ORÁCULO:
UM SISTEMA DE CRÍTICAS PARA A UML

Alexandre Ribeiro Dantas

MONOGRAFIA DE PROJETO FINAL DE CURSO SUBMETIDA AO DEPARTAMENTO DA CIÊNCIA DA COMPUTAÇÃO DO INSTITUTO DE MATEMÁTICA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO REQUISITO PARA A OBTENÇÃO DO GRAU EM BACHAREL EM INFORMÁTICA

Aprovada por:

_____
Prof.ª Cláudia Maria Lima Werner, D.Sc.
(Presidente)

_____
Alexandre Luis Correa, M.Sc.
(Co-orientador)

_____
Prof. Marcos Roberto da Silva Borges, Ph.D.

_____
Prof. Eber Assis Schmitz, Ph.D.

Rio de Janeiro, RJ
Junho de 2001

Agradecimentos

Agradeço amplamente a todos que colaboraram de alguma forma, durante toda a minha vida, para o meu crescimento acadêmico, profissional e pessoal.

Agradeço em especial a professora Cláudia Werner pela sempre precisa e amigável orientação, pela confiança e estímulo que foram determinantes para minha formação.

Agradeço aos demais amigos de muitas horas do Laboratório de Engenharia de Software, que contribuíram compartilhando seus conhecimentos e experiências, proporcionando também muitos momentos de alegria, cumplicidade e amizade. Em especial, Márcio de Oliveira Barros, Alexandre Luís Correa, Leonardo Gresta Paulino Murta, José Ricardo Xavier, Regina Maciel Braga, Robson Pinheiro, Gustavo Olanda Veronese, Alessandréia Marta de Oliveira, Marcelo Nascimento Costa e Marco Aurélio Mangan.

Agradeço a Deus pela proteção, sabedoria e coragem que me fizeram ser a pessoa que sou hoje; e pela esperança de me tornar a cada dia uma pessoa melhor para o mundo e para aqueles que me cercam.

ORÁCULO:
UM SISTEMA DE CRÍTICAS PARA A UML

Alexandre Ribeiro Dantas

Orientadora: D.Sc. Cláudia Maria Lima Werner.
Co-Orientador: M.Sc. Alexandre Luís Correa.

RESUMO

Este trabalho apresenta um mecanismo para verificação de críticas sobre modelos construídos durante o desenvolvimento de software, denominado Oráculo. As críticas existentes são baseadas nas regras de boa-formação da linguagem UML, e novos tipos de críticas podem ser facilmente adicionadas. São também analisadas a importância e a implementação de sistemas de críticas no contexto de reutilização de software.

ORÁCULO:
AN UML CRITICISM SYSTEM

Alexandre Ribeiro Dantas

Supervisor: D.Sc. Cláudia Maria Lima Werner.
Co-Supervisor: M.Sc. Alexandre Luís Correa.

ABSTRACT

This work presents a mechanism of criticism verification for constructed models from software development, named Oráculo. Critics are based on the well-formed rules from UML language specification, and new types of critics can be easily included. The importance and the implementation of a criticism system in the context of software reuse is also considered.

Sumário

CAPÍTULO I
INTRODUÇÃO

1.1 – MOTIVAÇÃO

A qualidade dos artefatos produzidos durante as fases iniciais de um processo de desenvolvimento de software é fundamental para o sucesso do decorrer do projeto e para a manutenção de sua viabilidade. Conrow e Shishido (1997) apontam resultados de uma pesquisa sobre 8.380 projetos de software comerciais, indicando que 31% dos projetos foram cancelados e 53% apresentavam sérios problemas, como aumentos de 189% e 222% sobre os custos e o cronograma iniciais, respectivamente, e apenas 61% dos requisitos inicialmente esperados. A maior parte dos erros de um sistema – 64% – está associada às fases de análise de requisitos e projeto, e só são descobertos em etapas mais avançadas como codificação e testes. O custo para correção de um erro ainda durante a análise equivale a 1/5 do que seria durante a fase de testes e a 1/15 depois que o sistema estivesse em uso (Kotonya e Sommerville, 1996). Apesar dos resultados, a expectativa e demanda por software mais complexo e distribuído cresce cada vez mais, tornando-se uma força propulsora em muitas organizações, que sofrem para suprir esta demanda de valor estratégico com qualidade, custos viáveis e em tempo hábil. Os resultados podem ser ainda mais catastróficos em organizações onde não se verifica um comprometimento mínimo com algum tipo de processo organizado e sistemático, e bons princípios de desenvolvimento.

Os principais artefatos das fases de análise e projeto são modelos que representam o problema sendo resolvido em um nível de abstração mais elevado do que a visão computacional, permitindo um melhor entendimento das várias visões do projeto e servindo como base para as etapas seguintes do processo, como codificação e validação. Um dos requisitos principais para um bom método de análise ou projeto é uma definição precisa da notação utilizada. Quando não há uma verificação da consistência e da correção dos modelos sendo construídos, em virtude de notações

imprecisas ou suscetíveis a diferentes interpretações, aliadas ao baixo conhecimento do domínio do problema, eleva-se o grau de incerteza sobre as especificações e as chances de ambigüidades e erros serem introduzidos despercebidamente.

O desenvolvimento de software baseado em componentes é uma abordagem viável e eficiente que pode ser aplicada para a construção de sistemas de forma a obter maior produtividade, qualidade e flexibilidade (Werner, 1999). A reutilização pode ocorrer sobre quase todos os artefatos produzidos durante o desenvolvimento, desde códigos-fonte até especificações e casos de testes. Neste caso, a reutilização é mais efetiva quando aplicada logo sobre os produtos da análise e projeto – os modelos – onde se concentra a maior parte dos esforços. Entretanto, as conseqüências das falhas e conflitos gerados sobre estes produtos quando reutilizados em diversos projetos podem ser desastrosas, estendendo-se àqueles que reutilizaram ou tiveram como base modelos inconsistentes.

1.2 – OBJETIVO

Segundo Booch (1996), nossa habilidade em imaginar novas aplicações complexas sempre será superior à nossa habilidade de desenvolvê-las, e a construção de coisas erradas é um dos motivos da maioria das falhas dos projetos de software. O objetivo deste trabalho é apresentar a importância da construção de modelos consistentes, segundo uma notação estabelecida e padronizada – a *Unified Modeling Language, UML* – como um fator relevante para o sucesso do desenvolvimento de software e, em especial, para o desenvolvimento com reutilização baseada em modelos de domínios de aplicações (Prieto-Diaz e Arango, 1991).

Neste sentido, será descrita uma proposta para a verificação da consistência dos modelos sendo construídos segundo as regras estabelecidas pela especificação da própria linguagem utilizada como notação. Apresentamos ainda a implementação desta proposta no contexto de uma infra-estrutura de desenvolvimento de software baseado em modelos de domínio.

1.3 – ORGANIZAÇÃO DO DOCUMENTO

Este trabalho está dividido em cinco capítulos. Nesta primeira etapa, foi ilustrada a motivação principal do trabalho, o objetivo e a organização deste documento.

O segundo capítulo analisa a utilização de modelos durante algumas das principais fases do desenvolvimento de software e também no contexto da reutilização de software. É apresentada a UML como uma linguagem padronizada para a construção de modelos e a importância da consistência destes modelos para o sucesso de desenvolvimento. Algumas abordagens para obtenção de modelos mais consistentes também são apresentadas, como formalismos. São introduzidos o conceito de sistemas de críticas e suas principais características como mecanismo para obtenção de modelos mais consistentes. Ao final, algumas ferramentas de modelagem são analisadas.

O terceiro capítulo apresenta uma proposta de um sistema de críticas para as regras de boa-formação definidas pela especificação da linguagem UML. São apresentadas as principais construções e características do subconjunto da UML utilizado, assim como suas regras. Ao final, um sistema de críticas é descrito através de seus principais requisitos, seu funcionamento e interação com o usuário, além de detalhes ao nível de projeto.

O quarto capítulo apresenta uma implementação do sistema de críticas proposto em um contexto de uma infra-estrutura de apoio a reutilização baseada em modelos de domínio – Odyssey – sendo desenvolvido pela equipe de reutilização em Engenharia de Software do Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ. É apresentada em detalhes a organização da informação semântica nesta infra-estrutura, assim como a ferramenta de diagramação, onde está inserido o sistema de críticas. Ao final, é mostrado como as principais funcionalidades do sistema foram implementadas e são executadas dentro da infra-estrutura Odyssey.

O quinto capítulo apresenta as conclusões deste trabalho, suas principais contribuições e limitações, assim como perspectivas de trabalhos futuros.

CAPÍTULO II
SISTEMA DE CRÍTICAS NA MODELAGEM

Este capítulo apresenta a importância da modelagem nas fases de desenvolvimento de software, a representação dos modelos através da UML e como sistemas de críticas podem influenciar positivamente nos resultados sobre os modelos sendo construídos. O capítulo é dividido em cinco seções. As duas primeiras, buscam oferecer um panorama geral da importância do uso de modelos nas atividades de desenvolvimento e na reutilização de software. A terceira seção analisa como a utilização de sistemas de críticas sobre ferramentas de modelagem é útil para a construção eficaz de modelos consistentes e no auxílio à tomada de decisões de projeto. A quarta seção apresenta a UML como linguagem padrão para a representação dos modelos. Finalmente, a seção final apresenta alguns mecanismos alternativos existentes na busca de modelos UML consistentes, assim como uma breve análise de ferramentas atualmente disponíveis quanto às suas principais características em relação a alguns fatores sugeridos.

2.1 – CONCEITOS DE ANÁLISE E PROJETO DE SISTEMAS

Análise e projeto representam fases importantes dos processos de desenvolvimento de software. O entendimento dos requisitos é essencial para o sucesso do desenvolvimento, provendo a especificação e os modelos do sistema e do ambiente a que ele deve pertencer, de forma clara e precisa, enquanto um bom projeto pode ser uma das bases para a qualidade do software em desenvolvimento (Pressman, 1997).

O objetivo da análise de requisitos é a compreensão e representação da informação sobre o domínio da aplicação, funcionalidades e comportamentos esperados e a construção de modelos que particionem estas informações de forma a constituir uma especificação básica do sistema sendo desenvolvido. O uso de modelos com múltiplas visões e notação gráfica é fundamental para facilitar o entendimento do problema,

diminuir a complexidade inerente aos projetos de software, atuar como base para as demais fases e servir como meio de comunicação padronizado entre os membros da equipe.

Muitos problemas e falhas surgem durante a especificação dos requisitos em virtude das dificuldades desta fase inicial do desenvolvimento. Incertezas e erros podem ocorrer em virtude da má comunicação ou falta de especialistas nas informações do domínio do problema sendo analisado. Membros da equipe podem apresentar vocabulário conflitante, ambíguo e má interpretação dos conceitos e funcionalidades envolvidos, em função do uso de linguagem natural, não precisa. Há ainda a própria dificuldade do usuário em explicitar de forma completa, clara e estável seus requisitos. Técnicas de entrevistas e comunicação auxiliam na tentativa de diminuição das dificuldades entre analistas e usuários. O uso de uma notação precisa que atue como base para evolução, documentação e comunicação, facilita a organização, o entendimento e a visualização padronizada do problema. O uso de modelos consistentes desde esta etapa, portanto, mostra-se determinante para evitar a propagação de erros para as fases posteriores e diminuir os esforços para corrigi-los.

Durante o projeto, verifica-se o refinamento dos resultados obtidos pela análise de requisitos, em direção a uma visão menos abstrata e mais próxima da realidade física e implementacional, ou seja, uma visão computacional do problema. Portanto, o objetivo principal é a construção de modelos que acomodem especificações sobre a estruturação dos dados e arquitetura do programa, além de detalhes implementacionais e de interface, relacionados diretamente às funcionalidades e comportamentos esperados. Uma série de conceitos determina a qualidade de um projeto, como coesão, acoplamento, abstração, particionamento hierárquico e de controle, entre outros. O uso de modelos ruins nesta etapa pode comprometer a codificação, a flexibilidade e a manutenção do sistema (Pressman, 1997).

Há, atualmente, vários métodos que descrevem como as fases de análise e projeto podem ser realizadas. A evolução ocorreu dos métodos centrados em processos para os centrados em dados e, posteriormente, para os centrados em objetos. Estes últimos têm apresentado maior destaque atualmente, pelo uso de conceitos simples e naturais de objetos e seus relacionamentos na representação dos problemas, ao invés do uso de blocos funcionais e fluxo de informação. A orientação a objetos apresenta-se favorável à modularização e reutilização de componentes, facilita a transição entre

análise e projeto, exibe melhores resultados em qualidade, produtividade e apresenta-se mais flexível a mudanças e adaptações. A análise e projeto orientados a objetos são voltados para a construção de modelos melhores, mais próximos da realidade através do uso dos principais conceitos da orientação a objetos como abstração, encapsulamento, herança e polimorfismo, criando um vocabulário e entendimento comuns entre usuários do sistema e os desenvolvedores.

2.2 – A REUTILIZAÇÃO DE SOFTWARE E A ENGENHARIA DE DOMÍNIO

A reutilização de software é uma aposta simples e poderosa para se atingir melhor produtividade no desenvolvimento através da construção de novos sistemas a partir do uso de qualquer artefato já produzido e utilizado em soluções de problemas similares. Desta forma, observa-se um maior ganho em flexibilidade e adaptação, melhores índices de qualidade gerais e menor esforço de desenvolvimento. Entretanto, várias dificuldades atuam contra a obtenção efetiva de reutilização, principalmente na identificação, recuperação e compreensão, além de possível má qualidade dos componentes (Werner, 1999).

A aplicação da reutilização requer uma gerência especial, com atividades que definam uma estratégia para reuso, sua efetiva implementação dentro da organização e que controlem a criação, gerência e utilização de componentes. A criação de componentes é uma das etapas mais importantes, também conhecida como desenvolvimento para reutilização, onde é realizada a engenharia de domínio. A utilização de componentes previamente criados, avaliados e armazenados para a construção de novos sistemas é a etapa conhecida como desenvolvimento com reutilização, onde é executada a engenharia de aplicação (Werner, 1999).

A engenharia de domínio tenta solucionar um dos problemas da reutilização, que é a criação de componentes que possam ser reutilizados em outras aplicações além daquela para a qual foram projetados. Através da análise de domínio, são criados componentes mais genéricos que capturam a funcionalidade essencial do domínio de aplicação, estando mais aptos à sua reutilização efetiva (Prieto-Diaz, 1987). A análise de domínio é uma atividade similar a análise de requisitos para sistemas individuais, porém trabalha em um nível de abstração maior que envolve um conjunto de possíveis aplicações que pertencem a uma família ou domínio comum, compartilhando várias

similaridades, variações e diferenças que o especialista do domínio considera essenciais. O resultado da análise de domínio é um modelo de domínio que contém estas características comuns dentro do domínio.

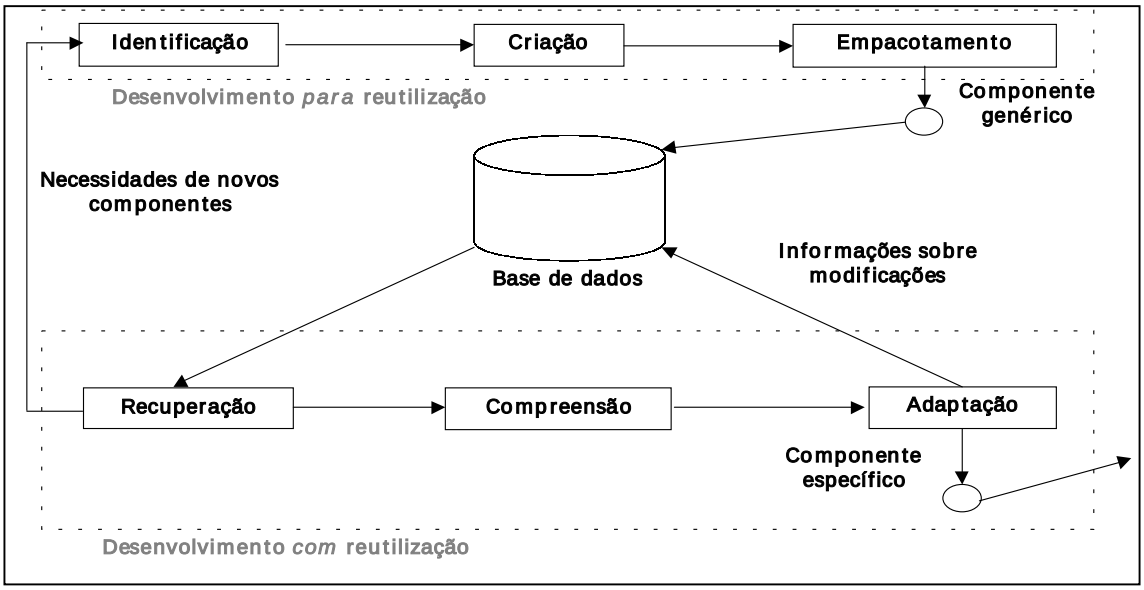

Figura 2.1 (Murta, 1999)

Um modelo para um sistema individual pode ser criado a partir do refinamento do modelo do seu domínio com os requisitos específicos para o sistema em questão. O uso de modelo de domínio e suas visões contendo inconsistências, ambigüidades e erros multiplica os efeitos desastrosos sobre a produtividade e qualidade, pois afeta toda uma família de sistemas sendo desenvolvidos a partir de componentes comuns.

