1 Introdução

1.1 Motivação

Mudanças são inevitáveis durante a existência de um software. Em muitas situações, a documentação é inexistente e o código fonte é a única fonte de informação disponível à manutenção do sistema.

Devido ao fato do código se apresentar em um nível de abstração baixo, pois é direcionado a ferramentas do gênero dos compiladores, e das dimensões de um software serem geralmente grandes (considerando a quantidade de linhas de código), o entendimento do sistema torna-se um processo não trivial. É necessário, então, prover suporte, de preferência automatizado, a este processo.

No contexto apresentado se insere a Engenharia Reversa. Este termo é empregado a um grande conjunto de tarefas relacionadas ao entendimento e modificação de sistemas de software. As duas principais atividades da Engenharia Reversa são a identificação dos componentes de um sistema e dos relacionamentos entre eles e a criação de descrições de alto nível de vários aspectos do sistema (CHIKOFSKY *et al*., 1993).

A aplicação da Engenharia Reversa tem sido uma prática comum na indústria há muito tempo. A indústria bélica se vale deste recurso para conseguir se manter competitiva. A espionagem industrial também se baseia nestes conceitos . Na última década este tópico da Engenharia de Software ganhou maior atenção da academia e do mercado. O primeiro reflexo deste reconhecimento se deu com a criação da primeira conferência dedicada exclusivamente a este campo de pesquisa denominada WCRE - *Working Conference on Reverse Engineering*, em 1993, sob a organização da ACM e da IEEE *Computer Society* (CHIKOFSKY E.J. *et al*., 1994).

Algumas limitações impedem que a Engenharia Reversa possa se tornar um processo totalmente automatizado. A principal limitação é a falta de rastreabilidade de conceitos entre as diferentes abstrações do projeto. Uma mesma especificação pode ser implementada de diversas formas por diferentes programadores. Todas as formas podem ser consideradas

corretas, se preencherem os requisitos propostos no projeto. Esta dificuldade de se obter um único caminho na direção projeto - código (Engenharia Direta) resulta na dificuldade de se seguir um único alvo no caminho inverso, código - projeto (Engenharia Reversa). O pouco emprego de metodologias de desenvolvimento e a pouca utilização de padrões ampliam este problema.

Durante a análise de dados do código utilizados na extração de conceitos do projeto, nos deparamos com situações onde a origem real do conceito implementado em algoritmo não é possível de ser atingida. Este é um problema conhecido na literatura como "Problema da Associação de Conceitos" (*Concept Assignment Problem*)(BIGGERSTAFF *et al.*, 1994).

Apesar das dificuldades de se obter um processo de extração dos conceitos do projeto automaticamente a partir do código, a Engenharia Reversa mostra-se bastante útil quando auxiliada por um agente humano.

A orientação a objetos vem sendo largamente utilizada nas últimas décadas. Entre os argumentos para a sua rápida difusão estão a maior facilidade de tradução de elementos do mundo real em elementos computacionais, e a promessa de permitir a construção de sistemas mais fáceis de se manter e reutilizar. Porém, o desenvolvimento de software orientado a objetos, não obedecendo a certos princípios de projetos bem conhecidos (encapsulamento, modularidade, coesão, acoplamento, etc.), deu origem a uma gama de sistemas difíceis de evoluir e reutilizar. Para que estes sistemas evoluam, muitas vezes é necessário que se façam revisões do código para adequação do sistema aos princípios citados. Este processo de reformulação do código é conhecido na literatura como Reestruturação (*Refactoring*) (FOWLER, 2001). Esta reestruturação pode ser auxiliada por ferramentas de Engenharia Reversa, através da criação de visões do código em níveis mais altos de abstração, possibilitando ao usuário a identificação de estruturas problemáticas no código que devem ser minimizadas ou substituídas por outras soluções.

Portanto, pode-se concluir que o surgimento de metodologias baseadas na orientação a objetos contribuiu para que o mapeamento da especificação em código fosse feito de forma mais suave, porém não garantiu a unicidade das rotas possíveis de implementação.

A UML (OMG, 2000) é uma linguagem de modelagem adotada como padrão para o desenvolvimento de sistemas orientados a objetos e visa fornecer um suporte à criação desses sistemas, através de um conjunto de diagramas e outras construções abstratas. Dentre os diagramas que compõem a fase de projeto, destacam-se como mais relevantes os diagramas de classe e de interação. O primeiro fornece uma visão estrutural do sistema e o último uma visão comportamental.

O surgimento de sistemas legados orientados a objetos na última década estimulou a criação de ferramentas que auxiliassem o processo de manutenção, fornecendo suporte à recuperação de modelos em um formato padrão, como a UML, a partir do código. Segundo (PRESSMAN, 2001), o termo "sistemas legados" é um eufemismo para sistemas, geralmente antigos, mal documentados e mal projetados que devem ser mantidos por muitos anos, por serem críticos para o negócio de uma organização. A Engenharia Reversa atua no auxílio a recuperação da documentação e ao entendimento desses sistemas, possibilitando que a sua manutenção seja realizada de forma menos árdua.

A Engenharia Reversa também fornece auxílio para que o ciclo de desenvolvimento não siga necessariamente um curso linear seqüencial, como o tradicional modelo em cascata. As metodologias OO prevêem idas e vindas entre as fases de desenvolvimento. O que acontece com a análise e projeto, onde elementos do projeto ajudam a refinar idéias da análise, pode acontecer com as fases de projeto e implementação, isto é, determinadas informações, antes ausentes em modelos de projeto e que auxiliam o entendimento do sistema, podem ser reconstituídas a partir do código.

A possibilidade de reconstituição de diagramas de classe e interação entre objetos a partir do código é de grande utilidade para projetistas e programadores. Entretanto, diagramas que mostrem a interação entre os objetos são muito difíceis de serem extraídos a partir do código. A estratégia utilizada para resolver este problema é o monitoramento do sistema através de ferramentas que atuam como depuradores (*debuggers*), que supervisionam a interação entre os objetos em tempo de execução (SYSTÄ, 1999).

KOLLMAN *et al*. (2001) descrevem uma abordagem para a captura do comportamento dinâmico de programas em Java a partir do código. Esta abordagem parte

de código Java para a criação de diagramas de colaboração da UML. Porém, não há referência no trabalho citado para alguma ferramenta que implemente esta abordagem.

Observamos, então, duas vertentes da Engenharia Reversa sobre programas Orientados a Objetos, que parte do nível de implementação para o nível de projeto: uma que trata da parte estática, e extrai as informações do código; a outra que trata a parte dinâmica. Esta última apresenta duas estratégias de solução: i) extração da dinâmica a partir do código, e ii) extração da dinâmica a partir de monitoramento do programa em tempo de execução e que possibilita a extração mais completa da interação entre os objetos.

Neste trabalho, foca-se a extração da estrutura estática do programa. A estrutura estática de um programa em Java pode ser extraída a partir do código, sem a necessidade de execução e monitoramento do sistema. Diagramas de classe da UML podem ser usados para representar esta estrutura. Basicamente, podemos dizer que é possível mapear estruturas conhecidas em tempo de compilação (os elementos estáticos de um programa, como as classes e seus relacionamentos) em uma representação gráfica, obtendo-se uma "fotografia" do sistema.

A partir da visão estrutural, é possível ter um entendimento maior sobre software e identificar algumas construções que representam potenciais barreiras à evolução do sistema. Esta visão fornece, portanto, um suporte à reestruturação destas construções ruins.

O processo de detecção de construções problemáticas em um modelo contribui para a melhoria da qualidade do sistema. Uma ferramenta de engenharia reversa deve fornecer uma estrutura para que tais construções possam ser detectadas. Em (CORREA, 1999), é apresentada uma ferramenta que auxilia a detecção de construções problemáticas (*Anti-Patterns*) de modelos de sistemas orientados a objetos. A ferramenta também detecta as boas construções (*Patterns*), contribuindo para o aprendizado do projetista.

Existem ainda algumas informações sobre a dinâmica comportamental que podem ser extraídas do código e apresentadas no diagrama classes. Neste caso, estamos nos referindo a estruturas que são conhecidas em tempo de execução, mas que podem ser extraídas a partir de uma investigação mais detalhada do código. Por exemplo, a partir da identificação de pontos onde determinados tipos de objetos são instanciados, é possível exibir em um

diagrama uma dependência entre a classe do objeto instanciador e a classe do objeto instanciado.

Este é um ponto de diferenciação deste trabalho com relação a algumas das abordagens encontradas na literatura, descritas e avaliadas no capítulo 2. A exibição de informações da dinâmica de interação dos objetos do programa que podem ser capturadas a partir do código, mais especificamente as dependências, é de grande utilidade para o projetista e/ou programador durante a reestruturação do modelo e do código.

Conforme dito anteriormente, nem todas as construções de código podem ser detectadas de maneira simples, ou de maneira única, tendo em vista as diferentes formas que podem existir na tradução de conceitos do mundo real em conceitos computacionais. Isto ainda é agravado pelo fato da UML ser uma linguagem semiformal, que não fornece meios para que o mapeamento para o código seja feito de forma única. Portanto, uma ferramenta que auxilie a recuperação de modelos a partir do código não deve inferir construções de modelo que possam não estar de acordo com os conceitos originais pensados quando da concepção do código.

Além do contexto apresentado da manutenção, a aplicação da Engenharia Reversa pode auxiliar a reutilização de componentes de software e auxiliar o entendimento de um domínio de aplicações. Para se reutilizar um componente de código é necessário, primeiro, que se entenda sua estrutura e comportamento.

No mercado e no meio acadêmico, existem algumas ferramentas, acopladas a ambientes CASE, que se propõem apoiar a Engenharia Reversa de código orientado a objetos. Porém, poucas a fazem de uma maneira que recupere detalhes importantes do código, cuja representação no modelo seria de grande importância para documentação, entendimento, manutenção ou reestruturação do sistema.

1.2 Objetivos

Este trabalho tem como objetivo analisar em detalhe as dificuldades de se extrair determinadas informações a partir do código fonte de programas orientados a objetos, especificamente escritos na linguagem Java e propor uma abordagem para procurar solucionar as dificuldades encontradas. Validaremos a abordagem através da construção de

uma ferramenta que auxilia a extração de modelos de classe UML a partir do código fonte de um programa em Java.

A ferramenta deve fornecer um diferencial em relação às abordagens existentes no que tange a precisão e quantidade dos dados extraídos, isto é, os dados extraídos do código devem ser trabalhados no sentido de produzir um modelo que auxilie o usuário o máximo possível no entendimento. Além disso, é desejável que a ferramenta ofereça um ambiente de uso confortável ao usuário.

A linguagem Java foi escolhida como fonte para extração dos modelos por ser uma linguagem orientada a objetos bastante difundida, tanto no mercado como no meio acadêmico. A ferramenta deve atuar, ainda, no contexto do Ambiente Odyssey, que provê uma infra-estrutura de reutilização, baseada em modelos de domínio, que está em desenvolvimento na COPPE/UFRJ (WERNER *et al*., 2000). A ferramenta proposta deve poder ser estendida para outras linguagens orientadas a objetos (por exemplo, C++, Delphi, etc.) em relação às fontes de extração dos modelos, sem que a estrutura principal seja comprometida.

É desejável que a ferramenta produzida possua estrutura flexível o suficiente para que ser acoplada, sem grandes modificações na sua estrutura, a outros ambientes e fornecer, no futuro, o suporte a recuperação de modelos a partir de outras linguagens OO.

1.3 Organização

No próximo capítulo, são identificadas algumas características de ferramentas de extração de modelos de classes da UML a partir de códigos em Java, que são relevantes para a reestruturação e entendimento do código. Com base nestas características, são definidos critérios que são utilizados para comparar um conjunto de ferramentas de Engenharia Reversa de código Java. Estas ferramentas possuem projeção tanto no mercado quanto no meio acadêmico.

No capítulo 3, são analisadas estratégias de mapeamento de estruturas da linguagem Java em estruturas da UML. É proposta uma abordagem baseada nestas estratégias. Apresentamos ainda uma arquitetura de uma ferramenta que busca atender às características identificadas no capítulo 2.

No capítulo 4, é apresentada a implementação da abordagem proposta. São explicadas as decisões de projeto e detalhes sobre instanciação da arquitetura. Ao final do capítulo, ilustra-se com um pequeno exemplo, as diferenças entre um modelo extraído do código e um modelo concebido na fase de projeto.

Finalmente, no capítulo 5, descrevemos as contribuições obtidas com este trabalho, analisamos suas limitações e encerramos com algumas possibilidades de trabalhos futuros.

2 Auxílio à Recuperação de Modelos de classe UML a partir de código Java

2.1 Introdução

Neste capítulo, identificamos as características desejadas de uma ferramenta de extração de modelos UML a partir de um código fonte em Java. As características foram identificadas a partir do estudo das ferramentas de maior destaque na área. Nesta identificação, consideramos o ponto de vista de um projetista ou programador, destacando as funcionalidades que devem estar presentes em uma ferramenta que auxilie a recuperação de modelos de projeto, mais especificamente modelos de classe da UML.

As ferramentas analisadas apresentam uma forma comum de interação com o usuário, com pequenas diferenças de uma ferramenta para outra. Basicamente, o usuário seleciona um conjunto de arquivos correspondente ao subsistema a ser avaliado (códigos fonte em Java do programa) e aciona a ferramenta para que o modelo seja extraído a partir dos arquivos fonte selecionados. A ferramenta interpreta o código fonte, analisa as informações extraídas e exibe o resultado em uma estrutura visualizável em um diagrama de classes. Em geral, estas ferramentas estão acopladas a ambientes CASE e visam estender o suporte destes ambientes além das fases de desenvolvimento, provendo suporte, também, à manutenção do software.

Considera-se que a ferramenta seja utilizada como meio de facilitação do entendimento, documentação, e/ou manutenção de um subsistema. Desta forma, uma característica importante a ser levada em conta é a correção das informações exibidas, para evitar que o usuário (projetista ou programador) tome decisões equivocadas baseado no modelo extraído que não corresponde a um possível modelo que represente o código.

A partir da identificação das características, apresentamos um resumo comparativo entre as ferramentas existentes no mercado e no meio acadêmico que fazem Engenharia

Reversa de código Java e apresentam os modelos extraídos em uma representação em UML.

Este capítulo está dividido em cinco seções. Esta seção introduz o capítulo. Na segunda seção, apresentamos uma breve visão dos elementos que compõem o diagrama de classes UML e que são objetos de estudo neste trabalho. Na terceira seção, são apresentadas as características identificadas a partir da análise das ferramentas. A quarta seção apresenta uma avaliação das ferramentas de destaque na área, e a última seção exibe um resumo comparativo da análise feita.

2.2 Elementos que compõem diagramas de classe da UML

O modelo estático de um programa orientado a objetos pode representar três elementos principais: classificadores (e sua estrutura interna), pacotes e os relacionamentos entre pacotes e entre classificadores. Estes elementos são definidos na especificação da UML e possuem representações específicas em diagramas de classe. A seguir detalhamos estes elementos.

2.2.1 Classificador

Um classificador pode ser uma classe, uma interface ou um tipo básico de dados (*DataType*).

Uma classe é um descritor para um conjunto de objetos com comportamento, estrutura e relacionamentos semelhantes (OMG, 2000). As classes são compostas por elementos que definem seu estado (*Atributos*) e elementos que definem o seu comportamento (*Operações*).

Uma interface especifica um comportamento a ser seguido pelas classes que a implementam. Este comportamento é definido por um conjunto de operações.

Um tipo básico de dados descreve um conjunto de valores sem identidade, correspondendo normalmente a números, cadeias de caracteres, data e hora (OMG, 2000).

2.2.2 Pacote

Os pacotes fornecem a organização da estrutura dos elementos do modelo em uma hierarquia. Um pacote é um agrupamento de elementos de modelo[1]. Pode conter classificadores e outros pacotes.

Pacotes bem projetados resultam do agrupamento de elementos que estão semanticamente próximos e que apresentem uma tendência de mudarem juntos. Desta forma, pacotes tendem a apresentar um fraco acoplamento entre si, uma forte coesão e um acesso bastante controlado aos elementos que o compõem. A análise das interdependências entre os *pacotes* pode trazer à tona várias construções problemáticas (CORREA, 1999).

2.2.3 Relacionamentos

Na UML são definidos quatro tipos de relacionamentos entre elementos do modelo, que podem ser representados em um diagrama de classes: herança, realização, associação e dependência.

A *herança* é um relacionamento de especialização/generalização. Um elemento específico (filho) herda de um elemento genérico (pai). O elemento filho, através deste relacionamento, adiciona informação extra (novos atributos, novas operações ou alteração de operações) ao elemento pai herdado. Os elementos participantes do relacionamento podem ser classificadores e pacotes. Entretanto, as linguagens de programação OO não fornecem suporte para herança entre pacotes.

A *realização* representa um relacionamento entre um classificador (a implementação) que implementa as operações definidas em um outro classificador, do tipo interface (a especificação). Essas operações definidas na interface visam estabelecer um comportamento que deve ser seguido pelos classificadores que a implementam.

Uma *associação* define um relacionamento semântico entre pelo menos dois classificadores. Este relacionamento possui algumas propriedades, inerentes aos participantes da ligação, que determinam sua natureza. Estas propriedades são apresentadas nesta seção dada a importância de se avaliar se algumas delas são detectadas pelas

[1] Elemento de modelo (*ModelElement* na especificação da UML) representa uma abstração do sistema sendo modelado. No contexto deste trabalho, um elemento de modelo pode ser um **classificador** ou um **pacote.**

ferramentas. As propriedades mais importantes são multiplicidade, navegabilidade, indicador de agregação, indicador de mutabilidade (*changeability*), indicador de escopo, papel e visibilidade. Tais propriedades dizem respeito à estrutura estática dos elementos participantes da associação (*Association Ends*) .

Uma *dependência* descreve que o funcionamento ou implementação de um elemento requer o funcionamento de outros elementos (OMG, 2000). A existência de quaisquer dos relacionamentos citados anteriormente (Herança, Realização ou Associação) configura uma dependência entre os seus participantes. Existem formas de dependências que podem ser capturadas e resultam da interação entre os objetos durante a execução. Portanto, a detecção de tais relacionamentos deve levar em conta aspectos da dinâmica do programa. As dependências podem ocorrer entre pacotes e entre classificadores. A exposição das dependências permite ao usuário uma visão genérica do programa, possibilitando a detecção de problemas que comprometam a estrutura do programa (o excesso de dependências e/ou existências de dependências circulares, por exemplo).

A especificação da UML propõe quatro tipos de dependências, que são representados no modelo através de estereótipos. Os tipos definidos são: abstração (*abstraction*), associação (*bind*), permissão (*permission*) e uso (*usage*). A análise da especificação nos leva a conclusão que apenas as dependências do tipo permissão e uso são detectáveis a partir do código fonte feito em Java. As outras dizem respeito a conceitos de projeto não explícitos (ou não existentes) na implementação que fazem sentido apenas no modelo.

2.3 Características de uma ferramenta de Engenharia Reversa Java-UML

Através do uso de diferentes ferramentas e sondagens junto a usuários[2] de ferramentas de Engenharia Reversa, foi possível identificar um conjunto de características desejáveis em uma ferramenta de Engenharia Reversa de Java para UML.

Dois grupos de características foram identificados: o primeiro refere-se à detecção dos elementos do código e representação destes elementos no modelo, sendo considerado

[2] As sondagens foram feitas informalmente através de perguntas aos integrantes da equipe que desenvolve o Ambiente Odyssey.

importante a quantidade e precisão dos dados extraídos. O segundo grupo refere-se ao ambiente de uso fornecido pela ferramenta, refletindo analisou-se a capacidade da ferramenta em prover determinadas facilidades de manipulação e visualização do modelo.

2.3.1 Capacidade de detecção

Considera-se que uma ferramenta deva ser capaz de representar, no modelo, elementos da estrutura estática presentes no código. Neste sentido, é preciso analisar como determinadas estruturas de código devem em Java ser mapeadas no modelo UML correspondente.

As estruturas da UML utilizadas foram apresentadas na seção anterior, tendo como objetivo analisar a capacidade das ferramentas em capturar estas estruturas a partir do código.

Verificamos que algumas das estruturas são detectadas no código por todas as ferramentas e, portanto, não convém utilizá-las como um diferencial entre as ferramentas. Estas estruturas são os classificadores, e seus elementos internos (Operações e Métodos), e os relacionamentos de herança e realização. O mapeamento destas estruturas de Java para UML é trivial, e se dá diretamente a partir de estruturas sintáticas da linguagem Java, não havendo necessidade de uma análise detalhada dos dados. Todas as ferramentas identificaram de forma correta a ocorrência destas estruturas em código e as representaram em UML. Logo, as características importantes a serem observadas com relação à capacidade de detecção dizem respeito às estruturas não triviais, que merecem uma análise mais cuidadosa da ferramenta, conforme detalhadas a seguir:

i) Detecção de associações

Visa mostrar os relacionamentos de associações, e suas propriedades, existentes entre os classificadores detectados a partir do código fonte selecionado. Entre as principais propriedades dos participantes da ligação, a serem detectadas e exibidas, destacamos:

- **Cardinalidade (Multiplicidade):** indica o número de instâncias de um classificador alvo do relacionamento que podem estar associadas a uma instância do classificador origem do relacionamento. Neste tópico, deve-se avaliar apenas como as ferramentas se comportam nas situações mais triviais, como no caso de atributos de cardinalidade

simples, ou então com atributos do tipo *array*. O item *v* é dedicado especialmente à análise do tratamento de coleções não tipadas.

- **Navegabilidade:** indica o sentido da associação;
- **Papel:** indica o papel realizado pelo classificador na associação. Um papel é um pseudo-atributo;
- **Mutabilidade (*changeability*):** indica se uma instância participante da associação pode ser alterada pela outra instância;
- **Escopo:** Indica, de acordo com o atributo, se o classificador destino está presente em uma classe ou em uma instância (objeto).
- **Visibilidade:** indica, do ponto de vista de um participante da associação, o grau de visibilidade do outro participante (público, privado ou protegido).

Uma outra propriedade importante da associação é o indicador de agregação. Este indicador especifica se a instância de um participante (o todo) é composta por (composição) ou agrega (agregação) elementos do outro participante (a parte), constituindo assim um relacionamento todo-parte.

Os relacionamentos de agregação e composição são de natureza mais conceitual e são explícitos apenas durante a modelagem. Nas linguagens OO conhecidas, não há estrutura de suporte explícita para este tipo de relacionamento. A detecção de agregações/composições em tempo de compilação não é possível, pois estes relacionamentos se caracterizam pela sua intenção (GAMMA *et al*., 1995). Em código, este relacionamento costuma ser implementado através do uso de coleções em atributos. A detecção destes elementos não depende apenas de sua implementação, mas também de conceitos envolvidos em projeto. Por se tratarem de coleções, estes relacionamentos podem ser representados como associações com cardinalidade múltipla (N).

