository•pred := UseCase•pred ∧ UseCaseSet•pred
85	UseCaseSet	UseCase	UseCaseSet•pred := UseCase•pred
86		UseCase . UseCaseSet	UseCaseSet•pred := UseCase•pred ∧ UseCaseSet'•pred
87	**UseCaseActorRepository**	'startRepositoryUseCaseActor' . 'endRepositoryUseCaseActor'	UseCaseActorRepository•pred := TRUE
88		'startRepositoryUseCaseActor' . UseCaseActor . 'endRepositoryUseCaseActor'	UseCaseActorRepository•pred := UseCaseActor•pred
89		'startRepositoryUseCaseActor' . UseCaseActor . UseCaseActorSet . 'endRepositoryUseCaseActor'	UseCaseActorRepository•pred := UseCaseActor•pred ∧ UseCaseActorSet•pred
90	UseCaseActorSet	UseCaseActor	UseCaseActorSet•pred := UseCaseActor•pred
91		UseCaseActor . UseCaseActorSet'	UseCaseActorSet•pred := UseCaseActor•pred ∧ UseCaseActorSet'•pred
92	**UseCaseRelationshipRepository**	'startRepositoryUseCaseRelationship' . 'endRepositoryUseCaseRelationship'	UseCaseRelationshipRepository•pred := TRUE
93		'startRepositoryUseCaseRelationship' . UseCaseRelationship . 'endRepositoryUseCaseRelationship'	UseCaseRelationshipRepository•pred := UseCaseRelationship•pred
94		'startRepositoryUseCaseRelationship' . UseCaseRelationship . UseCaseRelationshipSet . 'endRepositoryUseCaseRelationship'	UseCaseRelationshipRepository•pred := UseCaseRelationship•pred ∧ UseCaseRelationshipSet•pred
95	UseCaseRelationshipSet	UseCaseRelationship	UseCaseRelationshipSet•pred := UseCaseRelationship•pred

96		UseCaseRelationship . UseCaseRelationshipSet'	UseCaseRelationshipSet•pred := UseCaseRelationship•pred ∧ UseCaseRelationshipSet'•pred
97	**StateRepository**	'startRepositoryState' . 'endRepositoryState'	StateRepository•pred := TRUE
98		'startRepositoryState' . State . 'endRepositoryState'	StateRepository•pred := State•pred
99		'startRepositoryState' . State . StateSet . 'endRepositoryState'	StateRepository•pred := State•pred ∧ StateSet•pred
100	StateSet	State	StateSet•pred := State•pred
101		State . StateSet'	StateSet•pred := State•pred ∧ StateSet'•pred
102	**TransitionRepository**	'startRepositoryTransition' . 'endRepositoryTransition'	TransitionRepository•pred := TRUE
103		'startRepositoryTransition' . Transition . 'endRepositoryTransition'	TransitionRepository•pred := Transition•pred
104		'startRepositoryTransition' . Transition . TransitionSet . 'endRepositoryTransition'	TransitionRepository•pred := Transition•pred ∧ TransitionSet•pred
105	TransitionSet	Transition	TransitionSet•pred := Transition•pred
106		Transition . TransitionSet'	TransitionSet•pred := Transition•pred ∧ TransitionSet'•pred
107	**StatementRepository**	'startRepositoryStatement' . 'endRepositoryStatement'	StatementRepository•pred := TRUE
108		'startRepositoryStatement' . Statement . 'endRepositoryStatement'	StatementRepository•pred := Statement•pred
109		'startRepositoryStatement' . Statement . StatementSet . 'endRepositoryStatement'	StatementRepository•pred := Statement•pred ∧ StatementSet•pred
110	StatementSet	Statement	StatementSet•pred := Statement•pred
111		Statement . StatementSet'	StatementSet•pred := Statement•pred ∧ StatementSet'•pred
112	Statement	CompositeStatement	Statement•pred := CompositeStatement•pred
113		DoubleCompositeStatement	Statement•pred := DoubleCompositeStatement•pred
114		MultiplePointerStatement	Statement•pred := MultiplePointerStatement•pred
115		PointerStatement	Statement•pred := PointerStatement•pred
116		DoublePointerStatement	Statement•pred := DoublePointerStatement•pred
117		MessageStatement	Statement•pred := MessageStatement•pred
118		TextualStatement	Statement•pred := TextualStatement•pred
119	**LinkRepository**	'startRepositoryLink' . 'endRepositoryLink'	LinkRepository•pred := TRUE
120		'startRepositoryLink' . Link . 'endRepositoryLink'	LinkRepository•pred := Link•pred
121		'startRepositoryLink' . Link . LinkSet . 'endRepositoryLink'	LinkRepository•pred := Link•pred ∧ LinkSet•pred
122	LinkSet	Link	LinkSet•pred := Link•pred
123		Link . LinkSet'	LinkSet•pred := Link•pred ∧ LinkSet'•pred

	SustainmentTable, ReferenceTable		
124	**SustainmentTable**	'startSustainmentTable' . 'endSustainmentTable'	SustainmentTable•pred := TRUE
125		'startSustainmentTable' . specificationElementPair . 'endSustainmentTable'	SustainmentTable•pred := specificationElementPair•pred
126		'startSustainmentTable' . specificationElementPair . specificationElementPairSet . 'endSustainmentTable'	SustainmentTable•pred := specificationElementPair•pred ∧ specificationElementPairSet•pred
127	**ReferenceTable**	'startReferenceTable' . 'endReferenceTable'	ReferenceTable•pred := TRUE
128		'startReferenceTable' . SpecificationElementPair . 'endReferenceTable'	ReferenceTable•pred := specificationElementPair•pred
129		'startReferenceTable' . SpecificationElementPair . SpecificationElementPairSet . 'endReferenceTable'	ReferenceTable•pred := specificationElementPair•pred ∧ specificationElementPairSet•pred
130	SpecificationElementPairSet	SpecificationElementPair	SpecificationElementPairSet •pred := specificationElementPair•pred
131		SpecificationElementPair . SpecificationElementPairSet'	SpecificationElementPairSet•pred := specificationElementPair•pred ∧ specificationElementPairSet'•pred
132	SpecificationElementPair	SpecificationElement1 . SpecificationElement2	SpecificationElementPair•pred := specificationElement1•pred ∧ specificationElement2•pred ∧ (SpecificationElement1•var ≠ SpecificationElement2•var)
133	SpecificationElement1	SpecificationElement	SpecificationElement1•pred := SpecificationElement•pred
134	SpecificationElement2	SpecificationElement	SpecificationElement2•pred := SpecificationElement•pred
135	**SpecificationElement**	Concept	SpecificationElement•pred := Concept•pred
136		ConceptualModel	SpecificationElement•pred := ConceptualModel•pred
137	**Concept**	Class	Concept •pred := Class•pred
138		BinaryRelationship	Concept •pred := BinaryRelationship•pred
130		Aggregation	Concept •pred := Aggregation•pred
140		Inheritance	Concept •pred := Inheritance•pred
141		Attribute	Concept •pred := Attribute•pred
142		Method	Concept •pred := Method•pred
143		MethodParameter	Concept •pred := MethodParameter•pred
144		MethodTemporaryVariable	Concept •pred := MethodTemporaryVariable•pred
145		Type	Concept •pred := Type•pred
146		Object	Concept •pred := Object•pred
147		Message	Concept •pred := Message•pred
148		ExternalReference	Concept •pred := ExternalReference•pred
149		MessageWrapper	Concept •pred := MessageWrapper•pred

150		UseCase	Concept •pred := UseCase•pred
151		UseCaseActor	Concept •pred := UseCaseActor•pred
152		UseCaseRelationship	Concept •pred := UseCaseRelationship•pred
153		State	Concept •pred := State•pred
154		Transition	Concept •pred := Transition•pred
155		CompositeStatement	Concept •pred := CompositeStatement•pred
156		DoubleCompositeStatement	Concept •pred := DoubleCompositeStatement•pred
157		MultiplePointerStatement	Concept •pred := MultiplePointerStatement•pred
158		PointerStatement	Concept •pred := PointerStatement•pred
159		DoublePointerStatement	Concept •pred := DoublePointerStatement•pred
160		MessageStatement	Concept •pred := MessageStatement•pred
161		TextualStatement	Concept •pred := TextualStatement•pred
162		Link	Concept •pred := Link•pred
163	**ConceptualModel**	UseCaseDiagram	ConceptualModel•pred := UseCaseDiagram•pred
164		ActivityDiagram	ConceptualModel•pred := ActivityDiagram•pred
165		ClassDiagram	ConceptualModel•pred := ClassDiagram•pred
166		SequenceDiagram	ConceptualModel•pred := SequenceDiagram•pred
167		StateTransitionDiagram	ConceptualModel•pred := StateTransitionDiagram•pred
168		MethodBodyDiagram	ConceptualModel•pred := MethodBodyDiagram•pred
	ConceptualModelRepository		
169	**ConceptualModelRepository**	UseCaseDiagramRepository . ActivityDiagramRepository . ClassDiagramRepository . SequenceDiagramRepository . StateTransitionDiagramRepository . MethodBodyDiagramRepository	ConceptualModelRepository•pred := UseCaseDiagramRepository•pred ∧ ActivityDiagramRepository•pred ∧ ClassDiagramRepository•pred ∧ SequenceDiagramRepository•pred ∧ StateTransitionDiagramRepository •pred ∧ MethodBodyDiagramRepository•pred
170	**UseCaseDiagramRepository**	'startRepositoryUseCaseDiagram' . 'endRepositoryUseCaseDiagram'	UseCaseDiagramRepository•pred := TRUE
171		'startRepositoryUseCaseDiagram' . UseCaseDiagram . 'endRepositoryUseCaseDiagram'	UseCaseDiagramRepository•pred := UseCaseDiagram•pred
172	**ActivityDiagramRepository**	'startRepositoryActivityDiagram' . 'endRepositoryActivityDiagram'	ActivityDiagramRepository•pred := TRUE
173		'startRepositoryActivityDiagram' . ActivityDiagram . 'endRepositoryActivityDiagram'	ActivityDiagramRepository•pred := ActivityDiagram•pred

174	**ClassDiagramRepository**	'startRepositoryClassDiagram' . ClassDiagram . 'endRepositoryClassDiagram'	ClassDiagramRepository•pred := ClassDiagram•pred
175		'startRepositoryClassDiagram' . ClassDiagram . ClassDiagramSet . 'endRepositoryClassDiagram'	ClassDiagramRepository•pred := ClassDiagram•pred ∧ ClassDiagramSet•pred
176	ClassDiagramSet	ClassDiagram	ClassDiagramSet •pred := ClassDiagram•pred
177		ClassDiagram . ClassDiagramSet'	ClassDiagramSet •pred := ClassDiagram•pred ∧ ClassDiagramSet'•pred
178	**SequenceDiagramRepository**	'startRepositorySequenceDiagram' . 'endRepositorySequenceDiagram'	SequenceDiagramRepository•pred := TRUE
179		'startRepositorySequenceDiagram' . SequenceDiagram . 'endRepositorySequenceDiagram'	SequenceDiagramRepository•pred := SequenceDiagram•pred
180		'startRepositorySequenceDiagram' . SequenceDiagram . SequenceDiagramSet . 'endRepositorySequenceDiagram'	SequenceDiagramRepository•pred := SequenceDiagram•pred ∧ SequenceDiagramSet•pred
181	SequenceDiagramSet	SequenceDiagram	SequenceDiagramSet •pred := SequenceDiagram•pred
182		SequenceDiagram . SequenceDiagramSet'	SequenceDiagramSet •pred := SequenceDiagram•pred ∧ SequenceDiagramSet'•pred
183	**StateTransitionDiagramRepository**	'startRepositoryStateTransitionDiagram' . 'endRepositoryStateTransitionDiagram'	StateTransitionDiagramRepository•pred := TRUE
184		'startRepositoryStateTransitionDiagram' . StateTransitionDiagram . 'endRepositoryStateTransitionDiagram'	StateTransitionDiagramRepository•pred := StateTransitionDiagram•pred
185		'startRepositoryStateTransitionDiagram' . StateTransitionDiagram . StateTransitionDiagramSet . 'endRepositoryStateTransitionDiagram'	StateTransitionDiagramRepository•pred := StateTransitionDiagram•pred ∧ StateTransitionDiagramSet•pred
186	StateTransitionDiagramSet	StateTransitionDiagram	StateTransitionDiagramSet •pred := StateTransitionDiagram•pred
187		StateTransitionDiagram . StateTransitionDiagramSet'	StateTransitionDiagramSet •pred := StateTransitionDiagram•pred ∧ StateTransitionDiagramSet'•pred
188	**MethodBodyDiagramRepository**	'startRepositoryMethodBodyDiagram' . 'endRepositoryMethodBodyDiagram'	MethodBodyDiagramRepository•pred := TRUE
189		'startRepositoryMethodBodyDiagram' . MethodBodyDiagram . 'endRepositoryMethodBodyDiagram'	MethodBodyDiagramRepository•pred := MethodBodyDiagram•pred
190		'startRepositoryMethodBodyDiagram' . MethodBodyDiagram . MethodBodyDiagramSet . 'endRepositoryMethodBodyDiagram'	MethodBodyDiagramRepository•pred := MethodBodyDiagram•pred ∧ MethodBodyDiagramSet•pred
191	MethodBodyDiagramSet	MethodBodyDiagram	MethodBodyDiagramSet •pred := MethodBodyDiagram•pred
192		MethodBodyDiagram . MethodBodyDiagramSet'	MethodBodyDiagramSet •pred := MethodBodyDiagram•pred ∧ MethodBodyDiagramSet'•pred

	Concept	
193	**Class**	'startClass' . DocumentName . ElementLinks . External . ExternalOrigin . ConceptualModelReferers . Inheritance . Redefinable . Essential . Parametrizable . Abstract . Category . 'endClass'

Class•pred :=
Pred02 ∧ Pred03 ∧ Pred04 ∧ DocumentName•pred ∧ ElementLinks•pred ∧ External•pred ∧ ExternalOrigin•pred ∧ ConceptualModelReferers•pred ∧ Inheritance•pred ∧ Redefinable•pred ∧ Essential•pred ∧ Parametrizable•pred ∧ Abstract•pred ∧ Category•pred

Pred02 :=
(∀ Class1•var ∈ ConceptRepository•var . (Class•var ≠ Class1•var) ⇒ (Class•DocumentName•var ≠ Class1•DocumentName•var))
∧
((∃ Method•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = Class•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Method•var)))
∨ (∃ Attribute•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = Class•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Attribute•var))))
∧
((Class•Abstract•var = TRUE) ⇒
(∃ Method•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . (SpecificationElementPair•SpecificationElement1•var = Class•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Method•var) ∧ (Method•Classification•var = 'abstract'))
∨ (∃ Method•var ∈ ConceptRepository•var . (Method•var ∈ inheritedMethodsOf(Class•var)[2]) ∧ (Method•Classification•var = 'abstract')))
∧
((Class•Abstract•var = FALSE) ⇒
((∄ Method•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = Class•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Method•var) ∧ (Method•Classification•var = 'abstract')))
∨ (∄ Method•var ∈ ConceptRepository•var . ((Method•var ∈ inheritedMethodsOf(Class•var)) ∧ (Method•Classification•var = 'abstract')))))
∧
((Class•Parametrizable•var = TRUE) ⇒
((∃ Method•var ∈ ConceptRepository•var . ∃ SpecificationElementPair1•var ∈ SustainmentTable•var . ∃ MethodParameter•var ∈ ConceptRepository•var . ∃ SpecificationElementPair2•var ∈ SustainmentTable•var . ((SpecificationElementPair1•SpecificationElement1•var = Class•var) ∧ (SpecificationElementPair1•SpecificationElement2•var

---

[2] "inheritedMethodsOf(Class•var)" é uma função que retorna o conjunto de métodos herdados de uma ocorrência de Class (Class•var, no caso), contidas no repositório de conceitos (ConceptRepository•var). Uma descrição de classe não possui métodos associados. A associação de um método a uma classe ocorre através da definição de um par (classe, método) na tabela de sustentação (sustainment table), onde a classe é o elemento sustentador e o método, o elemento sustentado. A semântica disto é que a remoção de uma classe de uma especificação implica também na remoção do método. Determinar quais são os métodos de uma classe corresponde a buscar na tabela de sustentação quais métodos participam de pares com a classe considerada. Se uma ocorrência de classe referencia uma ocorrência de herança, então esta classe possui superclasse, que é a classe identificada como "Superclasse", apontada por esta ocorrência de herança. O conjunto dos métodos herdados da superclasse corresponde ao conjunto de métodos associados à superclasse, excluídos os métodos sobrepostos, ou seja aqueles com coincidência das características "DocumentName" (string) e "Parameters" (integer, que descreve o comprimento da lista de parâmetros). Caso a superclasse da classe considerada possua superclasse esta busca continua recursivamente, porém ao considerar o critério de sobreposição, além dos métodos associados à classe considerada, devem ser também levados em conta os métodos já incluídos no conjunto de métodos herdados. Este algoritmo está implementado no framework OCEAN.

= Method•var) ∧ (SpecificationElementPair2•SpecificationElement1•var = Method•var) ∧ (SpecificationElementPair2•SpecificationElement2•var = MethodParameter•var) ∧ (Method•MetaMethod•var = TRUE) ∧ (Method•ReturnedType•Class•var = Class•var)))
∧ (∃ Method•var ∈ ConceptRepository•var . ∃ MethodParameter•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((Method•var ∈ inheritedMethodsOf(Class•var)) ∧ (SpecificationElementPair•SpecificationElement1•var = Method•var) ∧ (SpecificationElementPair•SpecificationElement2•var = MethodParameter•var) ∧ (Method•MetaMethod•var = TRUE) ∧ (Method•ReturnedType•Class•var ∈ superclassesOf(Class•var))))))
∧
(∀ Link•var ∈ Class•ElementLinks•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . (SpecificationElementPair•SpecificationElement1•var = Class•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Link•var))

Pred03 :=
(Class•ExternalOrigin•var ∈ MotherSpecification•var)
∧
(∃ Class1•var ∈ Class•ExternalOrigin•ConceptRepository•var . ((Class•DocumentName•var = Class1•DocumentName•var) ∧ (Class•Redefinable•var = Class1•Redefinable•var) ∧ (Class•Essential•var = Class1•Essential•var) ∧ (Class•Parametrizable•var = Class1•Parametrizable•var) ∧ (Class•Abstract•var = Class1•Abstract•var)))

Pred04 :=
(Inheritance•.Subclass•var = Class•var)
∧
(∃ SpecificationElementPair•var ∈ ReferenceTable•var . (SpecificationElementPair•SpecificationElement1•var = Inheritance•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Class•var))

194	**Class**	'startClass' . DocumentName . ElementLinks . External . ConceptualModelReferers . Inheritance . Redefinable . Essential . Parametrizable . Abstract . Category . 'endClass'

Class•pred :=
Pred02 ∧ Pred04 ∧ DocumentName•pred ∧ ElementLinks•pred ∧ External•pred ∧ ConceptualModelReferers•pred ∧ Inheritance•pred ∧ Redefinable•pred ∧ Essential•pred ∧ Parametrizable•pred ∧ Abstract•pred ∧ Category•pred

195	**Class**	'startClass' . DocumentName . ElementLinks . External . ExternalOrigin . ConceptualModelReferers . Redefinable . Essential . Parametrizable . Abstract . Category . 'endClass'

Class•pred :=
Pred02 ∧ Pred03 ∧ DocumentName•pred ∧ ElementLinks•pred ∧ External•pred ∧ ExternalOrigin•pred ∧ ConceptualModelReferers•pred ∧ Redefinable•pred ∧ Essential•pred ∧ Parametrizable•pred ∧ Abstract•pred ∧ Category•pred

196	**Class**	'startClass' . DocumentName . ElementLinks . External . ConceptualModelReferers . Redefinable . Essential . Parametrizable . Abstract . Category . 'endClass'

Class•pred :=
Pred02 ∧ DocumentName•pred ∧ ElementLinks•pred ∧ External•pred ∧ ConceptualModelReferers•pred ∧ Redefinable•pred ∧ Essential•pred ∧ Parametrizable•pred ∧ Abstract•pred ∧ Category•pred

197	ElementLinks	'startElementLinks' 'endElementLinks'	ElementLinks•pred := TRUE
198		'startElementLinks' . Link . 'endElementLinks'	ElementLinks•pred := Link•pred
199		'startElementLinks' . Link . LinkSet . 'endElementLinks'	ElementLinks•pred := Link•pred ∧ LinkSet•pred
200	External	TRUE	External•pred := TRUE
201		FALSE	External•pred := TRUE
202	ExternalOrigin	OOSpecification	ExternalOrigin•pred := OOSpecification•pred
203	ConceptualModelReferers	'startConceptualModelReferers' . ConceptualModel . 'endConceptualModelReferers'	ConceptualModelReferers•pred := ConceptualModel•pred
204		'startConceptualModelReferers' . ConceptualModel . ConceptualModelSet . 'endConceptualModelReferers'	ConceptualModelReferers•pred := ConceptualModel•pred ∧ ConceptualModelSet•pred
205	ConceptualModelSet	ConceptualModel	ConceptualModelSet•pred := ConceptualModel•pred
206		ConceptualModel . ConceptualModelSet'	ConceptualModelSet•pred := ConceptualModel•pred ∧ ConceptualModelSet'•pred
207	Redefinable	TRUE	Redefinable•pred := TRUE
208		FALSE	Redefinable•pred := TRUE
209	Essential	TRUE	Essential•pred := TRUE
210		FALSE	Essential•pred := TRUE
211	Parametrizable	TRUE	Parametrizable•pred := TRUE
212		FALSE	Parametrizable•pred := TRUE
213	Abstract	TRUE	Abstract•pred := TRUE
214		FALSE	Abstract•pred := TRUE
215	Category	String	Category•pred := TRUE
216	**BinaryRelationship**	'startBinaryRelationship' . DocumentName . ElementLinks . External . ExternalOrigin . ConceptualModelReferers . Class1 . Class2 . Cardinality1 . Cardinality2 . ReferenceMainentance1 . ReferenceMainentance2 . DynamicRelationship1 . DynamicRelationship2 . Role1 . Role2 . 'endBinaryRelationship'	