2.3 – SISTEMAS DE CRÍTICAS

Nas seções anteriores, apresentamos a importância da construção de modelos consistentes para as principais fases de desenvolvimento de software - análise e projeto - como também no desenvolvimento baseado em componentes. Inicialmente analisamos o que são sistemas de críticas e qual a importância que eles exercem para a modelagem.

Um sistema de críticas pode ser entendido como um mecanismo inteligente que atua sobre ferramentas de modelagem, segundo um determinado processo, através de correções e/ou sugestões sobre os modelos sendo projetados. Esse mecanismo é válido não somente na verificação de construções sintáticas presas a uma linguagem de modelagem, mas também é de grande utilidade em um nível de abstração maior, como

no contexto de tomada de decisões de projeto, ou até mesmo na verificação de consistência entre múltiplas visões de um ambiente de desenvolvimento baseado em modelos de domínio e na criação de componentes e aplicações para uma família de sistemas (Gomaa *et al*., 1996).

Segundo Robbins (1999), projetistas estão sempre envolvidos no desafio de tomar boas decisões de projeto que requerem amplo conhecimento do problema, do domínio de soluções e também experiências acumuladas. Além disto, defende a teoria cognitiva de *reflection-in-action*, em que os projetistas conseguem as melhores avaliações sobre seus projetos enquanto estão envolvidos no contexto das tomadas de decisões, e não algum tempo depois.

A presença de um sistema de críticas sobre uma ferramenta de modelagem é motivada por vários aspectos, como conhecimento limitado dos projetistas sobre o domínio de conhecimento, baixo custo de revisão imediata, possibilidade de aprendizado contínuo, alto custo das falhas associadas a erros de projeto, redução de tempo de mercado e gerenciamento de riscos. As críticas atuam na descoberta antecipada de erros e incompletudes, assim como na sugestão de melhores alternativas e conselhos baseados em heurísticas. Desta forma, podemos perceber duas abordagens do funcionamento de um sistema de críticas: a autoritária, que analisa o modelo segundo a presença ou ausência de determinada propriedade, e a informativa, que visa detectar potenciais problemas e auxilia o projetista a tomar as melhores decisões e a evoluir seu projeto. Algumas das características desejadas para um sistema de críticas são (Robbins *et al*., 1998):

- Identificar oportunidades para evolução do projeto
- Atuar no contexto das decisões que dispararam a crítica
- Permitir o uso de regras baseadas em heurísticas
- Fornecer retorno ao usuário de forma útil e fácil

De modo geral, um processo para um sistema de críticas deve possuir as seguintes etapas (Robbins *et al.*, 1998):

- Ativação – Seleção de quais críticas são relevantes para as decisões correntes
- Detecção – Atividade de descoberta de erros pelas críticas ativas

- Aconselhamento – Sugestões oferecidas com base no conhecimento especificado pelas críticas
- Evolução – Possibilidade de melhorias no modelo em função das críticas
- Registro – Armazenamento das resoluções sobre cada crítica descoberta

2.4 – UNIFIED MODELING LANGUAGE

As seções anteriores apresentaram o uso de modelos como artefato determinante em várias etapas do desenvolvimento e também na reutilização de software. Modelos são documentos expressos em uma linguagem determinada, geralmente gráfica, que contêm semânticas interpretáveis de forma fácil para quem os lê. Através dos modelos, conseguimos obter múltiplas visões do sistema, particionando a complexidade para sua compreensão e atuando como meio de comunicação entre os participantes do projeto. Portanto, uma linguagem de modelagem padronizada é fundamental para a construção e o entendimento de bons modelos.

Uma linguagem deve conter elementos de modelagem que representem os conceitos e semânticas fundamentais, uma notação para visualização gráfica dos elementos de modelos e regras e conselhos de como usá-la. A sintaxe define as construções existentes e como elas compõem outras, de forma independente de notação. As semânticas definem como as instâncias das construções devem se relacionar com outras instâncias, de forma a exibir algum significado bem formado (OMG, 1999).

A partir de 1994, Grady Booch, Jim Rumbaugh e Ivar Jacobson iniciaram a unificação dos seus métodos, que já despontavam dentre os métodos orientados a objetos existentes na época. Desta unificação, foi criada a *Unified Modeling Language* (UML), posteriormente aceita pela OMG como a linguagem padrão (1997) e estando, neste momento, na versão 1.3 (1999). A UML é descrita abaixo (OMG, 1999):

> *A Unified Modeling Language (UML) é uma linguagem para especificação, visualização, construção e documentação de artefatos de sistemas de software, assim como para modelagem de negócios e outros tipos de sistemas. A UML representa uma coletânea das melhores práticas de engenharia que se mostraram vitoriosas na modelagem de sistemas grandes e complexos.*

A UML é independente de linguagens de programação. São definidos também mecanismos de extensibilidade e especialização, o que permite a construção de projetos individuais com novos conceitos e restrições específicas, além das oferecidas pela base da linguagem. Entretanto, ela não descreve um método ou processo, sendo apenas uma linguagem de modelagem que define uma forma padronizada de notação e ampla semântica, uma vez que diferentes organizações e problemas requerem diferentes processos. A linguagem é portanto independente de processo, porém adequada a abordagens orientadas a casos de uso, centradas em arquitetura, iterativas e incrementais (OMG, 1999).

A arquitetura da UML é baseada em quatro camadas: meta-metamodelo, metamodelo, modelo e objetos. A primeira define uma linguagem para especificar metamodelos, a segunda define uma linguagem para a construção de modelos, que compõem a terceira camada, definindo uma linguagem para descrever um domínio de informação. A quarta e última camada é composta pelos objetos que representam o domínio de informação sendo modelado. O foco de nossas atenções é como construir modelos (terceira camada) consistentes a partir da segunda camada, o metamodelo lógico, que é formado através de linguagem gráfica, natural e formal (*Object Constraint Language* - *OCL*).

A UML apresenta-se organizada em duas grandes categorias: elementos estáticos e dinâmicos. Dos estáticos fazem parte os mecanismos de extensão (estereótipo, restrição, propriedades), tipos de dados, a base de construtores estáticos fundamentais e os modelos estruturais representados pelos diagrama de classes, objetos, componentes e *deployment*. Dos dinâmicos fazem parte as colaborações, casos de uso, máquinas de estados e grafos de atividades. A linguagem apresenta nove tipos de diagramas no total, e diversos elementos sintáticos para ampla modelagem. No escopo deste trabalho, será estudado apenas um subconjunto da UML, considerado essencial e simples o suficiente para sua utilização na modelagem de sistemas de informação, abordando os diagramas de casos de uso, classes, estados e seqüência.

2.5 – ABORDAGENS PARA CRIAÇÃO DE MODELOS CONSISTENTES

Nesta seção são apresentadas algumas abordagens que procuram auxiliar ou prover a construção de modelos de forma consistente. Inicialmente, são analisadas as

tentativas de uso de formalismos como fator para obtenção de modelos mais precisos. Em seguida, é apresentado e analisado um conjunto de ferramentas de modelagem existentes, e de que forma elas oferecem suporte de sistema de críticas.

2.5.1 – Formalização da Linguagem

Como visto anteriormente, a semântica da linguagem UML é descrita pela segunda camada – o metamodelo – através de linguagem gráfica, natural e formal. Através do uso de uma linguagem formal (*Object Constraint Language*), percebe-se a primeira tentativa de criar um mecanismo para obter e verificar a consistência dos modelos construídos, de forma a eliminar ambigüidades e adicionar semânticas de restrições (regras de boa formação). A OCL é uma linguagem semi-formal "tipada", sendo desenvolvida para modelagem de negócios e para fácil uso e entendimento, ao contrário das linguagens formais tradicionais, cuja complexidade matemática inibe seu uso para uma fácil modelagem, adaptada a semântica dinâmica da UML. A OCL é usada para alguns dos seguintes propósitos (OMG, 1999):

- Especificar invariantes
- Descrever pré e pós-condições de operações
- Descrever condições de guarda e restrições
- Como linguagem navegacional

O uso da OCL para aumentar a consistência dos modelos, porém, é restrito à adição de semântica (invariantes, restrições, etc.) estática somada à bem conhecida sintaxe da UML. A semântica dinâmica ainda permanece descrita através do uso de linguagem natural e ambígua, e a UML apresenta amplo escopo passível de alterações e evoluções. Além disto, através da OCL não há mecanismos de provas formais e validações rigorosas sobre os modelos.

Apesar das especificações formais apresentarem-se complexas para pessoas não adaptadas ao uso matemático como linguagem e da dificuldade de integração destas especificações com ferramentas de desenvolvimento com suporte gráfico para a criação de modelos orientados a objetos, a formalização da UML tem sido estudada e proposta por diversos autores (Vasconcelos, 1999; Araújo, 2000; France *et al*, 1997; Evans *et al.*,

1999). As principais motivações para o uso de formalismo, segundo eles, são ganhos em clareza, consistência dos diagramas e especificações, correção através de provas formais, e refinamento de um modelo mais abstrato para um modelo implementacional correto (Evans *et al*, 1999).

O *UML Precise Group* (France *et al*, 1997) é uma das tentativas de estudo e desenvolvimento de uma referência formal para a UML, com o objetivo de criar um manual de referência para a linguagem UML, com uma descrição precisa dos seus componentes e regras para analisar suas propriedades. Segundo Vasconcelos (1999), há três principais métodos para formalização orientada a objetos. No primeiro, expressões formais substituem declarações em linguagem natural dos modelos; no segundo, notações formais existentes são estendidas para acomodar características da orientação a objetos; e no terceiro, há a criação de especificações formais a partir de modelos informais através de um mapeamento das estruturas sintáticas para artefatos do domínio formal. Araújo (2000), por exemplo, criou uma linguagem textual que descreve um subconjunto de diagramas da UML e permite analisar formalmente, através de regras sintáticas em gramáticas, critérios de consistência inter-diagramas e intra-diagramas. Vasconcelos (1999) compôs um *framework* semântico para suportar a formalização dos principais elementos de modelagem estáticos baseando-se em especificações algébricas e regras para o mapeamento de acordo com a sintaxe e semântica descrita pela UML.

2.5.2 – Análise de Ferramentas Existentes

Nesta seção são avaliadas algumas ferramentas de modelagem disponíveis atualmente no mercado ou via Internet para demonstração. O objetivo é verificar o subconjunto da linguagem UML que as ferramentas disponibilizam, assim como quaisquer mecanismos para controle e verificação da consistência dos modelos sendo editados e sua interação com o usuário.

Neste sentido, torna-se necessária a criação de critérios para uma avaliação e comparação entre as ferramentas. A análise será guiada pelos seguintes fatores:

- **Subconjunto da UML –** Analisa quantas e quais construções as ferramentas disponibilizam, segundo a especificação da linguagem, em termos de diagramas, elementos de modelagem e suas principais características. O

subconjunto da UML considerado essencial será descrito no próximo capítulo em detalhes.

- **Categorização das Críticas** – Analisa quantas e quais regras são verificadas, das que são sugeridas pela especificação da linguagem. Da mesma forma que o subconjunto da UML, as regras de boa-formação consideradas são listadas no próximo capítulo.

- **Configurabilidade** – Analisa quantas e quais opções o usuário possui para configurar as críticas verificadas pela ferramenta, seja através da ativação/desativação completa da verificação ou permitindo a seleção de determinadas críticas para serem verificadas. Analisa também as formas de expansão e edição do conjunto de críticas verificadas.

- **Interação com o Usuário** – Analisa como é a interação entre a verificação de críticas e o usuário da ferramenta de modelagem. A verificação deve atuar de forma clara e explicativa, preemptiva sobre as ações que geram inconsistências, e permitir a criação de relatórios de consistência de todos os modelos assíncronos à edição.

2.5.2.1 – ArgoUML (University of California)

A ferramenta de edição de diagramas apresenta praticamente quase todos os diagramas e elementos considerados essenciais definidos pela especificação da UML. É possível também utilizar OCL para especificar restrições com a ajuda de um compilador especial. As propriedades dos elementos, entretanto, não se apresentam de forma completa.

O ArgoUML possui agentes trabalhando transparentemente durante a edição que analisam críticas do projeto e possíveis melhorias, que variam desde erros sintáticos, pontos incompletos, guias de estilos e recomendações de projetistas experientes. O resultado deste trabalho nunca interrompe o usuário.

A figura 2.2 exibe a janela principal de edição da ferramenta, com destaque para o painel de Itens a fazer. A janela de configuração de críticas é mostrada pela figura 2.3.

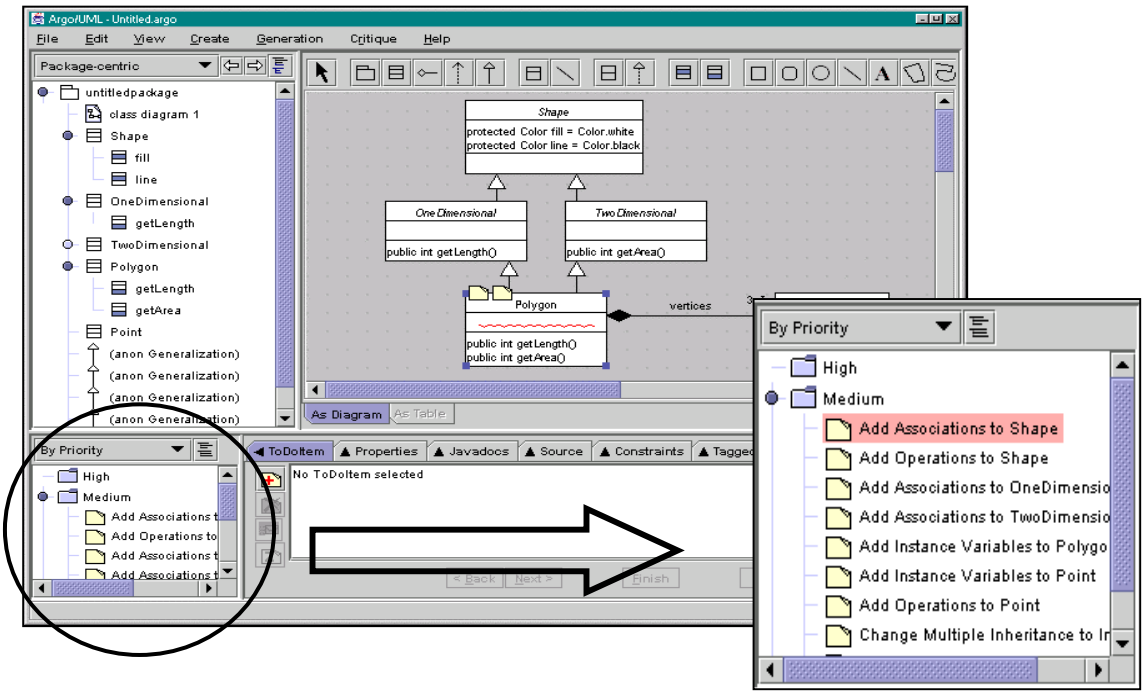

Figura 2.2 - ArgoUML

O conjunto de críticas da ferramenta não está vinculado estritamente às regras de boa-formação da UML, adotando regras e heurísticas de bons projetos que auxiliem o projetista também no contexto da tomada de decisões de projeto. Em relação às regras da UML, apresenta um conjunto básico de verificações para as máquinas de estado e casos de uso, porém algumas regras do modelo estrutural e regras básicas e genéricas (como questões de visibilidade entre pacotes) não são verificadas.

Não é possível criar relatórios, porém as críticas feitas e seus conselhos explicativos são exibidos em um painel de itens a fazer, segundo uma classificação de acordo com a gravidade (baixa, média, alta) ou prioridade. É possível resolver o item (inclusive com ajuda automática da ferramenta), congelá-lo e abandoná-lo (especificando a razão da desistência). Não há fontes de ajuda, mas é possível enviar e-mail para uma pessoa especialista.

O usuário tem a possibilidade de configurar se deseja trabalhar com a verificação de críticas ativadas ou selecionar quais elementos são interessantes para o seu projeto. Não há formas de expansibilidade, porém a ferramenta ArgoUML espera, no futuro, disponibilizar um editor de críticas representadas através de redes de predicados e

ações, além de um editor para possíveis correções automáticas para os problemas detectados pelas críticas. Atualmente, apenas poucas críticas podem ser corrigidas automaticamente pela ferramenta.

