COPPE/UFRJ

ROOSC: UMA ABORDAGEM DE REENGENHARIA DE SISTEMAS ORIENTADOS A OBJETOS PARA COMPONENTES BASEADA EM MÉTRICAS

Ana Maria da Mota Moura

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador(es): Cláudia Maria Lima Werner
Aline Pires V. de Vasconcelos

Rio de Janeiro
Março de 2009

ROOSC: UMA ABORDAGEM DE REENGENHARIA DE SISTEMAS ORIENTADOS A OBJETOS PARA COMPONENTES BASEADA EM MÉTRICAS

Ana Maria da Mota Moura

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

_____

Profª. Cláudia Maria Lima Werner, D.Sc.

_____

Profª.Aline Pires Vieira de Vasconcelos, D.Sc.

_____

Prof. Guilherme Horta Travassos, D.Sc.

_____

Prof.Antonio Francisco do Prado, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
MARÇO DE 2009

Moura, Ana Maria da Mota

ROOSC: Uma Abordagem de Reengenharia de Sistemas Orientados a Objetos para Componentes Baseada em Métricas/ Ana Maria da Mota Moura. – Rio de Janeiro: UFRJ/COPPE, 2009.

XVI, 137 p.: il.; 29,7 cm.

Orientador: Claudia Maria Lima Werner

Aline Pires Vieria de Vasconcelos

Dissertação (mestrado) – UFRJ/ COPPE/ Programa de Engenharia de Sistemas e Computação, 2009.

Referencias Bibliográficas: p. 119-125.

1. Reutilização de Software. 2. Reengenharia. 3. Desenvolvimento Baseado em Componentes. I. Werner, Claudia Maria Lima *et al*. Vasconcelos, Aline Pires Vieira de. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Titulo.

Aos meus pais.

Agradecimentos

A Deus, por todas as bênçãos!

Aos meus pais, simplesmente por tudo!

Ao meu irmão Luís Carlos, pelas palavras de carinho, amor e amizade inigualáveis!

Um agradecimento especial ao Rafael Monteiro, pelo companheirismo, carinho, paciência, otimismo e amor.

À professora Cláudia Werner, minha orientadora, por ter me aceito como sua orientanda e membro deste excelente grupo, além da paciência, confiança e compreensão dos momentos difíceis pelos quais passei. Muito obrigado por compartilhar parte de seu conhecimento, me mostrando o caminho da pesquisa.

À professora e amiga Aline Vasconcelos, minha co-orientadora, que acreditou em mim em todos os momentos, mesmo antes de ingressar no mestrado. Sem ela, tudo teria sido mais difícil, já que sua dedicação, compreensão e amizade contribuíram fundamentalmente para que eu seguisse adiante.

Ao professor Guilherme Travassos, por ter aceitado participar desta banca e, mostrar-se sempre interessado e atencioso no compartilhamento dos seus conhecimentos.

Ao professor Antônio Prado, por ter aceitado participar desta banca.

Aos meus amigos, pelo apoio, carinho, ajuda, companheirismo, paciência, por todos os momentos que cada um contribuiu com um toque especial. Em especial a Roberta Claudino, Priscila de Moura, Fernanda Paes, Lídia Moura Neta.

A minha família, pela confiança, otimismo, apoio incondicional e amor. Sem vocês, nada disto seria possível.

Um agradecimento especial ao Leonardo Murta, por ter aceitado participar dos estudos experimentais, por sua amizade e pelo conhecimento compartilhado.

Aos amigos do grupo de reutilização da COPPE/Sistemas, cujo apoio e amizade me ajudaram muito nessa caminhada.

Aos amigos da Petrobras, cujo apoio e amizade foram muito significativos.

À CAPES, pelo apoio financeiro ao desenvolvimento deste trabalho.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de mestre em Ciências (M.Sc.)

ROOSC: UMA ABORDAGEM DE REENGENHARIA DE SISTEMAS ORIENTADOS A OBJETOS PARA COMPONENTES BASEADA EM MÉTRICAS

Ana Maria da Mota Moura

Março/2009

Orientadoras: Cláudia Maria Lima Werner
Aline Pires Vieira de Vasconcelos

Programa: Engenharia de Sistemas e Computação

Embora os sistemas orientados a objetos (OO) possam ser construídos para ser manuteníveis, na medida em que eles sofrem modificações, podem vir a tornar-se difíceis de serem mantidos, como qualquer outro software. O conhecimento sobre o negócio, embutido nestes sistemas, é difícil de ser reutilizado caso bons princípios de projeto não sejam seguidos. A dificuldade de manter e reutilizar esses softwares faz com que as empresas busquem soluções como o desenvolvimento baseado em componentes (DBC), que possibilita uma maior manutenibilidade e reusabilidade, através da separação entre funcionalidades providas e sua implementação via interfaces. A fim de migrar para o paradigma de DBC, pode-se utilizar uma abordagem de reengenharia, que é o exame e alteração de um sistema para reconstituí-lo em uma nova forma e a implementação desta nova forma. Neste contexto, esta dissertação propõe a abordagem ROOSC de reengenharia de sistemas OO para componentes no nível de modelo, provendo um conjunto de métricas e reestruturações para a re-organização de agrupamentos de classes (i.e. pacotes) candidatos a componentes, além de um conjunto de diretrizes para a formação de componentes e interfaces, apoiados por um ferramental integrado a um ambiente de reutilização. As abordagens existentes não oferecem apoio sistemático como a ROOSC, ou trabalham no nível de programação, além de nem sempre prover apoio ferramental junto a um ambiente de desenvolvimento.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

ROOSC: A REENGINEERING APPROACH FROM OBJECT ORIENTED SYSTEMS TO COMPONENTS BASED ON METRICS

Ana Maria da Mota Moura

March/2009

Advisor: Cláudia Maria Lima Werner
Aline Pires Vieira de Vasconcelos

Department: Computer and Systems Engineering

Although Object Oriented (OO) systems may be constructed to be maintainable, whenever they suffer modifications, they can become difficult to be maintained just like any other software. The business knowledgement, embedded in these systems, is difficult to be reused if no good design principles are followed. The difficulty to maintain and reuse this software makes companies look for solutions such a component-based development (CBD), which allows greater reusability and maintainability, through separation of provided functionality and its implementation by interfaces. In order to migrate to the DBC paradigm, it can de used a reengineering approach, which is the examination and modification of a system to reconstitute it in a new form and the implementation of this new form. Thus, this dissertation proposes the ROOSC approach, OO systems reengineering to components at the model level, proving a set of metrics for the restructuring and re-organization of groups of classes (i.e. packages) to generating candidate components, besides a set of guidelines for the generation of components and interfaces, supported by a tool set integrated in a reuse environment. The existing approaches do not provide systematic support as ROOSC, or work at the programming level, and not always provide a tool support integrated into a development environment.

Índice:

Índice de Figuras

Índice de Tabelas

Lista de Acrônimos

Acrônimo	Termo Relacionado
ADS	Ambiente de Desenvolvimento de Software
Ares	Ferramenta de engenharia reversa
DBC	Desenvolvimento Baseado em Componentes
EA	Engenharia de Aplicação
ED	Engenharia de Domínio
ER	Engenharia Reversa
GenComp	Ferramenta de apoio à estratégia de geração de componentes e interfaces proposta
LP	Linha de Produtos
OO	Orientação a Objetos
ORC	Ferramenta de apoio à estratégia de reestruturação proposta
ROOSC	Abordagem proposta para a reengenharia de sistemas OO para componentes
TraceMining	Ferramenta para apoio à mineração e reconstrução de elementos arquiteturais.
Tracer	Ferramenta de coleta de rastros de execução,
UML	Linguagem de modelagem unificada.
XMI	*XML Metadata Interchange*
XML	*eXtensible Markup Language*

Capítulo 1 Introdução

1.1 Contexto

As organizações têm exigido cada vez mais softwares de qualidade e entregues em menor tempo. Neste contexto, técnicas de reutilização, como o desenvolvimento baseado em componentes, têm assumido um papel importante sobre o tradicional processo de desenvolvimento de software. O desenvolvimento baseado em componentes (DBC) é o desenvolvimento de software através da integração planejada de pedaços pré-existentes de software (BROWN, 2000). Com esta abordagem de desenvolvimento, busca-se maior produtividade, menor custo e maior qualidade, vindo ao encontro das necessidades da indústria.

Ao mesmo tempo, alguns softwares legados, indispensáveis ao funcionamento das atividades da organização, precisam ser mantidos. Ao longo da vida destes softwares, são realizadas modificações, seja para atender algum novo requisito não funcional, corrigir erros, realizar adaptações de *software* ou *hardware*, adicionar novos requisitos funcionais. É importante ressaltar que estes softwares guardam conhecimento a respeito de regras de negócio, decisões de projeto, lógica de programação, os quais não podem ser perdidos.

Estes softwares legados podem ter sido construídos utilizando-se diferentes paradigmas (ex.; orientado a objetos – OO). Embora os softwares OO possam ser construídos para ser manuteníveis, com o passar do tempo, as modificações realizadas deterioram o projeto inicialmente realizado e as manutenções vão ficando cada vez mais difíceis de ser realizadas, tornando estes softwares legado (NIERSTRASZ, *et al.*, 2005).

Na busca por soluções que aliem a crescente demanda por softwares caracterizada por softwares de qualidade, em menor tempo e custo, que atendam as necessidades dos usuários e que possam ser mantidos mais facilmente, tem-se buscado novas soluções. Neste contexto, pode-se mais uma vez utilizar a idéia do DBC que introduz a definição de componentes. Onde, segundo SAMETINGER (1997): "componentes de software são artefatos auto-contidos, claramente identificáveis, que descrevem ou realizam uma função específica e têm interfaces claras, documentação apropriada e um grau de reutilização definido".

Como as funcionalidades providas pelos componentes são acessíveis somente através de suas interfaces, as modificações realizadas em sua implementação não afetam os seus clientes, se as interfaces dos componentes forem mantidas. Isto facilita a manutenção do software. Além disto, a reutilização destas funcionalidades pode ocorrer de forma mais fácil que as classes de um projeto OO, uma vez que estes componentes estão acoplados através de interfaces e isto não é garantido em um sistema OO. Segundo PRADO (2005), no paradigma OO, a reusabilidade e manutenibilidade de software podem ser melhoradas, principalmente, com a adoção de padrões, *frameworks* (GAMMA, *et al.*, 2000) e com componentes de software já existentes e testados.

Apesar da mudança do paradigma da orientação a objetos para componentes trazer ganhos em manutenibilidade e reusabilidade, esta mudança de paradigma não é algo trivial. A fim de apoiar a migração do paradigma OO para componentes, pode-se utilizar a técnica de reengenharia, onde "reengenharia é o exame e alteração de um sistema para reconstituí-lo em uma nova forma e a implementação desta nova forma." (CHIKOFSKY e CROSS, 1990).

A reengenharia poderá trazer não somente o ganho de manutenibilidade, mas também a possibilidade do reúso dos componentes gerados em novos desenvolvimentos, possibilitando o aproveitamento de conhecimento presente no software legado OO.

Vale ressaltar que os componentes podem ser desenvolvidos com o paradigma OO (PRESSMAN, 2006), possibilitando o re-aproveitamento dos elementos estáticos do projeto OO (ex: classes, interfaces e pacotes) no projeto interno dos componentes. Este re-aproveitamento torna o projeto interno do componente mais elaborado, uma vez que aproveita padrões de projeto e outras soluções mais facilmente utilizadas quando o projeto é criado baseando-se em OO.

1.2 Motivação

Na literatura, existem diversas abordagens de reengenharia para componentes (LEE, *et al.*, 2001; LEE, *et al.*, 2003; GANESAN e KNODEL, 2005; PRADO, 2005; WASHIZAKI e FUKAZAWA, 2005; WANG, *et al.*, 2006; CHARDIGNY, *et al.*, 2008; WANG, *et al.*, 2008).

Entre as abordagens pesquisadas nesta dissertação, em (PRADO, 2005) o engenheiro de software faz a reengenharia a partir de sistemas procedurais para

componentes, minimizando as oportunidades de aproveitar as soluções de projeto OO como padrões de projeto.

Algumas abordagens estão relacionadas à linguagem de implementação do sistema alvo (WASHIZAKI e FUKAZAWA, 2005; WANG, *et al*., 2006), dificultando a aplicação da abordagem em sistemas desenvolvidos em diferentes linguagens de implementação.

Outras abordagens existentes não oferecem apoio sistemático como diretrizes para o agrupamento das classes nos componentes (GANESAN e KNODEL, 2005), ou a composição das interfaces destes componentes (LEE, *et al.*, 2001; CHARDIGNY, *et al.*, 2008; WANG, *et al.*, 2008). Além disto, algumas abordagens não realizam a reengenharia do sistema como um todo, mas sim de partes deste sistema (LEE, *et al.*, 2001; GANESAN e KNODEL, 2005; WASHIZAKI e FUKAZAWA, 2005).

Dependendo do tamanho do sistema alvo, a abordagem pode exigir um esforço muito grande por parte do engenheiro, se não existir um ferramental de apoio como em (LEE, *et al.*, 2001; LEE, *et al.*, 2003; GANESAN e KNODEL, 2005; CHARDIGNY, *et al.*, 2008; WANG, *et al.*, 2008). Além de possuir um ferramental de apoio, é importante que os artefatos gerados após a reengenharia possam ser reutilizados durante o processo de desenvolvimento. Esta reutilização é facilitada à medida que o ferramental de apoio esteja integrado a um ambiente de desenvolvimento (ADS) que apóie as etapas do processo de desenvolvimento.

Por fim, apesar das abordagens existentes na literatura, ainda há a necessidade de contribuições nos pontos destacados anteriormente.

1.3 Objetivos

Diante das motivações expostas, esta dissertação apresenta como objetivo central propor uma abordagem sistemática de apoio à reengenharia de software orientado a objetos (OO) para componentes baseada em métricas, denominada ROOSC (*Reengineering Object Oriented Software to Components*). Através desta abordagem, há a possibilidade de re-aproveitar conhecimento e esforço presentes em um software legado OO.

A fim de atingir esse objetivo central, a abordagem ROOSC, proposta nesta dissertação, envolve uma estratégia de reestruturação, que visa a re-organização de agrupamentos de classes (i.e. pacotes) para gerar candidatos a componentes de acordo com os princípios de DBC; e uma estratégia para a geração de componentes e interfaces

propriamente, também conforme os princípios de DBC, visando gerar um ganho em manutenibilidade e reusabilidade.

A fim de apoiar a reutilização, a ROOSC reestrutura os agrupamentos de classes (i.e. pacotes) com base em métricas adaptadas do contexto da OO para o contexto de DBC. Uma vez que pacotes são utilizados para agrupar elementos em OO segundo qualquer critério, a estratégia de reestruturação da abordagem ROOSC facilita a manutenção da aplicação ao re-organizar estes pacotes, tornando-os mais coesos e menos acoplados. Assim, ao final da reestruturação, os elementos que estão agrupados em cada pacote tendem a ser modificados em conjunto, facilitando a manutenção em pacotes localizados.

Finalmente, a estratégia de geração de componentes e interfaces visa apoiar a formação dos componentes e das interfaces conforme princípios de projeto de componentes, provendo maior manutenibilidade e reusabilidade ao software.

A fim de minimizar o esforço na utilização da abordagem ROOSC, foi desenvolvido um ferramental de apoio integrado a um ambiente de desenvolvimento de software. O seu ferramental é composto por duas ferramentas ORC (*Object Restructuring to Components*), apoio a reestruturação, e GenComp (*Generating Components*), apoio a geração de componentes e interfaces.

Para verificar se estes objetivos foram atendidos, estudos experimentais foram conduzidos sobre cada uma das estratégias que compõe a abordagem ROOSC. Através desses estudos, foi possível obter indícios da viabilidade da ROOSC na reengenharia de software OO para componentes.

1.4 Projetos Relacionados

Esta pesquisa se insere no contexto do Projeto Odyssey (WERNER, *et al.*, 1999), que visa o desenvolvimento do ambiente Odyssey (ODYSSEY, 2008), e do Projeto Reuse (WERNER, 2005), cujo objetivo principal é explorar técnicas e ferramentas que apóiem a reutilização de software, permitindo a evolução do ferramental desenvolvido pelo grupo de reutilização da COPPE/UFRJ.

O Odyssey é um ambiente de reutilização baseado em modelos de domínio, provendo tecnologias de apoio à engenharia de domínio (ED), linha de produto (LP) e DBC. O Odyssey cobre tanto o desenvolvimento para reutilização, através de processos de ED (BRAGA, 2000; BLOIS, 2006), quanto o desenvolvimento com reutilização, através da Engenharia de Aplicação (EA) (MILER, 2000). Os modelos criados na ED

são instanciados através de um processo de EA, que tem por objetivo a construção de aplicações em um determinado domínio.

1.5 Organização

Partindo desta Introdução, esta dissertação está organizada em mais 5 capítulos, da seguinte forma:

No **Capítulo 2,** é feita uma revisão da literatura acerca dos conceitos necessários para compreender a abordagem proposta nesta dissertação. São abordados temas essenciais sobre DBC, métricas no projeto OO de componentes e reengenharia de software. Além disto, são apresentadas algumas abordagens de reengenharia de software OO para componentes fazendo, ao final, um comparativo entre as abordagens.

No **Capítulo 3** é apresentada a abordagem proposta nesta dissertação, i.e. a ROOSC, sendo descritas detalhadamente as suas 2 estratégias, a saber: reestruturação de agrupamentos de classes (i.e. pacotes), e a geração de componentes e interfaces.

No **Capítulo 4**, é detalhado o ferramental de apoio à execução das atividades de reestruturação e geração de componentes e interfaces, descrevendo a arquitetura, tecnologias empregadas, funcionalidades (através de exemplos de utilização), as contribuições de cada um dos protótipos desenvolvidos.

No **Capítulo 5** são descritos os estudos experimentais conduzidos para avaliar a viabilidade das estratégias de reestruturação e geração de componentes e interfaces.

Finalmente, o **Capítulo 6** descreve as contribuições alcançadas nesta dissertação em função dos objetivos estabelecidos, limitações identificadas em cada etapa da abordagem proposta, além de perspectivas para trabalhos futuros.

O **Anexo A** apresenta a instrumentação do primeiro estudo experimental, que avalia a viabilidade da estratégia de reestruturação. O **Anexo B** apresenta a instrumentação do segundo estudo experimental, que avalia a viabilidade da estratégia de geração de componentes e interfaces.

Capítulo 2 Revisão da Literatura

2.1 Introdução

Uma vez que sistemas de software evoluem e requerem constantes modificações, estes devem estar preparados para evoluir ou serão desativados (NIERSTRASZ, *et al.*, 2005). Alguns sistemas são muito importantes para a organização e não podem ser desativados, apesar da dificuldade de manutenção. Nesse contexto, o conhecimento adquirido com um sistema pode ser utilizado como base para a sua evolução contínua e para o desenvolvimento de novos sistemas.

Na medida em que o software orientado a objetos (OO) sofre manutenções, este pode se tornar mal estruturado, cada vez mais difícil de evoluir ou reutilizar alguma de suas partes, assim como outro software qualquer. Pode-se encontrar em (ARAÚJO e TRAVASSOS, 2006) informações sobre estudos experimentais para a observação do decaimento de software. A dificuldade em atualizar o software continuamente tem motivado a investigação de soluções que possam diminuir o seu custo de desenvolvimento, garantam um tempo de vida maior e facilitem a sua manutenção (NIERSTRASZ, *et al.*, 2005). Ao mesmo tempo, surgem na literatura novos paradigmas que se propõem a estruturar o software de um modo diferente, tentando obter alguma vantagem sobre os paradigmas anteriores, como o desenvolvimento baseado em componentes (DBC), que visa obter um novo software a partir de componentes pré-existentes (FAVRE, *et al.*, 2001).

A transformação de um software OO para componentes pode trazer o aumento da sua manutenibilidade e reusabilidade, proporcionando a diminuição dos custos de novos desenvolvimentos e manutenções futuras. Estas vantagens provêem da separação entre a especificação do componente e a sua implementação, além da divisão dos serviços dos componentes em interfaces, evitando-se que o impacto das mudanças na implementação de um componente afete os demais, desde que os serviços continuem garantidos pelas interfaces. Apesar destas vantagens, esta mudança de paradigma não é trivial, podendo ser realizada através da reengenharia.

A reengenharia tem como principal objetivo melhorar o sistema, através de alterações significantes que proporcionem melhoria sem alterar suas funcionalidades

(WARDEN, 1992). Utilizando a reengenharia, os sistemas de software críticos das organizações podem ser aprimorados, fazendo com que o conhecimento de negócio intrínseco não seja perdido e, ao mesmo tempo, garantindo a manutenção das suas funcionalidades originais.

Várias definições de reengenharia indicam que o software deve ser melhorado qualitativamente ao sofrer uma reengenharia (WARDEN, 1992; SOMMERVILLE, 1995; PRESSMAN, 2006). Entretanto, não definem o que é considerado como um resultado de qualidade. Lemos *et al.* (2003) afirmam que é importante que o resultado da reengenharia seja confiável. Desta forma, a garantia da qualidade também pode ser considerada como mais uma etapa da reengenharia.

Segundo PRESSMAN (2006), os fatores que afetam a qualidade podem ser categorizados em dois amplos grupos: (1) fatores que podem ser medidos diretamente (por exemplo, defeitos por ponto por função) e; (2) fatores que são medidos indiretamente (por exemplo, usabilidade ou manutenibilidade). Em ambos os casos devem ocorrer medições, comparando o resultado referente ao software a algum valor a fim de se chegar a uma indicação de qualidade. Para alcançar este objetivo, podem ser utilizadas métricas que, segundo o IEEE *Standard Glossary* (IEEE, 1993), são definidas como uma medida quantitativa do grau em que um sistema, componente ou processo possui um determinado atributo. Por exemplo, a métrica CBO de (CHIDAMBER e KEMERER, 1994) mede o total de classes as quais uma determinada classe está acoplada.

Este capítulo apresenta a teoria sobre o DBC, as abordagens de reengenharia e métricas no contexto do DBC e da reengenharia. Deste modo, a Seção 2.2 apresenta o DBC e seus principais conceitos; a Seção 2.3 apresenta algumas métricas utilizadas no contexto do DBC; enquanto a Seção 2.4 apresenta os principais conceitos de reengenharia e uma comparação entre algumas abordagens de reengenharia para componentes. Por fim, a Seção 2.5 conclui o capítulo apresentando as considerações finais.

2.2 Desenvolvimento Baseado em Componentes (DBC)

O desenvolvimento baseado em componentes é o desenvolvimento de software através da integração planejada de pedaços pré-existentes de software (BROWN, 2000). Com esta abordagem de desenvolvimento, busca-se maior produtividade, menor custo e maior qualidade. Estes pedaços pré-existentes são chamados componentes. Várias

definições de componentes são encontradas na literatura. Dentre elas, a de que um componente pode ser definido como uma unidade de software independente, que encapsula dentro de si seu projeto e implementação, oferecendo serviços por meio de interfaces bem definidas para o meio externo (GIMENES, *et al.*, 2000). Na Seção 2.2.1, o conceito de componente será abordado em maiores detalhes.

O objetivo de construir sistemas a partir de pedaços bem definidos não é novo, vindo de uma longa história de trabalho em sistemas modulares, projeto estruturado e a maioria dos sistemas OO (BROWN, 2000). Segundo BROWN (2000), esta técnica se apóia na prática de "dividir e conquistar", enfatizando o projeto de soluções em termos de pedaços de funcionalidades providas como componentes, acessíveis somente através de interfaces bem definidas.

A técnica de DBC visa fornecer um conjunto de procedimentos, ferramentas e notações que possibilitem que, ao longo do processo de desenvolvimento de software, ocorra tanto a produção de novos componentes quanto a reutilização de componentes existentes (GIMENES, *et al.*, 2000).

Confirmando esta definição, SPAGNOLI e BECKER (2003) afirmam que o DBC pode considerar o desenvolvimento de componentes ou o desenvolvimento com componentes. A primeira perspectiva engloba as atividades envolvidas na concepção e implementação de um componente, enquanto a segunda considera a existência de componentes e engloba as atividades necessárias para o desenvolvimento de software a partir da composição destes componentes. A ênfase desta dissertação está na especificação de componentes a partir de software OO pré-existente, e, na Seção 2.2.2 , é feita uma apresentação geral de alguns métodos de DBC que permitem especificar componentes em UML (*Unified Modeling Language*), que é uma linguagem amplamente utilizada para a especificação de projeto OO.

VITHARANA *et al*. (2004) definem a fabricação de componentes como sendo o processo de desenvolver componentes e implementar suas interfaces correspondentes. Nestes casos, onde os componentes são desenvolvidos, a maior ênfase está no projeto visando à reutilização destes componentes em muitos sistemas, alguns que ainda nem mesmo existem (CRNKOVIC, *et al.*, 2006). De acordo com CHEESMAN e DANIELS (2001), o DBC se diferencia das outras abordagens de desenvolvimento (ex: orientação a objetos) através da separação entre a especificação do componente e a sua implementação, além da divisão dos serviços dos componentes em interfaces.

O software baseado em componentes visa prover algumas vantagens como:

capacidade de reutilização, rápido desenvolvimento, economia, acessibilidade e adaptabilidade (DISESSA, *et al.*, 2004). Para que estas vantagens possam ser alcançadas, o processo de desenvolvimento precisa ser preparado para o DBC. De acordo com WERNER e BRAGA (2005), o DBC, como toda abordagem de desenvolvimento de software, precisa de um método cuidadosamente planejado e controlado para ser efetivo.

2.2.1 Componentes

A idéia do DBC é montar produtos finais de software a partir de componentes reutilizáveis, mas a compreensão desta técnica de desenvolvimento é alcançada quando se entende o significado do que é um componente. Um componente, como dito anteriormente, é definido como uma unidade de software independente, que encapsula dentro de si seu projeto e implementação, e oferece interfaces bem definidas para o meio externo (GIMENES, *et al.*, 2000).

Uma outra definição clássica para componente e que é adotada nesta dissertação é a apresentada por SAMETINGER (1997): "componentes de software são artefatos auto-contidos, claramente identificáveis, que descrevem ou realizam uma função específica e têm interfaces claras, documentação apropriada e um grau de reutilização definido".

Segundo BLOIS (2006), um componente auto-contido é aquele que não precisa de outro componente para ser reutilizável, mas quando há a necessidade de cooperação com outros componentes, esta é feita através de suas interfaces. As interfaces dos componentes permitem separar a sua especificação da implementação. Por meio das suas interfaces, um componente pode se unir a outros componentes e dar origem aos sistemas baseados em componentes (D'SOUZA e WILLS, 1999). Cada componente é projetado para trabalhar em uma variedade de contextos e trabalhar em conjunto com uma variedade de outros componentes. Muitos produtos finais podem ser projetados com o auxílio de vários componentes, como dito anteriormente.

Componentes podem ser construídos pela própria organização ou adquiridos de terceiros. Os componentes comprados são conhecidos como "componentes comerciais" (*Commercial Off-The-Shelf* - COTS). Componentes COTS são componentes genéricos que podem ser prontamente adquiridos no mercado (de prateleira) (GIMENES, *et al.*, 2000), enquanto os componentes desenvolvidos na própria organização podem ser criados para as aplicações de um domínio. Componentes podem ainda ser adquiridos a

partir de bibliotecas públicas, podendo, neste caso, ser adquiridos através de contratos de licenças pagas ou gratuitas, por exemplo, em (SOURCEFORGE, 2009).

Um modelo de componentes define uma série de regras que devem ser seguidas pelo componente para que ele possa se integrar a outros componentes. A fim de se obter as vantagens da reutilização de componentes, é necessário que as regras do modelo de componentes sejam estabelecidas através de métodos precisos para a sua produção e utilização (CARNEY, 1997). Esta definição de componentes está bastante relacionada ao ambiente físico que suportará os componentes.

O produto final gerado pela abordagem proposta, nesta dissertação, é um modelo de componentes que está bem próximo do que CHEESMAN e DANIELS (2001) definem como arquitetura de componentes. Onde uma arquitetura de componentes é um conjunto de componentes no nível de aplicação, suas relações estruturais e dependências. Detalhando o conteúdo desta arquitetura, temos os seguintes itens:

- Conjunto de componentes de software, suas relações estruturais e dependências de comportamento;
- Definição lógica e independente de tecnologia;
- Relações estruturais: associações e herança entre interfaces e especificações de componentes e relações de composição entre componentes e;
- Dependências de comportamento: relações de dependência entre diferentes componentes, componentes e suas interfaces, e entre interfaces.

Como apresentado na Seção 1.1 componentes podem ser desenvolvidos utilizando outros paradigmas como a OO, combinando um conjunto de objetos e classes para alcançar um conjunto de serviços ou funcionalidades (VITHARANA, *et al.*, 2004). Ainda segundo PRESSMAN (2006), no contexto da engenharia de software OO, o componente contém um conjunto de classes colaborativas. Deste modo, é importante diferenciar componentes de simples objetos, porque os componentes podem ser vistos como um modo convencional de empacotar objetos e torná-los disponíveis através de montagem em um grande sistema de software.

Outra definição de componentes que é utilizada nesta dissertação é componentes de sistema, que é definida em (CHEESMAN e DANIELS, 2001), onde os componentes que contêm a lógica da aplicação podem ser divididos em: componentes de sistema e componentes de negócio. Componentes de sistema representam funcionalidades da aplicação determinadas pelos casos de uso, enquanto componentes de negócio representam conceitos do negócio e são responsáveis pelo seu gerenciamento.

2.2.1.1 Interfaces

As interfaces podem ser compreendidas como uma representação explícita das funcionalidades de um componente, sendo o meio de comunicação entre os mesmos, como mencionado anteriormente. Deste modo, evita-se que o impacto das mudanças na implementação de um componente afete os demais, desde que os serviços continuem garantidos pelas interfaces. Um componente pode apresentar interfaces providas, representando serviços que ele oferece a outros componentes, e requeridas, representando os serviços que ele necessita de outros componentes.

A Tabela 2.1, adaptada de CHEESMAN e DANIELS (2001), apresenta uma comparação sucinta entre interface e especificação de componentes. É necessário lembrar que, além das interfaces, os componentes precisam de especificações que realizem as interfaces definidas, onde uma especificação é a concretização das operações definidas na interface. Uma especificação de componente descreve uma ou mais classes, cujos objetos colaboram entre si para realizar as operações definidas nas interfaces.

Tabela 2.1: Interface vs Especificação de Componentes, adaptada de (CHEESMAN e DANIELS, 2001)

Interface do componente	Especificação do componente
• Representa o contrato de uso; • Provê uma lista de operações; • Define um modelo de informação lógica específico da interface, i.e. seus atributos; • Especifica como as operações afetam o modelo de informação; • Descreve somente os efeitos locais das operações sobre o modelo de informação.	• Representa a realização do contrato; • Provê uma lista de interfaces suportadas; • Define a unidade de tempo de execução, ou seja, é uma representação codificada; • Define os relacionamentos entre os modelos de informação das diferentes interfaces; • Especifica como as operações devem ser implementadas em termos de uso de outras interfaces.

2.2.1.2 Projeto OO de Componentes

Como mencionado anteriormente, os componentes podem ser desenvolvidos utilizando paradigmas como a OO, combinando um conjunto de objetos e classes para alcançar um conjunto de serviços ou funcionalidades (VITHARANA, *et al.*, 2004). De acordo com WASHIZAKI e FUKAZAWA (2005), nos casos em que se utiliza linguagens OO para a implementação de componentes, os componentes são definidos estruturalmente como um conjunto reutilizável e substituível de classes OO.

O processo para construir um componente requer, além das funcionalidades do componente, que o componente seja construído para ser reutilizado. Reusabilidade implica em generalidade e flexibilidade, e estes requisitos podem significar mudanças nas características do componente (CRNKOVIC, *et al.*, 2006). Neste contexto, PRESSMAN (2006) apresenta alguns princípios básicos de projeto que são bastante adotados quando a engenharia de software OO é aplicada.

Princípio Aberto-Fechado (*Open-Closed Principle* - OCP). "Um módulo (componente) deveria ser aberto para extensão e fechado para modificação" (MARTIN, 2000). De modo que o projetista especifique o componente, permitindo que ele seja estendido dentro do seu domínio funcional sem modificações internas (nível de código ou lógico). Para isto, o projetista trata as funcionalidades prováveis de serem estendidas com abstrações (ex.: interfaces).

Princípio da Substituição de Liskov (*Liskov Substitution Principle* - LSP). "classes base devem ser substituíveis por suas Subclasses" (MARTIN, 2000). Deste modo, todas as subclasses devem respeitar qualquer contrato implícito entre a classe base e os componentes que a utilizam.

Princípio da Inversão de Dependência (*Dependency Inversion Principle* - DIP). "Confie nas abstrações, não confie nas concretizações" (MARTIN, 2000). As abstrações são utilizadas no OCP, como visto anteriormente, por serem o ponto onde o projeto pode ser estendido com maior facilidade. Quanto mais um componente depende de componentes concretos aos invés de interfaces, mais difícil é estendê-lo.

Princípio da Segregação de Interface (*Interface Segregation Principle* – ISP). "Muitas interfaces específicas de clientes são melhores do que uma interface de propósito geral" (MARTIN, 2000). Onde ISP sugere que o projetista deve criar uma interface especializada para atender cada categoria importante de clientes. Somente as operações relevantes a uma categoria de clientes devem ser especificadas na interface daquele cliente. Se múltiplos clientes solicitam as mesmas operações, elas devem ser

especificadas em cada uma das interfaces especializadas.

Princípio de Equivalência de Liberação de Reúso (*Release Reuse Equivalency Principle* – RREP). “A granularidade do reúso é a granularidade da versão” (MARTIN, 2000). Em vez de tratar cada classe individualmente, é frequentemente aconselhável agrupar as classes reusáveis em pacotes que podem ser geridos e controlados à medida que as versões evoluem.

Princípio de Fecho Comum (*Common Closure Principle* - CCP). “Classes que se modificam juntas devem ficar juntas” (MARTIN, 2000). Classes devem ser empacotadas coesivamente, isto é, quando classes são empacotadas como parte de um projeto, elas devem tratar da mesma área funcional e comportamental. Quando há necessidade de modificação em alguma característica, provavelmente estas classes serão impactadas.

Princípio Comum de Reúso (*Common Reuse Principle* – CRP). “Classes que não são reutilizadas juntas, não devem ser agrupadas juntas” (MARTIN, 2000). Se classes não são agrupadas de forma coesa, é possível que uma classe sem relacionamento com outras classes do pacote seja modificada. Isto precipitará integração e testes desnecessários.

Na literatura, há outros trabalhos que enfatizam princípios de projetos de componentes (VITHARANA, *et al.*, 2004; GANESAN e KNODEL, 2005; GONÇALVES, *et al.*, 2006). Em síntese, os princípios são: não deve existir relacionamento de herança entre componentes distintos, pois um componente não deve conhecer a implementação do outro; os componentes devem prover uma funcionalidade específica; classes generalizadas são boas candidatas a comporem componentes, pois permitem maior adaptação e extensão em novos sistemas; a comunicação inter-componentes, feita via suas interfaces, é muito cara em termos de tempo e recursos de plataforma e, por isso, deve-se diminuir o acoplamento entre componentes, a fim de minimizar o tráfego na rede.