BinaryRelationship•pred :=
pred05 ∧ pred06 ∧ DocumentName•pred ∧ ElementLinks•pred ∧ External•pred ∧ ExternalOrigin•pred ∧ ConceptualModelReferers•pred ∧ Class1•pred ∧ Class2•pred ∧ Cardinality1•pred ∧ Cardinality2•pred ∧ ReferenceMainentance1•pred ∧ ReferenceMainentance2•pred ∧ DynamicRelationship1•pred ∧ DynamicRelationship2•pred ∧ Role1•pred ∧ Role2•pred

pred05 :=
(∀ BinaryRelationship1•var ∈ ConceptRepository•var . ((BinaryRelationship•var ≠ BinaryRelationship1•var) ⇒ ((BinaryRelationship•DocumentName•var ≠ BinaryRelationship1•DocumentName•var) ∨ (BinaryRelationship•Class1•var ≠ BinaryRelationship1•Class1•var) ∨ (BinaryRelationship•Class2•var ≠ BinaryRelationship1•Class2•var))))
∧

(∃ SpecificationElementPair•var ∈ SustainmentTable•var . (SpecificationElementPair•SpecificationElement1•var = BinaryRelationship•Class1•var) ∧ (SpecificationElementPair•SpecificationElement2•var = BinaryRelationship•var))
∧
(∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = BinaryRelationship•Class2•var) ∧ (SpecificationElementPair•SpecificationElement2•var = BinaryRelationship•var)))
∧
((BinaryRelationship•ReferenceMainentance1•var = TRUE) ⇒ (∃ Attribute•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = BinaryRelationship•Class1•var) ∧ SpecificationElementPair•SpecificationElement2•var = Attribute•var) ∧ (Attribute•Type•Class•var = BinaryRelationship•Class2•var))))
∧
((BinaryRelationship•ReferenceMainentance2•var = TRUE) ⇒ (∃ Attribute•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = BinaryRelationship•Class2•var) ∧ SpecificationElementPair•SpecificationElement2•var = Attribute•var) ∧ (Attribute•Type•Class•var = BinaryRelationship•Class1•var))))
∧
((BinaryRelationship•External•var = TRUE) ⇒ ((BinaryRelationship•Class1•External•var = TRUE) ∧ (BinaryRelationship•Class2•External•var = TRUE)))
∧
(∀ Link•var ∈ BinaryRelationship•ElementLinks•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = BinaryRelationship•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Link•var)))

pred06 :=
(BinaryRelationship•ExternalOrigin•var ∈ MotherSpecification•var)
∧
(BinaryRelationship•ExternalOrigin•var = BinaryRelationship•Class1•ExternalOrigin•var) ∧ (BinaryRelationship•ExternalOrigin•var = BinaryRelationship•Class2•ExternalOrigin•var)

217	**BinaryRelationship**	'startBinaryRelationship' . DocumentName . ElementLinks . External . ConceptualModelReferers . Class1 . Class2 . Cardinality1 . Cardinality2 . ReferenceMainentance1 . ReferenceMainentance2 . DynamicRelationship1 . DynamicRelationship2 . Role1 . Role2 . 'endBinaryRelationship'

BinaryRelationship•pred :=
pred05 ∧ DocumentName•pred ∧ ElementLinks•pred ∧ External•pred ∧ ConceptualModelReferers•pred ∧ Class1•pred ∧ Class2•pred ∧ Cardinality1•pred ∧ Cardinality2•pred ∧ ReferenceMainentance1•pred ∧ ReferenceMainentance2•pred ∧ DynamicRelationship1•pred ∧ DynamicRelationship2•pred ∧ Role1•pred ∧ Role2•pred

218	Class1	Class	Class1•pred := Class•pred
219	Class2	Class	Class2•pred := Class•pred
220	Cardinality1	'um'	Cardinality1•pred := TRUE
221		'muitos'	Cardinality1•pred := TRUE
222		'opcional'	Cardinality1•pred := TRUE
223		'umMais'	Cardinality1•pred := TRUE
224	Cardinality2	'um'	Cardinality2•pred := TRUE

225		'muitos'	Cardinality2•pred := TRUE
226		'opcional'	Cardinality2•pred := TRUE
227		'umMais'	Cardinality2•pred := TRUE
228	ReferenceMainentance1	TRUE	ReferenceMainentance1•pred := TRUE
229		FALSE	ReferenceMainentance1•pred := TRUE
230	ReferenceMainentance2	TRUE	ReferenceMainentance2•pred := TRUE
231		FALSE	ReferenceMainentance2•pred := TRUE
232	DynamicRelationship1	TRUE	DynamicRelationship1•pred := TRUE
233		FALSE	DynamicRelationship1•pred := TRUE
234	DynamicRelationship2	TRUE	DynamicRelationship2•pred := TRUE
235		FALSE	DynamicRelationship2•pred := TRUE
236	Role1	String	Role1•pred := TRUE
237	Role2	String	Role2•pred := TRUE
238	**Aggregation**	'startAggregation' . ElementLinks . External . ExternalOrigin . ConceptualModelReferers . ClassWhole . ClassPart . PartCardinality . 'endAggregation'	

Aggregation•pred :=
Pred07 ∧ Pred08 ∧ DocumentName•pred ∧ ElementLinks•pred ∧ External•pred ∧ ExternalOrigin•pred ∧ ConceptualModelReferers•pred ∧ ClassWhole•pred ∧ ClassPart•pred ∧ PartCardinality•pred

Pred07 :=
(∀ Aggregation1•var ∈ ConceptRepository•var . ((Aggregation•var ≠ Aggregation1•var) ⇒ ((Aggregation•ClassWhole•var ≠ Aggregation1•ClassWhole•var) ∨ (Aggregation•ClassPart•var ≠ Aggregation1•ClassPart•var))))
∧
(∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = Aggregation•ClassWhole•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Aggregation•var)))
∧
(∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = Aggregation•ClassPart•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Aggregation•var)))
∧
(∃ Attribute•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = Aggregation•ClassWhole•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Attribute•var) ∧ (Attribute•Type•Class•var = Aggregation•ClassPart•var)))
∧
((Aggregation•External•var = TRUE) ⇒ ((Aggregation•ClassWhole•External•var = TRUE) ∧ (Aggregation•ClassPart•External•var = TRUE)))
∧
((Aggregation•External•var = FALSE) ⇒ (Aggregation•ClassWhole•External•var = FALSE))
∧
(∀ Link•var ∈ Aggregation•ElementLinks•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = Aggregation•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Link•var)))

Pred08 :=
(Aggregation•ExternalOrigin•var ∈ MotherSpecification•var)
∧
((Aggregation•ExternalOrigin•var = Aggregation•ClassWhole•ExternalOrigin•var) ∧ (Aggregation•ExternalOrigin•var = Aggregation•ClassPart•ExternalOrigin•var))

239	**Aggregation**	'startAggregation' . ElementLinks . External . ConceptualModelReferers . ClassWhole . ClassPart . PartCardinality . 'endAggregation'

Aggregation•pred :=
Pred07 ∧ DocumentName•pred ∧ ElementLinks•pred ∧ External•pred ∧ ConceptualModelReferers•pred ∧ ClassWhole•pred ∧ ClassPart•pred ∧ PartCardinality•pred

240	ClassWhole	Class	ClassWhole•pred := Class•pred
241	ClassPart	Class	ClassPart•pred := Class•pred
242	PartCardinality	'um'	PartCardinality•pred := TRUE
243		'muitos'	PartCardinality•pred := TRUE
244		'opcional'	PartCardinality•pred := TRUE
245		'umMais'	PartCardinality•pred := TRUE
246	**Inheritance**	'startInheritance' . ElementLinks . External . ExternalOrigin . ConceptualModelReferers . Superclass . Subclass . 'endInheritance'	

Inheritance•pred :=
Pred09 ∧ Pred10 ∧ DocumentName•pred ∧ ElementLinks•pred ∧ External•pred ∧ ExternalOrigin•pred ∧ ConceptualModelReferers•pred ∧ Superclass•pred ∧ Subclass•pred

Pred09 :=
(∀ Inheritance1•var ∈ ConceptRepository•var . ((Inheritance•var ≠ Inheritance1•var) ⇒ ((Inheritance•Superclass•var ≠ Inheritance1•Superclass•var) ∨ (Inheritance•Subclass•var ≠ Inheritance1•Subclass•var))))
∧
(∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = Inheritance•Superclass•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Inheritance•var)))
∧
(∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = Inheritance•Subclass•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Inheritance•var)))
∧
(Inheritance•Superclass•var ≠ Inheritance•Subclass•var)
∧
(Inheritance•Subclass•var ∉ superclassesOf(Inheritance•Superclass•var))
∧
((Inheritance•External•var = TRUE) ⇒ ((Inheritance•Superclass•External•var = TRUE) ∧ (Inheritance•Subclass•External•var = TRUE)))
∧
((Inheritance•External•var = FALSE) ⇒ (Inheritance•Subclass•External•var = FALSE))
∧

(∀ Link•var ∈ Inheritance•ElementLinks•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = Inheritance•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Link•var)))

Pred10 :=
(Inheritance1•ExternalOrigin•var ∈ MotherSpecification•var)
∧
((Inheritance•ExternalOrigin•var = Inheritance•Superclass•ExternalOrigin•var) ∧ (Inheritance•ExternalOrigin•var = Inheritance•Subclass•ExternalOrigin•var))

247	**Inheritance**	'startInheritance' . ElementLinks . External . ConceptualModelReferers . Superclass . Subclass . 'endInheritance'

Inheritance•pred :=
Pred09 ∧ DocumentName•pred ∧ ElementLinks•pred ∧ External•pred ∧ ConceptualModelReferers•pred ∧ Superclass•pred ∧ Subclass•pred

248	Superclass	Class	Superclass•pred := Class•pred
249	Subclass	Class	Subclass•pred := Class•pred
250	**Attribute**	'startAttribute' . DocumentName . ElementLinks . External . ExternalOrigin . Type . ObjectIsSet . Class . MetaAttribute . 'endAttribute'	

Attribute•pred :=
Pred11 ∧ Pred12 ∧ DocumentName•pred ∧ ElementLinks•pred ∧ External•pred ∧ ExternalOrigin•pred ∧ Type•pred ∧ ObjectIsSet•pred ∧ Class•pred ∧ MetaAttribute•pred

Pred11 :=
(∀ Attribute1•var ∈ ConceptRepository•var . ∀ SpecificationElementPair•var ∈ SustainmentTable•var . (((Attribute•var ≠ Attribute1•var) ∧ (Attribute•DocumentName•var = Attribute1•DocumentName•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Attribute1•var)) ⇒
((SpecificationElementPair•SpecificationElement1•var ≠ Attribute•Class•var) ∧ (SpecificationElementPair•SpecificationElement1•var ∉ superclassesOf(Attribute•Class•var))))
∧
(∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = Attribute•Class•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Attribute•var)))
∧
(∃ SpecificationElementPair•var ∈ ReferenceTable•var . ((SpecificationElementPair•SpecificationElement1•var = Attribute•Type•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Attribute•var)))
∧
(Attribute•External•var = Attribute•Class•External•var)
∧
(∀ Link•var ∈ Attribute•ElementLinks•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = Attribute•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Link•var)))

Pred12 :=
(Attribute•ExternalOrigin•var ∈ MotherSpecification•var)
∧
(Attribute•ExternalOrigin•var = Attribute•Class•ExternalOrigin•var)

251	**Attribute**	'startAttribute' . DocumentName . ElementLinks . External . Type . ObjectIsSet . Class . MetaAttribute . 'endAttribute'

Attribute•pred :=
Pred11 ∧ DocumentName•pred ∧ ElementLinks•pred ∧ External•pred ∧ Type•pred ∧ ObjectIsSet•pred ∧ Class•pred ∧ MetaAttribute•pred

252	ObjectIsSet	TRUE	ObjectIsSet•pred := TRUE
253		FALSE	ObjectIsSet•pred := TRUE
254	MetaAttribute	TRUE	MetaAttribute•pred := TRUE
255		FALSE	MetaAttribute•pred := TRUE
256	**Method**	'startMethod' . DocumentName . ElementLinks . External . ExternalOrigin . Class . ReturnedType . ReturnIsSet . Parameters . Classification . Private . MetaMethod . Category . 'endMethod'	

Method•pred :=
Pred13 ∧ Pred14 ∧ Pred15 ∧ DocumentName•pred ∧ ElementLinks•pred ∧ External•pred ∧ ExternalOrigin•pred ∧ Class•pred ∧ ReturnedType•pred ∧ ReturnIsSet•pred ∧ Parameters•pred ∧ Classification•pred ∧ Private•pred ∧ MetaMethod•pred ∧ Category•pred

Pred13 :=
(∀ Method1•var ∈ ConceptRepository•var . ((Method•var ≠ Method1•var) ⇒ ((Method•DocumentName•var ≠ Method1•DocumentName•var) ∨ (Method•Class•var ≠ Method1•Class•var) ∨ (Method•Parameters•var ≠ Method1•Parameters•var))))
∧
(∀ Method1•var ∈ ConceptRepository•var . (((Method•var ≠ Method1•var) ∧ (Method•DocumentName•var = Method1•DocumentName•var) ∧ (Method•Parameters•var = Method1•Parameters•var) ∧ (Method1•Class•var ∈ superclassesOf(Method•Class•var))) ⇒
({MethodParameter•var | (∃ SpecificationElementPair•var ∈ SustainmentTable•var . (SpecificationElementPair•SpecificationElement1•var = Method•var) ∧ (SpecificationElementPair•SpecificationElement2•var = MethodParameter•var))} ⊆ {MethodParameter•var | (∃ SpecificationElementPair•var ∈ SustainmentTable•var . (SpecificationElementPair•SpecificationElement1•var = Method1•var) ∧ (SpecificationElementPair•SpecificationElement2•var = MethodParameter•var))}
∧ {MethodParameter•var | (∃ SpecificationElementPair•var ∈ SustainmentTable•var . (SpecificationElementPair•SpecificationElement1•var = Method1•var) ∧ (SpecificationElementPair•SpecificationElement2•var = MethodParameter•var))} ⊆ {MethodParameter•var | (∃ SpecificationElementPair•var ∈ SustainmentTable•var . (SpecificationElementPair•SpecificationElement1•var = Method•var) ∧ (SpecificationElementPair•SpecificationElement2•var = MethodParameter•var))})))
∧
(∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = Method•Class•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Method•var)))
∧
(Method•External•var = Method•Class•External•var)
∧
(∀ Link•var ∈ Method•ElementLinks•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = Method•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Link•var)))

Pred14 :=
(Method•ExternalOrigin•var ∈ MotherSpecification•var)
∧
(Method•ExternalOrigin•var = Method•Class•ExternalOrigin•var)

Pred15 :=
(∀ Method1•var ∈ ConceptRepository•var . (((Method•var ≠ Method1•var) ∧ (Method•DocumentName•var = Method1•DocumentName•var) ∧ (Method•Parameters•var = Method1•Parameters•var) ∧ (Method1•Class•var ∈ superclassesOf(Method•Class•var))) ⇒
(Method•ReturnedType•var = Method1•ReturnedType•var)))
∧
(∃ SpecificationElementPair•var ∈ ReferenceTable•var . ((SpecificationElementPair•SpecificationElement1•var = Method•ReturnedType•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Method•var)))

257	**Method**	'startMethod' . DocumentName . ElementLinks . External . Class . ReturnedType . ReturnIsSet . Parameters . Classification . Private . MetaMethod . Category . 'endMethod'

Method•pred :=
Pred13 ∧ Pred15 ∧ DocumentName•pred ∧ ElementLinks•pred ∧ External•pred ∧ Class•pred ∧ ReturnedType•pred ∧ ReturnIsSet•pred ∧ Parameters•pred ∧ Classification•pred ∧ Private•pred ∧ MetaMethod•pred ∧ Category•pred

258	**Method**	'startMethod' . DocumentName . ElementLinks . External . ExternalOrigin . Class . Parameters . Classification . Private . MetaMethod . Category . 'endMethod'

Method•pred :=
Pred13 ∧ Pred14 ∧ DocumentName•pred ∧ ElementLinks•pred ∧ External•pred ∧ ExternalOrigin•pred ∧ Class•pred ∧ Parameters•pred ∧ Classification•pred ∧ Private•pred ∧ MetaMethod•pred ∧ Category•pred

259	**Method**	'startMethod' . DocumentName . ElementLinks . External . Class . Parameters . Classification . Private . MetaMethod . Category . 'endMethod'

Method•pred :=
Pred13 ∧ DocumentName•pred ∧ ElementLinks•pred ∧ External•pred ∧ Class•pred ∧ Parameters•pred ∧ Classification•pred ∧ Private•pred ∧ MetaMethod•pred ∧ Category•pred

260	ReturnedType	Type	ReturnedType•pred := Type•pred
261	ReturnIsSet	TRUE	ReturnIsSet•pred := TRUE
262		FALSE	ReturnIsSet•pred := TRUE
263	Parameters	Integer (terminal)	Parameters•pred := TRUE
264	Classification	'base'	Classification•pred := TRUE
265		'template'	Classification•pred := TRUE
266		'abstract'	Classification•pred := TRUE
267	Private	TRUE	Private•pred := TRUE
268		FALSE	Private•pred := TRUE
269	MetaMethod	TRUE	MetaMethod•pred := TRUE
270		FALSE	MetaMethod•pred := TRUE
271	**MethodParameter**	'startMethodParameter' . DocumentName . ElementLinks . External . ExternalOrigin . Type . ObjectIsSet . Method . 'endMethodParameter'	

MethodParameter•pred :=
Pred16 ∧ Pred17 ∧ DocumentName•pred ∧ ElementLinks•pred ∧ External•pred ∧ ExternalOrigin•pred ∧Type•pred ∧ ObjectIsSet•pred ∧ Method•pred

Pred16 :=
(∀ MethodParameter1•var ∈ ConceptRepository•var . ((MethodParameter•var ≠ MethodParameter1•var) ⇒ ((MethodParameter•DocumentName•var ≠ MethodParameter1•DocumentName•var) ∨ (MethodParameter•Method•var ≠ MethodParameter1•Method•var))))
∧
(∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = MethodParameter•Method•var) ∧ (SpecificationElementPair•SpecificationElement2•var = MethodParameter•var)))
∧
(∃ SpecificationElementPair•var ∈ ReferenceTable•var . ((SpecificationElementPair•SpecificationElement1•var = MethodParameter•Type•var) ∧ (SpecificationElementPair•SpecificationElement2•var = MethodParameter•var)))
∧
(MethodParameter•External•var = MethodParameter•Method•External•var)
∧
(∀ Link•var ∈ MethodParameter•ElementLinks•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = MethodParameter•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Link•var)))

Pred16 :=
(MethodParameter•ExternalOrigin•var ∈ MotherSpecification•var)
∧
(MethodParameter•ExternalOrigin•var = MethodParameter•Method•ExternalOrigin•var)

272	**MethodParameter**	'startMethodParameter' . DocumentName . ElementLinks . External . Type . ObjectIsSet . Method . 'endMethodParameter'

MethodParameter•pred :=
Pred16 ∧ DocumentName•pred ∧ ElementLinks•pred ∧ External•pred ∧ Type•pred ∧ ObjectIsSet•pred ∧ Method•pred

273	**MethodTemporaryVariable**	'startMethodTemporaryVariable' . DocumentName . ElementLinks . Type . ObjectIsSet . Method . 'endMethodTemporaryVariable'

MethodTemporaryVariable•pred :=
(∀ MethodTemporaryVariable1•var ∈ ConceptRepository•var . ((MethodTemporaryVariable•var ≠ MethodTemporaryVariable1•var) ⇒ ((MethodTemporaryVariable•DocumentName•var ≠ MethodTemporaryVariable1•DocumentName•var) ∨ (MethodTemporaryVariable•Method•var ≠ MethodTemporaryVariable1•Method•var))))
∧
(∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = MethodTemporaryVariable•Method•var) ∧ (SpecificationElementPair•SpecificationElement2•var = MethodTemporaryVariable•var)))
∧
(∃ SpecificationElementPair•var ∈ ReferenceTable•var . ((SpecificationElementPair•SpecificationElement1•var = MethodTemporaryVariable•Type•var) ∧ (SpecificationElementPair•SpecificationElement2•var = MethodTemporaryVariable•var)))
∧