A escolha da associação com cardinalidade múltipla para representação de um possível relacionamento de agregação/composição pode acarretar a perda da semântica, porém não induz o usuário a tomar decisões erradas. Esta situação configura uma instância do problema de associação de conceitos (BIGGERSTAFF *et al*., 1994).

ii) Detecção de dependências

Este item visa avaliar como as ferramentas analisam as situações ou estruturas de códigos que acarretam os diferentes tipos de dependências. A detecção e exibição das dependências no modelo são de suma importância, pois permitem ao usuário a identificação de elementos que comprometem a evolução e/ou reutilização de elementos do sistema. É importante que as dependências sejam capturadas não apenas entre classificadores, mas também entre pacotes, pois fornece o comprometimento da estrutura geral do programa.

Cada tipo de dependência tem um conjunto de estereótipos definidos na especificação da UML, que determinam os casos específicos que causaram a dependência no modelo. Portanto, é preciso avaliar a capacidade da ferramenta em detectar as situações que acarretaram a dependência e não os estereótipos das dependências exibidos pela ferramenta. Isto porque os estereótipos são elementos usados para extensão da semântica da linguagem UML e têm seu uso livre pelas diferentes implementações de ferramentas.

Entre as causas e formas de dependências temos:

- **Importação de classificadores ou pacotes:** a importação de elementos configura a dependência entre o elemento importador e os elementos importados;
- **Passagens de parâmetros:** a passagem de parâmetro de uma instância em um método configura uma dependência do classificador portador do método em relação ao classificador da instância passada como parâmetro;
- **Retorno de objetos:** o retorno de uma instância por um método indica a dependência do classificador portador do método em relação ao classificador do objeto retornado;
- **Instanciação de objetos:** a instanciação de objetos por um classificador indica a dependência do classificador instanciador para o classificador instanciado;
- **Variáveis locais:** a declaração de variáveis locais em um método caracteriza uma dependência do classificador portador do método para o classificador da variável local declarada no método;
- **Conversões forçadas de tipo (*casts*):** conversões forçadas de tipo, visam transformar a instância de um classificador em uma instância de um outro classificador. O uso destas

conversões configura uma dependência do classificador, onde ocorre a conversão, para o tipo (classificador) alvo da conversão;

- **Chamadas a métodos a partir de objetos:** a invocação de métodos de uma instância de uma outra instância caracteriza a dependência do classificador portador do método invocador para o classificador portado do método invocado. O portador do método invocador também pode possuir dependência para o objeto retornado pelo método invocado, caso este possua um tipo de retorno que não seja nulo (*void*).
- **Acesso a atributos de objetos:** o acesso a atributos de um objeto configura a dependência do classificador que realiza o acesso, para o classificador do objeto que possui o atributo acessado. Um acesso a um atributo pode ser a leitura, atualização, envio de mensagem, enfim qualquer referência ao atributo em um determinado método. Esta característica está relacionada a anterior, pois as dependências representadas pelas duas situações (acesso a atributos e chamada a métodos) são acarretadas por estruturas de código que são tratadas de forma semelhante pela linguagem;
- **Exibição das dependências entre pacotes:** a dependência entre elementos de pacotes diferentes constitui uma dependência entre os pacotes que contem estes elementos. A exibição das dependências entre pacotes é de suma importância para o projetista/programador, pois podem exibir problemas que podem comprometer toda a estrutura do programa, sua evolução e reutilização;

iii) Detecção da estrutura de pacotes

Mostra a organização do programa. O usuário pode avaliar, através da estrutura de pacotes, características de projeto como coesão e acoplamento. A ferramenta deve exibir a hierarquia extraída de alguma forma, preferencialmente, por uma estrutura em árvore;

iv) Detecção de relacionamentos entre elementos de pacotes distintos

Um ponto importante a ser avaliado é a capacidade da ferramenta em detectar relacionamentos entre classificadores de pacotes distintos, por uma questão de coerência de dados do código e informações do modelo. A ferramenta deve ser capaz de detectar relacionamentos entre elementos pertencentes a pacotes distintos e é importante para se avaliar o grau de dependência entre os pacotes;

v) Forma de tratamento de coleções não tipadas

Atributos podem dar origem a relacionamentos. Os atributos implementados com coleções não tipadas constituem um tópico de estudo especial neste trabalho. Para se inferir o tipo de relacionamento concebido na fase de projeto que deu origem à coleção do atributo, é necessário que o método no interior do código seja investigado, pois os tipos dos elementos adicionados na coleção não são conhecidos em tempo de compilação. Os tipos dos objetos adicionados na coleção devem ser analisados. A ferramenta, ao se deparar com um atributo definido por uma coleção não tipada, deve poder decidir qual a melhor forma de se representar esta construção do código no modelo. Existem duas formas possíveis de se tratar em modelo atributos definidos por coleções não tipadas: através de associações de cardinalidade múltipla (N), do classificador portador do atributo para os classificadores dos objetos adicionados na coleção durante a execução, ou através de dependências, do classificador portador do atributo para os classificadores dos objetos adicionados.

2.3.2 Ambiente de uso

Algumas características relativas ao ambiente de uso fornecido pelas ferramentas de engenharia reversa de Java para UML foram identificadas. A definição destas características ocorreu de forma incremental, após sessões intensas de uso das ferramentas avaliadas.

É importante destacar que algumas das características apresentadas podem não fazer parte necessariamente da ferramenta de engenharia reversa, mas podem ser fornecidas pelo ambiente ao qual ela está acoplada. Este tipo de característica é avaliado por estar diretamente ligado à ferramenta de engenharia reversa, embora possa servir a outras ferramentas do ambiente.

As características do ambiente de uso estão listadas e detalhadas a seguir:

i) Junção e revisão de modelos

Uma ferramenta de engenharia reversa deve possibilitar ao usuário a recuperação de fragmentos de modelos de maneira incremental, de forma que o entendimento do sistema seja evolutivo. Para isto, a junção de modelos deve ser permitida. Isto é, um modelo pré-existente, carregado no ambiente de modelagem, extraído por engenharia reversa ou mesmo

construído em uma engenharia direta (em um processo de modelagem realizado por um projetista), deve poder ser unificado a um novo modelo extraído do código. Durante a junção dos modelos, deve-se verificar também a existência de relacionamentos entre elementos (classificadores e pacotes) do código com elementos do modelo existente. Nesta verificação, devem ser revisados, por exemplo, os atributos que representam eventuais associações. Ou seja, atributos que representam novos relacionamentos no modelo unificado devem ser removidos de seus classificadores e devem ser convertidos em relacionamentos no modelo, virando assim pseudo-atributos. A revisão também deve se aplicar a outros tipos de relacionamentos. Isto mantém a consistência e atualização do modelo.

ii) Visualização seletiva de relacionamentos

O fato do processo de engenharia reversa ser semi-automatizado pode fazer com que muitas informações sejam exibidas no diagrama de classe. Muitas destas informações extraídas podem não ser úteis em determinados contextos de uso. Um exemplo claro de tal situação acontece quando um usuário deseja fazer um modelo a partir de um determinado código apenas para expor as classes e relacionamentos básicos extraídos a outros interessados. Provavelmente, para este usuário, não é útil a exibição das dependências, que acabam tornando, no contexto apresentado, o modelo em um emaranhado de ligações, o que dificulta o entendimento do modelo. É, portanto, desejável que o ambiente de modelagem permita habilitar e desabilitar determinados tipos de relacionamentos selecionados pelo usuário.

iii) Notificação de conflitos entre modelos

Os classificadores extraídos do código podem já fazer parte do modelo existente. Portanto, é importante que haja notificação deste conflito ao usuário e que sejam relatadas as possíveis diferenças entre estes classificadores, permitindo ao usuário a opção de substituição ou não do classificador do modelo considerado antigo. Como possíveis diferenças entre classificadores conflitantes, consideramos métodos (apenas a assinatura) e atributos.

iv) Meio de exportação alternativo

Uma característica desejável é a possibilidade de exportação do modelo não apenas para um ambiente de modelagem com diagrama de classes, mas também para um meio com uma semântica bem definida, como por exemplo XMI (XML Metadata Interchange)(OMG, 2000). A exportação do modelo neste formato possibilita o intercâmbio entre diferentes ambientes de modelagem que utilizam a UML como formato. Não é considerada aqui, a possibilidade de exportação para uma figura, pois esta funcionalidade pode ser realizada sem maiores dificuldades com auxílios de aplicativos que manipulam gráficos. É ressaltada aqui a semântica do meio para o qual o modelo é exportado.

v) Seleção recursiva de código fonte

O ambiente deve ser confortável para que o entendimento de um sistema seja o menos custoso possível. É interessante dispor de um meio de se selecionar os arquivos que compõem o programa, ou parte do programa, de uma maneira recursiva. Isto é, dado um diretório do sistema operacional, deve-se mostrar ao usuário a opção de seleção de todos os arquivos de código fonte contidos em diretórios de níveis inferiores.

2.4 Avaliação das Ferramentas Existentes

Nesta seção, é apresentada a avaliação de algumas ferramentas de destaque existentes no mercado e no meio acadêmico, que realizam a Engenharia Reversa de código Java para diagramas de classe da UML. Todas as ferramentas estão acopladas a ambientes CASE de suporte ao desenvolvimento baseado na UML. Para realizar esta avaliação, as ferramentas foram submetidas a estruturas de código que correspondiam a elementos de modelo esperados. Avaliou-se, então, a conformidade dos resultados apresentados por cada ferramenta com o resultado esperado. Foram elaborados códigos que, propositalmente, visavam refletir casos de mapeamento da linguagem Java em um modelo em UML. Em paralelo, analisou-se o ambiente de uso fornecido.

Foram avaliadas as seguintes ferramentas: JVision 4.1 (OBJECT INSIGHT, 2001), SoftModeler 3.3 (SOFTERA, 2001), GDPro 5.0 (EMBARCADERO TECHNOLOGIES, 2001), Structure Builder (WEBGAIN, 2001), Fujaba (FUJABA, 2001), Together (TOGETHER SOFT, 2001) e Rational Rose (RATIONAL, 2001).

2.4.1 JVision 4.1

Esta é uma ferramenta CASE simples que apresenta boa conformidade com a especificação da UML. A versão 4.1 não é gratuita. Para este estudo, utilizamos uma versão para avaliação (*trial*) com limitação por tempo de uso. Porém, no *site* da ferramenta, encontra-se disponível a versão 2.1 pública.

Ao fazer a Engenharia Reversa, é apresentado ao usuário apenas o diagrama com os classificadores extraídos.

As **associações** apresentam como informação apenas a cardinalidade e visibilidade. Apenas algumas **dependências** de uso foram capturadas: de parâmetro, variável local e instanciação. Dependências representadas em tipos de retorno, chamadas a métodos, acesso a objetos e *import* não foram detectadas. O fato de não detectar as dependências de *import* impossibilita a detecção de relacionamentos entre classes de diferentes pacotes, o que mostra uma grande limitação da ferramenta.

A ferramenta não possibilita a visualização da estrutura de projeto em forma de **árvore de pacotes**. Esta ferramenta apresenta ainda uma outra deficiência: não analisa as coleções **não tipadas**.

Na avaliação do **ambiente de uso**, verificou-se que a ferramenta permite realizar a engenharia reversa de códigos fonte em Java de maneira recursiva. A visão do modelo fica restrita ao diagrama, isto é, a ferramenta não permite uma visualização em uma estrutura de árvore, o que pode ser útil ao projetista. A opção de diagramação automática dos classificadores é permitida. A Figura 1 ilustra o diagrama de classes da ferramenta com algumas informações extraídas pela Engenharia Reversa. O diagrama de classes não exibe o **relacionamento entre os pacotes**.

Uma outra característica do diagrama é a possibilidade de **visualização seletiva dos relacionamentos**, o que pode tornar o entendimento do modelo mais confortável quando um grande número de classificadores e relacionamentos é extraído.

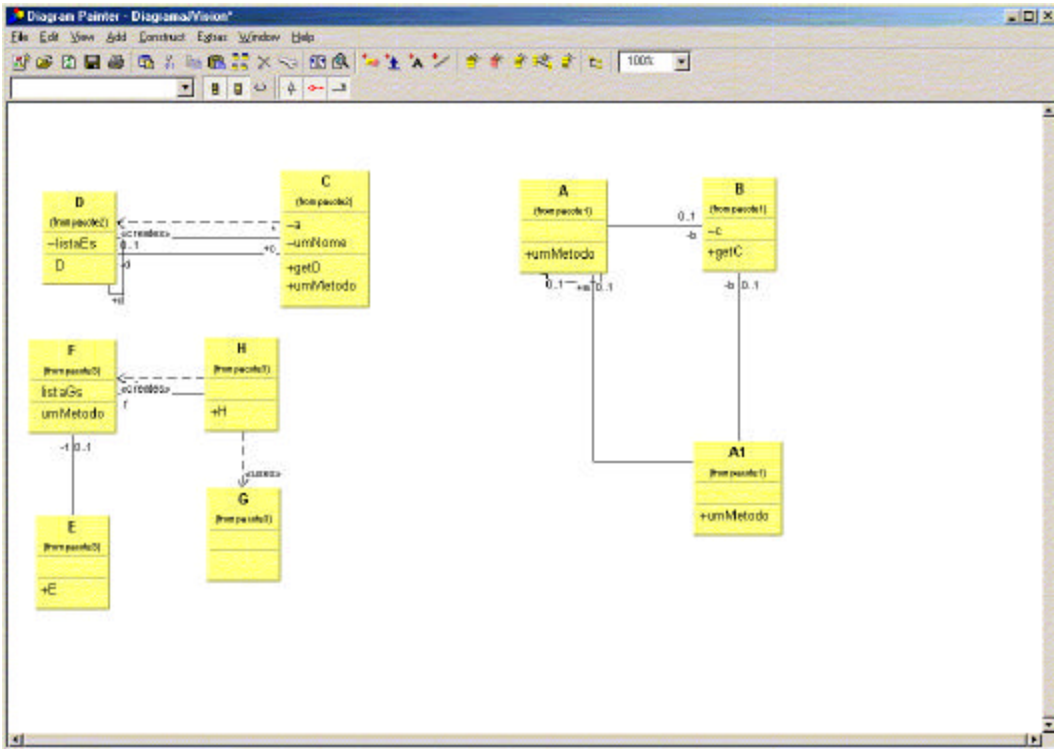

Figura 1: Engenharia Reversa na ferramenta JVision.

2.4.2 SoftModeler 3.3

Esta ferramenta não apresenta conformidade à especificação da UML quanto à diagramação, pois as classes são exibidas apenas com seus nomes. Os métodos e atributos não são exibidos. O modelo extraído pela ferramenta é diagramado automaticamente.

Esta ferramenta apresentou maior quantidade de informações sobre as **associações** (papel, navegabilidade etc.). As **dependências** e **coleções não tipadas** não são analisadas pela ferramenta.

Ao contrário da ferramenta JVision, esta consegue extrair **relacionamentos entre classes de pacotes diferentes**.

A ferramenta não possibilita a visualização da estrutura de projeto em forma de **árvore de pacotes**.

Na avaliação do **ambiente de uso**, verificou-se que entre as facilidades oferecidas há a possibilidade de **junção de modelos** e a manutenção da consistência entre os mesmos. A ferramenta é ilustrada na Figura 2.

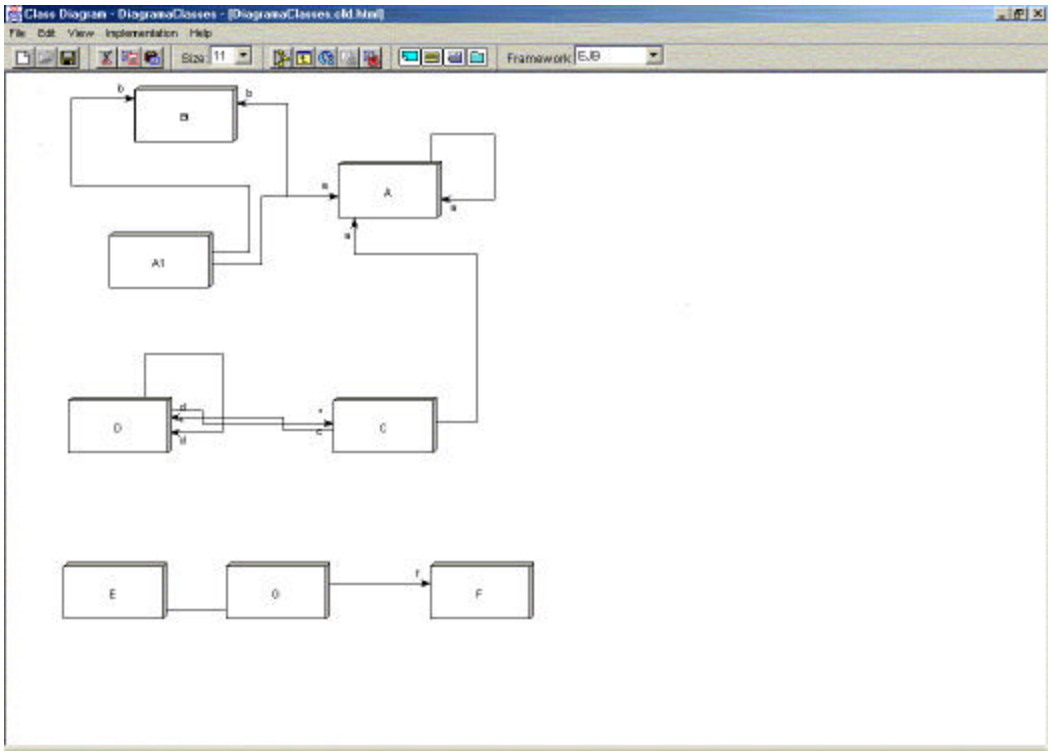

Figura 2: Engenharia Reversa na ferramenta SoftModeler.

2.4.3 GDPro 5.0

Entre algumas das facilidades da GDPro, encontram-se: a diagramação automática e visualização estrutural do modelo através de uma árvore de elementos.

As **associações** extraídas possuem poucas informações: apenas a navegabilidade e a cardinalidade simples. As coleções não são avaliadas.

A ferramenta também não exibe **dependências** entre os classificadores.

Uma qualidade da ferramenta é a detecção e exibição dos **pacotes e suas dependências** no diagrama, como é apresentado na Figura 3. Esta é a única ferramenta, dentre as avaliadas, que apresenta esta característica.

Na avaliação do ambiente de uso, verificou-se que a ferramenta permite a **seleção recursiva do código fonte** e também a **junção de modelos**, porém não mantém a consistência, considerando que os fragmentos de modelos mesclados (do código e um modelo pré-existente) pertencem sempre a modelos diferentes.

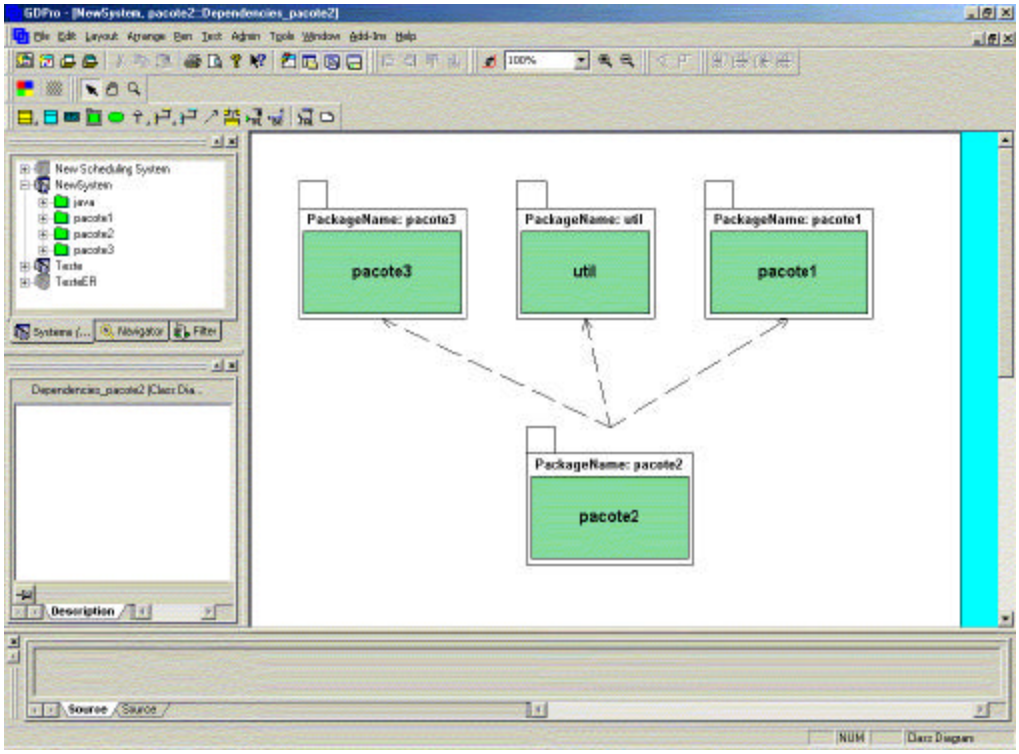

Figura 3: Diagrama dos pacotes e suas dependências na ferramenta GDPro 5.0.

2.4.4 Together 5.02

Esta ferramenta fornece integração com a máquina virtual Java, possibilitando que as tarefas de modelagem e codificação sejam sincronizadas. Alterações no código são refletidas no modelo e vice-versa. Pouquíssimas informações sobre os relacionamentos foram exibidas. As **associações** apresentam apenas a navegabilidade e nenhum tipo de **dependência** é extraído. Um outro ponto fraco é a não avaliação de coleções **não tipadas**.

Uma árvore com a **estrutura de pacotes** é exibida no lado esquerdo da tela (Figura 4), o que permite uma melhor visualização do projeto. A seleção de cada pacote faz com

que um diagrama com as classes do pacote selecionado seja exibido no lado direito da tela. A ferramenta extrai e exibe **relacionamentos entre diferentes pacotes**.

Para se extrair um modelo a partir de código Java, é necessário que se abra um projeto de trabalho na ferramenta, impedindo a **junção de modelos**. Após a abertura do projeto, a ferramenta permite que o diretório de trabalho tenha seus arquivos recuperados de maneira recursiva (**seleção recursiva**). A ferramenta realiza então a Engenharia Reversa sobre esses arquivos.

Uma característica interessante da ferramenta é a possibilidade de **exportação** do modelo no formato XMI, permitindo o intercâmbio de modelos com outras ferramentas de modelagem. Dentre as ferramentas avaliadas, esta foi a única a apresentar esta característica.