2.2.2 Métodos de DBC

Um método ou processo de DBC tanto tem características comuns aos processos de software convencionais quanto contém características peculiares. Essas características peculiares se refletem na adição de estágios técnicos ao processo convencional, bem como numa maior ênfase em práticas já realizadas nesses processos,

sejam eles processos que seguem a abordagem funcional ou orientada a objetos (GIMENES, *et al.*, 2000).

BARROCA *et al*. (2005) diferenciam o desenvolvimento baseado em componentes da maioria dos métodos de desenvolvimento pela possibilidade de adaptação dos requisitos do software aos componentes existentes, que pode ser realizada através de negociação, existindo inclusive uma preocupação com a reutilização de novos componentes gerados durante o processo de desenvolvimento da aplicação. Deste modo, além de reutilizar os componentes existentes, são gerados novos componentes para reutilizar na própria aplicação e nas aplicações futuras.

CRNKOVIC *et al*. (2006) descrevem algumas modificações que podem ser feitas aos processos de desenvolvimento de software para se adequar ao processo de DBC:

- ⇒ **Análise e especificação de requisitos** – Em uma abordagem baseada em componentes, é necessário analisar se os requisitos podem ser atendidos pelos componentes disponíveis. Quando componentes apropriados não estão disponíveis, é necessário desenvolver novos componentes ou negociar os requisitos para que possam ser atendidos pelos componentes existentes.
- ⇒ **Projeto do sistema** – O projeto do sistema é bastante afetado pelos componentes disponíveis, pois os componentes estarão implementados segundo um modelo de componentes e em alguma tecnologia específica. As decisões de projeto serão afetadas assim como os requisitos. Apesar de poder existir os impactos descritos por CRNKOVIC *et al*. (2006), pode-se realizar adaptações de componentes em alguns casos.
- ⇒ **Implementação e teste de unidade** – A implementação do sistema envolve a conexão dos componentes que é feita através do chamado "código de cola". Este código conterá adaptações e a implementação de novas funcionalidades. Apesar dos componentes serem pré-testados, isto não é suficiente. As unidades do projeto são criadas através da montagem de vários componentes com os códigos de cola e estas unidades precisam se testadas.
- ⇒ **Integração de sistema** – A integração do sistema envolve a integração de componentes de infra-estrutura padrão e os componentes da aplicação.
- ⇒ **Validação e verificação** – As técnicas tradicionais de verificação e teste são empregadas neste momento. Os componentes "caixa preta" podem apresentar

mensagens de erro, mas as causas dos erros podem ocorrer em outro componente. As interfaces (contratos) possuem uma função importante, neste momento, pois especificam as entradas e saídas, possibilitando a verificação destes valores (i.e valores de entrada e saída).

⇒ **Manutenção e suporte operacional** – Durante a manutenção e suporte operacional do sistema, pode ocorrer a integração de componentes novos ou modificados (ex.: nova versão), incluindo a adição ou alteração de código de cola. Deste modo, faz-se necessário fazer uma nova verificação do sistema porque novos erros podem ser introduzidos.

Entre os métodos de DBC presentes na literatura, encontram-se o *Catalysis* (D'SOUZA e WILLS, 1999) e o UML *Components* (CHEESMAN e DANIELS, 2001). Estes dois métodos de DBC foram selecionados por serem baseados em UML, que é a mesma linguagem de modelagem utilizada na abordagem proposta nesta dissertação, e por serem bastante referenciados na literatura, sendo descritos resumidamente a seguir.

Tabela 2.2: Associação entre os estágios do processo de desenvolvimento e o *Catalysis* (BARROCA, *et al.*, 2005).

Estágio	Objetivo	Modelo
Especificação de requisitos	Entender o contexto do sistema, a arquitetura e os requisitos não funcionais	Modelo de domínio. Contexto do sistema.
Especificação do sistema	Descrever o comportamento externo do sistema através do modelo do domínio do problema.	Cenários. Modelo de tipos e especificação de operações.
Projeto da arquitetura	Separa os componentes arquiteturais técnicos dos de aplicação e os seus conectores para se alcançar os objetivos de projeto.	Arquitetura lógica da aplicação. Plataforma e arquitetura física.
Projeto interno dos componentes	Projetar interfaces e classes para cada componente, construir e testar.	Especificação de classe e interface.

Catalysis não define um processo único de desenvolvimento, mas contém um conjunto de idéias, conceitos e procedimentos que possibilitam aos projetistas escolher

o melhor processo para o seu projeto (GIMENES, *et al.*, 2000). Na Tabela 2.2, pode-se visualizar uma associação, apresentada em (BARROCA, *et al.*, 2005), entre cada estágio de um processo de desenvolvimento baseado no Catalysis (especificação de requisitos, especificação do sistema, projeto da arquitetura e projeto interno dos componentes) e os respectivos modelos desenvolvidos.

Conforme GIMENES *et al*. (2000), a **especificação de requisitos** visa entender e representar o problema definindo o contexto do sistema envolvido e produzindo um modelo do domínio. A **especificação do sistema** trata a solução de software extraída do modelo do domínio representando os tipos e operações do sistema e seu comportamento exterior. O **projeto da arquitetura** pode ser dividido em duas partes, denominadas arquitetura da aplicação e arquitetura técnica. A arquitetura da aplicação representa a estrutura lógica do sistema como uma coleção de componentes cooperantes com os tipos e operações obtidos na especificação e distribuídos através dos componentes. A arquitetura técnica inclui as partes do sistema independentes do domínio como a infra-estrutura de comunicação de componentes (ex. CORBA ou Java/RMI), a plataforma de hardware e a plataforma de software. O **projeto interno dos componentes** envolve a definição das classes e interfaces dos componentes, a construção (código) e teste dos componentes.

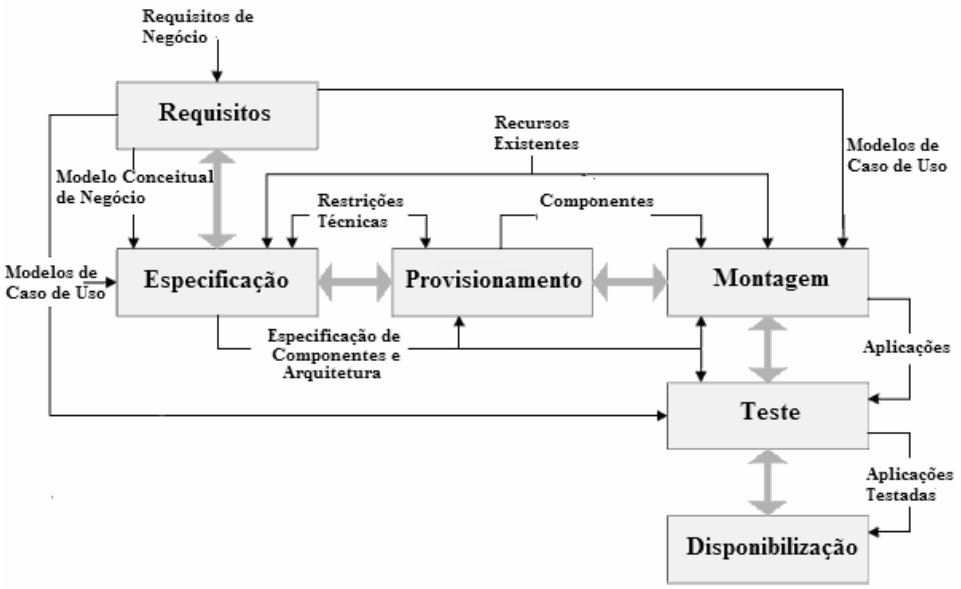


Figura 2.1: **Fluxos do UML *Components* adaptado de (CHEESMAN e DANIELS, 2001).**

UML *Components* é um outro método para DBC que utiliza os recursos da UML para o desenvolvimento de sistemas (CHEESMAN e DANIELS, 2001), o qual

contempla os seguintes fluxos de trabalho, presentes na Figura 2.1: **Requisitos, Especificação, Provisão, Montagem, Teste e Disponibilização**. O método UML *Components* é uma adaptação do RUP (*Rational Unified Process*) com a modificação ou adição de apenas alguns fluxos. Os fluxos de requisitos, teste e disponibilização (*deploymemt*) são os mesmos do RUP e não são discutidos por CHEESMAN e DANIELS (2001), enquanto os fluxos de especificação, provisão e montagem foram criados em substituição aos fluxos de análise e projeto e de implementação, sendo, por isso, discutidos com maior ênfase pelos autores. Na abordagem proposta, nesta dissertação, o método utilizado para gerar parte da especificação dos componentes e interfaces é baseado no UML *Components.* Por isso, os fluxos do UML *Component* são mostrados na Figura 2.1, com suas entradas e saídas, e são detalhados a seguir.

No fluxo de **Requisitos**, é realizado o levantamento e a definição dos requisitos. Este fluxo foi modificado apresentando uma menor elaboração em comparação ao mesmo fluxo no RUP. Neste fluxo, são criados dois artefatos: o modelo conceitual de negócio e o modelo de casos de uso. O modelo conceitual de negócio é um modelo conceitual do domínio do negócio que precisa ser compreendido e acordado, sendo construído com o diagrama de classes da UML. Seu principal propósito é criar um vocabulário comum entre as pessoas envolvidas no projeto. O modelo de casos de uso é um modo de especificar certos aspectos dos requisitos funcionais do sistema, descrevendo interações entre os atores e o sistema, e, auxiliando, dessa forma, a definição da fronteira deste sistema.

O fluxo de **Especificação** objetiva a identificação dos componentes necessários e definição da arquitetura de componentes. Este fluxo é subdividido em três estágios: identificação de componentes, interação de componentes e especificação de componentes. O principal objetivo do estágio de identificação de componentes é: identificar um conjunto inicial de interfaces de negócio para os componentes de negócio; e um conjunto inicial de interfaces de sistema para os componentes de sistema, além de colocá-los juntos em uma arquitetura de componentes (Seção 2.2.1) inicial. Como descrito anteriormente, os componentes de negócio representam os principais conceitos de um domínio e os componentes de sistema representam as atividades e processos relativos ao domínio da aplicação. Neste estágio, devem ser considerados componentes ou qualquer outro recurso de software existente.

O estágio de interação de componentes examina como cada operação do sistema será alcançada usando a arquitetura de componentes. Neste estágio, as dependências

entre componentes são compreendidas claramente, chegando ao nível de operações individuais. O estágio final, que é a especificação do componente, é onde são detalhadas as especificações de operações e restrições, criando especificações precisas das operações, interfaces e componentes.

O fluxo de requisitos e o fluxo de especificação influenciaram a abordagem proposta, nesta dissertação, a forma de especificar os componentes de sistema. Deste modo, estes fluxos foram descritos em maiores detalhes, enquanto os demais fluxos são apresentados apenas com os seus respectivos objetivos.

O fluxo de **Provisão** visa prover os componentes que formam a arquitetura de componentes especificada. Para este fim, pode-se implementar as especificações ou integrar algum software existente. O fluxo de **Montagem** objetiva integrar a aplicação, usando a arquitetura de componentes para definir a estrutura por inteiro e as partes individuais. Os fluxos de **Testes** e **Disponibilização** seguem a definição utilizada no RUP, onde o fluxo de Testes visa principalmente avaliar a qualidade do produto e o fluxo de Disponibilização visa assegurar que o software estará disponível para os seus usuários finais.

2.3 Métricas de Componentes

Na Seção 2.2.1.2, foi discutido o projeto de componentes desenvolvidos com orientação a objeto. Dentro deste contexto, esta seção de métricas para componentes objetiva apresentar sucintamente alguns trabalhos que propõem métricas aplicáveis ao projeto OO de componentes.

Como citado anteriormente, segundo o IEEE *Standard Glossary* (IEEE, 1993), as métricas são definidas como uma medida quantitativa do grau em que um sistema, componente ou processo possui um determinado atributo. Na literatura, encontram-se alguns trabalhos referentes a métricas utilizadas no contexto mencionado (VERNAZZA, *et al.*, 2000; CHO, *et al.*, 2001; VITHARANA, *et al.*, 2004; GONÇALVES, *et al.*, 2006).

Vale ressaltar que o conjunto de métricas proposto, nesta dissertação, é apenas inicial e pode ser estendido. Para determinar as métricas propostas, foram estudados alguns trabalhos da literatura. Estes trabalhos são descritos resumidamente a seguir.

CHO *et al.* (2001) propõem algumas métricas para medir a complexidade do componente (CPC – Complexidade Plana do Componente; CSC – Complexidade Estática do Componente; CDC – Complexidade Dinâmica do Componente; CCC –

Complexidade Ciclomática do Componente), sua capacidade de customização (CV – Variabilidade do Componente) e reusabilidade (CR – Reusabilidade do próprio Componente; CRL – nível de reúso do componente) de componentes. Algumas métricas propostas podem ser utilizadas durante a fase de projeto do componente ou após o seu desenvolvimento. Estas métricas são aplicadas à componentes desenvolvidos com projeto OO, pois as métricas que podem ser coletadas a partir do modelo consideram classes, interfaces, métodos, entre outros que farão parte do projeto interno do componente ou até mesmo suas interfaces providas.

Outras métricas para medir a reusabilidade de um componente estão presentes em (GONÇALVES, *et al.*, 2006). Estes autores utilizam três critérios para definir a reusabilidade de um componente: complexidade, portabilidade e adaptabilidade. Para definir a complexidade de um componente, os autores propõem duas métricas: NCSS (Linhas de Código que não são Comentários) e CC (Complexidade Ciclomática). Para definir a portabilidade, são propostas duas métricas: I (Instabilidade) e D (Distância da Seqüência Principal). A adaptabilidade é determinada pela métrica D (Distância da Seqüência Principal), assim como a portabilidade, pela métrica A (Abstração). O conjunto de métricas, desta dissertação, utiliza a métrica de abstração que também é utilizada em (GONÇALVES, *et al.*, 2006).

VERNAZZA *et al.* (2000) propõem métricas para componentes, considerando componentes como agrupamentos de classes. O conjunto de métricas proposto por estes autores é uma extensão para componentes do conjunto de métricas proposto em (CHIDAMBER e KEMERER, 1994), exceto pela métrica LCOM (Perda de Coesão em Métodos). Este conjunto de métricas é medido a partir do projeto interno do componente e é composto pelas métricas: WCC (Peso de Classe por Componente), NC (Número de Classes), MAXDIT (Máximo de DIT), MUT (Número de MAXDIT para Classes em Árvores não Relacionadas), NOCC (Número de Filhos para um Componente), EXTCBO (CBO Externo), RFCOM (Conjunto de Respostas para um Componente) e CC (Coesão de Componente). Este trabalho, assim como o conjunto de métricas desta dissertação, faz algumas extensões às métricas do (CHIDAMBER e KEMERER, 1994). Estas métricas são muito utilizadas por serem consideradas clássicas, na literatura, e bastante referenciadas.

Em (VITHARANA, *et al.*, 2004), existe um outro conjunto de métricas para medir características técnicas dos componentes. Estas características técnicas são traduzidas a partir dos objetivos gerenciais estabelecidos pelas estratégias de negócio,

visando que o projeto OO dos componentes seja baseado nestas estratégias de negócio. Por exemplo, se o objetivo gerencial está ligado à eficiência do custo para a criação dos componentes, então existe a necessidade de gerar os componentes com baixo acoplamento inter-componente. O conjunto de métricas proposto em (VITHARANA, *et al.*, 2004) é formado por: COUPL (Acoplamento), COHES (Coesão), NCOMP (Número de Componentes), CSIZE (Tamanho do Componente), COMPL (Complexidade). Este trabalho possui um conjunto de métricas com propósitos similares aos propósitos do conjunto de métricas desta dissertação, como as métricas para calcular acoplamento e coesão.

Os trabalhos apresentados possuem uma característica em comum, todos eles aplicam métricas mensuráveis a partir de um projeto OO no contexto do DBC. Além disto, grande parte destas métricas é aplicada no nível de modelo e independente de tecnologia. Algumas métricas destes trabalhos diferem pela fórmula aplicada para extrair a métrica, enquanto outros utilizam as mesmas métricas base. Os trabalhos sobre métricas foram pesquisados visando o apoio na formação do conjunto de métricas sugerido nesta dissertação.

2.4 Reengenharia de Software

Os sistemas são modificados por diversos motivos, como: correção de erros, migração para novas plataformas, ajustes para mudança de *hardware* ou sistema operacional, bem como adição de novas funcionalidades, entre outras motivações. A falta de planejamento e metodologia para esta modificação pode gerar algumas conseqüências como: o conhecimento sobre o sistema se deteriora, degradação da arquitetura e das estruturas originais do sistema, a documentação torna-se freqüentemente perdida e obsoleta. Deste modo, quando o sistema é difícil de ser mantido, mas é de grande utilidade, deve-se reconstruí-lo e, para isto, pode-se utilizar a reengenharia, cuja definição clássica é: “reengenharia é o exame e alteração de um sistema para reconstituí-lo em uma nova forma e a implementação desta nova forma.” (CHIKOFSKY e CROSS, 1990; IEEE, 2004),

PREMERLANI e BLAHA (1994) citam que o objetivo da reengenharia é reutilizar os esforços passados, objetivando reduzir custos de manutenção e melhoria na flexibilidade do software.

Segundo PRESSMAN (2006), a reengenharia de software examina sistemas de informação e aplicações com o objetivo de reestruturá-los ou reconstruí-los, de modo

que apresentem mais qualidade.

Para SOMMERVILLE (1995), a reengenharia de software é descrita como a reorganização e modificação de sistemas de software existentes, parcial ou totalmente, para torná-los mais manuteníveis.

Considerando-se as definições apresentadas, verifica-se que a reengenharia tem como ponto de partida um sistema pronto, diferenciando-se, dessa forma, de um novo desenvolvimento que se inicia a partir da especificação do sistema que ainda será construído.

Além destas definições da reengenharia, outros autores (CHIKOFSKY e CROSS, 1990; WARDEN, 1992; PRESSMAN, 2006) apresentam conceitos sobre as etapas da reengenharia. De acordo com WARDEN (1992), a reengenharia pode ser dividida em duas etapas principais: a engenharia reversa e a engenharia progressiva; onde cada uma das etapas pode ser dividida em várias outras atividades. Para este autor, a extração automática da descrição de uma aplicação e sua implementação em outra linguagem não são considerados reengenharia, mas sim uma tradução de código.

A seguir, é apresentada uma definição clássica e mais detalhada das etapas da reengenharia conforme CHIKOFSKY e CROSS (1990). Segundo estes autores, “a reengenharia geralmente inclui alguma forma de engenharia reversa (para alcançar uma descrição mais abstrata do software), seguida por alguma forma de reestruturação ou engenharia progressiva”.

A Figura 2.2 mostra as mudanças de nível de abstração realizadas durante um processo genérico de reengenharia e estas mudanças vêm ao encontro da definição de reengenharia apresentada por CHIKOFSKY e CROSS (1990).

CHIKOFSKY e CROSS (1990) afirmam que a engenharia reversa é utilizada na reengenharia para alcançar uma descrição mais abstrata do sistema e compreendê-lo melhor, visto que os sistemas legados não costumam possuir documentação atualizada. Esta mudança de nível de abstração é representada na Figura 2.2, com a mudança do nível de implementação para o nível de projeto. Estes autores ainda definem a engenharia reversa como sendo o processo de analisar um sistema para identificar os componentes do sistema e seus inter-relacionamentos, e criar representações do sistema em outra forma ou em nível de abstração mais alto.

Como mencionado anteriormente, a reengenharia pode envolver alguma forma de reestruturação, visando à melhoria da estrutura do sistema sem alterar seu comportamento.

A reestruturação é a transformação de uma representação para outra, no mesmo nível de abstração, enquanto o comportamento externo do sistema é preservado (CHIKOFSKY e CROSS, 1990). Como pode ser observado na Figura 2.2, as reestruturações não geram mudanças no nível de abstração. O objetivo da reestruturação pode ser, por exemplo, melhorar alguns atributos de qualidade do projeto recuperado pela engenharia reversa.

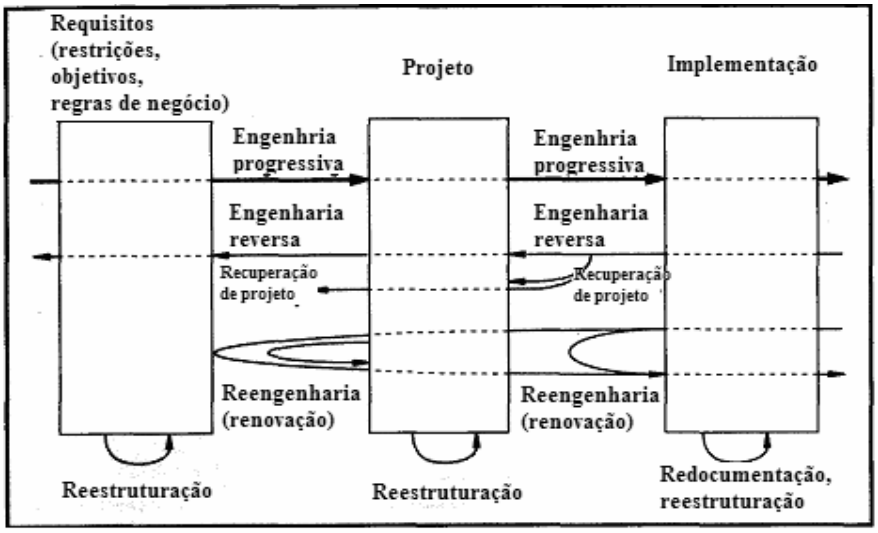


Figura 2.2: Mudanças de nível de abstração geradas por algumas técnicas adaptadas de (CHIKOFSKY e CROSS, 1990).

A última etapa da reengenharia é a engenharia progressiva, que é o processo tradicional de se mover a partir de abstrações e lógica de alto nível, projetos independentes de implementação, para a implementação física de um sistema (CHIKOFSKY e CROSS, 1990). Nesta etapa, pode ocorrer a mudança efetiva de paradigma a partir do software reestruturado.

2.4.1 Abordagens de Reengenharia de Software para Componentes

Como descrito no Capítulo 1, embora os sistemas de software orientados a objetos (OO) sejam construídos para ser manuteníveis, ao longo do tempo eles se degradam tanto quanto qualquer sistema de software, tornando-se legados (NIERSTRASZ, *et al.*, 2005). Apesar do paradigma OO considerar a combinação de funções e dados relacionados em uma única unidade, a classe, visando à construção de software mais manutenível, a facilidade de manter o sistema pode diminuir à medida que o software sofre manutenções, levando, em geral, à degradação da sua estrutura.

No Capítulo 1, as necessidades de contribuições à reengenharia de software para componentes foram descritas resumidamente. Nesta Seção, os trabalhos existentes, na literatura, são detalhados a fim de se caracterizar estas necessidades de contribuições.

Entre os trabalhos referentes a reengenharia com foco na obtenção de componentes, encontrados na literatura, encontra-se o trabalho de LEE *et al*. (2001). Estes autores definem um método para identificar componentes com métricas de coesão e acoplamento. Neste método, o engenheiro de software deve primeiramente definir a arquitetura do sistema, visto que a identificação do componente será realizada para parte do sistema (subsistema). Ao organizar a arquitetura, definindo subsistemas, o esforço necessário para encontrar os componentes será minimizado devido ao menor número de classes a ser considerado. Após definir a arquitetura, deve-se organizar as classes e, para isto, deve-se primeiramente classificar as funcionalidades do subsistema alvo (i.e. casos de uso) em funcionalidade chave, funcionalidade subsidiária ou funcionalidade opcional. Após classificar os casos de uso, classificam-se as classes participantes destes casos de uso em cinco classificações: entidades chave ou controladores de caso de uso com bastante lógica de negócio; entidades subsidiárias (utilizadas a partir de entidades chave); controladores de caso de uso com pouca lógica de negócio; classes de fronteira; classes com menor importância em comparação com as classes chave. Cada classificação tem um valor (peso), com isto, deve-se criar uma tabela com a prioridade das classes por caso de uso.

Após encontrar as classes consideradas como chave, é feito um mapeamento do acoplamento entre as classes, considerando o acoplamento criado pelos relacionamentos entre estas classes. O próximo passo consiste em mapear os casos de uso que farão parte de um mesmo componente e aqueles que não serão mapeados para componentes. Com os dados estabelecidos, utiliza-se um algoritmo de clusterização para gerar automaticamente os candidatos a componentes (agrupamentos das classes do subsistema). Apesar das sugestões de componentes, neste trabalho, não é proposta a forma de geração das interfaces destes componentes.

Por exemplo, em um sistema bancário, escolhe-se um subsistema para extrair os componentes. Selecionando o subsistema de depósito bancário, define-se os casos de uso que serão tratados como, por exemplo, “processar dados financeiros” e “realizar evento de negócio”. A partir destes casos de uso, define-se as classes que são necessárias para estes casos de uso e o seu peso. O passo seguinte é a definição do acoplamento estático e dinâmico. Analisando estes dados com um algoritmo de

clustering da abordagem gera-se as sugestões de componentes que envolvem cada caso de uso e as classes que ficaram no componente para desempenhar cada funcionalidade.

Em (LEE, *et al.*, 2003), é proposto um processo para a reengenharia de sistemas legados OO para um sistema baseado em componentes, onde os componentes são criados examinando-se os relacionamentos entre as classes. O processo é composto por duas etapas: criação de componentes iniciais, baseando-se nos relacionamentos de composição e hierarquia entre as classes do sistema, e refinamento dos componentes encontrados utilizando métricas de CS (*Connectivity Strength*), coesão e acoplamento.

Na primeira etapa, cada classe, que ainda não está em um componente, gera um componente novo ou é agrupada em um componente existente, se possuir um relacionamento de composição com alguma classe deste componente. As classes que compõem uma hierarquia devem estar unidas em um mesmo componente. Esta união pode ser criada através de um componente novo, se a classe em análise possuir superclasse e subclasses, ou através da movimentação da hierarquia para um componente existente, se a classe em análise possui superclasse e não possuir subclasses. As classes em análise que não foram chamadas por nenhuma outra classe ou componente e que não possuírem subclasses devem ser removidas.

Na última etapa, são analisados os valores das métricas de CS, complexidade e coesão nos componentes com o objetivo de refinar os componentes gerados na primeira etapa. Neste contexto, a métrica de CS indica o quanto um componente está ligado a outro componente. Um componente pode conter tantas classes coesivamente ligadas quanto a sua complexidade desde que não ultrapasse um limite de complexidade, o qual deve ser definido pelo engenheiro. Não fica claro, em (LEE, *et al.*, 2003), como o engenheiro de software deve escolher este limite.

A complexidade do componente é determinada de acordo com as métricas de complexidade propostas em (CHO, *et al.*, 2001) e mencionadas na Seção 2.2.2 . A conectividade dos componentes é determinada para cada par de componentes através de uma fórmula que considera os relacionamentos entre as classes de um dos componentes, as quais se comunicam com classes do outro componente. A coesão é obtida através do somatório do quanto cada classe do componente está acoplada a uma classe do mesmo componente. Este acoplamento é determinado através da análise dos relacionamentos para cada par de classes, onde cada tipo de relacionamento tem um peso específico. Através de um algoritmo de *clustering* proposto em (LEE, *et al.*, 2003), o engenheiro deve agrupar os componentes mais ligados entre si, enquanto não houver

mais possibilidades de agrupamento ou a complexidades dos componentes atingir o limite especificado pelo engenheiro.

As interfaces dos componentes são geradas de acordo com as duas diretrizes a seguir: as interfaces providas são formadas pelos métodos públicos de todas as classes do componente que são chamadas por outras classes ou componentes externos ao componente; e as interfaces requeridas são geradas através da união de todos os métodos presentes em classes de outros componentes que são requeridas.

No trabalho descrito em (PRADO, 2005), que se baseia no trabalho de (FONTANETTE, *et al.*, 2002), a reengenharia é realizada a partir de um software legado procedural para um software baseado em componentes. Neste contexto, o processo é dividido nas atividades: **organizar o código, recuperar o projeto do sistema atual, re-especificar, re-projetar e re-implementar**. Onde a **organização do código legado** é um passo preparatório para facilitar a transformação de um código procedural para OO. O engenheiro de software, com o auxílio de uma ferramenta, organiza o código legado procedimental segundo os princípios da OO, com o objetivo de identificar possíveis classes e relacionamento entre elas (ex.: em sistemas orientados a menu, cada tela de interação sugere uma classe de interface). A etapa **recuperar o projeto do sistema atual** tem como objetivo a representação do sistema atual em alto nível, através de modelos, ou seja, o engenheiro de software parte do código legado organizado e obtém as especificações UML do projeto do código legado. A partir do projeto, realiza-se o **re-projeto**, onde o engenheiro de software deve especificar os componentes, com base no seu conhecimento de técnicas de DBC que, em (PRADO, 2005), decidiu-se trabalhar com o método *Catalysis*, podendo ainda incluir funcionalidades novas. É importante ressaltar que não fica clara a forma como o engenheiro é orientado a respeito do agrupamento de classes ao formar cada componente. Por fim, a **re-implementação** é feita para a linguagem alvo da reengenharia.

Outro trabalho existente na literatura é o trabalho de GANESAN e KNODEL (2005), que objetiva identificar componentes reutilizáveis específicos de domínio. Neste trabalho, a reengenharia é realizada a partir de sistemas de software legados OO para um software baseado em componentes. Inicialmente, são utilizadas as métricas (ex.: NMPUB – número de métodos públicos implementados pela classe, OCAEC – número de vezes em que uma classe é utilizada como atributo em outra classe, CALLS – IN – número de vezes em que os métodos de uma classe são chamados em outras classes,

etc.) para identificar as classes mais reutilizadas no sistema alvo. A partir desta seleção, com o apoio do especialista do domínio é verificada a utilidade destas classes no domínio daquele software, as classes escolhidas pelo especialista irão compor componentes. Esta abordagem é composta por dez passos.

No **passo um**, é formulado o objetivo da reengenharia, ou seja, é especificado qual tipo de componente e são especificadas as funcionalidades que se deseja obter. No **passo dois**, é obtido o modelo de projeto atual e são coletadas as métricas sobre as classes do projeto. No **passo três**, o engenheiro determina os valores mínimos e máximos para as métricas. Como os autores determinam estes limites em seu trabalho, eles sugerem que o engenheiro utilize, na primeira, o cálculo da média dos valores das métricas. No **passo quatro**, o engenheiro seleciona as classes que atendem os valores determinados para as métricas no passo anterior.

No **passo cinco**, o engenheiro revisa as classes candidatas, podendo retornar ao passo três e rever os limites das métricas e ainda considerar as especificações feitas no passo um, caso considere que a quantidade de classes candidatas é muito grande ou muito pequena. No **passo seis**, um engenheiro especialista do domínio da aplicação revisa as classes que passaram do passo cinco, classificando-as de acordo com a sua utilidade no domínio. No **passo sete**, é realizado um novo filtro, onde o engenheiro tenta desconsiderar algumas classes que não são especificas do domínio com base na classificação feita no passo anterior. No **passo oito**, é feita a especificação do componente propriamente e o engenheiro trabalha junto aos especialistas do domínio para formar os componentes a partir das classes chave, mas não são propostos critérios para o agrupamento destas classes. No **passo nove**, são geradas as interfaces através da análise das dependências das classes escolhidas com relação ao resto do sistema, de acordo com a proposta de análise de interface descrita em (KNODEL, 2004). Neste trabalho, as interfaces requeridas são geradas analisando-se as dependências das classes do componente para as classes externas ao componente; enquanto as interfaces providas são geradas a partir das dependências de classes externas para as classes destes componentes, podendo ser consideradas somente as classes que estão sendo usadas ou todas as que tiverem operações públicas. Por fim, no **passo dez**, o engenheiro decide onde reutilizar os componentes, baseando-se nas interfaces obtidas no passo nove.

WASHIZAKI e FUKAZAWA (2005) propõem uma abordagem de refatoração para extração automática de componentes JavaBeans a partir de programas desenvolvidos em Java. O primeiro passo da refatoração consiste em definir um CRG

(*Class Relation Graph*). Este CRG é obtido através da análise estática das dependências entre as classes/interfaces Java. O segundo passo consiste em detectar os clusters possíveis através de um algoritmo de cluster. Este cluster é um candidato a componente e não possui dependências para elementos fora deste cluster.

O CRG é formado por três elementos: o conjunto das classes/interfaces, o conjunto de números que são utilizados para identificar as arestas e, o conjunto de arestas propriamente. Na Figura 2.3, pode-se observar o resultado gráfico de um CRG, estando representados todos os elementos citados anteriormente. O exemplo da Figura 2.3 está relacionado a uma implementação do padrão de projeto *Prototype* (GAMMA, *et al.*, 2000) para a construção de labirintos (*maze*) com cômodos (*room*) e portas (*doors*) de um cômodo para o outro. É importante ressaltar que o conjunto de arestas é determinado pela análise estática das dependências entre as classes/interfaces. Neste contexto, uma aresta é gerada para: cada relacionamento de hierarquia entre classes, classes e interfaces ou entre interfaces; para cada instanciação de classe (diferenciando a utilização de construtores default, que são os construtores sem argumento, e os construtores com algum argumento); e, por fim, para cada referência entre classes, classes e interfaces ou entre interfaces (considera-se referências feitas através de variáveis, campos, parâmetros, acesso a métodos ou campos, inclusive classes implementadas internamente, as quais também são denominadas *InnerClasses*).

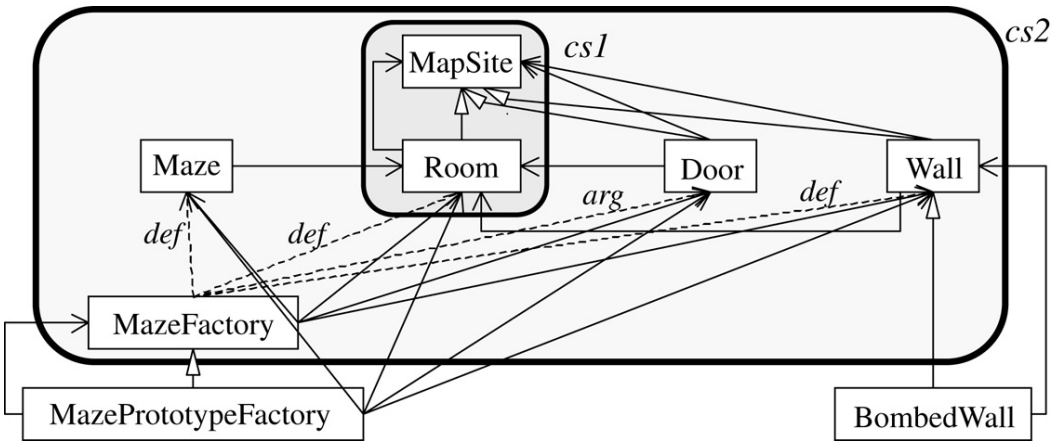


Figura 2.3: Exemplo de CRG para um sistema hipotético, retirado de (WASHIZAKI e FUKAZAWA, 2005).