(∀ Link•var ∈ MethodTemporaryVariable•ElementLinks•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = MethodTemporaryVariable•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Link•var)))
∧
DocumentName•pred ∧ ElementLinks•pred ∧ Type•pred ∧ ObjectIsSet•pred ∧ Method•pred

274	**Type**	'startType' . DocumentName . Class .'endType'	Type•pred := (∀ Type1•var ∈ ConceptRepository•var . ((Type•var ≠ Type1•var) ⇒ (Type•DocumentName•var ≠ Type1•DocumentName•var))) ∧ (∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = Type•Class•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Type•var))) ∧ (Type•DocumentName•var = Type•Class•DocumentName•var) ∧ DocumentName•pred ∧ Class•pred
275		'startType' . DocumentName . 'endType'	Type•pred := (∀ Type1•var ∈ ConceptRepository•var . ((Type•var ≠ Type1•var) ⇒ (Type•DocumentName•var ≠ Type1•DocumentName•var))) ∧ DocumentName•pred ∧ Class•pred
276	**Object**	'startObject' . DocumentName . ElementLinks . ConceptualModelReferers . Class . 'endObject'	Object•pred := (∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = Object•Class•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Object•var))) ∧ (∀ Link•var ∈ ObjectVariable•ElementLinks•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = Object•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Link•var))) ∧ DocumentName•pred ∧ ElementLinks•pred ∧ ConceptualModelReferers•pred ∧ Class•pred
277	**Message**	'startMessage' . ElementLinks . ConceptualModelReferers . OriginElement . OriginMessage . TargetElement . TargetMethod . MessageOrder . MessageWrapper . 'endMessage'	

Message•pred :=
Pred18 ∧ Pred19 ∧ Pred20 ∧ ElementLinks•pred ∧ ConceptualModelReferers•pred ∧ OriginElement•pred ∧ OriginMessage•pred ∧ TargetElement•pred ∧ TargetMethod•pred ∧ MessageOrder•pred ∧ MessageWrapper•pred

Pred18 :=
(∃ Object•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = Object•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Message•var) ∧ (Message•TargetElement•var = Object•var)))
∧
(∃ Method•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ ReferenceTable•var . ((SpecificationElementPair•SpecificationElement1•var = Method•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Message•var) ∧ (Message•TargetMethod•var = Method•var)))
∧
(∀ Link•var ∈ Message•ElementLinks•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = Message•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Link•var)))
∧
((Message•TargetMethod•Class•var = Message•TargetElement•Class•var) ∨ (Message•TargetMethod•Class•var ∈ superclassesOf(Message•TargetElement•Class•var)))

Pred19 :=
(∃ Object•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = Object•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Message•var) ∧ (Message•OriginElement•var = Object•var)))
∧
(∃ Message1•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((Message1•var ≠ Message•var) ∧ (SpecificationElementPair•SpecificationElement1•var = Message1•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Message•var) ∧ (Message•OriginMessage•var = Message1•var)))
∧
(Message•MessageOrder•var ≥ 1)
∧
((Message•OriginMessage•TargetMethod•Class•var = Message•OriginElement•Class•var) ∨ (Message•OriginMessage•TargetMethod•Class•var ∈ superclassesOf(Message•OriginElement•Class•var)))

Pred20 :=
(∃ MessageWrapper1•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ ReferenceTable•var . ((SpecificationElementPair•SpecificationElement1•var = MessageWrapper1•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Message•var) ∧ (Message•MessageWrapper•var = MessageWrapper1•var)))

278	**Message**	'startMessage' . ElementLinks . ConceptualModelReferers . OriginElement . OriginMessage . TargetElement . TargetMethod . MessageOrder . 'endMessage'

Message•pred :=
Pred18 ∧ Pred19 ∧ ElementLinks•pred ∧ ConceptualModelReferers•pred ∧ OriginElement•pred ∧ OriginMessage•pred ∧ TargetElement•pred ∧ TargetMethod•pred ∧ MessageOrder•pred

279	**Message**	'startMessage' . ElementLinks . ConceptualModelReferers . OriginElement . TargetElement . TargetMethod . MessageOrder . MessageWrapper . 'endMessage'

Message•pred :=
Pred18 ∧ Pred20 ∧ Pred21 ∧ ElementLinks•pred ∧ ConceptualModelReferers•pred ∧ OriginElement•pred ∧ TargetElement•pred ∧ TargetMethod•pred ∧ MessageOrder•pred ∧ MessageWrapper•pred

Pred21 :=
((∃ UseCaseActor•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = UseCaseActor•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Message•var) ∧ (Message•OriginElement•var = UseCaseActor•var) ∧ (Message•MessageOrder•var ≥ 1)))
∨ (∃ ExternalReference•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = ExternalReference•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Message•var) ∧ (Message•OriginElement•var = ExternalReference•var) ∧ (Message•MessageOrder •var = 1))))

280	**Message**	'startMessage' . ElementLinks . ConceptualModelReferers . OriginElement . TargetElement . TargetMethod . MessageOrder . 'endMessage'	

Message•pred :=
Pred18 ∧ Pred21 ∧ ElementLinks•pred ∧ ConceptualModelReferers•pred ∧ OriginElement•pred ∧ TargetElement•pred ∧ TargetMethod•pred ∧ MessageOrder•pred

281	OriginElement	Object	OriginElement•pred := Object•pred
282		ExternalReference	OriginElement•pred := ExternalReference•pred
283		UseCaseActor	OriginElement•pred := UseCaseActor•pred
284	OriginMessage	Message	OriginMessage•pred := Message•pred
285	TargetElement	Object	TargetElement•pred := Object•pred
286	TargetMethod	Method	TargetMethod•pred := Method•pred
287	MessageOrder	Integer	MessageOrder•pred := TRUE
288	**ExternalReference**	'startExternalReference' . ElementLinks . ConceptualModelReferers . 'endExternalReference'	ExternalReference•pred := (∀ Link•var ∈ ExternalReference•ElementLinks•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = ExternalReference• var) ∧ (SpecificationElementPair•SpecificationElement2•var = Link•var))) ∧ ElementLinks•pred ∧ ConceptualModelReferers•pred

289	**MessageWrapper**	'startMessageWrapper' . ElementLinks . WrapperType . ConstraintExpression . 'endMessageWrapper'	MessageWrapper•pred := (∃ Message•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair• SpecificationElement1•var = Message•var) ∧ (SpecificationElementPair• SpecificationElement2•var = MessageWrapper•var))) ∧ (∀ Link•var ∈ MessageWrapper•ElementLinks•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = MessageWrapper •var) ∧ (SpecificationElementPair•SpecificationElement2•var = Link•var))) ∧ ElementLinks•pred ∧ WrapperType•pred ∧ ConstraintExpression•pred
290	WrapperType	'if'	WrapperType•pred := TRUE
291		'while'	WrapperType•pred := TRUE
292		'repeat'	WrapperType•pred := TRUE
293		'nTimes'	WrapperType•pred := TRUE
294	ConstraintExpression	Predicate	ConstraintExpression•pred := Predicate•pred
295		Function	ConstraintExpression•pred := Function•pred
296	Predicate[3]	String	Predicate•pred := TRUE
297	Function	String	Function•pred := TRUE
298	**UseCase**	'startUseCase' . DocumentName . ElementLinks . Initial . ConceptualModelReferers . 'endUseCase'	UseCase•pred := (∀ UseCase1•var ∈ ConceptRepository•var . ((UseCase•var ≠ UseCase1• var) ⇒ (UseCase•DocumentName•var ≠ UseCase1•DocumentName•var))) ∧ (∀ Link•var ∈ UseCase•ElementLinks•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair• SpecificationElement1•var = UseCase•var) ∧ (SpecificationElementPair• SpecificationElement2•var = Link•var))) ∧ DocumentName•pred ∧ ElementLinks•pred ∧ Initial•pred ∧ ConceptualModelReferers•pred

[3] Predicados e funções são especificados em uma especificação como strings, não sendo portanto avaliados sintática ou semanticamente. No processo de geração de código o valor definido é diretamente copiado no código - assim, predicados e funções precisam ser escritos segundo a sintaxe da linguagem alvo e só são avaliados pelo compilador da linguagem. Esta limitação deve ser removida em uma futura manutenção do ambiente, em que se possibilitará que predicados e funções sejam definidos como combinações de operadores e operandos, através de um editor dirigido por sintaxe.

299	Initial	TRUE	Initial•pred := TRUE
300		FALSE	Initial•pred := TRUE
301	**UseCaseActor**	'startUseCaseActor' . DocumentName . ElementLinks . ConceptualModelReferers . ClassSet . 'endUseCaseActor'	

UseCaseActor•pred :=
(∀ UseCaseActor1•var ∈ ConceptRepository•var . ((UseCaseActor•var ≠ UseCaseActor1•var) ⇒ (UseCaseActor•DocumentName•var ≠ UseCaseActor1•DocumentName•var)))
∧
(∀ Class•var ∈ UseCaseActor•ClassSet•var . ∃ SpecificationElementPair•var ∈ ReferenceTable•var . ((SpecificationElementPair•SpecificationElement1•var = Class•var) ∧ (SpecificationElementPair•SpecificationElement2•var = UseCaseActor•var)))
∧
(∀ Link•var ∈ UseCaseActor•ElementLinks•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = UseCaseActor•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Link•var)))
∧
DocumentName•pred ∧ ElementLinks•pred
∧ ConceptualModelReferers•pred ∧ ClassSet•pred

302	**UseCaseRelationship**	'startUseCaseRelationship' . ElementLinks . ConceptualModelReferers . UseCase . UseCaseActor . 'endUseCaseRelationship'

UseCaseRelationship•pred :=
(∀ UseCaseRelationship1•var ∈ ConceptRepository•var . ((UseCaseRelationship•var ≠ UseCaseRelationship1•var) ⇒ ((UseCaseRelationship•UseCase•var ≠ UseCaseRelationship1•UseCase•var) ∨ (UseCaseRelationship•UseCaseActor•var ≠ UseCaseRelationship1•UseCaseActor•var))))
∧
(∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = UseCaseRelationship•UseCase•var) ∧ (SpecificationElementPair•SpecificationElement2•var = UseCaseRelationship•var)))
∧
(∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = UseCaseRelationship•UseCaseActor•var) ∧ (SpecificationElementPair•SpecificationElement2•var = UseCaseRelationship•var)))
∧
(∀ Link•var ∈ UseCaseRelationship•ElementLinks•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = UseCaseRelationship•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Link•var)))
∧
ElementLinks•pred ∧ ConceptualModelReferers•pred ∧ UseCase•pred ∧ UseCaseActor•pred

303	**State**	'startState' . DocumentName . ElementLinks . ConceptualModelReferers . Class . AttributeAssignmentSet . LifeTimePosition . Superstate . 'endState'

State•pred :=
Pred22 ∧ Pred23 ∧ DocumentName•pred ∧ ElementLinks•pred ∧ ConceptualModelReferers•pred ∧ Class•pred ∧ AttributeAssignmentSet•pred ∧ LifeTimePosition•pred ∧ Superstate•pred

Pred22 :=
(∀ State1•var ∈ ConceptRepository•var . ((State•var ≠ State1•var) ⇒ ((State•DocumentName•var ≠ State1•DocumentName•var) ∨ (State•Class•var ≠ State1•Class•var))))

∧
(∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = State•Class•var) ∧ (SpecificationElementPair•SpecificationElement2•var = State•var)))
∧
(∀ AssignmentStructure•var ∈ State•AttributeAssignmentSet•var . ∃ SpecificationElementPair•var ∈ ReferenceTable•var . ((SpecificationElementPair•SpecificationElement1•var = AssignmentStructure•Attribute•var) ∧ (SpecificationElementPair•SpecificationElement2•var = State•var)))
∧
(∀ Link•var ∈ State•ElementLinks•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = State•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Link•var)))
∧
(∃ AssignmentStructure1•var ∈ State•AttributeAssignmentSet•var . ∀ AssignmentStructure2•var ∈ State•AttributeAssignmentSet•var . ((AssignmentStructure1•var ≠ AssignmentStructure2•var) ⇒ (AssignmentStructure1•Attribute•var ≠ AssignmentStructure2•Attribute•var)))
∧
(∀ State1•var ∈ ConceptRepository•var . (((State•var ≠ State1•var) ∧ (State•var ∈ superstatesOf(State1•var)[4])) ⇒
(∀ AssignmentStructure•var ∈ State•AttributeAssignmentSet•var . ∃ AssignmentStructure1•var ∈ State1•AttributeAssignmentSet•var . ((AssignmentStructure•Attribute•var = AssignmentStructure1•Attribute•var) ∧ (AssignmentStructure•ValueSet•var ⊆ AssignmentStructure1•ValueSet•var) ∧ (AssignmentStructure1•ValueSet•var ⊆ AssignmentStructure•ValueSet•var) ∧ (AssignmentStructure•ValueRange•var = AssignmentStructure1•ValueRange•var) ∧ (AssignmentStructure•ValueExclusion•var = AssignmentStructure1•ValueExclusion•var)))))

Pred23 :=
(∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = State•Superstate•var) ∧ (SpecificationElementPair•SpecificationElement2•var = State•var)))
∧
(∀ State1•var ∈ ConceptRepository•var . (((State•var ≠ State1•var) ∧ (State•Class•var = State1•Class•var) ∧ (State•Superstate•var = State1•Superstate•var) ∧ (∀ AssignmentStructure•var ∈ State•AttributeAssignmentSet•var . ∃ AssignmentStructure1•var ∈ State1•AttributeAssignmentSet•var . (AssignmentStructure•Attribute•var = AssignmentStructure1•Attribute•var))) ⇒
(∃ AssignmentStructure•var ∈ State•AttributeAssignmentSet•var . ∃ AssignmentStructure1•var ∈ State1•AttributeAssignmentSet•var . ((AssignmentStructure•Attribute•var = AssignmentStructure1•Attribute•var) ∧ ((AssignmentStructure•ValueSet•var ⊄ AssignmentStructure1•ValueSet•var) ∨ (AssignmentStructure1•ValueSet•var ⊄ AssignmentStructure•ValueSet•var) ∨ (AssignmentStructure•ValueRange•var ≠ AssignmentStructure1•ValueRange•var) ∨ (AssignmentStructure•ValueExclusion•var ≠ AssignmentStructure1•ValueExclusion•var)))))

304	**State**	'startState' . DocumentName . ElementLinks . ConceptualModelReferers . Class . AttributeAssignmentSet . LifeTimePosition . 'endState'

State•pred :=
Pred22 ∧ Pred24 ∧ DocumentName•pred ∧ ElementLinks•pred ∧ ConceptualModelReferers•pred ∧ Class•pred ∧ AttributeAssignmentSet•pred ∧ LifeTimePosition•pred

Pred24 :=

4 "superstatesOf(State•var)" é uma função que retorna o conjunto dos superestados do estado que serve como argumento. Um estado conhece o seu superestado. O conjunto de seus superestados é construído buscando seu superestado, o superestado de seu superestado, e assim sucessivamente.

(∀ State1•var ∈ ConceptRepository•var . (((State•var ≠ State1•var) ∧ (State•Class•var = State1•Class•var) ∧ (∀ ConceptualModel•var ∈ State•ConceptualModelReferers•var . ∃ ConceptualModel1•var ∈ State1•ConceptualModelReferers•var . (ConceptualModel•var = ConceptualModel1•var)) ∧ (∀ AssignmentStructure•var ∈ State•AttributeAssignmentSet •var . ∃ AssignmentStructure1•var ∈ State1•AttributeAssignmentSet•var . (AssignmentStructure•Attribute•var = AssignmentStructure1•Attribute•var))) ⇒
(∃ AssignmentStructure•var ∈ State•AttributeAssignmentSet•var . ∃ AssignmentStructure1•var ∈ State1•AttributeAssignmentSet•var . ((AssignmentStructure•Attribute•var = AssignmentStructure1•Attribute•var) ∧ ((AssignmentStructure•ValueSet•var ⊄ AssignmentStructure1•ValueSet•var) ∨ (AssignmentStructure1•ValueSet•var ⊄ AssignmentStructure•ValueSet•var) ∨ (AssignmentStructure•ValueRange•var ≠ AssignmentStructure1•ValueRange•var) ∨ (AssignmentStructure•ValueExclusion•var ≠ AssignmentStructure1•ValueExclusion•var)))))

305	AttributeAssignmentSet	AssignmentStructure	AttributeAssignmentSet•pred := AssignmentStructure•pred
306		AssignmentStructureSet	AttributeAssignmentSet•pred := AssignmentStructureSet•pred
307	AssignmentStructureSet	AssignmentStructure	AssignmentStructureSet•pred := AssignmentStructure•pred
308		AssignmentStructure . AssignmentStructureSet'	AssignmentStructureSet•pred := AssignmentStructure•pred ∧ AssignmentStructureSet'•pred
309	AssignmentStructure	Attribute . ValueSet . ValueRange . ValueExclusion	AssignmentStructure•pred := Attribute•pred ∧ ValueSet•pred ∧ ValueRange•pred ∧ ValueExclusion•pred
310	ValueSet	Value	ValueSet•pred := Value•pred
311		Value . ValueSet'	ValueSet•pred := Value•pred ∧ ValueSet'•pred
312	Value[5]	String	Value•pred := TRUE
313		Integer	Value•pred := TRUE
314		Real	Value•pred := TRUE
315		TRUE	Value•pred := TRUE
316		FALSE	Value•pred := TRUE
317	ValueRange	TRUE	ValueRange•pred := TRUE
318		FALSE	ValueRange•pred := TRUE
319	ValueExclusion	TRUE	ValueExclusion•pred := TRUE
320		FALSE	ValueExclusion•pred := TRUE
321	Superstate	State	Superstate•pred := State•pred
322	LifeTimePosition	'initial'	LifeTimePosition•pred := TRUE
323		'intermediate'	LifeTimePosition•pred := TRUE
324		'final'	LifeTimePosition•pred := TRUE
325	**Transition**	'startTransition' . ElementLinks . ConceptualModelReferers . TransitionOrigin . TransitionTarget . TransitionAction . 'endTransition'	

Transition•pred :=
(∀ Transition1•var ∈ ConceptRepository•var . ((Transition•var ≠ Transition1•var) ⇒ ((Transition•TransitionOrigin•var ≠ Transition1•TransitionOrigin•var) ∨ (Transition• TransitionTarget•var ≠ Transition1•TransitionTarget•var) ∨ (Transition•TransitionAction•var ≠ Transition1•TransitionAction•var))))
∧

5 Atualmente apenas tipos standard de linguagens de programação podem ser associados como valores que definem estados.

(∃ State•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = State•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Transition•var) ∧ (Transition•TransitionOrigin•var = State•var)))
∧
(∃ State•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = State•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Transition•var) ∧ (Transition•TransitionTarget•var = State•var)))
∧
(∃ Method•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ ReferenceTable•var . ((SpecificationElementPair•SpecificationElement1•var = Method•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Transition•var) ∧ (Transition•TransitionAction•var = Method•var) ∧ ((Transition•TransitionAction•Class•var = Transition•TransitionTarget•Class•var) ∨ (Transition•TransitionAction•Class•var ∈ superclassesOf(Transition•TransitionTarget•Class•var)))))
∧
(Transition•TransitionOrigin•Class•var = Transition•TransitionTarget•Class•var)
∧
(∀ Link•var ∈ Transition•ElementLinks•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = Transition•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Link•var)))
∧
ElementLinks•pred ∧ ConceptualModelReferers•pred ∧ TransitionOrigin•pred ∧ TransitionTarget•pred ∧ TransitionAction•pred

326	**Transition**	'startTransition' . ElementLinks . ConceptualModelReferers . TransitionOrigin . TransitionTarget . 'endTransition'

Transition•pred :=
(∀ Transition1•var ∈ ConceptRepository•var . ((Transition•var ≠ Transition1•var) ⇒ ((Transition•TransitionOrigin•var ≠ Transition1•TransitionOrigin•var) ∨ (Transition•TransitionTarget•var ≠ Transition1•TransitionTarget•var))))
∧
(∃ UseCase•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = UseCase•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Transition•var) ∧ (Transition•TransitionOrigin•var = UseCase•var)))
∧
(∃ UseCase•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = UseCase•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Transition•var) ∧ (Transition•TransitionTarget•var = UseCase•var)))
∧
(∀ Link•var ∈ Transition•ElementLinks•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = Transition•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Link•var)))
∧
ElementLinks•pred ∧ ConceptualModelReferers•pred ∧ TransitionOrigin•pred ∧ TransitionTarget•pred ∧ TransitionAction•pred

327	TransitionOrigin	State	TransitionOrigin•pred := State•pred
328		UseCase	TransitionOrigin•pred := UseCase•pred
329	TransitionTarget	State	TransitionTarget•pred := State•pred
330		UseCase	TransitionTarget•pred := UseCase•pred
331	TransitionAction	Method	TransitionAction•pred := Method•pred

