ARES: Uma Ferramenta de Engenharia Reversa Java-UML

Gustavo Veronese, Alexandre Correa, Cláudia Werner[1]
Felipe Jezini Netto [2]
{veronese, alexcorr, werner}@cos.ufrj.br , fjezini@symprise.com
[1]COPPE/UFRJ – Programa de Engenharia de Sistemas e Computação
[2]IM-DCC/UFRJ – Departamento de Ciência da Computação
Universidade Federal do Rio de Janeiro
Caixa Postal 68511 – CEP. 21945-970
Rio de Janeiro – Brasil

Resumo

Neste trabalho apresentamos uma abordagem para o auxílio à recuperação de modelos de classe da UML a partir de códigos fonte feitos na linguagem Java. São analisadas as principais dificuldades envolvidas na mudança semi-automática de abstrações de código para abstrações de projeto. Com base na abordagem proposta, implementamos uma ferramenta que visa minimizar as deficiências de outras abordagens de projeção no mercado e no meio acadêmico. A ferramenta foi construída no contexto do ambiente de desenvolvimento de software Odyssey.

Abstract

This work presents an approach to assist a designer in recovering UML class models from Java code. First, we discuss the main problems involved in recovering design abstractions from source code, and then describe a tool that supports the design recovery approach and attempts to minimize the deficiencies found in other commercial and academic approaches. The tool was built in the context of the Odyssey software development environment.

1. Introdução

Mudanças são inevitáveis durante a vida de um software. Em muitas situações, a documentação é inexistente e o código é a única fonte de informação disponível para a manutenção do sistema. Entretanto, o entendimento de um sistema a partir do código não é um processo trivial, dado que as suas estruturas são direcionadas a ferramentas do gênero dos compiladores. É necessário, então, prover suporte, de preferência automatizado, a este processo. Neste contexto se insere a Engenharia Reversa. Este termo é empregado a um grande conjunto de tarefas relacionadas ao entendimento e modificação de sistemas de software. Suas duas principais atividades 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 [1].

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 uma notação padrão, como a UML (*Unified Modeling Language*), a partir do código. A Engenharia Reversa atua no auxílio à recuperação da documentação e ao entendimento desses sistemas, possibilitando que a sua manutenção seja realizada de forma menos árdua.

Existem algumas ferramentas disponíveis no mercado e no meio acadêmico, normalmente acopladas a ambientes CASE, que se propõem a apoiar a Engenharia Reversa de código orientado a objetos. Entre elas podemos citar: JVision 4.1 [2], GDPro 5.0 [3], Structure Builder [4], Fujaba [5], Together [6] e Rational Rose [7]. Poucas recuperam de uma maneira satisfatória os detalhes importantes

do código, cuja representação no modelo seria de grande valia para documentação, entendimento, manutenção ou reestruturação do sistema [8].

Observam-se 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, é abordada 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, pode-se 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.

Este artigo está dividido em 4 seções: A próxima seção trata do mapeamento de elementos de código Java em elementos do modelo UML, que serve de base para o funcionamento da ferramenta. Na terceira seção, é introduzida a ferramenta ARES, detalhando sua arquitetura, funcionamento e estrutura interna. Finalmente, a última seção exibe as conclusões e perspectivas futuras deste trabalho.

2. O Mapeamento Java-UML

Nesta seção, são apresentados os elementos da UML que compõem o diagrama de classes e como ocorre manifestação desses elementos em código Java. Os principais elementos são os classificadores, pacotes e relacionamentos, detalhados a seguir.

2.1 Classificadores e Pacotes

De acordo com a especificação da UML 1.3 [9], um classificador pode ser uma classe, uma interface, ou um tipo primitivo de dado (*DataType*). Os tipos primitivos não apresentam identidade e costumam ser utilizados em diagramas de classe para expressar os tipos dos atributos de classes e interfaces. Para cada interface e classe definida no código em Java é gerado um classificador UML correspondente com a sua representação gráfica. Os tipos primitivos não são descartados, mas não têm a mesma representação dos outros classificadores (interface e classe) e aparecem apenas como tipos de atributos, não originam a nenhum relacionamento. Esta é a representação utilizada em todas as ferramentas de Engenharia Reversa analisadas neste trabalho.

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 casos de uso, componentes etc. Do ponto de vista estrutural, um pacote pode conter os classificadores e seus relacionamentos, diagramas de classe e outros pacotes.

2.2 Relacionamentos

Os principais relacionamentos entre classificadores na UML são: generalização, realização, associação e dependência. Os relacionamentos de generalização e realização são fáceis de serem identificados, pois são mapeados diretamente a partir das palavras reservadas *extends* e *implements*, respectivamente. Os relacionamentos de associação e dependência apresentam maior complexidade e serão analisados em maior detalhe nos subitens a seguir.

Associação

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 a origem do relacionamento. Os classificadores que definem atributos do classificador origem são potenciais

candidatos a serem destinos de associações. As associações são criadas com classificadores que contêm 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 maior parte das propriedades das associações são mapeadas diretamente a partir de estruturas sintáticas da linguagem Java, entre elas destacam-se: navegabilidade, escopo, mutabilidade, visibilidade e papel. O mapeamento das cardinalidades é um pouco mais complexo. Deve-se 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 é menos trivial do que parece à primeira vista [10]. 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.