Para detectar os clusters candidatos a componentes, é realizada uma análise sobre o CRG, considerando as classes/interfaces que são acessíveis a partir de outras classes/interfaces. Por exemplo, na Figura 2.3, encontra-se o destaque cs1, onde dois elementos (*MapSite* e *Room*) são acessíveis devido à dependência do elemento Room

para o elemento *MapSite*. O algoritmo utilizado para detectar os clusters utiliza uma série de princípios, para que os clusters detectados atendam aos princípios de um componente reutilizável, como descrito por WASHIZAKI e FUKAZAWA (2005). Ao final do algoritmo de detecção aplicado no exemplo da Figura 2.3, foram encontrados dois *clusters* representados por cs1 e cs2. O componente resultante da refatoração inclui a nova interface para acessar o componente, a classe que realiza esta interface, representando uma fachada, e as modificações necessárias para criar um componente *JavaBean*, de modo que nenhuma classe acesse uma classe interna ao componente, além de sua interface e sua classe fachada, e nenhuma classe do componente necessite acessar uma classe externa. WASHIZAKI e FUKAZAWA (2005) afirmam que os componentes resultantes não são sempre componentes reutilizáveis semanticamente.

WANG *et al.* (2006) definem uma estratégia de reengenharia de sistemas legados desenvolvidos em C++ para J2EE. Esta estratégia é composta por quatro passos: tradução do código C++ para código Java; extração de componentes utilizando técnicas de clusterização; a modelagem das interfaces dos componentes; e partição dos componentes no ambiente distribuído J2EE. O primeiro passo transformará o código em C++ para a linguagem Java utilizando um *framework* de conversão proposto em (WANG, *et al.*, 2006), Após a conversão automática, ainda pode ser necessário algum esforço de modificação manual, mas a versão inicial de código Java gerada possui alguuns flags que guiam o esforço manual (ex.: [*File*; *test* Java] [*Line*; 120][*Type: multi-inherence*][comentários de modificação]). Neste exemplo, o ajuste deverá ocorrer no arquivo test.java, na linha 120 seguindo as orientações pertinentes ao tipo.

No passo de extração de componentes, é aplicado um algoritmo de *clustering*, onde inicialmente cada entidade do código Java (ex.: classes e interfaces) são distribuídas em componentes distintos. Após a divisão inicial, todas as superclasses e interfaces são re-organizadas para os componentes que possuem suas subclasses ou classes realizadoras. Nos casos em que há muitas subclasses, é feita uma cópia da superclasse para cada componente que possui uma subclasse. Por fim, o algoritmo de extração utiliza uma métrica de similaridade de componentes para verificar as possibilidades de junção dos componentes definidos até o momento.

Para a modelagem de interfaces, WANG *et al.* (2006) apresentam duas heurísticas. A primeira heurística determina a criação de uma fachada para troca de mensagens entre os componentes através de *Message Facade* ou *Session Facade*. A segunda heurística diz respeito aos dados persistentes que devem ser componentes de

entidade do tipo *Bean-Managed-Persistent* (BMP) ou *Container-Managed-Persistent* (CMP). Por fim, os componentes devem ser alocados no ambiente distribuído J2EE. Para isto, os autores propõem algumas formas de distribuição e analisam os benefícios e desvantagens destas formas. Por exemplo, existe a distribuição geral, onde um ou mais componentes podem ser alocados no mesmo local, considerando-se a maximização da localidade do dado e a minimização da comunicação remota entre componentes. Ao final, toda a aplicação está componentizada.

CHARDIGNY *et al.* (2008) propõem a abordagem ROMANTIC. Esta abordagem objetiva extrair uma arquitetura de componentes a partir de software OO existente. A abordagem pode ser aplicada em duas etapas: definição de um modelo de correspondência entre os elementos do código e os conceitos arquiteturais; extração dos elementos arquiteturais a partir do modelo de correspondência definido anteriormente.

Na primeira etapa, a definição do modelo de correspondência deve considerar os elementos de código (ex.: classes, interfaces, pacotes, etc.) e seus correspondentes em elementos arquiteturais (ex.: componentes, conectores, interfaces, etc.). Neste modelo, as classes do sistema são particionadas e cada partição representa um componente. As interfaces destes componentes são formadas pelas classes que têm algum relacionamento com classes externas aos componentes. Para simplificação, os autores consideram que os conectores são as ligações (relacionamentos) existentes entre os componentes. Neste contexto, a arquitetura é formada pelo modelo particionado.

Na segunda e última etapa, o modelo de correspondência obtido na etapa anterior é utilizado para obter uma arquitetura relevante. Os autores definem que uma arquitetura relevante deve seguir quatro princípios: deve ser semanticamente correta; deve ter boas propriedades de qualidade; deve respeitar precisamente a recomendação do arquiteto e, o quanto for possível, respeitar especificações e restrições definidas em documentações; por fim, deve ser capaz de ser adaptada à especificidade de uma arquitetura de hardware. Os autores acreditam que o princípio relativo à semântica seja o mais relevante entre os outros e, por isso, definiram uma função para avaliar a semântica da arquitetura. Esta função é utilizada em um algoritmo de *clustering* aplicado sobre o modelo de correspondência, para obter uma melhor arquitetura sob o aspecto semântico.

Ao final das duas etapas, toda a aplicação será especificada em componentes. Por não existir um ferramental de apoio, pode exigir um esforço maior por parte do engenheiro de software para utilizar a abordagem.

WANG *et al*. (2008) apresentam uma nova proposta para identificação de componentes baseando-se em métricas de WCS (*Weighted Connectivity Strength*) em relação aos trabalhos que utilizam métricas de CS (*Connectivity Strength*) como (LEE, *et al.*, 2003). Esta métrica baseia-se no quanto as classes estão acopladas umas as outras, analisando os tipos de dependência e atribuindo pesos distintos. A identificação dos componentes é realizada em duas etapas: análise da relação de composição e hierarquia entre classes, para obter um conjunto inicial de componentes; e avaliação do valor de WCS entre os componentes para agrupar os componentes que possuírem o maior valor de WCS em um novo componente até que os valores de WCS estejam abaixo de um valor mínimo especificado.

Na primeira etapa, os autores indicam que cada classe, que ainda não está em um componente, gera um componente novo ou é agrupada em um componente existente, se possuir um relacionamento de composição com alguma classe deste componente. As classes que compõem uma hierarquia devem estar unidas em um mesmo componente. Esta união pode ser criada através de um componente novo, se existir mais de uma subclasse em componentes distintos, ou através da movimentação da hierarquia para um componente existente, se as subclasses já estiverem agrupadas em um único componente.

Na última etapa, o engenheiro de software deve utilizar o seguinte algoritmo para aperfeiçoar os componentes gerados na etapa inicial: considere os componente que tenham o valor de WCS, calculado através da soma de todos os valores de WCS por pares de classes no componente, abaixo de um limite especificado, e calcule o valor de WCS entre cada componente; enquanto o valor de WCS, entre os componentes, exceder o limite especificado, encontre o par de componentes com o maior valor de WCS; Se o valor já estiver abaixo do limite estipulado, examine os pares de componentes que não estavam no conjunto inicial; Se encontrar um par com o maior valor de WCS acima do limite especificado, agrupe o par formando um novo componente; repita o algoritmo, até que todos os pares tenham sido examinados e os valores de WCS estejam abaixo do limite especificado para WCS entre componentes.

Ao final, toda a aplicação estará dividida em componentes, mas o trabalho proposto em (WANG, *et al.*, 2008) não deixa clara a forma como as interfaces destes componentes podem ser geradas.

2.5 Considerações Finais

Este capítulo apresentou os conceitos necessários à compreensão da abordagem proposta nesta dissertação. As definições sobre o DBC visam contextualizar o resultado que se espera ao final da aplicação da abordagem e a forma como os componentes são especificados. As definições sobre reengenharia visam mostrar a estrutura básica das abordagens de reengenharia, bem como suas principais motivações. Além disto, são apresentadas algumas abordagens de reengenharia de software para componentes.

Nesta seção, as abordagens de reengenharia para componentes são comparadas de acordo com alguns critérios estabelecidos para esta dissertação. O primeiro critério diz respeito ao paradigma do sistema alvo da reengenharia, porque partindo-se de sistemas OO tem-se a vantagem de reutilizar padrões de projeto e outras soluções projetadas para sistemas OO. O segundo critério trata a generalidade da abordagem, ou seja, a capacidade da abordagem poder ser aplicada a sistemas que foram desenvolvidos em diferentes linguagens de programação, uma vez que abordagens muito específicas têm seu escopo de aplicação limitado.

O apoio à formação de componentes também é avaliado, visto que um projeto de sistema legado pode exigir um grande esforço do engenheiro para definir estes componentes. O terceiro critério avaliado é o apoio à formação de interfaces, fornecendo diretrizes para guiar o engenheiro de software nesta atividade. A fim de avaliar o resultado obtido pelas abordagens, é verificado o seu objetivo quanto à reengenharia da aplicação inteira ou apenas a extração de componentes (partes da aplicação). O apoio ferramental e a possibilidade de integração a um ambiente de desenvolvimento de software (ADS) também são investigados, a fim de reduzir o esforço da reengenharia e prover meios que facilitem a reutilização dos artefatos produzidos em novos desenvolvimentos.

Nesse contexto, as contribuições e os pontos em aberto das abordagens são ressaltados na Tabela 2.3, sendo apresentado em forma de quadro comparativo entre as abordagens pesquisadas.

De acordo com o exposto, Quanto ao apoio à composição de componentes, alguns trabalhos utilizam algoritmos de *clustering* ou apresentam alguns princípios sem medidas precisas para atingi-los. Quanto à formação de interface, as abordagens apresentam pouca orientação para a composição das interfaces e a composição das operações nestas interfaces.

Algumas abordagens estão diretamente relacionadas á uma linguagem de programação específica, limitando a sua aplicabilidade em sistemas alvo que foram implementados em outras linguagens de programação.

Além disto, a falta de ferramental de apoio em algumas abordagens aumenta o

Tabela 2.3: Comparação entre abordagens.

Trabalho	Paradigma de Origem	Generalidade	Apoio à composição de Componentes (agrupamento de classes)	Apoio à composição de interfaces (agrupamento de operações)	Reengenharia de toda a aplicação	Integração a um ADS	Ferramental
LEE *et al.* (2001)	OO	Sim. Aplicável no nível de modelo	Sim. As sugestões de componentes são geradas automaticamente por algoritmo de *clustering*	Não	Não. Apenas extração de componentes	Não	Não
LEE *et al.* (2003)	OO	Sim. Aplicável no nível de modelo	Sim. Através de algoritmos de *clustering*	Sim	Sim	Não	Não
PRADO (2005)	Procedural	Parcial. Para cada sistema alvo, é preciso customizar a ferramenta que apóia a análise do código fonte	Parcial. É indicada a utilização do método *Catalysis*, mas não é indicada uma forma sistemática para agrupar as classes	Parcial. É indicada a utilização do método *Catalysis*, mas não é indicada uma forma sistemática para agrupar as operações	Sim	Parcial. Os autores sugerem duas ferramentas que se comunicam indiretamente através de importação de arquivos.	Sim
GANESAN e KNODEL (2005)	OO	Sim. Aplicável no nível de modelo	Parcial. Apóia a identificação das classes que irão compor os componentes	Sim. É indicada análise de interface proposta por (KNODEL, 2004)	Não. Apenas extração de componentes	Não	Não
WASHIZAKI e FUKAZAWA (2005)	OO em Java	Não. Aplicável somente para sistemas desenvolvidos em Java	Sim. Os componentes são extraídos automaticamente	Sim	Não. Apenas extração de componentes	Não	Sim
WANG *et al.* (2006)	OO em C++	Não. Aplicável somente para sistemas desenvolvidos em C++	Sim. Os componentes são gerados automaticamente através de algoritmo de *clustering*	Sim. Através de duas heurísticas para interfaces em componentes em J2EE	Sim	Não	Parcial. Existe ferramenta para a conversão de código C++ em Java
CHARDIGNY *et al.* (2008)	OO	Sim. Aplicável no nível de modelo	Sim	Não	Sim	Não	Não
WANG *et al.* (2008)	OO	Sim. Aplicável no nível de modelo	Sim. Através de *clustering*	Não	Sim	Não	Não

esforço necessário para aplicá-las, enquanto a falta de um ferramental integrado a um ADS impede que todos os artefatos trabalhados nas abordagens sejam tratados em um único ambiente.

Diante deste contexto, existe a necessidade de abordagens que contribuam nestes pontos que ainda estão deficientes nas abordagens analisadas.

Vale ressaltar que as abordagens estudadas influenciaram a abordagem proposta nesta dissertação. Por exemplo, a utilização de métricas é perceptível em grande parte das abordagens, assim como a utilização dos critérios de coesão e acoplamento para agrupar classes em um determinado componente. As abordagens que utilizam algoritmo de *clustering* mostram-se vulneráveis a projetos ruins, onde o acoplamento entre as classes está mal planejado, pois o engenheiro de software não interfere durante o *clustering*, determinando resultados baseados somente no projeto. Deste modo, a abordagem proposta, nesta dissertação, utiliza métricas para apoiar o engenheiro na tomada de decisão, mas permite que o engenheiro utilize o seu conhecimento de domínio para gerar os agrupamentos de classes candidatos a componentes.

As abordagens que provêem algum suporte para a formação de interfaces analisam as dependências entre as classes/interfaces que compõem os componentes. Este fator influenciou parte da estratégia para a geração de interfaces.

Capítulo 3 ROOSC: Abordagem de Reengenharia de Sistemas Orientados a Objetos para Componentes Apoiada por Métricas

3.1 Introdução

Como apresentado no Capítulo 2, apesar das vantagens que a mudança do paradigma de orientação a objetos (OO) para o desenvolvimento baseado em componentes (DBC) pode trazer, essa mudança de paradigma não é trivial. Faz-se necessária uma abordagem de reengenharia para apoiar a realização desta migração. Como afirmam LEMOS *et al.* (2003), a reengenharia ainda precisa utilizar diretrizes de qualidade para que o seu resultado seja confiável. As abordagens existentes na literatura, discutidas no Capítulo 2, apóiam a reengenharia para componentes em diferentes contextos e formatos, mas ainda há a necessidade de suprir algumas deficiências.

Neste contexto, a abordagem proposta, ROOSC (*Reengineering Object Oriented Software to Components*), de reengenharia de *software* OO para componentes (MOURA, *et al.*, 2008c), traz algumas contribuições como a reengenharia da aplicação como um todo, partindo-se de um sistema OO que, como visto no capítulo 2, pode ser utilizado no desenvolver de um componente, eliminando quaisquer transformações adicionais. A abordagem é realizada no nível arquitetural e permite que os sistemas alvo tenham sido desenvolvidos em diferentes linguagens OO. Entre as contribuições, encontram-se o apoio à formação dos componentes com o seu projeto interno e o apoio à formação de interfaces, permitindo que o engenheiro de *software* seja guiado de acordo com princípios de projeto, mas que possa utilizar também o seu conhecimento do domínio para obter componentes mais coesos e semânticos.

A abordagem ROSSC é composta por duas etapas: a reestruturação dos modelos de pacotes e classes, que visa obter agrupamentos de classes (i.e. pacotes) candidatos a componentes, de modo que atendam aos princípios de DBC; e a geração dos componentes propriamente, que visa obter um modelo de componentes com suas interfaces, a partir dos agrupamentos de classes (i.e. pacotes) reestruturados. A Figura 3.1 ilustra uma visão geral da abordagem proposta para a reengenharia de *software* OO

para componentes, mostrando seus principais artefatos de entrada, etapas e produto resultantes.

Os primeiros elementos de entrada para a abordagem são os agrupamentos de classes (i.e. pacotes), suas classes e relacionamentos, os quais devem ser previamente obtidos através de alguma abordagem de engenharia reversa (BOJIC e VELASEVIC, 2000; DING e MEDVIDOVIC, 2001; VERONESE e NETTO, 2001; VASCONCELOS, 2007). Deste modo, as decisões de projeto e implementação são re-aproveitadas e garante-se, assim, que o modelo esteja atualizado e reflita os acoplamentos reais existentes entre as classes do sistema. O modelo com estes elementos é representado na Figura 3.1, onde é mostrada ainda a sua utilização na primeira etapa da abordagem (i.e. reestruturação).

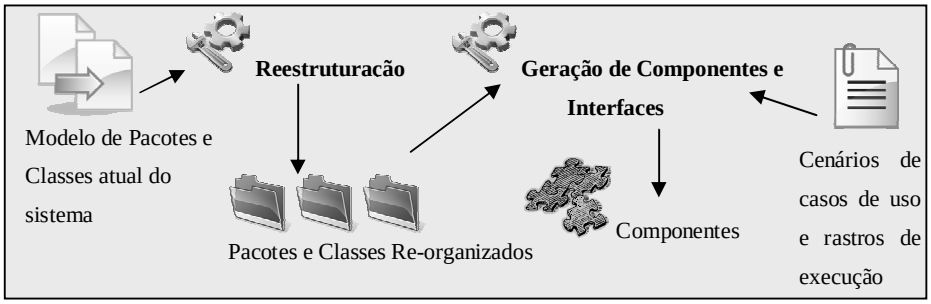


Figura 3.1: Visão geral da abordagem ROOSC.

A primeira etapa (i.e. reestruturação) da abordagem, como mostra a Figura 3.1, tem como resultado um conjunto de pacotes e classes re-organizados, ou seja, os agrupamentos de classes (i.e. pacotes) estão re-organizados e podem dar origem aos componentes na próxima etapa da abordagem. A estratégia utilizada para re-organizar estes agrupamentos de classes baseia-se na aplicação de um conjunto de reestruturações guiada por um conjunto de métricas (MOURA, *et al.*, 2008a; MOURA, *et al.*, 2008d). Esta estratégia será detalhada na Seção 3.2.

Estes agrupamentos reestruturados farão parte dos modelos de entrada para a segunda etapa da abordagem. É possível visualizar na Figura 3.1 que, juntamente com estes agrupamentos, será necessário utilizar cenários de casos de uso do sistema alvo. Estes cenários devem ser recuperados através de alguma abordagem de engenharia reversa que permita obter os rastros de execução destes cenários (RICHNER e DUCASSE, 1999; DING e MEDVIDOVIC, 2001; VASCONCELOS, 2007), ou seja, obter as chamadas de métodos necessárias para realizar aquele cenário. Os detalhes sobre como estes rastros são obtidos serão discutidos na Seção 3.3.

Como mostra a Figura 3.1, a segunda etapa da abordagem utiliza os pacotes

reestruturados e os cenários de caso de uso para gerar os componentes e interfaces propriamente. A estratégia para a geração de componentes e interfaces utiliza um conjunto de heurísticas durante a geração e um conjunto de métricas para refinar os componentes e interfaces gerados. Esta estratégia será apresentada na Seção 3.3.

Como citado anteriormente, a abordagem proposta é realizada no nível de modelo. O engenheiro de *software* pode utilizar uma abordagem de engenharia reversa específica para a linguagem de programação do sistema alvo que gere um modelo do projeto deste sistema. O resultado da abordagem também é no nível de modelo, para que a especificação dos componentes possa ser re-gerada em uma tecnologia de componentes (ex: .NET, EJB etc.) à escolha do engenheiro de *software*, por meio de ferramentas de geração de código.

Vale ressaltar que a abordagem é semi-automatizada e requer um engenheiro com conhecimento do domínio para guiar o processo, pois as duas estratégias mencionadas anteriormente (i.e. estratégia de reestruturação e estratégia de geração de componentes e interfaces) têm ênfase na qualidade do projeto interno dos componentes sob aspectos técnicos. O conhecimento do domínio complementará o apoio à tomada de decisão e possibilitará a geração de componentes com funcionalidades mais coerentes e semânticas no contexto deste domínio.

As seções a seguir apresentam a descrição detalhada das estratégias de reestruturação e geração de componentes e interfaces. Um exemplo envolvendo um sistema de controle de locação de vídeos hipotético é utilizado para ilustrar a forma como as métricas e reestruturações são combinadas. Considerando este sistema hipotético, a entrada para a abordagem é o modelo de projeto do sistema, onde são encontradas as representações de suas classes, pacotes e seus respectivos relacionamentos.

Visando à compreensão da abordagem ROOSC de reengenharia de *software* orientado a objetos (OO) para componentes, este capítulo descreve as suas estratégias, estando, deste modo, organizado da seguinte forma: a Seção 3.2 descreve a estratégia de reestruturação; a Seção 3.3 descreve a estratégia para a geração de componentes e interfaces; a Seção 3.4 apresenta as considerações finais sobre a abordagem ROOSC.

3.2 Reestruturação

A reestruturação do modelo de classes visa que os agrupamentos de classes, candidatos a componentes, representem “bons projetos” para componentes. Na

literatura, há trabalhos que enfatizam princípios de projeto OO de componentes (VITHARANA, *et al.*, 2004; GANESAN e KNODEL, 2005; GONÇALVES, *et al.*, 2006). Em síntese, os princípios são (Capítulo 2): não deve existir relacionamento de herança entre componentes distintos, pois um componente não deve conhecer a implementação do outro; os componentes devem prover uma funcionalidade específica; classes generalizadas são boas candidatas a comporem componentes, pois permitem maior adaptação e extensão em novos sistemas; a comunicação inter-componentes, feita via suas interfaces, é muito cara em termos de tempo e recursos de plataforma e, por isso, deve-se diminuir o acoplamento entre componentes, a fim de minimizar o tráfego na rede.

A etapa de reestruturação é tratada através de uma estratégia que, como mostra a Figura 3.2, é realizada a partir do modelo de pacotes e classes, que representa o projeto detalhado atual do sistema. Este modelo é recuperado através de alguma abordagem de engenharia reversa, como mencionado anteriormente, e ao final da aplicação de toda a abordagem, este modelo estará sob a forma de componentes. O engenheiro inicia a reestruturação destes modelos, solicitando a coleta das métricas. Os resultados desta coleta são exibidos junto aos valores de referência e interpretação das métricas, como descrito na Figura 3.2. A cada métrica está associado um conjunto de reestruturações que são sugeridas, comparando-se os valores obtidos da métrica aos seus valores de referência (obtidos na literatura, quando disponíveis ou obtidos de uma *baseline* que pode ser criada na organização) ou à média dos valores da métrica extraídos para o projeto. Na Figura 3.2, após o engenheiro selecionar as reestruturações, o modelo é atualizado para refletir a reestruturações aplicadas.

A partir deste momento, um novo ciclo é iniciado e as métricas são coletadas novamente sobre o modelo atualizado, assim como novas sugestões de reestruturações são geradas. Dessa forma, as sugestões de reestruturações estão sempre consistentes com o estado atual do modelo.

Como mostra a Figura 3.2, o engenheiro de *software* após o primeiro ciclo pode fazer uma verificação do modelo reestruturado, com o objetivo de avaliar a preservação das suas funcionalidades e melhoria da qualidade. Os detalhes desta verificação serão apresentados na Seção 3.2.3. Após a verificação, o engenheiro de *software* pode optar por continuar a reestruturar, baseando-se nos princípios de DBC, nos valores das métricas e em seu conhecimento do domínio, para que o modelo tenha melhores agrupamentos. Ele ainda pode optar por gerar componentes a partir de cada

agrupamento reestruturado, caso já tenha conseguido bons agrupamentos, com bons valores de métricas e semanticamente organizados, em função da verificação realizada.

Como mencionado anteriormente, a abordagem requer um engenheiro com conhecimento do domínio para guiar o processo, apoiando a seleção de reestruturações e a verificação do modelo.

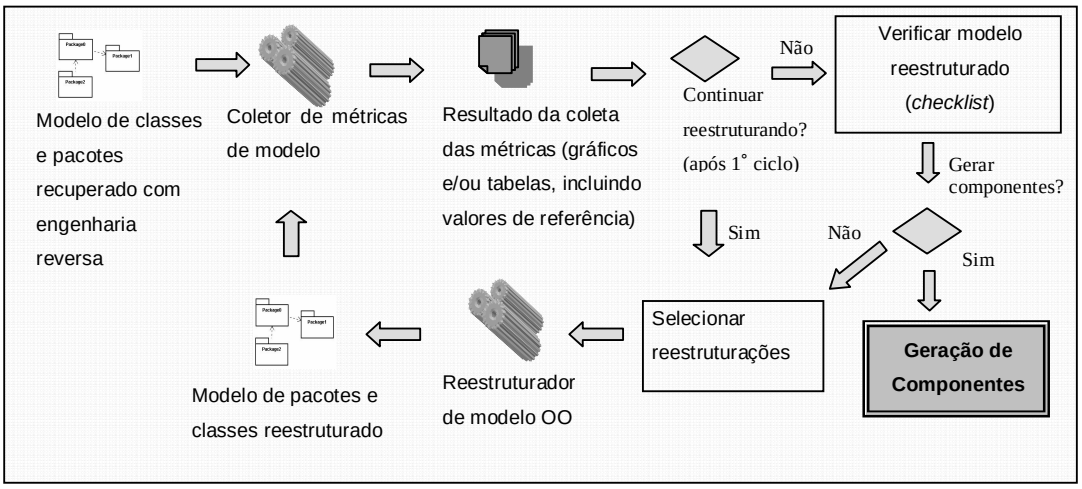


Figura 3.2: Estratégia proposta de reestruturação de modelos OO estáticos.

Na Figura 3.3, os pacotes do sistema hipotético de locação de vídeo e seus relacionamentos são apresentados. As classes contidas em cada pacote são apresentadas na Figura 3.4, através de uma visualização em formato de árvore.

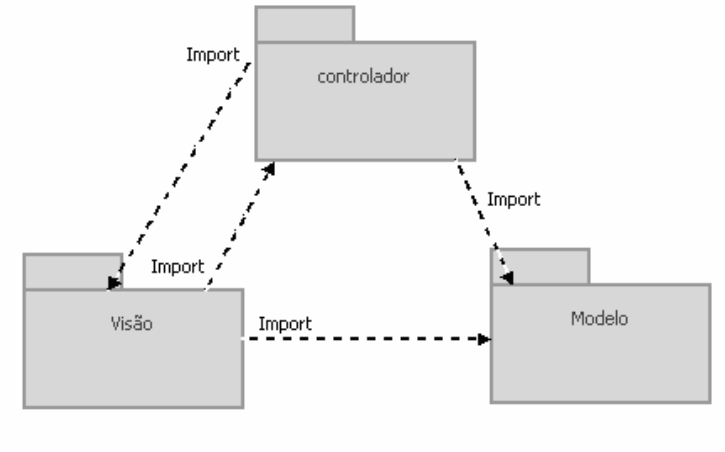


Figura 3.3: Pacotes do sistema de locação de vídeo hipotético.

O pacote visão apresentado, na Figura 3.3, contém classes relacionadas à interface gráfica da aplicação. Estas classes, por sua vez, solicitam as funcionalidades das classes responsáveis pelo controle dos casos de uso, que estão presentes no pacote controlador de acordo com a Figura 3.4. Estes dois pacotes dependem das classes do pacote modelo, uma vez que estas classes representam os conceitos do negócio.

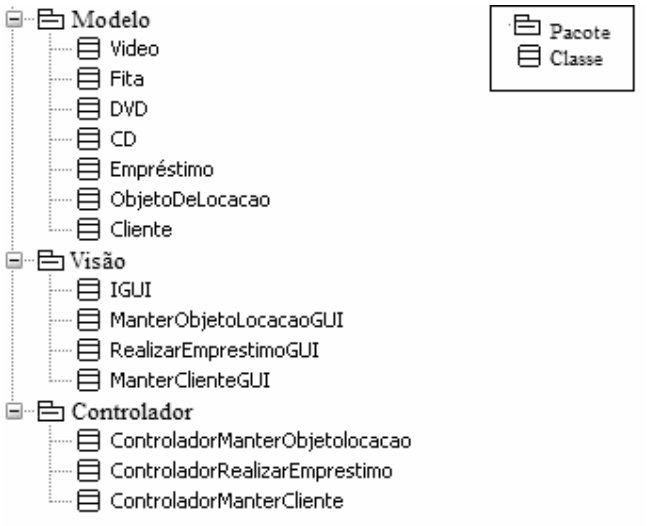


Figura 3.4: Classes do sistema de locação de vídeo hipotético, agrupadas em pacotes.

3.2.1 Métricas

Como apresentado anteriormente, as reestruturações são apoiadas por um conjunto de métricas. Cada métrica foi selecionada ou adaptada com base nos seguintes critérios:

- ser mensurável a partir de um modelo OO;
- poder ser coletada de forma automática;
- auxiliar o cumprimento dos princípios de DBC citados anteriormente.

A seguir, o conjunto inicial de métricas é apresentado com o seguinte *template* (modelo) proposto:

- **Nome** – Nomenclatura da métrica, que inclui a sigla utilizada para referenciar a mesma e o significado das siglas.
- **Nível de granularidade (pacote ou classe)** – O nível em que as métricas são coletadas, ou seja, algumas métricas podem ser medidas no nível de uma classe, enquanto outras são medidas no nível de agrupamento de classes (i.e. pacote). Esta organização facilita a utilização das métricas por parte do engenheiro de *software*.
- **Interpretação / justificativa** – Descreve como o engenheiro de *software* deve interpretar a métrica, justificando a utilização desta métrica OO no contexto do DBC, pois algumas métricas foram adaptadas e outras somente inseridas neste contexto.
- **Descrição** – Descreve textualmente o que será mensurado.

- **Fórmula** – Descreve o que será mensurado através de fórmula, explicando o significado de cada elemento da fórmula.
- **Valor de referência** – Valor que deve ser utilizado para apoiar a análise dos resultados junto à interpretação da métrica. Quando não especificados com base na literatura, estes valores devem ser comparados à média dos valores obtidos durante a coleta ou a alguma linha base (*baseline*) da organização. É apontada também a possibilidade de reconfiguração, ou seja, o engenheiro pode alterar o valor de referência de acordo com a sua experiência ou criação de uma *baseline*.

As características do *template* visam facilitar a compreensão da métrica. Como estas métricas são obtidas a partir de modelos OO, mas visam à obtenção de candidatos a componentes, toda vez que a métrica se refere a um componente, o mesmo representa um pacote. Cada valor de referência de métrica foi obtido na literatura, quando disponível, ou este valor pode ser comparado à média dos valores extraídos para a métrica, havendo ainda a possibilidade deste valor ser determinado em uma *baseline* à medida que a abordagem seja utilizada pela organização.

Alguns valores de referência podem ser reconfigurados à medida que a organização crie uma base de referência (*baseline*). Os valores que não podem ser reconfigurados estão diretamente relacionados aos princípios de DBC e não variam de acordo com a maturidade da organização ao utilizar a abordagem e desenvolver uma *baseline*. Além disto, a compreensão destes valores é apoiada pela interpretação da métrica.

Para a compreensão das métricas, considere que C_i denota um componente num sistema S, onde S= $\{C_1...C_i\}$, i≥1, onde i corresponde ao número total de componentes de S. Cada componente C_i, por sua vez, é composto por um grupo de classes/interfaces, onde um $C_i = \{c_1...c_k\}$, k≥1, sendo k igual ao número total de classes/interfaces de C_i. A seguir, as métricas são apresentadas conforme o *template* mostrado anteriormente. Junto a cada métrica, será mostrado um exemplo de coleta utilizando o sistema de locação de vídeos hipotético.

Número de Filhos por Superclasse para cada Componente (NOCC – *Number Of Children per Component Superclass*). Adaptada de NOC (CHIDAMBER e KEMERER, 1994).

- **Nível de Granularidade**: Classe.

- **Interpretação**: Segundo VITHARANA *et al*. (2004), superclasses não devem possuir subclasses em componentes distintos ao qual ela pertence, pois este relacionamento faz com que um componente conheça detalhes da implementação do outro. Deste modo, uma superclasse não deve possuir subclasses em componentes distintos ao qual ela pertence.
- **Descrição**: Mensura o número de filhos da superclasse em cada componente. Esta métrica é exemplificada na Figura 3.5.
- **Fórmula**: $NOCC_{ki} = SUB(c_k\ C_i)$, $1 \leq k \leq n$, onde: NOCC da superclasse c_k para o componente i é igual ao valor da função SUB para superclasse c_k, que retorna o número de subclasses da superclasse c_k considerando somente as suas subclasses pertencentes ao componente C_i . Sendo n igual ao número total de superclasses do modelo e k varia de 1 a n.
- **Valor de referência**: 0 para componentes distintos ao da superclasse, ou seja, aquele em que a superclasse se encontra (não reconfigurável).

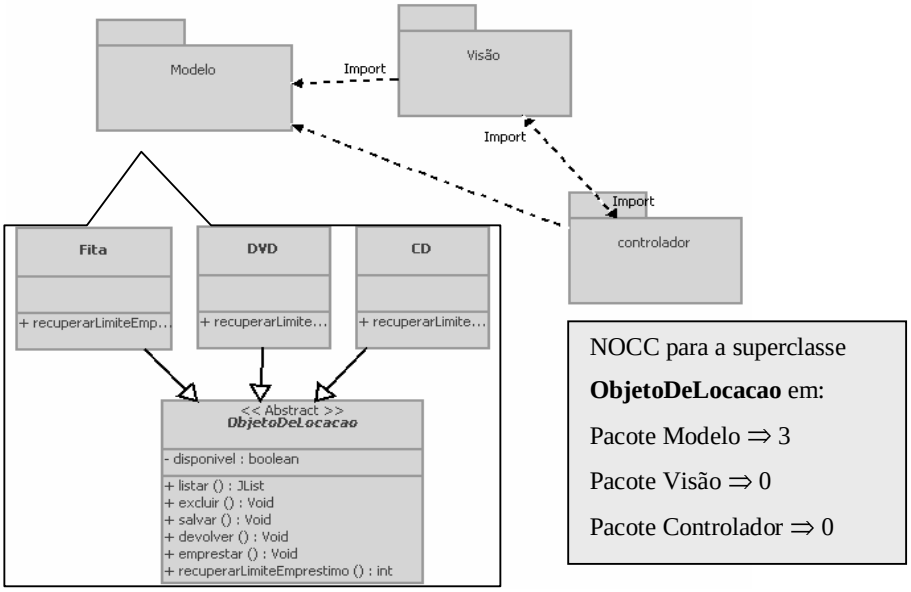


Figura 3.5: Exemplo de resultado da coleta da métrica NOCC no sistema hipotético.

A Figura 3.5 apresenta o exemplo da métrica NOCC para a superclasse ObjetoDeLocacao. As subclasses imediatas desta superclasse estão no pacote modelo, por isso, esta métrica obteve o valor 3 para o pacote modelo e 0 para os demais pacotes.

Número de Implementações de Interfaces por Componente (NOIC – *Number Of Implementation per Component*).

- **Nível de Granularidade**: Classe.

- **Interpretação**: Classes que implementam uma mesma interface estão inseridas em um contexto comum e fornecem os mesmos serviços, de modo que uni-las deixará a funcionalidade em um único componente.
- **Descrição**: Mensura o número de classes que implementam uma determinada interface em cada componente. Esta métrica é exemplificada na Figura 3.6.
- **Fórmula**: $NOIC_{ki} = NOI(c_k\ C_i)$, $c_k \in C_i$, $1 \leq k \leq n$, onde: NOI da interface c_k no componente C_i é igual ao número de classes que implementam a interface c_k considerando somente as classes pertencentes ao componente C_i. Sendo n igual ao número total de superclasses do modelo e k varia de 1 a n.
- **Valor de referência**: 0 para componentes distintos ao da interface, ou seja, àquele em que a interface se encontra (reconfigurável).