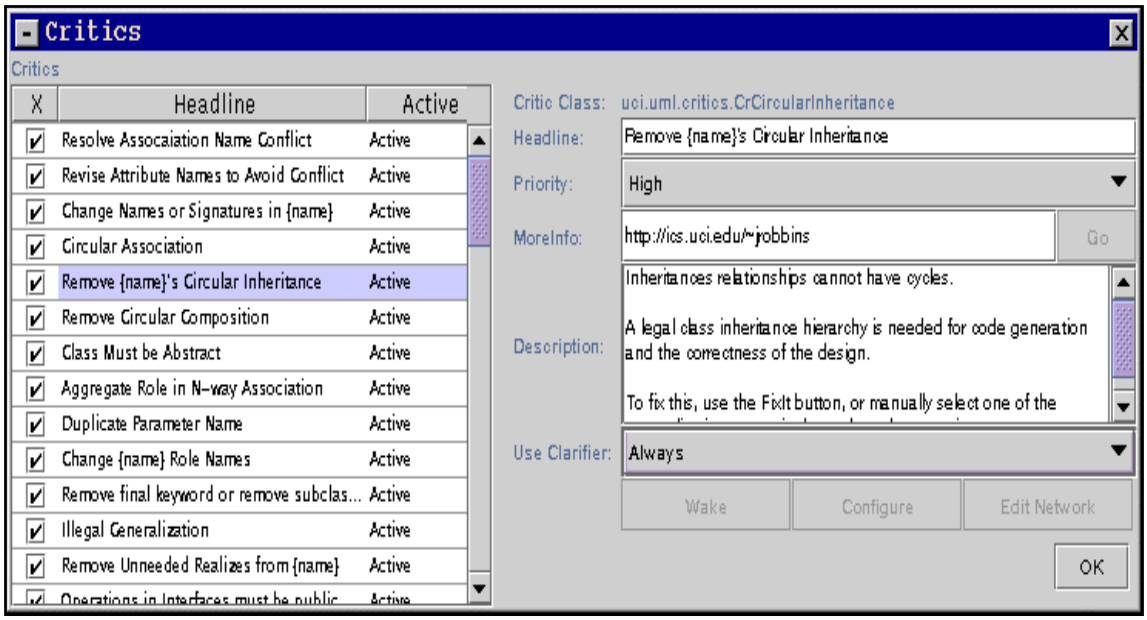

Figura 2.3 – Configuração de Críticas (ArgoUML)

2.5.2.2 – Magic Draw (NoMagic)

A ferramenta Magic Draw apresenta a edição de diagramas de forma compatível com a especificação da sintaxe, diagramas, propriedades e notação dos principais elementos de modelagem da UML. A usabilidade da diagramação é intuitiva e fácil para o usuário.

A principal vantagem da ferramenta é dispor das características dos elementos descritos pela UML, assim como sua sintaxe e notação. As regras de modelos de casos de uso e máquina de estados foram todas encontradas, porém poucas regras do modelo de classes e regras básicas são verificadas. A figura 2.4 exibe a janela principal de edição de modelos, com destaque para construções incorretas segundo a especificação da UML.

A ferramenta não possui nenhuma forma de desativação completa da verificação ou configuração de quais críticas estão ativas, assim como edição e expansão para novas críticas. A ação do mecanismo de verificação não permite a geração de relatórios assíncronos, tampouco exibe explicações sobre as verificações efetuadas, que ocorrem de forma preemptiva sobre as ações do usuário. É possível também, dispor de ajuda

sobre a UML. Não há, também, nenhum apoio para correção automática de críticas, uma vez que a ferramenta impede o prosseguimento da ação que iria causar uma inconsistência.

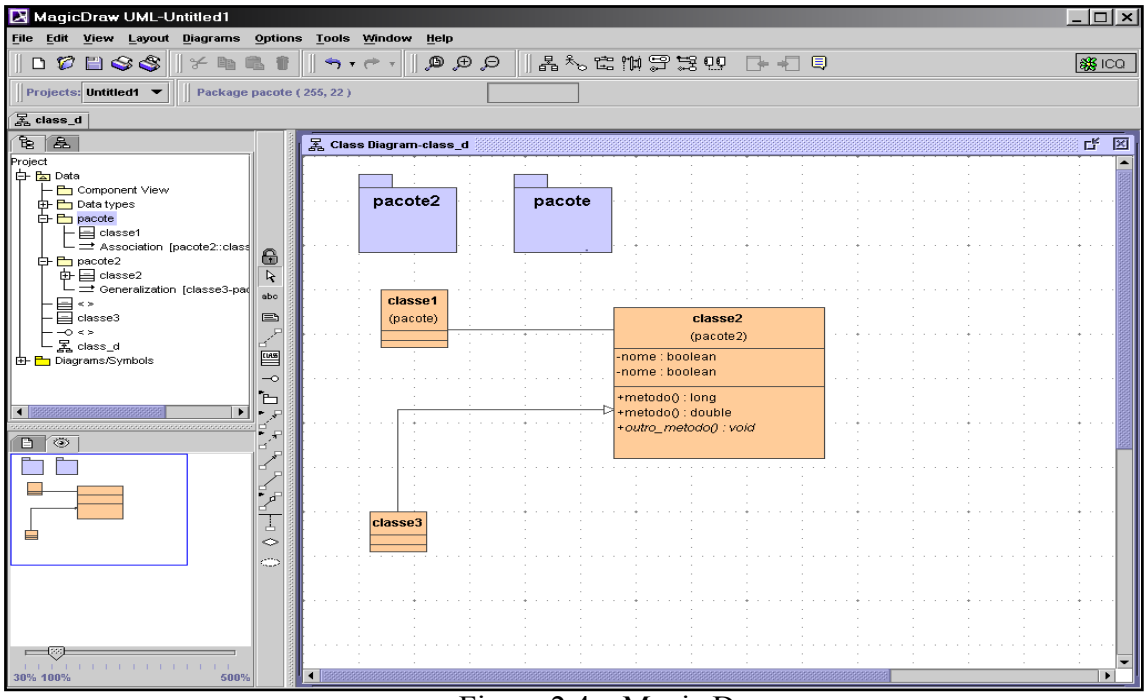

Figura 2.4 – Magic Draw

2.5.2.3 – Objecteering (Softeam)

A diagramação desta ferramenta contém todos os diagramas da linguagem e seus elementos de modelagem. As propriedades dos elementos, sintaxe e notação também estão totalmente de acordo com o descrito pela UML. A figura 2.5 ilustra a janela principal da ferramenta com o diálogo de verificação de consistência ativo.

A ferramenta Objecteering apresenta-se como a mais completa na verificação das regras semânticas estáticas definidas pela especificação da UML. As regras básicas e estruturais foram quase todas encontradas, assim como no caso do ArgoUML, com algumas exceções. As regras de casos de uso também são verificadas, porém algumas regras em relação à pseudoestados de máquinas de estados não são avaliadas.

O mecanismo de verificação de consistência atua preemptivamente sobre as ações de edição do usuário, impedindo que ele prossiga com o erro, oferecendo boas explicações sobre a verificação efetuada. Há a possibilidade de habilitar e desabilitar

este mecanismo, porém não foi encontrada uma forma de configuração em um nível de críticas, assim como sua edição e expansão.

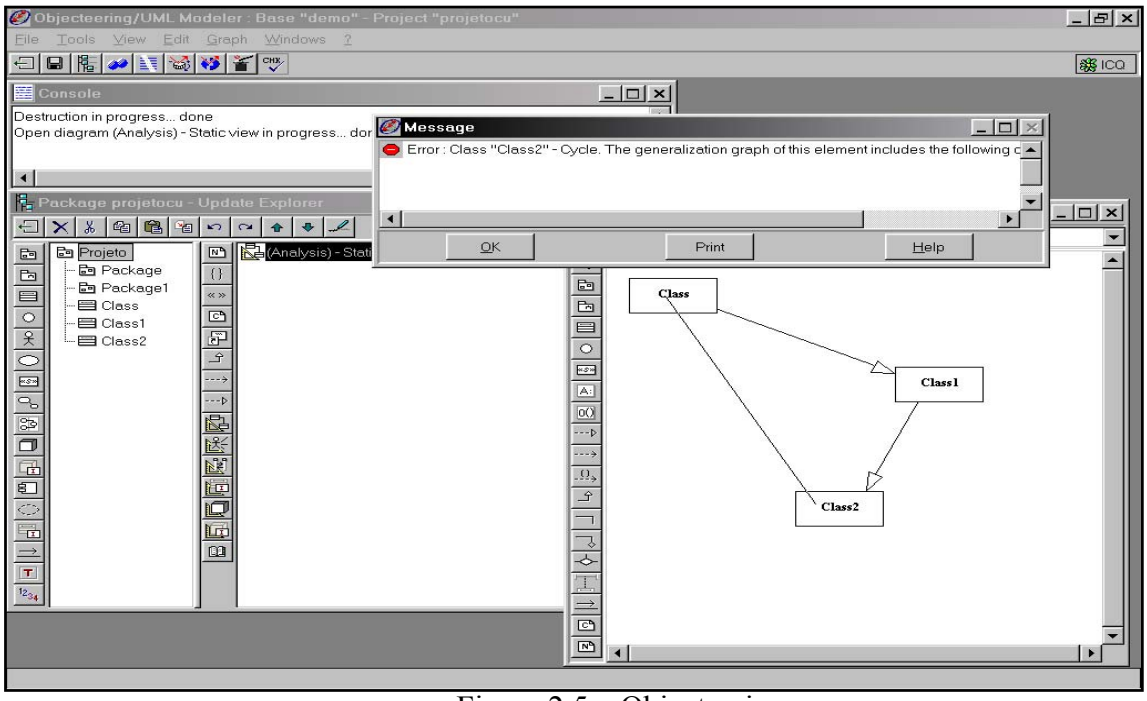

Figura 2.5 – Objecteering

2.5.2.4 – Rational Rose (Rational)

A ferramenta Rational Rose pode ser considerada como uma ferramenta comercial bem sucedida e aceita no mercado. Sua versão 2000 apresenta a criação e edição de modelos com praticamente todos os diagramas e elementos da linguagem UML. A sintaxe e notação das construções também se apresentam dentro da especificação, possibilitando facilidade para a construção de diagramas. A figura 2.6 exibe algumas janelas da ferramenta.

A ferramenta analisa as principais regras de casos de uso e algumas regras básicas de máquinas de estados. Entretanto, muitas regras estruturais e genéricas para as construções UML não são avaliadas, como para nomes de atributos, métodos e interfaces. Por outro lado, a ferramenta avalia questões de visibilidade entre pacotes.

Não foram encontradas formas de desativação completa da verificação, nem possibilidades de seleção, edição e expansão do conjunto de críticas utilizado. Entretanto, a atuação do mecanismo é feita de forma preemptiva às ações do usuário, de forma explicativa, oferecendo também fontes de ajuda sobre a UML. O mecanismo de

verificação não cria relatórios, mas pode ser feito de forma assíncrona para algumas poucas e específicas verificações, através da opção *Check Model*.

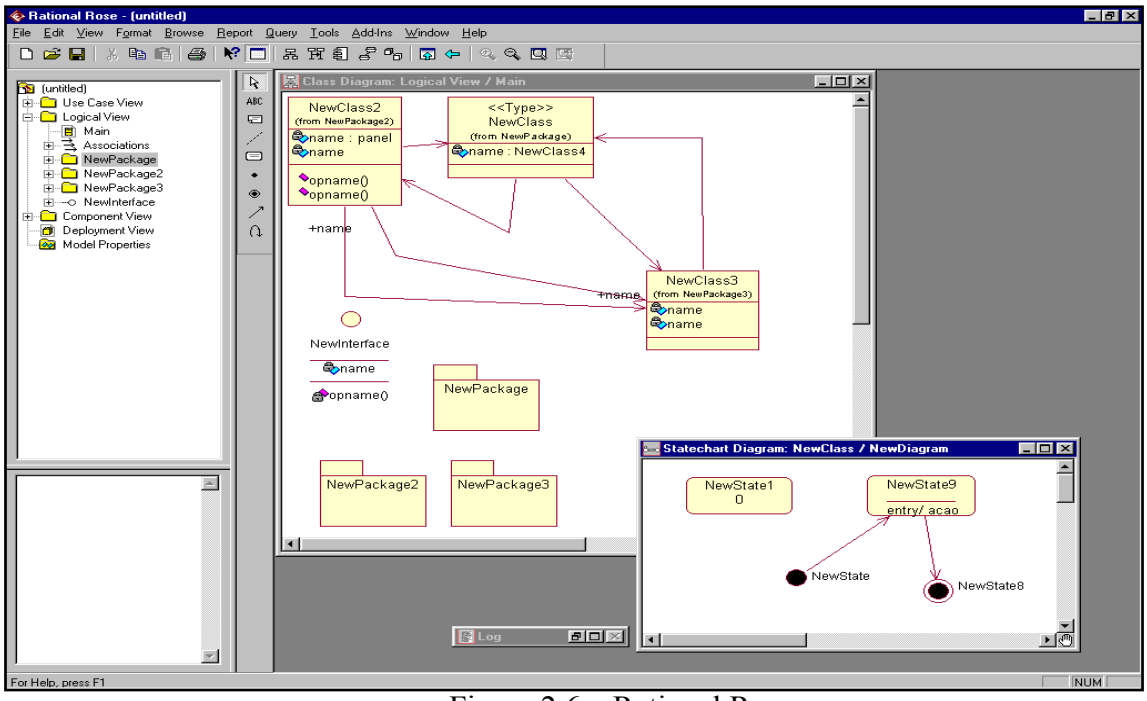

Figura 2.6 – Rational Rose

2.5.2.5 – SoftModeler (Softera)

Na ferramenta SoftModeler os diagramas da UML não são totalmente encontrados, seus elementos e propriedades apresentam-se quase completos em relação à especificação da linguagem. A utilização de classes e pacotes é restrita a diagramas, impedindo que um diagrama de classes contenha relações entre classes de diferentes pacotes, já que todo elemento pertence (e só é visível) a um pacote.

Assim como as demais ferramentas, as regras para casos de uso e regras básicas para máquina de estados são avaliadas. Poucas regras genéricas e do modelo de classes são verificadas como herança circular, agregação e nomes de atributos. A figura 2.7 exibe as janelas de trabalho na ferramenta, ilustrando um caso de inconsistência sendo tentada (herança circular) e a ação do mecanismo de verificação impedindo o prosseguimento.

O mecanismo para esta consistência não pode ser desativado, porém sua atuação é bastante transparente e simples ao usuário. Não há formas de ativar ou desativar críticas específicas, nem mesmo adicionar novos itens para verificação. O mecanismo não permite que o usuário prossiga com ações que causam erros e o impede de

prosseguir avisando-o, e também limita as ações que ele pode tomar, em determinados casos, que iriam gerar algum tipo de inconsistência. Não há criação de relatórios, porém a ferramenta conta com fontes de ajuda.

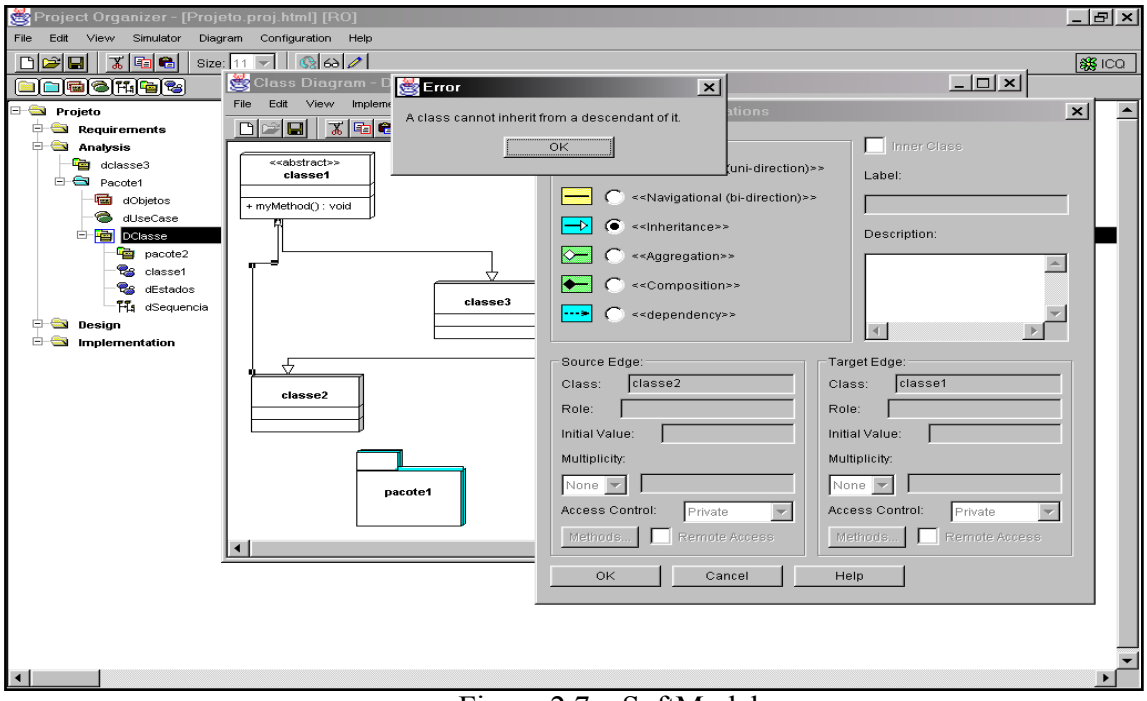

Figura 2.7 – SoftModeler

2.5.2.6 – Quadro Comparativo entre as Ferramentas

Com base nas descrições das ferramentas acima, é apresentado um quadro comparativo entre elas, segundo os fatores previamente sugeridos. Como podemos observar na tabela 2.1, nenhuma ferramenta apresenta-se completa através desta análise. Neste sentido, o próximo capítulo apresenta uma proposta para um mecanismo de verificação de consistência de modelos UML visando atender todos os fatores.