332	**TextualStatement**	'startTextualStatement' . ElementLinks . ConceptualModelReferers . Method . TextualHeader . StatementOrder . Container . StatementAuxOrder . TextualSentence . 'endTextualStatement'	TextualStatement•pred := ((∃ CompositeStatement•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = CompositeStatement•var) ∧ (SpecificationElementPair• SpecificationElement2•var = TextualStatement•var) ∧ (TextualStatement• Container•var = CompositeStatement•var))) ∨ (∃ DoubleCompositeStatement•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = DoubleCompositeStatement•var) ∧ (SpecificationElementPair• SpecificationElement2•var = TextualStatement•var) ∧ (TextualStatement• Container•var = DoubleCompositeStatement•var)))) ∧ ElementLinks•pred ∧ ConceptualModelReferers•pred ∧ Method•pred ∧ TextualHeader•pred ∧ StatementOrder•pred ∧ Container•pred ∧ StatementAuxOrder•pred ∧ TextualSentence•pred
333		'startTextualStatement' . ElementLinks . ConceptualModelReferers . Method . TextualHeader . StatementOrder . TextualSentence . 'endTextualStatement'	TextualStatement•pred := ElementLinks•pred ∧ ConceptualModelReferers•pred ∧ Method•pred ∧ TextualHeader•pred ∧ StatementOrder•pred ∧ TextualSentence•pred
334	TextualHeader	'comment'	TextualHeader•pred := TRUE
335		'task'	TextualHeader•pred := TRUE
336		'external'	TextualHeader•pred := TRUE
337	StatementOrder	Integer	StatementOrder•pred := TRUE
338	Container	CompositeStatement	Container•pred := CompositeStatement•pred
339		DoubleCompositeStatement	Container•pred := DoubleCompositeStatement•pred
340	StatementAuxOrder	Integer	StatementAuxOrder•pred := TRUE
341	TextualSentence	String	TextualSentence•pred := TRUE
342	**PointerStatement**	'startPointerStatement' . ElementLinks . ConceptualModelReferers . Method . PointerHeader . StatementOrder . Container . StatementAuxOrder . ReferedObject . 'endPointerStatement'	

PointerStatement•pred :=
Pred25 ∧ Pred26 ∧ ElementLinks•pred ∧ ConceptualModelReferers•pred ∧ Method•pred ∧ TextualHeader•pred ∧ StatementOrder•pred ∧ Container•pred ∧ StatementAuxOrder•pred ∧ ReferedObject•pred

Pred25 :=
((∃ MethodParameter•var ∈ ConceptRepository•var . (PointerStatement•ReferedObject•var = MethodParameter•var)) ⇒

(∃ SpecificationElementPair•var ∈ ReferenceTable•var . ((SpecificationElementPair•SpecificationElement1•var = MethodParameter•var) ∧ (SpecificationElementPair•SpecificationElement2•var = PointerStatement•var))) ∧ (∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = PointerStatement•Method•var) ∧ (SpecificationElementPair•SpecificationElement2•var = MethodParameter•var))))
∧
((∃ MethodTemporaryVariable•var ∈ ConceptRepository•var . (PointerStatement•ReferedObject•var = MethodTemporaryVariable•var)) ⇒
((∃ SpecificationElementPair•var ∈ ReferenceTable•var . ((SpecificationElementPair•SpecificationElement1•var = MethodTemporaryVariable•var) ∧ (SpecificationElementPair•SpecificationElement2•var = PointerStatement•var))) ∧ (∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = PointerStatement•Method•var) ∧ (SpecificationElementPair•SpecificationElement2•var = MethodTemporaryVariable•var))))
∧
((∃ Attribute•var ∈ ConceptRepository•var . (PointerStatement•ReferedObject•var = Attribute•var)) ⇒
(∃ SpecificationElementPair•var ∈ ReferenceTable•var . ((SpecificationElementPair•SpecificationElement1•var = Attribute•var) ∧ (SpecificationElementPair•SpecificationElement2•var = PointerStatement•var))) ∧ (∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = PointerStatement•Method•Class•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Attribute•var))))
∧
((∃ Attribute•var ∈ ConceptRepository•var . (PointerStatement•ReferedObject•var = Attribute•var)) ⇒ (PointerStatement•Method•ReturnedType•var = Attribute•Type•var))
∧
((∃ MethodTemporaryVariable•var ∈ ConceptRepository•var . (PointerStatement•ReferedObject•var = MethodTemporaryVariable•var)) ⇒ (PointerStatement•Method•ReturnedType•var = MethodTemporaryVariable•Type•var))
∧
((∃ MethodParameter•var ∈ ConceptRepository•var . (PointerStatement•ReferedObject•var = MethodParameter•var)) ⇒ (PointerStatement•Method•ReturnedType•var = MethodParameter•Type•var))
∧
((∃ Class•var ∈ ConceptRepository•var . (PointerStatement•ReferedObject•var = Class•var)) ⇒ (PointerStatement•Method•ReturnedType•Class•var = Class•var))
∧
((PointerStatement•ReferedObject•var = 'self') ⇒ (PointerStatement•Method•ReturnedType•Class•var = PointerStatement•Method•Class•var))
∧
(∀ Link•var ∈ PointerStatement•ElementLinks•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = PointerStatement•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Link•var)))

Pred26 :=
((∃ CompositeStatement•var ∈ ConceptRepository•var . (PointerStatement•Container•var = CompositeStatement•var)) ⇒ (PointerStatement•StatementOrder•var = lenthOf(PointerStatement•Container•StatementSet•var)[6]))
∧
((∃ DoubleCompositeStatement•var ∈ ConceptRepository•var . ((PointerStatement•Container•var = DoubleCompositeStatement•var) ∧ (PointerStatement•StatementAuxOrder•var = 1))) ⇒ (PointerStatement•StatementOrder•var = lenthOf(PointerStatement•Container•FirstStatementSet•var)))
∧

6 "lenthOf(Set•var)" é uma função que retorna o comprimento do conjunto do argumento, ou seja, a quantidade de elementos do conjunto. Se o argumento for um conjunto vazio, a função retorna zero.

((∃ DoubleCompositeStatement•var ∈ ConceptRepository•var . ((PointerStatement•Container•var = DoubleCompositeStatement•var) ∧ (PointerStatement•StatementAuxOrder•var = 2))) ⇒ (PointerStatement•StatementOrder•var = lenthOf(PointerStatement•Container•SecondStatementSet•var)))
∧
((∃ CompositeStatement•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = CompositeStatement•var) ∧ (SpecificationElementPair•SpecificationElement2•var = PointerStatement•var) ∧ (PointerStatement•Container•var = CompositeStatement•var)))
∨ (∃ DoubleCompositeStatement•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = DoubleCompositeStatement•var) ∧ (SpecificationElementPair•SpecificationElement2•var = PointerStatement•var) ∧ (PointerStatement•Container•var = DoubleCompositeStatement•var))))

343	**PointerStatement**	'startPointerStatement' . ElementLinks . ConceptualModelReferers . Method . PointerHeader . StatementOrder . ReferedObject . 'endPointerStatement'

PointerStatement•pred :=
Pred25 ∧ ElementLinks•pred ∧ ConceptualModelReferers•pred ∧ Method•pred ∧ TextualHeader•pred ∧ StatementOrder•pred ∧ ReferedObject•pred

344	PointerHeader	'return'	PointerHeader•pred := TRUE
345	ReferedObject	Attribute	ReferedObject•pred := Attribute•pred
346		MethodParameter	ReferedObject•pred := MethodParameter•pred
347		MethodTemporaryVariable	ReferedObject•pred := MethodTemporaryVariable•pred
348		Class	ReferedObject•pred := Class•pred
349		Value	ReferedObject•pred := Value•pred
350		'self'	ReferedObject•pred := TRUE
351	**DoublePointerStatement**	'startDoublePointerStatement' . ElementLinks . ConceptualModelReferers . Method . DoublePointerHeader . StatementOrder . Container . StatementAuxOrder . FirstReferedObject . SecondReferedObject . 'endDoublePointerStatement'	

DoublePointerStatement•pred :=
Pred27 ∧ Pred28 ∧ ElementLinks•pred ∧ ConceptualModelReferers•pred ∧ Method•pred ∧ TextualHeader•pred ∧ StatementOrder•pred ∧ Container•pred ∧ StatementAuxOrder•pred ∧ FirstReferedObject•pred ∧ SecondReferedObject•pred

Pred27 :=
((∃ MethodTemporaryVariable•var ∈ ConceptRepository•var . (DoublePointerStatement•FirstReferedObject•var = MethodTemporaryVariable•var)) ⇒
((∃ SpecificationElementPair•var ∈ ReferenceTable•var . ((SpecificationElementPair•SpecificationElement1•var = MethodTemporaryVariable•var) ∧ (SpecificationElementPair•SpecificationElement2•var = DoublePointerStatement•var)))
∧ (∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = DoublePointerStatement•Method•var) ∧ (SpecificationElementPair•SpecificationElement2•var = MethodTemporaryVariable•var)))))
∧
((∃ Attribute•var ∈ ConceptRepository•var . (DoublePointerStatement•FirstReferedObject•var = Attribute•var)) ⇒
((∃ SpecificationElementPair•var ∈ ReferenceTable•var . ((SpecificationElementPair•SpecificationElement1•var = Attribute•var) ∧ (SpecificationElementPair•SpecificationElement2•var = DoublePointerStatement•var)))
∧ (∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = DoublePointerStatement•Method•Class•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Attribute•var)))))

∧
((∃ MethodParameter•var ∈ ConceptRepository•var . (DoublePointerStatement•SecondReferedObject•var = MethodParameter•var)) ⇒
((∃ SpecificationElementPair•var ∈ ReferenceTable•var . ((SpecificationElementPair•SpecificationElement1•var = MethodParameter•var) ∧ (SpecificationElementPair•
SpecificationElement2•var = DoublePointerStatement•var)))
∧ (∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = DoublePointerStatement•Method•var) ∧
(SpecificationElementPair•SpecificationElement2•var = MethodParameter•var)))))
∧
((∃ MethodTemporaryVariable•var ∈ ConceptRepository•var . (DoublePointerStatement•SecondReferedObject•var = MethodTemporaryVariable•var)) ⇒
((∃ SpecificationElementPair•var ∈ ReferenceTable•var . ((SpecificationElementPair•SpecificationElement1•var = MethodTemporaryVariable•var) ∧ (SpecificationElementPair•
SpecificationElement2•var = DoublePointerStatement•var)))
∧ (∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = DoublePointerStatement•Method•var) ∧
(SpecificationElementPair•SpecificationElement2•var = MethodTemporaryVariable•var)))))
∧
((∃ Attribute•var ∈ ConceptRepository•var . (DoublePointerStatement•SecondReferedObject•var = Attribute•var)) ⇒
((∃ SpecificationElementPair•var ∈ ReferenceTable•var . ((SpecificationElementPair•SpecificationElement1•var = Attribute•var) ∧ (SpecificationElementPair•SpecificationElement2•var
= DoublePointerStatement•var)))
∧ (∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = DoublePointerStatement•Method•Class•var) ∧
(SpecificationElementPair•SpecificationElement2•var = Attribute•var)))))
∧
((∃ Class•var ∈ ConceptRepository•var . (DoublePointerStatement•SecondReferedObject•var = Class•var)) ⇒
(∃ SpecificationElementPair•var ∈ ReferenceTable•var . ((SpecificationElementPair•SpecificationElement1•var = Class•var) ∧ (SpecificationElementPair•SpecificationElement2•var =
DoublePointerStatement•var))))
∧
((∃ Attribute1•var ∈ ConceptRepository•var . ∃ Attribute2•var ∈ ConceptRepository•var . ((DoublePointerStatement•FirstReferedObject•var = Attribute1•var) ∧
(DoublePointerStatement•SecondReferedObject•var = Attribute2•var))) ⇒
(((Attribute1•Type•var = Attribute2•Type•var) ∨ (Attribute1•Type•Class•var ∈ superclassesOf(Attribute2•Type•Class•var))) ∧ (Attribute1•ObjectIsSet•var = Attribute2•
ObjectIsSet•var)))
∧
((∃ Attribute•var ∈ ConceptRepository•var . ∃ MethodParameter•var ∈ ConceptRepository•var . ((DoublePointerStatement•FirstReferedObject•var = Attribute•var) ∧
(DoublePointerStatement•SecondReferedObject•var = MethodParameter•var))) ⇒
(((Attribute•Type•var = MethodParameter•Type•var) ∨ (Attribute•Type•Class•var ∈ superclassesOf(MethodParameter•Type•Class•var))) ∧ (Attribute•ObjectIsSet•var =
MethodParameter•ObjectIsSet•var)))
∧
((∃ Attribute•var ∈ ConceptRepository•var . ∃ MethodTemporaryVariable•var ∈ ConceptRepository•var . ((DoublePointerStatement•FirstReferedObject•var = Attribute•var) ∧
(DoublePointerStatement•SecondReferedObject•var = MethodTemporaryVariable•var))) ⇒
(((Attribute•Type•var = MethodTemporaryVariable•Type•var) ∨ (Attribute•Type•Class•var ∈ superclassesOf(MethodTemporaryVariable•Type•Class•var))) ∧ (Attribute•
ObjectIsSet•var = MethodTemporaryVariable•ObjectIsSet•var)))
∧
((∃ Attribute•var ∈ ConceptRepository•var . ∃ Class•var ∈ ConceptRepository•var . ((DoublePointerStatement•FirstReferedObject•var = Attribute•var) ∧ (DoublePointerStatement•
SecondReferedObject•var = Class•var))) ⇒
(((Attribute•Type•Class•var = Class•var) ∨ (Attribute•Type•Class•var ∈ superclassesOf(Class•var))) ∧ (Attribute•ObjectIsSet•var = FALSE)))

∧
((∃ MethodTemporaryVariable•var ∈ ConceptRepository•var . ∃ Attribute•var ∈ ConceptRepository•var . ((DoublePointerStatement•FirstReferedObject•var = MethodTemporaryVariable•var) ∧ (DoublePointerStatement•SecondReferedObject•var = Attribute•var))) ⇒
(((MethodTemporaryVariable•Type•var = Attribute•Type•var) ∨ (MethodTemporaryVariable•Type•Class•var ∈ superclassesOf(Attribute•Type•Class•var))) ∧ (MethodTemporaryVariable•ObjectIsSet•var = Attribute•ObjectIsSet•var)))
∧
((∃ MethodTemporaryVariable•var ∈ ConceptRepository•var . ∃ MethodParameter•var ∈ ConceptRepository•var . ((DoublePointerStatement•FirstReferedObject•var = MethodTemporaryVariable•var) ∧ (DoublePointerStatement•SecondReferedObject•var = MethodParameter•var))) ⇒
(((MethodTemporaryVariable•Type•var = MethodParameter•Type•var) ∨ (MethodTemporaryVariable•Type•Class•var ∈ superclassesOf(MethodParameter•Type•Class•var))) ∧ (MethodTemporaryVariable•ObjectIsSet•var = MethodParameter•ObjectIsSet•var)))
∧
((∃ MethodTemporaryVariable1•var ∈ ConceptRepository•var . ∃ MethodTemporaryVariable2•var ∈ ConceptRepository•var . ((DoublePointerStatement•FirstReferedObject•var = MethodTemporaryVariable1•var) ∧ (DoublePointerStatement•SecondReferedObject•var = MethodTemporaryVariable2•var))) ⇒
(((MethodTemporaryVariable1•Type•var = MethodTemporaryVariable2•Type•var) ∨ (MethodTemporaryVariable1•Type•Class•var ∈ superclassesOf(MethodTemporaryVariable2•Type•Class•var))) ∧ (MethodTemporaryVariable1•ObjectIsSet•var = MethodTemporaryVariable2•ObjectIsSet•var)))
∧
((∃ MethodTemporaryVariable•var ∈ ConceptRepository•var . ∃ Class•var ∈ ConceptRepository•var . ((DoublePointerStatement•FirstReferedObject•var = MethodTemporaryVariable•var) ∧ (DoublePointerStatement•SecondReferedObject•var = Class•var))) ⇒
(((MethodTemporaryVariable•Type•Class•var = Class•var) ∨ (MethodTemporaryVariable•Type•Class•var ∈ superclassesOf(Class•var))) ∧ (MethodTemporaryVariable•ObjectIsSet•var = FALSE)))
∧
(∀ Link•var ∈ DoublePointerStatement•ElementLinks•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = DoublePointerStatement•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Link•var)))

Pred28 :=
((∃ CompositeStatement•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = CompositeStatement•var) ∧ (SpecificationElementPair•SpecificationElement2•var = DoublePointerStatement•var) ∧ (DoublePointerStatement•Container•var = CompositeStatement•var)))
∨ (∃ DoubleCompositeStatement•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = DoubleCompositeStatement•var) ∧ (SpecificationElementPair•SpecificationElement2•var = DoublePointerStatement•var) ∧ (DoublePointerStatement•Container•var = DoubleCompositeStatement•var))))

352	**DoublePointerStatement**	'startDoublePointerStatement' . ElementLinks . ConceptualModelReferers . Method . DoublePointerHeader . StatementOrder . FirstReferedObject . SecondReferedObject . 'endDoublePointerStatement'

DoublePointerStatement•pred :=
Pred27 ∧ ElementLinks•pred ∧ ConceptualModelReferers•pred ∧ Method•pred ∧ TextualHeader•pred ∧ StatementOrder•pred ∧ FirstReferedObject•pred ∧ SecondReferedObject•pred

353	DoublePointerHeader	'assignment'	DoublePointerHeader•pred := TRUE
354	FirstReferedObject	Attribute	FirstReferedObject•pred := Attribute•pred

355		MethodTemporaryVariable	FirstReferedObject•pred := MethodTemporaryVariable•pred
356	SecondReferedObject	Attribute	SecondReferedObject•pred := Attribute•pred
357		MethodParameter	SecondReferedObject•pred := MethodParameter•pred
358		MethodTemporaryVariable	SecondReferedObject•pred := MethodTemporaryVariable•pred
359		Class	SecondReferedObject•pred := Class•pred
360		Value	SecondReferedObject•pred := Value•pred
361	**MessageStatement**	'startMessageStatement' . ElementLinks . ConceptualModelReferers . Method . MessageHeader . StatementOrder . Container . StatementAuxOrder . FirstReferedObject . MessageTargetObject . TargetMethod . ParameterAssignmentSet . 'endMessageStatement'	

MessageStatement•pred :=
Pred29 ∧ Pred30 ∧ Pred31 ∧ ElementLinks•pred ∧ ConceptualModelReferers•pred ∧ Method•pred ∧ TextualHeader•pred ∧ StatementOrder•pred ∧ Container•pred ∧ StatementAuxOrder•pred ∧ FirstReferedObject•pred ∧ MessageTargetObject•pred

Pred29 :=
((∃ MethodParameter•var ∈ ConceptRepository•var . (MessageStatement•MessageTargetObject•var = MethodParameter•var)) ⇒
((∃ SpecificationElementPair•var ∈ ReferenceTable•var . ((SpecificationElementPair•SpecificationElement1•var = MethodParameter•var) ∧ (SpecificationElementPair•SpecificationElement2•var = MessageStatement•var)))
∧ (∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = MessageStatement•Method•var) ∧ (SpecificationElementPair•SpecificationElement2•var = MethodParameter•var)))))
∧
((∃ MethodParameter•var ∈ ConceptRepository•var . (MessageStatement•MessageTargetObject•var = MethodParameter•var)) ⇒
(MethodParameter•ObjectIsSet•var = FALSE))
∧
((∃ MethodTemporaryVariable•var ∈ ConceptRepository•var . (MessageStatement•MessageTargetObject•var = MethodTemporaryVariable•var)) ⇒
((∃ SpecificationElementPair•var ∈ ReferenceTable•var . ((SpecificationElementPair•SpecificationElement1•var = MethodTemporaryVariable•var) ∧ (SpecificationElementPair•SpecificationElement2•var = MessageStatement•var)))
∧ (∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = MessageStatement•Method•var) ∧ (SpecificationElementPair•SpecificationElement2•var = MethodTemporaryVariable•var)))))
∧
((∃ MethodTemporaryVariable•var ∈ ConceptRepository•var . (MessageStatement•MessageTargetObject•var = MethodTemporaryVariable•var)) ⇒
(MethodTemporaryVariable•ObjectIsSet•var = FALSE))
∧
((∃ Attribute•var ∈ ConceptRepository•var . (MessageStatement•MessageTargetObject•var = Attribute•var)) ⇒
((∃ SpecificationElementPair•var ∈ ReferenceTable•var . ((SpecificationElementPair•SpecificationElement1•var = Attribute•var) ∧ (SpecificationElementPair•SpecificationElement2•var = MessageStatement•var)))
∧ (∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = MessageStatement•Method•Class•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Attribute•var)))))
∧
((∃ Attribute•var ∈ ConceptRepository•var . (MessageStatement•MessageTargetObject•var = Attribute•var)) ⇒ (Attribute•ObjectIsSet•var = FALSE))