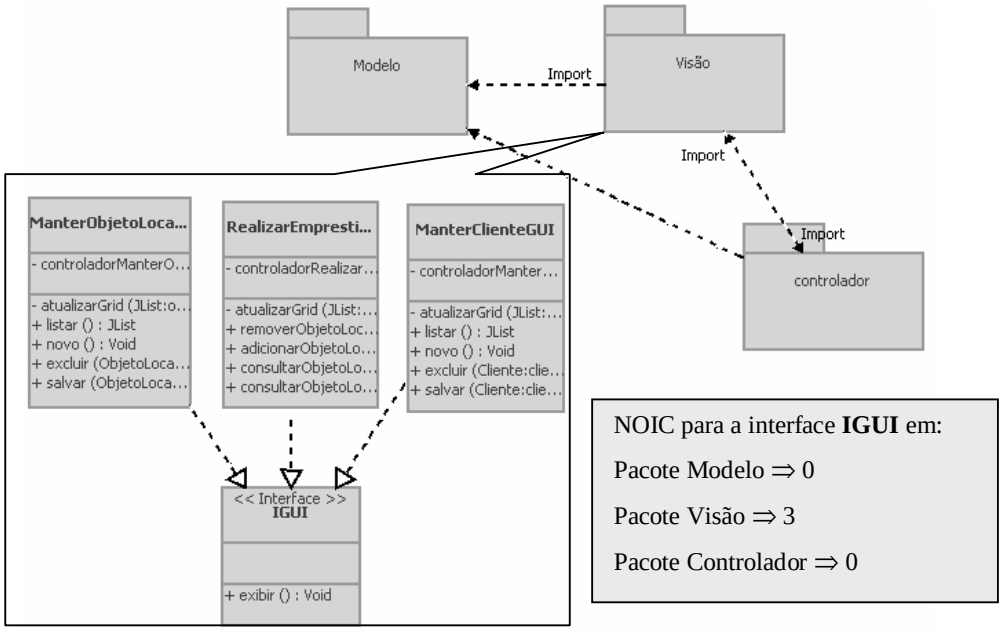


Figura 3.6: Exemplo de resultado da coleta da métrica NOIC no sistema hipotético.

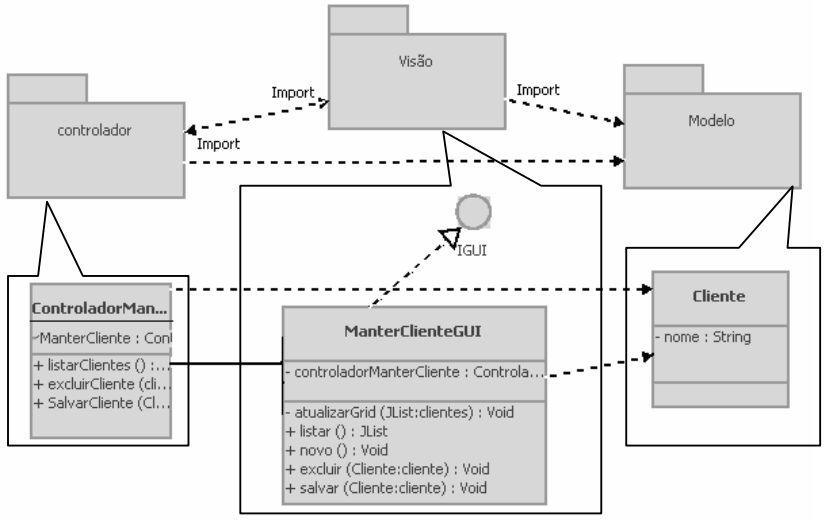
A Figura 3.6 apresenta o exemplo da métrica NOIC para a interface IGUI. As classes que realizam esta interface estão no pacote visão, por isso, esta métrica obteve o valor 3 para o pacote visão e 0 para os demais pacotes.

Acoplamento entre Classes por Componente (CBOC – *Coupling Between Object Classes per Component*). Adaptada de CBO (CHIDAMBER e KEMERER, 1994).

- **Nível de Granularidade**: Classe.
- **Interpretação**: o acoplamento da classe com relação ao componente ao qual ela pertence deve ser igual ou maior do que o acoplamento com relação aos demais

componentes.

- **Descrição**: mensura o valor de CBO (*Coupling Between Object Classes*) de cada classe k para cada componente i. Esta métrica é exemplificada na Figura 3.7.
- **Fórmula**: $CBOC_{ki} = COUPL(c_k, C_i)$, $c_k \in C_i$, $1 \leq k \leq n$, onde: CBOC da classe c_k no componente C_i é igual ao valor da função COUPL para a classe c_k, onde a função COUPL retorna o número de classes que estão acopladas a classe c_k considerando somente seus relacionamentos com as classes que pertencem ao componente C_i. Sendo n igual ao número total de superclasses do modelo e k varia de 1 a n.
- **Valor de referência**: não possui, devendo ser montado um gráfico mostrando o crescimento do acoplamento da classe por componente.



CBOC para a classe **ManterClienteGUI** em:

Pacote Modelo ⇒ 1 Pacote Visão ⇒ 1 Pacote Controlador ⇒ 1

Figura 3.7: Exemplo de resultado da coleta da métrica CBOC no sistema hipotético.

A Figura 3.7 apresenta o exemplo da métrica CBOC para a classe ManterClienteGUI. Esta classe obteve o valor 1 para a métrica CBOC em relação aos pacotes controlador, visão e modelo. Isto significa que esta classe está acoplada a uma outra classe em cada um destes pacotes.

Abstração (A - *Abstraction*). Proposta por (GONÇALVES, *et al.*, 2006).

- **Nível de Granularidade**: Pacote.
- **Interpretação**: um componente deve ser abstrato ou bastante abstrato, sendo mais facilmente adaptado em diferentes contextos de reutilização dentro de um

domínio segundo GONÇALVES *et al.* (2006).

- **Descrição**: razão entre o número de classes abstratas e interfaces e o número total de classes e interfaces de cada componente. Esta métrica é exemplificada na Figura 3.8.
- **Fórmula**: $A_i = AC_i/TC_i$, onde: AC_i é igual ao Total de classes abstratas e interfaces do componente i e, TC_i é igual ao total de classes e interfaces do componente i, onde i varia de 1 ao número total de componentes do sistema S.
- **Valores de referência**: 0 – 0,25: bastante concreto; 0,26 – 0,5: concreto; 0,51 – 0,75: abstrato; 0,76 – 1: bastante abstrato (reconfigurável).

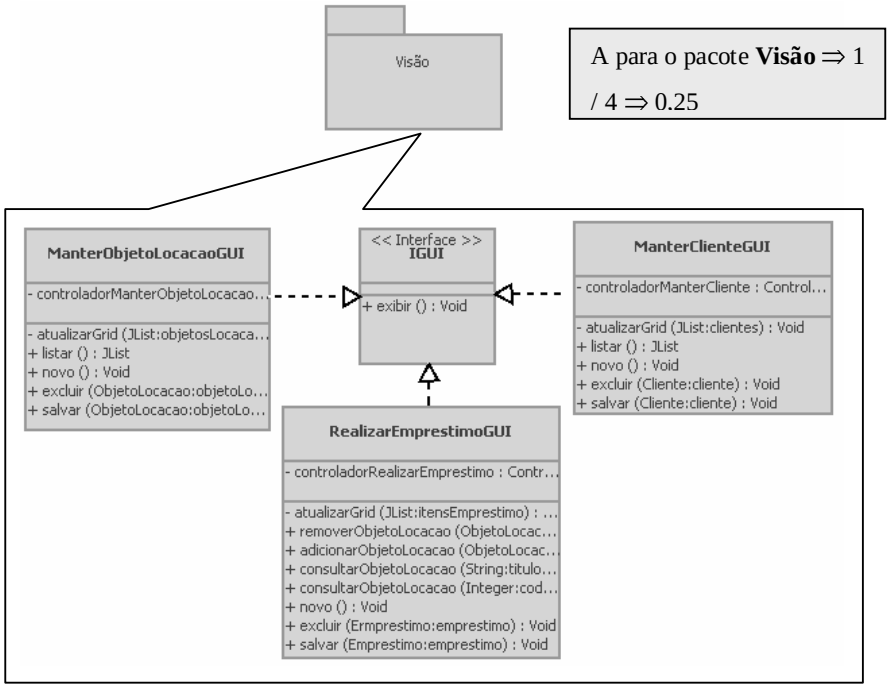


Figura 3.8: Exemplo de resultado da coleta da métrica A no sistema hipotético.

A Figura 3.8 apresenta o exemplo da métrica A para o pacote visão. Para este pacote, obteve-se o valor 0,25 relativo à métrica A, considerando-se o total de classes abstratas e interfaces, que é igual a 1 (i.e. IGUI), sobre o total de classes e interfaces, que é igual a 4 (i.e. IGUI, ManterObjetoDeLocacaoGUI, ManterClienteGUI, RealizarEmprestimoGUI).

Componentes Conectados (ConnComp – *Connected Components*). Proposta por (SDMETRICS, 2007).

- **Nível de Granularidade**: Pacote.
- **Interpretação**: grupos isolados de classes no componente podem significar que o componente possui mais de uma funcionalidade e perda de coesão.

- **Descrição**: mensura o número de grupos de classes conectadas, isoladas dos demais grupos, em um componente. Esta métrica é exemplificada na Figura 3.9.
- **Fórmula**: $ConnComp_i = SUBCONJ(C_i)$, $1 \leq i \leq t$, onde: SUBCONJ é uma função que retorna o total de grupos de classes conectadas de um componente C_i. Considere que t é igual ao número total de componentes do sistema S e i varia de 1 a t.
- **Valor de referência**: 1 (não reconfigurável).

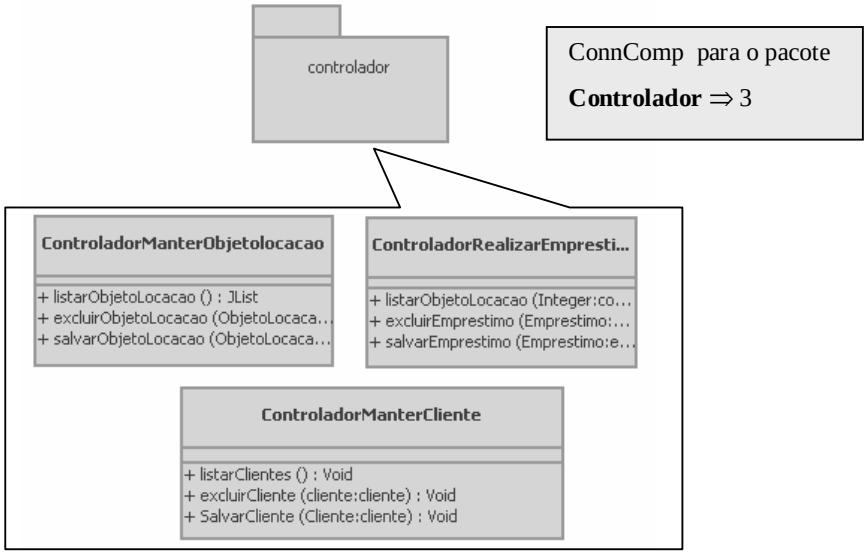


Figura 3.9: Exemplo de resultado da coleta da métrica ConnComp no sistema hipotético.

A Figura 3.9 apresenta o exemplo da métrica ConnComp para o pacote controlador. Para este pacote, obteve-se o valor 3 relativo a esta métrica, considerando-se o total de grupos isolados de classes dentro do pacote.

Acoplamento por Componente Requerido (CRC – *Coupling per Required Component*). Adaptada de (BLOIS, 2006).

- **Nível de Granularidade**: Pacote.
- **Interpretação**: verifica quantos componentes são necessários ao utilizar cada componente, pois o número de componentes requeridos deve ser o menor possível. A comunicação inter-componentes é muito custosa (VITHARANA, *et al.*, 2004).
- **Descrição**: a métrica soma todos os componentes que possuem classes, das quais o componente avaliado depende. Os relacionamentos considerados são as associações e as dependências, cuja navegabilidade possui sentido para classes

de outros componentes. Esta métrica é exemplificada na Figura 3.10.

- **Fórmula**: $CRC_i = \sum DEPEND(C_i, C_j)$, $1 \leq i \leq t$, $1 \leq j \leq t$, $i \neq j$, onde: DEPEND é uma função que retorna o valor um se o componente C_i possui ao menos uma classe que dependa de ao menos uma classe do componente C_j. Considere que t é igual ao número total de componentes do sistema S e os componentes, i e j, variam de 1 a t.
- **Valor de referência**: não possui, devendo ser montado um gráfico mostrando o número de componentes requeridos por componentes.

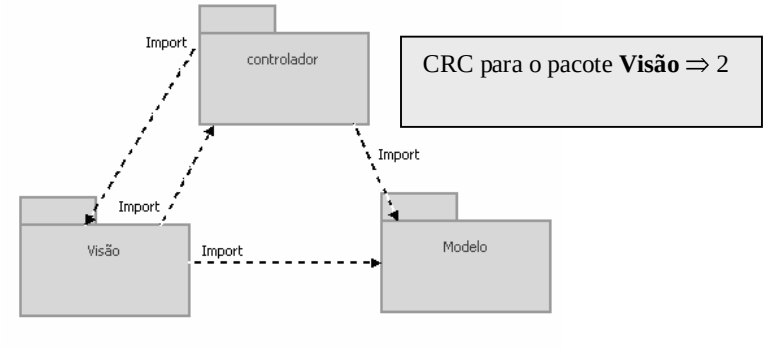


Figura 3.10: Exemplo de resultado da coleta da métrica CRC no sistema hipotético.

A Figura 3.10 apresenta o exemplo da métrica CRC para o pacote visão. Para este pacote, obteve-se o valor 2 relativo a esta métrica, considerando-se o total de pacotes que o pacote visão requer. Neste exemplo, o pacote visão tem classes que dependem de classes que pertencem aos pacotes modelo e controlador.

O conjunto de métricas proposto é apenas inicial, podendo ser estendido à medida que a abordagem seja utilizada e sejam descobertas novas métricas de interesse. É proposta uma ordem de priorização das métricas para guiar o engenheiro nas reestruturações. Esta ordem é sugerida com base na análise das reestruturações que são indicadas por cada métrica. Deste modo, a ordem leva em consideração métricas de classe antes de métricas de pacotes, porque nesta ordem os pacotes podem deixar de existir ou tornarem-se mais coesos, minimizando as sugestões de reestruturações a partir das métricas de pacote. Dentro desta ordem, a sub-priorização de métricas de classe é: CBOC, seguida de NOCC ou NOIC, porque nesta ordem as classes serão movidas para os pacotes aos quais estejam mais acopladas e, por isso, as hierarquias e realizações poderão ser movidas para os pacotes aos quais a maioria de seus elementos esteja mais acoplada. A sub-priorização das métricas de pacote é: A, CC e CRC, porque nesta ordem serão geradas superclasses ou interfaces que podem unir grupos de classes

isolados e, em seguida, a extração de novos componentes ocorrerá em menor número e, conseqüentemente, a geração de novos componentes poderá ser feita a partir de pacotes mais coesos e menos acoplados.

Os relacionamentos entre pacotes devem ser revistos após as reestruturações, pois as movimentações de classes, criação e eliminação de pacotes que venham a ficar vazios, podem afetar as dependências originais entre os pacotes.

3.2.2 Reestruturações

As reestruturações são utilizadas para re-organizar os agrupamentos de classes (i.e. pacotes) e cada tipo de reestruturação é guiado por uma ou mais métricas. É importante ressaltar que, a cada ciclo de reestruturações aplicado, as métricas são coletadas novamente sobre o novo modelo reestruturado. Estas métricas foram descritas na Seção 3.2.1 e, nesta seção, será feita a descrição dos tipos de reestruturação. O conjunto de reestruturações é apresentado com o seguinte *template*:

- **Nome** – Nomenclatura utilizada para referir-se à reestruturação. Para esta nomenclatura, não foi definida uma sigla correspondente.
- **Métrica relacionada** – Cita a(s) métrica(s) que está(ão) relacionada(s) a esta reestruturação. A partir do resultado da coleta desta(s) métrica(s) que as sugestões, deste tipo de reestruturação, são criadas.
- **Mecanismo** – Descreve o funcionamento da reestruturação, ou seja, como ela deve ser aplicada.
- **Opcionalidade** – A opcionalidade pode ser definida como: opcional (O) ou fortemente recomendada (FR). Onde fortemente recomendada significa a possibilidade de ferir algum princípio de DBC diretamente relacionado à definição de componente adotada neste trabalho (Capítulo 2) e opcional significa que irá melhorar o projeto do componente, mas não obrigatório para gerar o componente. A opcionalidade não influencia a ordem de aplicação das reestruturações, porque as mesmas são guiadas pelas respectivas métricas (Seção 3.2.1) que lhes deram origem. As métricas que guiam as reestruturações fortemente recomendadas não são reconfiguráveis por estarem diretamente relacionados aos princípios de DBC adotados neste trabalho, como visto na Seção 3.2.1 .

A seguir, são listadas as reestruturações conforme o *template* descrito anteriormente:

⇒ **<u>Extração de Componente</u>** – **Métrica relacionada**: Componentes Conectados (ConnComp). **Mecanismo**: Crie novos pacotes ou componentes a partir dos grupos de classes conectados e isolados dos demais no pacote. **Opcionalidade**: FR.

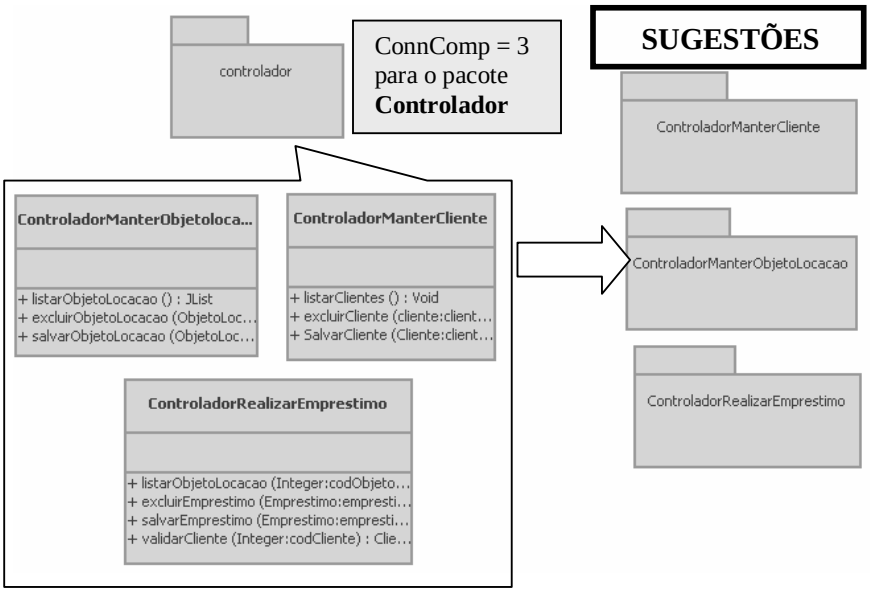


Figura 3.11: Exemplo de sugestões para extração de componente.

A Figura 3.11 mostra um exemplo das sugestões para o pacote controlador baseadas na coleta da métrica ConnComp para este pacote, que resulta no valor três. Deste modo, cada classe isolada do pacote poderá dar origem a um novo pacote, gerando três sugestões. Os novos pacotes são nomeados com o nome de uma de suas classes e, durante a verificação, o engenheiro de software poderá escolher um nome mais adequado. Para os exemplos das próximas reestruturações, considera-se que todas as sugestões da Figura 3.11 são aceitas pelo engenheiro de *software*. O pacote controlador que passa a estar vazio é removido.

⇒ **<u>Mover Classe</u>** – **Métrica relacionada**: Acoplamento entre Classes por Componentes (CBOC). **Mecanismo**: Mova as classes para os pacotes aos quais elas possuam o maior valor de acoplamento para os demais elementos. **Opcionalidade**: O.

A Figura 3.12 mostra um exemplo de sugestão de movimentação de classe para a classe ManterClienteGUI. Esta sugestão é feita porque a classe ManterClienteGUI possui duas classes, no pacote ControladorManterCliente, as quais está acoplada, conforme mostra a Figura 3.7. De acordo com esta sugestão, a classe será movida para o pacote ControladorManterCliente.

⇒ **<u>Mover Hierarquia</u>** – **Métricas relacionadas**: Número de Filhos de Superclasse por

Componente (NOCC) e Número de Implementações de Interfaces por Componente (NOIC). **Mecanismo**: mova as superclasses e/ou interfaces para os pacotes que possuam o maior número de subclasses ou classes que implementam esta interface imediatas, além de toda a hierarquia relacionada acima e subclasses, ou demais classes que implementam a interface para o pacote de destino. **Opcionalidade**: O para interfaces e FR para superclasses.

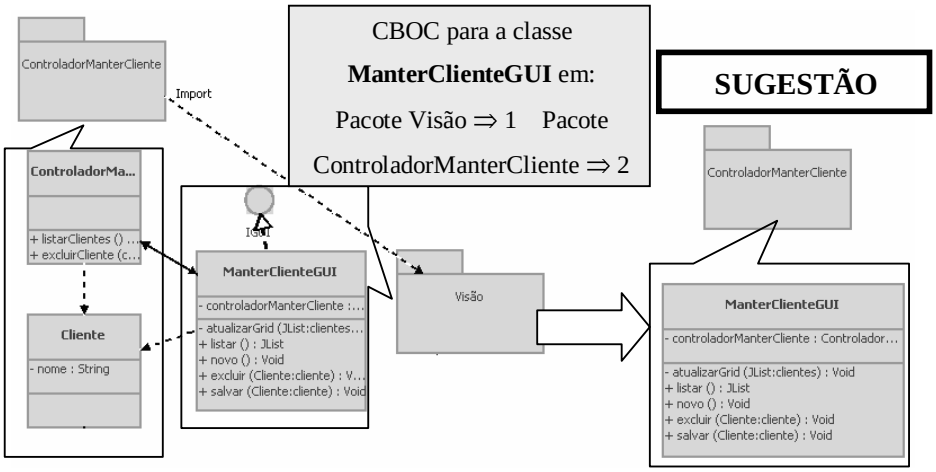


Figura 3.12: Exemplo de sugestão para mover classe.

No exemplo da

Figura 3.13, é apresentada uma sugestão de movimentação de hierarquia baseada na métrica NOIC para a interface IGUI. Após a reestruturação da Figura 3.12, a hierarquia formada pela interface IGUI está dividida nos pacotes Visão e ControladorManterCliente, gerando o valor dois para a interface IGUI no pacote Visão e o valor um para a mesma interface no pacote ControladorManterCliente. De acordo com a sugestão da

Figura 3.13, a hierarquia será movida inteiramente para o pacote Visão. Para a continuidade dos exemplos, considere que esta sugestão não é aceita pelo engenheiro de *software*, pois através do seu conhecimento do domínio ele opta por manter a classe ManterClienteGUI em um pacote para controlar as funcionalidades relativas ao cliente.

⇒ **Extração de Superclasse** – **Métrica relacionada**: Abstração (A). **Mecanismo**: Crie superclasses para as classes do pacote que possuam atributos com o mesmo nome e/ou operações com assinaturas similares, quando a métrica estiver indicando que o componente é bastante concreto ou concreto. No caso de operações com o mesmo

nome, mas com variação de parâmetros (i.e. sobrecarga), tipos de retorno ou quando houver diferenças nos tipos de atributos, uma notificação será gerada e o engenheiro deverá escolher a(s) opção(ões) que julgar mais adequada(s) para a superclasse. É importante ressaltar que não são geradas sugestões que levem à criação de classes com herança múltipla. **Opcionalidade**: O.

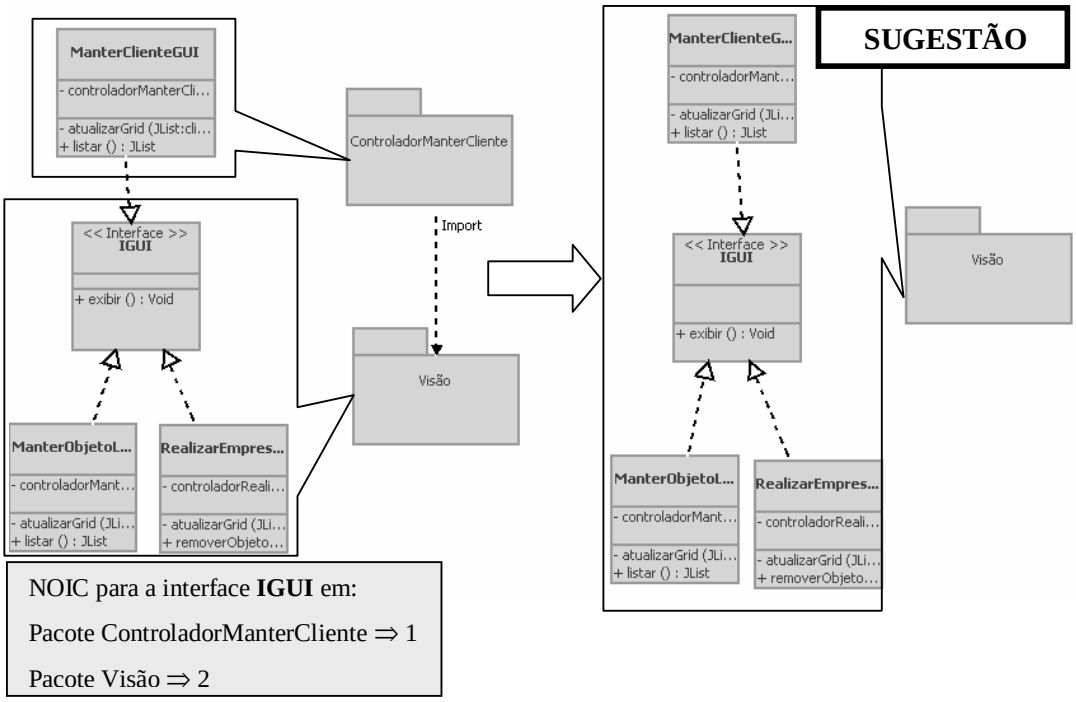


Figura 3.13: Exemplo de sugestão para mover hierarquia.

A Figura 3.14 mostra um exemplo de sugestão para a reestruturação de extração de superclasse. Esta reestruturação foi sugerida porque o pacote Modelo obteve o valor aproximado de 0,14 na coleta da métrica A e isto significa que o pacote ainda é bastante concreto. É importante ressaltar que quanto mais abstrato for o pacote, ou seja, quanto mais próximo de 1 for o valor da métrica A, mais adaptável ele se torna. Neste exemplo, é sugerido que seja gerada uma superclasse para as classes Cliente e Vídeo, porque as duas possuem o atributo nome em comum. O engenheiro de *software* deve utilizar o seu conhecimento do domínio, como em todas as sugestões, para decidir se a sugestão é interessante sob o ponto de vista semântico. Uma vez que os conceitos cliente e vídeo não pertencem semanticamente ao mesmo tipo, o engenheiro acabará por decidir pela não criação da herança.

⇒ **Extração de Interface** – **Métrica relacionada**: Abstração (A). **Mecanismo**: Crie interfaces para as classes do pacote que possuam somente operações com a mesma assinatura ou assinaturas similares, quando a métrica estiver indicando que o

componente é bastante concreto ou concreto. No caso de variação de parâmetros, uma operação sem parâmetro será gerada para a interface e as classes que realizam esta interface poderão sobrecarregar este método com seus respectivos parâmetros. Quando houver divergência no tipo de retorno, não será gerada a sugestão para criação de interface. **Opcionalidade**: O.

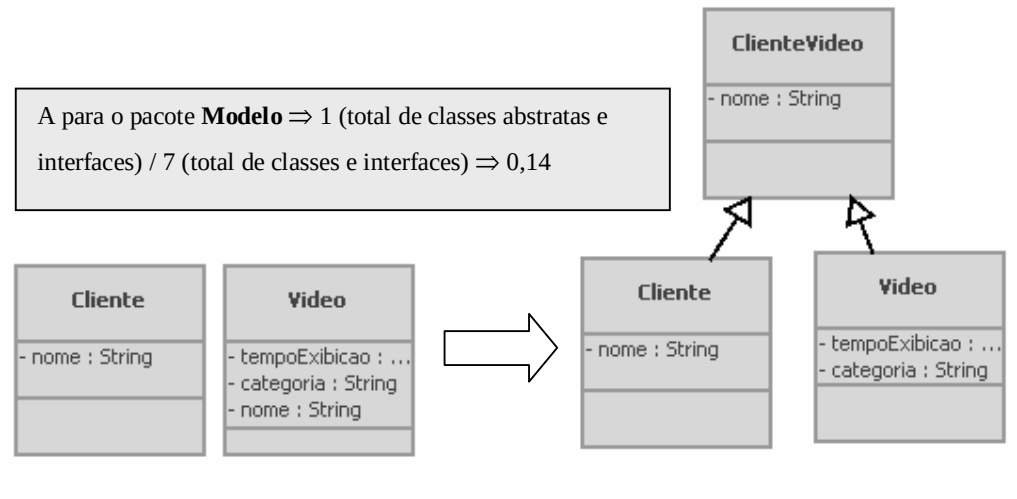


Figura 3.14: Exemplo de sugestão para extrair superclasses.

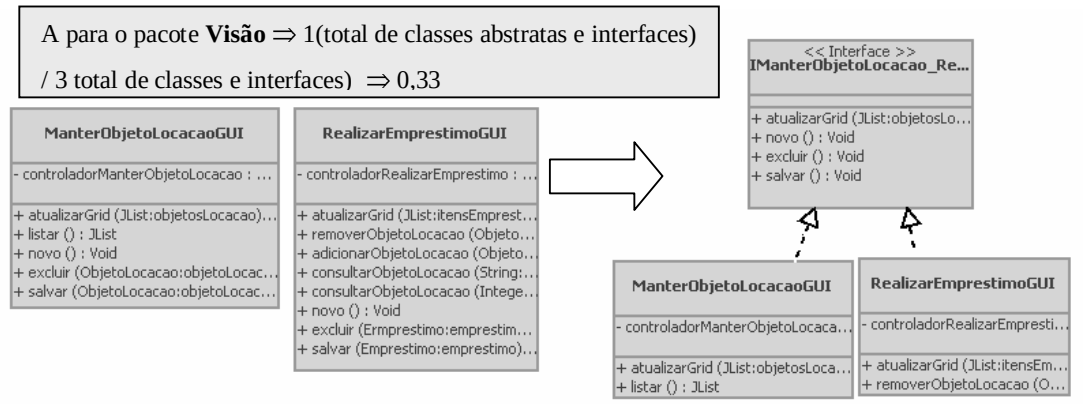


Figura 3.15: Exemplo de sugestão para extrair interface.

Para ilustrar a reestruturação de extração de interface, a Figura 3.15 mostra uma sugestão considerando operações em comum entre as classes ManterObjetoLocacaoGUI e RealizarEmprestimoGUI. As operações que possuíam divergência de parâmetro, como salvar e excluir, foram criadas sem parâmetros para que o engenheiro de software analise quais as assinaturas mais adequadas a nova estrutura criada. A necessidade deste tipo de sugestão surgiu porque o pacote Visão, antes das reestruturações aplicadas nesta seção, possuía o valor 0,33 para a métrica A.

⇒ **<u>Colapso de Componentes</u>** – **Métrica relacionada**: Acoplamento por Componente Requerido (CRC). **Mecanismo**: Unir dois ou mais projetos internos de pacotes em

um dos pacotes envolvidos ou criar um novo pacote englobando os pacotes envolvidos, considerando-se as seguintes situações: um pacote requer outros pacotes, os quais não são requeridos por outros pacotes; ou os pacotes estão envolvidos em uma dependência circular. **Opcionalidade**: O.

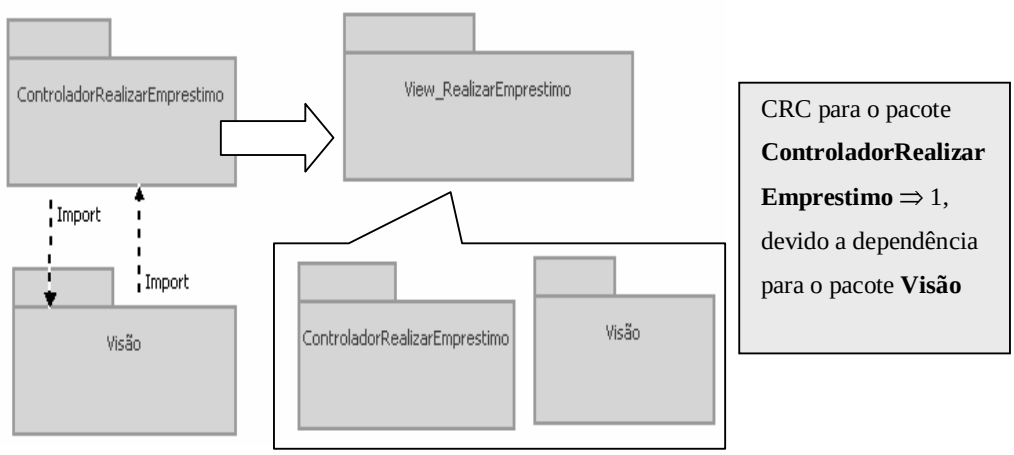


Figura 3.16: Exemplo de sugestão para unir componentes.

De acordo com a Figura 3.16, os pacotes ControladorRealizarEmprestimo e Visão requerem-se mutuamente, ou seja, pertencem a uma dependência circular. Deste modo, a sugestão de reestruturação envolve a criação de um novo pacote que agrupe os dois pacotes envolvidos na dependência circular, ou esta união deverá feita através da união de seus elementos internos dentro deste novo pacote.

3.2.3 *Checklist* para Verificação dos Modelos Reestruturados

Como mencionado na Seção 3.2, e mostrado na Figura 3.2, após o primeiro ciclo de reestruturações, o engenheiro de *software* pode fazer uma verificação do modelo reestruturado para encontrar não conformidades que podem prejudicar a geração de componentes e interfaces. A verificação é feita através de um *checklist* que deve apoiar a verificação da manutenção das funcionalidades após as reestruturações, assim como as características do modelo reestruturado que afetam a geração de componentes e interfaces. Deste modo, o *checklist* será composto de itens de verificação que são descritos utilizando-se o seguinte *template*:

- **Item do *checklist***: nomenclatura que define resumidamente o item do *checklist*;
- **Descrição**: descreve o que deve ser verificado neste item e a principal forma pela qual esta não conformidade pode acontecer;
- **Impactos na geração de componentes e interfaces**: descreve quais os impactos para a estratégia de geração de componentes e interfaces caso estas não

conformidades sejam ignoradas;

- **Ação**: descreve a ação que será sugerida para que a não conformidade seja resolvida.