Quase todas as ferramentas apresentam bom desempenho em relação ao subconjunto da UML utilizado, com exceção para a SoftModeler. Na questão de configurabilidade, apenas a ferramenta ArgoUML se destaca em algumas situações, enquanto as demais não oferecem praticamente nenhuma forma de suporte com esta finalidade. Em termos de interação com usuário, as ferramentas se apresentam equilibradas, com destaque para Rose e SoftModeler em detrimento do ArgoUML.

Fator de Análise	Ferramenta	Argo	Magic	Object'ing	Rose	S'Modeler
Subconjunto da UML	Diagramas	☺	☺	☺	☺	☹
	Elementos	☺	☺	☺	☺	😐
	Propriedades	😐	☺	☺	☺	😐
Categorias Das Críticas	Classes e Core	😐	☹	😐	☹	☹
	Casos de Uso	☺	☺	☺	☺	☺
	Estados	😐	☺	😐	😐	😐
Configuração	Desativação	☺	☹	☺	☹	☹
	Por Crítica	☺	☹	☹	☹	☹
	Expansibilidade	☹	☹	☹	☹	☹
Interação com Usuário	Relatório	😐	☹	☹	😐	☹
	Ajuda UML	😐	☺	☹	☺	☺
	Preemptiva	☹	☺	☺	☺	☺
	Explicativa	☺	☹	☺	☺	☺

Tabela 2.1 – Comparação entre as Ferramentas

CAPÍTULO III
UM SISTEMA DE CRÍTICAS PARA UML

No capítulo anterior, foram introduzidos os principais conceitos relacionados à modelagem, assim como a importância da construção de modelos consistentes, utilizando-se uma notação padronizada. Destacamos, também, a utilização de sistemas de críticas como um mecanismo genérico para apoio ao trabalho de desenvolvimento de projetos. Neste capítulo, é apresentada uma proposta para um sistema de críticas que atue na verificação das regras de boa-formação da sintaxe e semântica estática sugeridas pela especificação da linguagem UML. As duas primeiras seções descrevem em maiores detalhes o subconjunto que é utilizado pela proposta, assim como o conjunto de regras que são verificadas. A terceira seção apresenta as principais características e comportamento esperados para a execução do sistema de críticas.

3.1 – UM SUBCONJUNTO DA UML

No capítulo dois, a UML foi descrita como linguagem padrão para a modelagem de sistemas de software. A seguir, é apresentada uma breve descrição dos principais elementos e diagramas que formam o subconjunto utilizado no escopo deste trabalho, de acordo com a versão 1.3 da especificação da linguagem UML (OMG , 1999).

3.1.1 – Mecanismos de Extensão, Especialização e Genéricos

Um elemento de modelagem pode ser estendido através dos mecanismos de extensibilidade e especialização, representados pelos estereótipos, propriedades e restrições. Os estereótipos são usados para criar uma classificação nova e superior sobre alguma construção existente, indicando diferenças de significado, restrições ou propriedades que não estão explícitas pelo elemento, qualificando-o. Propriedades

(*tagged values*) são pares atributo-valor que expressam alguma informação adicional sobre os elementos de modelagem. Restrições e comentários também atuam como mecanismo de extensão através do uso de linguagem arbitrária para adicionar nova semântica às construções existentes que devem ser satisfeitas. Geralmente, é utilizada a linguagem OCL para descrevê-las.

Um espaço de nome (*namespace*) representa um elemento que apenas contém um conjunto de outros elementos de modelagem, inclusive outros espaços de nome, porém um elemento só pode pertencer a um deles. A propriedade Visibilidade determina a visibilidade do elemento fora deste escopo. As possíveis opções de visibilidade para um elemento de modelagem são:

- **Privada** – Apenas o elemento e suas partes constituintes podem vê-lo.
- **Protegida** – Apenas os descendentes do elemento podem vê-lo.
- **Pública** – Qualquer elemento do espaço de nome pode vê-lo.

As construções representantes de espaços de nomes são os pacotes e os modelos em si. É possível criar permissões para visibilidade entre espaços de nomes através do uso de relações de dependência com estereótipo de importação ou acesso.

3.1.3 – Diagrama de Classes

Diagramas de Classe representam a estrutura estática mais importante da modelagem UML, sendo formados por classes de objetos com características e comportamento semelhantes e seus relacionamentos mais comuns, como associações, agregações, heranças e dependências.

O principal elemento de modelagem deste diagrama estrutural é a classe, que representa a descrição de um conjunto de objetos envolvidos no sistema. Objetos apresentam um estado, exibem algum comportamento bem definido e possuem uma identidade única que os diferenciam dos demais objetos. O estado de um objeto é caracterizado pelo seu conjunto de propriedades estáticas e seus valores dinâmicos. O seu comportamento é caracterizado pela atividade do objeto em função das mudanças de estados e troca de mensagens. Uma classe possui as seguintes características:

- **Ativa** – Especifica se os objetos de uma classe são responsáveis pelo seu próprio controle de execução ou estão subordinados a um objeto controlador.
- **Raiz** – Determina que a classe é raiz de generalização, não possuindo ancestrais.
- **Folha** – Determina que a classe é uma folha na árvore de generalizações, não podendo ter descendentes.
- **Abstrata** – Determina que a classe não possui instâncias, não podendo ser instanciada.
- **Visibilidade** – Conforme definido anteriormente.

As propriedades estáticas dos objetos são representadas pelos Atributos, enquanto sua atividade é representada pelos serviços (operações) que ela possui. Os atributos representam um conjunto de valores que o objeto pode assumir para uma determinada propriedade, e possuem as seguintes características:

- **Tipo** – Determina o classificador das instâncias dos valores, que pode ser uma classe, um tipo de dado primitivo ou uma interface.
- **Multiplicidade** – Determina quantas instâncias de valores um determinado atributo pode conter.
- **Valor Inicial** – Determina o valor do atributo quando o objeto é inicializado.
- **Escopo** – Determina se cada valor está relacionado a uma instância da classe ou se está relacionado diretamente à classe (todos os seus objetos).
- **Mutabilidade** – Determina se o valor do atributo pode ser alterado após a criação do objeto. As opções possíveis são mutável (*changeable*), congelado (*frozen*) e adicional (*addOnly*). Na primeira, não há restrições; na segunda, valores não podem ser alterados nem adicionados após a instanciação e inicialização dos valores; e na terceira, apenas é possível adicionar novos valores (multiplicidade maior que um) a um conjunto.
- **Visibilidade** – Conforme definido anteriormente.

As operações de uma classe representam um serviço bem definido e possuem as seguintes características:

- **Método** – Corresponde a implementação da operação.
- **Visibilidade** – Conforme definido anteriormente.
- **Abstrata** – Indica que a operação não possui método correspondente.
- **Folha** – Indica se a operação não pode ser sobrescrita por uma operação em uma classe descendente, ou seja, não é possível usar polimorfismo.
- **Raiz** – Indica se a operação pode herdar uma declaração de um ancestral ou não.
- ***isQuery*** – Indica se a execução causa mudanças de estado no sistema ou não.
- **Concorrência** – Determina a semântica de concorrência para chamadas passivas. As opções são seqüencial (*sequential*), restrita (*guarded*) ou concorrente (*concurrent*). Na primeira, as chamadas são feitas uma a uma; na segunda, múltiplas chamadas podem ser feitas, porém apenas uma estará ativa; e na terceira, múltiplas chamadas podem ser feitas e ativadas.
- **Parâmetros** – São argumentos utilizados na especificação da operação que representam valores que podem ser alterados, passados e retornados. Cada parâmetro possui um nome que o identifica, um tipo de classificador a que sua instância pertence, um valor e uma categoria que pode ser, basicamente, de entrada ou saída.

Um conjunto de operações não implementadas que definem um protocolo de comportamento de um elemento é representado por uma Interface. Estas podem ser usadas em relações de realização, indicando que algum classificador implementa suas operações definidas.

Um relacionamento de herança define um mecanismo de generalização vs. especialização, em que um elemento mais genérico é especializado por um outro elemento descendente. As heranças podem ser simples (apenas um ancestral direto) ou múltiplas (vários ancestrais diretos para um único elemento).

Um relacionamento de dependência expressa que a implementação de um cliente requer a existência de um ou mais fornecedores. Os possíveis tipos de dependência expressam permissões (acesso a elementos em diferentes espaços de nome), relacionamentos de uso (em que um elemento precisa de outro, como, por exemplo, variáveis locais e parâmetros instâncias de outros classificadores dentro de um classificador), abstrações (expressam relacionamentos entre diferentes níveis de

abstração de um mesmo conceito) e *bindings* (expressam relacionamentos entre *templates* e elementos gerados por eles).

Uma associação representa um relacionamento semântico entre dois classificadores (classes, interfaces, tipos de dados). Uma classe associativa é um elemento que se apresenta tanto como uma classe como uma associação. A representação da conexão da associação com um classificador específico é um fim de associação (*AssociationEnd*). Cada um deles possui as seguintes características:

- **Agregação** – Determina se a relação com o classificador é de agregação. As possíveis opções são: nenhuma (*none*), agregação (*aggregate*) e composição (*composite*). A primeira indica que a conexão não representa uma agregação; a segunda indica que o classificador relacionado representa um todo, enquanto o classificador da outra extremidade representa uma parte deste todo; e a terceira indica uma agregação em que as partes não existem sem o todo nem fazem parte de outro.
- **Mutabilidade** – Determina se uma instância do classificador conectado pode ser alterado pela extremidade oposta, conforme definido anteriormente.
- **Ordenação** – Determina se o conjunto de ligações da instância origem para o destino está ordenado.
- **Navegabilidade** – Determina se o classificador da extremidade é navegável a partir da extremidade oposta da associação.
- **Multiplicidade** – Determina o número de instâncias associadas a uma origem única dentro da associação.
- **Escopo** – Determina se o classificador representa uma instância ou um classificador, analogamente a definição anterior.
- **Visibilidade** – Define a visibilidade do fim de associação para o classificador da extremidade oposta, conforme definido anteriormente.
- **Papel** – Representa o papel da conexão da associação com um classificador destino, atuando como um pseudo-atributo para o classificador origem, na outra extremidade da associação.

3.1.3 – Diagrama de Casos de Uso

Os diagramas de casos de uso foram sugeridos pelo método OOSE de Ivar Jacobson (Jacobson *et al*., 1992) e são uma forma de representar os requisitos funcionais de um sistema em um alto nível de abstração, sem considerar os objetos que fazem parte do sistema, a estrutura de classes e detalhes implementacionais. Neste sentido, indicam um caminho para a análise e compreensão do sistema como um todo sem a preocupação com o seu funcionamento interno. Ao utilizar notação simples e linguagem natural, esses diagramas criam uma forma clara e simples de comunicação e especificação entre usuários e membros da equipe de desenvolvimento. Os elementos participantes são os casos de uso, atores e seus relacionamentos.

Um caso de uso é a especificação de um determinado serviço esperado pelo sistema, composta por uma seqüência de ações e variações que representam os cenários de interação a partir dos eventos externos iniciais. Cada cenário representa uma seqüência de realização de um caso de uso determinado. Cada caso de uso pode ser posteriormente refinado em uma ou mais classes de objetos e um conjunto de colaborações entre eles para a realização da funcionalidade esperada. É possível a utilização de pontos de extensão e inclusão de referências a outros casos de uso dentro de um determinado caso de uso. Esses relacionamentos são expressos por ligações especiais com estereótipo de uso e extensão, além dos relacionamentos de herança que também podem fazer parte deste diagrama.

Um ator é a representação de uma entidade do mundo externo que atua diretamente sobre o sistema, interagindo com ele de acordo com um papel definido e não apenas um indivíduo. Um ator pode se comunicar com casos de uso, através de uma ligação de comunicação, que reflete sua participação neste caso. É também possível o uso de heranças entre atores.

3.1.4 – Diagrama de Estados

Os diagramas de estados são utilizados para representação do comportamento discreto de um sistema através de máquinas de estado finitas, que podem ser ilustradas como dispositivos que recebem um número finito de estímulos como entrada e conseguem processá-las e oferecer respostas. Desta forma, é possível modelar o

comportamento de vários elementos através dos seus estados e os estímulos (transições) que participam da máquina de estados. Este diagrama é composto principalmente por estados, pseudo-estados e transições.

Um evento é uma ocorrência observável que ocorre em um instante de tempo. Os tipos de evento são Sinal (*signal*), Chamada (*call*), Tempo (*time*) e Mudança (*change*). Os eventos de chamada estão relacionados com operações.

Uma ação é a representação da abstração de um procedimento computacional através de envio de mensagens, que causa mudanças no estado do modelo. As ações podem ser de Construção (*create*), Destruição (*destroy*), Chamada (*call*), Retorno (*return*), Envio (*send*), Término (*terminate*) ou Não-Interpretada (*uninterpreted*). A cada ação de criação corresponde a instanciação de uma classe, enquanto a cada chamada corresponde uma operação e a cada envio, um sinal. Um sinal representa um estímulo assíncrono, uma comunicação entre duas instâncias de elementos.

Um estado é uma condição ou situação temporária de um objeto que satisfaz alguma condição, espera algum evento ou responde a algum estímulo como resultado de uma evolução de seqüência de estímulos passados. Um estado pode ser simples, final ou composto, neste caso, possui subvértices (estados ou pseudo-estados). Um estado final representa o fim de execução de um estado composto ou de uma máquina de estados. Todo estado possui um compartimento de eventos que são executados quando se entra ou sai do estado ou durante sua permanência. Há também uma lista de transições internas, ou seja, transições cujo efeito não causa a saída do estado atual. Pseudo-estados representam vértices especiais como estados iniciais, vértices de junção (*join*) e disjunção (*fork*), entre outros.

Uma transição é um relacionamento entre dois vértices (estados ou pseudo-estados) que indica a mudança de um estado origem para outro destino em resposta a uma instância de um evento. A cada transição há um efeito ou ação associada, controlada ou não por condições de guarda.

3.1.5 – Diagrama de Seqüência

Os diagramas de seqüência são um tipo de diagrama de interação, que exibem o comportamento dinâmico do sistema, focalizando a interação entre objetos através da representação da troca de mensagens entre eles para a realização de determinada tarefa ou funcionalidade especificada em um caso de uso. Os diagramas de colaboração

diferem do de seqüência apenas em função da sua organização e do enfoque desejado. Enquanto os primeiros favorecem a visão simples e plana dos objetos e suas interações, os outros favorecem uma visão seqüencial de como as coisas acontecem ao longo do tempo. Os principais participantes do diagrama de seqüência são objetos e mensagens.

Um objeto, conforme visto anteriormente, é uma instância de uma classe, que possui características e comportamento (serviços) bem definidos e representa um papel na colaboração sendo modelada. Cada objeto possui uma linha de vida que indica o ciclo de vida do objeto durante uma interação.

Mensagens representam uma comunicação entre duas instâncias em uma interação. Cada mensagem está associada a um papel remetente e um destinatário (instâncias de classificadores definem papéis em uma colaboração), além de possuir uma ação que envia um estímulo (um sinal ou uma chamada de operação) . Há estímulos especiais que definem a criação ou destruição de objetos em um diagrama de seqüência. Os estímulos também podem ser assíncronos ou síncronos e a ordem vertical das mensagens indica como elas se sucedem ao longo do tempo.

3.2 – REGRAS DE BOA-FORMAÇÃO DA UML

As categorias de críticas são representadas pelos conselhos relacionados às regras que elas contém. Uma completa descrição de todas as regras de boa-formação da semântica estática da UML pode ser encontrada na especificação da linguagem (OMG, 1999).