As linguagens orientadas a objetos apresentam estruturas de dados que mantém conjuntos de referências para objetos em memória, cujos tipos são definidos apenas em tempo de execução. O objetivo destas estruturas é a manipulação de coleções de objetos. Em Java, por exemplo, temos as coleções não tipadas que implementam o *framework Collection*. Uma coleção não tipada mantém referências para objetos genéricos. A definição do tipo do objeto se dá apenas na recuperação do objeto, pela realização de uma conversão forçada de tipo (*cast*).

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

Ao se deparar com um atributo que define uma coleção deste tipo, deve-se decidir qual a melhor forma de representar no modelo esta construção do código. Se a ferramenta representar um relacionamento de cardinalidade N com os classificadores dos objetos adicionados na coleção, ela poderá estar cometendo um equívoco, tendo em vista a intenção do projetista de não representar uma associação de cardinalidade N. Se optar por representar dependências para os classificadores dos objetos adicionados, 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.

Dependência

As dependências podem ocorrer entre pacotes e classificadores. A especificação da UML propõe sete tipos de dependências, que são representados no modelo por meio de estereótipos. Dentre os tipos especificados, apenas a dependência de permissão (*Permission*) e uso (*Usage*) são detectáveis a partir de um código Java. A dependência de permissão garante a um elemento de modelo acessar elementos em outro *namespace*[1]. A dependência de uso decorre de um relacionamento no qual um elemento requer um outro elemento para sua implementação ou operação. Este tipo de dependência abrange 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 maioria das ferramentas analisadas se mostrou ineficiente em detecções de características relevantes de associações e dependências [8]. Principalmente, dependências entre pacotes foram ignoradas. Apenas uma ferramenta (Structure Builder [4]) tratou o caso de coleções não tipadas em atributos, decidindo sempre por representar estes atributos como associações de cardinalidade N (o que nem sempre é verdadeiro).

[1] *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*.

3. Uma Ferramenta de Engenharia Reversa de Java para UML

A ferramenta proposta neste trabalho encontra-se acoplada ao Ambiente de Desenvolvimento de Software (ADS) Odyssey [11]. O Odyssey 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 [12], assim como as de Engenharia de Aplicação, o Odyssey-AE [13]. O ambiente utiliza extensões de diagramas da UML para representar o conhecimento de um domínio, e permite que esses diagramas sejam reutilizados para facilitar a construção de aplicações.

Nesta seção, apresenta-se a arquitetura da ferramenta ARES e detalha-se o funcionamento de cada módulo.

3.1 Arquitetura Proposta

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, pacotes e seus relacionamentos.

A arquitetura da ferramenta está organizada em três filtros (Figura 1). 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 de Inferência, no qual 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.

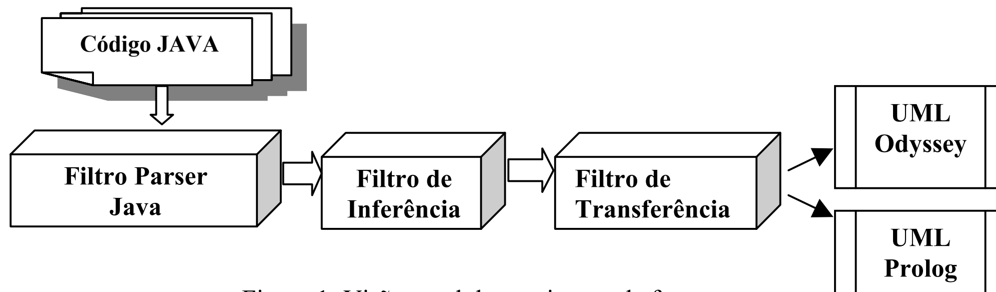

Figura 1: Visão geral da arquitetura da ferramenta.

1. 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. A saída do Filtro *Parser* é uma estrutura em memória de elementos do metamodelo. A estrutura criada é independente da linguagem utilizada, por ser a instância do metamodelo derivado daquele da UML.

2. Filtro de Inferência: é responsável pela manipulação dos dados brutos extraídos pelo Filtro *Parser*. O elemento de entrada é a lista de arquivos. 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.

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. 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 pacote é visitado, e cada classificador contido no pacote é analisado. O classificador visitado é considerado origem de um conjunto de possíveis relacionamentos. Após a análise do pacote, o próximo pacote deve ser visitado (o irmão à direita, primeiro sobrinho ou termina o percurso caso não existam).

3. Filtro de Transferência: A função deste filtro é exportar, para um 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 largura e cada nó (um pacote) é exportado. Posteriormente, a lista de relacionamentos também é percorrida, e cada entrada na lista (uma ligação) é exportada.