Item do *Checklist* – Relacionamentos entre as classes e/ou interfaces corrompidos

Descrição: São apontados ao engenheiro de *software* cada relacionamento entre classes e/ou interfaces que foi corrompido durante a reestruturação, pois somente os relacionamentos entre pacotes deverão ser alterados ao longo da reestruturação.

Impactos na geração de componentes e interfaces: A quebra de relacionamentos entre as classes e/ou interfaces causará incompatibilidades na geração das interfaces dos componentes baseados em análise estática e afetará também o atendimento das funcionalidades providas pelo *software*, uma vez que a colaboração entre duas ou mais classes foi rompida.

Ação: Sugerir, ao engenheiro, a restauração de cada relacionamento corrompido, possibilitando que ele selecione cada restauração que deseja aplicar para que a abordagem a execute.

Este item aponta não conformidades, se o engenheiro de *software* alterar algum relacionamento manualmente, porque as reestruturações presentes na abordagem levam somente a modificações nos relacionamentos entre pacotes. Neste contexto, não são apontadas não conformidades, após a reestruturação do sistema hipotético de locação de vídeo, pois a reestruturação foi guiada inteiramente pelos passos descritos na abordagem.

Item do *Checklist* – Nomenclatura de pacotes fora do contexto do domínio

Descrição: São questionados todos os nomes de pacotes quanto a sua semântica, pois durante a reestruturação são gerados novos pacotes com uma nomenclatura padrão, que é formada pelo nome de uma das classes que compõe o pacote. Pacotes já existentes podem ter a sua estrutura modificada, de modo que o seu nome pode não mais refletir a sua funcionalidade.

Impactos na geração de componentes e interfaces: A falta de uma nomenclatura semântica pode afetar a geração de componentes, pois o engenheiro de *software* pode não conseguir ver um determinado pacote como um possível candidato a componente. A semântica dos nomes dos pacotes também apoiará a geração de componentes com nomes mais significativos e facilitará a visualização do modelo de componentes resultante.

Ação: Sugerir, ao engenheiro, a alteração do nome de cada pacote, de modo que ele julgue cada nome com relação à semântica e à funcionalidade que o pacote provê.

A Figura 3.17 mostra um exemplo de alteração de nome. Esta não conformidade com relação aos nomes é apontada para todos os pacotes, porque, após a reestruturação, as pacotes resultantes foram re-organizados e seus nomes podem não fazer mais sentido. Deste modo, o engenheiro deve verificar cada nome e fazer as devidas modificações, como na Figura 3.17, onde o pacote ControladorManterCliente não refletia mais o seu papel por possuir mais que um controlador e foi renomeado para ManutencaoDeCliente.

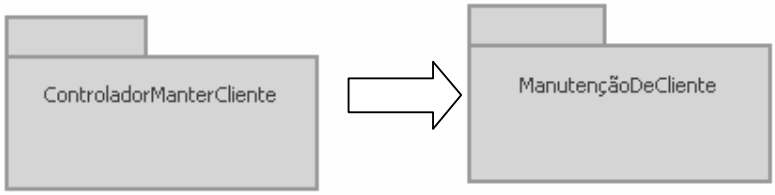


Figura 3.17: Exemplo de pacote gerado durante a utilização da abordagem.

Item do *Checklist* – Reestruturações fortemente recomendadas não aplicadas

Descrição: São questionadas todas as reestruturações fortemente recomendadas que não foram aplicadas.

Impactos na geração de componentes e interfaces: As reestruturações fortemente recomendadas impactam diretamente alguns princípios do desenvolvimento baseado em componente (DBC) e precisam ser analisadas pelo engenheiro de *software*, de modo que ele esteja consciente das conseqüências da sua não aplicação durante a geração de componentes.

Ação: Sugerir ao engenheiro a aplicação das reestruturações fortemente recomendadas, mostrando sua importância.

As reestruturações fortemente recomendadas são: extração de componentes e mover hierarquia (para superclasse). De acordo com as reestruturações aplicadas durante os exemplos desta seção, não restaram sugestões de reestruturações fortemente recomendadas que não tenham sido aplicadas. Deste modo, não são apontadas não conformidades deste tipo.

O projeto reestruturado do sistema hipotético de locação de vídeos pode ser visto na Figura 3.18a. Estes são os pacotes resultantes e seus relacionamentos. Na Figura 3.16b, são apresentadas as classes que estão agrupadas em cada um dos pacotes. De acordo com a Figura 3.18a, os pacotes (i.e. visão controlador e modelo) foram reorganizados em três novos pacotes e o pacote visão permaneceu, após a

reorganização, com apenas a interface IGUI (Figura 3.18b). Os novos pacotes estão mais relacionados aos conceitos do domínio e possuem maior facilidade de reúso neste contexto, enquanto o pacote visão está relacionado a funcionalidades técnicas de apresentação de tela e terá mais facilidade de ser reutilizado em domínios diferentes que necessitem deste tipo de funcionalidade. O pacote visão também pode ser usado futuramente como uma biblioteca de classes, da qual alguns componentes podem depender.

O estilo arquitetural utilizado no exemplo hipotético é o MVC (*Model-View-Controller*). Ao reestruturar a aplicação, este estilo arquitetural pode ser desfeito agrupando-se elementos da visão, do controle e do modelo em um novo pacote. Alguns agrupamentos de classe podem ser gerados para prover funcionalidades dentro dos pacotes visão, controlador e modelo, mantendo-se o estilo arquitetural inicial. O resultado depende da forma como as classes estejam organizadas e da aplicação do conhecimento do domínio do engenheiro de software. A abordagem ROOSC tem por objetivo gerar componentes com agrupamentos de classes estruturalmente coesos e pouco acoplados com os componentes externos, independente do estilo arquitetural do sistema alvo.

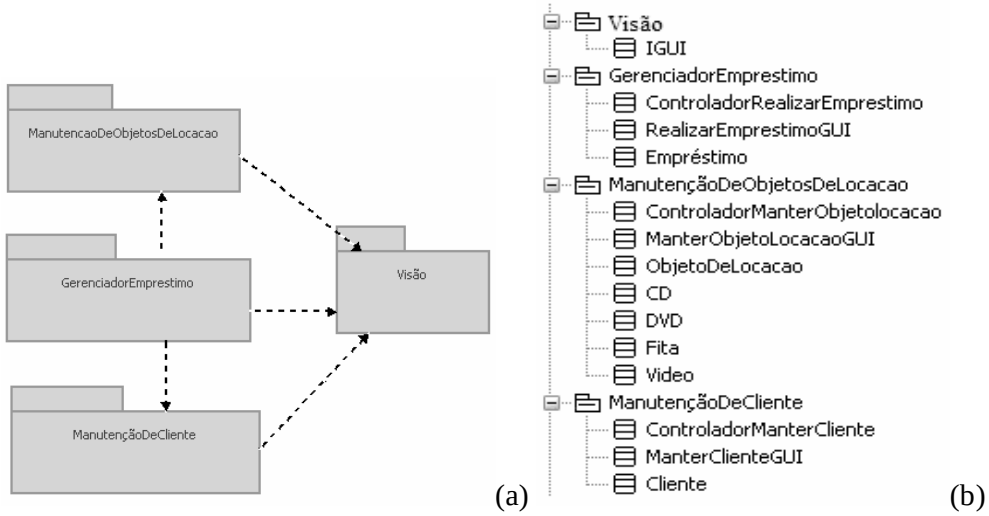


Figura 3.18: Pacotes reestruturados (a) e suas respectivas classes (b).

3.3 Geração de Componentes e Interfaces

Após a reestruturação dos agrupamentos de classes, o engenheiro de *software* passa a utilizar a segunda fase da abordagem ROOSC. Esta seção descreve os aspectos da estratégia de geração de componentes e interfaces que é utilizada na segunda fase da abordagem ROOSC.

O principal artefato de entrada para a geração de componentes e interfaces, como mostra a Figura 3.19, são os agrupamentos de classes (i.e. pacotes) do sistema alvo resultantes das reestruturações da fase anterior. No primeiro passo para a geração de componentes, o engenheiro de *software* determina um nível hierárquico inicial e outro final para que todos os pacotes, desde o nível inicial até o nível final, sejam sugeridos como componentes. A determinação do nível hierárquico se faz necessária porque os pacotes, após as reestruturações, podem possuir diferentes níveis hierárquicos e a percepção dos níveis que poderiam se tornar componentes depende do conhecimento do engenheiro de *software* acerca do domínio.

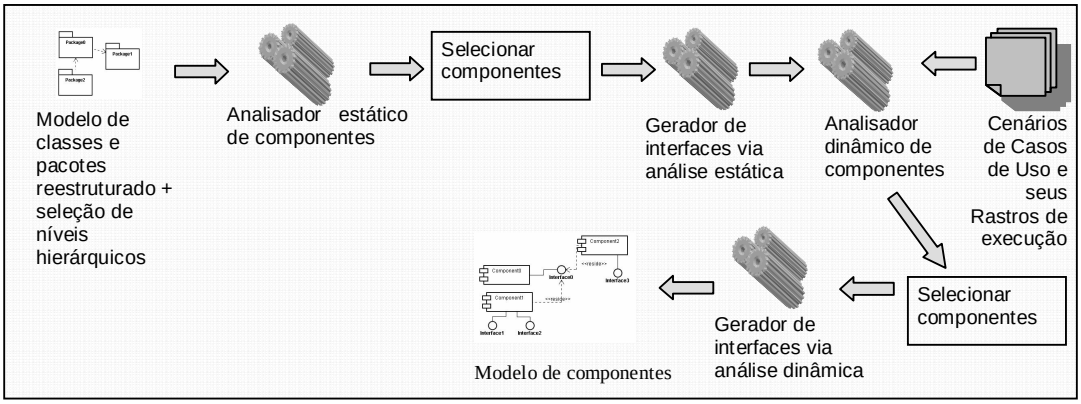


Figura 3.19: Estratégia proposta de geração de componentes e interfaces.

Para o sistema hipotético de locação de vídeo, utilizado para exemplificar a utilização da abordagem, só existe um nível hierárquico para os pacotes, como apresentado na Figura 3.18b.

De acordo com a estratégia apresentada na Figura 3.19, utiliza-se as informações de entrada para fazer uma análise sobre os elementos estáticos. Esta análise gera sugestões de componentes a partir de cada agrupamento de classe (i.e. pacote) que esteja no intervalo determinado pelos níveis hierárquicos definidos pelo engenheiro. O engenheiro de *software* seleciona os componentes que deseja gerar e as interfaces serão geradas a partir destes componentes de acordo com as regras de formação de interface, descritas na Seção 3.3.1, baseadas nos relacionamentos entre as classes dos diferentes componentes.

Após o engenheiro determinar alguns componentes com base nos elementos estáticos (i.e. classes e pacotes) reestruturados, a estratégia contempla uma análise de aspectos dinâmicos do sistema alvo. Para realizar esta análise, a abordagem necessita do segundo conjunto de artefatos de entrada para esta estratégia de geração de componentes, que são os cenários de caso de uso e os seus respectivos rastros de

execução. Os casos de uso são utilizados em algumas abordagens da literatura para a especificação de componentes, como visto no Capítulo 2, representando as funcionalidades do sistema alvo.

Um caso de uso (**uc**) representa a descrição de seqüências de ações, incluindo variantes, que uma entidade (ex: um sistema) executa para produzir um resultado de valor observável para um ator (BOOCH, *et al.*, 1998). Um **uc** visa atender a um requisito funcional (**rf**) do sistema, indicando que: **uc** ⇔ **rf**. Cada seqüência de ações em um **uc** determina um cenário de caso de uso, representando uma forma de obter aquele **rf** (BOOCH, *et al.*, 1998). Além disso, seqüências de eventos disparados por ações do usuário no sistema (como seleção de opção de menu, preenchimento de campos, clique de botão etc.) levam à execução de operações do sistema que implementam ou realizam um cenário de caso de uso, gerando cenários concretos (VASCONCELOS, 2007). Cada **uc** possui vários cenários de casos de uso relacionados, sendo que cada cenário de caso de uso possui vários cenários concretos relacionados, ou seja, vários caminhos de execução no software para se obter aquele cenário.

Neste contexto, é preciso ter algumas diretrizes para a determinação dos cenários e uma forma de obter os rastros de execução destes cenários. Alguns trabalhos presentes na literatura realizam a engenharia com análise dinâmica (BOJIC e VELASEVIC, 2000; DING e MEDVIDOVIC, 2001; RIVA e RODRIGUEZ, 2002; VASCONCELOS, 2007). Entre estes trabalhos, foi selecionado o trabalho de VASCONCELOS (2007) para a recuperação destas informações, ou seja, os rastros de execução dos cenários dos casos de uso porque, diferente das outras abordagens, define critérios para a redefinição de cenários a partir de sistema legado, uma vez que esta documentação geralmente está defasada, além disto, possui um ferramental de apoio integrado a um ambiente de desenvolvimento de *software*. VASCONCELOS (2007) apresenta algumas heurísticas para a definição dos cenários de casos de uso com base no sistema alvo (Tabela 3.1).

A partir da definição de cada cenário, deve-se extrair os rastros de execução, ou seja, deve-se extrair as chamadas de métodos necessários à sua execução. A Seção 3.3.2 mostra exemplos de cenários para alguns casos de uso do sistema hipotético de locação de vídeo, estes cenários foram definidos de acordo com as regras apresentadas na Tabela 3.1.

Antes de informar os rastros de execução destes cenários, o engenheiro de *software* irá definir os cenários de caso de uso que serão considerados para a sugestão de componentes de sistema (CHEESMAN e DANIELS, 2001), descrevendo, para cada

cenário, o seu nome. A partir destas definições, o engenheiro de *software* deve verificar a possibilidade de agrupar os cenários, visto que os componentes de sistema podem corresponder tanto a um caso de uso quanto a um cenário. Os rastros de execução por sua vez são obtidos no nível de cenário, porque representam os cenários concretos realizados durante a execução do software. Ao determinar os cenários que serão agrupados, o engenheiro de *software* deve ainda nomear cada grupo criado. No próximo passo, o engenheiro deve selecionar cada sugestão de componente de sistema que concordar. Estas sugestões são derivadas dos grupos de cenários ou cenários selecionados como entrada para a geração de componentes com base na análise dinâmica.

Tabela 3.1: Diretrizes para a definição de cenários de casos de uso em ArchMine. Extraída de Vasconcelos (2007)

Diretriz para a Definição de Cenário de Caso de Uso	Descrição
DUC_1	Para cada opção de menu principal e menu *popup* derive um cenário de caso de uso. Se as opções contiverem subgrupos de opções, então escolha a opção no último nível da hierarquia para a derivação do cenário.
DUC_2	Preencha todos os dados de entrada dos painéis disponibilizados quando uma opção de menu é acionada, garantindo assim que classes que tratem dos diferentes valores de entrada serão acionadas.
DUC_3	Para cada conjunto de cenários concretos que correspondam ao mesmo cenário de caso de uso, como, por exemplo, o mesmo cenário executado através de uma opção de menu principal e de menu *popup*, apenas um cenário concreto deve ser executado.
DUC_4	Botões de barras de ferramentas também devem ser consultados como candidatos à derivação de cenários de casos de uso, entretanto, deve ser respeitada a heurística DUC_3.
DUC_5	Abas de painéis com abas (t*abbed pane*s) podem indicar cenários de casos de uso distintos ou painéis distintos para o preenchimento de dados de um objeto. A semântica dos painéis deve ser investigada nesse caso e a heurística DUC_3 deve ser respeitada.
DUC_6	Botões não triviais em painéis, i.e. que não sejam "Ok", "Cancel", "Next", "Back", "Close", "Finish" etc., podem indicar cenários de casos de uso distintos em relação à opção de menu que levou à abertura do painel.
DUC_7	A inicialização do sistema, ex: *login* de usuário, deve derivar um cenário de caso de uso.
DUC_8	Cenários de exceção devem gerar cenários de casos de uso sempre que exercitarem classes do sistema não exercitadas nos demais cenários definidos para o caso de uso.

A partir dos componentes definidos, o gerador de interfaces via análise dinâmica (Figura 3.19) define as interfaces para componentes de sistema conforme as regras para formação de interface, definidas na Seção 3.3.2 . De um modo geral, estas regras

determinam quais interfaces serão geradas, suas nomenclaturas e suas operações. Após estas etapas o modelo de componentes está gerado por completo.

Por fim, o resultado é um modelo de componentes e suas respectivas interfaces. Como o objetivo da abordagem é realizar a reengenharia de sistemas OO para componentes no nível arquitetural, a implementação deste modelo não é contemplada pela abordagem.

3.3.1 Geração de Componentes Baseada em Análise Estática

A fim de gerar as sugestões para componentes com base na análise de elementos estáticos, deve-se considerar cada pacote que estiver definido entre os níveis hierárquicos determinados pelo engenheiro de software.

De acordo com o exemplo de sistema hipotético de locação de vídeos, através da análise destes elementos estáticos (pacotes) são geradas as possibilidades de componentes mostradas na Figura 3.20. A relação entre os pacotes de origem e os componentes gerados também é mostrada nesta figura.

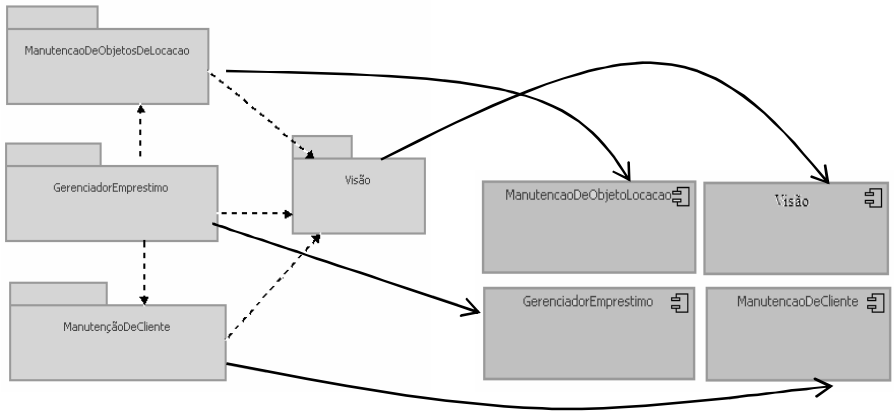


Figura 3.20: sugestões de componentes com base em análise de elementos estáticos.

Para cada componente, será necessário definir as suas interfaces providas e requeridas em modo mais abstrato do que as funcionalidades oferecidas por cada classe. Estas regras consideram os componentes gerados por análise de elementos estáticos, ou seja, os componentes gerados a partir dos pacotes reestruturados, e os componentes grados por análise de elementos dinâmicos, ou seja, os componentes gerados a partir dos cenários de casos de uso. As regras estão descritas na Tabela 3.2.

A regra RFI_1 é identificada através da análise dos elementos estáticos, ou seja, as dependências encontradas no modelo de classes e pacotes reestruturados são consideradas. Por exemplo, na análise da Figura 3.20, cada dependência das classes do componente GerenciadorEmprestimo para alguma classe do componente

ManutencaoDeObjetoLocacao significa que será gerada uma única interface provida pelo componente ManutencaoDeObjetoLocacao para atender às necessidades do componente GerenciadorEmprestimo. De acordo com a regra RFI_4, esta interface conterá todas as operações públicas das classes que estão no componente ManutencaoDeObjetoLocacao e são requeridas pelo componente GerenciadorEmprestimo, ou seja, devido a impossibilidade de conhecer quais são as operações realmente requisitadas apenas utilizando o modelo, considera-se todas as operações públicas da classe requerida.

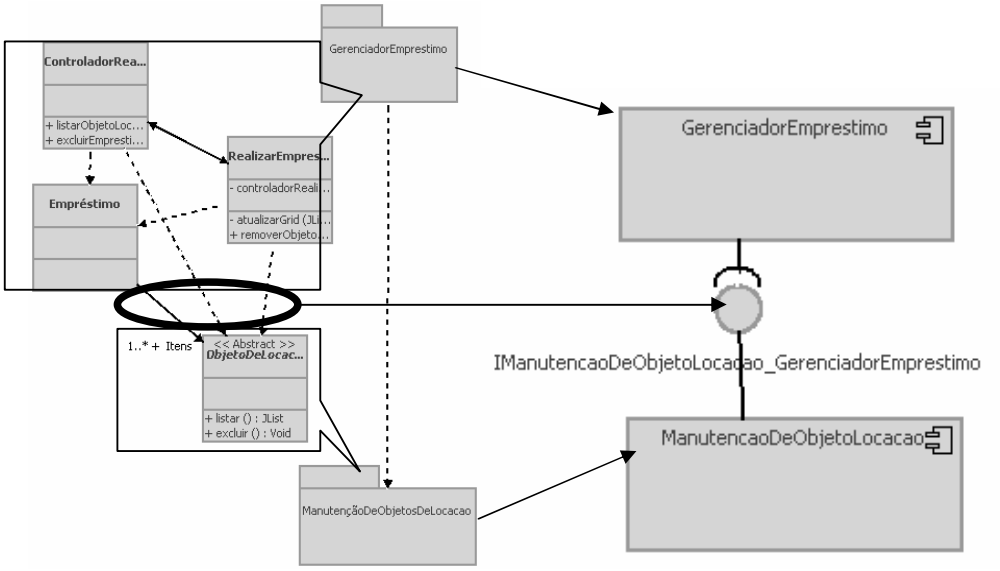


Figura 3.21: Definição de componentes e suas interfaces baseada em análise de elementos estáticos.

Tabela 3.2: Regras para formação de interfaces.

Regras de Formação de Interfaces	Descrição
RFI_1	Cada par de componentes que possuam classes que se comunicam (i.e. associação e dependência) deve derivar uma interface provida no componente requerido para atender o componente requerente.
RFI_2	Cada interface provida gerada pela regra RFI_1 deve tornar-se uma interface requerida pelo componente requerente.
RFI_3	As interfaces geradas pela RFI_1 devem utilizar o padrão de nomenclatura I<ComponenteRequerido_ + ComponenteRequerente> ou I<ComponenteRequerido_ + NomeCenario> para interfaces geradas a partir da regra RFI_6.
RFI_4	As interfaces geradas pela RFI_1 devem possuir todas as operações, de visibilidade pública, pertencentes às classes requisitadas pelo componente requerente.

Conforme a regra RFI2, automaticamente, a interface gerada será considerada uma interface requerida, do componente GerenciadorEmprestimo. O formato de nomenclatura destas interfaces é descrito pela regra RFI3. Seguindo-se o exemplo, o nome da interface provida pelo componente ManutencaoDeObjetoLocacao para atender as necessidades do componente GerenciadorEmprestimo é, conforme mostra a Figura 3.20, IManutencaoDeObjetoLocacao_GerenciadorEmprestimo.

3.3.2 Geração de Componentes Baseada em Análise Dinâmica

Como mencionado anteriormente, a geração de componentes através da análise de informações dinâmicas começa a partir dos casos de uso (agrupamento de cenários) ou cenários. A Tabela 3.3 mostra possíveis cenários e os casos de uso ao qual pertencem. Estes cenários foram definidos utilizando as diretrizes propostas por (VASCONCELOS, 2007), baseando-se no sistema hipotético de locação de vídeo. A Figura 3.22 ilustra as sugestões de componentes de sistema geradas a partir dos cenários dispostos na Tabela 3.3. Alguns cenários foram agrupados para exemplificar a possibilidade de agrupamento, como descrita anteriormente.

Tabela 3.3: Exemplo de um conjunto de cenários de casos de uso para um sistema de locação de vídeo hipotético.

Cenários de Casos de Uso	Caso de Uso
1. Incluir cliente.	Manter Cliente
2. Excluir cliente.	Manter Cliente
3. Criar empréstimo.	Realizar Empréstimo
4. Encerrar empréstimo.	Realizar Empréstimo
5. Incluir objeto de locação.	Manter Objeto de Locação
6. Consultar objeto de locação.	Manter Objeto de Locação
7. Incluir vídeo.	Manter Objeto de Locação

Figura 3.22: Sugestões de componentes com base em análise de elementos dinâmicos.

O engenheiro de *software* deve utilizar o seu conhecimento do domínio para selecionar as sugestões a serem aplicadas. Após a seleção dos componentes de sistema, é preciso gerar as suas interfaces. Estas interfaces, assim como os componentes de sistema, também serão geradas através da análise de informações dinâmicas. As regras

para a formação destas interfaces encontram-se dispostas na Tabela 3.4.

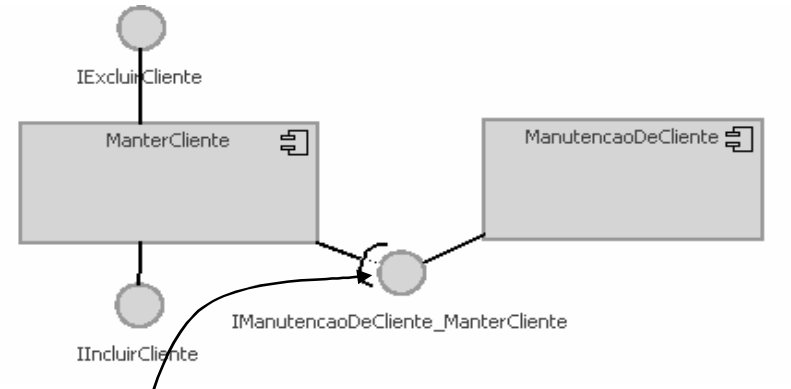
O componente de sistema terá uma interface provida para cada cenário de caso de uso que representa, conforme a regra RFI_1. A partir deste ponto, de acordo com a regra RFI_2, o componente de sistema conterá interfaces requeridas para os demais componentes gerados via análise estática, mas somente aqueles que contêm as classes necessárias à realização do(s) cenário(s). As classes necessárias à realização daquele cenário ou caso de uso (agrupamento de cenários) são descobertas através dos rastros de execução. Um exemplo é mostrado na Figura 3.23, onde o componente de sistema ManterCliente requer que uma interface seja provida pelo componente ManutencaoDeCliente, devido à informação em destaque presente no rastro de execução do exemplo. Os rastros devem ser examinados considerando-se as chamadas que estão localizadas no primeiro nível da árvore de rastros. Os demais níveis não são considerados, porque eles já foram tratados durante a geração dos componentes via análise estática, observando-se que as chamadas a partir das classes de primeiro nível estão refletidas no modelo estático através de algum relacionamento.

Tabela 3.4: Regras para a formação de interfaces.

Regras de Formação de Interfaces	Descrição
RFI_1	Cada componente de sistema deve derivar uma interface provida para cada cenário que representa.
RFI_2	Cada componente que possuir uma das classes que apareçam no primeiro nível dos rastros dos cenários deve derivar uma interface provida para atender o componente de sistema.
RFI_3	Cada interface provida gerada pela regra RFI_2 deve tornar-se uma interface requerida pelo componente requerente de processo.
RFI_4	As interfaces geradas pela regra RFI_1 devem utilizar o padrão de nomenclatura I< NomeCenario> ou I<NomeCasoDeUso>.
RFI_5	As interfaces geradas pela regra RFI_1 devem possuir uma operação com o nome no formato cenário_<<Nome_Cenário>>.
RFI_6	As interfaces geradas pela RFI_2 devem possuir todas as operações, de visibilidade pública, pertencentes às classes requisitadas pelo componente requerente de processo.
RFI_7	Interfaces existentes no projeto interno do componente (oriundas do modelo estático) serão expostas com suas operações.

As operações das interfaces requeridas pelos componentes de sistema obedecem

à regra RFI_6. Deste modo, a interface provida pelos componentes gerados via análise estática será composta por todas as operações públicas das classes que são utilizadas no primeiro nível da árvore dos rastros de execução. No exemplo da Figura 3.23, a interface, provida pelo componente ManutencaoDeCliente para atender o componente de sistema ManterCliente, conterá todas as operações públicas da classe ManterClienteGUI. Esta mesma interface deverá ser nomeada com padrão de nomenclatura estabelecido na regra RFI_3 da Tabela 3.4, para componentes gerados via análise estática. No exemplo da Figura 3.23, o nome da interface provida pelo componente ManutencaoDeCliente para o componente de sistema ManterCliente é IManutencaoDeCliente_ManterCliente.




```
<?xml version="1.0" encoding="ISO-8859-1" ?>
- <tree type="Call Tree" threadSelection="All thread groups" threadStatus="Runnable" aggregationLevel="Methods">
  - <node leaf="false" class="java/awt.EventDispatchThread" methodName="run" methodSignature="()V" time="242635" count="1" inherentTime="191255" lineNumber="-1"
      percent="100,0">
    + <node leaf="false" class="View.ManterClienteGUI$BtnIncluirListener" methodName="actionPerformed" methodSignature="(Ljava/awt/event/ActionEvent;)V"
        time="32631" count="1" inherentTime="720" lineNumber="-1" percent="13,4">
    </node>
</tree>
```

Figura 3.23: Definição de componentes e suas interfaces baseada em análise dinâmica.

Além das interfaces requeridas pelos componentes de sistema, o gerador de interfaces via análise dinâmica determina as interfaces providas pelos componentes de sistema conforme a regra RFI_1. No Exemplo da Figura 3.23, o componente ManterCliente prove duas interfaces referentes a cada cenário que agrupa (i.e. inclusão e exclusão de cliente (Tabela 3.3)). Estas interfaces são nomeadas de acordo com a regra RFI_4, como no exemplo da Figura 3.23, IIncluirCliente e IExcluirCliente. A fim de gerar as operações destas interfaces providas, utiliza-se a regra RFI_5 e de acordo com ela a interface do exemplo, IIncluirCliente conterá uma única operação, denominada

incluirCliente().

As classes, as quais sua utilização for evidente somente dentro do próprio pacote que as contêm, não terão suas operações expostas através de uma interface, não sendo enquadradas por nenhuma regra de formação de interface. As interfaces que existirem dentro do próprio pacote, que deu origem ao componente, serão expostas sem nenhuma alteração em sua nomenclatura ou operações, conforme regra RFI_{11}. No exemplo da Figura 3.24, a interface originalmente presente no pacote Visão será uma interface para o componente gerado a partir deste pacote. Estas interfaces serão consumidas pelos componentes já definidos ou ficarão disponíveis para consumo em futuras aplicações. Nos casos em que componentes implementem interfaces presentes em componentes distintos, como no exemplo da Figura 3.24, onde a interface IGUI é provida pelo componente Visão e pelo componente ManutencaoDeObjetoLocacao, estas interfaces serão replicadas para estes componentes. Dessa forma, esses componentes podem ser substituídos entre si para os componentes que requeiram esta interface. Na Figura 3.24, o componente Visão pode ser substituído pelo componente ManutencaoDeObjetoLocacao nos componentes que necessitem da funcionalidade disponível na interface IGUI.

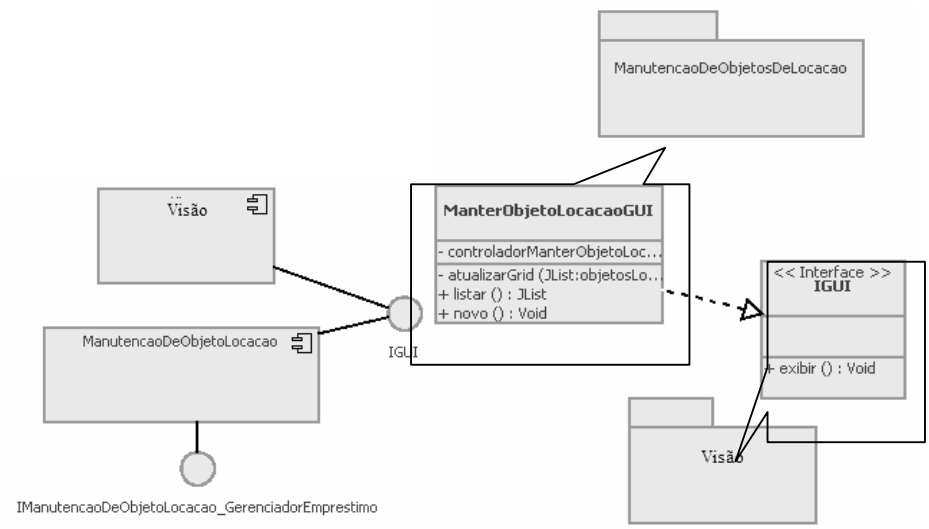


Figura 3.24: Definição de interface pré-existente em modelo estático.

Esta abordagem possui a desvantagem de agrupar uma mesma operação em mais de uma interface no mesmo componente, pois o agrupamento de operações é feito com relação à necessidade dos componentes requerentes. Diferentes componentes podem ter classes em seu projeto interno que dependam de uma mesma classe presente em outro

componente e, de acordo com esta abordagem, as operações públicas desta classe serão expostas duas vezes. Apesar desta redundância, conforme o princípio da segregação de interface (ISP): “Muitas interfaces específicas de clientes são melhores do que uma interface de propósito geral”, descrito no Capítulo 2, de acordo com o exposto em (PRESSMAN, 2006), se múltiplos clientes solicitam as mesmas operações, elas devem ser especificadas em cada uma das interfaces especializadas por cliente (componente requerente).

Uma outra forma de gerar interfaces é criar uma interface provida para cada classe do componente que for solicitada através de algum relacionamento (i.e. associação ou dependência) por outro componente. Esta interface seria composta pelas operações de visibilidade pública da classe. Para cada componente que necessitasse desta classe, a interface com as operações públicas desta classe seria uma interface requerida para o componente. Esta abordagem implicaria em um número maior de interfaces providas e requeridas, aumentando o número de dependências entre os componentes.

3.4 Considerações Finais

Este capítulo apresentou a abordagem ROOSC proposta nesta dissertação para a reengenharia de *software* orientado a objetos (OO) para componentes, onde o produto final da abordagem é um modelo de componentes. Além dos componentes e suas interfaces, a ROOSC apóia o projeto interno de cada componente, o qual contempla as classes e/ou pacotes existentes a princípio no *software* OO.

ROOSC pode ser utilizada para a obtenção de componentes a partir de *software* OO, com o propósito de apoiar a reutilização destes componentes em novos desenvolvimentos, assim como pode ser utilizada para apoiar a manutenção. O impacto de manutenções nos componentes não afetará o restante do *software* devido ao acoplamento entre os componentes ser realizado através de interfaces, as quais devem ser respeitadas durante as manutenções.

A Tabela 3.5 apresenta um comparativo entre as abordagens de reengenharia de componentes apresentadas no Capítulo 2 e ROOSC, em função dos requisitos estabelecidos nesta dissertação. Dois dos requisitos estabelecidos, i.e. apoio ferramental e integração a um ambiente de reutilização, são avaliados e expostos na tabela do Capítulo 4, que apresenta o ambiente e ferramental de apoio à execução da abordagem ROOSC.

Em comparação com as abordagens analisadas, ROOSC traz algumas contribuições, como a reengenharia da aplicação como um todo, partindo-se de um sistema OO que, como visto no Capítulo 1, traz a vantagem de reutilizar soluções para projeto OO.

A abordagem é realizada no nível de modelo, permitindo alguma generalidade em relação à linguagem de implementação do sistema alvo. Apesar disso, a generalidade é comprometida por não considerar algumas construções da OO, como por exemplo, a herança múltipla, não havendo garantia de que a abordagem é aplicável à sistemas desenvolvidos em qualquer linguagem de programação OO.