3.2.1 – Considerações sobre Nomes

Atuam na verificação de questões relacionadas aos nomes dos elementos para modelagem estrutural. Apenas uma regra está definida.

- **Nome Válido** - Um elemento de modelagem deve ter nome iniciado por letras ou caracteres '$' ou '_', e ser formado por apenas letras, dígitos, '$' e '_', sem espaços em branco.

3.2.2 – Regras de Modelo de Casos de Uso

Atuam na verificação de questões sobre a semântica estática dos Modelos de Caso de Uso, segundo as regras de boa-formação da UML.

- **Ligações para Atores** – Atores só podem ter associações binárias com Casos de Uso, Subsistemas e Classes. É permitida a herança entre Atores.
- **Ligações para Casos de Uso** – Casos de Uso não possuem associações especificando a mesma entidade. As associações podem ser de inclusão ou extensão. É permitida a herança entre Casos de Uso.

3.2.3 – Regras de Máquina de Estados

Atuam na verificação sobre diagramas de estados.

- **Estado Inicial Único** - Uma máquina de estados só possui um estado inicial.
- **Segmento de Fork** - Um segmento de *fork* deve sempre se destinar a um estado, e ele não contém guarda ou evento.
- **Segmento de Join** - Um segmento de *join* deve sempre se originar de um estado, e ele não contém guarda ou evento.
- **Estado Final** - Não há transições saindo de um estado final.
- **Estado Inicial** - Não há transições chegando em um estado inicial e há apenas uma transição saindo.
- **PseudoEstados** - Transições saindo de um pseudoestado não contém evento.
- **Vértice Join** - Um vértice de junção deve ter no mínimo uma transição saindo e apenas uma entrando.
- **Vértice Fork** – Um vértice de disjunção deve ter apenas uma transição entrando e no mínimo uma saindo.

3.2.4 – Regras de Modelos de Classes

Atuam na verificação das regras de boa-formação da semântica estática dos modelos estruturais segundo a UML.

- **Assinatura de Operações** - Nenhuma operação pode ter assinatura equivalente a outra em uma classe.
- **Type** (estereótipo de Classe) - Uma classe Type não contém métodos, o pai de uma classe Type deve ser uma classe Type.
- **Implementation Class** (estereótipo de classe) - O pai de uma Implementation Class deve ser uma Implementation Class.
- **Composição -** Uma instância não pode pertencer por composição a mais de uma instância de composição.
- **Herança de Folha -** Não é possível generalização a partir de um elemento folha.
- **Interface** - Uma interface só contém operações, e estas devem ser públicas.
- **Associação** - Uma associação deve ter uma única combinação de nome e classes associadas.
- **Atributo** - O nome de um atributo deve ser único na classe e diferente de qualquer nome de papel de fim de associação oposto.
- **Parâmetro -** Todos os parâmetros devem ter nome único em uma operação.
- **Fim de Associação** - Os nomes de fins de associações opostos devem ser diferentes; o nome de um fim de associação deve ser diferente de qualquer atributo da classe oposta.
- **Agregação** - No máximo, apenas um fim de associação pode ser uma agregação ou uma composição.
- **Navegabilidade** - Uma associação não é navegável a partir de uma interface ou de um DataType.

3.2.5 – Considerações Genéricas e sobre Visibilidade

- **Herança Circular** - Não é permitida herança circular.
- **Visibilidade Herança** - Um elemento só pode generalizar outro elemento de mesmo categoria.
- **Fim de Associação** - As classes relacionadas em uma associação devem ser incluídas no espaço de nomes de associação.
- **Ligações entre Pacotes** - Ligações entre pacotes são apenas ligações de herança e dependência.

- **Nome Único** - Nenhum elemento pertencente ou importado, que não seja associação, tem o mesmo nome que outro em um espaço de nomes; nenhuma associação tem a mesma combinação de nome e classes associadas em um espaço de nomes.

3.3 – UM SISTEMA DE CRÍTICAS PARA A UML

A proposta deste trabalho é a construção de um sistema de críticas que atue somente no nível de verificação de consistência de modelos construídos segundo a linguagem de modelagem UML. Este sistema deve constituir um mecanismo integrante de uma ferramenta de modelagem. O mecanismo está baseado nas principais características identificadas em sistemas de críticas, como visto no capítulo anterior, e nos critérios utilizados para a avaliação das ferramentas existentes, no contexto de críticas de modelos UML.

Uma das questões mais importantes para a implementação do sistema de críticas proposto está relacionada a organização e representação do conhecimento pelo ambiente de modelagem. A partir da estrutura de representação interna das construções, são implementadas as regras de verificação. Os principais requistos para o sistema de críticas proposto são:

- possibilidade de ativação/desativação completa do mecanismo;
- possibilidade de ativação/desativação ao nível de críticas e regras;
- resposta preemptiva durante a modelagem (*reflection-in-action*);
- possibilidade de criação de relatórios de verificação de todos os modelos;
- existência de fontes de ajuda independentes sobre a UML e suas regras;
- representação e armazenamento das críticas e regras de forma flexível, independente do ambiente e facilmente modificável;
- servir como base para novos tipos de verificações, além das regras de boa-formação da UML.

A categorização de diversas regras ou heurísticas afins em conjuntos bem definidos de críticas, permite que o usuário tenha controle direto sobre a ativação destas categorias disponíveis, assim como também torna flexível a expansão com novas regras

e categorias de críticas, não necessariamente relacionadas a UML. É válido perceber que a granularidade não necessariamente obedece ao mapeamento de uma regra para uma categoria, em função da possível grande quantidade de regras propostas. Este trabalho, a princípio, envolve apenas o subconjunto e regras da linguagem UML conforme previamente descritos neste capítulo.

Como a atuação do mecanismo de críticas está intimamente relacionada ao funcionamento de uma ferramenta de modelagem, podem ser identificados três pontos principais de interação entre eles:

- na inclusão de um novo elemento de modelagem;
- na remoção de um elemento de modelagem;
- na edição das propriedades de um elemento de modelagem.

O mecanismo de criticas deve verificar se alguma das regras ativas (previamente configuradas pelo usuário) foi violada, ou se alguma possível heurística indica uma boa sugestão para melhoria do projeto. O resultado da ação de verificação de cada categoria de críticas ocorre no momento em que o usuário efetuou alguma das ações de modelagem listadas acima, impedindo o seu prosseguimento inconsistente. Em adição a atuação preemptiva, o mecanismo deve permitir a criação de relatórios de avaliação completa de todos os modelos, quando solicitado pelo usuário; como também a possibilidade de oferecer fontes de ajuda com conhecimento mais detalhado sobre a questão que motivou a ocorrência da crítica. Toda interação com o usuário deve ser clara e simples, permitindo que ele obtenha todas as informações para que possa evoluir de maneira precisa o projeto e aumentar seu aprendizado.

O diagrama de classes ilustrado na figura 3.1 apresenta a estrutura básica do mecanismo de críticas, em nível de projeto. Um gerente de críticas é a entidade responsável pelo controle de execução do mecanismo, delegando a verificação individualmente para cada uma de suas críticas cadastradas ativas. Cada crítica, por sua vez, deve conter um nome genérico e um conjunto de regras que caracterizam aquela categoria de crítica, além do agente que irá realizar a verificação. Um agente é uma classe que contém a implementação algorítmica de todas as regras pertencentes a uma crítica. Cada regra é a definição de uma heurística ou regra de consistência que deve ser verificada, contendo um nome, conselhos e correções associadas e o método que

implementa a sua verificação. Relacionado ao gerente de críticas, existe um objeto especial responsável por carregar e instanciar as regras e críticas.

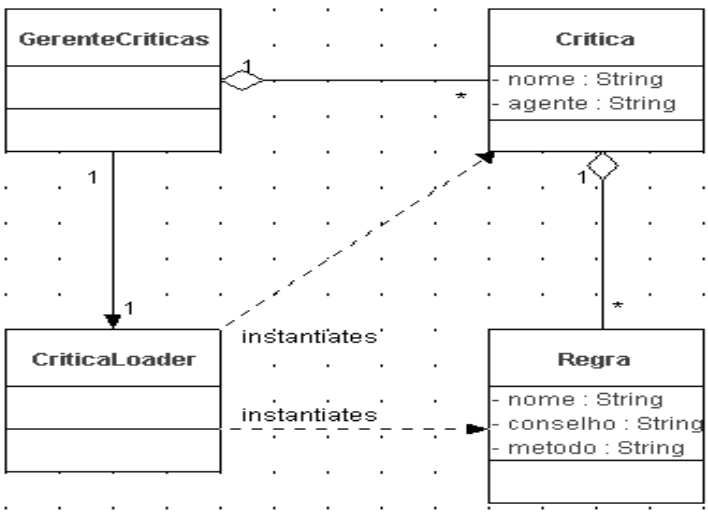

Figura 3.1 - Diagrama de classes básico do sistema de críticas

No próximo capítulo, veremos como a proposta aqui sugerida de um sistema de críticas para a UML foi implementada como uma ferramenta de suporte a uma infra-estrutura de reutilização baseada em modelos de domínio, denominada Odyssey.

CAPÍTULO IV
O SISTEMA DE CRÍTICAS NO ODYSSEY

No capítulo anterior, foi discutida uma proposta para um sistema de críticas que atue sobre ferramentas de modelagem. Este capítulo apresenta uma implementação dos requisitos discutidos sobre sistemas de críticas, dentro de uma infra-estrutura de reutilização baseada em modelos de domínio. As três primeiras seções apresentam a infra-estrutura Odyssey, como são organizadas as informações semânticas dos modelos construídos pela ferramenta e como essas informações são usadas pelo editor de diagramas na construção dos modelos, respectivamente. O restante do capítulo descreve como os principais requisitos desejados para a proposta do sistema de críticas são implementados nesta infra-estrutura.

4.1 – A INFRA-ESTRUTURA ODYSSEY

Como visto anteriormente, um sistema de críticas é considerado de grande importância no contexto do desenvolvimento baseado em componentes e em modelos de domínios. Um dos requisitos para ambientes de desenvolvimento de software baseados em modelos de domínio é a existência de mecanismos que atuem na detecção e aconselhamento sobre situações de erros e inconsistências assim que elas surgem, permitindo que o usuário tenha a possibilidade de configurar como se processará esta intervenção, visando ajudá-lo na decomposição da sua tarefa e na determinação de questões relevantes (Lima e Werner, 1998).

O Odyssey é uma infra-estrutura de reutilização que oferece ferramentas para apoio automatizado do desenvolvimento para reutilização e também do desenvolvimento com reutilização. A infra-estrutura suporta as atividades de Engenharia de Domínio definidas por um processo próprio chamado Odyssey-DE (Braga, 2000), assim como as de Engenharia de Aplicação, o Odyssey-AE (Miller, 2000). A figura 4.1 exibe as características arquiteturais fundamentais da infra-estrutura.

A infra-estrutura Odyssey é composta por ferramentas que procuram automatizar as diversas etapas definidas pelo Odyssey-DE e Odyssey-AE. Podemos citar ferramentas como as de captura de conhecimento de domínios (Zopelari, 1998); documentação de componentes (Murta, 1999); especificação e instanciação de arquiteturas específicas de domínios (Xavier, 2001); planejamento e análise de risco (Barros, 2000); camada de mediação e navegador inteligente (Braga, 2000); gerador de código executável (Werner *et al*., 2000); futuras ferramentas para acompanhamento de processos (Murta, 2000) e apoio a engenharia reversa (Veronese e Netto, 2001); e, finalmente, a ferramenta de diagramação UML, objeto de atenção deste trabalho. Toda esta infra-estrutura é implementada utilizando-se a linguagem JAVA (SUN, 2001).

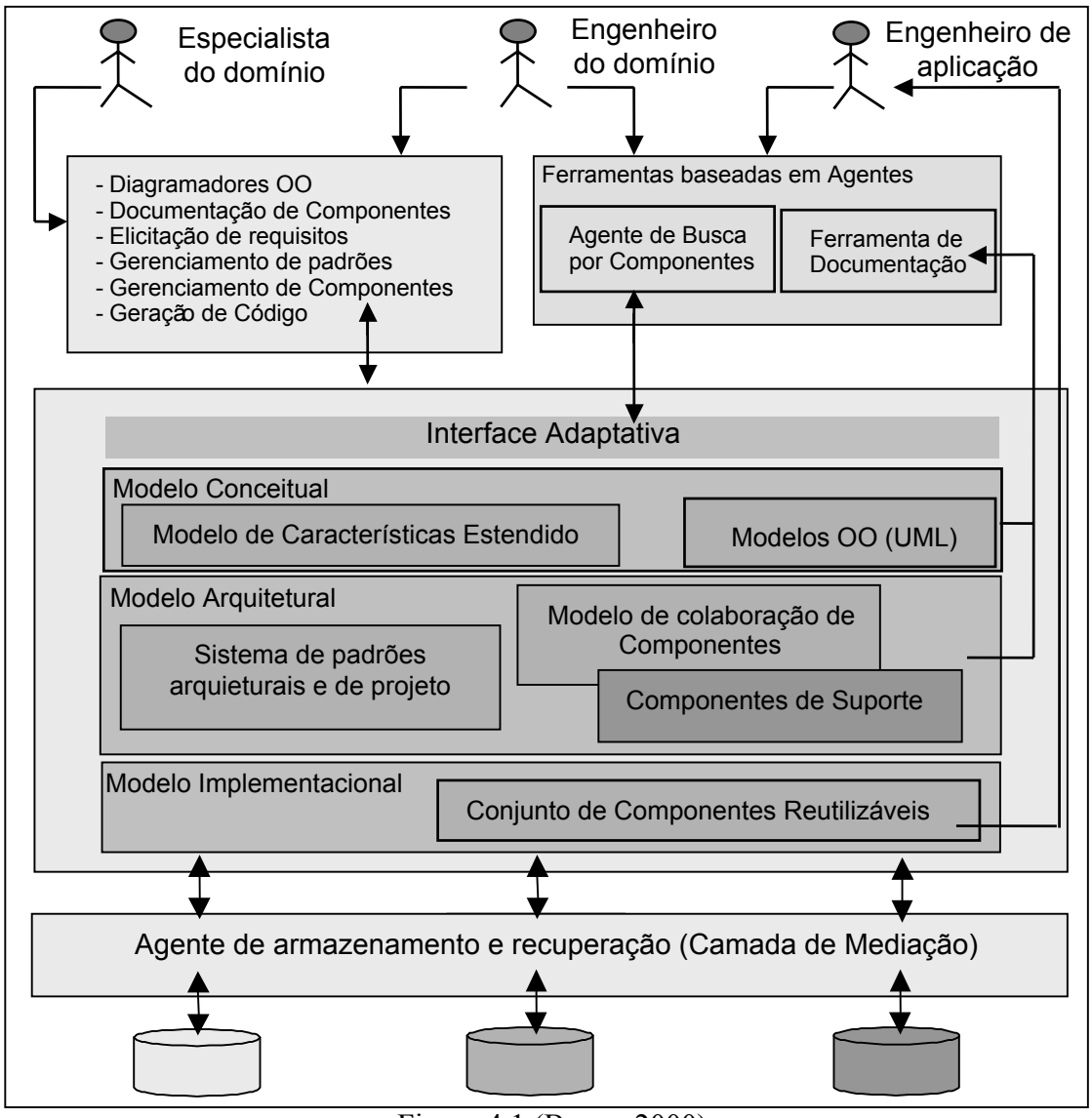

Figura 4.1 (Braga, 2000)

O editor de diagramas do Odyssey é uma ferramenta que permite a construção de modelos segundo um subconjunto da linguagem UML, conforme foi especificado neste trabalho. Além destes modelos da UML, é gerado um modelo de características (*features*) estendido com duas visões: funcional e conceitual, com o objetivo de capturar os principais conceitos e funcionalidades de domínios em um alto nível de abstração (Braga, 2000) (Miller, 2000) (Xavier, 2001). Os diversos modelos possuem entre si a capacidade de rastreabilidade, para facilitar o entendimento do domínio e seus componentes.

Modelos de características foram propostos pelo método FODA (Kang *et al*, 1990), apresentando uma estrutura hierárquica que permite a modelagem dos serviços no contexto de domínios, suas similaridades e diferenças, facilitando a identificação de características para reutilização. O modelo de características do Odyssey é uma extensão das notações apresentadas no FODA (Kang *et al.*, 1990) e FODACom (Griss *et al.*, 1998). O objetivo do modelo de características estendido é ser uma taxonomia detalhada do domínio, com uma melhor apresentação visual e os conceitos de múltiplos níveis e visões. O modelo de funcionalidades apresenta o relacionamento entre as principais funcionalidades do domínio, enquanto o de conceitos apresenta os conceitos do domínio e os seus relacionamentos (Braga, 2000). Maiores detalhes sobre esse modelo podem ser encontrados em Braga (2000) e Miller (2000).