∧
((∃ Attribute•var ∈ ConceptRepository•var . (MessageStatement•MessageTargetObject•var = Attribute•var)) ⇒ (((MessageStatement•TargetMethod•Class•var = Attribute•Class•var) ∨ (Message•TargetMethod•Class•var ∈ superclassesOf(Attribute•Class•var))) ∧ (MessageStatement•TargetMethod•MetaMethod•var = FALSE)))
∧
((∃ MethodParameter•var ∈ ConceptRepository•var . (MessageStatement•MessageTargetObject•var = MethodParameter•var)) ⇒
(((MessageStatement•TargetMethod•Class•var = MethodParameter•Method•Class•var) ∨ (Message•TargetMethod•Class•var ∈ superclassesOf(MethodParameter•Method•Class•var))) ∧ (MessageStatement•TargetMethod•MetaMethod•var = FALSE)))
∧
((∃ MethodTemporaryVariable•var ∈ ConceptRepository•var . (MessageStatement•MessageTargetObject•var = MethodTemporaryVariable•var)) ⇒
(((MessageStatement•TargetMethod•Class•var = MethodTemporaryVariable•Method•Class•var) ∨ (Message•TargetMethod•Class•var ∈ superclassesOf(MethodTemporaryVariable•Method•Class•var))) ∧ (MessageStatement•TargetMethod•MetaMethod•var = FALSE)))
∧
((∃ Class•var ∈ ConceptRepository•var . (MessageStatement•MessageTargetObject•var = Class•var)) ⇒
(((MessageStatement•TargetMethod•Class•var = Class•var) ∨ (Message•TargetMethod•Class•var ∈ superclassesOf(Class•var))) ∧ (MessageStatement•TargetMethod•MetaMethod•var = TRUE)))
∧
((MessageStatement•MessageTargetObject•var = 'self') ⇒
(((MessageStatement•TargetMethod•Class•var = MessageStatementMethod•Class•var) ∨ (Message•TargetMethod•Class•var ∈ superclassesOf(MessageStatementMethod•Class•var))) ∧ (MessageStatement•TargetMethod•MetaMethod•var = MessageStatement•Method•MetaMethod•var)))
∧
(∀ ParameterStructure•var ∈ MessageStatement•ParameterAssignmentSet•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = MessageStatement•TargetMethod•var) ∧ (SpecificationElementPair•SpecificationElement2•var = ParameterStructure•MethodParameter•var)))
∧
((∃ MethodParameter•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = MessageStatement•TargetMethod•var) ∧ (SpecificationElementPair•SpecificationElement2•var = MethodParameter•var))) ⇒
(∃ ParameterStructure•var ∈ MessageStatement•ParameterAssignmentSet•var . (ParameterStructure•MethodParameter•var = MethodParameter•var)))
∧
((∃ ParameterStructure1•var ∈ MessageStatement•ParameterAssignmentSet•var . ∀ ParameterStructure2•var ∈ MessageStatement•ParameterAssignmentSet•var . (ParameterStructure1•var ≠ ParameterStructure2•var)) ⇒ (ParameterStructure1•MethodParameter•var ≠ ParameterStructure2•MethodParameter•var))
∧
((∃ Attribute•var ∈ ConceptRepository•var . ∃ ParameterStructure•var ∈ MessageStatement•ParameterAssignmentSet•var . (ParameterStructure•AssignedElement•var = Attribute•var)) ⇒
(∃ SpecificationElementPair1•var ∈ ReferenceTable•var . ∃ SpecificationElementPair2•var ∈ SustainmentTable•var . (((ParameterStructure•MethodParameter•Type•var = Attribute•Type•var) ∨ (ParameterStructure•MethodParameter•Type•Class•var ∈ superclassesOf(Attribute•Class•var))) ∧ (SpecificationElementPair1•SpecificationElement1•var = Attribute•var) ∧ (SpecificationElementPair1•SpecificationElement2•var = MessageStatement•var) ∧ (((SpecificationElementPair2•SpecificationElement1•var = MessageStatement•Method•Class•var) ∧ (SpecificationElementPair2•SpecificationElement2•var = Attribute•var)) ∨ ((SpecificationElementPair2•SpecificationElement1•var ∈ superclassesOf(MessageStatement•Method•Class•var)) ∧ (SpecificationElementPair2•SpecificationElement2•var = Attribute•var))))))
∧
((∃ MethodParameter•var ∈ ConceptRepository•var . ∃ ParameterStructure•var ∈ MessageStatement•ParameterAssignmentSet•var . (ParameterStructure•AssignedElement•var = MethodParameter•var)) ⇒

(∃ SpecificationElementPair1•var ∈ ReferenceTable•var . ∃ SpecificationElementPair2•var ∈ SustainmentTable•var . (((ParameterStructure•MethodParameter•Type•var = MethodParameter•Type•var) ∨ (ParameterStructure•MethodParameter•Type•Class•var ∈ superclassesOf(MethodParameter•Method•Class•var))) ∧ (SpecificationElementPair1• SpecificationElement1•var = MethodParameter•var) ∧ (SpecificationElementPair1•SpecificationElement2•var = MessageStatement•var) ∧ (SpecificationElementPair2• SpecificationElement1•var = MessageStatement•Method•var) ∧ (SpecificationElementPair2•SpecificationElement2•var = MethodParameter•var))))
∧
((∃ MethodTemporaryVariable•var ∈ ConceptRepository•var . ∃ ParameterStructure•var ∈ MessageStatement•ParameterAssignmentSet•var . (ParameterStructure•AssignedElement• var = MethodTemporaryVariable•var)) ⇒
(∃ SpecificationElementPair1•var ∈ ReferenceTable•var . ∃ SpecificationElementPair2•var ∈ SustainmentTable•var . (((ParameterStructure•MethodTemporaryVariable•Type•var = MethodTemporaryVariable•Type•var) ∨ (ParameterStructure•MethodTemporaryVariable•Type•Class•var ∈ superclassesOf(MethodTemporaryVariable•Method•Class•var))) ∧ (SpecificationElementPair1•SpecificationElement1•var = MethodTemporaryVariable•var) ∧ (SpecificationElementPair1•SpecificationElement2•var = MessageStatement•var) ∧ (SpecificationElementPair2•SpecificationElement1•var = MessageStatement•Method•var) ∧ (SpecificationElementPair2•SpecificationElement2•var = MethodTemporaryVariable•var))))
∧
((∃ Class•var ∈ ConceptRepository•var . ∃ ParameterStructure•var ∈ MessageStatement•ParameterAssignmentSet•var . (ParameterStructure•AssignedElement•var = Class•var)) ⇒
(∃ SpecificationElementPair•var ∈ ReferenceTable•var . (((ParameterStructure•MethodParameter•Type•Class•var = Class•var) ∨ (ParameterStructure•MethodParameter•Type•Class• var ∈ superclassesOf(Class•var))) ∧ (SpecificationElementPair•SpecificationElement1•var = Class•var) ∧ (SpecificationElementPair1•SpecificationElement2•var = MessageStatement• var))))
∧
((∃ ParameterStructure•var ∈ MessageStatement•ParameterAssignmentSet•var . (ParameterStructure•AssignedElement•var = 'self')) ⇒
((ParameterStructure•MethodParameter•Type•Class•var = MessageStatement•Method•Class•var) ∨ (ParameterStructure•MethodParameter•Type•Class•var ∈ superclassesOf(MessageStatement•Method•Class•var))))
∧
(∀ Link•var ∈ MessageStatement•ElementLinks•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = MessageStatement•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Link•var)))

Pred30 :=
((∃ MethodTemporaryVariable•var ∈ ConceptRepository•var . (MessageStatement•FirstReferedObject•var = MethodTemporaryVariable•var)) ⇒
((∃ SpecificationElementPair•var ∈ ReferenceTable•var . ((SpecificationElementPair•SpecificationElement1•var = MethodTemporaryVariable•var) ∧ (SpecificationElementPair• SpecificationElement2•var = MessageStatement•var)))
∧ (∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = MessageStatement•Method•var) ∧ (SpecificationElementPair• SpecificationElement2•var = MethodTemporaryVariable•var)))))
∧
((∃ MethodTemporaryVariable•var ∈ ConceptRepository•var . (MessageStatement•FirstReferedObject•var = MethodTemporaryVariable•var)) ⇒
(∃ Method•var ∈ ConceptRepository•var . ((MessageStatement•TargetMethod•var = Method•var) ∧ ((MessageStatement•FirstReferedObject•Type•var = MessageStatement• TargetMethod•ReturnedType•var) ∨ (MessageStatement•FirstReferedObject•Type•Class•var ∈ superclassesOf(MessageStatement•TargetMethod•ReturnedType•Class•var)))))
∧
((∃ Attribute•var ∈ ConceptRepository•var . (MessageStatement•FirstReferedObject•var = Attribute•var)) ⇒
((∃ SpecificationElementPair•var ∈ ReferenceTable•var . ((SpecificationElementPair•SpecificationElement1•var = Attribute•var) ∧ (SpecificationElementPair•SpecificationElement2•var = MessageStatement•var)))

∧ (∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = MessageStatement•Method•Class•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Attribute•var)))))
∧
((∃ Attribute•var ∈ ConceptRepository•var . (MessageStatement•FirstReferedObject•var = Attribute•var)) ⇒
(∃ Method•var ∈ ConceptRepository•var . ((MessageStatement•TargetMethod•var = Method•var) ∧ ((MessageStatement•FirstReferedObject•Type•var = MessageStatement• TargetMethod•ReturnedType•var) ∨ (MessageStatement•FirstReferedObject•Type•Class•var ∈ superclassesOf(MessageStatement•TargetMethod•ReturnedType•Class•var)))))

Pred31 :=
((∃ CompositeStatement•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = CompositeStatement•var) ∧ (SpecificationElementPair•SpecificationElement2•var = MessageStatement•var) ∧ (MessageStatement•Container•var = CompositeStatement•var)))
∨ (∃ DoubleCompositeStatement•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = DoubleCompositeStatement•var) ∧ (SpecificationElementPair•SpecificationElement2•var = MessageStatement•var) ∧ (MessageStatement•Container•var = DoubleCompositeStatement•var))))

362	**MessageStatement**	'startMessageStatement' . ElementLinks . ConceptualModelReferers . Method . MessageHeader . StatementOrder . FirstReferedObject . MessageTargetObject . TargetMethod . ParameterAssignmentSet . 'endMessageStatement'

MessageStatement•pred :=
Pred29 ∧ Pred30 ∧ ElementLinks•pred ∧ ConceptualModelReferers•pred ∧ Method•pred ∧ TextualHeader•pred ∧ StatementOrder•pred ∧ FirstReferedObject•pred ∧ MessageTargetObject•pred

363	**MessageStatement**	'startMessageStatement' . ElementLinks . ConceptualModelReferers . Method . MessageHeader . StatementOrder . Container . StatementAuxOrder . MessageTargetObject . TargetMethod . ParameterAssignmentSet . 'endMessageStatement'

MessageStatement•pred :=
Pred29 ∧ Pred31 ∧ ElementLinks•pred ∧ ConceptualModelReferers•pred ∧ Method•pred ∧ TextualHeader•pred ∧ StatementOrder•pred ∧ Container•pred ∧ StatementAuxOrder•pred ∧ MessageTargetObject•pred

364	**MessageStatement**	'startMessageStatement' . ElementLinks . ConceptualModelReferers . Method . MessageHeader . StatementOrder . MessageTargetObject . TargetMethod . ParameterAssignmentSet . 'endMessageStatement'

MessageStatement•pred :=
Pred29 ∧ ElementLinks•pred ∧ ConceptualModelReferers•pred ∧ Method•pred ∧ TextualHeader•pred ∧ StatementOrder•pred ∧ MessageTargetObject•pred

365	MessageHeader	'message'	MessageHeader•pred := TRUE
366	MessageTargetObject	Attribute	MessageTargetObject•pred := Attribute•pred
367		MethodParameter	MessageTargetObject•pred := MethodParameter•pred
368		MethodTemporaryVariable	MessageTargetObject•pred := MethodTemporaryVariable•pred
369		Class	MessageTargetObject•pred := Class•pred
370		'self'	MessageTargetObject•pred := TRUE
371	ParameterAssignmentSet	ParameterStructure	ParameterAssignmentSet•pred := ParameterStructure•pred

372		ParameterStructureSet	ParameterAssignmentSet•pred := ParameterStructureSet•pred
373	ParameterStructureSet	ParameterStructure	ParameterStructureSet•pred := ParameterStructure•pred
374		ParameterStructure . ParameterStructureSet'	ParameterStructureSet•pred := ParameterStructureSet•pred ∧ ParameterStructureSet'•pred
375	ParameterStructure	MethodParameter . AssignedElement	ParameterStructure•pred := Parameter•pred ∧ AssignedElement•pred
376	AssignedElement	Attribute	AssignedElement•pred := Attribute•pred
377		MethodParameter	AssignedElement•pred := MethodParameter•pred
378		MethodTemporaryVariable	AssignedElement•pred := MethodTemporaryVariable•pred
379		Class	AssignedElement•pred := Class•pred
380		Value	AssignedElement•pred := Value•pred
381		'self'	AssignedElement•pred := TRUE
382	**MultiplePointerStatement**	'startMultiplePointerStatement' . ElementLinks . ConceptualModelReferers . Method . MultiplePointerHeader . StatementOrder . ReferedObjectSet . 'endMultiplePointerStatement'	MultiplePointerStatement•pred := (∀ MethodTemporaryVariable•var ∈ MultiplePointerStatement• ReferedObjectSet•var . ∃ SpecificationElementPair•var ∈ SustainmentTable• var . ((SpecificationElementPair•SpecificationElement1•var = MultiplePointerStatement•Method•var) ∧ (SpecificationElementPair• SpecificationElement2•var = MethodTemporaryVariable•var))) ∧ (MultiplePointerStatement•StatementOrder = 1) ∧ (∀ Link•var ∈ MultiplePointerStatement•ElementLinks•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = MultiplePointerStatement•var) ∧ (SpecificationElementPair• SpecificationElement2•var = Link•var))) ∧ ElementLinks•pred ∧ ConceptualModelReferers•pred ∧ Method•pred ∧ MultiplePointerHeader•pred ∧ StatementOrder•pred ∧ ReferedObjectSet• pred
383	MultiplePointerHeader	'variables'	MultiplePointerHeader•pred := TRUE
384	ReferedObjectSet	MethodTemporaryVariable	ReferedObjectSet•pred := MethodTemporaryVariable•pred
385		MethodTemporaryVariable . ReferedObjectSet'	ReferedObjectSet•pred := MethodTemporaryVariable•pred ∧ ReferedObjectSet'•pred

386	**CompositeStatement**	'startCompositeStatement' . ElementLinks . ConceptualModelReferers . Method . CompositeStHeader . StatementOrder . Container . StatementAuxOrder . StatementSet . ConstraintExpression . 'endCompositeStatement'	CompositeStatement•pred := Pred32 ∧ Pred33 ∧ ElementLinks•pred ∧ ConceptualModelReferers•pred ∧ Method•pred ∧ CompositeStHeader •pred ∧ StatementOrder•pred ∧ Container•pred ∧ StatementAuxOrder•pred ∧ StatementSet•pred ∧ ConstraintExpression•pred Pred32 := (∀ Statement•var ∈ CompositeStatement•StatementSet•var . ((Statement•Method•var = CompositeStatement•Method•var) ∧ (Statement•Container•var = CompositeStatement•var) ∧ (Statement•StatementAuxOrder•var = 1) ∧ (Statement•StatementOrder•var ≥ 1) ∧ (Statement•StatementOrder•var ≤ lenthOf(CompositeStatement• StatementSet•var) ∧ (∀ Statement1•var ∈ CompositeStatement•StatementSet•var . ((Statement1• var ≠ Statement•var) ⇒ (Statement1•StatementOrder•var ≠ Statement• StatementOrder•var))))) ∧ (∀ Link•var ∈ CompositeStatement•ElementLinks•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = CompositeStatement •var) ∧ (SpecificationElementPair•SpecificationElement2•var = Link•var))) Pred33 := ((∃ CompositeStatement1•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = CompositeStatement1•var) ∧ (SpecificationElementPair• SpecificationElement2•var = CompositeStatement•var) ∧ (CompositeStatement•Container•var = CompositeStatement1•var))) ∨ (∃ DoubleCompositeStatement•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = DoubleCompositeStatement•var) ∧ (SpecificationElementPair• SpecificationElement2•var = CompositeStatement•var) ∧ (CompositeStatement•Container•var = DoubleCompositeStatement•var))))

387		'startCompositeStatement' . ElementLinks . ConceptualModelReferers . Method . CompositeStHeader . StatementOrder . RoundedStatementSet . ConstraintExpression . 'endCompositeStatement'	CompositeStatement•pred := Pred32 ∧ ElementLinks•pred ∧ ConceptualModelReferers•pred ∧ Method•pred ∧ CompositeStHeader •pred ∧ StatementOrder•pred ∧ StatementSet•pred ∧ ConstraintExpression•pred
388	CompositeStHeader	'if'	CompositeStHeader•pred := TRUE
389		'while'	CompositeStHeader•pred := TRUE
390		'repeat'	CompositeStHeader•pred := TRUE
391		'nTimes'	CompositeStHeader•pred := TRUE
392		'external'	CompositeStHeader•pred := TRUE
393	RoundedStatementSet	'startStatementSet' . Statement . 'endStatementSet'	RoundedStatementSet•pred := Statement•pred
394		'startStatementSet' . Statement . StatementSet . 'endStatementSet'	RoundedStatementSet•pred := Statement•pred ∧ StatementSet•pred
395	StatementSet	Statement	StatementSet•pred := Statement•pred
396		Statement . StatementSet'	StatementSet•pred := Statement•pred ∧ StatementSet'•pred
397	**DoubleCompositeStatement**	'startDoubleCompositeStatement' . ElementLinks . ConceptualModelReferers . Method . DoubleCompositeStHeader . StatementOrder . Container . StatementAuxOrder . FirstStatementSet . SecondStatementSet . ConstraintExpression . 'endDoubleCompositeStatement'	

DoubleCompositeStatement•pred :=
Pred34 ∧ Pred35 ∧ ElementLinks•pred ∧ ConceptualModelReferers•pred ∧ Method•pred ∧ DoubleCompositeStHeader •pred ∧ StatementOrder•pred ∧ Container•pred ∧ StatementAuxOrder•pred ∧ FirstStatementSet•pred ∧ SecondStatementSet•pred ∧ ConstraintExpression•pred

Pred34 :=
(∀ Statement•var ∈ DoubleCompositeStatement•FirstStatementSet•var . ((Statement•Method•var = DoubleCompositeStatement•Method•var) ∧ (Statement•Container•var = DoubleCompositeStatement•var) ∧ (Statement•StatementAuxOrder•var = 1) ∧ (Statement•StatementOrder•var ≥ 1) ∧ (Statement•StatementOrder•var ≤ lenthOf(DoubleCompositeStatement•FirstStatementSet•var) ∧ (∀ Statement1•var ∈ DoubleCompositeStatement•FirstStatementSet•var . ((Statement1•var ≠ Statement•var) ⇒ (Statement1•StatementOrder•var ≠ Statement•StatementOrder•var)))))
∧
(∀ Statement•var ∈ DoubleCompositeStatement•SecondStatementSet•var . ((Statement•Method•var = DoubleCompositeStatement•Method•var) ∧ (Statement•Container•var = DoubleCompositeStatement•var) ∧ (Statement•StatementAuxOrder•var = 2) ∧ (Statement•StatementOrder•var ≥ 1) ∧ (Statement•StatementOrder•var ≤ lenthOf(DoubleCompositeStatement•FirstStatementSet•var) ∧ (∀ Statement1•var ∈ DoubleCompositeStatement•SecondStatementSet•var . ((Statement1•var ≠ Statement•var) ⇒ (Statement1•StatementOrder•var ≠ Statement•StatementOrder•var)))))
∧
(∀ Link•var ∈ DoubleCompositeStatement•ElementLinks•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = DoubleCompositeStatement•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Link•var)))

Pred35 :=

((∃ CompositeStatement•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = CompositeStatement•var) ∧ (SpecificationElementPair•SpecificationElement2•var = DoubleCompositeStatement•var) ∧ (DoubleCompositeStatement•Container•var = CompositeStatement•var)))
∨ (∃ DoubleCompositeStatement1•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = DoubleCompositeStatement1•var) ∧ (SpecificationElementPair•SpecificationElement2•var = DoubleCompositeStatement•var) ∧ (DoubleCompositeStatement•Container•var = DoubleCompositeStatement1•var))))

398	**DoubleCompositeStatement**	'startDoubleCompositeStatement' . ElementLinks . ConceptualModelReferers . Method . DoubleCompositeStHeader . StatementOrder . FirstStatementSet . SecondStatementSet . ConstraintExpression . 'endDoubleCompositeStatement'

DoubleCompositeStatement•pred :=
Pred34 ∧ ElementLinks•pred ∧ ConceptualModelReferers•pred ∧ Method•pred ∧ DoubleCompositeStHeader •pred ∧ StatementOrder•pred ∧ FirstStatementSet•pred ∧ SecondStatementSet•pred ∧ ConstraintExpression•pred

399	DoubleCompositeStHeader	'ifElse'	DoubleCompositeStHeader•pred := TRUE
400	FirstStatementSet	RoundedStatementSet	FirstStatementSet•pred := RoundedStatementSet•pred
401	SecondStatementSet	RoundedStatementSet	SecondStatementSet•pred := RoundedStatementSet•pred
402	**Link**	'startLink ' . MainDocumentName . PointedDocumentTypeName . PointedDocumentName . SemanticMeaning . 'endLink '	Link•pred := MainDocumentName•pred ∧ PointedDocumentTypeName•pred ∧ PointedDocumentName•pred ∧ SemanticMeaning•pred
403	MainDocumentName	String	MainDocumentName•pred := TRUE
404	PointedDocumentTypeName	String	PointedDocumentTypeName•pred := TRUE
405	PointedDocumentName	String	PointedDocumentName•pred := TRUE
406	SemanticMeaning	TRUE	SemanticMeaning•pred := TRUE
407		FALSE	SemanticMeaning•pred := TRUE