Entre as contribuições, ainda encontram-se o apoio à formação dos componentes com o seu projeto interno e o apoio à formação de interfaces, permitindo que o engenheiro de *software* seja guiado de acordo com os princípios de projeto, mas que possa utilizar também o seu conhecimento do domínio para obter componentes mais coesos e semânticos. As métricas para a formação do projeto interno dos componentes e as diretrizes para a formação dos componentes e interfaces propriamente podem ser aplicadas em diferentes domínios e linguagens de programação, diferentemente de outras abordagens (WASHIZAKI e FUKAZAWA, 2005; WANG, *et al.*, 2006).

Convém ressaltar que o conjunto de métricas proposto para reestruturação do projeto OO é apenas inicial, podendo ser estendido futuramente através da seleção ou adaptação de métricas existentes ou através da criação de novas métricas à medida que a abordagem é utilizada, como mencionado anteriormente. De acordo com as estratégias adotadas em ROOSC, a extensão deste conjunto de métricas pode levar a extensão do conjunto de reestruturações, pois as métricas indicam alguma reestruturação.

Dessa forma, pela, verifica-se que o desenvolvimento de ROOSC foi motivado por não haver uma abordagem de reengenharia para componentes, dentre as pesquisadas, que atendesse plenamente ou minimamente a todos os requisitos estabelecidos nesta dissertação.

Tabela 3.5: Comparação entre abordagens.

Trabalho	Paradigma de Origem	Generalidade	Apoio à composição de Componentes (agrupamento de classes)	Apoio à composição de interfaces (agrupamento de operações)	Reengenharia de toda a aplicação	Integração a um ADS	Ferramental
LEE *et al.* (2001)	OO	Sim. Aplicável no nível de modelo	Sim. As sugestões de componentes são geradas automaticamente por algoritmo de *clustering*	Não	Não. Apenas extração de componentes	Não	Não
LEE *et al.* (2003)	OO	Sim	Sim. Através de algoritmos de *clustering*	Sim	Sim	Não	Não
PRADO (2005)	Procedural	Parcial. Para cada sistema alvo, é preciso customizar a ferramenta que apóia a análise do código fonte	Não	Parcial. É indicada a utilização do método *Catalysis*	Sim	Parcial. Os autores sugerem duas ferramentas que se comunicam indiretamente através de importação de arquivos.	Sim
GANESAN e KNODEL (2005)	OO	Sim. Aplicável no nível de modelo	Parcial. Apóia a identificação das classes que irão compor os componentes	Sim. É indicada análise de interface proposta por (KNODEL, 2004)	Não. Apenas extração de componentes	Não	Não
WASHIZAKI e FUKAZAWA (2005)	OO em Java	Não. Aplicável somente para sistemas desenvolvidos em Java	Sim. Os componentes são extraídos automaticamente	Sim	Não. Apenas extração de componentes	Não	Sim
WANG *et al.* (2006)	OO em C++	Não. Aplicável somente para sistemas desenvolvidos em C++	Sim. Os componentes são gerados automaticamente através de algoritmo de *clustering*	Sim. Através de duas heurísticas para interfaces em componentes em J2EE	Sim	Não	Parcial. Existe ferramenta para a conversão de código C++ em Java
CHARDIGNY *et al.*, (2008)	OO	Sim	Sim	Não	Sim	Não	Não
WANG *et al.* (2008)	OO	Sim	Sim. Através de *clustering*	Não	Sim	Não	Não
ROOSC	**OO**	**Parcial**	**Sim**	**Sim**	**Sim**	**Sim**	**Sim**

Capítulo 4 Ferramental de Apoio à Abordagem ROOSC

4.1 Introdução

O capítulo 3 apresentou a abordagem ROOSC de reengenharia de software orientado a objetos (OO) para componentes, proposta nesta dissertação. Conforme descrito, a abordagem é composta de 2 estratégias: estratégia de reestruturação e a estratégia de geração de componentes e interfaces. Este capítulo apresenta as ferramentas **ORC** (*Object Restructuring to Component*) e **GenComp** (*Generating Component*), construídas para apoiar a execução destas estratégias, respectivamente.

As ferramentas desenvolvidas nesta dissertação apresentam a possibilidade de integração com o ambiente de apoio à reutilização, Odyssey (ODYSSEY, 2008). O Odyssey foi selecionado por oferecer recursos que apóiam a abordagem proposta, sendo um ambiente de apoio à reutilização e ao DBC, englobando processos de desenvolvimento "com" e "para" reutilização, além de estar sendo desenvolvido no mesmo contexto de pesquisa desta dissertação, i.e. pelo grupo de reutilização de software da COPPE/UFRJ.

A título de ilustração do apoio oferecido pelas ferramentas desenvolvidas (ORC e GenComp), é apresentado um exemplo de reengenharia de software OO para componentes em que o sistema alvo é a ferramenta TraceMining. TraceMining é uma ferramenta utilizada na recuperação de arquitetura de referência através de técnicas de mineração de dados e está presente no ambiente Odyssey (VASCONCELOS, 2007), assim como o ferramental de apoio apresentado neste capítulo. O modelo de pacotes da TraceMining, no ambiente Odyssey, é apresentado na Figura 4.1. Como mostra esta Figura, o modelo é composto por quatro pacotes: *Integration* (contém classes para exportar alguns elementos para o Odyssey), *Preparation* (prepara os dados de entrada para a mineração), *StaticClustering* (contém classes responsáveis por gerar *clusterings* a partir de modelo estático), GUI (contém as classes de interface gráfica da aplicação).

Em relação às abordagens de reengenharia de software OO para componentes, dois requisitos estabelecidos no capítulo 2 são avaliados neste capítulo, a saber: apoio ferramental, a fim de reduzir o esforço de execução das atividades do processo, e integração a um ambiente de reutilização, para que os modelos recuperados possam ser reutilizados em novos desenvolvimentos, principalmente, naqueles baseados em

componentes.

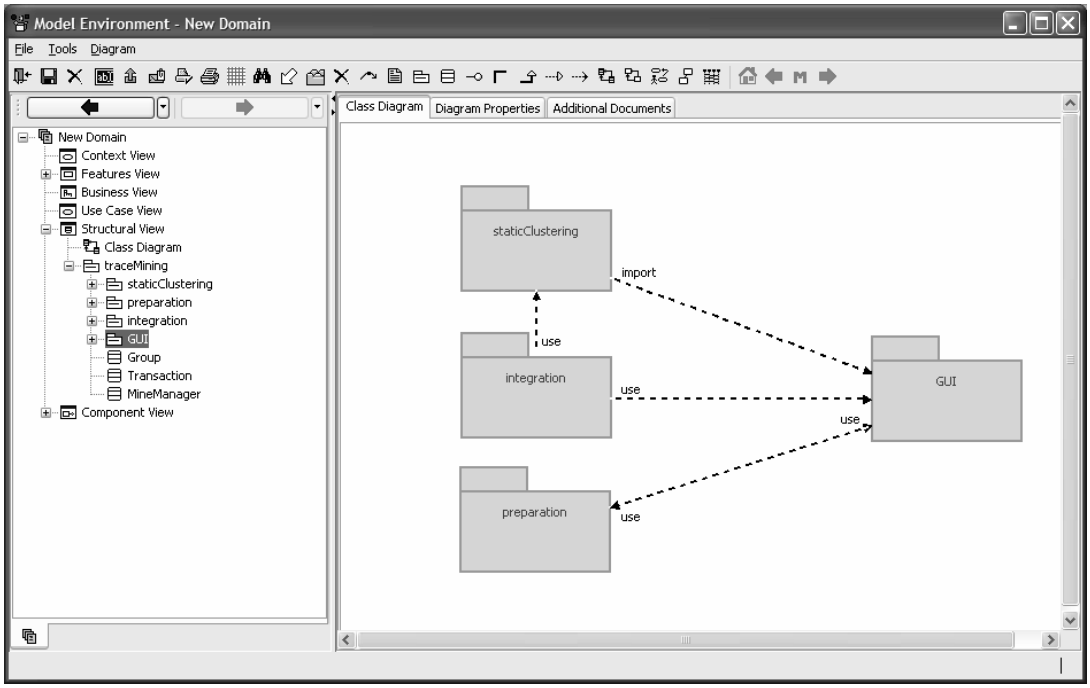


Figura 4.1: Modelo de pacotes da ferramenta TraceMining.

Partindo desta Introdução, o restante do capítulo está organizado da seguinte forma: a Seção 4.2 apresenta as principais características do ambiente Odyssey para o apoio ao Desenvolvimento Baseado em componentes (DBC); a Seção 4.3 apresenta a ferramenta ORC de apoio à estratégia de reestruturação; a Seção 4.4 apresenta a ferramenta GenComp de apoio à geração de componentes e interfaces; e a Seção 4.5 apresenta as considerações finais do capítulo.

4.2 O Ambiente Odyssey

O ambiente Odyssey (ODYSSEY, 2008) é um ambiente de desenvolvimento de software baseado em reutilização, que disponibiliza funcionalidades e ferramentas para apoio a abordagens de reutilização, como o DBC. O Odyssey começou a ser desenvolvido em 1998 (WERNER, *et al.*, 1999) e vem sendo evoluído até os dias atuais, através de uma série de trabalhos de doutorado (BRAGA, 2000; BLOIS, 2006; MANGAN, 2006; MURTA, 2006), mestrado (MILER, 2000; XAVIER, 2001; MURTA, 2002; TEIXEIRA, 2003; DANTAS, 2005; LOPES, 2005; GONÇALVES, *et al.*, 2006; MAIA, 2006) e graduação (MURTA, 1999; DANTAS, 2001; VERONESE e NETTO, 2001; MELO JR, 2005; SILVA, 2005), além de outros trabalhos que desenvolvem ou estendem funcionalidades e ferramentas para o ambiente.

O ambiente Odyssey possui uma distribuição "*light*", denominada OdysseyLight. Como mostra a Figura 4.2, o ambiente Odyssey conta com um núcleo de funcionalidades, além de ferramentas disponibilizadas na forma de *plugins*. Funcionalidades como a modelagem de domínio, a gerência de processos (MURTA, 2002) e a geração de componentes de software (BLOIS, 2006) são mantidas no núcleo, ao passo que funcionalidades classificadas como secundárias são mantidas na forma de *plugins*, como, por exemplo, o suporte a padrões (XAVIER, 2001), o suporte à extração de modelos de classes (VERONESE e NETTO, 2001), a notificação de críticas sobre modelos (DANTAS, *et al.*, 2001), a edição concorrente de modelos, a exportação e importação de modelos em XMI, dentre outras.

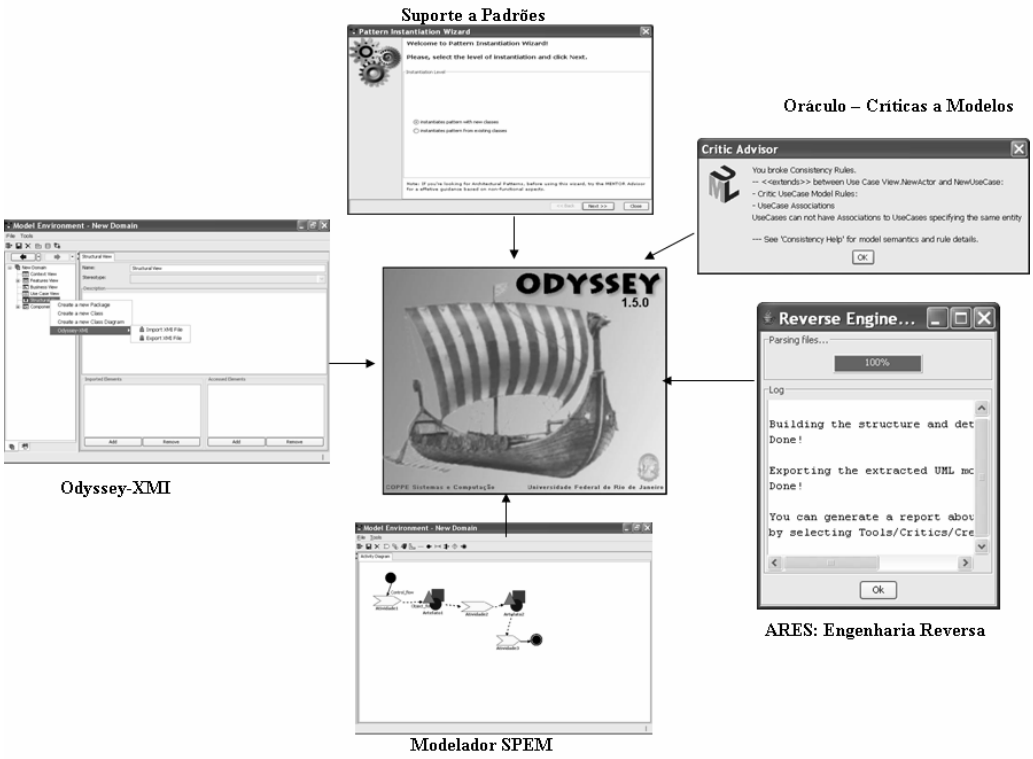


Figura 4.2: Ambiente OdysseyLight e exemplos de alguns de seus *plugins*.

Os *plugins* devem ser baixados pelos usuários do Odyssey sob demanda. Uma lista de *plugins* é mantida no servidor do grupo de reutilização da COPPE e acessada por cada instância do Odyssey, permitindo que os seus usuários selecionem os *plugins* que desejam baixar. Os *plugins* conhecem o núcleo do Odyssey e podem acessá-lo, enquanto o Odyssey não precisa conhecer os seus *plugins*. O padrão de projeto Observer (GAMMA, *et al.*, 2000) foi utilizado para garantir que o Odyssey não possua

acoplamento explícito com os seus *plugins*.

Cada *plugin* do Odyssey deve implementar a interface *Tool* (Figura 4.3), a fim de que suas opções de menu sejam disponibilizadas no Odyssey. Essa interface especifica serviços como *getEnvironmentMenu*(), para disponibilizar uma opção de menu na tela principal do Odyssey, e *getPopupMenu*(), para disponibilizar uma opção de menu *popup* no ambiente. Uma vez que um *plugin* seja baixado por um usuário, ele entra na lista de ferramentas que o Odyssey acessa através dos serviços da interface *Tool*. Maiores detalhes sobre o mecanismo de carga dinâmica do Odyssey podem ser obtidos em (MURTA, *et al.*, 2004).

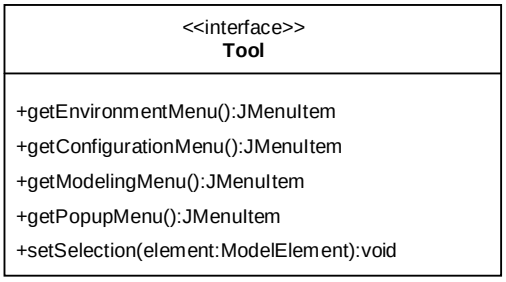


Figura 4.3: Interface *Tool* implementada pelos *plugins* do OdysseyLight.

4.3 ORC – *Object Restructuring to Components*

A ferramenta ORC (MOURA, *et al.*, 2008b) é aplicada sobre modelos de projeto do sistema alvo, tornando fundamental a sua obtenção através de alguma ferramenta de engenharia reversa, como dito anteriormente na descrição da estratégia de reestruturação (Seção 3.2). Para sistemas desenvolvidos em Java, pode-se utilizar a ferramenta Ares (VERONESE e NETTO, 2001) para realizar a engenharia reversa, favorecendo-se de sua integração com o ambiente Odyssey. Nos casos em que o sistema alvo for desenvolvido em outra linguagem de programação, deve-se utilizar outra ferramenta que realize a engenharia reversa e exporte em formato XMI compatível com o formato aceito pelo Odyssey (XMI versão 1.2). O modelo de projeto (Figura 4.1) utilizado como exemplo foi obtido utilizando a ferramenta de engenharia reversa Ares (VERONESE e NETTO, 2001).

A Figura 4.4 apresenta o funcionamento geral da ferramenta ORC. A ferramenta é um *plugin* para o ambiente Odyssey e está implementada na linguagem Java. Para a criação do *plugin*, a ferramenta possui uma classe para a integração com o ambiente Odyssey, que implementa uma interface padrão para o desenvolvimento de *plugins* do Odyssey, denominada “ToolAdapter” (Figura 4.3). Esta classe está presente no pacote

Integration, que é mostrado na Figura 4.5. De acordo com a Figura 4.4, O modelo recuperado através de alguma ferramenta de engenharia reversa é acessado pela ferramenta ORC diretamente no ambiente Odyssey, após a importação do modelo em formato XMI, caso o mesmo não seja obtido pela ferramenta de engenharia reversa do Odyssey, i.e. a Ares, como mencionado anteriormente. O modelo passa por uma coleta de métricas e, com base em seus resultados, são sugeridas algumas reestruturações no modelo.

Como mostra a Figura 4.4, a ferramenta ORC possui um módulo coletor de métricas, que coleta as métricas a partir do modelo no Odyssey, e um reestruturador de modelos, que atualiza o modelo recuperado no Odyssey, de acordo com as reestruturações selecionadas. Após a atuação do coletor de métricas, as métricas podem ser visualizadas na própria ferramenta, enquanto a cada reestruturação, o modelo reestruturado pode ser visualizado através do ambiente Odyssey.

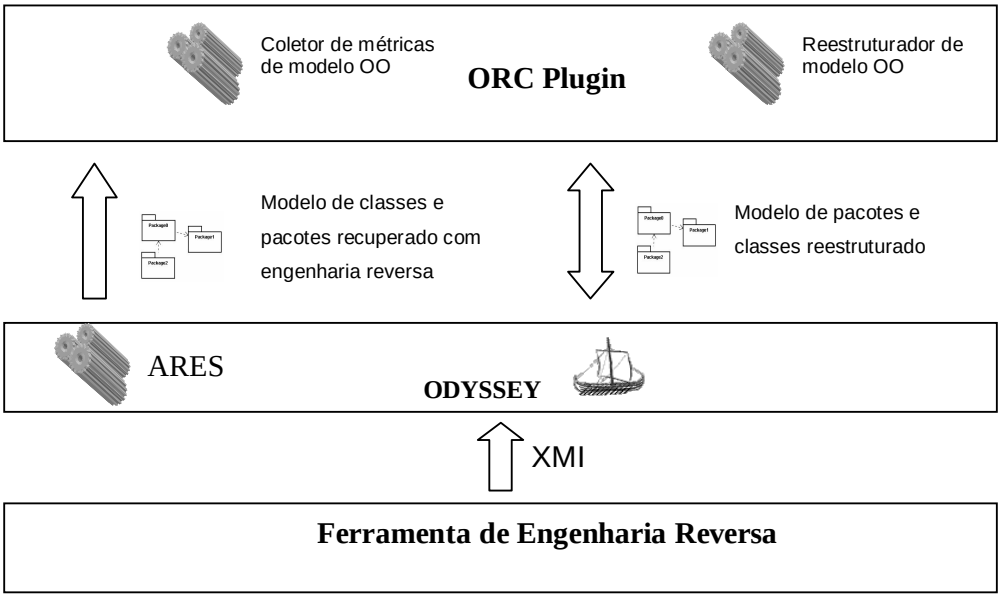


Figura 4.4: Integração da ferramenta ORC ao ambiente Odyssey.

A Figura 4.5 apresenta uma visão geral da arquitetura da ferramenta. Ela foi implementada com um total de 6 pacotes, como mostra a Figura 4.5, e 20 classes. Os pacotes possuem os seguintes propósitos: classes gerais de interface gráfica que serviriam para qualquer interface gráfica, como classes para a geração de gráficos (Gui); integração com o ambiente Odyssey (*Integration*); classes específicas do reestruturador e do coletor de métricas (*Restruct*) e internamente ao pacote *Restruct* há mais 3 pacotes, a saber: classes de interface gráfica específicas para a estratégia de reestruturação, como

as telas da ferramenta apresentadas na Seção 4.3.1 ; classes do coletor de métricas (*MetricCollector*); e classes próprias para o reestruturador (*Restructurer*).

A ORC possui um formato de *wizard*, onde cada tela é um passo para a utilização da estratégia de reestruturação. Nesta seção, cada tela do *wizard* é mostrada durante a apresentação do exemplo de utilização.

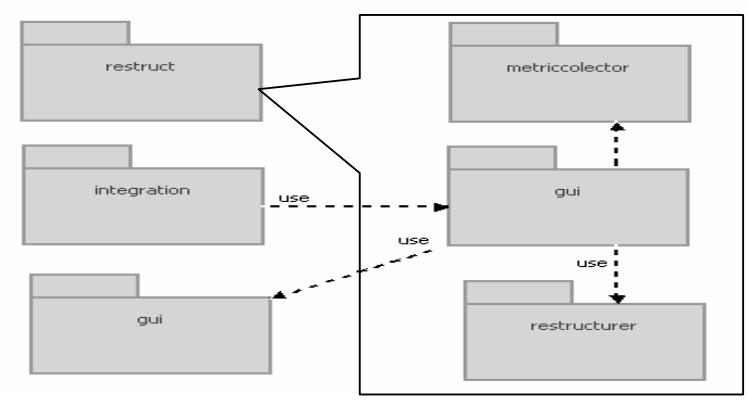


Figura 4.5: Visão geral dos elementos arquiteturais da ferramenta ORC.

4.3.1 Exemplo de Utilização

A ferramenta ORC é aplicada sobre modelos, portanto para ilustrar a utilização da ferramenta, primeiramente, foi obtido o modelo OO estático do projeto do sistema alvo. O projeto possui uma série de agrupamentos de classes, que são alvos da reestruturação. O sistema alvo é a ferramenta TraceMining (VASCONCELOS, 2007), como dito anteriormente.

Inicialmente, o engenheiro de software é informado sobre o objetivo da estratégia de reestruturação que a ferramenta apóia, como mostrado na Tela da Figura 4.6. Além do objetivo, ele é informado sobre as métricas que serão disponibilizadas para ele. Esta explicação geral da estratégia de reestruturação é abordada em detalhes na Seção 3.2.

O segundo passo da estratégia é a seleção do nível de granularidade (i.e. classe ou pacote) que determina o conjunto de métricas e sugestões de reestruturação que serão visualizados. Deste modo, o engenheiro de software pode reestruturar o modelo em duas etapas. A ferramenta apresenta uma explicação sobre o nível de granularidade, incluindo informações sobre a ordem de utilização das métricas mais adequada, como apresentado na Figura 4.7, visando à minimização do número de reestruturações a serem aplicadas, e neste ponto o engenheiro de software seleciona o nível por onde deseja começar a reestruturar. Durante todo o tempo de reestruturação, o engenheiro de

software poderá retornar a este ponto e alterar o nível de granularidade sem necessitar recomeçar a reestruturação ou perder as reestruturações já aplicadas. No exemplo da Figura 4.7, a opção selecionada corresponde ao nível de granularidade de pacote e isto significa que somente as métricas que são coletadas sobre os pacotes são apresentadas.

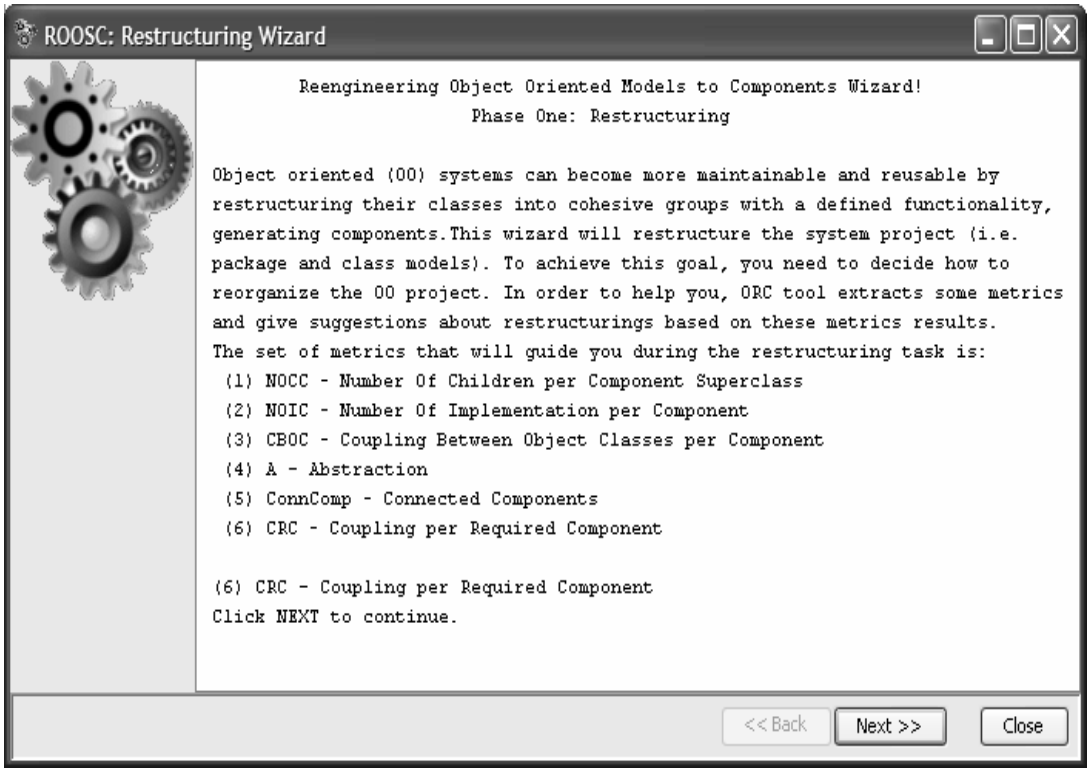


Figura 4.6: Tela de introdução da ferramenta ORC.

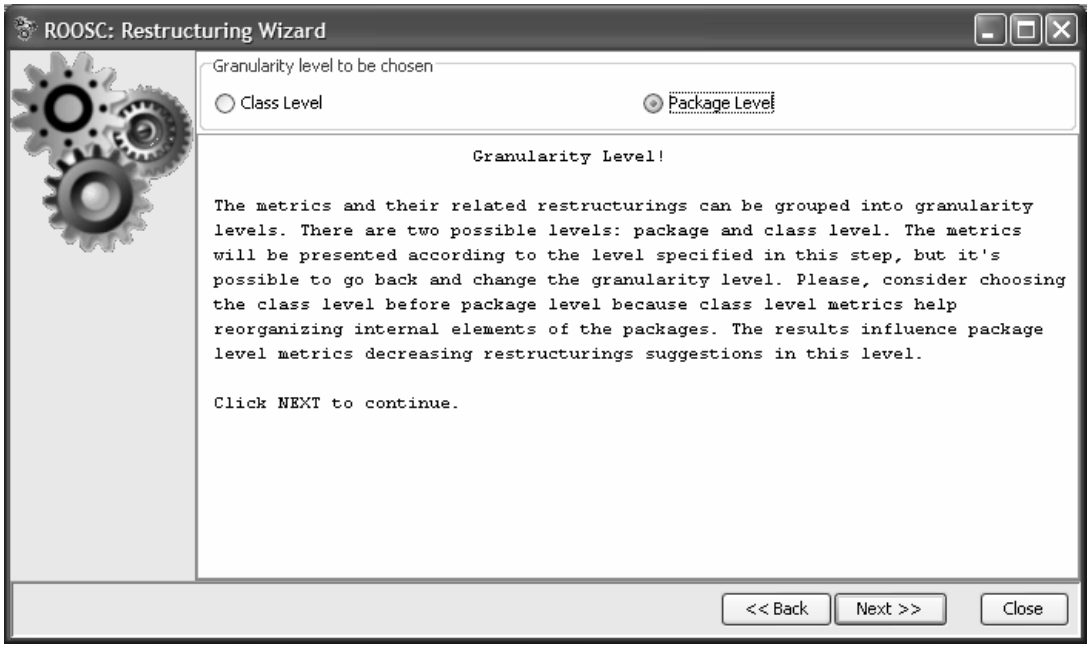


Figura 4.7: Tela de seleção do nível de granularidade.

O próximo passo é a realização da coleta de métricas, que é feita automaticamente, e a geração de sugestões de reestruturações, que também é automatizada. O resultado da coleta de métricas é visualizado através de gráficos, como mostra a Figura 4.8. Cada métrica possui uma página (Aba) específica que é identificada pelo nome da métrica e sua sigla. No exemplo da Figura 4.8, são apresentadas as abas referentes às métricas: abstração (A), componentes conectados (CC) e acoplamento por componente requerido (CRC), todas descritas na Seção 3.2.1. A visualização das métricas é feita através de gráficos que apresentam os valores das métricas coletadas em colunas e os valores de referência em linha, quando não representam um intervalo de valores, caso contrário os valores de referência serão representados por faixas de valores delimitadas pelo intervalo. Na Figura 4.8, pode-se visualizar a apresentação do valor 6 para o pacote GUI, sendo representado em coluna, como descrito anteriormente. O valor de referência, neste exemplo, é igual a 1 e está representado em linha.

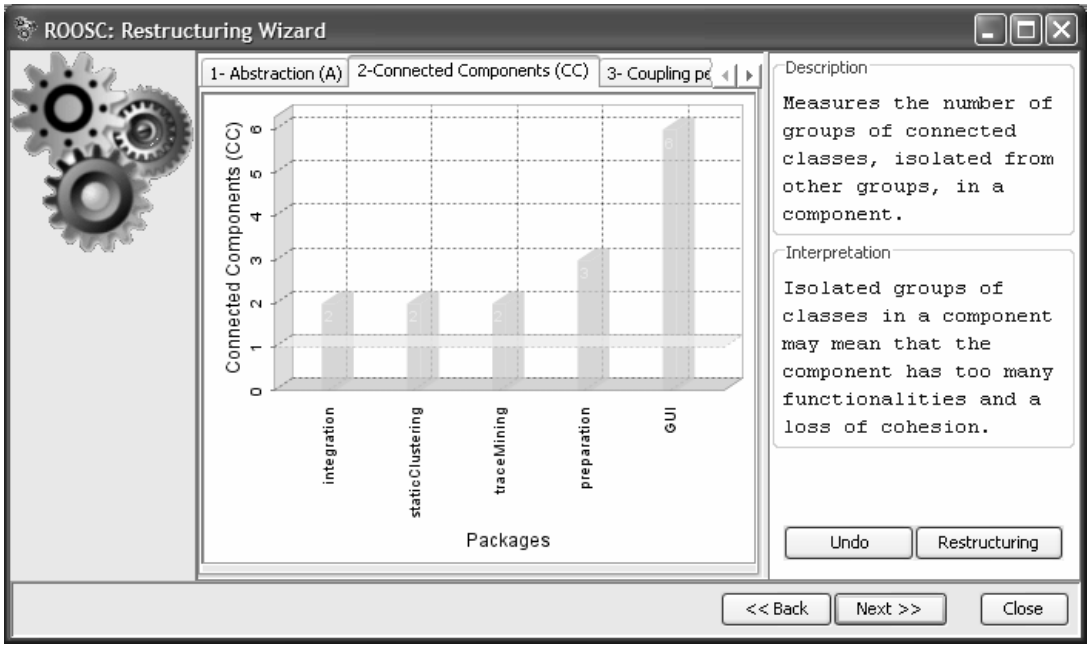


Figura 4.8: Tela de apresentação de resultados da coleta de métricas.

A tela da Figura 4.8 apresenta ainda um painel com as informações sobre a métrica (i.e. descrição e interpretação) e opções de reestruturar ou desfazer reestruturações aplicadas. A métrica mostrada nesta Figura é a métrica componentes conectados (ConnComp – *Connected Components*), que indica o número de grupos isolados de classes por pacote. O resultado desta métrica indica uma possível perda de coesão no pacote (Seção 3.2.1). No exemplo da Figura 4.8, o valor apresentado pelo

pacote GUI é igual a 6. A reestruturação proposta para este pacote é a sua divisão em outros pacotes, um para cada grupo isolado encontrado.

As sugestões de reestruturações são visualizadas por métrica através de uma opção própria, como dito anteriormente, podendo ser visualizada na Figura 4.9. A lista de sugestões tem uma visualização particular que permite a aplicação de uma ou mais sugestões de reestruturação a partir de uma mesma métrica. Como apresentado na Figura 4.9, as sugestões de reestruturação envolvem a extração de componentes para cada pacote que apresentou valor acima do valor de referência (ex.: pacote GUI na Figura 4.8). Para compreender melhor as reestruturações sugeridas, são apresentadas algumas informações particulares de cada reestruturação (i.e. mecanismo de funcionamento e forte recomendação ou não de utilização para a aderência aos princípios de DBC), como descrito na Seção 3.2.2. As reestruturações sugeridas na Figura 4.9 são fortemente recomendadas para que os componentes possuam funcionalidades específicas. Como dito anteriormente, o pacote GUI apresentou o valor 6 para a métrica ConnComp e, por isso, é sugerido que cada grupo isolado de classes seja movido para um novo pacote. Um exemplo de sugestão está na Figura 4.9, onde é sugerida a criação de um novo pacote a partir do pacote GUI. Este novo pacote deve conter as classes *Row*, *TableSorter*, *TableModelHandler*, *MouseHandler*, *Arrow*, *SortableHeaderRenderer* e *Directive* e, consequentemente, as funcionalidades providas por estas classes estarão isoladas neste novo pacote. As classes citadas formam um dos seis grupos isolados de classes que estão presentes no pacote GUI.

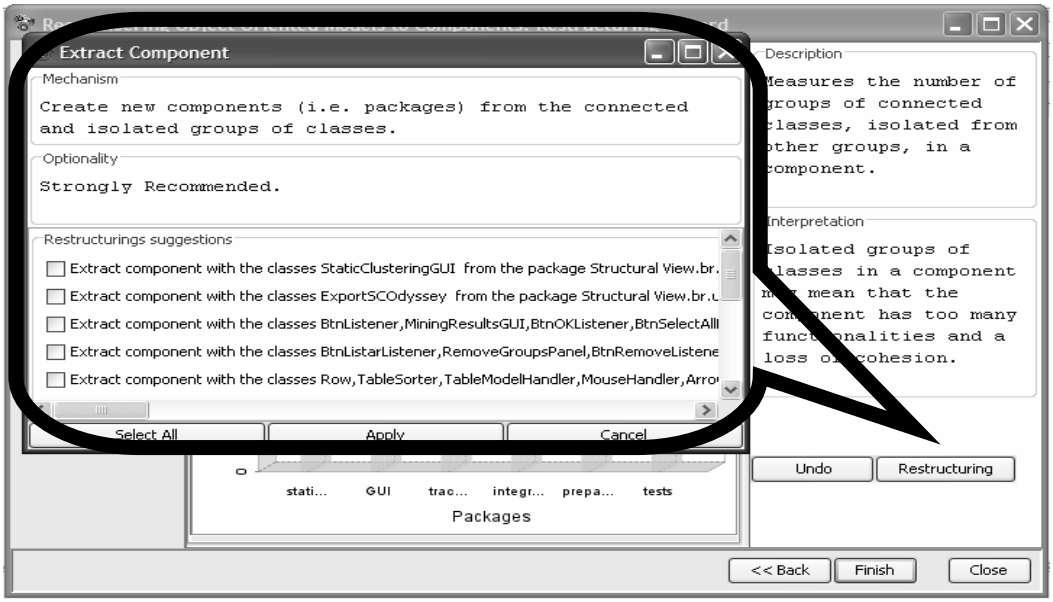


Figura 4.9: **Tela de aplicação de sugestões de reestruturações.**

Toda reestruturação aplicada pode ser desfeita em qualquer momento de reestruturação. Para isto, a ferramenta ORC prove uma interface própria como mostra a Figura 4.10. Neste momento, o engenheiro de software deve escolher o ponto até o qual deseja retornar e todas as reestruturações realizadas após aquele ponto deverão ser desfeitas também. No exemplo da Figura 4.10, selecionou-se a reestruturação que gerou um pacote com as classes/interfaces: *IWriteLog*, *WriteLog*, *WriteLogAR* e *WriteLogMUW* para ser desfeita, mas havia uma reestruturação que foi aplicada antes desta que também será desfeita por conseqüência desta ordem. Deste modo, estas duas reestruturações aparecem em destaque na Figura. Para confirmar a ação, o engenheiro de software deve selecionar a opção *Undo*.