4.2 – ESTRUTURA SEMÂNTICA

A estrutura semântica tem importância fundamental para a implementação da proposta. Com base nela são implementados os algoritmos que verificam se as condições determinadas pelas regras são satisfeitas. Vários tipos de representação interna de informação podem ser usados em ferramentas, como por exemplo XMI (OMG, 1999) e linguagens formais (Vasconcelos, 2000). No caso da infra-estrutura Odyssey, a informação utilizada é organizada hierarquicamente através de uma árvore semântica de objetos. A figura 4.2 exibe o diagrama das principais classes que formam a estrutura semântica do Odyssey.

Todo objeto da árvore semântica é chamado de *ModeloAbstrato*, e representa um elemento de modelagem. A partir de um destes elementos, é possível percorrer a árvore hierarquicamente e as relações entre os objetos, a fim de obter as informações

necessárias. Por exemplo, a partir de um objeto representando uma ligação, é possível descobrir quem são os objetos que representam a origem e o destino da ligação.

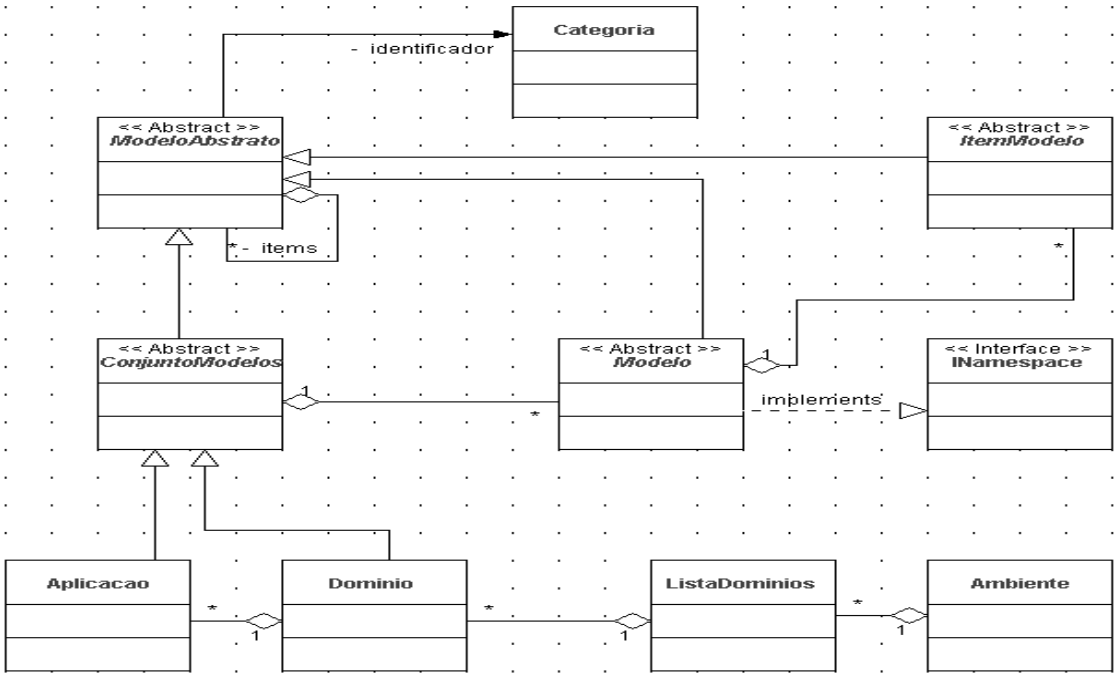

Figura 4.2 – *Kernel* da representação interna

A árvore semântica é organizada através de categorias de modelos (*Modelo*). As categorias de modelos definidas no Odyssey são de contextos, *features* (características), casos de uso e classes. A raiz da árvore, portanto, é um conjunto de modelos (*ConjuntoModelos*) que representa um domínio (*Dominio*) ou aplicação (*Aplicacao*) sendo modelado. Cada modelo é composto por diversos itens de modelagem (*ItemModelo*), que representam pacotes, diagramas, nós e ligações específicos à categoria do modelo.

Todo objeto de modelagem está associado a uma única categoria, e cada categoria possui uma representação padronizada (*RepresentacaoCategoria*) através de painéis de exibição e edição das suas características (*PainelEdicaoCategoria*). Uma categoria é um identificador existente em cada objeto do Odyssey que o associa a uma das construções da linguagem de modelagem utilizada, ou seja, elementos da UML e elementos próprios do Odyssey (*features* e contextos). Neste sentido, exemplos de categorias são classe, contexto, estado, *feature*, ligação de associação, etc.

A figura 4.3 mostra um detalhamento da estrutura. Observe pelas figuras 4.2 e 4.3 que tanto *Modelo* quanto *NoPacote* são espaço de nomes (*INamespace*). Todos os nós específicos aos modelos herdam de *No* (*NoClasse*, *NoUseCase*, etc), assim como todos os diagramas (*DiagramaClasse*, *DiagramaEstado*, etc) herdam de *Diagrama* e as ligações (*LigAssociacao*, *LigTransicao*) herdam de *Ligacao*. A próxima seção mostra como construir modelos através dos elementos semânticos do Odyssey, através da ferramenta diagramador.

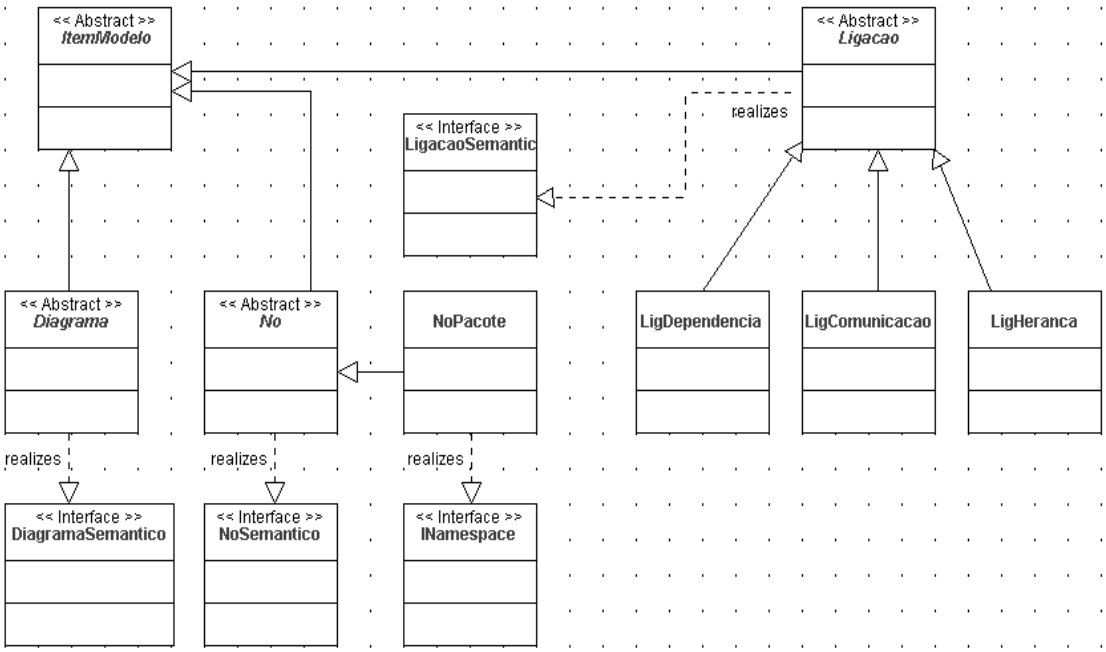

Figura 4.3 – Detalhamento da estrutura semântica

4.3 – ESTRUTURA DO DIAGRAMADOR

O editor de diagramas é a ferramenta em destaque para a implementação do sistema de críticas proposto. Como visto no capítulo anterior, a atuação do mecanismo de crítica pode ser induzida em três momentos: inclusão de novo elemento, edição ou remoção de algum elemento de modelagem, ou seja, um *ModeloAbstrato*. O sistema implementado, entretanto, trabalha apenas sobre objetos que sejam nós e ligações – *NoSemantico* e *LigacaoSemantica*.

A figura 4.4 mostra a janela da ferramenta de diagramação do Odyssey. A esquerda, está localizada a árvore de objetos de modelagem semânticos instanciados, de acordo com a hierarquia vista na seção anterior. A direita é exibido o painel de edição da categoria que atualmente está selecionada na árvore. Se o objeto for um nó (ligações

não são visíveis pela árvore), são apresentados painéis para exibição e edição das propriedades do nó em questão. Se o objeto for um diagrama, é apresentado o desenho dos nós e arestas contidos naquele diagrama. Na parte superior da janela, há a barra de ferramentas que auxilia o usuário nas tarefas de modelagem. Esta barra, muda dinamicamente, conforme a categoria do item selecionado na árvore, ou seja, há ações específicas para cada categoria ou conjunto de categorias de objetos do Odyssey.

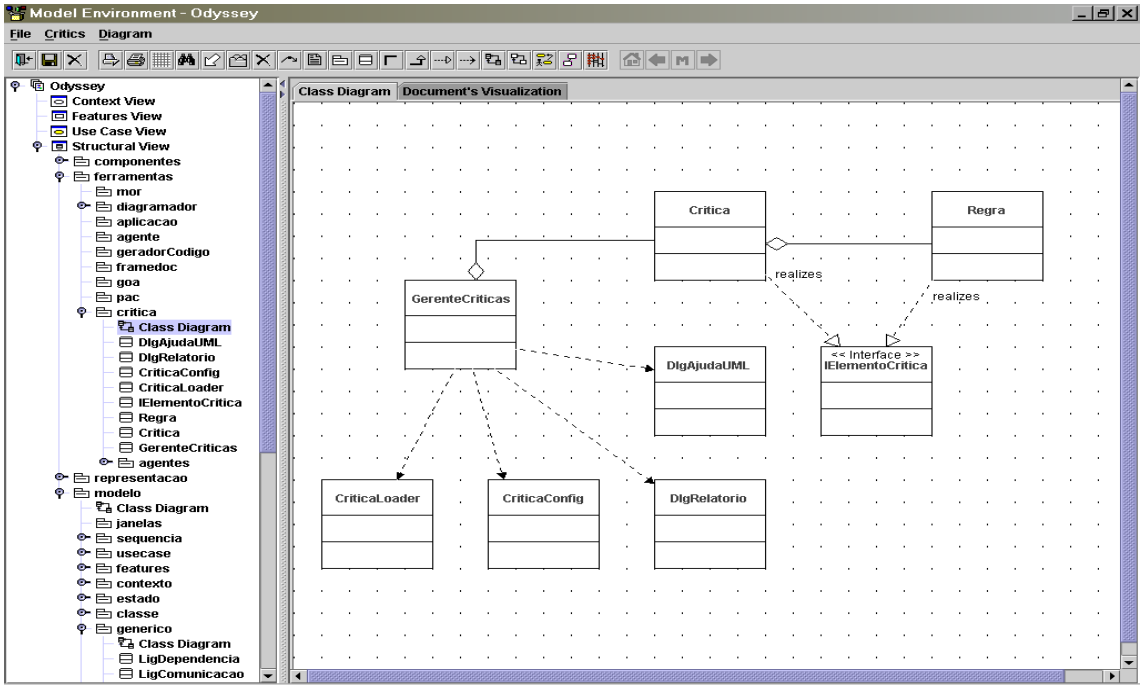

Figura 4.4 – Janela da ferramenta de diagramação

No Odyssey, além da estrutura semântica de elementos, há a estrutura léxica formada pelos diagramas específicos a cada modelo. Para todo item de modelo semântico, correspondem zero ou vários itens léxicos, nós e arestas que formam os desenhos dos diagramas, e um diagrama semântico contém apenas um diagrama léxico. As categorias de diagramas (*DiagramaClasse*, *DiagramaSequencia*, etc) são representadas por um painel específico, chamado *PainelDiagramador*. A função deste painel é apresentar o desenho do diagrama léxico, ou seja, seus nós e arestas léxicos que sabem se desenhar de acordo com a notação determinada pela especificação da linguagem UML. Cada um destes itens, por sua vez, está associado a um único item semântico, seja nó ou ligação, e todas as suas propriedades. A figura 4.5 mostra o diagrama com as principais classes da estrutura do editor de diagramas.

Para o funcionamento do sistema de críticas, é importante saber quem é responsável e como é o funcionamento da edição dos diagramas, permitindo inserção, edição e remoção de seus itens. Tal conhecimento é a implementação dos três casos em que o sistema de críticas é posto em execução, e estas tarefas estão sob responsabilidade de objetos agentes de diagramação. Todo diagramador específico possui um único agente ativo em um determinado instante, dentre vários agentes que ele pode conter. Os três tipos de agente são de edição (*AgenteDiagramacaoEdicao*), inserção de arestas (*AgenteDiagramacaoAresta*) e inserção de nós (*AgenteDiagramacaoNo)*. O único agente comum a todos os diagramas, é o de edição, que tem a responsabilidade de trabalhar sobre os elementos do diagrama (deslocamento, redimensionamento, etc) e também invocar o mecanismo de edição de um determinado objeto selecionado.

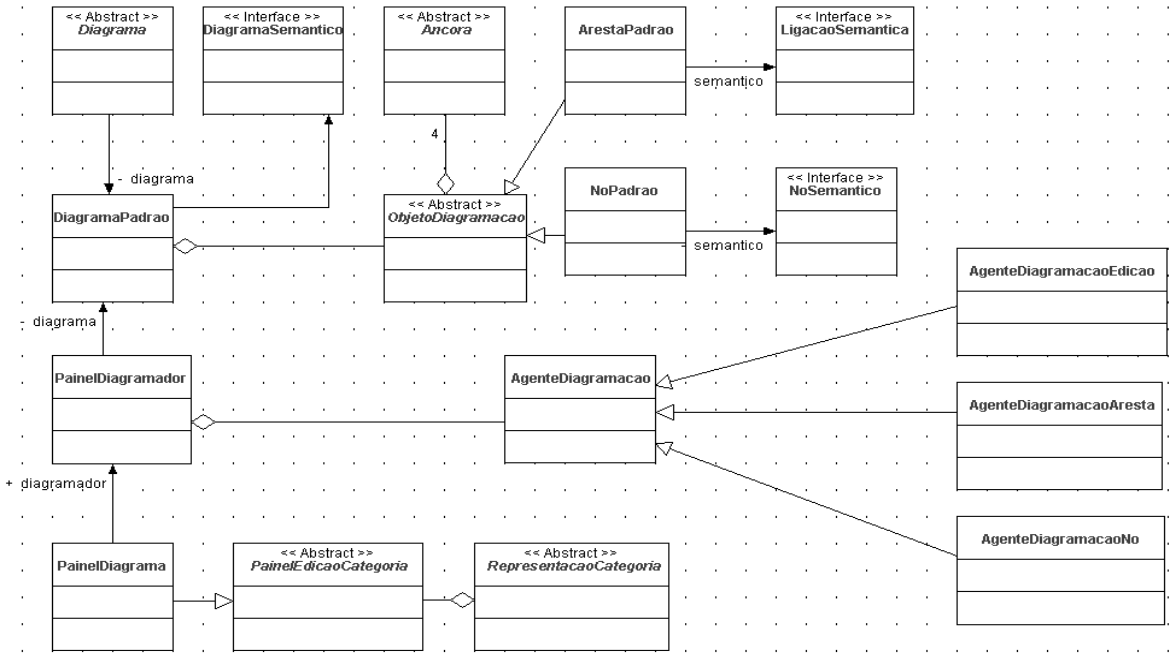

Figura 4.5 – Principais classes do editor de diagramas

Quando um elemento é selecionado na árvore semântica ou selecionado através do seu correspondente léxico em um diagrama para ser editado, é exibida a representação da categoria do elemento, formada por alguns painéis padronizados (*templates*) vazios que são preenchidos automaticamente com os dados específicos do objeto semântico selecionado. Após a edição dos campos deste painel, seja selecionando outro elemento na árvore ou fechando a janela de edição, os dados novos são capturados e atualizados no item semântico que estava sendo editado. Neste momento, é invocada a

execução do mecanismo de críticas, pois a alteração das propriedades de um elemento (por exemplo, o nome de um atributo quando se está editando uma classe) pode causar inconsistências ou representar um momento razoável para sugestões de melhorias de projeto.

Além da edição de elementos, o agente de edição também é responsável por capturar eventuais eventos de remoção. Outra forma possível de remover um elemento, é através de uma ação específica do ambiente de modelagem, que remove o elemento atualmente selecionado na árvore. A diferença entre as duas formas se baseia no fato de que a remoção pela árvore atua diretamente sobre o elemento semântico, enquanto que pelo diagramador atua sobre o elemento léxico. Ao apagar um item léxico, seu correspondente semântico permanece intacto, com exceção das ligações, que oferecem a possibilidade de remoção semântica automática quando for feita a remoção léxica. O mecanismo de críticas entrará em execução sempre que uma remoção de elemento semântico for feita, significando que um elemento pai (na árvore) perderá um filho. Um bom exemplo de inconsistência gerada a partir de uma remoção é o caso da ligação de dependência. Ligações de dependência são a forma natural de criação de visibilidade entre espaços de nomes, o que permite que elementos de um pacote consigam ser visíveis por outro pacote. Ao remover uma ligação de dependência (semântica), podemos estar quebrando este vínculo de visibilidade necessária para a boa formação de um diagrama.