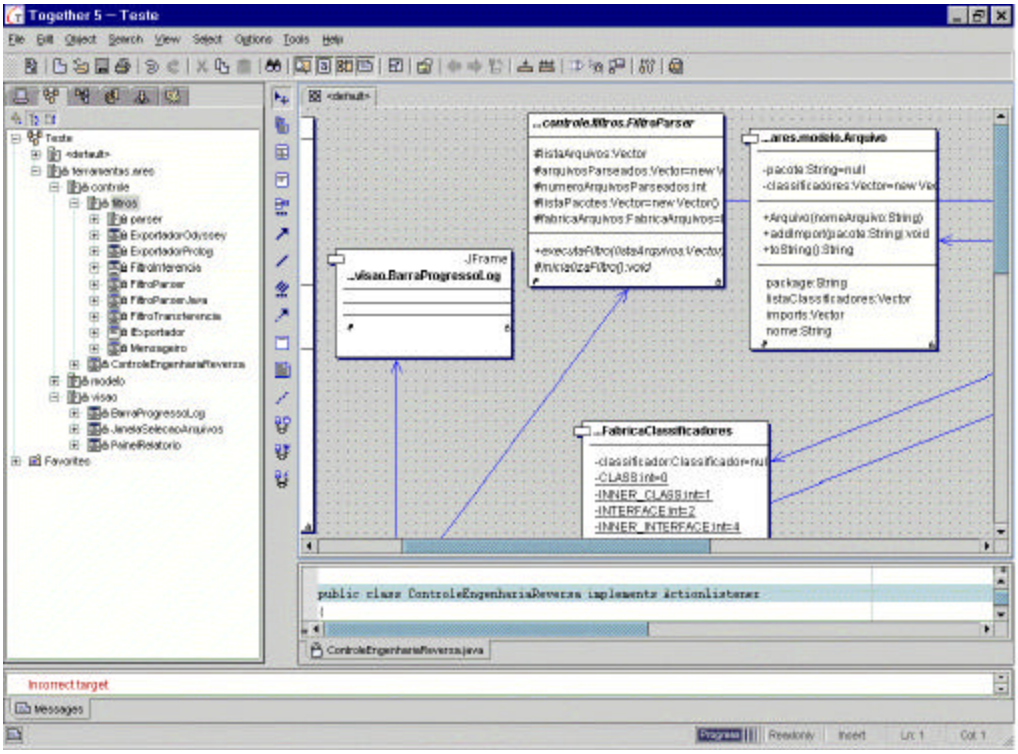

Figura 4: Engenharia reversa na ferramenta Together 5.02.

2.4.5 Structure Builder 3.3

Na avaliação da **capacidade de detecção**, verificou-se que a ferramenta não extrai das **associações** a cardinalidade quando derivada de estruturas em *arrays*, mutabilidade e papéis. Não são extraídos, também, **relacionamentos entre pacotes distintos** e nenhum tipo de **dependência**.

Uma das características mais interessantes é o fato de ser a única ferramenta que detecta associações implementadas com coleções **não tipadas** (por exemplo, *Vector*). A ferramenta se comporta da seguinte forma: investiga os objetos adicionados na coleção no interior dos métodos e exibe uma associação de cardinalidade N do classificador portador da coleção para os classificadores dos objetos adicionados. Este comportamento tem a desvantagem de considerar o pressuposto de que o programador tem sempre a intenção de implementar uma associação com o uso desta coleção, o que pode não ser verdadeiro. Uma coleção não tipada, do ponto de vista de projeto, pode implementar outros conceitos que não sejam necessariamente uma associação com cardinalidade N.

Na avaliação do ambiente de uso, verificou-se que a ferramenta permite a **junção de modelos**, mantendo a consistência. Outras características interessantes foram notadas nesta ferramenta: a **seleção recursiva de arquivos**, **notificação de conflitos** ao usuário quando um classificador extraído tem o mesmo nome de um classificador existente no mesmo pacote (porém não exibe as diferenças), possui ambiente integrado com a máquina virtual Java, permitindo a compilação e atualização automática do modelo quando se altera o código e vice-versa.

A Figura 5 ilustra o diagrama de classes extraído a partir de um projeto exemplo feito em Java.

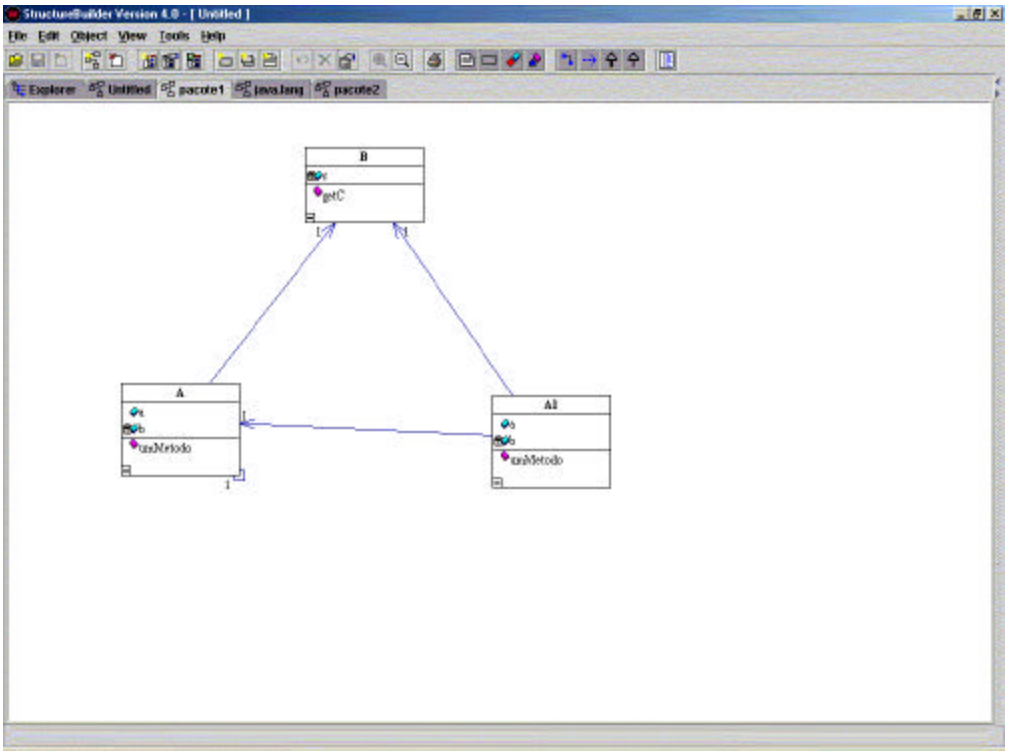

Figura 5: Engenharia reversa na ferramenta Structure Builder.

2.4.6 Fujaba 2.5.4

Fujaba é uma ferramenta acadêmica, distribuída livremente e com código aberto, que se propõe, principalmente, a realizar *round-trip engineering*, isto é, realizar engenharia reversa e direta de forma integrada, utilizando Java e UML. Esta ferramenta reconstitui diagramas de classe a partir do código de uma maneira que não preenche alguns dos requisitos identificados. Uma funcionalidade provida por ela é a criação de diagramas de estados a partir do código. A criação destes diagramas, porém, se baseia em convenções de código, o que é uma hipótese nada recomendável quando se trabalha com projetos reais. Outra característica da ferramenta Fujaba é uma infra-estrutura de suporte a padrões de projeto, permitindo a instanciação e detecção de padrões de projeto. Como algumas das ferramentas citadas, esta diagrama automaticamente o modelo extraído do código.

A **capacidade de detecção** da ferramenta não se mostrou em um nível adequado. As **dependências** não foram capturadas. O mesmo ocorreu com **associações** entre classes de pacotes distintos. O diagrama não apresentou indicações de ocorrência de cardinalidades múltiplas, tanto com *arrays* quanto com coleções não tipadas, quando submetida a

situações que exigiam tal comportamento. Outro ponto fraco da ferramenta é a não extração da **estrutura de pacotes** do código analisado.

A **junção de modelos** pré-existentes com modelos extraídos do código é permitida, mantendo-se, inclusive, a consistência entre os modelos. Esta foi a única característica presente das identificadas relativas ao **ambiente de uso**.

A Figura 6 ilustra o diagrama de classes extraído pela ferramenta.

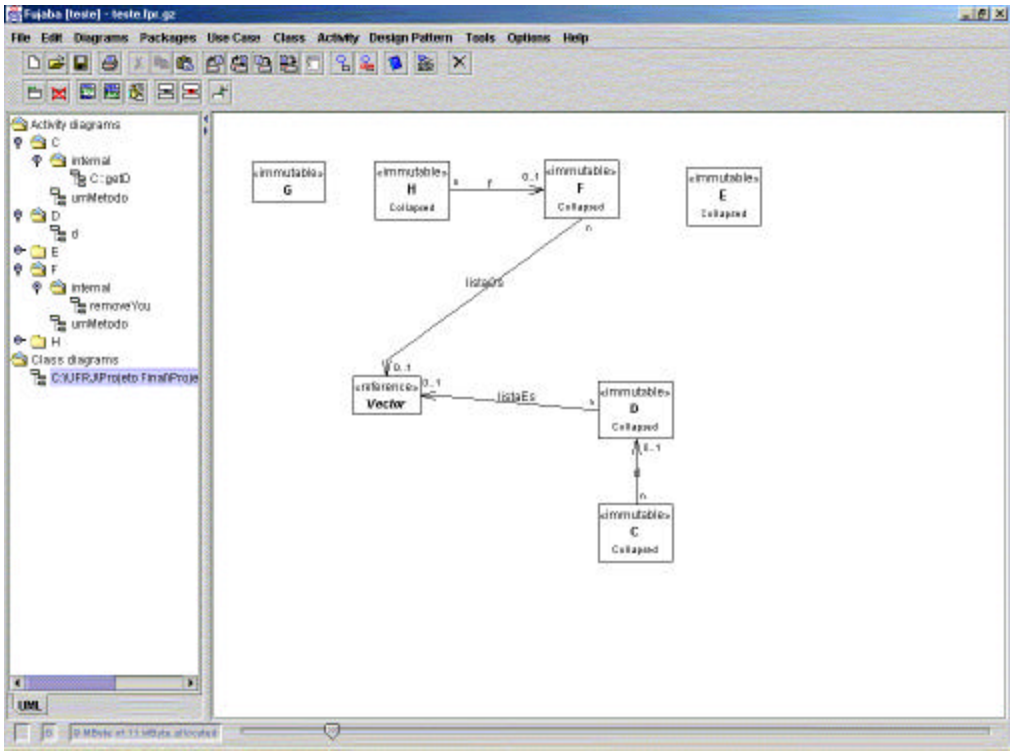

Figura 6: Engenharia reversa na Ferramenta Fujaba.

2.4.7 Rational Rose 2000

A ferramenta Rose provê suporte a engenharia reversa de códigos em Java, C++ e Delphi. Esta é, certamente, a mais conhecida e difundida entre as ferramentas avaliadas.

Na análise da **capacidade de detecção**, a ferramenta apresentou algumas deficiências, tais como: as **dependências** não são capturadas e algumas informações importantes das **associações** não são exibidas, como cardinalidade e mutabilidade. Nenhum

tipo de **coleção** é avaliado pela ferramenta. Além disso, os **relacionamentos entre classes de pacotes** distintos não são detectados.

A ferramenta exibe uma árvore dos elementos capturados e permite que os elementos sejam arrastados para o diagrama (Figura 7). Na árvore, a **estrutura de pacotes** completa é exibida.

Na análise do **ambiente de uso**, verificou-se que é permitida apenas a **junção de modelos**, porém a consistência dos dados não é mantida, e é possível realizar a **seleção recursiva** do código fonte..

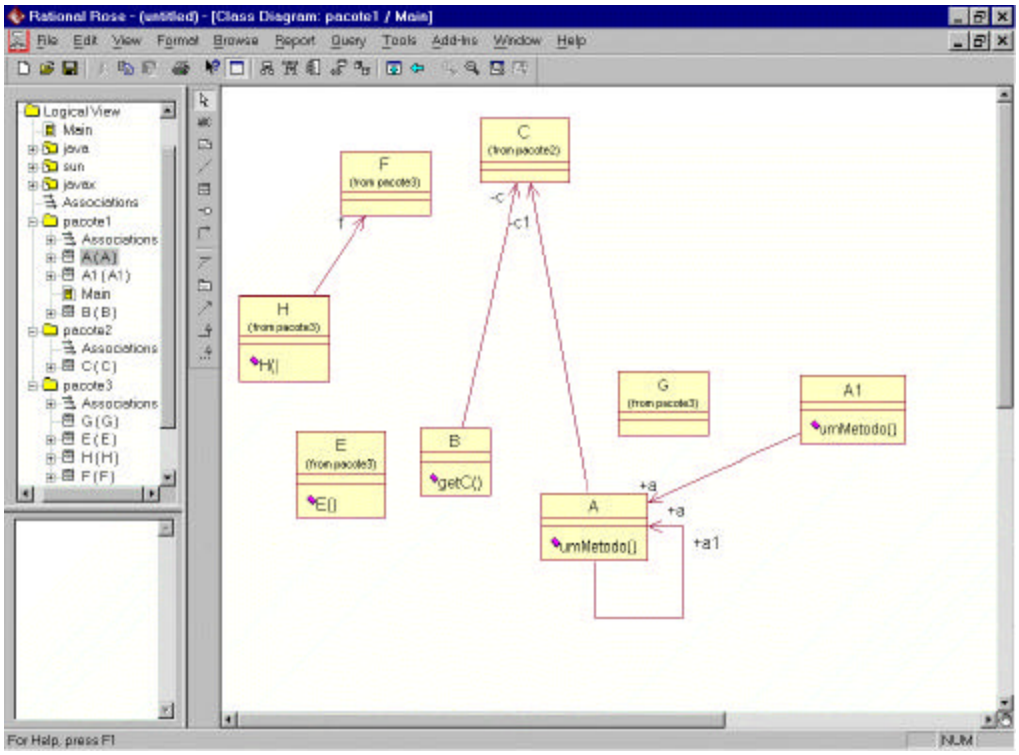

Figura 7: Engenharia Reversa na ferramenta Rose 2000.

2.5 Quadro Comparativo

A Tabela 1 exibe um resumo sobre a avaliação de acordo com as características que dizem respeito à capacidade de detecção das ferramentas.

		Ferramentas de Engenharia Reversa de Java para UML						
		Jvision	**SoftModeler**	**GDPro**	**Together**	**Fujaba**	**Rose**	**StructureBuilder**
Associação	Cardinalidade Simples	✓	✓	✓				✓
	Cardinalidade N (com *Array*)	✓	✓					
	Navegabilidade		✓	✓	✓	✓	✓	✓
	Papel		✓				✓	
	Mutabilidade							
	Visibilidade					✓	✓	
Dependência	Importação de Elementos							
	Passagem de parâmetros	✓						
	Retorno de objetos	✓						
	Instanciação de objetos	✓						
	Declaração de variáveis locais							
	Conversões de tipo (*cast*)							
	Chamadas/Acesso a atributos							
	Dependências entre pacotes			✓				
Detecção/Exibição da Estrutura de pacotes				✓	✓		✓	✓
Relacionamentos entre elementos de pacotes distintos		✓	✓		✓	✓		
Forma de tratamento de atributos derivados de coleções não tipadas								Cria uma associação com cardinalidade N para classificadores adicionados na coleção

Tabela 1: Quadro comparativo das ferramentas considerando a detecção das principais estruturas.

Pode-se notar que nenhuma das ferramentas apresentou todas as características identificadas. Dois dados interessantes podem ser observados: apenas uma ferramenta (StructureBuilder) trata de alguma forma os atributos derivados de coleções não tipadas, que se representam como um elemento importante no modelo; e o requisito de identificação de dependências entre pacotes também foi resolvido por uma ferramenta apenas (GDPro). Esta detecção é fundamental para que o usuário possa verificar se a estrutura do sistema, como um todo, está comprometida ou não.

A Tabela 2 exibe o quadro comparativo das ferramentas analisadas em relação às características do ambiente de uso. Nesta tabela, nota-se que nenhuma das ferramentas fornece, segundo a visão deste trabalho, um ambiente de uso totalmente amigável.

	Ferramentas de Engenharia Reversa de Java para UML						
	Jvision	**SoftModeler**	**GDPro**	**Together**	**Fujaba**	**Rose**	**StructureBuilder**
Junção e revisão de modelos[3]	parcial	✓	parcial		✓	parcial	✓
Visualização seletiva de relacionamentos	✓			✓			✓
Notificação de conflitos entre modelos							parcial[4]
Meio de exportação alternativo				XMI			
Seleção recursiva de código fonte	✓		✓	✓		✓	✓

Tabela 2: Quadro comparativo, considerando as funcionalidades do ambiente de uso.

Neste capítulo, foram identificados os principais problemas envolvidos na atividade de Engenharia Reversa de código Java. Junto aos problemas identificamos características de uma ferramenta que realizasse tal atividade. Com base nestas características, apresentamos uma comparação de algumas ferramentas de destaque no mercado e no meio acadêmico. Verificamos que estas ferramentas não realizam a Engenharia Reversa de forma completa. Diversos pontos são apontados, na comparação, que precisam ser solucionados com o objetivo de minimizar as deficiências apresentadas.

[3] Valor considerado parcial se a revisão dos relacionamentos não é feita.
[4] Não exibe diferenças entre os classificadores. Apenas notifica o usuário da existência do conflito.

No próximo capítulo, são apresentadas algumas soluções propostas que visam prover formas de mapeamento entre Java e UML. Estuda-se ainda a viabilidade da implementação de tais soluções. Com base nas formas de mapeamento exibidas, apresentamos também uma arquitetura de uma ferramenta que objetiva realizar a Engenharia Reversa de Java para UML de tal modo que as deficiências desta atividade sejam minimizadas.

3 Abordagem Proposta

3.1 Introdução

Neste capítulo, apresentamos uma abordagem para a Engenharia Reversa de códigos orientados a objetos feitos em Java, tendo como alvo a criação de modelos UML. A abordagem proposta visa minimizar as deficiências identificadas nas encontradas na literatura, apresentadas no capítulo 2.

Inicialmente, discutimos a estratégia adotada para que as características identificadas no capítulo anterior possam ser tratadas pela abordagem. São apresentadas, então, formas de mapeamento de estruturas da linguagem Java em elementos do metamodelo da UML. Estas formas de mapeamento são utilizadas pela abordagem como solução para obtenção das características identificadas e determinam a direção da implementação da abordagem. Em seguida, é proposta uma arquitetura para uma ferramenta de Engenharia Reversa de códigos em Java, capaz de exportar os modelos extraídos para meios que possuam suporte a representação de modelos em UML

Para que o mapeamento entre as estruturas da linguagem Java em elementos de modelo da UML seja explicado, utilizaremos um pequeno exemplo para ilustrar as diversas situações de mapeamento. Este exemplo é referenciado ao longo deste e do próximo capítulo. O exemplo foi retirado de um trabalho realizado na disciplina de Projeto de Sistemas de Informação, ministrada no ano de 2001 (WERNER *et al.*, 2001). Trechos de código e fragmentos do modelo de um protótipo, construído nesta disciplina, são utilizados para ilustração. O protótipo representa um mini-sistema de controle de contas corrente, movimentações sobre as contas e clientes.

3.2 O Mapeamento Java-UML

Nesta seção, destaca-se o mapeamento desejado de estruturas de código em Java que são importantes para a extração de modelos em UML. Utiliza-se como foco do

mapeamento as estruturas do modelo de classes. Para cada estrutura de código, são ilustrados trechos de código que visam exemplificar cada situação de mapeamento.

A linguagem Java possui sintaxe semelhante à da linguagem C++. Um arquivo do programa possui extensão ".java" e pode conter um ou mais classificadores. Antes da declaração dos classificadores, deve-se ter a declaração do pacote ao qual o arquivo e seus classificadores pertencem (através da declaração "**package"**). Em seguida, através de declarações "**import**", devem ser explicitados os pacotes, ou classificadores, que serão utilizados no código dos classificadores definidos no arquivo.

```
package modelo; //Declaração do pacote onde o arquivo está
import java.util.Date; //Pacotes importados
public class Operacao
{
  private String codigo; //Atributos (…)
  /** Construtor. **/
  public Operacao (String aCodigo, String aNome, float aValor)
  {
    (...)
  }
  /** Um método **/
  public synchronized String getCodigo ()
  {
    return codigo;
  }
  /** Exemplo de classe aninhada*/
  public static class Deposito extends Operacao
  {
    public Deposito (float valor)
    {
      super ("DPO", "Deposito", valor);
    }
  }
}
```

Figura 8: Trecho de código exemplificando a estrutura da linguagem Java.

Após as declarações iniciais dos arquivos, seguem as declarações e especificações dos classificadores. Cada classificador tem duas partes: a primeira para declarações de atributos e a segunda para declarações e implementações de métodos. O trecho de código apresentado na Figura 8 ilustra a estrutura genérica de um programa em Java. Mais detalhes sobre a especificação da linguagem Java podem ser obtidos em (SUN MICROSYSTEMS, 2001).

A seguir, reapresentamos os elementos da UML, que foram exibidos na seção 2.2. Porém, desta vez, enfatiza-se na manifestação em código desses elementos e são expostas formas de mapeamento das estruturas de código nos elementos do modelo.

3.2.1 Pacote

Na UML, o pacote tem uma aplicação mais ampla do que a empregada na linguagem Java, tendo em vista que um pacote pode conter outras visões do sistema como diagramas de use cases, componentes etc. Do ponto de vista estrutural, um pacote pode conter os classificadores e seus relacionamentos, diagramas de classe e outros pacotes. A sintaxe, em Java, para o nome de um pacote é dada por uma cadeia de nomes dos pacotes separada por ponto ("."), sendo que o pacote mais a esquerda na cadeia contém o pacote subseqüente. Por exemplo, "**componentes.janelas.dialogos**" é o nome do pacote "**dialogos"**, que está contido em "**janelas"**, que por sua vez está contido em "**componentes"**.

Na linguagem Java, a abstração de pacotes é correspondente a um diretório do sistema operacional. Portanto, se um pacote chamado "**persistencia**", por exemplo, contém os classificadores responsáveis pela interação com o banco de dados, estes classificadores devem estar fisicamente contidos em arquivos neste diretório. O arquivo que contém o(s) classificador(es) deve possuir a declaração "package persistencia" antes da declaração do(s) classificador(es) correspondente(s).