	ConceptualModel	
408	**UseCaseDiagram**	'startUseCaseDiagram' . DocumentName . ElementLinks . UseCaseRepository . UseCaseActorRepository . UseCaseRelationshipRepository . 'endUseCaseDiagram'

UseCaseDiagram•pred :=
(∃ ActivityDiagram•var ∈ ConceptualModelRepository•var)
∧
(∃ UseCase•var ∈ ConceptRepository•var . (UseCase•var ∈ UseCaseDiagram•UseCaseRepository•var))
∧
(∀ UseCase•var ∈ UseCaseDiagram•UseCaseRepository•var . (UseCase•var ∈ ConceptRepository•var))
∧
(∀ UseCase•var ∈ ConceptRepository•var . (UseCase•var ∈ UseCaseDiagram•UseCaseRepository•var))

∧
(∀ UseCaseActor•var ∈ UseCaseDiagram•UseCaseActorRepository•var . (UseCaseActor•var ∈ ConceptRepository•var))
∧
(∀ UseCaseActor•var ∈ UseCaseDiagram•UseCaseActorRepository•var . ∃ UseCaseRelationship•var ∈ UseCaseDiagram•UseCaseRelationshipRepository•var .
(UseCaseRelationshipRepository•UseCaseActor•var = UseCaseActor•var))
∧
(∀ UseCaseActor•var ∈ ConceptRepository•var . (UseCaseActor•var ∈ UseCaseDiagram•UseCaseActorRepository•var))
∧
(∀ UseCaseRelationship•var ∈ UseCaseDiagram•UseCaseRelationshipRepository•var . (UseCaseRelationship•var ∈ ConceptRepository•var))
∧
(∀ UseCaseRelationship•var ∈ ConceptRepository•var . (UseCaseRelationship•var ∈ UseCaseDiagram•UseCaseRelationshipRepository•var))
∧
(∀ UseCase•var ∈ UseCaseDiagram•UseCaseRepository•var . ∃ Link•var ∈ UseCase•ElementLinks•var . ((Link•SemanticMeaning•var = TRUE) ∧ (Link•MainDocumentName•var =
Specification•DocumentName•var) ∧ (Link•PointedDocumentTypeName•var = stringOf(SequenceDiagram)[1])))
∧
((∃ UseCase•var ∈ UseCaseDiagram•UseCaseRepository•var . ∃ Link•var ∈ UseCase•ElementLinks•var . ∃ Link1•var ∈ UseCase•ElementLinks•var . ((Link•SemanticMeaning•var =
TRUE) ∧ (Link1•var ≠ Link•var))) ⇒ (Link1•SemanticMeaning•var = FALSE))
∧
((∃ UseCase•var ∈ UseCaseDiagram•UseCaseRepository•var . ∃ UseCase1•var ∈ ConceptRepository•var . ∃ Link•var ∈ UseCase•ElementLinks•var . ∃ Link1•var ∈ UseCase1•
ElementLinks•var . ((UseCase1•var ≠ UseCase•var) ∧ (Link•SemanticMeaning•var = TRUE) ∧ (Link1•SemanticMeaning•var = TRUE)))
⇒ (Link•PointedDocumentTypeName•var ≠ Link1•PointedDocumentTypeName•var))
∧
(∀ Link•var ∈ UseCaseDiagram•ElementLinks•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = UseCaseDiagram
•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Link•var)))
∧
DocumentName•pred ∧ ElementLinks•pred ∧ UseCaseRepository•pred ∧ UseCaseActorRepository•pred ∧ UseCaseRelationshipRepository•pred

409	**ActivityDiagram**	'startActivityDiagram' . DocumentName . ElementLinks . UseCaseRepository . TransitionRepository . 'endActivityDiagram' . 'endActivityDiagram'

ActivityDiagram•pred :=
(∃ UseCaseDiagram•var ∈ ConceptualModelRepository•var)
∧
(∃ UseCase•var ∈ ConceptRepository•var . (UseCase•var ∈ ActivityDiagram•UseCaseRepository•var))
∧
(∀ UseCase•var ∈ ActivityDiagram•UseCaseRepository•var . (UseCase•var ∈ ConceptRepository•var))
∧
(∀ UseCase•var ∈ ConceptRepository•var . (UseCase•var ∈ ActivityDiagram•UseCaseRepository•var))

---

1 "SequenceDiagram" é um símbolo não-terminal da gramática, que representa o modelo com o mesmo nome. "stringOf(não-terminal)" é uma função que retorna um string correspondente ao nome do símbolo não-terminal do argumento. No caso de "stringOf(SequenceDiagram)" o retorno é o string 'SequenceDiagram'.

∧
(∀ Transition•var ∈ ActivityDiagram•TransitionRepository•var . (Transition•var ∈ ConceptRepository•var))
∧
(∃ UseCase•var ∈ ActivityDiagram•UseCaseRepository•var . (UseCase•Initial•var = TRUE))
∧
((∃ UseCase•var ∈ ActivityDiagram•UseCaseRepository•var . ∃ UseCase1•var ∈ ActivityDiagram•UseCaseRepository•var . ((UseCase•Initial•var = TRUE) ∧ (UseCase1•var ≠ UseCase•var))) ⇒ (UseCase1•Initial•var = FALSE))
∧
((∃ UseCase•var ∈ ActivityDiagram•UseCaseRepository•var . ∃ UseCase1•var ∈ ActivityDiagram•UseCaseRepository•var . ((UseCase•Initial•var = TRUE) ∧ (UseCase1•var ≠ UseCase•var))) ⇒ (UseCase1•var ∈ accessibleFrom(UseCase•var)[2]))
∧
(∀ Link•var ∈ ActivityDiagram•ElementLinks•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = ActivityDiagram•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Link•var)))
∧
DocumentName•pred ∧ ElementLinks•pred ∧ UseCaseRepository•pred ∧ TransitionRepository•pred

410	**ClassDiagram**	'startClassDiagram' . DocumentName . ElementLinks . ClassRepository . BinaryRelationshipRepository . AggregationRepository . InheritanceRepository . 'endClassDiagram'

ClassDiagram•pred :=
(∀ ClassDiagram1•var ∈ ConceptualModelRepository•var . (ClassDiagram•var ≠ ClassDiagram1•var) ⇒ (ClassDiagram•DocumentName•var ≠ ClassDiagram1•DocumentName•var))
∧
(∀ Class•var ∈ ClassDiagram•ClassRepository•var . (Class•var ∈ ConceptRepository•var))
∧
(∀ BinaryRelationship•var ∈ ClassDiagram•BinaryRelationshipRepository•var . (BinaryRelationship•var ∈ ConceptRepository•var))
∧
(∀ Aggregation•var ∈ ClassDiagram•AggregationRepository•var . (Aggregation•var ∈ ConceptRepository•var))
∧
(∀ Inheritance•var ∈ ClassDiagram•InheritanceRepository•var . (Inheritance•var ∈ ConceptRepository•var))
∧
(∀ BinaryRelationship•var ∈ ClassDiagram•BinaryRelationshipRepository•var . (BinaryRelationship•Class1•var ∈ ClassDiagram•ClassRepository•var) ∧ (BinaryRelationship•Class2•var ∈ ClassDiagram•ClassRepository•var)))
∧
(∀ Aggregation•var ∈ ClassDiagram•AggregationRepository•var . (Aggregation•ClassWhole•var ∈ ClassDiagram•ClassRepository•var) ∧ (Aggregation•ClassPart•var ∈ ClassDiagram•ClassRepository•var)))
∧

---

[2] "accessibleFrom(UseCase•var)" é uma função que retorna os elementos acessíveis ao argumento, a partir de interligações promovidas por ocorrências de transição. No caso de um activity diagram, um use case A é acessível ao use case do argumento se houver uma transição em que origin for o use case do argumento e target, o use case A. Se um use Case A é acessível a um use case B e use case B é acessível a use case C, então use case A é acessível a use case C.

(∀ Inheritance•var ∈ ClassDiagram•InheritanceRepository•var . (Inheritance•Superclass•var ∈ ClassDiagram•ClassRepository•var) ∧ (Inheritance•Subclass•var ∈ ClassDiagram•ClassRepository•var)))
∧
(∀ Link•var ∈ ClassDiagram•ElementLinks•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = ClassDiagram•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Link•var)))
∧
DocumentName•pred ∧ ElementLinks•pred ∧ ClassRepository•pred ∧ BinaryRelationshipRepository•pred ∧ AggregationRepository•pred ∧ InheritanceRepository•pred

411	**SequenceDiagram**	'startSequenceDiagram' . DocumentName . ElementLinks . ObjectRepository . MessageRepository . ExternalReferenceRepository . UseCaseActorRepository . 'endSequenceDiagram'

SequenceDiagram•pred :=
(∀ SequenceDiagram1•var ∈ ConceptualModelRepository•var . (SequenceDiagram•var ≠ SequenceDiagram1•var) ⇒ (SequenceDiagram•DocumentName•var ≠ SequenceDiagram1•DocumentName•var))
∧
(∃ Message•var ∈ ConceptRepository•var . (Message•var ∈ SequenceDiagram•MessageRepository•var))
∧
(∀ Object•var ∈ SequenceDiagram•ObjectRepository•var . (Object•var ∈ ConceptRepository•var))
∧
(∀ Message•var ∈ SequenceDiagram•MessageRepository•var . (Message•var ∈ ConceptRepository•var))
∧
(∀ UseCaseActor•var ∈ SequenceDiagram•UseCaseActorRepository•var . (UseCaseActor•var ∈ ConceptRepository•var))
∧
(∀ ExternalReference•var ∈ SequenceDiagram•ExternalReferenceRepository•var . (ExternalReference•var ∈ ConceptRepository•var))
∧
(∀ Message•var ∈ SequenceDiagram•MessageRepository•var . (Message•TargetElement•var ∈ SequenceDiagram•ObjectRepository•var))
∧
((∃ Message•var ∈ ConceptRepository•var . ∃ Object•var ∈ ConceptRepository•var . ((Message•OriginElement•var = Object•var) ∧ (Message•var ∈ SequenceDiagram•MessageRepository•var))) ⇒ (Object•var ∈ SequenceDiagram•ObjectRepository•var))
∧
((∃ Message•var ∈ ConceptRepository•var . ∃ UseCaseActor•var ∈ ConceptRepository•var . ((Message•OriginElement•var = UseCaseActor•var) ∧ (Message•var ∈ SequenceDiagram•MessageRepository•var))) ⇒ (UseCaseActor•var ∈ SequenceDiagram•UseCaseActorRepository•var))
∧
((∃ Message•var ∈ ConceptRepository•var . ∃ ExternalReference•var ∈ ConceptRepository•var . ((Message•OriginElement•var = ExternalReference•var) ∧ (Message•var ∈ SequenceDiagram•MessageRepository•var))) ⇒ (ExternalReference•var ∈ SequenceDiagram•ExternalReferenceRepository•var))
∧
(∀ Object•var ∈ SequenceDiagram•ObjectRepository•var . ∃ Message•var ∈ SequenceDiagram•MessageRepository•var . ((Message•OriginElement•var = Object•var) ∨ (Message•TargetElement•var = Object•var)))
∧
(∀ UseCaseActor•var ∈ SequenceDiagram•UseCaseActorRepository•var . ∃ Message•var ∈ SequenceDiagram•MessageRepository•var . ((Message•OriginElement•var = UseCaseActor•var) ∧ (Message•TargetElement•Class•var ∈ UseCaseActor•ClassSet•var)))

∧
(∀ ExternalReference•var ∈ SequenceDiagram•ExternalReferenceRepository•var . ∃ Message•var ∈ SequenceDiagram•MessageRepository•var . (Message•OriginElement•var = ExternalReference•var))
∧
(∀ ExternalReference•var ∈ ConceptRepository•var . ∄ Message•var ∈ SequenceDiagram•MessageRepository•var . ((Message•OriginElement•var = ExternalReference•var) ⇒ (∃ Link•var ∈ ConceptRepository•var . ∃ UseCase•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((Link•SemanticMeaning•var = TRUE) ∧ (Link•MainDocumentName•var = Specification•DocumentName•var) ∧ (Link•PointedDocumentTypeName•var = stringOf(SequenceDiagram)) ∧ (Link•PointedDocumentName•var = SequenceDiagram•DocumentName•var) ∧ (SpecificationElementPair•SpecificationElement1•var = UseCase•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Link•var)))))
∧
((∃ ExternalReference•var ∈ ConceptRepository•var . ∃ Message•var ∈ SequenceDiagram•MessageRepository•var . (Message•OriginElement•var = ExternalReference•var))
⇒ ((∃ Link•var ∈ ConceptRepository•var . ∃ UseCase•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((Link•SemanticMeaning•var = TRUE) ∧ (Link•MainDocumentName•var = Specification•DocumentName•var) ∧ (Link•PointedDocumentTypeName•var = stringOf(SequenceDiagram)) ∧ (Link•PointedDocumentName•var = SequenceDiagram•DocumentName•var) ∧ (SpecificationElementPair•SpecificationElement1•var = UseCase•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Link•var)))
∨ (∃ Link•var ∈ ConceptRepository•var . ∃ Message1•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((Message1•var ≠ Message•var) ∧ (SpecificationElementPair•SpecificationElement1•var = Message1•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Link•var)))))
∧
((∃ ExternalReference•var ∈ ConceptRepository•var . ∃ Message•var ∈ SequenceDiagram•MessageRepository•var . ∃ Message1•var ∈ ConceptRepository•var . ((Message•OriginElement•var = ExternalReference•var) ∧ (Message1•var ≠ Message•var) ∧ (Message1•TargetMethod•var = Message•TargetMethod•var)))
⇒ ((∃ Object•var ∈ ConceptRepository•var . (Message•OriginElement•var = Object•var))
∨ (∃ UseCaseActor•var ∈ ConceptRepository•var . (Message•OriginElement•var = UseCaseActor•var))))
∧
((∃ ExternalReference•var ∈ ConceptRepository•var . ∃ Message•var ∈ SequenceDiagram•MessageRepository•var . ∃ Message1•var ∈ ConceptRepository•var . ((Message•OriginElement•var = ExternalReference•var) ∧ (Message1•var ≠ Message•var) ∧ (Message1•TargetMethod•var = Message•TargetMethod•var)))
⇒ (∄ Message2•var ∈ ConceptRepository•var . ((Message2•var ≠ Message1•var) ∧ (Message2•OriginElement•var = Message1•var))))
∧
((∃ ExternalReference•var ∈ ConceptRepository•var . ∃ Message•var ∈ SequenceDiagram•MessageRepository•var . ∃ Message1•var ∈ ConceptRepository•var . ((Message•OriginElement•var = ExternalReference•var) ∧ (Message1•var ≠ Message•var) ∧ (Message1•TargetMethod•var = Message•TargetMethod•var)))
⇒ (∃ Link•var ∈ ConceptRepository•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((Link•SemanticMeaning•var = TRUE) ∧ (Link•MainDocumentName•var = Specification•DocumentName•var) ∧ (Link•PointedDocumentTypeName•var = stringOf(SequenceDiagram)) ∧ (Link•PointedDocumentName•var = SequenceDiagram•DocumentName•var) ∧ (SpecificationElementPair•SpecificationElement1•var = Message1•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Link•var))))
∧
(∀ Link•var ∈ SequenceDiagram•ElementLinks•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = SequenceDiagram•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Link•var)))
∧
DocumentName•pred ∧ ElementLinks•pred ∧ ObjectRepository•pred ∧ MessageRepository•pred ∧ ExternalReferenceRepository•pred ∧ UseCaseActorRepository•pred

412	**StateTransitionDiagram**	'startStateTransitionDiagram' . DocumentName . ElementLinks . Class . State . StateRepository . TransitionRepository . 'endStateTransitionDiagram'

StateTransitionDiagram•pred :=
Pred36 ∧ Pred37 ∧ DocumentName•pred ∧ ElementLinks•pred ∧ Class•pred ∧ State•pred ∧ StateRepository•pred ∧ TransitionRepository•pred

Pred36 :=
(∀ StateTransitionDiagram1•var ∈ ConceptualModelRepository•var . (StateTransitionDiagram•var ≠ StateTransitionDiagram1•var) ⇒ (StateTransitionDiagram•DocumentName•var ≠ StateTransitionDiagram1•DocumentName•var))
∧
(∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = StateTransitionDiagram•Class•var) ∧ (SpecificationElementPair•SpecificationElement2•var = StateTransitionDiagram•var)))
∧
(∃ State•var ∈ ConceptRepository•var . (State•var ∈ StateTransitionDiagram•StateRepository•var))
∧
(∀ State•var ∈ StateTransitionDiagram•StateRepository•var . (State•var ∈ ConceptRepository•var))
∧
(∀ State•var ∈ StateTransitionDiagram•StateRepository•var . (State•Class•var = StateTransitionDiagram•Class•var))
∧
(∃ State•var ∈ StateTransitionDiagram•StateRepository•var . (State•LifeTimePosition•var = 'initial'))
∧
((∃ State•var ∈ StateTransitionDiagram•StateRepository•var . ∃ State1•var ∈ StateTransitionDiagram•StateRepository•var . ((State•LifeTimePosition•var = 'initial') ∧ (State1•var ≠ State•var))) ⇒ (State1•LifeTimePosition•var ≠ 'initial'))
∧
((∃ State•var ∈ StateTransitionDiagram•StateRepository•var . ∃ State1•var ∈ StateTransitionDiagram•StateRepository•var . ((State•LifeTimePosition•var = 'initial') ∧ (State1•var ≠ State•var))) ⇒ (State1•var ∈ accessibleFrom(State•var)))
∧
(∀ Transition•var ∈ StateTransitionDiagram•TransitionRepository•var . (Transition•var ∈ ConceptRepository•var))
∧
(∀ Transition•var ∈ StateTransitionDiagram•TransitionRepository•var . (Transition•Origin•var ∈ StateTransitionDiagram• StateRepository•var) ∧ (Transition•Target•var ∈ StateTransitionDiagram• StateRepository•var)))
∧
(∀ Transition•var ∈ StateTransitionDiagram•TransitionRepository•var . ∃ State•var ∈ StateTransitionDiagram•StateRepository•var . ((Transition•Origin•var = State•var) ∧ (State•LifeTimePosition•var ≠ 'final')))
∧
(∀ Link•var ∈ StateTransitionDiagram•ElementLinks•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = StateTransitionDiagram•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Link•var)))

Pred37 :=
(∀ StateTransitionDiagram1•var ∈ ConceptualModelRepository•var . (StateTransitionDiagram•var ≠ StateTransitionDiagram1•var) ⇒ ((StateTransitionDiagram•Class•var ≠ StateTransitionDiagram1•Class•var) ∨ (StateTransitionDiagram•State•var ≠ StateTransitionDiagram1•State•var)))

∧
(∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = StateTransitionDiagram•State•var) ∧ (SpecificationElementPair•SpecificationElement2•var = StateTransitionDiagram•var)))
∧
(∀ State•var ∈ StateTransitionDiagram•StateRepository•var . (State•Superstate•var = StateTransitionDiagram•State•var))
∧
((StateTransitionDiagram•State•LifeTimePosition•var = 'final') ⇒ (∃ State•var ∈ StateTransitionDiagram•StateRepository•var . (State•LifeTimePosition•var = 'final')))

413	**StateTransitionDiagram**	'startStateTransitionDiagram' . DocumentName . ElementLinks . Class . StateRepository . TransitionRepository . 'endStateTransitionDiagram'

StateTransitionDiagram•pred :=
Pred36 ∧ Pred38 ∧ DocumentName•pred ∧ ElementLinks•pred ∧ Class•pred ∧ StateRepository•pred ∧ TransitionRepository•pred

Pred38 :=
(∀ StateTransitionDiagram1•var ∈ ConceptualModelRepository•var . (((StateTransitionDiagram•var ≠ StateTransitionDiagram1•var) ∧ (StateTransitionDiagram•Class•var = StateTransitionDiagram1•Class•var)) ⇒
((StateTransitionDiagram1•State•var ∈ StateTransitionDiagram•StateRepository•var) ∨ (∃ State•var ∈ StateTransitionDiagram•StateRepository•var . (State•var ∈ superstatesOf(StateTransitionDiagram1•State•var))))))

414	**MethodBodyDiagram**	'startMethodBodyDiagram' . DocumentName . ElementLinks . Method . StatementRepository . 'endMethodBodyDiagram'

MethodBodyDiagram•pred :=

(∀ MethodBodyDiagram1•var ∈ ConceptualModelRepository•var . (MethodBodyDiagram•var ≠ MethodBodyDiagram1•var) ⇒ (MethodBodyDiagram•DocumentName•var ≠ MethodBodyDiagram1•DocumentName•var))
∧

(∀ MethodBodyDiagram1•var ∈ ConceptualModelRepository•var . (MethodBodyDiagram•var ≠ MethodBodyDiagram1•var) ⇒ (MethodBodyDiagram•Method•var ≠ MethodBodyDiagram1•Method•var))
∧

(∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = MethodBodyDiagram•Method•var) ∧ (SpecificationElementPair•SpecificationElement2•var = MethodBodyDiagram•var)))
∧

\\teste de modelo vazio: \\
((∃ PointerStatement•var ∈ ConceptRepository•var . (PointerStatement•var ∈ MethodBodyDiagram•StatementRepository•var))
∨ (∃ MessageStatement•var ∈ ConceptRepository•var . (MessageStatement•var ∈ MethodBodyDiagram•StatementRepository•var))
∨ (∃ DoublePointerStatement•var ∈ ConceptRepository•var . (DoublePointerStatement•var ∈ MethodBodyDiagram•StatementRepository•var))
∨ (∃ TextualStatement•var ∈ ConceptRepository•var . ((TextualStatement•var ∈ MethodBodyDiagram•StatementRepository•var) ∧ (TextualStatement•TextualHeader•var = 'task')))

∧
\\return como último statement \\
(∃ PointerStatement•var ∈ ConceptRepository•var . ((PointerStatement•var ∈ statementsOutsideContainerOf(StateTransitionDiagram•StatementRepository•var)[3]) ⇒
(PointerStatement•StatementOrder•var = lenthOf(statementsOutsideContainerOf(StateTransitionDiagram•StatementRepository•var))))

∧
\\necessidade de inicialização e uso de variáveis temporárias: \\
((∃ MethodTemporaryVariable•var ∈ ConceptRepository•var . (MethodTemporaryVariable•Method•var = MethodBodyDiagram•Method•var)) ⇒
\\task e não há uso explícito de variável:\\
((∃ TextualStatement•var ∈ MethodBodyDiagram•StatementRepository•var . ((TextualStatement•TextualHeader•var = 'task') ∧ (¬ ((∃ PointerStatement•var ∈ MethodBodyDiagram•StatementRepository•var . (PointerStatement•ReferedObject•var = MethodTemporaryVariable•var)) ∨ (∃ DoublePointerStatement•var ∈ MethodBodyDiagram•StatementRepository•var . (DoublePointerStatement•SecondReferedObject•var = MethodTemporaryVariable•var)) ∨ (∃ MessageStatement•var ∈ MethodBodyDiagram•StatementRepository•var . (MessageStatement•MessageTargetObject•var = MethodTemporaryVariable•var)) ∨ (∃ MessageStatement•var ∈ MethodBodyDiagram•StatementRepository•var . ∃ ParameterStructure•var ∈ MessageStatement•ParameterAssignmentSet•var . (ParameterStructure•AssignedElement•var = MethodTemporaryVariable•var))))))

\\OU task precede uso de variável: \\
∨ (∃ TextualStatement•var ∈ MethodBodyDiagram•StatementRepository•var . ((TextualStatement•TextualHeader•var = 'task')) ∧
((∃ PointerStatement•var ∈ MethodBodyDiagram•StatementRepository•var . ((PointerStatement•ReferedObject•var = MethodTemporaryVariable•var) ∧
(firstRunsBeforeSecond (TextualStatement•var, PointerStatement•var)[4])))

---

[3] "statementsOutsideContainerOf(StatementSet•var)" é uma função cujo argumento é um conjunto de statements e retorna um subconjunto do argumento em que todos os statements não estão contidos em outro statement, isto é, não possuem referência a uma ocorrência de "Container" em sua estrutura.

[4] "firstRunsBeforeSecond (Statement1•var, Statement2•var)" é uma função cujos argumentos são duas ocorrências de Statement de um mesmo modelo, e que retorna TRUE se em tempo de execução o primeiro statement for executado antes do segundo (isto é, seus equivalentes em código). Assim, o retorno será TRUE se uma das condições a seguir for verificada (e FALSE, caso contrário):
- se as duas ocorrências de Statement não estiverem contidas em uma ocorrência de CompositeContainer ou DoubleCompositeStatement, ou se referenciarem uma mesma ocorrência de CompositeContainer ou DoubleCompositeStatement e se
Statement1•StatementOrder•var < Statement2•StatementOrder•var (isto no caso de valores iguais para StatementAuxOrder; no caso de valores diferentes, a função retorna TRUE);
- se Statement1•var não estiver contido em uma ocorrência de CompositeContainer ou DoubleCompositeStatement e Statement2•var estiver (em qualquer nível de embutimento) e se Statement1•StatementOrder•var < Container2•StatementOrder•var (sendo Container2•var o container mais externo que contém Statement2•var);
- se Statement1•var estiver contido em uma ocorrência de CompositeContainer ou DoubleCompositeStatement (em qualquer nível de embutimento, sendo Container1•var o container mais externo) e Statement2•var não estiver e se Container1•StatementOrder•var < Statement2•StatementOrder•var;
- se as duas ocorrências de Statement estiverem contidas em ocorrências distintas de CompositeContainer ou DoubleCompositeStatement em qualquer nível de embutimento, sem container comum, sendo Container1•var e Container2•var os containers mais externos de Statement1•var e de Statement2•var, respectivamente, e se Container1•StatementOrder•var < Container2•StatementOrder•var;
- se as duas ocorrências de Statement estiverem contidas em uma mesma ocorrência de CompositeContainer ou DoubleCompositeStatement, porém em níveis de embutimento distintos e Container2•var contiver Statement2•var e referenciar o mesmo container referenciado por Statement1•var, se Statement1•StatementOrder•var < Container2•StatementOrder•var;
- se as duas ocorrências de Statement estiverem contidas em uma mesma ocorrência de CompositeContainer ou DoubleCompositeStatement, porém em níveis de embutimento distintos e Container1•var contiver Statement1•var e referenciar o mesmo container referenciado por Statement2•var, se Container1•StatementOrder•var < Statement2•StatementOrder•var.

∨
(∃ DoublePointerStatement•var ∈ MethodBodyDiagram•StatementRepository•var . ((DoublePointerStatement•SecondReferedObject•var = MethodTemporaryVariable•var) ∧ (firstRunsBeforeSecond (TextualStatement•var, DoublePointerStatement•var))))
∨
(∃ MessageStatement•var ∈ MethodBodyDiagram•StatementRepository•var . ((MessageStatement•MessageTargetObject•var = MethodTemporaryVariable•var) ∧ (firstRunsBeforeSecond (TextualStatement•var, MessageStatement•var))))
∨
(∃ MessageStatement•var ∈ MethodBodyDiagram•StatementRepository•var . ∃ ParameterStructure•var ∈ MessageStatement•ParameterAssignmentSet•var . ((ParameterStructure•AssignedElement•var = MethodTemporaryVariable•var) ∧ (firstRunsBeforeSecond (TextualStatement•var, MessageStatement•var))))))

\\OU inicialização precede uso de variável: \\
∨ (∃ DoublePointerStatement•var ∈ MethodBodyDiagram•StatementRepository•var . ∃ PointerStatement•var ∈ MethodBodyDiagram•StatementRepository•var . ((DoublePointerStatement•FirstReferedObject•var = MethodTemporaryVariable•var) ∧ (PointerStatement•ReferedObject•var = MethodTemporaryVariable•var) ∧ (firstRunsBeforeSecond (DoublePointerStatement•var, PointerStatement•var))))
∨
(∃ DoublePointerStatement1•var ∈ MethodBodyDiagram•StatementRepository•var . ∃ DoublePointerStatement2•var ∈ MethodBodyDiagram•StatementRepository•var . ((DoublePointerStatement1•var ≠ DoublePointerStatement2•var) ∧ (DoublePointerStatement1•FirstReferedObject•var = MethodTemporaryVariable•var) ∧ (DoublePointerStatement2•SecondReferedObject•var = MethodTemporaryVariable•var) ∧ (firstRunsBeforeSecond (DoublePointerStatement1•var, DoublePointerStatement2•var))))
∨
(∃ DoublePointerStatement•var ∈ MethodBodyDiagram•StatementRepository•var . ∃ MessageStatement•var ∈ MethodBodyDiagram•StatementRepository•var . ((DoublePointerStatement•FirstReferedObject•var = MethodTemporaryVariable•var) ∧ (MessageStatement•MessageTargetObject•var = MethodTemporaryVariable•var) ∧ (firstRunsBeforeSecond (DoublePointerStatement•var, MessageStatement•var))))
∨
(∃ DoublePointerStatement•var ∈ MethodBodyDiagram•StatementRepository•var . ∃ MessageStatement•var ∈ MethodBodyDiagram•StatementRepository•var . ∃ ParameterStructure•var ∈ MessageStatement•ParameterAssignmentSet•var . ((DoublePointerStatement•FirstReferedObject•var = MethodTemporaryVariable•var) ∧ (ParameterStructure•AssignedElement•var = MethodTemporaryVariable•var) ∧ (firstRunsBeforeSecond (DoublePointerStatement•var, MessageStatement•var))))
∨
(∃ MessageStatement•var ∈ MethodBodyDiagram•StatementRepository•var . ∃ PointerStatement•var ∈ MethodBodyDiagram•StatementRepository•var . ((MessageStatement•FirstReferedObject•var = MethodTemporaryVariable•var) ∧ (PointerStatement•ReferedObject•var = MethodTemporaryVariable•var) ∧ (firstRunsBeforeSecond (MessageStatement•var, PointerStatement•var))))
∨

(∃ MessageStatement•var ∈ MethodBodyDiagram•StatementRepository•var . ∃ DoublePointerStatement•var ∈ MethodBodyDiagram•StatementRepository•var . ((MessageStatement•FirstReferedObject•var = MethodTemporaryVariable•var) ∧ (DoublePointerStatement•SecondReferedObject•var = MethodTemporaryVariable•var) ∧ (firstRunsBeforeSecond (MessageStatement•var, DoublePointerStatement•var))))
∨
(∃ MessageStatement1•var ∈ MethodBodyDiagram•StatementRepository•var . ∃ MessageStatement2•var ∈ MethodBodyDiagram•StatementRepository•var . ((MessageStatement1•FirstReferedObject•var = MethodTemporaryVariable•var) ∧ (MessageStatement2•MessageTargetObject•var = MethodTemporaryVariable•var) ∧ (firstRunsBeforeSecond (MessageStatement1•var, MessageStatement2•var))))
∨
(∃ MessageStatement1•var ∈ MethodBodyDiagram•StatementRepository•var . ∃ MessageStatement2•var ∈ MethodBodyDiagram•StatementRepository•var . ∃ ParameterStructure•var ∈ MessageStatement2•ParameterAssignmentSet•var . ((MessageStatement1•FirstReferedObject•var = MethodTemporaryVariable•var) ∧ (ParameterStructure•AssignedElement•var = MethodTemporaryVariable•var) ∧ (firstRunsBeforeSecond (MessageStatement1•var, MessageStatement2•var))))
)))

∧
\\compatibilidade entre Method Body Diagrams e Sequence Diagrams\\
((∄ MessageStatement•var ∈ ConceptRepository•var . (MessageStatement•var ∈ MethodBodyDiagram•StatementRepository•var))
⇒ (∀ SequenceDiagram•var ∈ ConceptualModelRepository•var . ∃ Message1•var ∈ SequenceDiagram•MessageRepository•var . ∃ Message2•var ∈ SequenceDiagram•MessageRepository•var . (((Message1•var ≠ Message2•var) ∧ (Message2•OriginMessage •var ≠ Message1•var)) ⇒ (Message1•TargetMethod •var ≠ MethodBodyDiagram•Method•var))))

∧
(∃ SequenceDiagram•var ∈ ConceptualModelRepository•var . ∃ Message1•var ∈ SequenceDiagram•MessageRepository•var . ∃ Message2•var ∈ SequenceDiagram•MessageRepository•var . (((Message1•var ≠ Message2•var) ∧ (Message2•OriginMessage •var = Message1•var) ∧ (Message1•TargetMethod •var = MethodBodyDiagram•Method•var)) ⇒
(∃ MessageStatement•var ∈ MethodBodyDiagram•StatementRepository•var . ((MessageStatement•TargetMethod•var = Message2•TargetMethod•var) ∧ (∀ CompositeStatement•var ∈ ConceptRepository•var . ((CompositeStatement•CompositeHeader•var = 'external') ⇒ (MessageStatement•Container•var ≠ CompositeStatement•var))))))

∧
(∃ SequenceDiagram•var ∈ ConceptualModelRepository•var . ∃ Message1•var ∈ SequenceDiagram•MessageRepository•var . ∃ Message2•var ∈ SequenceDiagram•MessageRepository•var . ∃ MessageWrapper•var ∈ ConceptRepository•var . ((Message1•var ≠ Message2•var) ∧ (Message2•OriginMessage •var = Message1•var) ∧ (Message1•TargetMethod •var = MethodBodyDiagram•Method•var) ∧ (Message2•MessageWrapper •var = MessageWrapper•var)) ⇒
((∃ MessageStatement•var ∈ MethodBodyDiagram•StatementRepository•var . ∃ CompositeStatement•var ∈ ConceptRepository•var . ((MessageStatement•TargetMethod•var = Message2•TargetMethod•var) ∧ (MessageStatement•Container•var = CompositeStatement•var) ∧ (MessageWrapper •WrapperType•var = CompositeStatement•CompositeHeader•var) ∧ (MessageWrapper •ConstraintExpression•var = CompositeStatement•ConstraintExpression•var)))
∨ (∃ MessageStatement•var ∈ MethodBodyDiagram•StatementRepository•var . ∃ DoubleCompositeStatement•var ∈ ConceptRepository•var . ((MessageStatement•TargetMethod•var = Message2•TargetMethod•var) ∧ (MessageStatement•Container•var = DoubleCompositeStatement•var) ∧ (MessageWrapper •WrapperType•var = 'if')))))

∧

(∀ (C1 = {Message•var | ∃ Message1•var ∈ ConceptRepository•var . ((Message•OriginMessage •var = Message1•var) ∧ (Message1•TargetMethod •var = MethodBodyDiagram• Method•var)) }[5]) .
∃ (C2 = {MessageStatement•var | MessageStatement•var ∈ MethodBodyDiagram•StatementRepository•var }) ∈ sameRunningMessageSet(MethodBodyDiagram• StatementRepository•var[6]) .
((lenthOf(C1) = lenthOf(C2)) ∧
(∀ Message2•var ∈ C1 . ∃ MessageStatement2•var ∈ C2 . ((MessageStatement2•TargetMethod•var = Message2•TargetMethod•var)
∧ (∃ MessageWrapper2•var ∈ ConceptRepository•var . (Message2•MessageWrapper•var = MessageWrapper2•var) ⇒
((∃ CompositeStatement2•var ∈ ConceptRepository•var . ((MessageStatement2•Container•var = CompositeStatement2•var) ∧ (MessageWrapper2 •WrapperType•var = CompositeStatement2•CompositeHeader•var) ∧ (MessageWrapper2 •ConstraintExpression•var = CompositeStatement2•ConstraintExpression•var)))
∨ (∃ DoubleCompositeStatement2•var ∈ ConceptRepository•var . ((MessageStatement2•Container•var = DoubleCompositeStatement2•var) ∧ (MessageWrapper2 •WrapperType• var = 'if'))))))))))

∧
(∀ (C1 = {Message•var | ∃ Message1•var ∈ ConceptRepository•var . ((Message•OriginMessage •var = Message1•var) ∧ (Message1•TargetMethod •var = MethodBodyDiagram• Method•var)) }) .
∃ (C2 = {MessageStatement•var | MessageStatement•var ∈ MethodBodyDiagram•StatementRepository•var }) ∈ sameRunningMessageSet(MethodBodyDiagram• StatementRepository•var) .
(∀ {Message2•var, Message3•var} ⊆ C1 . ∃ {MessageStatement2•var, MessageStatement3•var} ⊆ C2 .
(((Message2•MessageOrder•var < Message3•MessageOrder•var) ⇒ (firstRunsBeforeSecond (MessageStatement2•var, MessageStatement3•var)))
∧ ((Message3•MessageOrder•var < Message2•MessageOrder•var) ⇒ (firstRunsBeforeSecond (MessageStatement3•var, MessageStatement2•var)))
∧ (MessageStatement2•TargetMethod•var = Message2•TargetMethod•var)
∧ (MessageStatement3•TargetMethod•var = Message3•TargetMethod•var)
∧ (∃ MessageWrapper2•var ∈ ConceptRepository•var . ((Message2•MessageWrapper•var = MessageWrapper2•var) ⇒
((∃ CompositeStatement2•var ∈ ConceptRepository•var . ((MessageStatement2•Container•var = CompositeStatement2•var) ∧ (MessageWrapper2 •WrapperType•var = CompositeStatement2•CompositeHeader•var) ∧ (MessageWrapper2 •ConstraintExpression•var = CompositeStatement2•ConstraintExpression•var)))

---

[5] É usado aqui um recurso de representação semelhante a "let...in" de VDM em que C1 é definido como uma variável que representa um conjunto e que é utilizada ao longo da expressão para evitar repetir a descrição do conjunto, que é bastante longa.

[6] "sameRunningMessageSet(StatementSet•var" é uma função cujo argumento é um conjunto de statements e que retorna o conjunto de todos os subconjuntos que podem ser obtidos a partir do argumento, sendo que os subconjuntos obedecem às seguintes restrições:
- cada subconjunto é composto somente por ocorrências de MessageStatement, que não estejam contidas em ocorrências de CompositeStatement, cujo valor de CompositeHeader seja 'external', em qualquer nível de embutimento;
- ocorrências de MessageStatement que não estejam contidas em ocorrências de CompositeStatement ou de DoubleCompositeStatement fazem parte de todos os subconjuntos;
- ocorrências de MessageStatement que referenciem uma mesma ocorrência de CompositeStatement ou de DoubleCompositeStatement (isto é, que estejam contidas nesta) se apresentarem valores iguais para StatementAuxOrder participam dos mesmos subconjuntos e se possuirem valores diferentes, nunca participam dos mesmos subconjuntos;
- se uma ocorrência de MessageStatement (m1) e uma ocorrência de CompositeContainer cujo valor de CompositeHeader seja 'if', ou 'while' ou de DoubleCompositeStatement (c1) não estiverem contidas em outra ocorrência de CompositeContainer ou DoubleCompositeStatement, ou se referenciarem uma mesma ocorrência de CompositeContainer ou DoubleCompositeStatement e apresentarem mesmo valor de StatementAuxOrder e ainda, se uma outra ocorrência de MessageStatement (m2) referencia c1 (ou está contida em c1 em qualquer nível de embutimento), então todo subconjunto que contém m2 também contém m1 e nem todo subconjunto que contém m1 contém m2.

∨ (∃ DoubleCompositeStatement2•var ∈ ConceptRepository•var . ((MessageStatement2•Container•var = DoubleCompositeStatement2•var) ∧ (MessageWrapper2 •WrapperType• var = 'if'))))))
∧ (∃ MessageWrapper3•var ∈ ConceptRepository•var . ((Message3•MessageWrapper•var = MessageWrapper3•var) ⇒
((∃ CompositeStatement3•var ∈ ConceptRepository•var . ((MessageStatement3•Container•var = CompositeStatement3•var) ∧ (MessageWrapper3 •WrapperType•var = CompositeStatement3•CompositeHeader•var) ∧ (MessageWrapper3 •ConstraintExpression•var = CompositeStatement3•ConstraintExpression•var)))
∨ (∃ DoubleCompositeStatement3•var ∈ ConceptRepository•var . ((MessageStatement3•Container•var = DoubleCompositeStatement3•var) ∧ (MessageWrapper3 •WrapperType• var = 'if')))))))))

∧
(∀ Link•var ∈ MethodBodyDiagram•ElementLinks•var . ∃ SpecificationElementPair•var ∈ SustainmentTable•var . ((SpecificationElementPair•SpecificationElement1•var = MethodBodyDiagram•var) ∧ (SpecificationElementPair•SpecificationElement2•var = Link•var)))

∧
DocumentName•pred ∧ ElementLinks•pred ∧ Method•pred ∧ StatementRepository•pred

Bibliografia

[ALE 77] ALEXANDER, C. **A pattern language**. New York: Oxford University Press, 1977.

[ALP 98] ALENCAR, P.S.C. *et al*. A model for gluing components. In: INTERNATIONAL WORKSHOP ON COMPONENT-ORIENTED PROGRAMMING, (WCOP), 3., 1998, Brussels. **Proceedings**... Brussels: [s.n.], 1998.

[ARA 91] ARANGO, G.; PRIETO-DÍAZ, R. Domain analysis concepts and research directions. In: PRIETO-DÍAZ, R.; ARANGO, G. **Domain analysis and software modeling**. Los Alamitos: IEEE Computer Society Press, 1991.

[ASS 97] ASSMANN, U., SCHMIDT, R. Towards a model for composed extensible components. In: FOUNDATIONS OF COMPONENT-BASED SYSTEMS WORKSHOP, 1997, Zurich. **Proceedings**... Zurich: [s.n.], 1997.

[BEC 94] BECK, K.; JOHNSON, R. **Patterns generate architectures**. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, (ECOOP), 8., 1994. **Proceeddings**... Berlin: Springer Verlag, 1994.

[BOO 91] BOOCH, G. **Object oriented design** - with applications. Redwood City, California: Benjamin Cummings, 1991.

[BOS 97] BOSCH, J. Adapting object-oriented components. In: INTERNATIONAL WORKSHOP ON COMPONENT-ORIENTED PROGRAMMING, (WCOP), 2., 1997, Jyväskylä. **Proceedings**... Jyväskylä: [s.n.], 1997.