Além do agente de edição, cada diagramador específico possui alguns agentes para inserção de nós e arestas - *AgenteDiagramacaoNo* e *AgenteDiagramacaoAresta*. Para cada categoria distinta de ligação e nó que um determinado diagrama aceita, há um destes agentes responsável pela criação de elementos léxicos a partir do diagramador. É possível criar um item léxico selecionando um item semântico a partir de uma lista de itens semânticos pertencentes a categoria desejada, e é também possível criar um elemento léxico de um semântico ainda inexistente. Neste caso, o item semântico é criado automaticamente no instante da criação léxica. Outra forma possível de criar elementos semânticos é através de ações específicas do ambiente de modelagem para a árvore de modelos abstratos. Para cada item selecionado na árvore, o ambiente de modelagem disponibiliza um conjunto de ações válidas para aquele item. Por exemplo, para um modelo de classes, é possível criar novos pacotes, novas classes e diagramas de classe; para uma classe é possível abrir um diagrama de estados ou de seqüência. Neste

sentido, na circunstância de um novo nó semântico ser inserido em um determinado pai (na árvore), o mecanismo de críticas deverá ser executado. Um bom exemplo de inconsistência deste caso é a criação de um elemento com mesmo nome de outro elemento visível pelo espaço de nomes do elemento pai.

4.4 – GERENTE DE CRÍTICAS

Nesta seção descrevemos o funcionamento do sistema de críticas, uma vez que já sabemos como ele é ativado pelas ações da árvore semântica ou pelo editor de diagramas.

4.4.1 – Cadastramento de Regras

As regras representam o centro funcional do sistema de críticas. São elas que verificam se as condições esperadas são satisfeitas ou se alguma melhoria pode ser sugerida ao projetista. Sendo assim, para garantir transparência e flexibilidade ao mecanismo de verificação, é importante ter uma boa representação para a estrutura de críticas, assim como sua manipulação.

Como visto no capítulo anterior, o sistema de críticas é composto por objetos da classe *Critica*. Cada crítica, por sua vez, é formada por nome, agente e várias regras. Cada regra é formada por um nome, conselhos e um método que contém a forma algorítmica correspondente a sua verificação. Criar objetos de crítica e regra*s* amarradas ao código fonte oferecem pouca flexibilidade ao sistema. O uso de um arquivo com formatação especial é uma boa alternativa, pois permite uma fácil reconfiguração das características das críticas e regras, como também uma fácil adição e remoção desses elementos do sistema. O arquivo deve conter todas as informações sobre as críticas e regras. Desta forma, para modificar o conselho associado a uma regra, por exemplo, é necessário apenas modificar o texto relacionado ao conselho da regra em questão no arquivo descritor, sem nenhuma mudança de código fonte. Da mesma forma, se incluirmos novas entradas no arquivo, estas são automaticamente reconhecidas pelo mecanismo de críticas durante o carregamento do arquivo. Se quisermos remover regras ou críticas, basta remover as entradas correspondentes, também pelo arquivo descritor. Foi escolhida a XML (*Extended Markup Language*) para representar um arquivo descritor dos elementos do sistema de críticas. A figura 4.6 mostra um trecho do arquivo sendo utilizado. O uso de XML (XML, 2000) garante padronização e representação

estruturada do conhecimento, através do uso de *tags* pré-estabelecidas com significado semântico. Além disto, com a difusão do uso da XML, o suporte para o trabalho de criação, leitura e edição destes tipos de arquivos torna-se bastante facilitado e padronizado.

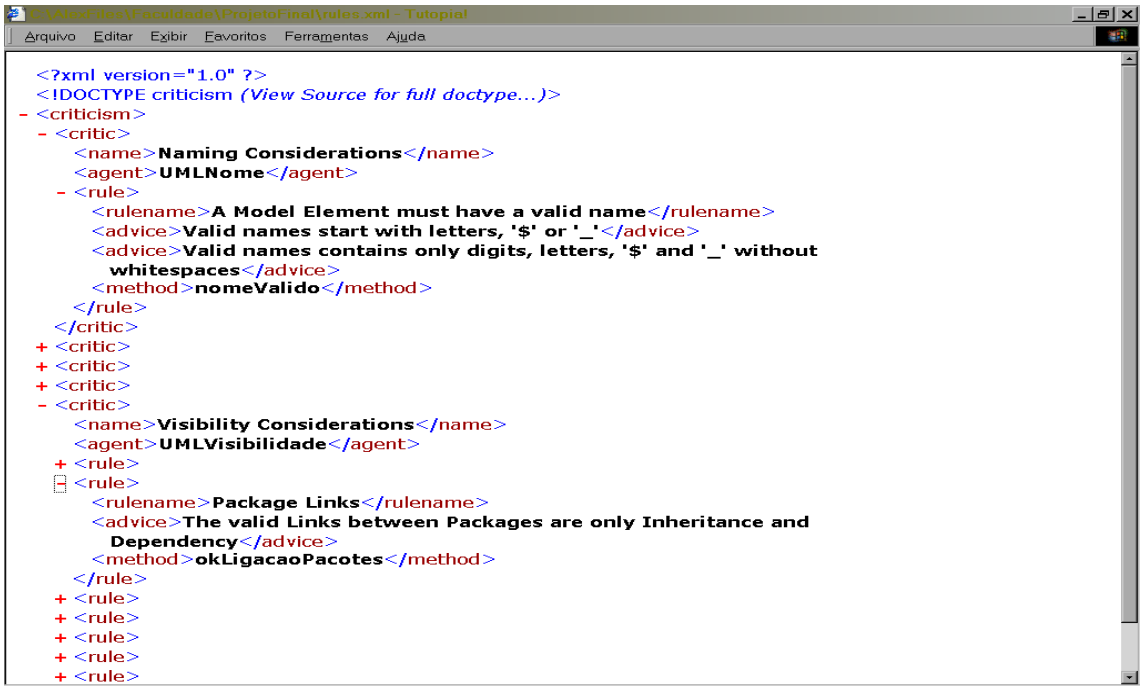

Figura 4.6 – Arquivo em XML de elementos do sistema de críticas

Uma vez escolhido o formato da representação dos elementos do sistema de críticas, é necessário trazer as informações do arquivo descritor para o Odyssey. A cada vez que o Odyssey for iniciado, o gerente de críticas recuperará todas as informações das críticas existentes do arquivo em XML, através de um objeto especial, o *CriticaLoader*. Sua função é ler e instanciar toda a estrutura hierárquica de críticas a partir do arquivo, entregando uma simples lista de objetos de crítica para o gerente de críticas. Essa leitura é feita através de um *parser*. O *parser* trabalha de forma orientada a eventos. A medida que ele reconhece um início ou fim de *tag*, são invocados métodos específicos como *startElement* e *endElement*, assim como *characters* é invocado em resposta a leitura do conteúdo de uma *tag*, armazenando-o em variáveis específicas temporárias. Desta forma, um simples autômato finito pode ser utilizado para se controlar a leitura do arquivo. Sempre que uma *tag* de fim de regra for lida, um objeto regra deve ser instanciado com base nas informações específicas para regra, armazenadas nas variáveis temporárias do autômato. O objeto regra também é

armazenado em uma lista relacionada a *tag* de uma crítica em aberto. Quando uma *tag* de fim de crítica for lida, um objeto crítica deve ser instanciado da mesma forma, com base nas variáveis temporárias e na lista de regras temporária, dando origem a um novo ciclo de leituras de instanciações, até o final do arquivo.

Assim que o gerente de críticas recebe todas as críticas do carregador, é necessário saber quais críticas e regras estão ativas ou inativas. Esse é um dos pontos mais importantes para a garantia da configuração persistente do mecanismo de críticas. O gerente de críticas não armazena os objetos de críticas, tendo que criá-los a partir da leitura do arquivo descritor externo ao Odyssey. Entretanto, o gerente armazena (através dos mecanismos de persistência do Odyssey) quais elementos (críticas e regras) estão inativos. Este armazenamento consiste em uma coleção dos nomes dos elementos que foram configurados como inativos pelo usuário. Após todos os objetos de críticas terem sido instanciados na inicialização do Odyssey, o gerente deve atualizar o estado de cada um dos objetos de acordo com seu estado (ativo ou inativo). As regras inativas não serão verificadas, e as críticas inativas não têm nenhuma de suas regras verificadas. Atualmente, o arquivo descritor em XML é apenas fonte das informações das críticas e regras, não sendo editado pelo mecanismo de críticas.

4.4.2 – Configuração do Sistema

A opção de configurabilidade é uma das características mais importantes para um sistema de críticas. A figura 4.7 mostra como é realizada a configuração do mecanismo no Odyssey, através do menu *Config* da janela principal. Uma nova janela é aberta, exibindo do lado esquerdo a estrutura hierárquica (árvore) de todos os elementos de crítica cadastrados no sistema.

Ao usuário é permitido visualizar as propriedades de um elemento selecionado (críticas e regras) através de campos específicos no lado direito da janela. A única opção de edição permitida é sobre o estado de ativação do elemento. Os elementos que estiverem com a *CheckBox* de ativação desmarcada são apresentados em tom de cinza, e não serão considerados durante a execução do mecanismo de verificação. Observe que se uma crítica for desativada, nenhuma de suas regras será verificada (devido a estrutura hierárquica entre críticas e regras).

Entretanto, apenas permitir que o usuário configure o mecanismo em uma sessão de execução do Odyssey não é suficiente. É preciso garantir alguma forma de persistência para que essas configurações sejam utilizadas novamente em outras sessões, conforme visto anteriormente. Neste sentido, o objeto *GerenteCriticas* deve ser serializável e armazenado pelos principais mecanismos de armazenamento do Odyssey (Werner *et al.*, 2000). A única informação que deve ser recuperada é o conjunto de nomes de elementos (regras e críticas) que foram configurados como inativos.

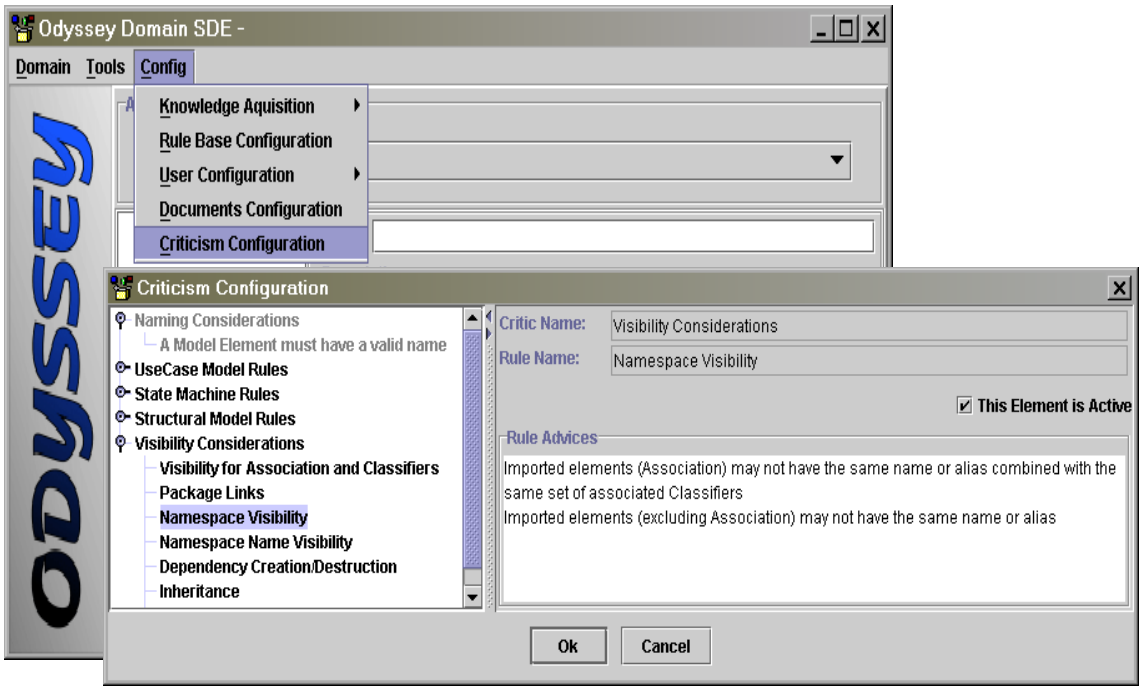

Figura 4.7 – Configuração do sistema de críticas

4.4.3 – Interação com Usuário

A forma como o mecanismo se apresenta ao usuário é um dos fatores importantes para a definição de um sistema de críticas, como visto no capítulo anterior. A figura 4.8 mostra uma das telas de resposta do mecanismo, no instante em que uma operação fez com que a execução do sistema descobrisse alguma inconsistência ou sugestão. Neste caso, o usuário está tentando criar um atributo da classe *Professor* chamado *depto*, porém existe um papel de fim de associação oposto, entre as classes *Departamento* e *Professor* com o mesmo nome, quebrando uma das regras de boa-formação da UML.

A resposta do sistema atua de forma preemptiva quando o usuário tenta completar uma operação que irá causar uma falha ou que merece melhoria. Neste

exemplo, verificamos a tentativa de criação de um atributo com mesmo nome de um fim de associação oposto à classe. A janela de edição (*Edit Professor*) é exibida quando o usuário seleciona a opção de editar um item léxico do diagrama. Ao tentar fechar a janela e capturar os dados dos painéis, o que atualiza as propriedades do item semântico, a execução do mecanismo de críticas impede que os atuais valores sejam aceitos. Se a regra que verifica esta determinada situação estiver inativa, a operação será completada sem restrições, porém uma inconsistência será introduzida no modelo.

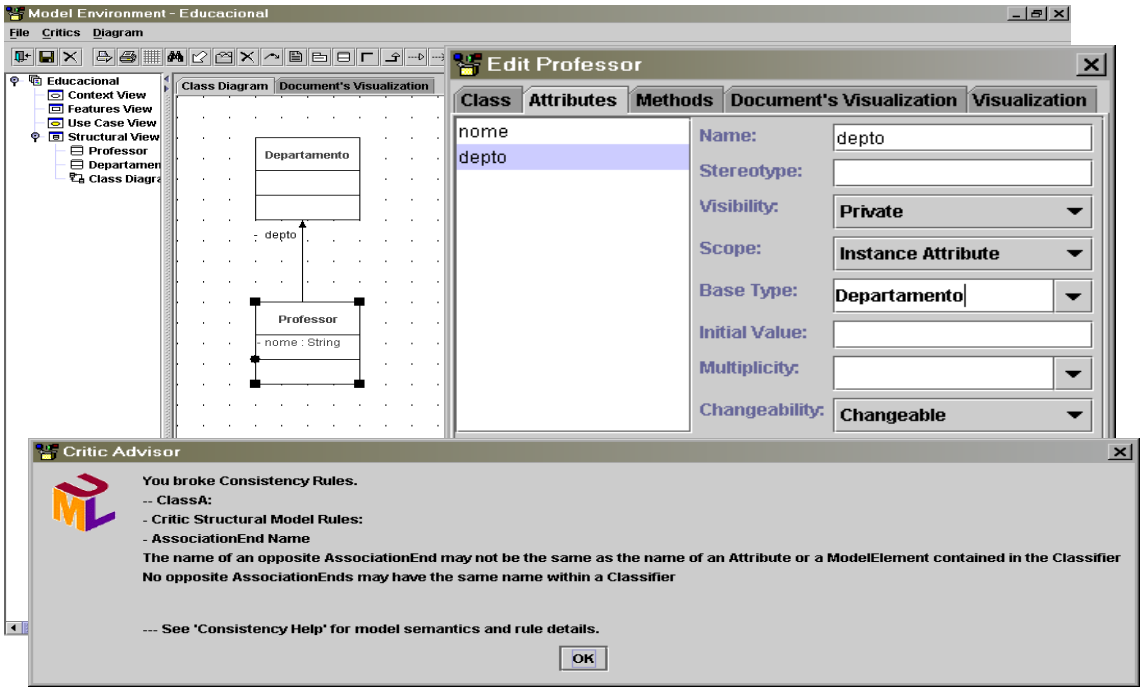

Figura 4.8 - Janela de resposta do mecanismo

Dois outros mecanismos de interação com o usuário são as fontes de ajuda e a criação de relatórios, conforme visto nos itens abaixo.