3.2.2 Classificador

Conforme visto no capítulo anterior, de acordo com a especificação da UML (OMG, 2000), um classificador pode ser uma classe, uma interface, ou um tipo primitivo de dado (*DataType*). Os tipos primitivos não apresentam identidade e costumam ser representados

em diagramas de classe como atributos de classes e interfaces, quando compõem o estado destes classificadores. Neste trabalho, consideramos como classificadores apenas as classes e interfaces. Os tipos primitivos não são descartados, mas não possuem a mesma representação dos outros classificadores. Quando aparecem como atributos de uma classe em Java, eles são expostos, em UML, também como atributos. Isto é, tipos primitivos não derivam relacionamentos no modelo. Esta é a representação utilizada em todas as ferramentas avaliadas.

Um classificador possui um conjunto de modificadores, que são listados abaixo com os respectivos valores possíveis:

- **Tipo**: interface (*interface*) ou classe (*class*).
- **Visibilidade**: público (*public*), privado (*private*) ou protegido (*protected*).
- **Indicador de Instanciação**: concreto (na ausência de indicador) se pode ser instanciado ou abstrato (*abstract*), se não pode ser instanciado. As interfaces são sempre abstratas.
- **Mutabilidade** (*changeability*): indica se a classe pode sofrer polimorfismo (na ausência de indicador) ou não (*final*).

A linguagem Java possui uma estrutura de aninhamento de classificadores. É possível definir, através da linguagem, classes ou interfaces em vários níveis de aninhamento. A especificação da UML, na sua atual versão, não contempla este tipo de construção. No caso dos classificadores aninhados, existe a possibilidade de se ter mais um modificador presente na sua declaração: um indicador de escopo, que pode assumir valores de classe (*static*) ou instância (na ausência do modificador), indicando que o classificador aninhado pertence apenas ao classificador que o encapsula, ou se pertence às instâncias de seu encapsulador.

O mapeamento do classificador (e seus modificadores) é feito baseado na análise da sua declaração. Pela especificação da linguagem Java, a declaração de um classificador é feita da forma "**class** | **interface** modificadores nomeDoClassificador".

Um exemplo da declaração de um classificador seria:

```
public class GerentePersistencia //Campo de declaração
{
    ... //Corpo da classe
}
```

Nesta situação, o mapeamento deve ser feito de acordo com os modificadores declarados: o classificador é do tipo classe, tem visibilidade pública, não é abstrato e tem o nome "GerentePersistencia".

3.2.3 Relacionamentos

Nesta subseção analisamos, primeiramente, o mapeamento dos relacionamentos de herança e realização, que são os mais triviais. Em seguida, apresentamos formas de detecção e mapeamento dos relacionamentos mais complexos: associações e dependências, com detalhes sobre suas propriedades.

3.2.3.1 Herança

Este é um relacionamento trivial de ser extraído, tendo em vista, neste caso, a correspondência direta Java/UML. A palavra reservada *extends*, utilizada na declaração de um classificador, indica a existência de uma herança do classificador, declarado em relação ao classificador presente após a palavra reservada, conforme mostra o exemplo a seguir.

```
class Deposito extends Operacao
```

Neste exemplo, o classificador **Deposito** deve ser criado no modelo. Se **Operacao** já existe no modelo, ou está sendo extraído a partir do código, o relacionamento deve ser criado no modelo.

3.2.3.2 Realização

A palavra reservada *implements*, de forma semelhante à herança (com *extends*), indica a realização do classificador declarado em relação à interface presente após a palavra reservada.

Se a ferramenta se deparar com o código:

```
class Sistema implements ISistema
```

deve se comportar de forma análoga à descrita para a herança, criando um relacionamento de realização. O exemplo mostra que o classificador **Sistema** realiza a interface **ISistema**.

3.2.3.3 Associação

Uma associação define um relacionamento semântico entre pelo menos dois classificadores. Os classificadores participantes exercem papéis de origem e de destino da associação de acordo com sua posição no relacionamento. Uma associação pode conter os papéis dos participantes, a visibilidade dos mesmos e a navegabilidade.

As associações que partem de um determinado classificador são detectadas, a partir do código, analisando-se os atributos deste classificador. O classificador analisado, neste caso, é fonte do relacionamento. Os atributos do classificador são potenciais candidatos a serem o destino da associação. Para que ele seja qualificado como destino de uma associação, analisa-se o seu tipo declarado. As associações são criadas com classificadores que possuem o tipo declarado no atributo e que estejam presentes no modelo. Os atributos que são considerados relacionamentos não devem aparecer no modelo, pois, de acordo com a especificação da UML, estes são pseudo-atributos e apenas a presença do relacionamento já indica que sua implementação em código deva ser um atributo.

A associação possui algumas propriedades, que são inerentes aos seus participantes. Para cada propriedade da associação, temos um mapeamento esperado. As principais propriedades são detalhadas a seguir:

- **Navegabilidade**

Indica a direção da associação. A presença do pseudo-atributo indica que a direção da associação é da origem para o destino.

O mapeamento de relacionamentos bidirecionais é menos trivial do que parece a primeira vista e exige investigações de código no interior das operações, para se verificar. GOGOLLA *et al*., (2000) analisam detalhadamente esse mapeamento, onde são estudadas condições indicadoras para a criação de associações inversas (com a navegabilidade bidirecional). Neste trabalho, as associações são criadas em apenas um sentido. Ou seja, se existe conceitualmente uma associação inversa entre classificadores A e B, da forma "A $\leftrightarrow$ B", são analisados e expostos os relacionamentos "A$\rightarrow$ B" e "B $\rightarrow$ A". A inferência da associação inversa de "A $\leftrightarrow$ B" pode não ser correta, dependendo do contexto. Este é mais um caso onde a semântica de projeto pode ser difícil de se recuperar através da implementação.

- **Escopo**

Indica, de acordo com o atributo, se o classificador destino está presente em uma classe ou em uma instância (objeto).

O mapeamento é feito baseado no modificador de o escopo do atributo. Se o modificador *static* estiver presente, então o pseudo-atributo declarado (o destino da ligação) tem o escopo de classe. Caso contrário, tem escopo de objeto.

Por exemplo, considere a declaração do atributo abaixo:

```
static JanelaPrincipal janela;
```

que indica uma associação com o classificador **JanelaPrincipal**, sendo que o atributo **janela** tem escopo de classe.

- **Multiplicidade**

Indica uma faixa de possíveis cardinalidades que um conjunto pode assumir.

O mapeamento deve considerar se o atributo é ou não uma coleção. Se o tipo do atributo declarado for um classificador (classe ou interface, neste contexto), que não seja

uma coleção, então a cardinalidade deve ser simples. O ajuste de cardinalidades também é menos trivial do que parece a primeira vista (GOGOLLA *et al*., 2000). Se o atributo for uma coleção, deve-se considerar a cardinalidade múltipla ("*", "0..", "1..*", em UML). Em Java, uma coleção de objetos pode ser tipada ou não. Um *array*, por exemplo, indica que a coleção tem o tipo dos objetos que nela serão adicionados, definido em tempo de compilação.

Considere o classificador **Sistema** com os atributos:

```
JanelaPrincipal janela;
Cliente[] clientes;
```

Neste exemplo, há uma associação de cardinalidade simples de **Sistema** para a classe **JanelaPrincipal**, há também uma segunda associação de **Sistema** para **Cliente**, mas esta última apresenta cardinalidade múltipla.

O mapeamento para atributos que não são coleções ou que são coleções definidas por *arrays* (coleções tipadas) é trivial de ser feito. Em ambos os casos, deve-se criar uma associação do classificador que contém os atributos para os classificadores que definem o tipo dos atributos. Se o atributo não está em uma coleção, a associação deve possuir cardinalidade simples, indicadas no diagrama por "1" ou "0..1". A escolha da alternativa correta também depende de conceitos envolvidos no projeto. Neste trabalho, adota-se sempre o indicador "0..1" para cardinalidades simples. (GOGOLLA *et al*., 2000) estudam indicações que possam auxiliar a escolha. Se o atributo é definido por um *array*, a cardinalidade é múltipla ("0..n","n","1..n" etc.). Neste trabalho, adota-se sempre o indicador "0..n" para cardinalidades múltiplas.

O caso das coleções não tipadas deve ser considerado com maior cuidado e por isso reservamos um tópico de estudo reservado neste capítulo.

- **Mutabilidade** (*Changeability*): Indica se o participante pode ser ou não alterado.

Este indicador pode assumir três valores: (i) mutável (*changeable*), quando não há restrições à modificação; (ii) congelado (*frozen*), quando nenhuma referência pode ser

alterada; e (iii) apenas-inclusão (*add-only*), quando referências podem ser apenas adicionadas.

A linguagem Java não possui construção para classificar um atributo como apenas-inclusão (*add-only*). Em Java, é possível dizer se o atributo é mutável ou congelado a partir da presença ou não da palavra reservada *final* na declaração do atributo candidato a destino da associação. Se a palavra *final* estiver presente, então o alvo da associação é congelado (*fronzen*). Caso contrário, será mutável (*changeable*).

O exemplo abaixo indica que, caso o classificador **CaixaDialogo** exista no modelo, será criada uma associação do classificador portador do atributo para o classificador **CaixaDialogo**, que terá papel destino "**dialogo"** e indicador de mutabilidade 'congelado' (*fronzen*).

```
final CaixaDialogo dialogo;
```

- **Visibilidade**: Indica em que nível (público, privado ou protegido) os participantes são visíveis externamente.

O indicador de visibilidade é mapeável diretamente a partir do modificador de acesso a atributos (*public*, *private* ou *protected*).

- **Papel**: Representa o pseudo-atributo do participante na ligação.

A tradução é feita diretamente a partir do nome do atributo, candidato a destino da associação, na sua declaração.

No exemplo, o papel do destino da associação seria "**janela**".

```
private JanelaPrincipal janela;
```

3.2.3.4 Dependência

As dependências podem ocorrer entre pacotes e classificadores. A especificação da UML propõe sete tipos de dependências, que são expostos no modelo através de estereótipos. Dentre os tipos especificados, apenas a dependência de permissão (*Permission*) e uso (*Usage*) são detectáveis a partir de um código Java. Vejamos em

detalhes estes dois tipos e os respectivos subtipos mais importantes definidos pela especificação da UML:

i) Permission

Esta dependência garante a um elemento de modelo acessar elementos em outro *namespace*[5]. Pode ter os seguintes estereótipos pré-definidos:

- *access* - é uma dependência entre pacotes em que os elementos públicos acessados não fazem parte do *namespace* do pacote que contém o elemento de modelo fonte da dependência. Um pacote em Java não apresenta nenhum código que possa especificar esta dependência. É apenas um diretório do sistema operacional que contém classificadores. Porém, esta dependência pode acontecer quando um classificador de um pacote realiza um acesso a um classificador de um outro pacote. Neste caso, o primeiro pacote é dependente, por acesso, do segundo pacote. Este acesso pode ser realizado, por exemplo, para a declaração de uma variável, passagem de parâmetro ou uma conversão forçada de tipo. O exemplo a seguir ilustra uma dependência de acesso acarretada por uma passagem de parâmetro. A não importação da classe passada como parâmetro (**GerentePersistencia**) tornou necessário o seu acesso por meio do seu nome completo (o seu caminho). Apesar de acessada, a classe não pertence o *namespace* do classificador portador do método.

```
public Sistema (persistencia.GerentePersistencia gerente)
```

- *import* - os elementos públicos acessados passam a fazer parte do *namespace* do pacote que contém a fonte da ligação. Em Java, a importação de elementos de modelo (neste caso, pacotes e classificadores) se dá através da palavra reservada *import* (similar à diretiva "*#include*" em C++) e deve constar no arquivo antes das declarações dos classificadores. A presença desta declaração configura uma dependência do classificador importador para os pacotes importados. Portanto, considera-se que, para que exista uma dependência com o estereótipo <<*import*>>,

[5] *Namespace*, segundo a especificação da UML, é a parte do modelo que contém elementos de modelo. No contexto específico de um diagrama de classes, um *namespace*, assume o papel do elemento *pacote*.

deve existir esta declaração no arquivo do classificador fonte. O exemplo destacado ilustra a importação de todas as classes do pacote **componentes**.

```
import componentes.*;
```

- *friend* - possibilita o acesso a elementos de outro *namespace* independente da visibilidade. Este caso não é contemplado pela linguagem Java, e sua análise deve ser desconsiderada.

ii) Usage

Esta dependência decorre de um relacionamento no qual um elemento requer um outro elemento para sua implementação ou operação.

Este tipo de dependência cobre o acesso a parâmetros, variáveis locais, atributos, conversões forçadas de tipos (*casts*), retorno de objetos, chamadas de métodos e acesso a atributos de objetos.

A especificação sugere alguns estereótipos pré-definidos para esta dependência, entre os mais importantes destacamos:

- <<*call*>> - representa a dependência de um classificador que invoca uma determinada operação para um classificador que contém uma operação chamada.

A invocação de chamadas deve ser considerada durante a detecção de dependências do tipo <<*call*>>.

Considere a chamada a seguir:

```
gerentePersistencia.getClientes();
```

Esta chamada indica a existência de uma dependência do tipo <<*call*>> da operação que possui esta invocação, com o classificador do objeto **gerentePersistencia** e do tipo retornado por `getClientes()`.

- *create* - indica que o classificador origem instancia um objeto do tipo do classificador destino da dependência.

Em Java, a instanciação de objetos é feita utilizando-se a palavra reservada *new*. Portanto, cada ocorrência da palavra *new* indica dependências com estereótipo <<*create*>>

As dependências restantes são avaliadas de acordo com a situação correspondente (através da declaração de parâmetros, variáveis locais, atributos, conversões forçadas de tipos e tipos de retorno).

É importante ressaltar que a detecção de um relacionamento a partir da fonte deste relacionamento não implicará necessariamente na exibição deste no modelo, dado que o participante destino pode não estar presente no código. Caso o participante destino seja extraído por uma posterior engenharia reversa do código, o modelo deverá refletir o aparecimento deste participante, exibindo a ligação automaticamente. Esta característica de atualização do modelo visa manter a consistência entre o modelo e o código, permitindo que se faça a engenharia reversa de maneira incremental.

3.2.3.5 Extração de relacionamentos a partir de coleções não tipadas

As linguagens orientadas a objetos apresentam estruturas de dados que mantém conjuntos de referências para objetos em memória, cujo tipo não é definido em tempo de compilação, mas em tempo de execução. Ou seja, o objetivo destas estruturas é a manipulação de coleções de objetos. Em Java, as coleções não tipadas implementam o *framework Collection*, que define uma interface de manipulação de objetos em uma coleção. **Vector** e **Hashtable**, por exemplo, implementam esta interface. Destacamos dois métodos principais definidos nesta interface: (i) `void add(Object o)`, que adiciona um objeto genérico na coleção, e (ii) `Object remove(int index)`, que retorna um objeto genérico da coleção, na posição determinada. Existem outras variações destes métodos, mas a funcionalidade é a mesma, adição ou remoção de objetos.

Todo objeto Java em memória é do tipo **Object**. Portanto, uma coleção não tipada mantém referências para objetos genéricos. A definição do tipo do objeto se dá apenas na recuperação do objeto, através da realização de uma conversão forçada de tipo (*cast*).

Tomemos como exemplo a estrutura **Vector** e o trecho de código mostrado na Figura 9.

```
public class Departamento
{
       Aluno umAluno = new Aluno(“João”,123);
       Professor umProfessor = new Professor(“Maria”,456);
       Funcionario umFuncionario = new Funcionario(“Ana”,789);
       Vector umaColecao = new Vector();

       public void umMétodo()
       {
           umaColecao.add(umAluno);
           umaColecao.add(umProfessor);
           umaColecao.add(umFuncionario);

       }
}
```

Figura 9: Trecho de código ilustrando o uso de coleções não tipadas.

No momento da recuperação do objeto **umProfessor** deverá ser feita uma conversão de tipo para Professor, como é ilustrado abaixo:

```
Aluno outroAluno = (Aluno)umaColecao.remove(0);
```

Se a hierarquia da Figura 10 fosse implementada, o objeto poderia ser recuperado da seguinte forma:

```
Aluno outroAluno = (Pessoa)umaColecao.remove(0);
```

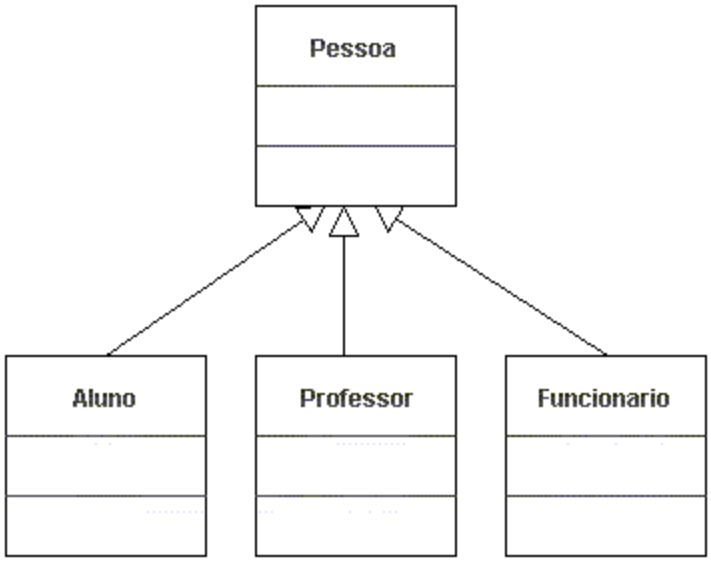

Figura 10: Hierarquia de classes usada como exemplo.

Na primeira forma apresentada, o modelo de projeto especificado pode ter representado uma associação de cardinalidade N com a classe pessoa (Figura 11).

Figura 11: Uma associação que originou a coleção.

Na segunda forma, a representação no modelo de projeto seria de uma dependência para os classificadores dos objetos adicionados na coleção, mesmo os que não fizessem parte da hierarquia de **Pessoa** (Figura 12).

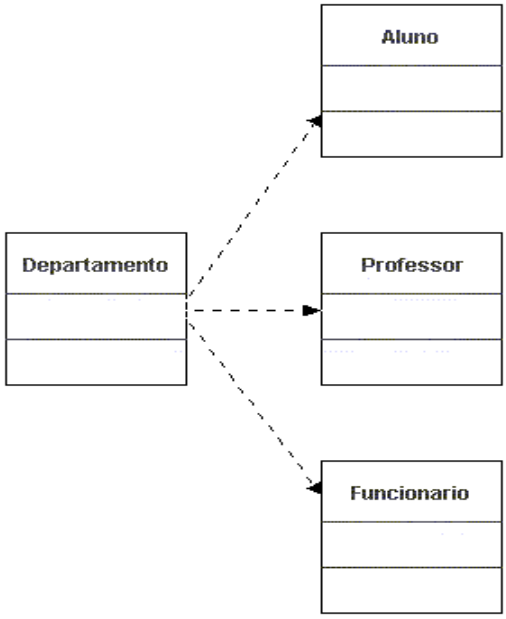

Figura 12: Dependências que originaram a coleção

Para se inferir qual o tipo de relacionamento concebido no que projeto deu origem à coleção citada, é necessário que o método no interior do código seja investigado. Os tipos dos objetos adicionados na coleção devem então ser analisados.

Como foi observado na seção 2.3, a ferramenta ao se deparar com um atributo que define uma coleção deste tipo deve decidir qual a melhor forma de se representar no modelo esta construção do código. Se a ferramenta representar da primeira forma, pode estar cometendo um equívoco, tendo em vista a intenção do projetista de não representar uma associação de cardinalidade N. Se optar pela segunda forma, não estará cometendo equívoco mesmo que a construção do modelo tenha configurado uma associação, pois a implementação de uma associação acarreta uma dependência. Esta segunda forma pode apresentar uma perda de semântica na recuperação do modelo quando a associação se configura, mas é coerente com todas as situações possíveis. Optamos neste trabalho pela segunda forma.

Como foi comentado no capítulo 1, o processo de Engenharia Reversa não é totalmente automático e necessita do auxílio humano. O usuário envolvido neste processo, tendo razoável conhecimento do modelo, pode inferir se a coleção, mostrada como atributo da classe **Departamento,** representa de fato uma associação de cardinalidade N, ou ainda, mais especificamente, uma agregação ou composição.

3.3 Arquitetura Proposta para a Ferramenta

A execução de uma ferramenta de Engenharia Reversa pressupõe como entrada de dados uma lista de arquivos, representando o código fonte da aplicação, sobre a qual se deseja aplicar o processo.

A ferramenta processa os arquivos de entrada, procurando encontrar trechos de código que representam, dentro da Orientação a Objetos, informações sobre a estrutura de classes e pacotes e seus relacionamentos.

Cada etapa do processo deve ser executada completamente, antes da ferramenta prosseguir para a próxima etapa. Para implementar este processamento, em que toda a massa de dados recebe um tratamento a cada etapa do processo, foram utilizados três módulos. Cada módulo manipula a informação, com um objetivo específico, que segue um fluxo seqüencial. Um módulo trata da leitura do código fonte, um segundo módulo interpreta os dados lidos e o último módulo exporta a informação trabalhada para um meio externo à ferramenta. Esta arquitetura corresponde ao estilo *Pipes and Filters* (SHAW *et al.*, 1996), onde cada módulo representa um filtro. Adotadas esta arquitetura, a massa de dados, composta inicialmente da lista de arquivos com o código fonte, segue por um caminho de execução (*pipe*), enquanto vai sofrendo seguidas transformações através dos filtros.

Os três filtros supracitados são:

- **Filtro *Parser***: processa os arquivos do código fonte a procura de informações relevantes para a construção do modelo e as coloca em memória, ainda sem a organização necessária;
- **Filtro de Inferência**: organiza os componentes do modelo em uma árvore de pacotes e detecta a existência de relacionamentos entre os classificadores e entre os pacotes;
- **Filtro de Transferência**: percorre a estrutura de pacotes do modelo instanciando pacotes, classificadores e relacionamentos nos meios externos. Os meios externos considerados neste trabalho são o Odyssey e uma base de fatos em Prolog.