[BRA 95] BRANT, J. **HotDraw**. Urbana: University of Illinois at Urbana-Champaign, 1995. Master thesis.

[BUS 96] BUSCHMANN, F. *et al*. **Pattern-oriented software architecture - a system of patterns**. Baffins Lane: John Wiley & Sons, 1996.

[CAM 97] CAMPO, M. R. **Compreensão visual de frameworks através da introspeção de exemplos**. Porto Alegre: UFRGS/II/CPGCC, mar. 1997. Tese de Doutorado.

[CAN 97] CANAL, C. *et al*. On the composition and extension of software components. In: FOUNDATIONS OF COMPONENT-BASED SYSTEMS WORKSHOP, 1997, Zurich. **Proceedings**... Zurich: [s.n.], 1997.

[CCI 88] CCITT. **Specification and description language (SDL)**. Geneva: CCITT, 1988. (Recommendation Z.100).

[CHE 76] CHEN, P. P. S. The entity-relationship model - toward a unified view of data. **ACM Transactions on Database Systems**, New York, v.1, n.1, Mar. 1976.

[CHI 90] CHIKOFSKY, E., CROS II, J. Reverse Engineering and design recovery: a taxonomy. **IEEE Transactions on Software Engineering**, New York, v.SE-4, n.1, Jan. 1977.

[CHR 92] CHRISTEL, M.; KANG, K. **Issues in requirements elicitation**. Pittsburgh: Software Engineering Institute, 1992. (Technical Report CMU/SEI-92-TR-12).

[COA 92] COAD, P.; YOURDON, E. **Análise baseada em objetos**. Rio de Janeiro: Campus, 1992.

[COA 93] COAD, P.; YOURDON, E. **Projeto baseado em objetos**. Rio de Janeiro: Campus, 1993.

[COL 94] COLEMAN, D. *et al.* **Object-oriented development**: the Fusion method. Englewood Cliffs: Prentice Hall, 1994.

[COL 97] COLEMAN, D. *et al*. UML: the language of blueprints for software? **SIGPLAN Notices**, New York, v.32, n.10, Oct.1997. Trabalho apresentado na OOPSLA, 1997.

[COM 81] COMMER. Principles of program design induced from experience with small public programs. **IEEE Transactions on Software Engineering**, New York, v.SE-7, n.2, 1981.

[DEC 91] DECHAMPEAUX, D. *et al*. Structured analysis an object oriented analysis. In: OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES AND APPLICATIONS CONFERENCE; EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, Oct. 1990, Ottawa. **Addendum to the proceeddings**... Ottawa: [s.n.], 1991.

[DEM 79] DEMARCO, T. **Structured analysis and system specification**. Englewood Cliffs: Prentice Hall, 1979.

[DER 76] DEREMER, F.; KRON, H.H. Programming-in-the-large versus programming-in-the-small. **IEEE Transactions on Software Engineering**, New York, v.SE-2, n.2, 1976.

[DEU 89] DEUTSCH, P. Frameworks and reuse in the smalltalk 80 system. In: BIGGERSTAFF, T. **Software reusability**. New York: ACM Press, 1989. v.1, p. 57-71.

[DSO 97] D'SOUZA, D. F. *et al*. **Objects, Components, and Frameworks With Uml**: The Catalysis Approach. [S.l.]: Addison-Wesley, 1997.

[DUR 96] DURHAM, A., JOHNSON, R. A framework for run-time systems and its visual programming language. **SIGPLAN Notices**, New York, v.31, n.10, Oct.1996. Trabalho apresentado na OOPSLA, 1996.

[ESP 93] ESPERANÇA, L. **A implementação de um gerador de editores dirigido por sintaxe e seu uso para uma linguagem de especificação formal**. Porto Alegre: UFRGS/II/CPGCC, jul. 1993. Dissertação de Mestrado.

[FAI 86] FAIRLEY, R. **Software Engineering concepts**. [S.l.]: McGraw-Hill, 1986.

[FAY 96] FAYAD, M.; CLINE, P. Aspects of software adaptability. **Communications of the ACM**, New York, v.39, n.10, Oct. 1996.

[FIO 95] FIORINI, S. *et al.* Integrando processos de negócio à elicitação de requisitos. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, (SBES), 9., Out. 1995, Recife. **Anais...** Recife: [s.n.], 1995.

[FOW 94] FOWLER, M. Describing and comparing object-oriented analysis and design methods. In: CARMICHAEL. **Object development methods**. New York: SIGS Books, 1994. p. 79-109.

[FRA 88] FRAGA, J.S. *et al*. A software environment for distributed applications. In: IFAC/IFIC, 15., 1988, Valencia. **Proceedings**... Valencia: [s.n.], 1988.

[GAM 94] GAMMA, E. **Design patterns**: elements of reusable object-oriented software. Reading: Addison-Wesley, 1994.

[GAN 78] GANE, C.; SARSON, T. **Structured systems analysis**: tolls and techniques. Englewood Cliffs: Prentice Hall, 1979.

[GAR 98] GARLAN, D. Higher-order connectors. In: WORKSHOP ON COMPOSITIONAL SOFTWARE ARCHITECTURES, 1998, Monterey. **Proceedings**... Monterey: [s.n.], 1998.

[GOM 94] GOMAA, H. *et al.* A prototype domain modeling enviroment for reusable software architectures. In: INTERNATIONAL CONFERENCE ON SOFTWARE REUSE, 3., 1994, Rio de Janeiro. **Proceedings**... Rio de Janeiro: [s.n.], 1994.

[HAR 87] HAREL, D. *et al.* Statecharts: a visual formalism for complex systems. **Science of Computer Progamming**, [S.l.], n.8, p.231-274, 1987.

[HED 98] HELTON, D. The impact of large-scale component and framework application development on business. In: INTERNATIONAL WORKSHOP ON COMPONENT-ORIENTED PROGRAMMING, (WCOP), 3., 1998, Brussels. **Proceedings**... Brussels: [s.n], 1998.

[HEL 90] HELM, R. et al. Contracts: specifying behaviour composition in object-oriented systems. **SIGPLAN Notices**, New York, v.25, n.10, Oct.1990. Trabalho apresentado na OOPSLA, 1990.

[HEN 95] HENNINGER, S. Developing domain knowledge through the reuse of project experiences. In: SYMPOSIUM ON SOFTWARE REUSABILITY, (SSR), 1995, Seattle. **Proceedings**... Seattle: [s.n], 1995.

[HOD 94] HODGSON, R. Contemplating the universe of methods. In: CARMICHAEL. **Object development methods**. New York: SIGS Books, 1994. p. 111-132.

[HON 97] HONDT, K. *et al*. Reuse contracts as component interface descriptions. In: INTERNATIONAL WORKSHOP ON COMPONENT-ORIENTED PROGRAMMING, (WCOP), 2., 1997, Jyväskylä. **Proceedings**... Jyväskylä: [s.n], 1997.

[IBM 96] IBM. **ObjChart white paper**. IBM Corporation, 1996. Disponível por www em www.software.ibm.com/clubopendoc/ocmoe.html (jul. 97).

[ICC 97] IC&C. **ADvance for Smalltalk 2.0**. IC&C GmbH, 1997. Disponível por www em www.io.com/~icc/products/advance/index.htm (jul. 97).

[JAC 91] JACOBSON, I. Re-engineering of old systems to an object-oriented architecture. **SIGPLAN Notices**, New York, v.26, n.10, Oct.1991. Trabalho apresentado na OOPSLA, 1991.

[JAC 92] JACOBSON, I. *et al.* **Object-oriented software engineering -** a use case driven approach. Reading: Addison-Wesley, 1992.

[JAC 97] JACOBSON, I. *et al.* **Software reuse**: architecture, process and organization for business reuse. [S.l.]: Addison-Wesley, 1997.

[JAM 83] JACKSON, M. **System development**. Englewood Cliffs: Prentice-Hall, 1983.

[JOH 88] JOHNSON, R. E., FOOTE, B. Designing reusable classes. **Journal of Object-Oriented Programming**, [S.l.], p. 22-35, June/July 1988.

[JOH 91] JOHNSON, R. E.; RUSSO, V. F. **Reusing object-oriented designs**. Urbana: University of Illinois , 1991. (Technical Report UIUCDCS91-1696).

[JOH 92] JOHNSON, R. E. Documenting frameworks using patterns. **SIGPLAN Notices**, New York, v.27, n.10, Oct.1992. Trabalho apresentado na OOPSLA, 1992.

[JOH 93] JOHNSON, R. E. **How to design frameworks**. 1993. Disponível por FTP anônimo em st.cs.uiuc.edu (dez. 98).

[JOH 94] JOHNSON, R. E. Documenting frameworks. **Frameworks Digest**, [S.l.], v.1, n.13, Oct. 1994.

[JOH 97] JOHNSON, R. E. **Components, frameworks, patterns.** Feb. 1997. Disponível por FTP anônimo em st.cs.uiuc.edu (dez. 98).

[KOZ 98] KOZACZYNSKI, W.; BOOCH, G. Component-based software engineering. **IEEE Software**, Los Alamitos, v.15, n.5, Sept. 1998.

[KRA 97] KRAJNC, M. **What is component-oriented programming?** Zurich: Oberon Microsystems, 1997.

[KRA 97b] KRAJNC, M. **Why component-oriented programming?** Zurich: Oberon Microsystems, 1997.

[LEI 92] LEITE, J. *et al.* Draco-PUC, experiências e resultados de reengenharia de software. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, (SBES), 6., 1992, Gramado. **Anais**... Gramado: [s.n.], 1992.

[LEI 94] LEITE, J. **Elicitação de requisitos**. Rio de Janeiro: PUC, 1994.

[LEW 95] LEWIS, T. *et al.* **Object-oriented application frameworks**. Greenwich: Manning, 1995.

[LIO 96] LIONS, J. L. **ARIANE 5 failure** - full report. Paris: ESA, 1996. Disponível por WWW em http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html)

[LUC 97] LUCAS, C. **Documenting reuse and evolution with reuse contracts**. Vrije: Universiteit Brussel, 1997. PhD thesis.

[MAR 91] MARTIN, J. **Técnicas estruturadas e CASE**. São Paulo: Makron Books, 1991.

[MAR 95] MARTIN, J.; ODELL, J. **Análise e projeto orientados a objetos**. São Paulo: Makron Books, 1995.

[MCI 68] MCILROY, M. D. Mass-produced software components. In: NORTH ATLANTIC TREATY ORGANIZATION CONFERENCE ON SOFTWARE ENGINEERING, 1968, Garmisch-Partenkirchen. **Proceedings**... Garmisch-Partenkirchen: [s.n.], 1968.

[MEL 99] MELLOR, S. **Automatic code generation from UML models**. 1999. Disponível por FTP anônimo em http://www.projtech.com/pdfs/autocg-1.pdf (maio 1999).

[MEU 97] MEUSEL, M. *et al*. A model for structuring user documentation of object-oriented frameworks using patterns and hypertext. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, (ECOOP), 1997. **Proceedings**... [S.l.:s.n.], 1997.

[MEY 88] MEYER, B. **Object-oriented software construction**. Englewood Cliffs: Prentice Hall, 1988.

[MEY 97] MEYER, B. **Object-oriented software construction**. 2.ed. [S.l.]: Prentice Hall PTR, 1997.

[MIC 94] MICROGOLD SOFTWARE. How to - user manual. In: **With Class** - help index. Microgold Software, 1994. Conteúdo do arquivo wc2.hpl, que acompanha a versão shareware, disponível por FTP anônimo em www.microgold.com, no arquivo Stage/WC31Demo.html (jul. 97).

[MIL 97] MILI, R. *et al*. Storing and retrieving software components: a refinement based system. **IEEE Transactions on Software Engineering**, New York, v.23, n.7. July 1997.

[MON 92] MONARCHI, D. E.; PUHR, G. A research typology for object-oriented analysis and design. **Communications of the ACM**, New York, v.35, n.9, Sept. 1992.

[MOR 94] MOREIRA, A. M. D. **Rigorous object-oriented analysis**. Stirling: University of Stirling, 1994. Doctor of Philosophy Thesis.

[MUR 96] MURER, T. *et al*. Improving component interoperability. In: SPECIAL ISSUES IN OBJECT-ORIENTED PROGRAMMING, WORKSHOP OF THE ECOOP, 1996, Linz. **Proceedings**... Linz: [s.n.], 1996.

[NEI 89] NEIGHBORS, J. Draco: a method for engineering reusable software systems. In: PRIETO-DÍAZ, R.; ARANGO, G. **Domain analysis and software modeling**. Los Alamitos: IEEE Computer Society Press, 1991.

[NOW 97] NOWACK, P. Frameworks - representations and perspectives. In: WORKSHOP ON LANGUAGE SUPPORT FOR DESIGN PATTERNS AND FRAMEWORKS, 1997, Jyväskylä. **Proceedings**... Jyväskylä: [s.n.], 1997.

[OLA 96] ÓLAFSSON, A.; DOUG, B. On the need for "required interfaces" of components. In Special Issues in Object-Oriented Programming, Workshop of the ECOOP, 1996, Linz. **Proceedings**... Linz: [s.n.], 1996.

[OLI 96] OLIVEIRA, G. **As ferramentas CASE orientadas pelos objetos**. Lisboa: Universidade Nova de Lisboa, 1996. Dissertação de mestrado.

[OPD 92] OPDYKE, W. **Refactoring object-oriented frameworks**. Urbana: University of Illinois at Urbana-Champaign, 1992. Doctor of Philosophy Thesis.

[PAD 86] PARNAS, D.; CLEMENTS, P. A rational design process: how and why to fake it. **IEEE Transactions on Software Engineering**, New York, v.SE-12, n.2, 1986.

[PAR 94] PARCPLACE SYSTEMS. **VisualWorks User's Guide**. Sunnyvale: ParcPlace Systems, 1994.

[PAS 94] PASTOR, E. A. **Estudo comparativo de metodologias de desenvolvimento de software**: trabalho individual. Porto Alegre: UFRGS/II/CPGCC, 1994.

[PEN 95] PENTEADO, R. *et al*. Engenharia reversa orientada a objetos do ambiente StatSim: método utilizado e resultados obtidos. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, (SBES), 9., 1995, Recife. **Anais...** Recife: [s.n.], 1995. p.345-360.

[PER 95] PEREZ, M., ANTONETI, J. A qualitative comparison of object-oriented methodologies. In: INTERNATIONAL SYMPOSIUM ON SYSTEMS RESEARCH, INFORMATICS AND CYBERNETICS, 5., 1995, Baden. **Proceedings**... Baden: [s.n.], 1995.

[PET 62] PETRI, C. A. **Kommunikation mit automaten**. Bonn: Institut für Instrumentelle Mathematik, 1962. (Schriften des IIM Nr. 2).

[PFI 96] PFISTER, C. Object models: SOM and COM. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, (ECOOP), 1996, Linz. **Tutorial Notes**... Linz: [s.n.], 1996.

[PRE 94] PREE, W. **Design patterns for object oriented software development**. Reading: Addison-Wesley, 1994.

[PRE 95] PREE, W. *et al*. Active guidance of framework development. **Software - Concepts and Tools**, [S.l.], v.16, n.3, 1995.

[PRR 82] PRESSMAN, R. **Software engineering**. a practioner´s approach. [S.l.]: McGraw-Hill, 1982.

[PRI 87] PRIETO-DÍAZ, R. Domain analysis for reusability. In: PRIETO-DÍAZ, R.; ARANGO, G. **Domain analysis and software modeling**. Los Alamitos: IEEE Computer Society Press, 1991.

[RAT 97] RATIONAL SOFTWARE CORPORATION. **UML notation guide**. 1997. Disponível por FTP em www.rational.com, no arquivo uml/ad970805_uml11_Notation2.zip (set. 98).

[ROS 77] ROSS, D. T. Structured analysis (SA): a language for communicating ideas. **IEEE Transactions on Software Engineering**, New York, v.SE-3, n.1, 1977.

[RUM 94] RUMBAUGH, J. *et al.* **Modelagem e projetos baseados em objetos**. Rio de Janeiro: Campus, 1994.

[SEE 98] SEETHARAMAN, K. The CORBA connection. **Communications of the ACM**, New York, v.33, n.9, Sept. 1998.

[SHA 96] SHAW, M.; GARLAN, D. **Software architecture** - perspectives on an emerging discipline. Upper Saddle River: Prentice Hall, 1996.

[SIL 96] SILVA, R. P.; PRICE, R. T. Em direção a uma metodologia para o desenvolvimento de frameworks de aplicação orientados a objetos. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, (SBES), 10., 1996, São Carlos. **Anais**... São Carlos: [s.n.], 1996. p.325-340.

[SIL 97] SILVA, R. P.; PRICE, R. T. O uso de técnicas de modelagem no projeto de frameworks orientados a objetos. In: INTERNATIONAL CONFERENCE OF THE ARGENTINE COMPUTER SCIENCE AND OPERATIONAL RESEARCH SOCIETY, (JAIIO), 26.; ARGENTINE SYMPOSIUM ON OBJECT ORIENTATION, (ASOO), 1., 1997, Buenos Aires. **Proceedings**... Buenos Aires: [s.n.], 1997. p.87-94.

[SIL 98] SILVA, R. P.; PRICE, R. T. A busca de generalidade, flexibilidade e extensibilidade no processo de desenvolvimento de frameworks orientados a objetos. In: WORKSHOP IBEROAMERICANO DE ENGENHARIA DE REQUISITOS E AMBIENTES DE SOFTWARE, (IDEAS), 1998, Torres. **Anais**... Porto Alegre: Instituto de Informática / UFRGS, 1998. v.2, p.298-309.

[SIL 98b] SILVA, R. P.; PRICE, R. T. Tool support for helping the use of frameworks. In: INTERNATIONAL CONFERENCE OF THE CHILEAN COMPUTER SCIENCE SOCIETY, (SCCC), 18., 1998, Antofagasta. **Proceedings**... Los Alamitos: IEEE Computer Society, 1998. p.192-201.

[SIL 99] SILVA, R. P.; PRICE, R. T. Suporte ao desenvolvimento e uso de componentes flexíveis. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, (SBES), 13., 1999, Florianópolis. **Anais**... Florianópolis: [s.n.], 1999. p.13-28.

[SIM 95] SIMOS, M. Organization Domain Modeling (ODM): formalizing the core domain modeling life cycle. In: SYMPOSIUM ON SOFTWARE REUSABILITY, (SSR), 1995, Seattle. **Proceedings...** Seattle: [s.n.], 1995.

[SZY 96] SZYPERSKI, C. Component-oriented programming: a refined variation on object-oriented programming. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, (ECOOP), 1996, Linz. **Tutorial Notes**... Linz: [s.n.], 1996.

[SZY 96b] SZYPERSKI, C. *et al*. Sumary of the First International Workshop on Component-Oriented Programming. In: INTERNATIONAL WORKSHOP ON COMPONENT-ORIENTED PROGRAMMING, (WCOP), 1., 1996, Linz. **Proceedings**... Linz: [s.n.], 1996.

[SZY 97] SZYPERSKI, C. *et al*. Sumary of the Second International Workshop on Component-Oriented Programming. In: INTERNATIONAL WORKSHOP

ON COMPONENT-ORIENTED PROGRAMMING, (WCOP), 2., 1997, Jyväskylä. **Proceedings**... Jyväskylä: [s.n.], 1997.

[TAL 94] TALIGENT. **Building object-oriented frameworks**. [S.l.]: Taligent, 1994. White paper.

[TAL 95] TALIGENT. **Leveraging object-oriented frameworks**. [S.l.]: Taligent, 1995. White paper.

[UNI 96] UNIVERSITY OF ALBERTA. **What is meta-CASE ?** 1996. Disponível por www em www.cs.ualberta.ca, no arquivo ~softeng/meta_case.html (jul. 97).

[WEC 96] WECK, W. **Independently extensible component frameworks**. In: INTERNATIONAL WORKSHOP ON COMPONENT-ORIENTED PROGRAMMING, (WCOP), 1., 1996, Linz. **Proceedings**... Linz: [s.n.], 1996.

[WEL 97] WELCH, I. *et al*. Adaptation of connectors in software architectures. In: INTERNATIONAL WORKSHOP ON COMPONENT-ORIENTED PROGRAMMING, (WCOP), 3., 1998, Brussels. **Proceedings**... Brussels: [s.n.], 1998.

[WES 93] WEST, R. **Reverse Engineering - an overview**. London: HSMO, 1993.

[WIR 90] WIRFS-BROCK, R.; JOHNSON, R. E. Surveying current research in object-oriented design. **Communications of the ACM**, New York, v.33, n.9, Sept. 1990.

[WIA 91] WIRFS-BROCK, A. *et al*. Designing reusable designs: Experiences designing object-oriented frameworks. In: OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES AND APPLICATIONS CONFERENCE; EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 1991, Ottawa. **Addendum to the proceeddings**... Ottawa: [s.n.], 1991.

[YOU 89] YOURDON, E. **Modern structured analysis**. Englewood Cliffs: Prentice Hall, 1989.

[ZAN 98] ZANCAN, J. **Nautilus**: uma ferramenta de apoio à navegação sobre estruturas de frameworks. Porto Alegre: UFRGS/II/PPGC, 1998. Dissertação de Mestrado.