4.4.3.1 – Fontes de Ajuda

As fontes de ajuda presentes no mecanismo de críticas oferecem apoio conceitual ao desenvolvedor, permitindo que ele adquira conhecimento sobre questões de modelagem e projeto e diminua incertezas e ambigüidades, facilitando seu trabalho de modelagem. A figura 4.9 exibe a janela de ajuda para UML presente no Odyssey, formada por documentos hipertexto.

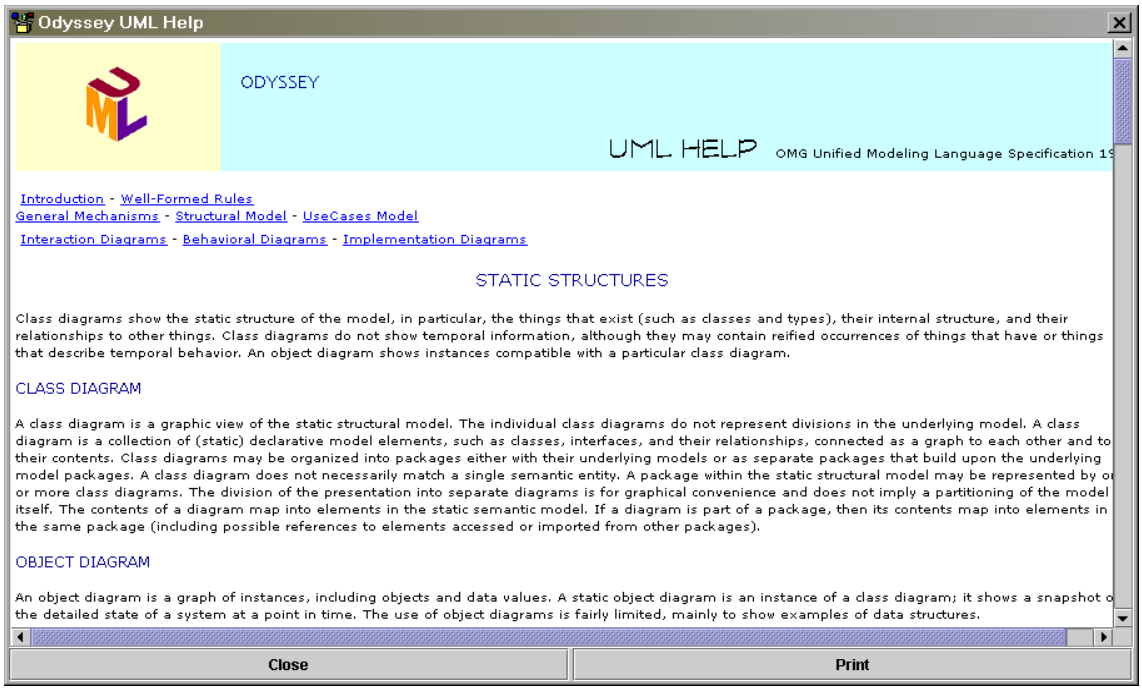

Figura 4.9 - Janela de Ajuda sobre UML

4.4.3.2 – Criação de Relatório

A criação de relatórios sobre um modelo é uma característica adicional importante em sistemas de críticas, uma vez que inconsistências podem estar embutidas despercebidamente, após o usuário ter desativado o mecanismo e efetuado várias operações inconsistentes. O relatório é criado a partir da chamada assíncrona no menu do ambiente de modelagem pelo usuário. A figura 4.10 mostra o menu de críticas, a chamada da criação de relatórios e a janela gerada em resposta ao usuário. A ativação e desativação completa da verificação de críticas podem ser feitas através de um dos itens de menu, como também mostra a figura 4.10.

O mecanismo de criação de relatórios é o mesmo utilizado na verificação preemptiva de críticas, porém, ao invés de trabalhar a partir da inserção/edição/remoção de um único objeto semântico, é preciso verificar todos os objetos dos modelos existentes, para descobrir se, em algum ponto, alguma regra sugerida (e ativa) não está sendo seguida. O relatório é estruturado da seguinte forma: para cada objeto semântico de cada modelo sendo verificado, se houver alguma inconsistência, é mostrada a categoria de crítica seguida por todas as regras desta categoria que foram violadas, e seus respectivos conselhos.

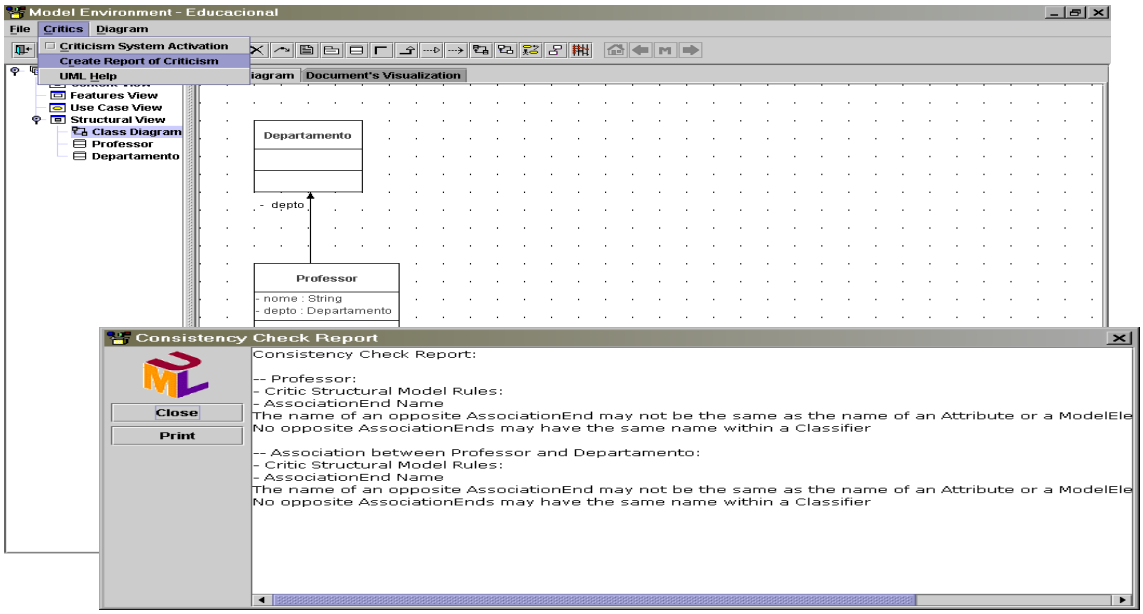

Figura 4.10 – Criação e exibição de relatório

CAPÍTULO V
CONCLUSÕES

As principais contribuições deste trabalho envolvem a melhoria de qualidade de especificações e modelos de projetos de software, principalmente quando há criação de modelos como artefatos de reutilização em diversas aplicações desenvolvidas. A detecção antecipada de inconsistências, erros e ambigüidades diminui o esforço de retrabalho e os custos do desenvolvimento, garantindo a viabilidade do projeto. Em adição, os projetistas envolvidos também apresentam relativo ganho de aprendizado sobre a linguagem e suas regras para a criação de bons modelos e projetos.

A partir da análise de ferramentas feitas no capítulo dois, e da proposta apresentada neste trabalho, implementada sobre a infra-estrutura Odyssey, é possível construir uma nova comparação entre as ferramentas, que evidencia as principais contribuições deste trabalho. A tabela 5.1 mostra este resultado.

Fator de Análise / Ferramenta		Argo	Magic	Object'ing	Rose	S'Modeler	Odyssey
Subconjunto da UML	Diagramas	☺	☺	☺	☺	☹	☺
	Elementos	☺	☺	☺	☺	😐	☺
	Propriedades	😐	☺	☺	☺	😐	☺
Categorias das Críticas	Classes e Core	😐	☹	😐	☹	☹	☺
	Casos de Uso	☺	☺	☺	☺	☺	☺
	Estados	😐	☺	😐	😐	😐	☺
Configuração	Desativação	☺	☹	☺	☹	☹	☺
	Por Crítica	☺	☹	☹	☹	☹	☺
	Expansibilidade	☹	☹	☹	☹	☹	☺
Interação com Usuário	Relatório	😐	☹	☹	😐	☹	☺
	Ajuda UML	😐	☺	☹	☺	☺	☺
	Preemptiva	☹	☺	☺	☺	☺	☺
	Explicativa	☺	☹	☺	☺	☺	☺

Tabela 5.1 – Nova Comparação entre Ferramentas

5.1 – LIMITAÇÕES E TRABALHOS FUTUROS

Este trabalho focaliza apenas questões de consistência a partir de regras da linguagem UML, porém estas regras estão limitadas apenas a verificações notacionais, de sintaxe e semântica estática. Conhecimentos mais completos sobre a semântica dinâmica são apenas descritos de forma textual.

O mecanismo de críticas proposto e implementado na infra-estrutura Odyssey, porém, é flexível para futuras expansões de verificações de modelos, além do escopo da linguagem de modelagem. Em relação ao armazenamento das críticas, uma futura implementação pode vir a permitir a inclusão ou exclusão de críticas e regras através de uma interface com o usuário chamada de dentro do próprio Odyssey, assim como o armazenamento das informações de ativação diretamente no arquivo descritor.

Outros trabalhos podem estar relacionados à criação de críticas baseadas em boas heurísticas no contexto de suporte a tomada de decisões de projeto, e a verificação de consistência entre as diferentes visões de modelos de domínio para apoiar o processo de reutilização. No primeiro caso, seria permitido que o usuário aceitasse ou não as sugestões do sistema de críticas, pois as verificações não seriam somente baseada na violação de regras. Algum tipo de suporte automático da ferramenta para a execução da sugestão indicada também poderia ser disponibilizado. Isto não é necessário para a verificação das regras da UML porque o sistema visa impedir que o usuário insira inconsistências nos modelos. Para o caso da verificação de modelos de domínio, podem ser implementadas regras que verifiquem questões a respeito do desenvolvimento do diagrama de *features*, assim como regras que atuem durante o processo de engenharia de aplicações. Algumas regras valorizariam questões como restrições de inclusão e exclusão, *features* externas e essenciais, entre outras. Alguns trabalhos têm sido desenvolvido neste sentido (Oliveira *et al.*, 2001; Gomaa *et al*., 1996).

.

REFERÊNCIAS BIBLIOGRÁFICAS

ARGO UML 0.81, *University of California* - http://www.argouml.org/, Abril, 2001.

ARAÚJO B.A. *Uma Contribuição para a Consistência de Diagramas da UML.* Tese de M.Sc. Universidade Federal de Minas Gerais. Belo Horizonte, MG. 2000.

BARROS, M. *Reutilização de Conhecimento no Gerenciamento de Projetos Baseado em Cenários.* IN : Engenharia de Domínio e Desenvolvimento Baseado em Componentes, Relatório Técnico do Projeto Odyssey 10/2000, COPPE/UFRJ, Rio de Janeiro, Brasil.

BOOCH, G. *Object Solutions. Managing the Object-Oriented Project*. Addison-Wesley. 1996.

BRAGA, R.M.M. *Busca e Recuperação de Componentes em Ambientes de Reutilização de Software*. Tese de D.Sc. COPPE/UFRJ. Rio de Janeiro, Dezembro, 2000.

CONROW E.H., SHISHIDO P.S., *Implementation Risk Management on Software Intensive Projects*. IEEE Software, Volume 14, Número 3, p. 83-89, maio/junho 1997.

EVANS A., LANO K., FRANCE R., RUMPE B. *Meta-Modeling Semantics of UML.* IN: Behavioural Specifications for Businesses and Systems, Kluwer Academic Publishers, Editor: Haim Kilov, Chapter 4, 1999 .

FRANCE R., EVANS A., LANO K., RUMPE B. *The UML as a Formal Modeling Notation.* OOPSLA'97 Workshop on Object-Oriented Behavioral Semantics, p. 75-81. Atlanta, EUA. Outubro, 1997.

GOMAA H., KERSCHERBERG L., SUGUMARAN V., BOSCH C., TAVAKOLI I, O'HARA L. *A Knowledge-Based Software Engineering Environemt for Reusable Software Requirements and Architectures*. IN: Automated Software Engineering 3; p. 285-307, 1996, Kluwer Academic Publishers.

GRISS, M., FAVARO, J., D'ALESSANDRO, M. Integrating Feature Modeling with RSEB, *5th International Conference on Software Reuse (ICSR-5)*, ACM/IEEE, Vitória, Canadá, Junho, 1998.

JACOBSON, I., CHRISTERSON, M., JONSSON, P., *et alli*, *Object-Oriented Software Engineering – A Use-case Driven Approach*, 1 ed., Massachusetts, Addison-Wesley Longman, 1992.

KANG, K., COHEN, S., HESS, J. *et alli. Feature-Oriented Domain Analysis (FODA) - Feasibility Study*, SEI Technical Report CMU/SEI-90-TR-21. 1990.

KOTONYA G. SOMMERVILE I., *Requirements Engineering with Viewpoints*. Software Engineering Journal. Janeiro 1996.

LIMA K.V.C., WERNER C.M.L. *Framework para Comparação de Ambientes de Desenvolvimento de Software orientados a Dominio*. Publicações Técnicas. COPPE UFRJ. Março de 1998.

MAGIC DRAW 4.0, *NoMagic* - http://www.nomagic.com/, Abril, 2001.

MILLER, N., *A Engenharia de Aplicações no Contexto da Reutilização Baseada em Modelos de Domínio.* Tese de M.Sc., COPPE/UFRJ, Rio de Janeiro, Brasil, 2000.

MURTA, L.G.P., *FRAMEDOC: Um FrameWork para a Documentação de componentes Reutilizáveis*, Projeto Final de Curso, DCC/IM/UFRJ, Novembro, 1999.

MURTA, L. *Uma Máquina de Processos de Desenvolvimento de Software Baseada em Agentes Inteligentes*, XIV Simpósio Brasileiro de Engenharia de Software, Workshop de Teses. João Pessoa. Outubro, 2000.

OBJECTEERING 4.3.2, *Softeam* - http://www.objecteering.com/, Abril, 2001.

ODYSSEY, *COPPE UFRJ* - http://www.cos.ufrj.br/~odyssey, em Abril, 2001.

OLIVEIRA, H.L.R, ROCHA, C.R.P, GONÇALVES, K.M., SOUZA C.R.B. *Utilização de Sistemas Críticos nas Atividades de Engenharia de Domínio e Aplicações.* Artigo submetido ao XV SBES, Rio de Janeiro, 2001.

OMG. *OMG Unified Modeling Language Specification.* Versao 1.3. 1999.

PRESSMAN, R. *Sofware Engineering. A Practitioner's Approach.* 4a. Edicao. McGraw-Hill. 1997.

PRIETO-DIAZ R. *Domain Analysis for Reusability*. Proceedings of the 11th COMPSAC - Computer Software & Applications Conference. Toquio, Japão. 1987. p. 23-29.

PRIETO-DIAZ, R., ARANGO, G., *Domain Analysis and Software System Modeling*, IEEE Computer Society Press Tutorial. 1991.

RATIONAL ROSE, *Rational* - http://www.rational.com/, Abril, 2001.

ROBBINS, J.E. *Design Critiquing System.* Tech Report UCI-98-41, University of California, Irvine, Novembro, 1998.

ROBBINS J.E., HILBERT D.M., REDMILES D.F. *Software Architecture Critics in Argo*. Formal Demonstration at the 1998 Conference on Intelligent User Interfaces (IUI'98).

SOFTMODELER Enterprise Edition 2.5, *Softera* - http://www.softera.com/, Abril, 2001.

SUN Microsystems. Página Web em http://java.sun.com. Abril, 2001.

VASCONCELOS A.P.V. *Formalization of UML using Algebraic Specifications.* Tese de M.Sc. Vrije Universiteit Brussel. Bélgica. 1999.

VERONESE G.O., NETTO F.J. *Uma Ferramenta de Apoio a Recuperação de Projetos no Ambiente Odyssey*. Projeto Final de Curso, DCC/IM/UFRJ. Abril, 2001 (em andamento).

WERNER C.M.L. *Notas de aula do curso de Fundamentos da Engenharia de Software*. Universidade Federal do Rio de Janeiro. Rio de Janeiro, RJ. 1999.

WERNER, C., BRAGA, R., MATTOSO, M., *et alli. Odyssey: Estágio Atual.* Caderno de Ferramentas do XV Simpósio Brasileiro de Engenharia de Software (XIV SBES). João Pessoa, Brasil. Outubro, 2000.

XAVIER, J.R. *Criação e Instanciação de Arquiteturas de Software Específicas de Domínio no contexto de uma Infra-estrutura de Reutilização*. Tese de M.Sc., COPPE/UFRJ, Rio de Janeiro, Brasil, 2001.

XML. Página da Web em http://www.xml.org , Abril , 2001.

ZOPELARI, M. *Uma Proposta de Sistemática para Aquisição de Conhecimento no contexto de Análise de Domínio*, Tese de M.Sc., COPPE/UFRJ, Novembro, Rio de Janeiro, Brasil, 1998.