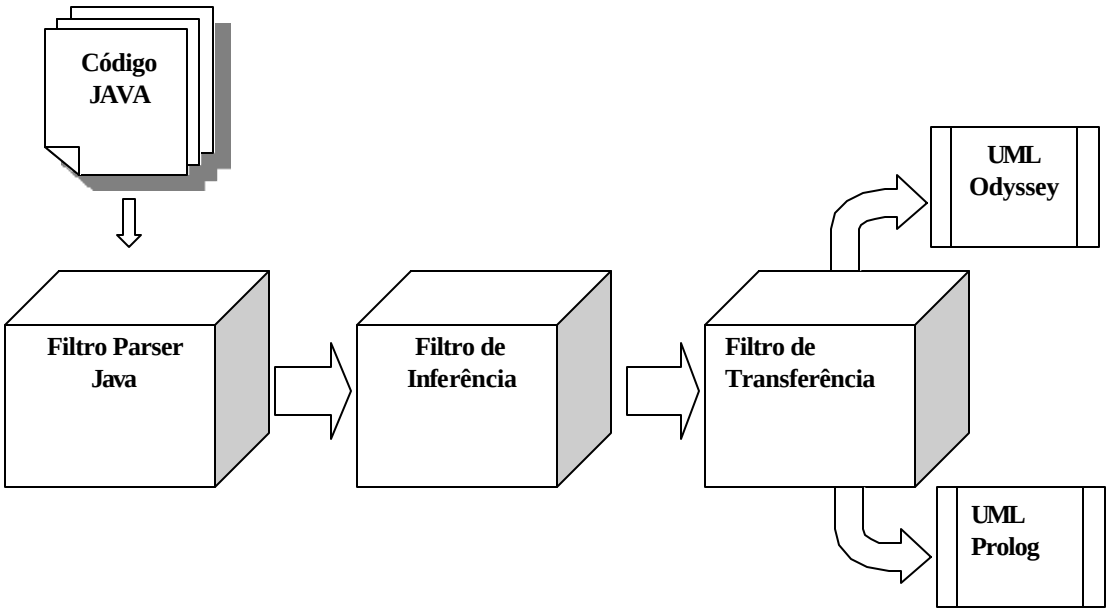

Figura 13: Visão geral da arquitetura da ferramenta.

A Figura 13 apresenta uma visão geral da arquitetura da ferramenta. Pode-se notar o fluxo das informações ao longo da Engenharia Reversa. O código dos arquivos selecionados pelo usuário é lido pelo Filtro Parser especializado para a linguagem Java. As informações são processadas no filtro inferência, onde é construída a estrutura hierárquica do programa e são criados os relacionamentos. O Filtro de Transferência percorre a árvore e a lista de relacionamentos que são exportados para dois meios: o diagramador do ambiente Odyssey e uma base de fatos em Prolog.

Neste capítulo, foram apresentadas soluções para os problemas encontrados na Engenharia Reversa de código Java para diagramas de classe UML, expondo formas de mapeamento de Java para UML. Ao final, apresentamos uma arquitetura de uma ferramenta para a implementação das soluções propostas. No contexto deste trabalho, esta arquitetura foi instanciada para o Ambiente Odyssey (WERNER *et al.*, 2000). O detalhamento desta instanciação e o esclarecimento das decisões de projeto são realizados no próximo capítulo.

4 Uma Ferramenta de Engenharia Reversa de Java para UML

4.1 Introdução

Neste capítulo, apresentamos a ferramenta, denominada ARES[6], que implementa a arquitetura proposta no capítulo anterior. Esta ferramenta encontra-se acoplada ao Ambiente Odyssey, aqui detalhado.

A estrutura interna da ferramenta é analisada, através de diversos modelos apresentados. Cada passo da Engenharia Reversa, no contexto do trabalho, é mostrado. Ao final do capítulo, é apresentado um exemplo, que objetiva ilustrar a atividade de recuperação de modelos estáticos. São ainda analisadas as conseqüências da troca de abstrações.

A escolha da tecnologia para o desenvolvimento da ferramenta foi guiada principalmente pelo fato da ferramenta ter sido idealizada para fazer parte do Ambiente Odyssey. Sua construção, no entanto, foi balizada pela intenção de garantir sua independência do Odyssey. A ferramenta foi programada na linguagem Java e é uma ferramenta de Engenharia Reversa para códigos nesta mesma linguagem.

Este capítulo está organizado em sete seções. A próxima apresenta o Ambiente Odyssey, o seu propósito e sua estrutura semântica, cujo conhecimento foi necessário para instanciação da ferramenta. A terceira seção detalha a estrutura interna da ferramenta, focando na divisão de pacotes utilizada. A três seções subseqüentes exploram cada módulo da arquitetura proposta. Finalmente, a última seção exemplifica o resultado apresentando um pequeno exemplo.

[6] O nome ARES é inspirado na mitologia grega, da Odisséia, de Homero, escrita no século VIII a.c. e foi escolhido por seguir a linha de nomes do ferramental que compõe o Ambiente Odyssey. Na Odisséia, Ares é filho de Zeus e Deus da guerra.

4.2 O Ambiente Odyssey

O Ambiente Odyssey, em desenvolvimento pela Equipe de Reutilização da COPPE/UFRJ, desde 1998, procura fornecer suporte ao desenvolvimento de software baseado em modelos de domínio. São suportadas pelo ambiente 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 Engenharia de Domínio prevê o desenvolvimento de artefatos comuns em diversas aplicações do domínio, conhecido como desenvolvimento *para* reutilização. A Engenharia de Aplicação prevê o desenvolvimento de aplicações, através da reutilização dos artefatos produzidos pela ED, conhecido como desenvolvimento *com* reutilização.

O ferramental disponibilizado visa cobrir as diversa fases envolvidas na ED e na EA, fornecendo suporte às fases de análise, projeto e implementação. Pode-se dizer que, a partir da realização deste trabalho, o Odyssey passa a fornecer suporte a uma parcela da fase de manutenção, tendo em vista que a engenharia reversa pode ser considerada uma atividade de auxílio à manutenção.

O suporte fornecido às fases de desenvolvimento encontra-se dividido entre as várias ferramentas, das quais podemos citar: a ferramenta de captura de conhecimento de domínios (ROSETI, 1998); documentação de componentes (MURTA, 1999); especificação e instanciação de arquiteturas específicas de domínios (XAVIER, 2001); camada de mediação e navegador inteligente (BRAGA, 2000); gerador de código (WERNER *et al.*, 2000); ferramenta de críticas de modelos UML (DANTAS, 2001); ferramenta para acompanhamento de processos (MURTA, 2000) e a ferramenta de diagramação UML. Esta última representa um foco de estudo neste capítulo. Toda esta infra-estrutura é implementada utilizando-se a linguagem JAVA.

A Figura 14 ilustra o ambiente de modelagem do Odyssey. A parte esquerda do ambiente, mostra as visões de modelos existentes: a visão de contextos, a visão de *features*, a visão de casos de uso e a visão estrutural, que é o objeto de nosso estudo. A visão estrutural pode conter classificadores, pacotes e relacionamentos, além de diagramas de classe, estado e seqüência. As visões estão organizadas em uma árvore. Os elementos

pertencentes à árvore são denominados elementos semânticos, enquanto que os elementos que aparecem na parte direita do ambiente (no painel de diagramação) são os elementos léxicos.

O editor de diagramas do Odyssey é uma ferramenta que permite a construção de modelos segundo um subconjunto da linguagem UML. 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). Os diversos modelos apresentam ligações entre si o que permite de certa forma a rastreabilidade de conceitos e funções que facilitam o entendimento do domínio e seus componentes.

Modelos de características apresentam uma estrutura hierárquica que permite a modelagem dos serviços no contexto de domínios, suas similaridades e diferenças, facilitando a identificação de características para reutilização. Maiores detalhes sobre esse modelo podem ser encontrados em MILLER (2000).

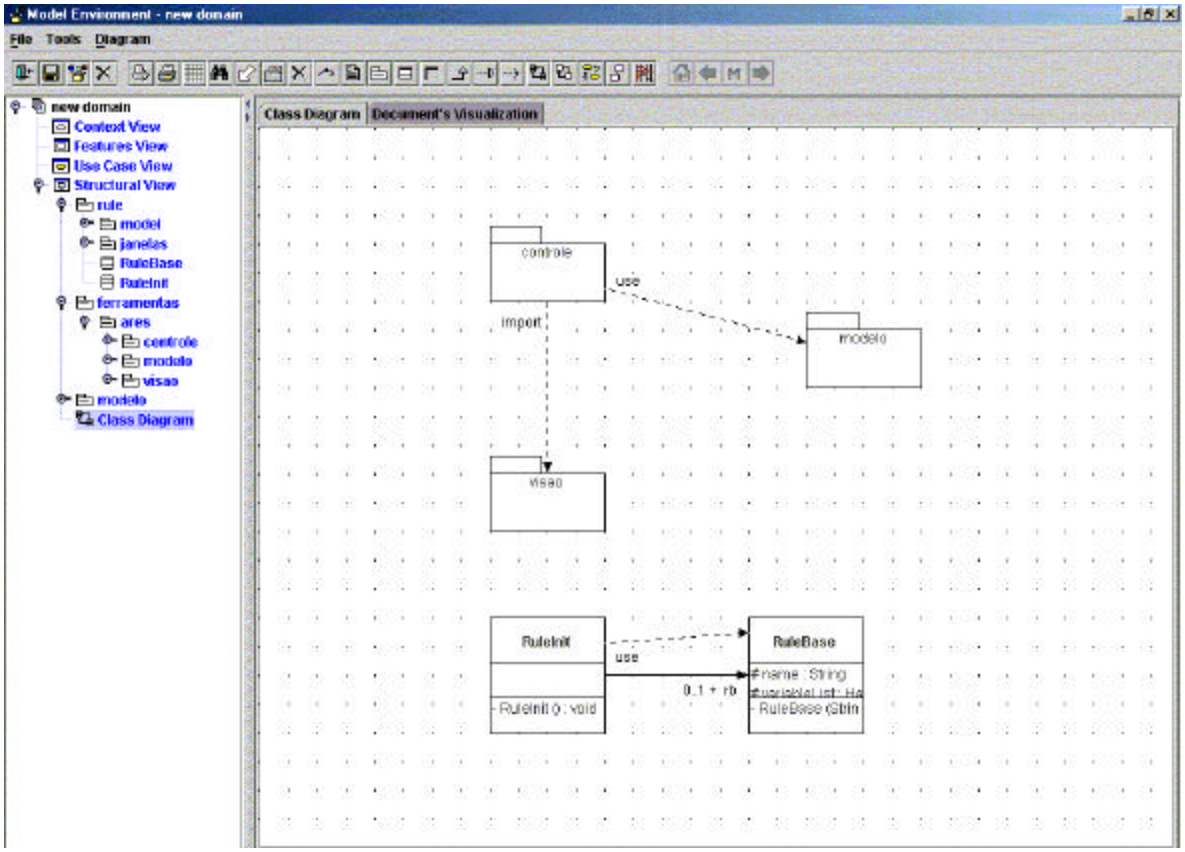

Figura 14: O ambiente de modelagem do Odyssey.

O ambiente de modelagem possui a funcionalidade de "arrastar e soltar" (*drag and drop*). Um elemento semântico, que possua representação léxica válida em um diagrama ativo na parte esquerda, pode ser arrastado da árvore para o diagrama. A ferramenta obtém, através do código, os elementos do modelo e os insere na árvore semântica do ambiente de modelagem. Uma vez tendo os elementos do modelo (que foram extraídos do código) na árvore, o usuário pode arrastá-los para um diagrama de classes. Automaticamente, os relacionamentos (léxicos) são exibidos, na medida em que os elementos participantes aparecem no diagrama.

4.2.1 Estrutura Semântica do Odyssey

No Ambiente Odyssey, as informações dos modelos são organizadas hierarquicamente através de uma árvore semântica de objetos. A Figura 15 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 que represente uma ligação, é possível descobrir quem são os objetos que são a origem e o destino da ligação.

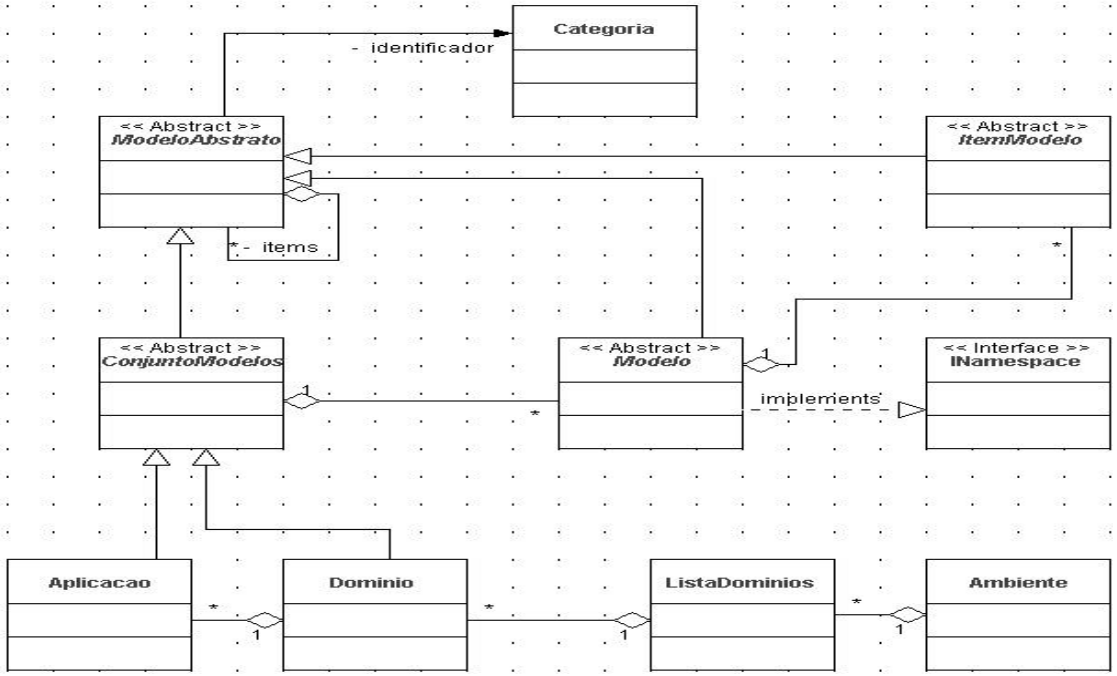

Figura 15: Modelo conceitual do núcleo do Ambiente Odyssey (DANTAS, 2001).

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.

A Figura 16 mostra um detalhamento da estrutura. 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**.

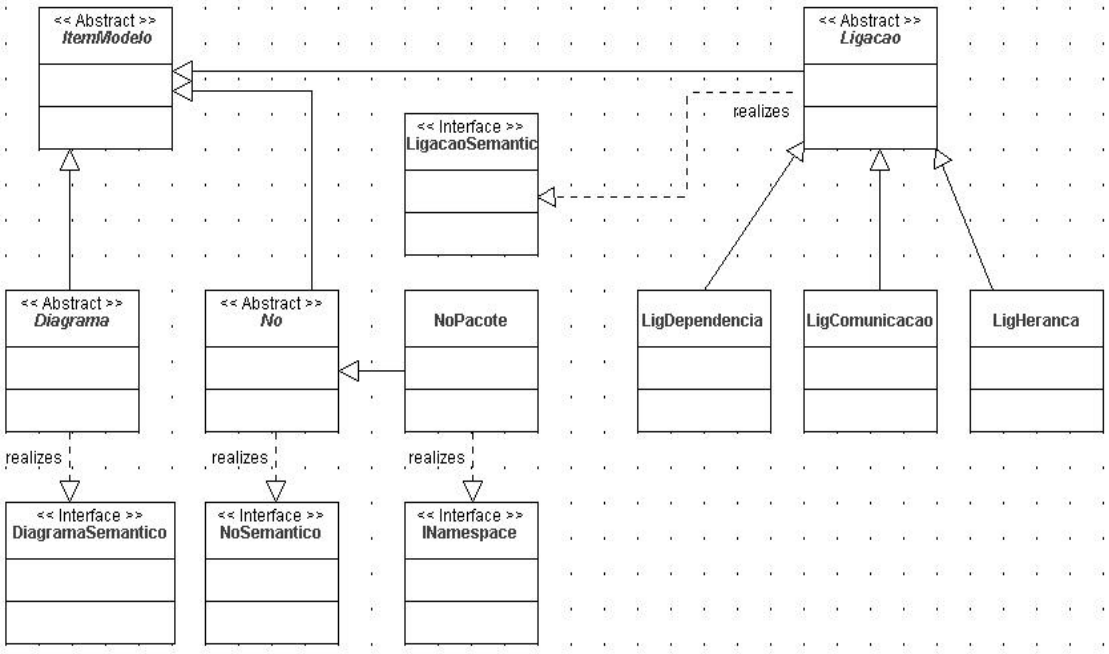

Figura 16: Detalhamento da estrutura semântica (DANTAS, 2001).

Deste modelo apresentado, as classes representantes das ligações e dos nós semânticos são fundamentais para a instanciação da ferramenta no Odyssey, pois são utilizadas para o mapeamento entre duas representações da UML: a criada pela ferramenta e a representação utilizada pelo ambiente.

Antes de detalharmos cada módulo da arquitetura, apresentamos a estrutura interna da ferramenta, do ponto de vista da organização em pacotes.

4.3 Estrutura Interna da Ferramenta

O Odyssey tem um pacote específico para as ferramentas de apoio ao ambiente. Este pacote é chamado “ferramentas”. A estrutura de pacotes da ferramenta pode ser visualizada na Figura 14. A ferramenta de engenharia reversa está no pacote **ferramentas.ares** e contém os subpacotes descritos abaixo:

- **Pacote Modelo**: contém as classes do metamodelo utilizado pela ferramenta, representa o domínio da ferramenta.

- **Pacote Visão**: controla a interface com o usuário, define a janela de seleção de arquivos e opções de execução e janela de acompanhamento do processo.
- **Pacote Controle**: contém a classe de controle central, chamada **ControleEngenhariaReversa** e todas as classes que controlam o fluxo da ferramenta. Neste pacote, encontram-se os filtros que coletam dados do código e os transforma em informações de modelo. Este pacote representa o núcleo da ferramenta e por este motivo estaremos explorando seu conteúdo com mais detalhes.

Os pacotes que compõem a ferramenta são detalhados a seguir.

4.3.1 Pacote Modelo

Este pacote contém as classes relativas ao metamodelo da UML. Além dessas classes, estão presentes neste pacote classes que realizam a função de fábrica de elementos. As fábricas de elementos são detalhadas mais adiante. A Figura 17 ilustra as principais classes do pacote **modelo**.

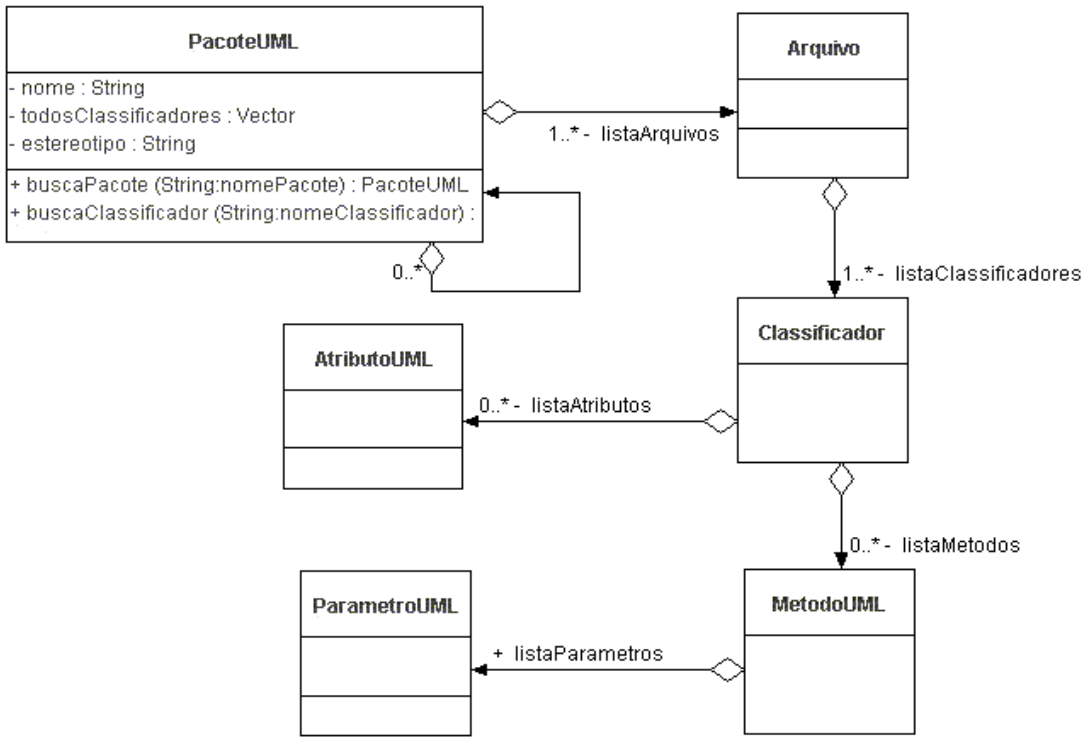

Figura 17: Implementação adaptada do metamodelo UML.

4.3.2 Pacote Visão

O pacote **visao** contém três dasses responsáveis pelas interfaces de interação com o usuário. Foi criada uma classe para representar a janela de seleção dos arquivos e opções de configuração de engenharia reversa. Outra para exibição de uma barra de acompanhamento do processo e exibição de uma tela de *log*, responsável pela captura de erros encontrados durante o processo de leitura dos arquivos. Uma terceira janela é responsável por notificar o usuário de conflitos ocorridos durante o processo de exportação de classificadores para o Odyssey.

4.3.3 Pacote Controle

Este pacote contém a base de funcionamento da ferramenta. O pacote controle é composto pela classe **ControleEnhenhariaReversa**, responsável por monitorar o fluxo de informações ao longo dos filtros, e pelo pacote **filtros**, que contém as classes que representam os filtros da arquitetura.

A Figura 18 mostra o relacionamento entre as principais classes do pacote **controle** e do pacote **filtros.**

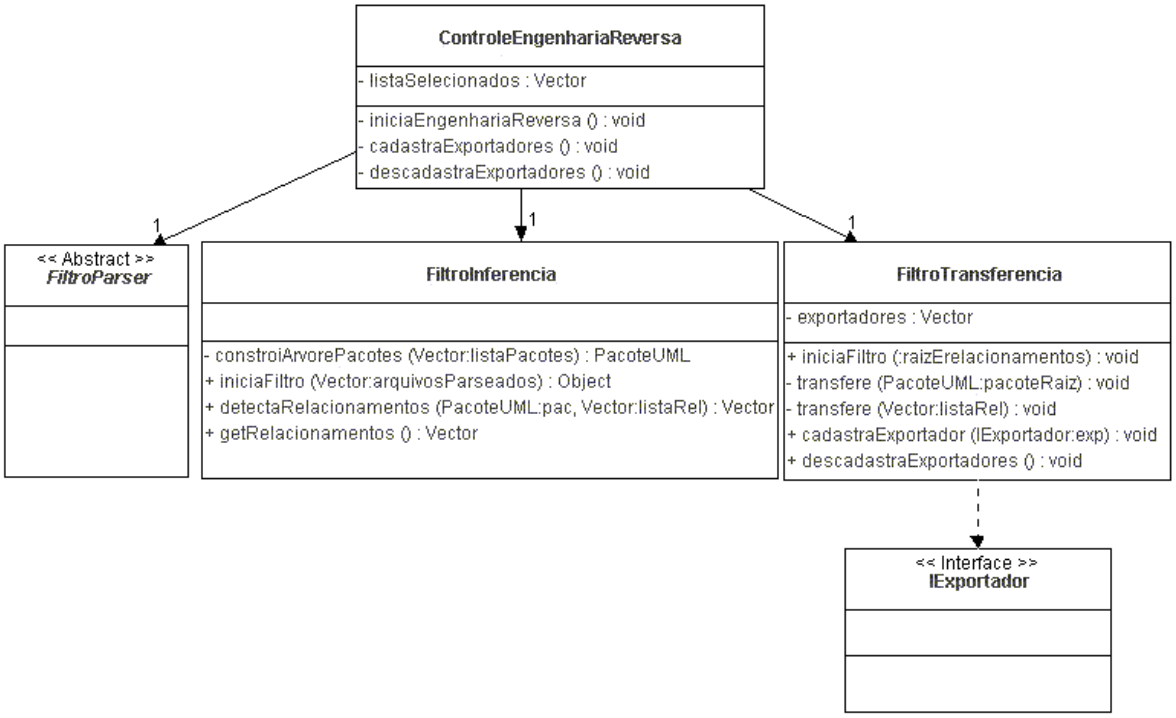

Figura 18: Diagrama de classes do controle.