Ao final da reestruturação é preciso verificar o modelo reestruturado. Esta verificação é baseada no *checklist* apresentado na Seção 3.2.3. A Figura 4.11 apresenta esta tela de verificação, onde a primeira opção possibilita que o engenheiro de software recupere todas as ligações entre classes/interfaces que possam ter sido perdidas. Esta perda pode ser ocasionada pelo engenheiro ter a liberdade de alterar o modelo.

A segunda opção da tela de verificação da Figura 4.11 permite ao engenheiro renomear os pacotes resultantes, pois os mesmos podem não possuir nomes adequados às funcionalidades prestadas pelas classes que contém. A terceira e última opção permite que o engenheiro retorne a reestruturar o modelo caso ainda haja reestruturações fortemente recomendadas (mover hierarquia e extrair componentes), motivando-o pelos princípios de DBC que não devem ser feridos.

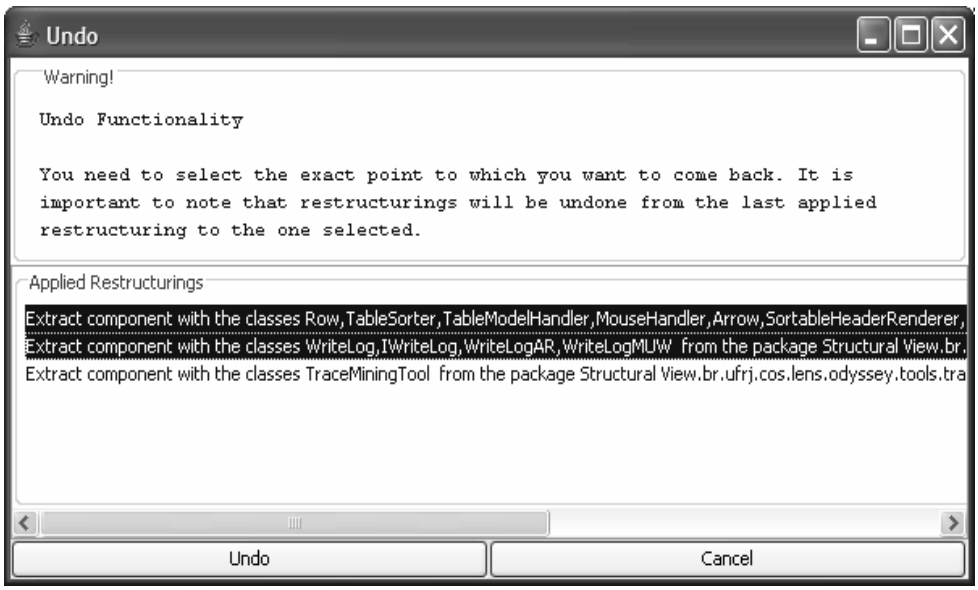


Figura 4.10: Tela para desfazer as reestruturações aplicadas na ORC.

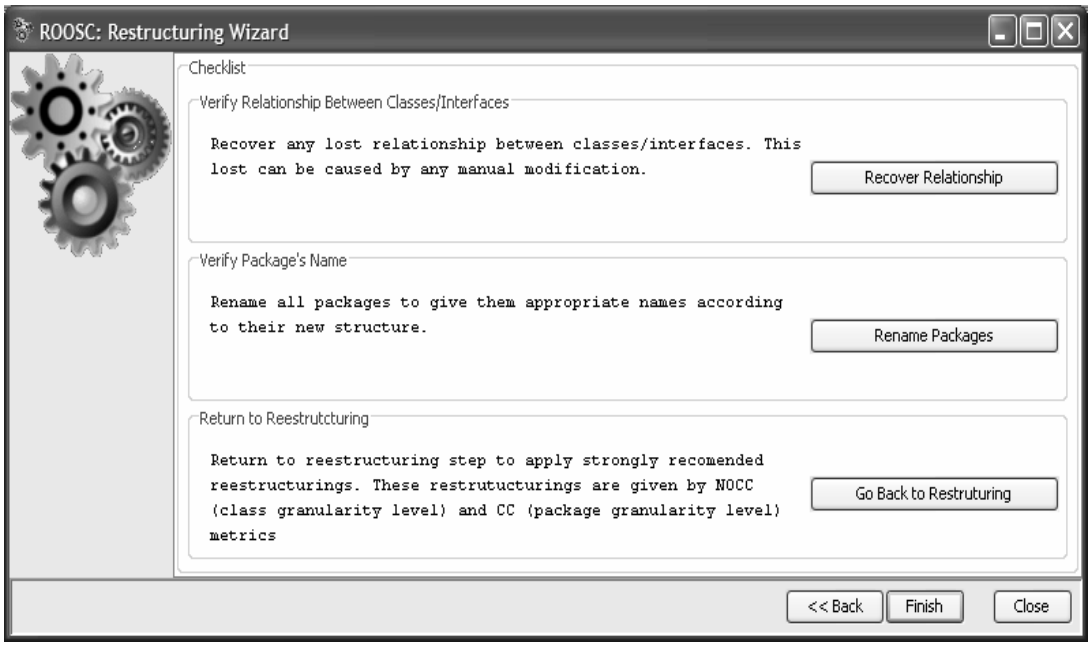


Figura 4.11: Tela de verificação do modelo reestruturado.

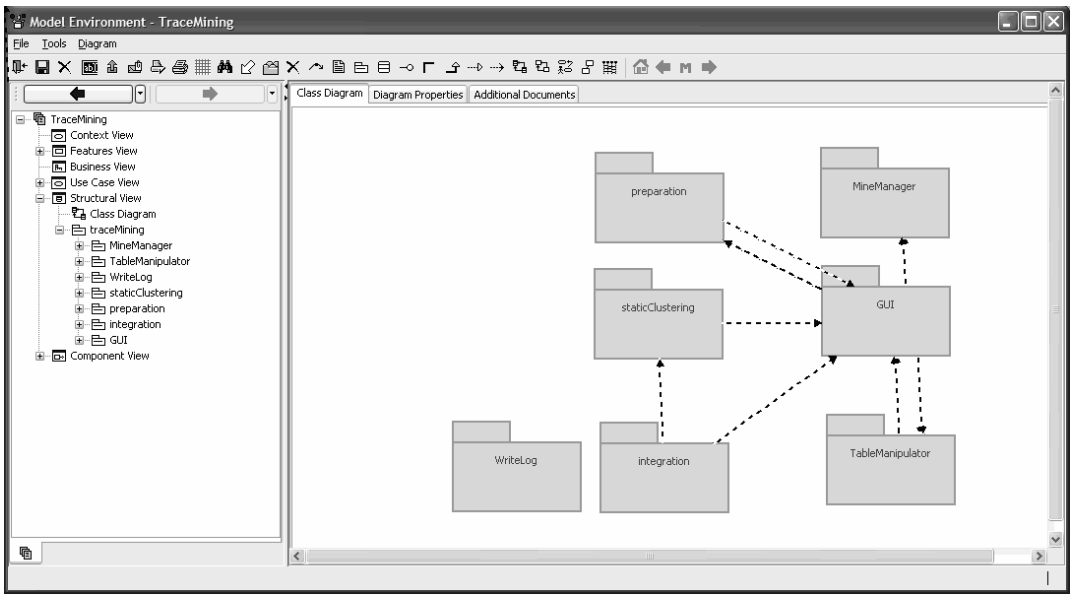


Figura 4.12: Modelo de pacotes depois da reestruturação.

Na Figura 4.1, encontra-se o modelo do sistema alvo antes da aplicação da estratégia de reestruturação. O modelo resultante após a aplicação da estratégia pode ser visualizado no próprio ambiente Odyssey, assim como o modelo inicial. O modelo resultante é apresentado na Figura 4.12 e possui algumas melhorias como: o isolamento de algumas funcionalidades, indicado pela métrica ConnComp, que resultou nos pacotes *TableManipulator* e *Writelog* (Figura 4.12), os quais poderão ser mais facilmente reutilizados; e pacotes mais coesos como o pacote GUI, que englobava a função de

manipulação de tabela que foi extraída para o pacote *TableManipulator*, e o pacote *Peparation*, que englobava a funcionalidade de escrita de *log* que foi extraída para o pacote *WriteLog*.

Após a reestruturação, o projeto do sistema alvo é utilizado para gerar os componentes propriamente, servindo como um dos artefatos de entrada para a estratégia de geração de componentes e interfaces. A Seção 4.4 apresenta a ferramenta GenComp de apoio à esta estratégia, que foi descrita em detalhes na Seção 3.3.

4.4 GenComp – *Generating Components*

A ferramenta GenComp (*Generating Components*) recebe como entrada o modelo de projeto do sistema alvo reestruturado apresentado na Figura 4.12. O modelo reestruturado é recuperado pela ferramenta GenComp diretamente no ambiente Odyssey, como mostra a Figura 4.13, e a partir deste momento, este modelo é passado por um conjunto de passos até que dê origem aos componentes e interfaces propriamente. De acordo com a Figura 4.13, além do modelo reestruturado, a GenComp utiliza como entrada um conjunto de cenários de caso de uso, que são definidos no próprio ambiente Odyssey, aplicando as heurísticas definidas em (VASCONCELOS, 2007), e os seus respectivos rastros de execução, os quais serão informados pelo engenheiro de software. Estes rastros podem ser obtidos por alguma ferramenta que monitore a execução de cenários de casos de uso e gere rastros de execução em XML, como as ferramentas Tracer (CEPEDA e VASCONCELOS, 2006; VASCONCELOS, 2007) e JProfile (JPROFILE, 2008). Além disto, o arquivo XML gerado deve conter elementos com algum atributo, que seja denominado *class*, especificando cada classe que está sendo executada.

Assim como a ORC, esta ferramenta foi implementada na linguagem Java e é um *plugin* do ambiente Odyssey. Além disto, ela é distribuída junto à ferramenta ORC. Como mostra a Figura 4.13, a ferramenta GenComp possui quatro módulos implementados: um módulo analisador de componentes via análise estática, que gera as sugestões para formação de componentes a partir dos pacotes do modelo reestruturado no Odyssey; um gerador de componentes e interfaces, que gera os componentes a partir das sugestões selecionadas e suas interfaces; um analisador de componentes via análise dinâmica, que gera as sugestões de formação de componentes de sistema com base nos cenários de caso de uso, e; por fim, um gerador de componentes e interfaces via análise dinâmica, que se baseia nas sugestões de componentes de sistema selecionadas pelo

engenheiro de software e nos rastros de execução dos cenários de caso de uso.

A ferramenta foi implementada com a adição de dois pacotes ao total de 6 pacotes criados inicialmente para a ferramenta ORC. A arquitetura completa das ferramentas pode ser visualizada na Figura 4.14. Os pacotes adicionados possuem os seguintes propósitos: pacote GenComp que agrupa classes específicas para a ferramenta GenComp que tratam a lógica de funcionamento e as estruturas necessárias para o apoio à geração de componentes e; seu pacote interno Gui, que agrupa classes de interface gráfica específicas para a estratégia de geração de componentes, como as telas da ferramenta apresentadas na Seção 4.4.1 .

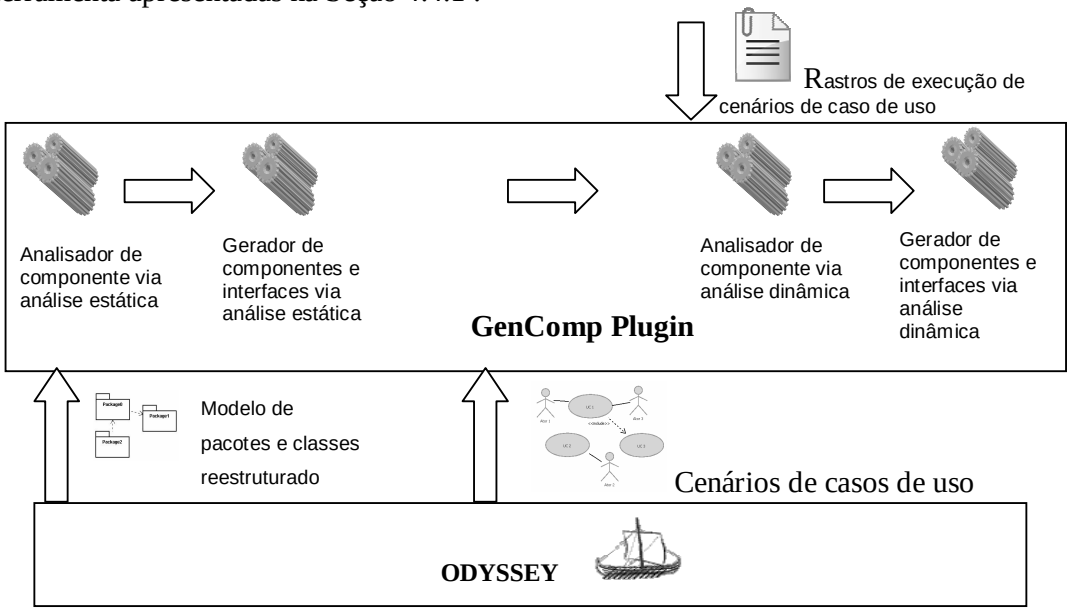


Figura 4.13: Integração da ferramenta GenComp ao ambiente Odyssey.

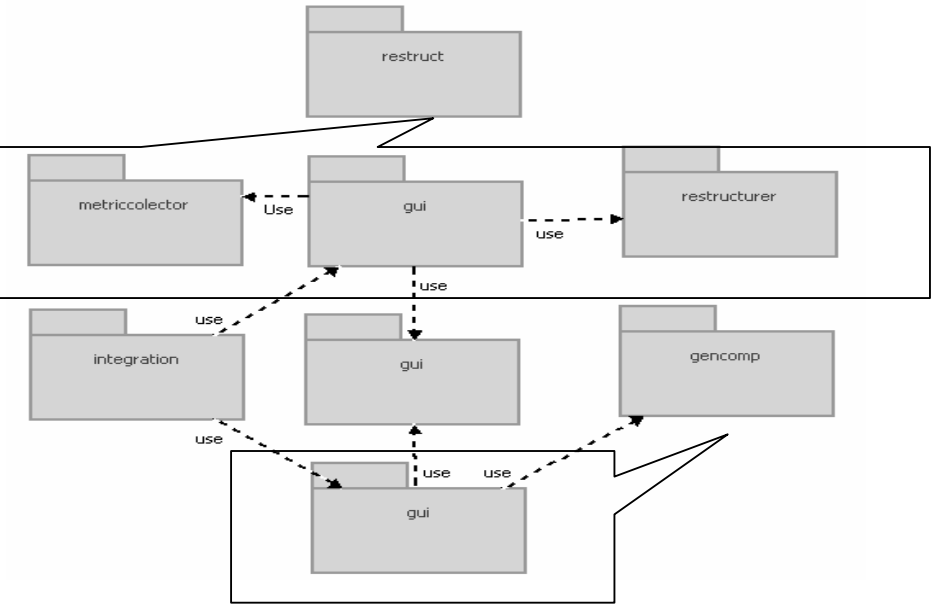


Figura 4.14: Visão dos elementos arquiteturais das ferramentas ORC e GenComp.

A GenComp, assim como a ORC, possui um formato de *wizard*. Deste modo, cada tela é um passo para a utilização da estratégia de geração de componentes e interfaces. Nesta Seção, cada tela do *wizard* é mostrada durante a apresentação do exemplo de utilização desta ferramenta.

4.4.1 Exemplo de Utilização

A fim de ilustrar o uso da ferramenta GenComp, são mostrados exemplos com o mesmo sistema alvo apresentado para mostrar as funcionalidades da ferramenta ORC. A estratégia de geração de componentes e interfaces apoiada pela GenComp é dividida em duas etapas principais: a geração de componentes e interfaces com base na análise de elementos estáticos (pacotes) ; e a geração de componentes e interfaces com base na análise de informações dinâmicas (cenários de caso de uso e rastros de execução).

4.4.1.1 Geração de Componentes e Interfaces via Análise Estática

Inicialmente, o engenheiro de software é informado sobre o objetivo da estratégia de geração de componentes e interfaces que a ferramenta apóia, como mostra a Figura 4.15. Além dessa informação, é provida uma visão geral dos passos da estratégia.

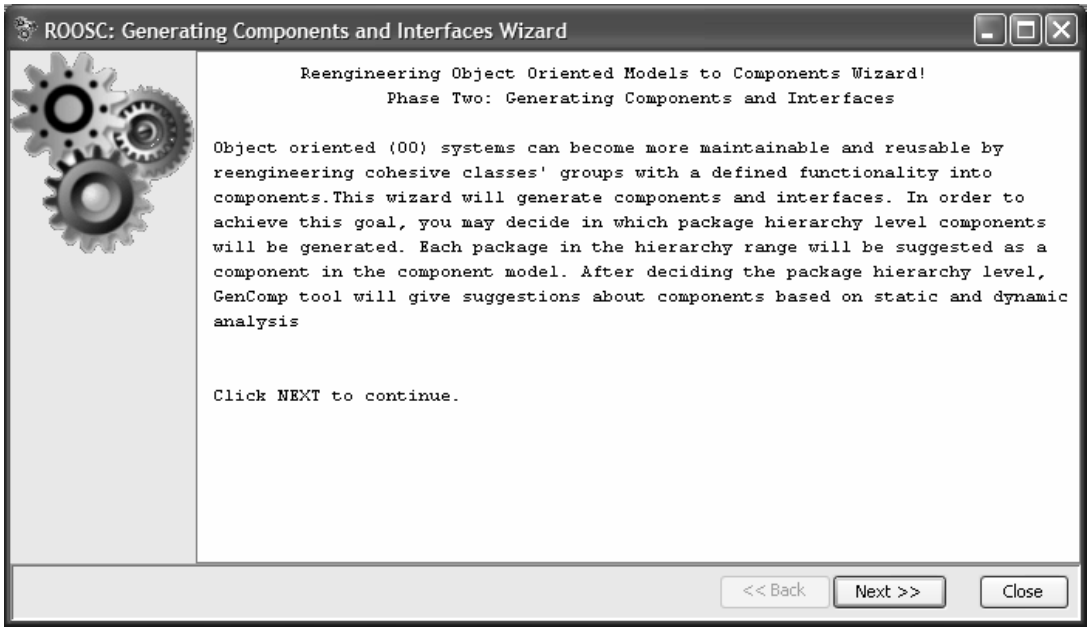


Figura 4.15: Tela inicial da ferramenta GenComp.

Como descrito anteriormente, A ferramenta GenComp utiliza, como primeiro artefato de entrada, o modelo de projeto do sistema alvo que foi reestruturado. Este modelo reestruturado possui um conjunto de pacotes candidatos a componentes. Por sua

vez, estes pacotes podem estar em diferentes níveis hierárquicos, onde pacotes são agrupados por outros pacotes. Como o projeto interno dos componentes gerados nesta estratégia é OO, este pode conter classes, interfaces e até mesmo outros pacotes. Neste contexto, o segundo passo do *wizard* possibilita a definição da faixa de níveis hierárquicos que deve ser considerada na geração de sugestões para componentes baseados nos pacotes reestruturados. Deste modo, é importante ressaltar que a faixa de níveis hierárquicos é flexível, permitindo que o engenheiro de software considere os pacotes, candidatos a componentes, que pareçam mais relevantes.

A Figura 4.16 mostra a tela de seleção da faixa de nível hierárquico, na ferramenta GenComp, para o sistema alvo TraceMining. No exemplo desta Figura, o engenheiro de software considera somente o maior nível hierárquico, por isso, o nível inicial é igual a 2, assim como o nível final. Este é o nível hierárquico que contém os pacotes que sofreram alterações durante a reestruturação, enquanto o nível acima apenas agrupa os pacotes reestruturados com o propósito de organizá-los em um pacote que represente o sistema alvo, a ferramenta TraceMining.

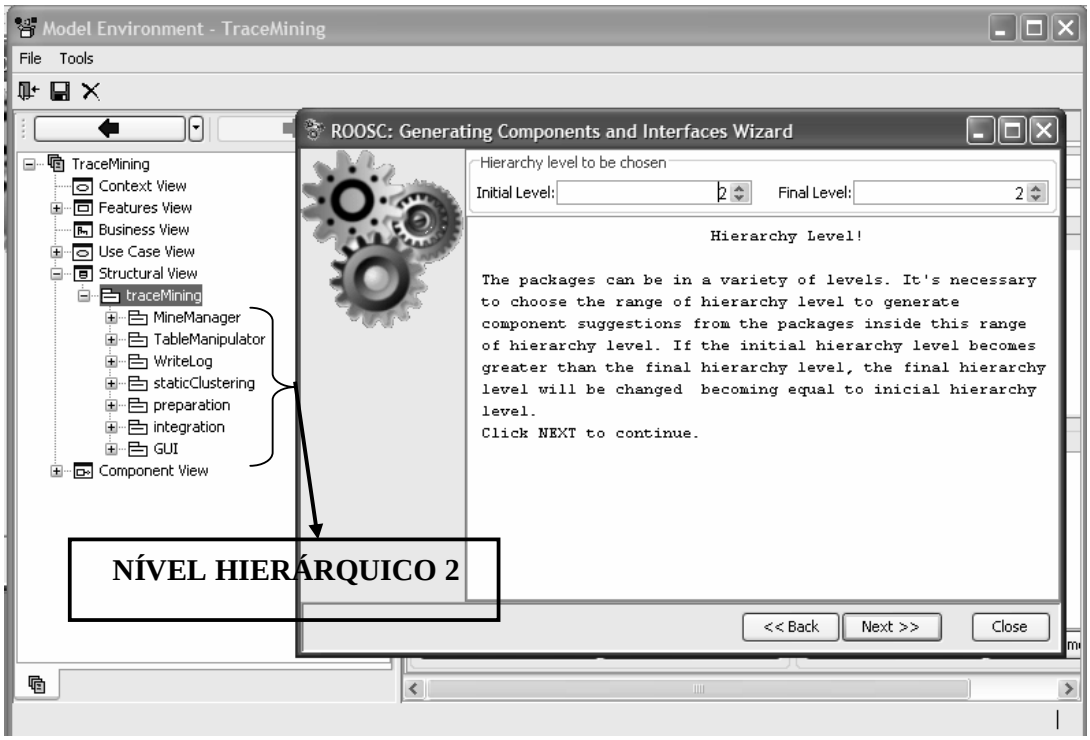


Figura 4.16: Tela de seleção de faixa de nível hierárquico.

Após a seleção da faixa de nível hierárquico (Figura 4.16), a ferramenta GenComp gera um conjunto de sugestões de componentes, baseando-se no modelo de projeto reestruturado da Figura 4.12. Deste modo, cada pacote, que tenha o seu nível

hierárquico dentro da faixa de níveis especificada pelo engenheiro, é sugerido como um novo componente.

De acordo com a faixa de nível hierárquico para o sistema alvo (nível inicial e nível final igual a 2) foram geradas as sugestões presentes na Figura 4.17. No exemplo desta Figura, existe uma sugestão que indica a criação do componente GUI. Esta sugestão é motivada pela existência do pacote GUI no modelo de projeto reestruturado. Este pacote foi considerado por estar em um nível hierárquico pertencente a faixa de níveis especificada. Para escolher as sugestões, o engenheiro seleciona todas as sugestões que deseja aplicar e seleciona a opção *Next*. Ao avançar para o próximo passo, a GenComp gera os componentes de acordo com as sugestões selecionadas e gera as suas respectivas interfaces providas e requeridas.

As interfaces geradas através de análise estática são formadas a partir dos relacionamentos entre classes pertencentes ao projeto interno dos componentes gerados. Deste modo, um componente que possua classes dependentes de outras classes que estão localizadas em outros componentes faz com que seja gerada uma interface provida no componente requerido com todas as operações públicas presentes nas classes requeridas, como a interface IPreparation_GUI da Figura 4.18, provida pelo componente *Preparation* para atender as necessidades do componente GUI.

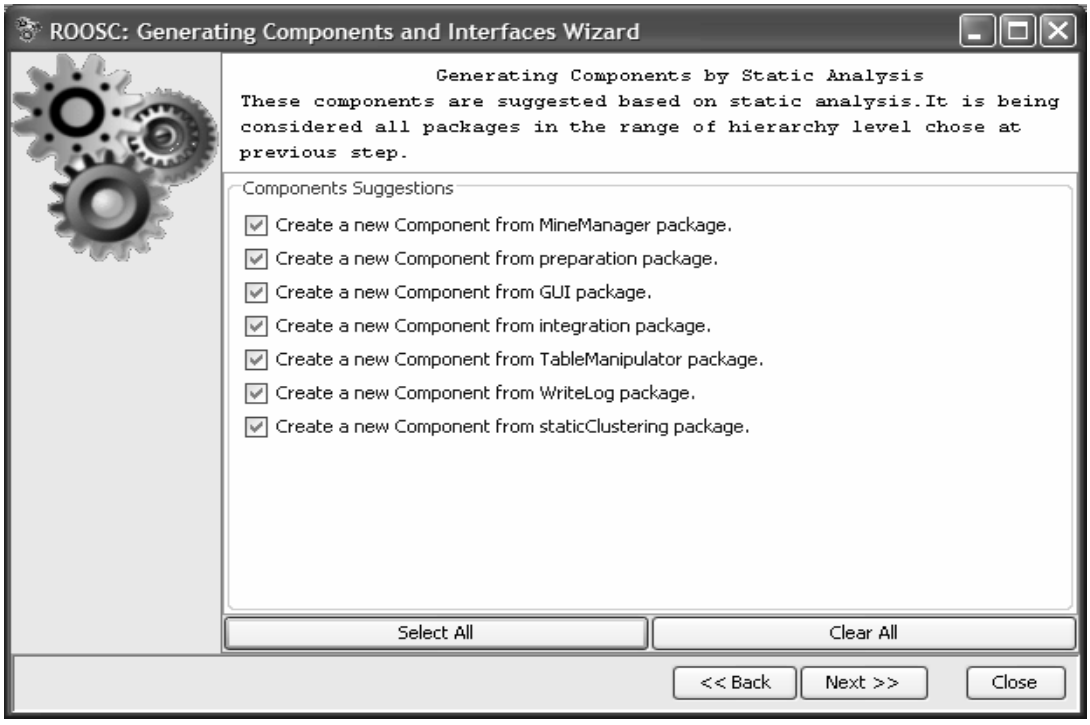


Figura 4.17: Sugestões para geração de componentes com base em análise estática.

Esta mesma interface torna-se uma interface requerida pelo componente que possui as classes que dependem das demais. No exemplo da Figura 4.18, a interface IPreparation_GUI é requerida pelo componente GUI. Esta mesma Figura 4.18 mostra o modelo de componentes após a aplicação das sugestões selecionadas na Figura 4.17. Como todas as sugestões foram selecionadas, cada um dos pacotes do modelo de projeto reestruturado (Figura 4.12) dá origem a um componente, respectivamente.

As operações das interfaces são nomeadas pela junção do nome da operação com o nome de sua classe de origem. Deste modo, permite-se que operações que possuam a mesma assinatura, mas estejam em classes distintas, possam ser expostas em uma mesma interface.

4.4.1.2 Geração de Componentes e Interfaces via Análise Dinâmica

Após a geração dos componentes via análise estática, é realizada a geração dos componentes via análise dinâmica. Nesta etapa, são utilizados os cenários de casos de uso da aplicação e seus respectivos rastros de execução, onde cada rastro representa um cenário concreto (VASCONCELOS, 2007).

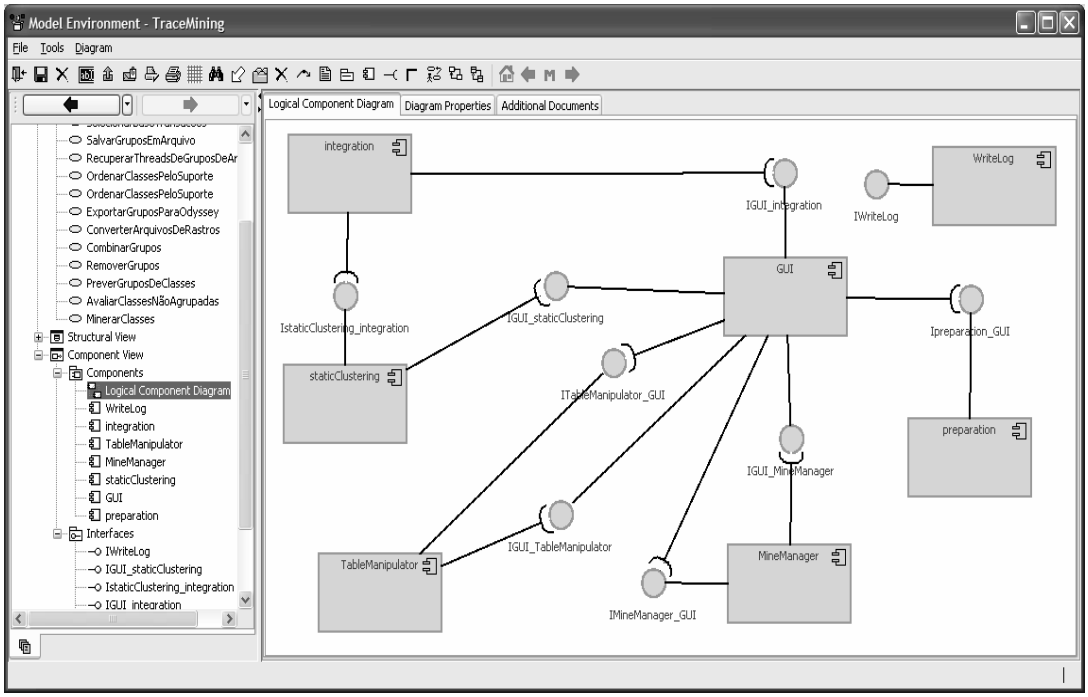


Figura 4.18: Modelo de componentes

Como descrito no Capítulo 3, os cenários compõem casos de uso, podendo ser agrupados. Na Figura 4.19, são apresentados os cenários selecionados pelo engenheiro

de software para o sistema alvo TraceMining. A escolha destes cenários foi realizada com base nas diretrizes apresentadas na Seção 3.3 .

Figura 4.19: Cenários de caso de uso definidos no ambiente Odyssey.

Estes cenários são obtidos diretamente do ambiente Odyssey e listados na tela de agrupamento de cenários, como mostra a Figura 4.20. Nesta mesma Figura, é possível visualizar um grupo denominado MinerarEAgruparClasses e seus respectivos cenários (MinerarClasses, AvaliarClassesNãoAgrupadas, PreverGruposDeClasses, RemoverGrupos, ExportarGruposParaOdyssey e SalvarGruposEmArquivo). Este grupo foi criado porque a funcionalidade de mineração de classes é sempre utilizada junto a funcionalidade de agrupamento destas classes, no contexto do sistema alvo.

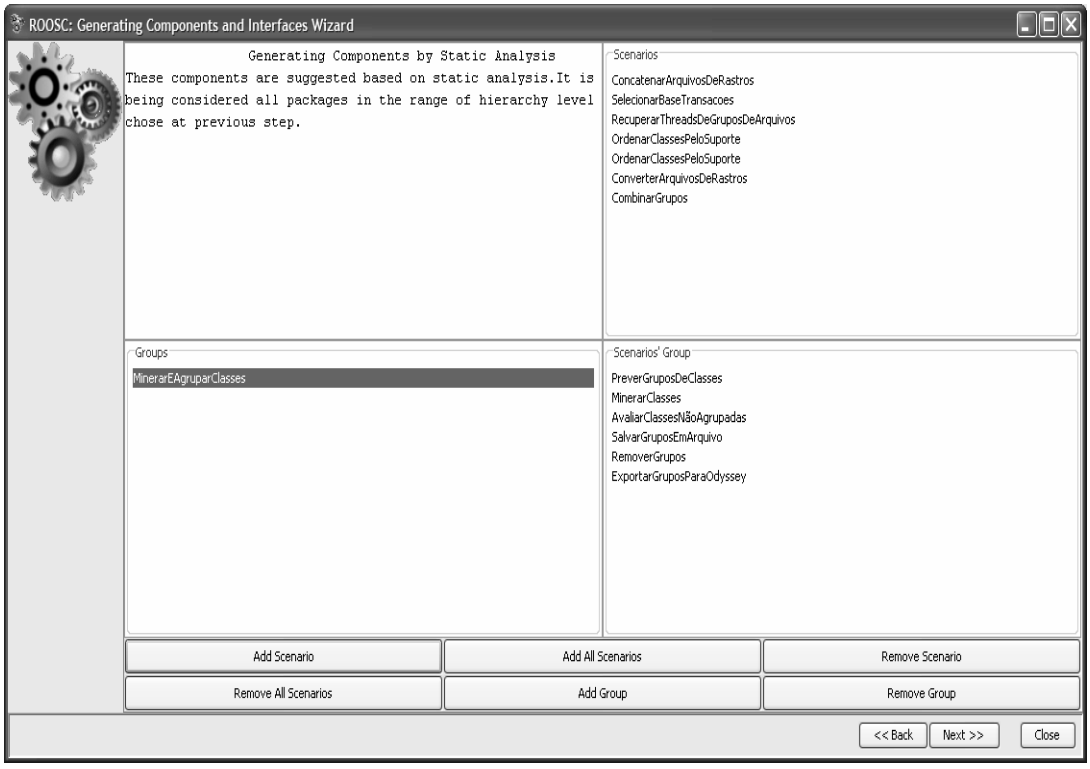


Figura 4.20: Tela de agrupamento de cenários.

Após o passo de agrupamento dos cenários, a ferramenta GenComp gera sugestões de componentes de sistema com base nestes cenários ou grupos de cenários (i.e. casos de uso). No exemplo da TraceMining, são geradas as sugestões apresentadas na Figura 4.21. Por exemplo, uma das sugestões da lista é a criação de um componente de sistema denominado MinerarEAgruparClasses, onde este componente trata todos os cenários do grupo MinerarEAgruparClasses.