Neste trabalho, foram implementadas duas formas de exportação do modelo. Uma faz a exportação para o ambiente de modelagem do Odyssey. Na segunda forma, a ferramenta possibilita a exportação do modelo UML extraído a partir do código para uma base de fatos Prolog. A implementação desta forma de exportação foi feita com o objetivo de verificar a flexibilidade da ferramenta em adequar novas formas de exportação. A base em Prolog apresenta uma sintaxe bastante simplificada, além de ser útil para trabalhos de interesse da equipe. Uma futura forma de exportação pode ser implementada para dar suporte ao formato XMI [9], para permitir o intercâmbio com outros ambientes/ferramentas de modelagem que utilizam UML. 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 possibilitou a implementação, no ambiente Odyssey, de uma abordagem de detecção de construções de projeto boas (*Patterns*) e ruins (*Anti-Patterns*) proposta em [14]. Os predicados utilizados para o mapeamento do modelo extraído são definidos no trabalho citado.

O ambiente de modelagem do ADS tem a funcionalidade de "arrastar e soltar" (*drag and drop*). Um elemento semântico, que tenha representação léxica válida em um diagrama ativo na parte esquerda, pode ser arrastado da árvore para o diagrama (Figura 2). A ferramenta obtém 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 diagramas de classes. Os relacionamentos (léxicos) são exibidos automaticamente conforme os elementos participantes sejam arrastados para o diagrama.

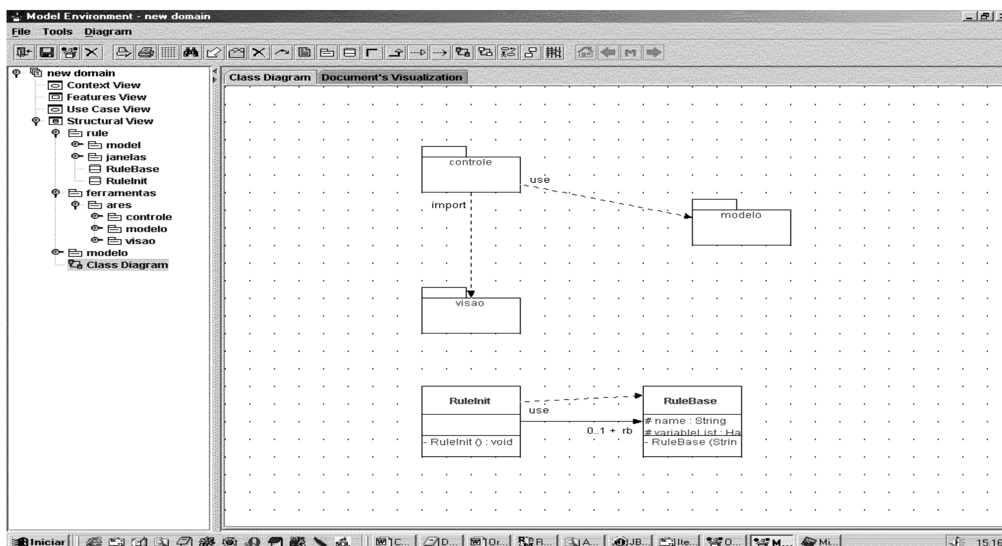

Figura 2: O ambiente de modelagem do Odyssey.

4. Conclusões

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 a diminuição de custos nela envolvidos. A ferramenta 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 ela fosse capaz de extrair diagramas de colaboração e/ou seqüência a partir do código.
Agradecimentos: Ao CNPq pelo apoio financeiro.

5. Referências

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

[2] OBJECT INSIGHT, "JVision 1.4.2", disponível na Internet em http://www.object-insight.com. Último acesso: fevereiro de 2001.

[3] EMBARCADERO TECHNOLOGIES, "GDPro 5.1", disponível na Internet em http://www.gdpro.com. Último acesso: março de 2001.

[4] WEBGAIN, "Structure Builder 4.0", disponível na Internet em http://www.webgain.com. Último acesso: março de 2001.

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

[6] TOGETHER SOFT, "Together 5.01", disponível na Internet em http://www.togethersoft.com. Último acesso: março de 2001.

[7] RATIONAL, "Rose 2000", disponível na Internet em www.rational.com/rose. Último acesso: fevereiro de 2001.

[8] VERONESE, G. O., NETTO, F. J., "ARES: Uma Ferramenta de Auxílio à Recuperação de Modelos UML de Projeto", Orientadores: Claudia Werner e Alexandre Luis Correa, IM-DCC/UFRJ, outubro de 2001.

[9] OMG, "OMG Unified Modeling Language Specification", disponível na Internet em http://www.omg.org/uml. Último acesso: abril de 2001.

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

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

[12] BRAGA, R. M. M., "Busca e Recuperação de Componentes em Ambientes de Reutilização de Software", Tese de Doutorado, COPPE/UFRJ, Rio de Janeiro, 2000.

[13] MILLER, N., "A Engenharia de Aplicações no Contexto da Reutilização Baseada em Modelos de Domínio", Dissertação de Mestrado, COPPE/UFRJ, Rio de Janeiro, 2000.

[14] CORREA, A. L., "Uma Arquitetura de apoio para Análise de Modelos Orientados a Objetos", Dissertação de Mestrado, COPPE/UFRJ, Rio de Janeiro, 1999.