O filtro *parser* interage com uma classe que gerencia o processo de leitura do arquivo. Esta classe está presente no subpacote **parser** contido no pacote **controle**. As classes do pacote **parser** são geradas automaticamente pelo gerador de *parsers* utilizado neste trabalho. Quaisquer mudanças desejadas no *parser* devem ser feitas diretamente no arquivo de definição da gramática utilizada como base, re-gerando as classes deste pacote através do gerador.

O filtro inferência, representado pela classe **FiltroInferencia,** encapsula os algoritmos que refinam a informação extraída pelo *parser*.

Como podemos observar na Figura 18, o controle está associado aos três filtros citados. Note que o filtro de transferência, representado pela classe **FiltroTransferencia**, é dependente da abstração **IExportador**, que define o protocolo a ser realizado pelas classes exportadoras. O filtro de transferência irá utilizar os serviços da interface de exportação.

Cada exportador implementa sua forma de exportar para o meio específico. Essa estrutura garante maior flexibilidade da ferramenta em comportar novos meios de

exportação. Entre os meios de exportação, podemos considerar formatos de arquivos ou outras ferramentas de manipulação de modelos UML.

4.4 Filtro Parser

Este filtro é responsável por ler os arquivos contendo o código fonte da aplicação e colocar as informações sobre a estrutura dos arquivos em uma representação interna da ferramenta, derivada do metamodelo descrito na especificação da UML.

Os arquivos são selecionados pelo usuário da forma ilustrada na Figura 19. A seleção dos arquivos pode ser feita de maneira recursiva.

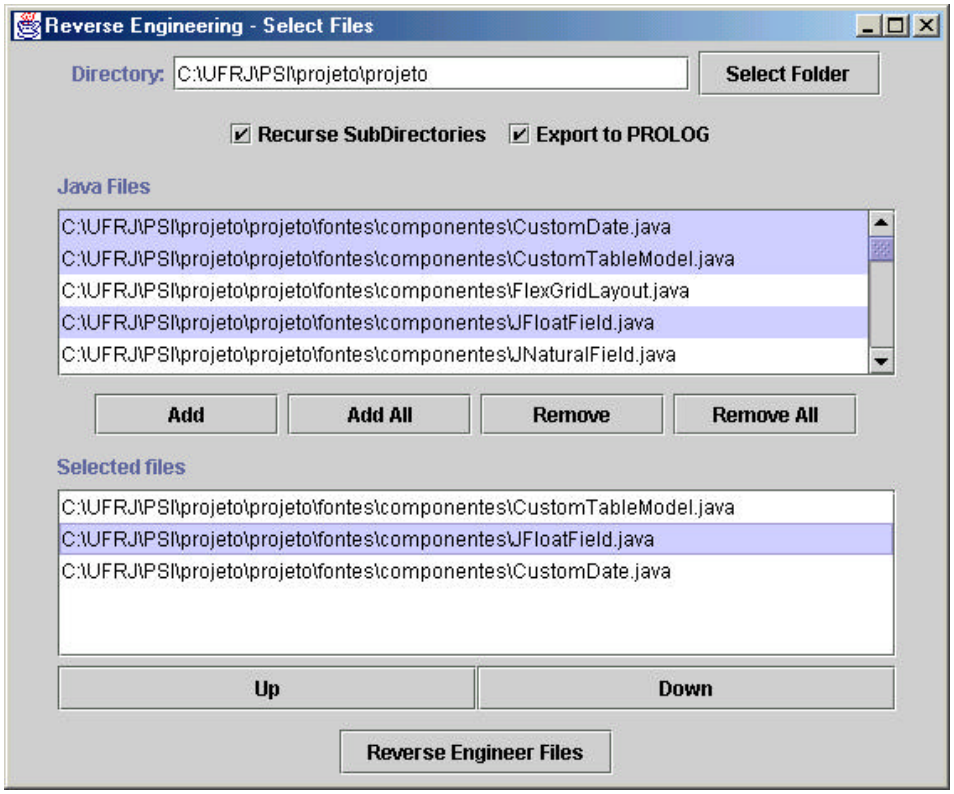

Figura 19: Seleção dos arquivos que serão lidos pela ferramenta.

A leitura do arquivo consiste da análise léxica e sintática do arquivo (AHO *et al.*, 1995). A leitura é realizada por um *parser* gerado automaticamente pela ferramenta JavaCC (METADATA, 2000), distribuída gratuitamente. O *parser* é gerado baseado na gramática da linguagem Java, versão 1.2, que é distribuída com a ferramenta. O *parser* é colocado em um pacote isolado chamado **ferramentas.ares.controle.filtros.parser**.

Para realizar a sua principal responsabilidade, o *parser* deve, além de simplesmente verificar a consistência sintática do código, encontrar trechos de código que descrevam as estruturas da linguagem importantes para extrair a estrutura da aplicação. Dentro desta filosofia, buscou-se alterar a gramática para fazer com que, ao final da execução do *parser*, a massa de dados, inicialmente composta de arquivos de texto, fosse transformada em um conjunto de pacotes e classificadores representados no metamodelo da ferramenta. Isto é, o *parser* gerado a partir da gramática inicial apenas verifica a formação sintática do arquivo. Para que as estruturas do código fossem extraídas, foi preciso inserir ações semânticas no arquivo de definição da gramática. Estas ações inseridas nas produções extraem do código construções sintáticas que são analisadas pelo filtro seguinte.

A sintaxe da gramática é definida pela ferramenta e é semelhante à sintaxe utilizada pelo difundido gerador de *parser* YACC (YACC, 2001). As alterações na gramática são feitas na forma de código intermediário em Java, descrevendo as ações semânticas, inserido nas funções que identificam produções sintáticas, ou seja, existindo uma produção na gramática para a detecção da declaração de atributos, trechos de código (ações semânticas) foram inseridos para capturar estes dados encontrados pelo *parser*. Isto foi feito para produções que detectam classes, interfaces, métodos, atributos, variáveis locais, instanciações e conversões de tipo (*casts)*, além dos métodos auxiliares aos citados.

O trecho de código exibido na Figura 20 foi retirado da gramática e é responsável pela detecção de declarações de classes. Os trechos em negrito representam os trechos de código inseridos (as ações semânticas) para fazer com que os dados encontrados fossem trazidos para o modelo interno na ferramenta.

```
String ClassDeclaration() :
{       String nomeClasse;   }
{
        ( "abstract"
        { fabricaClassificadores.setAbstract(true); }
        | "final"
        { fabricaClassificadores.setFinal(true); }
        | "public"
        { fabricaClassificadores.setNivelAcesso(IS_PUBLIC); }
        | "strictfp")*
        nomeClasse = UnmodifiedClassDeclaration()
    {
            fabricaClassificadores.setNomeClassificador(nomeClasse);
            return nomeClasse;
    }
}
```

Figura 20: Trecho de código retirado da gramática para exemplificação.

Neste exemplo, as chamadas aos métodos da instância da **FabricaClassificadores**, representam as ações semânticas inseridas na gramática, que tem o objetivo de armazenar informações relevantes do código, sendo posteriormente refinadas durante o processo de inferência.

A unidade de informação do *parser* é um arquivo. De acordo com o metamodelo adaptado, um arquivo agrega classificadores, que por sua vez agrega métodos e atributos. Os métodos agregam parâmetros.

Informações de conteúdo de um arquivo não são capturadas imediatamente. A estrutura da gramática faz com que os elementos internos sejam capturados, de maneira recursiva. O mesmo acontece com definições de classificadores e métodos, pois são elementos contendores de outros elementos. Por sua vez, informações do tipo de atributos e parâmetros podem ser capturadas diretamente pela gramática.

Devido ao fato dos elementos arquivo, classificador e método serem agregadores de outros elementos, foi necessária a criação de classes que monitorassem a criação destes elementos, o que não se dá de maneira imediata. A essas classes especiais atribuímos os nomes de **FabricaArquivos**, **FabricaClassificadores** e **FabricaMetodos**, respectivamente,

para controlar o preenchimento dos elementos arquivo, classificador e método. Estas classes estão contidas no pacote **ferramentas.ares.modelo**.

O filtro *parser* interage diretamente com a fábrica de arquivos. A principal informação manipulada pela fábrica de arquivos é uma lista de arquivos fabricados. Antes de enviar o arquivo para o *parser*, um arquivo é criado em memória. Cada chamada ao *parser* é feita pelo filtro *parser*, enviando-se como parâmetro a *stream* de um dos arquivos selecionados pelo usuário. O resultado final da execução do *parser* sobre o arquivo é uma instância da classe **Arquivo**. Este arquivo é então adicionado na lista de arquivos da fábrica. O *parser* é chamado para todos os arquivos selecionados pelo usuário. O resultado deste processo é uma lista dos arquivos fabricados.

A Figura 21 ilustra os relacionamentos das fábricas e os elementos controlados por elas.

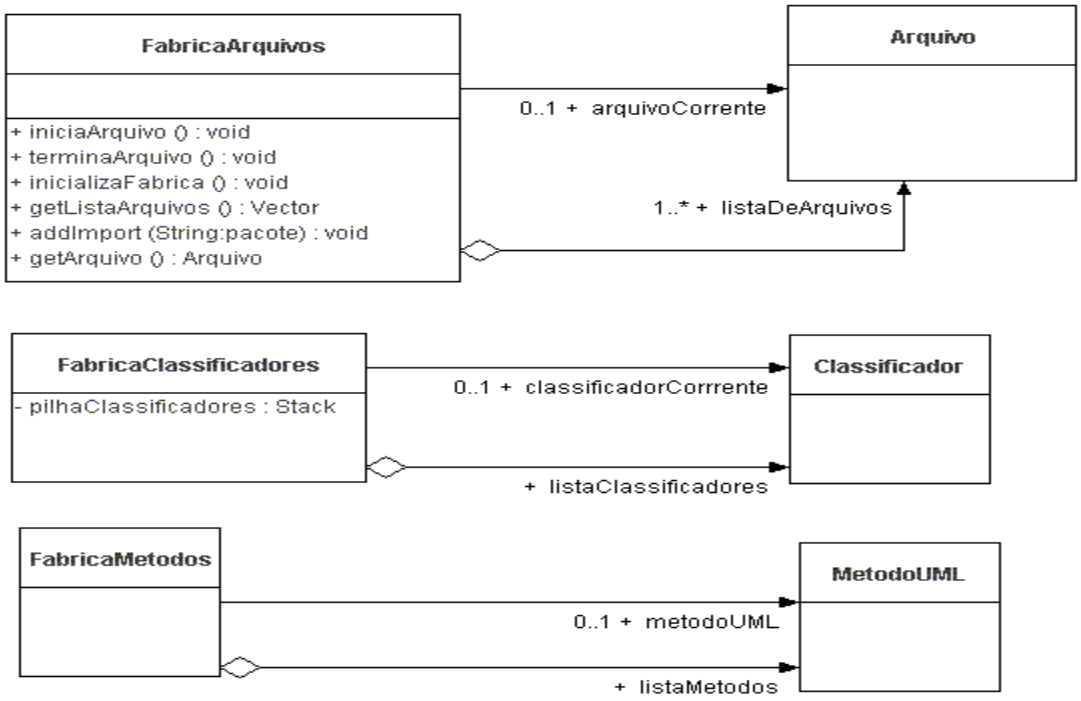

Figura 21: Fábricas utilizadas pelo filtro *parser*.

Cada fábrica apresenta uma referência para o elemento corrente sendo fabricado, em um determinado momento, durante a leitura do arquivo. Ao final do processamento de um classificador, este é adicionado na lista de classificadores da sua fábrica, e cada classificador terá sido preenchido com todos os métodos fabricados pela **FabricaMetodos**.

O mesmo acontece com a fábrica de arquivos. Ao término do processamento de um arquivo, a fábrica terá o arquivo adicionado na sua lista. Por sua vez, o arquivo terá todos os classificadores que foram processados durante sua leitura.

A forma de fabricação de métodos é semelhante. Um método é completado, após coleta de suas informações (tipo de retorno, nome, parâmetros e seus tipos). Estas informações não necessitam de uma estrutura do tipo fábrica, pois podem ser capturadas diretamente das produções da gramática.

Os classificadores em Java podem aninhar outros classificadores. Para que esses classificadores aninhados fossem capturados, foi necessário que a fábrica de classificadores contivesse uma pilha que armazenasse os contextos dos classificadores pesquisados. Assim, ao final do processamento de classificadores em níveis mais internos, estes, retirados do topo da pilha, seriam adicionados à lista de classificadores da fábrica. A cada novo aninhamento encontrado, o classificador corrente é adicionado no topo da pilha.

Um arquivo contém uma lista de classificadores processados. Cada classificador processado tem a informação de que arquivo ele pertence e, a partir do arquivo, temos a informação do pacote ao qual ele pertence, assim como os *imports* (pacotes importados) que influenciam no seu comportamento.

Conforme citado anteriormente, a saída do filtro *parser* é uma estrutura em memória de elementos do metamodelo adaptado. A estrutura criada é independente da linguagem utilizada, por ser a instância do metamodelo derivado daquele da UML. Ou seja, a ferramenta é apenas dependente da linguagem escolhida (no caso Java) nas classes que estendem o filtro, definido na classe abstrata **FiltroParser**. Cada filtro *parser* implementado deve realizar a interação com o *parser* específico para a linguagem.

A classe responsável por exercer o papel do primeiro filtro é chamada **FiltroParserJava** e é uma especialização da classe abstrata **FiltroParser**. A classe **FiltroParserJava** realiza a interação com a classe principal do *parser* gerado (**JavaParser**, pertencente ao pacote **ferramentas.ares.controle.filtros.parser**). A classe **FiltroParser** foi criada com o intuito de fornecer os serviços básicos de classes que gerenciem a interface com o *parser* de outras linguagens de programação OO, possibilitando, desta forma, a extensão da ferramenta sem comprometimento da sua estrutura interna.

O diagrama de seqüência na Figura 22 ilustra, de forma genérica, como ocorre a interação do filtro com as classes envolvidas no *parser*.

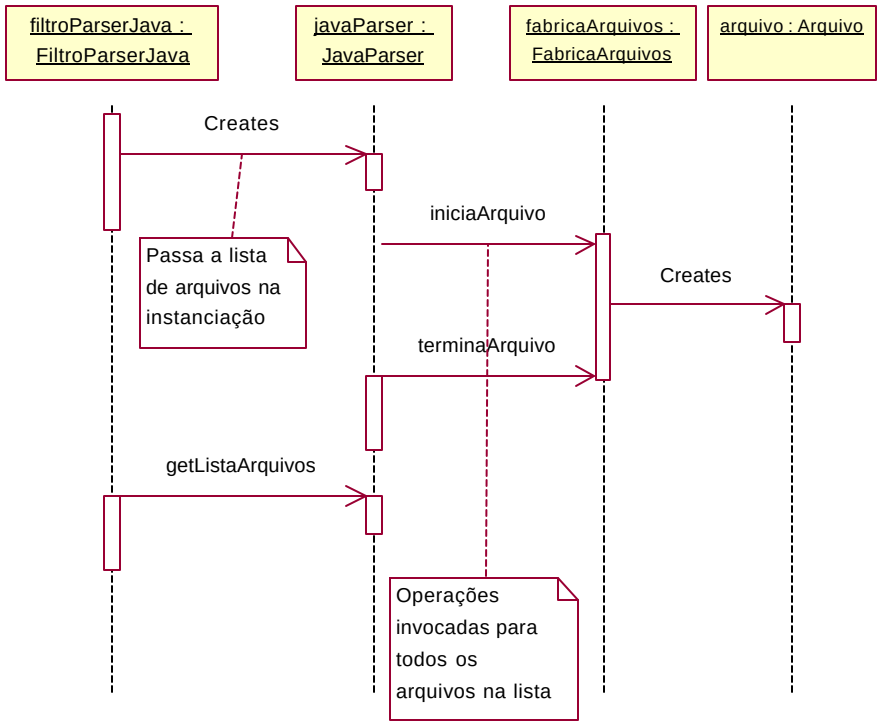

Figura 22: Diagrama de seqüência que ilustra a interação do FiltroParserJava com o *parser*.

O filtro, inicialmente, instancia a classe que gerencia o processo de leitura de arquivo, chamada **JavaParser**. Na instanciação, é passada a lista dos arquivos selecionados pelo usuário, através de uma *stream*. Estes arquivos são do tipo **File**, do pacote de classes de processamento de entrada e saída (**java.io**) que é fornecido pela linguagem Java. Para cada arquivo do sistema operacional processado, é criado um objeto do tipo **Arquivo**. Cada arquivo é "preenchido" com suas informações relevantes e, ao final da sua leitura, é adicionado em uma lista de arquivos da fábrica de arquivos (**FabricaArquivos**). Quando a lista de arquivos é completamente processada ela é retornada ao filtro *parser* que então a encaminha para o filtro subseqüente.

4.5 Filtro Inferência

O segundo filtro da arquitetura é responsável pela manipulação dos dados brutos extraídos pelo filtro *parser*. O elemento de entrada é a lista de arquivos (instâncias da classe **Arquivo**). O Filtro de Inferência agrega duas responsabilidades principais: organizar o modelo enviado pelo **FiltroParser** em uma hierarquia de pacotes e inferir os relacionamentos entre classificadores e entre pacotes. O resultado, que é uma árvore de pacotes e a lista de relacionamentos, é enviado para o filtro seguinte, o **FiltroTransferência**, cuja responsabilidade é exportar o modelo para um meio externo.

4.5.1 Árvore de Pacotes

No começo da execução, o filtro tem apenas uma lista dos arquivos disponível para iniciar o trabalho. Cada arquivo, de acordo com a definição do metamodelo proposto, contém um ou mais classificadores, mas pertence apenas a um pacote. Como citado na seção 2.2.1, um arquivo em Java deve conter uma declaração indicativa do pacote que o encapsula, exceto no caso em que o arquivo pertence a uma raiz.. A árvore é então criada baseada na lista de todos os pacotes declarados no conjunto de arquivos selecionados pelo usuário. Estes pacotes são descritos, inicialmente por *strings*, pois são capturados diretamente pelo *parser* como *strings*. O filtro inferência é responsável por manipular esse conjunto de *strings* de maneira que se obtenha ao, final do processo, uma árvore montada. Os elementos desta árvore passam a ser objetos da classe **Pacote**.

Como exemplo de funcionamento do algoritmo, suponha que três arquivos foram lidos pelo *parser* e continham, em suas declarações de pacote, os pacotes abaixo:

ferramentas.mor.tratadores

ferramentas.mor

modelo.classes

Os pacotes contidos em um mesmo nível na declaração e que contenham o mesmo nome são unificados. Se possuírem nomes diferentes, ramos diferentes na árvore são abertos. Portanto, o primeiro nível da estrutura sugerida teria dois filhos: **ferramentas** e **modelo**. O pacote ferramentas teria como filho o pacote **mor** e como neto o pacote

tratadores. O pacote modelo apresenta como filho o pacote **classes**. A Figura 23 ilustra a estrutura criada.

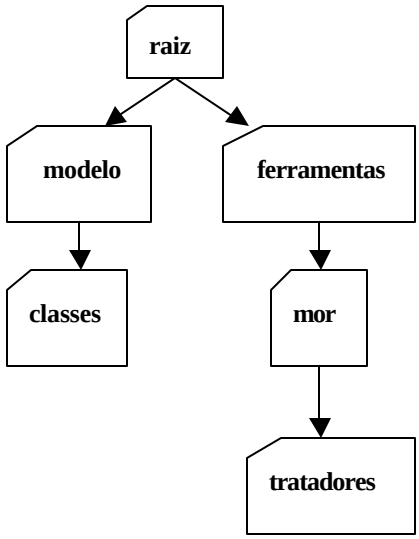

Figura 23: Hierarquia de pacotes

A ferramenta permite que seja feita engenharia reversa de arquivos dispersos, porém, sempre haverá um pacote raiz, de onde partirão os pacotes e classificadores órfãos, ou seja, mesmo que os arquivos processados pertençam a diretórios completamente distintos, na raiz da árvore haverá um pacote 'virtual' que será o ascendente de todos os pacotes processados.

A árvore de pacotes é uma árvore n-ária, onde cada pacote pode ser pai de n pacotes e um pacote tem sempre apenas um pai, exceto a raiz, que é órfã.

4.5.2 Detecção dos relacionamentos

A detecção de relacionamentos é a segunda função a ser realizada pelo filtro de inferência. Nesta fase, estruturas que representam os relacionamentos devem ser criadas em memória.

Um relacionamento é representado pela classe abstrata **LigacaoGenerica**. Esta classe define apenas qual a origem e o destino da ligação. A referência para os participantes

da ligação se dá através de uma string que descreve o nome completo do participante (que inclui o nome do caminho até o seu pacote).

No momento da detecção dos relacionamentos, a árvore de pacotes e classificadores está montada. O próximo passo é percorrer esta árvore, visitando todos os classificadores extraídos e processá-los de modo que os relacionamentos sejam construídos.

O percurso da árvore de pacotes é feito em nível. Cada classificador visitado é considerado origem de um conjunto de possíveis relacionamentos.

O relacionamento de herança é o primeiro a ser construído por ser o mais simples. A indicação da existência da relação de herança se dá através da consulta ao atributo **nomeClassePai** do classificador visitado. Este campo contém o nome completo ou atômico[7] da classe estendida.

O segundo relacionamento a ser construído é a realização. O campo **interfaces**, atributo da classe **Classificador**, contém a lista de todas as interfaces realizadas (implementadas) pela instância desta classe. Uma instância da classe **Classificador** representa um classificador encontrado no código.

O terceiro relacionamento a ser inferido é o relacionamento de associação. Este relacionamento é derivado dos atributos que o classificador possui. Portanto, a lista de atributos do classificador visitado é percorrida. O tipo do atributo é o objeto de análise. O tipo deve ser procurado no universo de busca da ferramenta. Se o tipo buscado existe neste universo, ele deve ser então transformado no alvo de uma associação e o atributo deve desaparecer da lista de atributos do classificador origem. Os modificadores do atributo (de escopo, mutabilidade, visibilidade, etc.) são armazenados na ligação criada para que possam qualificá-la. O nome do atributo é utilizado como papel do destino da associação.

O quarto relacionamento analisado é a dependência e deve ser pesquisado após os outros relacionamentos, pois é dependente da construção dos anteriores. As dependências entre classificadores detectáveis a partir do código são causadas pelas situações a seguir:

[7] Consideramos um campo atômico aquele que não contém ponto(s) (".") na *string* que o descreve, ou seja, o caminho indicador do nome possui tamanho 1.

- **Parâmetro**: Se um classificador A tem um método que recebe um parâmetro do tipo B, então existe uma dependência de A para B.
- **Tipo de Retorno**: Se um classificador A tem um método que retorna um objeto do tipo B, então existe uma dependência de A para B.
- **Variável Local:** Se no código de um método do dassificador A é declarada uma variável local do tipo B, então existe uma dependência de A para B.
- **Instanciação**: Se no código de um método do classificador A existe uma operação de instanciação de um objeto para o tipo B, então existe a dependência de A para B.
- **Conversão forçada de tipo (*cast*):** Se no código de um método do classificador A existe uma conversão de tipo de um objeto para o tipo B, então existe uma dependência de A para B.
- **Importação**: Se um arquivo que contém um classificador A apresenta uma declaração de importação de um pacote P e A usa um B contido em P, então existe uma relação de dependência do classificador A para o classificador B.

As dependências causadas pela importação de elementos são instanciadas com o estereótipo <<*import*>>, que é recomendado pela especificação da UML. As dependências restantes, que são detectáveis, são instanciadas com o estereótipo de <<*use*>>. A exibição do estereótipo não é o mais importante, mas sim a exibição da dependência. O estereótipo pode variar de ferramenta para ferramenta, tendo em vista que é um mecanismo de extensão da semântica da UML. Na especificação, são sugeridos conjuntos de estereótipos que descrevam as causas da dependência, porém é dada a liberdade para a criação de outros.

Além dos casos de dependências citados, o acesso a atributos e chamadas a métodos também configuram dependências. O exemplo abaixo ilustra a chamada em cadeia de métodos:

```
ControleConta.getConta(id).getStatus()
```

Considere que o método `getConta()` retorna um objeto do tipo **Conta** e que `getStatus()` retorna um objeto do tipo **Status**. Supondo que esta chamada é feita em um método do classificador **GerenteSistema**, é configurada, então, uma dependência de **GerenteSistema** para **ControleConta**, de **GerenteSistema** para **Conta** e de **GerenteSistema** para **Status**.

Este tipo de dependência não é detectado pela ferramenta proposta. A principal dificuldade para sua detecção encontra-se na captura pelo *parser* das informações que constituem chamadas de métodos e acessos a objetos. No *parser* gerado, expressões que representam as chamadas e acessos são interpretadas pelas mesmas produções que capturam expressões mais genéricas da linguagem Java, como por exemplo, comparações lógicas ou expressões condicionais. A separação destas expressões exige esforço bem maior que a extração de outras informações.

A estratégia para se detectar tais dependências é, em um primeiro momento, isolar expressões de chamadas e acessos a objetos de expressões não importantes para análise dos relacionamentos.

As chamadas e acessos seriam então armazenados como *strings* em uma lista que cada método poderia conter. Este seria o papel do *parser*. O filtro inferência atuaria nesta lista durante a detecção das dependências.

O último relacionamento detectado pelo filtro de inferência é a dependência entre pacotes. Para a construção desta ligação, a lista de relacionamentos é percorrida. Quando um relacionamento contém origem e destino pertencentes a pacotes diferentes, uma dependência entre os pacotes contendores deve ser instanciada.

O processo de busca é aplicado na detecção de todos os relacionamentos e é fundamental para o processo de inferência. O espaço de busca considerado é a árvore de pacotes extraída a partir do código e a árvore de pacotes e classificadores do ambiente Odyssey. Portanto, um destino de uma ligação é procurado nestas duas árvores. Este procedimento visa manter a consistência entre os dois modelos, possibilitando ao usuário a execução da engenharia reversa de maneira incremental.

Suponha, por exemplo, a busca por um classificador pai em uma herança. A *string* que descreve a referência para o classificador pai pode ser completa, se indica o caminho dos pacotes, ou atômica, quando a classe pai está no mesmo pacote, em um pacote importado, ou ainda, quando a classe é importada diretamente. Por exemplo, se o código abaixo é encontrado na declaração de um classificador **JCadastro**, representando uma janela de cadastro no sistema, que herda de uma janela simples:

```
class JCadastro extends componentes.janelas.JSimples
```

Esta declaração pode ser expressa da forma:

```
import componentes.janelas.JSimples;
class JCadastro extends JSimples
```

ou ainda,

```
import componentes.janelas.*;
class JCadastro extends JSimples
```

As três situações acima são consideradas durante o processo de busca pelo classificador B. Sendo que, para cada situação, as duas árvores são analisadas, a árvore extraída e árvore de um modelo pré-existente no ambiente. O pseudocódigo abaixo ilustra o processo de busca de um classificador destino em um relacionamento:

```
1. Buscar destino no pacote da origem na árvore de pacotes
   extraída.
   1.1.  se não achou, buscar no mesmo pacote na árvore do Odyssey.
2. se não achou, verificar lista de pacotes importados
   2.1.  para cada pacote, realizar a busca do classificador
         destino nas duas árvores.
3. se não achou, verificar lista de classificadores importados
   3.1.  se classificador procurado se encontra na lista então
         buscar o classificador nas duas árvores.
4. se não achou então o relacionamento não existe, caso contrário o
   relacionamento pode ser instanciado.
```

O algoritmo de busca se aplica a todos os relacionamentos procurados. Vale destacar que buscas sem necessidade podem ser feitas quando considerarmos tipos primitivos da linguagem durante a inferência de associações. Para se evitar tal situação, uma lista de exclusão é consultada. Se o tipo a ser procurado se encontra na lista de exclusão, sua busca é descartada.

A busca por um elemento na árvore do Odyssey causa uma dependência do filtro de inferência para a interface **IExportador**, que será explicada na seção 4.6.1. Tal dependência mostra a necessidade de que determinadas informações de controle do processo caminhem no sentido do meio externo para a o filtro de inferência, o que indica que a implementação da estrutura de filtros não se dá da maneira mais pura possível.

4.6 Filtro de Transferência

A função deste filtro é exportar, para o meio externo à ferramenta, a estrutura de elementos e relacionamentos extraída a partir do código. Entre os meios externos, pode-se considerar ferramentas de modelagem, arquivos ou outras formas de exportação.

O filtro de transferência recebe duas estruturas: uma referência para a raiz da árvore de pacotes e uma lista dos relacionamentos extraídos. O funcionamento básico do filtro é simples: a árvore de pacotes é percorrida em nível e cada nó (um pacote) é exportado. Posteriormente, a lista de relacionamentos também é percorrida, e cada entrada na lista (uma ligação) é exportada. A exportação para os dois tipos de dado (pacote e ligação) é feita da mesma forma. Invoca-se uma operação chamada **exporta(Elemento)**, definida na interface **IExportador**.

O Filtro de transferência tem acesso à lista de exportadores cadastrados na classe de controle (**ControleEngenhariaReversa**). Cada exportador tem a sua forma de exportar para um determinado meio através da implementação dos métodos **exporta(Elemento)** definidos na interface **IExportador**, que devem implementar. Percebe-se a dependência do filtro transferência para a abstração **IExportador** e não para as implementações específicas para cada exportador. Esta dependência da interface torna o modelo flexível para acomodar novas formas de exportação. Para se construir um novo exportador, basta criar uma classe

exportadora que implemente a interface descrita e cadastrá-la na lista de exportadores habilitados.

O objetivo das classes exportadoras é prover o mapeamento entre o modelo em memória extraído pela ferramenta e o modelo do meio a ser exportado.

Neste trabalho, implementamos duas formas de exportação do modelo. Uma faz a exportação para o ambiente de modelagem do Odyssey. A outra exporta o modelo UML para uma base de fatos em Prolog. A Figura 24 ilustra o modelo conceitual simplificado do filtro de exportação.

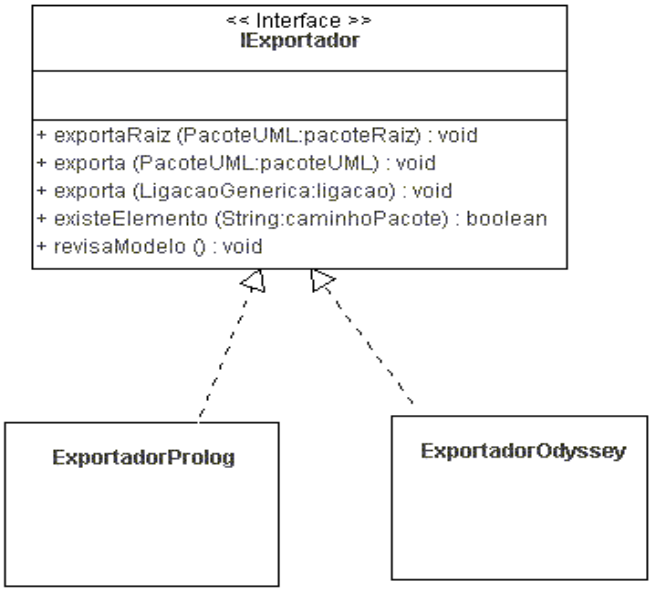

Figura 24: Estrutura dos exportadores.

Os serviços providos pela interface **IExportador** encontram-se no Anexo I., onde são descritos em detalhe.

4.6.1 Classe ExportadorOdyssey

Esta classe implementa a interface **IExportador** e é responsável por exportar o modelo extraído do código para o ambiente de modelagem da infra-estrutura Odyssey.

Como visto na seção 4.2, o Odyssey representa sua estrutura de modelos em uma classe chamada **ModeloAbstrato**, que define um protocolo que deve ser seguido pelos

modelos, **itens** e **conjuntos de modelos** do ambiente. Os modelos representam no Odyssey o mesmo conceito de modelos na UML (por exemplo, modelo de casos de uso, modelo de classes etc.). Itens são elementos que pertencem aos modelos (por exemplo, ligações, estados, objetos, classes, use cases etc.). Conjuntos de modelos agrupam os modelos. No Odyssey, existem dois conjuntos de modelos, a **aplicação** e o **domínio**. A aplicação agrupa os modelos que são criados para a construção de aplicações (desenvolvimento *com* reutilização). O domínio agrupa os modelos criados para o domínio (desenvolvimento *para* reutilização) e que serão consumidos durante a criação das aplicações.

O Odyssey possui uma estrutura hierárquica para organização dos modelos abstratos. O primeiro nível é representado por um conjunto de modelos (domínio ou aplicação). O segundo nível é representado pelos modelos (classes, casos de uso, *features* e contexto). Os níveis restantes são preenchidos por itens de modelos (classes, casos de uso, objetos, *features*, contextos, objetos, ligações etc.).

A árvore de pacotes e as ligações extraídas do código pela ferramenta devem ser mapeadas na árvore do Odyssey. No contexto apresentado, o modelo abstrato que nos interessa é aquele que representa o modelo de classes. Este modelo é tratado como um atributo da classe **ExportadorOdyssey**.

A invocação do método destacado

```
Ambiente.getInstancia().getAmbienteModelagem().getModeloCorrente()
```

retorna o conjunto de modelos sendo trabalhado, ou seja, o domínio ou a aplicação. Portanto, esta chamada retorna um Modelo Abstrato, especificamente, um Conjunto de Modelos. A partir desta referência, podemos recuperar o Modelo Abstrato responsável que representa o modelo de classes através da chamada a seguir:

```
pegaModeloCategoria(Categoria.idClasse)
```

Uma vez com esta referência, os elementos extraídos do código deverão ser inseridos nos seus níveis internos. Tratamos este modelo abstrato como “raiz do modelo de classes”.

O método de exportação de pacotes, inicialmente, faz uma chamada ao método de criação de um pacote através da invocação:

```
criaPacoteOdyssey(pacoteUML, pai)
```

Esta chamada cria o modelo abstrato do Odyssey. O parâmetro "pai" representa o modelo abstrato pai do elemento e é necessário para que o elemento seja fabricado de maneira correta pela fábrica de elementos semânticos do Odyssey. Em seguida, o método de exportação de pacotes percorre a lista de classificadores do pacote, e invoca, para cada classificador, o método de exportação de classificadores. A última tarefa a ser feita pelo método de exportação de pacotes é a inserção do pacote fabricado no nível dos filhos do modelo abstrato, que representa o seu pacote pai. Para isto, é necessário ter a referência para o modelo abstrato pai através da chamada a seguir:

```
raizModeloClasses.buscaDescendente(nomePacotePai)
```

A inserção se dá pela chamada "`pai.insere(pacoteOdyssey)`", onde "pai " é o modelo abstrato recuperado na árvore de itens semânticos do Odyssey. Note que "pai" pode ser um pacote, ou mesmo, a raiz do modelo de classes, caso o pacote exportado seja do primeiro nível na árvore (não contenha pacote pai).

O método de exportação do pacote raiz se difere do método anterior apenas pelo fato de não fabricar o modelo abstrato do pacote, pois este não existe. Apenas os classificadores são exportados e inseridos diretamente na raiz do modelo de classes (que representa o elemento que contém estes classificadores).

O método de exportação de classificadores tem visibilidade restrita na classe (**private**), já que é utilizado apenas no escopo da classe exportadora. Um método de fabricação do modelo abstrato correspondente é invocado. O modelo abstrato fabricado no Odyssey é então preenchido com os dados do classificador extraído do código (métodos, atributos, visibilidade, escopo etc.). Após a fabricação do classificador correspondente no Odyssey, é verificado se o classificador extraído já existe na árvore de elementos semânticos do ambiente. Essa verificação se dá através da chamada do método destacado:

buscaDescendente(nomeCompletoDoClassificador)

a partir da raiz do modelo de classes. Este método retorna um nulo, se o elemento não existe, ou o próprio modelo abstrato do classificador, caso ele já exista no modelo. No primeiro caso, o classificador é inserido na árvore no seu pacote pai. No último caso, o usuário é notificado com as diferenças (atributos e métodos que são exclusivos do código e exclusivos do modelo existente) entre os dois classificadores considerados conflitantes, e é dada a opção de atualização ou não do modelo (Figura 25).

Figura 25: Notificação de classificadores conflitantes.

A verificação de classificadores conflitantes entre o modelo e o código permite que o usuário tenha controle sobre o processo de engenharia reversa, impedindo a eventual perda de informações.

Após a exportação de pacotes e seus classificadores, o filtro de transferência invoca a operação de exportação de ligações. Para o método correspondente, é passado como parâmetro uma ligação abstrata. De acordo com o tipo de ligação (herança, realização, dependência ou associação), um método específico de fabricação da ligação é chamado e a ligação fabricada é então preenchida com seus dados (origem e destino para todas as ligações, navegabilidade, papéis, cardinalidade para associações e estereótipos e navegabilidade para dependências).

Uma decisão tomada a ser relatada diz respeito ao *namespace* de inserção das ligações. De acordo com uma detalhada pesquisa na especificação da UML, verificou-se que o lugar mais apropriado para inserção dos relacionamentos seria no *namespace* que contivesse todos os outros, neste caso a raiz do modelo de classes.

A última tarefa a ser realizada pelo exportador é a revisão dos modelos. Por revisão de modelo, chamamos a transformação de alguns atributos do modelo em associações. Um atributo vira uma associação caso o seu tipo (representado por um classificador), antes não existente no modelo, passa a fazer parte do modelo através da engenharia reversa. A revisão visa manter a consistência entre os modelos unificados (modelo do código + modelo antigo). A revisão se dá através da invocação do método `revisaModelo()`, definido na interface e implementado no exportador.

O método de revisão percorre os classificadores no modelo já unificado. Ou seja, no momento da chamada deste método, todos os elementos do código já foram exportados para o modelo do usuário. Para cada classificador visitado, sua lista de atributos é percorrida. Para cada atributo visitado, é verificado se existe algum classificador, no *namespace* do classificador pesquisado, que seja do tipo definido no atributo. Se for, o atributo é removido e uma ligação de associação entre os classificadores é criada. A busca é feita apenas no *namespace* do classificador pesquisado por uma limitação de implementação do Odyssey. Os classificadores no Odyssey não armazenam informação de *imports* , o que impede a pesquisa em outros *namespaces*.

4.6.2. Classe ExportadorProlog

A outra forma alternativa de exportação escolhida para a ferramenta foi uma base de fatos em Prolog, que representasse o modelo em UML. A implementação deste exportador foi feita com o objetivo de verificar a flexibilidade da ferramenta em adequar novas formas de exportação. Outras formas poderiam ser escolhidas. A criação de um exportador para o formato XMI foi avaliada e seria de grande utilidade para o Odyssey, por permitir o intercâmbio com outros ambientes/ferramentas de modelagem que utilizam UML, porém sua implementação exigiria um estudo detalhado sobre o formato de um documento em XMI, o que estava fora do escopo do trabalho. Uma base em Prolog apresenta uma sintaxe bastante simplificada, além de ser útil para futuros trabalhos. Foi possível, desta forma, validar a flexibilidade da arquitetura proposta.

A disponibilidade de um modelo UML em uma base em Prolog possibilita a realização de inferências sobre a estrutura do subsistema modelado. A exportação para uma base de fatos é um primeiro passo para instanciação, no ambiente Odyssey, da abordagem de detecção de construções de projeto boas (*Patterns*) e ruins (*Anti-Patterns*) proposta em (CORREA, 1999). Os predicados utilizados para o mapeamento do modelo extraído são definidos no trabalho citado.

O exportador em Prolog implementa a interface **IExportador** e é definido na classe **ExportadorProlog**. A diferença dos métodos de exportação deste exportador em relação ao exportador do Odyssey está na forma como os elementos são fabricados. No Odyssey, os elementos fabricados são os modelos abstratos e neste exportador são cadeias de caracteres. Portanto, um pacote neste exportador deve ser fabricado em um formato *string* do tipo "pacote(NomePacote, Estereótipo, NomePacotePai)", como definido no trabalho citado.

O exportador tem um método responsável por gravar os fatos em um arquivo texto definido pelo usuário. Este método é invocado ao final do processo de engenharia reversa, caso o usuário tenha optado por esta forma de exportação. Neste exportador, não foi implementado o método `revisaModelo()`, pois foi considerado o pressuposto que a base de fatos é vazia, sem modelo previamente construído. A revisão não foi considerada em Prolog, pois o foco deste trabalho é o Ambiente Odyssey. Este recurso, porém, pode ser adicionado facilmente, caso se faça necessário para futuros trabalhos.

Ao final do processo de exportação, é sugerida ao usuário a ativação da ferramenta de críticas (DANTAS, 2001) para a geração de um relatório sobre o modelo, extraído e unificado com um modelo pré-existente (Figura 26).

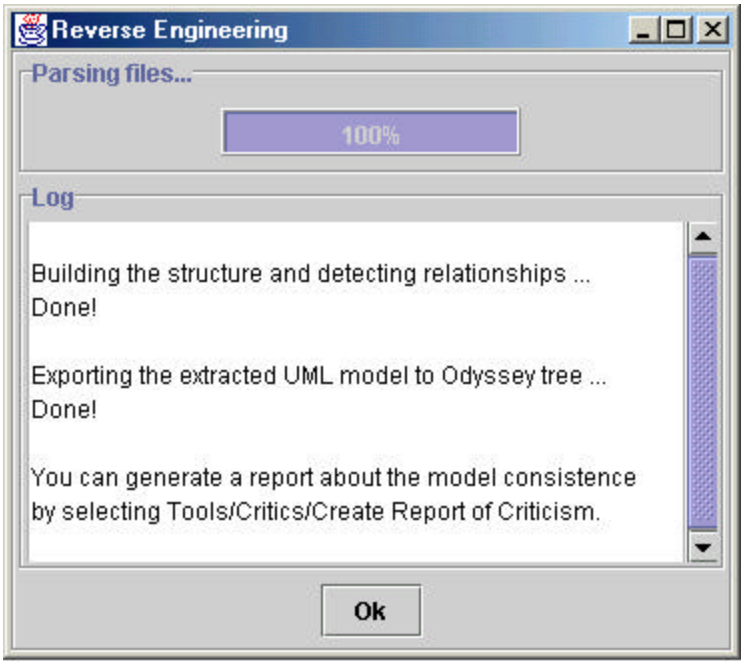

Figura 26: Fim do processo de engenharia reversa e sugestão para a geração do relatório de críticas.

4.7 Um Pequeno Exemplo

Apresentamos aqui um pequeno exemplo de uso da ferramenta. O exemplo aqui apresentado é baseado em no sistema de controle de contas, clientes e movimentações, introduzido no capítulo 3. Buscamos, através do exemplo, confrontar o modelo extraído do código com o modelo original, concebido durante o projeto do sistema.

A Figura 27 ilustra o modelo extraído a partir do código fonte de um conjunto de classes selecionadas do programa.

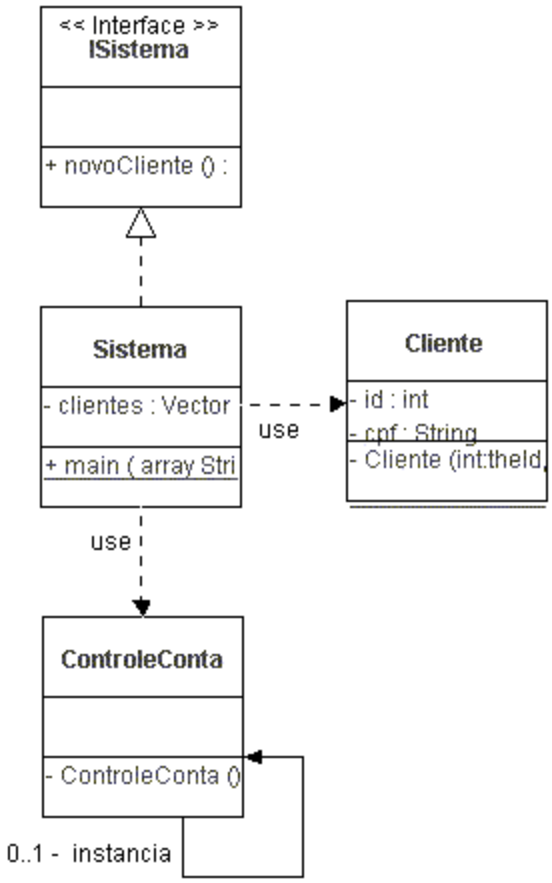

Figura 27: Fragmento de modelo extraído do código.

A classe **Sistema** é uma classe de controle, que contém como atributo, uma coleção não tipada de objetos da classe **Cliente**. Pode ser observada a presença do atributo clientes, do tipo **Vector**. Note que a intenção de projeto era de representar uma agregação de objetos do tipo **Cliente**, porém a ferramenta não tem dados suficientes para inferir uma agregação, o que acarreta a perda de semântica. A ferramenta extraiu a dependência de uso causada pela adição de elementos na coleção. Esta dependência auxilia, de certa forma, a recuperação do conceito original de projeto.

A classe **Sistema** realiza a interface **ISistema**. A recuperação deste relacionamento não acarretou a perda da semântica, pois foi mapeado diretamente a partir da palavra reservada *implements.*

A dependência da classe **Sistema** para a classe **ControleConta** é acarretada pela declaração de uma variável local em um de seus métodos e não causou a perda na tradução

dos conceitos. Note o auto-relacionamento da classe **ControleConta**. Esta classe é um *Singleton*, isto é, um padrão de projeto que representa uma classe que tem apenas uma instância própria. A definição deste padrão em uma fase de projeto mais detalhado se perde, parcialmente, quando a ferramenta realiza a Engenharia Reversa. Uma melhor forma de representação deste padrão poderia ser feita por um estereótipo <<*Singleton*>>. A Figura 28 ilustra o modelo original, concebido na fase de projeto.

O trecho de código da classe **Sistema** que implementa os relacionamentos destacados se encontra no Anexo II.

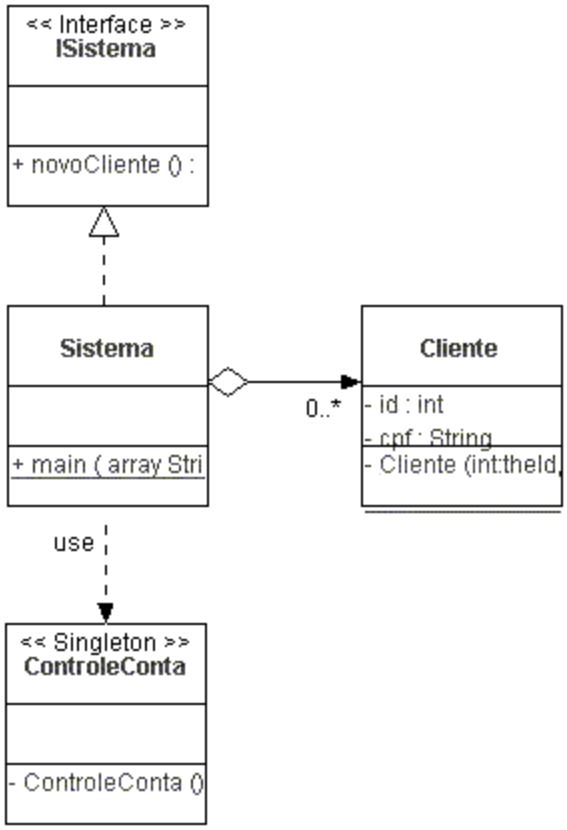

Figura 28: Fragmento do modelo original concebido na fase de Projeto.

Nota-se que, apesar da perda de semântica durante a variação das abstrações, o processo pode fornecer um bom auxílio à recuperação do modelo.

Foi realizado um estudo de caso com o objetivo de se constatar a eficácia da ferramenta quando submetida a um grande volume de código. Realizou-se a Engenharia Reversa de dois projetos de pesquisa feitos em Java. Ambos representam ambientes de

desenvolvimento de software. O primeiro foi o próprio Ambiente Odyssey e o outro foi o ArgoUML, desenvolvido pela Universidade da Califórnia, cujo código está disponível na internet (UNIVERSITY OF CALIFORNIA, 2001).

O exemplo foi executado em um microcomputador com processador Pentium III de 800 MHz de *clock* e com 256 megabytes de memória. A ferramenta não disparou erros durante a execução dos exemplos.

O Odyssey, na atual versão, apresenta um total de 412 classes. O tempo gasto para que a engenharia reversa fosse feita foi de 3 minutos e 40 segundos. Sendo que a fase que envolve a leitura de arquivos (parser) consumiu cerca de 10 segundos desse total. O ArgoUML, na versão 0.8.1, possui 686 classes. A engenharia reversa levou cerca de 6 minutos para ser executada completamente. A leitura dos arquivos consumiu cerca de 15 segundos.

Neste capítulo apresentamos o processo de instanciação da abordagem proposta no ambiente Odyssey. Foram explicadas as decisões envolvidas no projeto da ferramenta. Procurou-se detalhar cada módulo da arquitetura e os seus relacionamentos. Ilustramos ainda, com um pequeno exemplo, como acontece a troca entre as abstrações e o que isto acarreta.

No próximo capítulo, são comentadas as principais conclusões do trabalho.

5 Conclusões

5.1 Contribuições

O entendimento de um sistema a partir do código fonte é uma tarefa complexa, principalmente se a documentação é restrita, inconsistente com o estado do código, ou até mesmo inexistente. O auxílio provido pela ferramenta proposta diminui, do ponto de vista do projetista/programador, o tempo envolvido na etapa de compreensão, que antecede a correção, evolução, adaptação e/ou a reutilização do programa estudado. Qualquer tarefa que diminua o tempo necessário para a realização de uma mudança em um artefato implica em diminuição de custos nela envolvidos.

Além das vantagens citadas, o uso da ferramenta provê suporte a um ciclo de vida que não segue o modelo seqüencial tradicional, permitindo ao usuário o retorno a fases anteriores à codificação. Fornece ainda suporte à reutilização de artefatos de código no ambiente Odyssey.

No capítulo 2, foram identificadas as características de ferramentas que se propõem a fazer a Engenharia Reversa de códigos orientados a objetos escritos em Java. As características identificadas serviram de base para a criação de uma ferramenta que auxiliasse ao máximo o processo de entendimento de um sistema a partir do código. A ferramenta desenvolvida neste trabalho possui a grande parte destas características, provendo ao usuário o auxílio desejável ao processo de migração entre abstrações de código para abstrações de modelo.

A Tabela 3 e a Tabela 4 apresentam as características identificadas no capítulo 2, desta vez, comparando a ferramenta desenvolvida neste trabalho com as ferramentas existentes no mercado e no meio acadêmico.

Constatamos que a maior parte dos requisitos foi preenchida, porém nota-se que esta não se trata de uma abordagem totalmente nova no âmbito da Engenharia Reversa. Não se pode esperar automatização completa desta tarefa, dada a falta de rastreabilidade entre conceitos

de projeto que são inerentes à informalidade (ou grau de liberdade) dos métodos e representações utilizados no desenvolvimento. O uso de representações formais em fases incipientes de projeto poderia possibilitar o aumento da automatização da Engenharia Reversa, em decorrência do aumento de rastreabilidade que é conseqüente da formalidade. Porém, a complexidade imposta por estas representações não é muito bem vinda nas fases iniciais de desenvolvimento.

		Ferramentas de Engenharia Reversa de Java para UML							
		Jvision	**SoftModeler**	**GDPro**	**Together**	**Fujaba**	**Rose**	**StructureBuilder**	**ARES**
Associação	Cardinalidade Simples	✓	✓	✓				✓	✓
	Cardinalidade N (com *Array*)	✓	✓						✓
	Navegabilidade		✓	✓	✓	✓	✓	✓	✓
	Papel		✓				✓		✓
	Mutabilidade								✓
	Visibilidade					✓	✓		✓
Dependência	Importação de Elementos								✓
	Passagem de parâmetros	✓							✓
	Retorno de objetos	✓							✓
	Instanciação de objetos	✓							✓
	Declaração de variáveis locais								✓
	Conversões de tipo (*cast*)								✓
	Chamadas/Acesso a atributos								
	Dependências entre pacotes			✓					✓
Detecção/Exibição da Estrutura de pacotes				✓	✓		✓	✓	✓
Relacionamentos entre elementos de pacotes distintos		✓	✓		✓	✓			✓
Forma de tratamento de coleções não tipadas								Associação com cardinalidade N	Dependências

Tabela 3: Tabela comparativa de capacidade de detecção de dependências das ferramentas.

Nas características relacionadas à capacidade de detecção da ferramenta, nota-se que todas as propriedades das associações são detectadas. Além disso, apenas as dependências provocadas por acesso a objetos e chamadas a métodos não são detectadas na versão atual. A ferramenta trata do uso de coleções não tipadas e extrai relacionamentos de dependências

entre pacotes. Poucas ferramentas fazem esta detecção: apenas a ferramenta Structure Builder analisa coleções não tipadas, enquanto que apenas a GDPro extrai a estrutura de pacotes com as suas dependências.

O ambiente de uso fornecido pela ferramenta atendeu todas as características identificadas (Tabela 4). A funcionalidade de visualização seletiva de relacionamentos foi implementada no ambiente de diagramação do Odyssey, para que alguns detalhes do modelo extraído possam ser ocultados, evitando a "poluição" do diagrama. Esta funcionalidade pode ser utilizada também fora do contexto da ferramenta.

	Ferramentas de Engenharia Reversa de Java para UML							
	Jvision	**SoftModeler**	**GDPro**	**Together**	**Fujaba**	**Rose**	**StructureBuilder**	**ARES**
Junção e revisão de modelos[8]	parcial	✓	parcial		✓	parcial	✓	✓
Visualização seletiva de relacionamentos	✓			✓			✓	✓
Notificação de conflitos entre modelos							parcial[9]	✓
Meio de exportação alternativo				XMI				Prolog
Seleção recursiva de código fonte	✓		✓	✓		✓	✓	✓

Tabela 4: Tabela comparativa que avalia o ambiente de uso provido ao usuário.

Vale destacar que a arquitetura projetada permite a adaptação da ferramenta a novas formas de exportação bem como a extensão a outras linguagens orientadas a objetos, sem comprometimento da sua estrutura.

5.2 Limitações

Dentre as características identificadas, a ferramenta não conseguiu atender a detecção de dependências causadas por chamadas de métodos e acesso a objetos. Para que esta detecção seja feita, é necessária uma evolução do módulo de extração de informações do

[8] Valor considerado parcial se a revisão dos relacionamentos não é feita.
[9] Não exibe diferenças entre os classificadores. Apenas notifica o usuário da existência do conflito.

parser. Pode-se verificar que, dentre as ferramentas avaliadas, nenhuma delas possui esta característica.

A ferramenta apresentada não realiza engenharia reversa sobre arquivos *byte code* produzidos pelo compilador da linguagem. *Byte codes* não se apresentam em formato texto como o código fonte. Estes arquivos possuem extensão ".class" e exibem informações de atributos, assinatura de métodos e modificadores dos classificadores. Neste trabalho, nos dedicamos a estudar os códigos fonte pois, apesar de apresentar maior complexidade e exigir maior esforço, podíamos ter acesso ao código interno dos métodos. Os arquivos *byte code* não permitem acesso ao interior do código dos métodos, o que inviabilizaria a detecção das diversas formas de dependência.

A ferramenta ARES se propõe a fazer engenharia reversa de código Java, tendo como alvo diagramas de classe em UML. A estrutura estática do programa é recuperada.. A ferramenta possibilita a recuperação e a diagramação (que é realizada manualmente pelo usuário) dos classificadores, pacotes e relacionamentos existentes em código Java, no ambiente Odyssey. Entretanto, a ferramenta não extrai a dinâmica de interação entre objetos. Seria desejável que se fosse capaz de extrair diagramas de colaboração e/ou seqüência a partir do código, porém esse é um problema altamente complexo e que está fora do escopo deste trabalho.

5.3 Trabalhos Futuros

Um ponto a ser evoluído no sistema é a resolução de detecção de dependências causadas por acesso a objetos e chamada de métodos. Este tipo de dependência não é atualmente detectado pela ferramenta.

Conforme pudemos observar, durante o processo de recuperação de modelos a partir do código fonte, inevitavelmente, corremos o risco de perda da semântica originalmente definida pelo desenvolvedor. Isto ocorre na medida em que as técnicas de Engenharia Reversa apresentam limitações quanto ao que é possível inferir automaticamente a partir do código fonte. Uma das possibilidades para a solução parcial deste problema seria a visualização e acompanhamento deste processo de decisão sobre possíveis "buracos"

semânticos, durante a recuperação, como um processo de cooperação entre diversos elementos envolvidos no desenvolvimento.

Avaliações sobre o modelo extraído do código podem auxiliar na reestruturação do sistema. Em CORREA (1999) é proposta uma técnica para detecção de construções de projeto boas (*Patterns*) e ruins (*Anti-Patterns*), e são apresentadas heurísticas de projeto encontradas na literatura. A técnica apresentada é implementada em uma ferramenta chamada OOPDTool, que funciona como um *add-in* da ferramenta Rose (RATIONAL, 2001). Esta ferramenta representa o modelo de classes (extraído pela Engenharia Reversa do Rose) em uma base de fatos e então efetua a análise desta base, através de um conjunto cláusulas em Prolog que descrevem os padrões e anti-padrões. Após a análise, relata ao usuário sugestões para reestruturação do código e do modelo. Um trabalho futuro é a implementação desta abordagem no Odyssey. Uma vez extraída a base de fatos do código, que é feito pela ferramenta ARES, poderão ser feitas inferências sobre as estruturas do sistema analisado.

Referências Bibliográficas

AHO, A., SETHI, R., et al., 1995, "Compiladores, Princípios, Técnicas e Ferramentas", *LTC*, Rio de Janeiro.

BRAGA, R. M. M., 2000, "Busca e Recuperação de Componentes em Ambientes de Reutilização de Software", tese de DSc, *COPPE/UFRJ*.

BIGGERSTAFF, T. J., MITBANDER, B. G., WEBSTER, D. E., 1994, "Program Understanding and the Concept Assignment Problem", *ACM*, pp. 72-82, volume 37 nº 5.

CHIKOFSKY E.J., WATERS C.R., 1994, "Reverse Engineering", *Commnunications of ACM*, pp. 23-24, volume 37.

CHIKOFSKY, E. J., SELFRIDGE, P. G., WATERS, C. R., 1993, "Challenges to the Field of Reverse Engineering -- A Position Paper", *Computer Society Press*, *IEEE*, pp. 144-150, Baltimore, Maryland, USA.

CORREA, A. L., 1999, "Uma Arquitetura de apoio para Análise de Modelos Orientados a Objetos", Dissertação de MSc., *COPPE - Universidade Federal do Rio de Janeiro*.

DANTAS, A. R., 2001, "Oráculo: Uma ferramenta de Críticas para UML", Projeto Final de Curso de Bacharelado em Informática- Instituto de Matemática/UFRJ (29/06/2001).

EMBARCADERO TECHNOLOGIES, 2001, "GDPro 5.1", disponível na Internet em http://www.gdpro.com.

FOWLER, M., 2001, "Refactoring", disponível na Internet em http://www.refactoring.com.

FUJABA, 2001, "Fujaba 2.5.4", disponível na Internet em http://www.uni-paderborn.de/fachbereich/AG/schaefer/ag_dt/PG/Fujaba/.

GAMMA, E., HELM, R., JOHNSON, R., et al., 1995, "Padrões de Projeto", *Bookman*.

GOGOLLA, M., KOLLMAN, R., 2000, "Re-Documentation of Java with UML Class Diagrams", *Reegineering Forum*, Burlington, Massachussets, USA.

KOLLMAN, R., GOGOLLA, M., 2001, "Capturing Dynamic Program Behaviour with UML Collaboration Diagrams", *IEEE*, Los Alamitos.

METADATA, 2000, "Java Compiler Compiler - JavaCC", disponível na Internet em http://www.metadata.com.

MILLER, N., 2000, "A Engenharia de Aplicações no Contexto da Reutilização Baseada em Modelos de Domínio", Dissertação de MSc., *COPPE/UFRJ*.

MURTA, L. G. P., 1999, "FRAMEDOC: Um Framework Para Documentação de Componentes Reutilizáveis", Projeto Final de Curso de Bacharelado em Informática- Instituto de Matemática/UFRJ.

MURTA, L. G. P., 2000, "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.

OBJECT INSIGHT, 2001, "JVision 1.4.2", disponível na Internet em http://www.object-insight.com.

OMG, 2000, "OMG Unified Modeling Language Specification", disponível na Internet em http://www.omg.org/uml.

PRESSMAN, R. S., 2001, "Software Engineering, A practitioner's Approach", 5 edição, *McGraw-Hill*.

RATIONAL, 2001, "Rose 2000", disponível na Internet em www.rational.com/rose.

ROSETI, M. Z., 1998, "Uma Proposta de Sistemática para Aquisição de Conhecimento no contexto de Análise de Domínio", Dissertação de MSc., *COPPE/UFRJ*.

SHAW, M., GARLAN, D., 1996, "Software Architecture: Perspective on an Emerging Discipline", *Prentice-Hall*.

SOFTERA, 2001, "SoftModeler 3.3", disponível na Internet em http://www.softera.com.

SUN MICROSYSTEMS, 2001, "Java 2 Plataform API Specification, 1999.", disponível na Internet em http://java.sun.com/products/jdk/1.2/docs/api/index.html.

SYSTÄ, T., 1999, "On the Relationships between Static and Dynamic Models in Reverse Engineering Java Software", Department of Computer Science, University of Tampere.

TOGETHER SOFT, 2001, "Together 5.01", disponível na Internet em http://www.togethersoft.com.

UNIVERSITY OF CALIFORNIA, 2001, "ARGO UML 0.81", disponível na Internet em http://www.argouml.org.

XAVIER, J. R., 2001, "Criação e Instanciação de Arquiteturas de Software Específicas de Domínio no contexto de uma Infra-estrutura de Reutilização", Dissertação de MSc, *COPPE/UFRJ*.

WEBGAIN, 2001, "Structure Builder 4.0", disponível na Internet em http://www.webgain.com.

WERNER, C. M. L., BRAGA, R. M. M., MATTOSO, M., et al., 2000, "Infra-estrutura Odyssey: estágio atual", *XIV Simpósio Brasileiro de Engenharia de Software, Sessão de Ferramentas*, João Pessoa, outubro 2000.

WERNER, C. M. L., CORREA, A. L., BARROS, M., et al., 2001, "Notas de aula da disciplina Projeto de Sistemas de Informação".

YACC, 2001, "YACC - Yet Another Compiler Compiler", disponível na Internet em http://www.combo.org/lex_yacc_page/.

Anexo I

Interface IExportador

Nesta interface, são definidos cinco métodos que devem ser implementados pelos exportadores:

```
public void exportaRaiz(PacoteUML pacoteRaiz);
```

Este método tem por objetivo exportar o pacote raiz da árvore de pacotes. Foi concebido para tratar um caso especial que é o do pacote raiz. Este pacote deve ser tratado em separado, pois não existe no modelo real, isto é, em um projeto em Java, arquivos que pertencem à raiz não necessitam de um pacote (diretório) encapsulador. Ele é criado durante o processo de inferência apenas para acomodar os classificadores que pertencem à raiz do projeto e não possuem pacote definido. A implementação deste método deve considerar estes classificadores e exportá-los individualmente.

```
public void exporta(PacoteUML pacoteUML);
```

Método responsável pela exportação de um pacote, ou seja, exporta um nó da árvore de pacotes. Sua implementação deve criar a abstração de pacote no meio correspondente e exportar os classificadores contidos no próprio pacote.

```
public void exporta(LigacaoGenerica ligacao);
```

Método responsável por exportar as ligações. O tipo **LigacaoGenerica**, enviado como parâmetro, corresponde à classe abstrata que define uma ligação no metamodelo implementado. De acordo com a ligação enviada, a implementação do método deve criar a abstração da ligação e exportá-la para o meio determinado.

```
public boolean existeElemento(String caminhoElemento);
```

Este método visa fornecer informações do modelo já existente no meio externo à ferramenta. Ao contrário dos métodos anteriores, que operam no sentido "ARES → meio externo", este funciona no sentido "meio externo → ARES", para que fosse possível consultar se um determinado elemento, que é passado como parâmetro, existe no modelo externo. Essa funcionalidade é necessária para que seja possível fazer o processo de engenharia reversa se realizar de maneira incremental. Através desta consulta, pode-se tomar decisões quanto à construção de relacionamentos entre elementos de modelo extraídos pela ferramenta e elementos já existentes em um modelo previamente construído pelo usuário.

```
public void revisaModelo();
```

Este método é responsável por revisar o modelo resultante da junção do modelo exportado com o modelo previamente construído no meio exportado. O objetivo deste método é reformular os atributos do modelo "antigo" e recriar relacionamentos. Esta funcionalidade foi incluída para que a consistência entre os dois modelos fosse mantida. Por exemplo: considere que um classificador C no modelo antigo contenha um atributo do tipo T. Se um determinado classificador T é extraído pela ferramenta de engenharia reversa e exportado para o meio considerado, deve-se construir uma associação de C para T e eliminar o atributo correspondente (pois este se tornou um pseudo-atributo), isto se o classificador T pertence ao *namespace* do classificador C. Esta revisão mantém os modelos consistentes.

Anexo II

Trecho de código da classe **Sistema**, utilizada como exemplo analisado pela ferramenta:

```
package controle;
import modelo.*;
import visao.*;
import componentes.*;
import persistencia.GerentePersistencia;
import excecoes.*;

public class Sistema implements ISistema
{
  /** Lista de clientes existentes na base de dados */
  private Vector clientes;
  /** Janela principal do sistema de interação com o usuário */
  private JanelaPrincipal janela;
  public static void main (String []args)
  {
    new Sistema();
  }
  public Sistema ()
  {
    Cliente[] temp = null;
    GerentePersistencia persistencia = GerentePersistencia.getInstancia();
    if (persistencia != null)
      temp = persistencia.getClientes();
    clientes = new Vector();
    if (temp != null)
      for (int i=0; i<temp.length; i++)
        clientes.addElement (temp[i]);
  }
  public void novaConta()
  {
    Cliente clienteSelecionado = (Cliente)janela.getClienteSelecionado();
    if (clienteSelecionado != null)
    {
      ControleConta cConta = ControleConta.getInstancia
     (...) //Código não relevante para explicação
   }
```

Relacionamento de realização

Coleção não tipada que implementa o conceito da agregação

A adição dos objetos na coleção acarreta uma dependência.

Declaração de variável local que acarreta uma dependência de uso

SUMÁRIO

Índice das Tabelas e Figuras