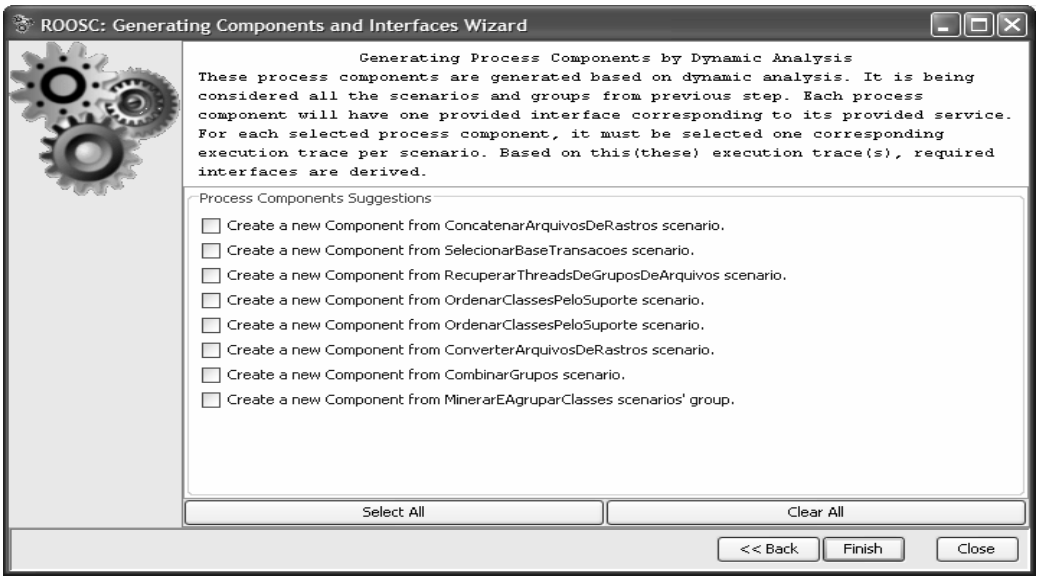


Figura 4.21: Sugestões para criação de componentes de sistema a partir dos cenários definidos para a TraceMining.

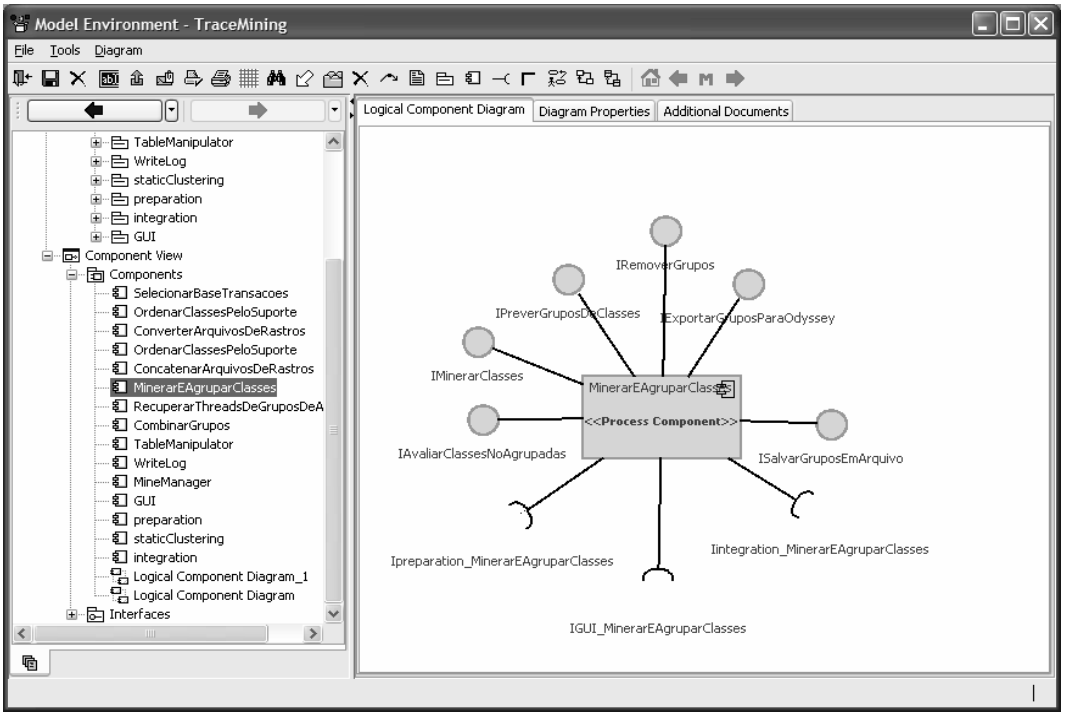


Figura 4.22: Componente de sistema MinerarEAfruparClasses e suas interfaces.

Após selecionar as sugestões e finalizar o *wizard*, são gerados os componentes de sistema e suas respectivas interfaces providas e requeridas. As interfaces providas dos componentes de sistema são formadas baseando-se em cada cenário de caso de uso que o componente de sistema representa e possuem uma única operação para a execução do cenário.

As interfaces requeridas pelo componente de sistema MinerarEAgruparClasses são determinadas pelos rastros de execução dos cenários que são agrupados por este componente. Como descrito anteriormente, estes rastros podem ser obtidos através de alguma ferramenta de extração de rastros de execução. O rastro de execução do cenário de caso de uso PreverGrupos é apresentado na Figura 4.23, onde é destacada a classe TraceMiningGUI que motivou a interface requerida IGUI_ MinerarEAgruparClasses que por sua vez é provida pelo componente GUI, gerado através da análise estática. O algoritmo que procura as classes participantes do caso de uso nos rastros, procura por nós que contenham um atributo denominado *class* e verifica se a classe, indicada neste atributo, existe no modelo de projeto reestruturado do sistema alvo.

Deste modo, as interfaces requeridas dos componentes de sistema estão sempre relacionadas a algum componente gerado pela análise estática, pois eles possuem as classes que participam da realização destes cenários. Caso o componente que possua a classe necessária não tenha sido mapeado, não será gerada a interface requerida.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
- <tree type="Call Tree" threadSelection="All thread groups" threadStatus="Runnable" aggregationLevel="Methods">
  - <node leaf="false" class="java.awt.EventDispatchThread" methodName="run" methodSignature="()V"
      time="146507" count="1" inherentTime="126146" lineNumber="-1" percent="100,0">
    + <node leaf="false" class="br.ufrj.cos.lens.odyssey.tools.traceMining.GUI.TraceMiningGUI$11"
        methodName="actionPerformed" methodSignature="(Ljava/awt/event/ActionEvent;)V" time="20013"
        count="1" inherentTime="2350" lineNumber="-1" percent="13,7">
      <node leaf="true" class="br.ufrj.cos.lens.odyssey.tools.traceMining.GUI.TraceMiningGUI"
        methodName="windowActivated" methodSignature="(Ljava/awt/event/WindowEvent;)V" time="259"
        count="1" inherentTime="259" lineNumber="-1" percent="0,2" />
      <node leaf="false" class="br.ufrj.cos.lens.odyssey.tools.traceMining.GUI.TraceMiningGUI$14"
        methodName="stateChanged" methodSignature="(Ljavax/swing/event/ChangeEvent;)V" time="78"
        count="2" inherentTime="62" lineNumber="-1" percent="0,1" />
      <node leaf="true" class="br.ufrj.cos.lens.odyssey.tools.traceMining.GUI.TraceMiningGUI"
        methodName="windowDeactivated" methodSignature="(Ljava/awt/event/WindowEvent;)V" time="11"
        count="1" inherentTime="11" lineNumber="-1" percent="0,0" />
    </node>
  </tree>
```

Figura 4.23: Rastro de execução do cenário de caso de uso PreverGrupos.

A partir deste momento, os componentes e suas interfaces estão gerados e o engenheiro pode visualiza-los no próprio ambiente Odyssey.

4.5 Considerações Finais

Este Capítulo apresentou o ferramental que suporta as atividades da abordagem de reengenharia de software OO para componentes, ROOSC, proposta nesta dissertação. O ferramental é composto pelas ferramentas ORC (*Object Restructuring to Components*) e GenComp (*Generating Components*). É importante ressaltar que este ferramental de apoio foi utilizado nos estudos de caso, recebendo *feedback* que foi utilizado em seu refinamento.

A ORC visa apoiar a reestruturação do modelo de projeto do sistema alvo para obter pacotes candidatos a componentes, conforme os princípios de DBC adotados nesta dissertação. É importante ressaltar que a ORC apresenta flexibilidade com relação a linguagem na qual o sistema alvo está implementado, uma vez que o ambiente Odyssey, ao qual está integrada, lê arquivos XMI. Porém a versão do XMI e UML precisa ser compatível com as versões adotadas pelo ambiente Odyssey, i.e. a versão 1.2 da especificação XMI e a versão 1.4 do metamodelo da UML.

A ORC pode ser utilizada para apoiar a reutilização, uma vez que ela reestrutura os pacotes do projeto do sistema alvo deixando-os mais coesos e menos acoplados de modo que os mesmos possam agrupar funcionalidades para serem reutilizadas em outros contextos. A ORC também apóia a manutenção, uma vez que visa agrupar as classes que tendem a ser modificadas juntas.

A GenComp visa apoiar a geração de componentes e interfaces a partir do modelo de projeto reestruturado do sistema alvo e dos cenários de caso de uso deste sistema e seus rastros de execução. A GenComp apresenta as mesmas características de flexibilidade da ORC devido a integração com o ambiente Odyssey.

A GenComp também apóia a reutilização, uma vez que os componentes gerados foram projetados para agrupar uma funcionalidade específica para ser reutilizada em novos sistemas. Esta ferramenta também apóia a manutenção, uma vez que as modificações realizadas nos componentes não afetam as demais partes do sistema, pois a comunicação com os componentes é feita através de interfaces.

No Capítulo 2, algumas abordagens de reengenharia para componentes foram pesquisadas de tal modo que alguns critérios fossem ser analisados, como apresentado na Tabela 3.5. Critérios estes que foram estabelecidos para esta dissertação de acordo com a abordagem que está sendo proposta. Neste Capítulo, dois destes critérios são destacados, o ferramental de apoio e a integração deste ferramental a um ambiente de

desenvolvimento de software (ADS). Onde a abordagem ROOSC possui um ferramental de apoio para suportar as suas atividades e minimizar o esforço do engenheiro de software. Além disto, este ferramental está integrado a um ADS, facilitando a reutilização dos componentes gerados em um novo desenvolvimento.

Capítulo 5 Avaliação da Abordagem ROOSC

5.1 Introdução

Este capítulo apresenta os estudos experimentais que foram realizados no contexto desta dissertação. Em (TRAVASSOS, *et al.*, 2002), os autores destacam a importância da experimentação e validação para novos métodos técnicas, linguagens e ferramentas, onde a experimentação representa o centro do processo científico, consistindo na única forma de se verificar uma nova teoria. Segundo WOHLIN *et al*. (2000), existem três principais estratégias de experimentação, que se diferenciam pelo propósito da avaliação e das condições para o estudo experimental (ex: disponibilidade de recursos):

- **Investigação (*survey*)**: representa uma investigação realizada em retrospectiva, quando, por exemplo, uma técnica ou ferramenta foi utilizada por um tempo e se deseja avaliá-la sob algum aspecto. As principais ferramentas utilizadas para apoiar a investigação são entrevistas e questionários.
- **Estudos de Caso**: são utilizados para monitorar projetos, atividades ou atribuições. Dados são coletados para um propósito específico através do estudo e análises estatísticas podem ser conduzidas com base nos dados coletados. Normalmente, visa rastrear um atributo específico ou estabelecer relacionamentos entre diferentes atributos. O nível de controle é menor do que em um experimento.
- **Experimentos**: são normalmente conduzidos em laboratório, o que provê um alto nível de controle sobre o processo e os fatores ou variáveis que afetam o estudo. O objetivo do experimento é manipular uma ou mais variáveis, mantendo as demais sob controle. Experimentos são apropriados para confirmar teorias, avaliar a predição dos modelos ou validar medidas. Apresenta maior facilidade de repetição.

Um experimento objetiva atingir um determinado objetivo. Segundo TRAVASSOS *et al.* (2002), os objetivos relacionados à execução de experimentos em Engenharia de Software são a caracterização, avaliação, previsão, controle e melhoria a

respeito de produtos, processos, recursos, modelos, teorias entre outros. A importância e o esforço aumentam de um experimento com o objetivo de "caracterização" a um experimento com o objetivo de "melhoria".

A fim de avaliar a viabilidade da aplicação da abordagem de reengenharia de software orientado a objeto (OO) para componentes ROOSC no apoio à manutenção e reutilização de software, realizaram-se dois estudos como prova de conceito. Estas provas de conceito foram organizadas como estudos de observação que são caracterizados por serem aplicados em um ambiente onde os indivíduos desempenham tarefas enquanto são observados pelos pesquisadores. Enquanto estudos de viabilidade costumam objetivar a caracterização de uma tecnologia, verificando se ela realmente faz o que se propõe a fazer e se é viável continuar a despender recursos para desenvolvê-la.

Os estudos, descritos neste capítulo, são desenvolvidos segundo o conjunto de etapas proposto em (WOHLIN, *et al.*, 2000) para um processo de experimentação. Exceto pela etapa de empacotamento do experimento.

A primeira etapa compreende a definição do estudo experimental. Nesta etapa, os autores sugerem o uso da estrutura proposta por (BRIAND et al., 1996), a qual consiste dos seguintes elementos:

Analisar <<objeto de estudo>>

Com o propósito de <<qual é a intenção>>

Com respeito a <<qual o efeito a ser estudado>>

Do ponto de vista <<da perspectiva de observação>>

No contexto de <<onde o estudo será conduzido>>

Na segunda etapa, deve ser feito o planejamento do estudo experimental, que é composto pelas seguintes atividades:

- ⇒ Seleção do contexto onde o experimento será desenvolvido;
- ⇒ Formulação das hipóteses do estudo;
- ⇒ Seleção das variáveis dependentes (a(s) resposta(s) do experimento) e independentes (servem como entrada para o experimento);
- ⇒ Seleção dos participantes;
- ⇒ Plano do estudo, contendo o treinamento, planejamento das atividades, e as diferentes formas de participação (e.g. usando uma tecnologia, ad-hoc);
- ⇒ Instrumentação, ou seja, o que será utilizado para avaliar o objeto de estudo;
- ⇒ Avaliação da validade do estudo.

Após o planejamento, ocorrem as etapas de execução do experimento, análise e interpretação dos resultados obtidos, e apresentação e empacotamento do experimento.

Neste contexto, este capítulo está organizado da seguinte forma: na Seção 5.2 , é apresentado o estudo de viabilidade da estratégia de reestruturação de modelos OO visando à obtenção de candidatos a componentes; a Seção 5.3 apresenta o estudo de viabilidade da estratégia de geração de componentes e interfaces, a partir de um modelo OO reestruturado conforme alguns princípios de DBC; por fim, a Seção 5.4 apresenta as considerações finais deste capítulo.

5.2 Estudo de Viabilidade da Reestruturação de Modelos OO

5.2.1 Objetivo Global

O propósito deste estudo é verificar a capacidade da estratégia de reestruturação de modelos OO baseada em métricas em apoiar a reestruturação de um modelo OO, onde os agrupamentos de classes (i.e. pacotes) representem bons candidatos a componentes sob o aspecto semântico e sob o aspecto dos princípios de DBC encontrados na literatura e apresentados no Capítulo 2.

5.2.2 Objetivo do Estudo

Analisar a estratégia de reestruturação de modelos OO baseada em métricas

Com o propósito de caracterizar

Com respeito à capacidade da estratégia de reestruturação proposta em gerar agrupamentos candidatos a componentes

Do ponto de vista do engenheiro de software

No contexto do engenheiro de software com conhecimento do domínio da ferramenta ArchTrace (MURTA, *et al.*, 2006)

5.2.3 Planejamento do Estudo

5.2.3.1 Questões

A fim de avaliar os resultados produzidos pela estratégia de reestruturação da abordagem ROOSC, o paradigma GQM (*Goal Question Metric*) de BASILI *et al.* (1994) é adotado. Tendo como base os princípios de DBC adotados nesta dissertação e o conhecimento do domínio por parte do engenheiro de software, elaboraram-se algumas questões para apoiar a avaliação dos resultados obtidos a partir da estratégia de

reestruturação:

Q1: Qual percentual de agrupamentos reestruturados possui uma funcionalidade específica e comum ao domínio, representando candidatos a componentes do domínio?

M1:

$$\textbf{PercentFuncionalPacote} = \frac{\text{NumPacoteFuncional}}{\text{NumPacoteTotal}}$$

Visando obter o percentual de agrupamentos de classes (pacotes) candidatos a componentes, que possuem uma funcionalidade específica, os seguintes valores são considerados:

NumPacoteTotal: total de agrupamentos resultantes da reestruturação, ou seja, número total de pacotes no modelo reestruturado.

NumPacoteFuncional: número de agrupamentos (pacotes) resultantes da reestruturação com uma funcionalidade específica sob o ponto de vista do engenheiro de software.

Q2: Qual o percentual de redução do número de sugestões de reestruturações fortemente recomendadas?

M2:

$$\textbf{PercentReduçãoReestruturaçõesFR} = 1 - \frac{\text{NumReestruturaçõesFRFinal}}{\text{NumReestruturaçõesFRInicial}}$$

A fim de obter o percentual de redução da quantidade de reestruturações fortemente recomendadas ao final da reestruturação, os seguintes valores são considerados:

NumReestruturaçõesFRFinal: número de sugestões de reestruturações fortemente recomendadas (FR) depois que o engenheiro de software finaliza as reestruturações.

NumReestruturaçõesFRInicial: número de sugestões de reestruturações fortemente recomendadas (FR) antes do engenheiro de software iniciar as reestruturações.

Q3: Qual o percentual de apoio das métricas para a seleção das reestruturações?

M3:

$$\textbf{PercentDominioMétrica} = \frac{\text{NumMétricasEDomínio}}{\text{NumMétricasTotal}}$$

A fim de obter o percentual de utilização do conhecimento do domínio junto às métricas, os seguintes valores são considerados:

NumMétricasEDomínio: número de métricas utilizadas junto ao conhecimento do domínio para apoiar a seleção de reestruturações.

NumMétricasTotal: número de total de métricas apresentadas pela abordagem.

5.2.3.2 Seleção do Contexto

O estudo foi realizado em ambiente acadêmico sobre uma única aplicação. Esta aplicação foi selecionada pelo fato de haver um especialista disponível para apoiar o estudo. A aplicação é uma ferramenta de evolução de rastros entre elementos arquiteturais e elementos do código fonte, desenvolvida em parceria entre a COPPE/UFRJ e a Universidade da Califórnia em Irvine (UCI), denominada ArchTrace (MURTA, *et al.*, 2006). Maiores detalhes sobre a aplicação são apresentados na Tabela 5.1.

Tabela 5.1: Características da ArchTrace.

Aplicação	Pacotes	Classes	LOC	Estilo Arquitetural
ArchTrace	19	80	4695	Modelo-Visão-Controlador (*Model-View-Controller* -MVC)

5.2.3.3 Seleção dos Participantes

A seleção do participante deve considerar a importância do engenheiro de software possuir algum conhecimento sobre o domínio da aplicação e sobre DBC. Neste estudo, conseguiu-se o engenheiro de software que desenvolveu a aplicação utilizada no estudo. Além disto, este engenheiro é também um especialista do domínio e conhece DBC, portanto ele aplicou a estratégia de reestruturação. Além da participação na aplicação da estratégia, o engenheiro de software ajudou a avaliar os agrupamentos reestruturados.

5.2.3.4 Seleção de Variáveis

As variáveis independentes deste estudo são:

- A experiência dos executores sobre DBC;
- O conhecimento do participante sobre o domínio do sistema;
- O modelo de projeto recuperado por engenharia reversa;

A variável dependente deste estudo é o modelo de projeto reestruturado resultante.

5.2.3.5 Definição das Hipóteses

Hipótese nula (H0): A utilização da estratégia de reestruturação não apóia a geração de agrupamentos de classes que sejam candidatos a componentes sob o aspecto semântico, do ponto de vista do engenheiro de software com conhecimento do domínio e sob o aspecto dos princípios de DBC.

5.2.3.6 Validade do Estudo

As seguintes ameaças à validade do estudo foram identificadas:

Validade Interna

Refere-se ao risco de um outro fator, que não o tratamento (i.e. a aplicação da abordagem ROOSC), ter influenciado nos resultados do estudo.

Esforço na Avaliação: O fato da estratégia de reestruturação envolver um grande número de métricas a serem analisadas leva a um desgaste do engenheiro de software quando a aplicação possui um grande número de elementos (i.e. pacotes e classes). Este excesso de informação pode prejudicar a tomada de decisão. A aplicação selecionada não é extensa e, por isso, não deverá ser influenciada por este risco e, além disto, a abordagem sugere uma ordem de análise das métricas.

Viés do Modelo Reestruturado: se o engenheiro de software não tiver conhecimento do domínio da aplicação e dos princípios de DBC, ele pode ter dificuldade em decidir quais reestruturações aplicar. Por isso, o engenheiro de software convidado foi o próprio autor da aplicação em questão, que possui também conhecimento sobre DBC.

Validade de Conclusão

Refere-se a habilidade de concluir corretamente os relacionamentos entre o tratamento e o resultado do experimento.

Tratamento da hipótese nula: Como o estudo experimental foi realizado com apenas um participante e este participante possui um vasto conhecimento do domínio e

de DBC, as conclusões do estudo podem ser influenciadas de tal modo que a estratégia de reestruturação pareça viável.

Validade de Construção

Refere-se aos relacionamentos entre a teoria e a conclusão, de tal modo que o tratamento reflita a causa bem e o resultado reflita o efeito bem.

Projeto do estudo: O pesquisador pode planejar o estudo visando obter os resultados esperados.

5.2.4 Operação do Estudo

5.2.4.1 Preparação

O engenheiro de software participante do estudo realiza a preparação dos dados de entrada, a fase de reestruturação do modelo de projeto do sistema, recuperado através de engenharia reversa, e a avaliação do modelo reestruturado.

Nesta etapa, o engenheiro de software recebe o material do estudo, contendo os formulários de: consentimento (Anexo A/A.1), avaliação do modelo reestruturado (Anexo A/A.2), roteiro de reestruturação (Anexo A/A.3) e caracterização (Anexo A/A.4), recebendo ainda uma apresentação presencial sobre a utilização da ferramenta ORC (*Object Restruct to Component*), que apóia a estratégia de reestruturação da abordagem ROOSC, sobre o preenchimento dos formulários e o roteiro de realização do estudo.

Durante a preparação dos dados, o engenheiro executou a aplicação para a engenharia reversa (ER) do sistema. Deste modo, a preparação dos dados envolveu a utilização da ferramenta Ares (VERONESE e NETO, 2001). Esta escolha é devido ao fato de que a aplicação foi desenvolvida em Java e a ferramenta Ares é utilizada para a realização de ER sobre sistemas desenvolvidos nesta linguagem. A utilização desta ferramenta também é motivada pela sua integração com o ambiente Odyssey, como descrito no Capítulo 4.

Durante esta etapa, um treinamento presencial foi mostrado ao engenheiro de software, como dito anteriormente, explicando a respeito da utilização da ferramenta ORC que apoiará toda a estratégia de reestruturação. Este treinamento utilizou como exemplo a reestruturação da aplicação TraceMining (VASCONCELOS, 2007). TraceMining é uma ferramenta utilizada na recuperação de arquitetura de referência através de técnicas de mineração de dados e também pode ser utilizada integrada ao

ambiente Odyssey. Esta aplicação utilizada no exemplo foi utilizada no contexto do primeiro estudo, também como exemplo de utilização. Durante a demonstração, foram mostradas todas as métricas e suas respectivas reestruturações, para que o engenheiro de software compreendesse toda a abordagem e sua aplicação utilizando o ferramental de apoio.

5.2.4.2 Operação

Uma vez preparado o artefato de entrada (i.e. modelo de projeto do sistema alvo), o engenheiro de software realizou a reestruturação do modelo de projeto da sistema ArchTrace. Esta etapa foi realizada com o apoio da ferramenta ORC descrita no Capítulo 4. A atividade de reestruturação é uma atividade semi-automatizada e requer bastante a tomada de decisão por parte do engenheiro. O modelo reestruturado foi acompanhado, visualizando-o no próprio ambiente Odyssey para posterior prosseguimento do restante do formulário de avaliação.

Toda a reestruturação foi realizada conforme o roteiro da reestruturação apresentado ao engenheiro de software. A cada ciclo de reestruturação completo, o engenheiro de software avaliou algumas questões do formulário de avaliação sobre o modelo reestruturado. Após o material ter sido preenchido, o material foi entregue para o pesquisador. Quanto às dúvidas que surgiram durante a reestruturação (ex.: a respeito da ferramenta), o engenheiro de software entrou em contato com o pesquisador, que esteve presente no local do estudo para o esclarecimento de dúvidas.

5.2.5 Análise dos Resultados Obtidos

Durante o planejamento desse estudo, foi definido como objetivo principal avaliar a viabilidade da estratégia de reestruturação que faz parte da abordagem ROOSC. Essa viabilidade, conforme os questionamentos estabelecidos, é avaliada através da análise dos valores dos percentuais estabelecidos e através da análise das questões subjetivas respondidas pelo engenheiro do sistema alvo.

É importante ressaltar que o engenheiro de software foi instruído a avaliar o modelo reestruturado sob o ponto de vista de pacotes que representem possíveis componentes, de acordo com o conhecimento do domínio do engenheiro. Convém mencionar que o engenheiro de software foi orientado a considerar todos os níveis hierárquicos dos pacotes e esta observação vale para a contagem do total de pacotes resultantes após a reestruturação.

Os pacotes reorganizados representam possíveis componentes com funcionalidades específicas do domínio?

A resposta a esta pergunta foi calculada com base na métrica descrita na questão Q1. A questão avalia os elementos reestruturados (i.e. pacotes) sob o aspecto funcional destes elementos para que eles sejam candidatos a componentes para serem reutilizados no mesmo domínio do sistema alvo. O percentual de pacotes funcionais atingido foi calculado extraindo-se o valor de "NumPacoteFuncional" da resposta da questão Q2 do formulário 1 (Anexo A/A.2), i.e. a quantidade de pacotes reestruturados que o engenheiro julga possuir uma funcionalidade específica e bem definida. O valor de "NumPacoteTotal" foi extraído da resposta da questão Q1 do formulário 1 (Anexo A/A.2), i.e. a quantidade de pacotes resultantes após a reestruturação. A Tabela 5.2 apresenta os valores obtidos e o percentual de pacotes funcionais calculado.

Tabela 5.2: Percentual de pacotes funcionais.

Sistema	NumPacoteFuncional	NumPacoteTotal	Percentual de pacotes funcionais
ArchTrace	13	30	43,33%

Qual o percentual de redução do número de reestruturações fortemente recomendadas?

A resposta a esta pergunta foi calculada com base na métrica descrita na questão Q2. A questão avalia se as reestruturações aplicadas ao projeto reduziram a necessidade de aplicar reestruturações fortemente recomendadas (FR). É importante ressaltar que estas reestruturações estão diretamente relacionadas aos princípios de DBC e à própria definição de componente adotada nesta dissertação. O percentual de redução do número de reestruturações FR atingido foi calculado extraindo-se o valor de "NumReestruturaçõesFRFinal" da resposta da questão Q4 do formulário 1 (Anexo A/A.2), i.e. a quantidade de reestruturações FR que não foram aplicadas durante a reestruturação. O valor de "NumReestruturaçõesFRFinal" foi extraído da resposta da questão Q3 do formulário 1 (Anexo A/A.2), i.e. a quantidade de reestruturações FR antes da primeira reestruturação. A Tabela 5.3 apresenta os valores obtidos e o percentual de redução do número de reestruturações calculado.

Tabela 5.3: Percentual de redução de reestruturações fortemente recomendadas.

Sistema	NumReestruturações-FRInicial	NumReestruturações-FRFinal	Percentual de redução do número de reestruturações FR
ArchTrace	35	24	31%

Qual o percentual de utilização do conhecimento do domínio junto às métricas na tomada de decisão?

A resposta a esta pergunta foi calculada com base na métrica descrita na questão Q3. A questão avalia o quanto o conhecimento do domínio apoiou a tomada de decisão junto às métricas. É importante ressaltar que as métricas são obtidas a partir da estrutura do modelo de projeto sem considerar a semântica dos elementos (i.e. pacotes e classes). O percentual de utilização do conhecimento do domínio junto às métricas atingido foi calculado extraindo-se o valor de "NumMétricasEDomínio" e subtraindo-se o total de métricas da resposta da questão Q5 do formulário 1 (Anexo A/A.2), i.e. a quantidade de métricas utilizadas junto ao conhecimento do domínio. O valor de "NumMétricasTotal" foi extraído do total de métricas da abordagem ROOSC. A Tabela 5.3 apresenta os valores obtidos e o percentual de redução do número de reestruturações calculado.

Tabela 5.3: Percentual de redução de reestruturações fortemente recomendadas.

Sistema	NumMétricasEDomínio	NumMétricasTotal	Percentual de utilização do conhecimento do domínio junto às métricas
ArchTrace	6	6	100%

Além das questões formuladas na Seção 5.2.3.1, foram analisadas questões subjetivas que não poderiam ser expressas em números. As questões e suas respectivas respostas estão sumarizadas na Tabela 5.4. As respostas comentadas serão detalhadas a seguir. É importante considerar que algumas questões foram comentadas para a melhoria de algumas características do apoio ferramental e este tipo de detalhamento de resposta que for relevante somente como lição aprendida para aperfeiçoamento do ferramental de apoio, será discutido na Seção 5.2.5.2.

A questão Q6 (Anexo A/A2) obteve resposta "sim", apesar disto o engenheiro de software questionou a existência de evidências que comprovem que o DBC seria mais manutenível do que o estilo arquitetural MVC utilizado no desenvolvimento da ferramenta ArchTrace. Neste caso, tal relato não é uma preocupação para esta avaliação, pois esta "quebra do MVC" foi discutida no Capítulo 3. Conforme a discussão estabelecida, a quebra do MVC está relacionada a forma como o projeto está estruturado, podendo-se obter resultados em que sejam gerados componentes que não agrupem elementos de camadas distintas. Isto se deve ao fato de que a abordagem ROOSC considera um modelo de entrada OO sem que ele precise possuir um estilo

arquitetural em particular, reorganizando este modelo de tal modo que os elementos estejam nos pacotes aos quais estão mais acoplados e aos quais estejam mais próximos coesivamente.

Tabela 5.4: Respostas das questões de avaliação subjetiva (Anexo A/A2).

Questões	Sim	Não	Parcialmente
Q6 - Os elementos (pacotes) do modelo reestruturado representam possíveis componentes, que apóiem a manutenção e a reutilização de partes da aplicação? Esta questão deve ser respondida olhando a estrutura de pacotes recuperada.	X		
Q7 - O modelo reestruturado poderia ser utilizado para apoiar a reengenharia da aplicação do paradigma de Orientação a Objetos para componentes?	X		
Q8 - Em algum momento você desejou realizar alguma reestruturação que não foi sugerida?		X	
Q9 – As métricas e seus *templates* são de fácil compreensão?	X		
Q10 - As sugestões de reestruturações e os *templates* das reestruturações são de fácil compreensão?			X
Q11 - Em algum momento você desejou ter alguma informação (métrica), para apoiar a tomada de decisão sobre que reestruturação aplicar, e esta informação não foi fornecida pela abordagem?		X	

A questão Q10 (Anexo A/A2) foi respondida como “Parcialmente”, conforme a Tabela 5.4. Neste caso, o engenheiro de software sentiu a necessidade de saber o grau de relevância das sugestões de reestruturação. Tal relato é considerado um retorno muito importante para uma futura ampliação no *template* das reestruturações, de modo que indique a relevância da reestruturação que vai além da melhoria dos valores das métricas.

As demais questões não foram comentadas ou possuem comentários referentes ao ferramental de apoio. Aquelas que forem referentes ao ferramental de apoio serão comentadas nas lições aprendidas relatadas na Seção 5.2.5.2.

5.2.5.1 Considerações em Relação à Hipótese Nula

O resultado da questão Q1, mostrado ao longo da Seção 5.2.5 não indicou que 100% dos pacotes resultantes representavam pacotes funcionais candidatos a componentes, mas tal valor não é preocupante para esta avaliação, visto que o projeto OO de um componente pode conter pacotes em seu interior e até mesmo sub-componentes. Considerando-se que o total de pacotes no menor nível hierárquico é igual a dois, este seria o menor número de pacotes que poderia ser transformado em componentes, pois os demais pacotes poderiam fazer parte de seu projeto como pacotes ou sub-componentes.

O resultado da questão Q2 (percentual de redução do número de reestruturações fortemente recomendadas) foi de apenas 31% enquanto as diretrizes da abordagem ROOSC indicam que a redução deve ser de 100%, porque as reestruturações fortemente recomendadas (FR) estão diretamente ligadas aos princípios de DBC e à própria definição de componentes adotadas nesta dissertação. Se estas reestruturações fossem aplicadas a percepção do engenheiro sobre o número de pacotes que poderiam tornar-se componentes poderia ser de um número maior. O detalhamento da questão Q10 do formulário 2 (Anexo A/A2) indicou a necessidade de mais informação no *template* das reestruturações e este fato pode ter influenciado o valor obtido na questão Q2.

Com o resultado da questão Q3 é possível observar o apoio do conhecimento do domínio junto às métricas para gerar pacotes mais coesos semanticamente além de estruturalmente. Unindo este resultado ao resultado das questões Q8 e Q11 (Tabela 5.4) do formulário 2 (Anexo A/A2), é possível observar que as métricas e suas informações foram suficientes para a análise da estrutura do modelo de projeto e foram complementadas pelo conhecimento do domínio por parte do engenheiro de software. Isto pode oferecer indícios de que, neste estudo, foi possível favorecer a formação de componentes com um projeto interno pouco acoplado, bastante coeso e com mais semântica.

Desta forma, os resultados do estudo possibilitaram observar o possível apoio da estratégia de reestruturação para gerar pacotes candidatos a componentes, conforme os princípios de DBC adotados nesta dissertação. Entretanto, devido as limitações do estudo, investigações adicionais devem ser realizadas para verificar a repetição deste comportamento

5.2.5.2 Lições Aprendidas

Este estudo permitiu obter um *feedback* do engenheiro de software, onde constam alguns pontos de melhoria com relação à estratégia de reestruturação e ao seu ferramental de apoio (ORC). Apesar da ferramenta ORC não ter sido avaliada, neste estudo, a sua utilização, durante o estudo, pode trazer algum *feedback* para o seu aperfeiçoamento.

Dentre os aspectos onde a estratégia de reestruturação poderia ser aprimorada, considerando-se o seu estágio naquele momento, destacaram-se:

- O nível hierárquico dos pacotes que devem ser considerados como componente varia de acordo com a visão do engenheiro de software e

assim deve ser tratado durante esta estratégia, ao avaliar o resultado, e durante toda a abordagem;

- As reestruturações precisam deixar evidente o seu grau de relevância para orientar ainda mais o engenheiro de software.

Dentre os aspectos onde a ORC poderia ser aprimorada, considerando-se o seu estágio naquele momento, destacaram-se:

- As sugestões de reestruturação baseadas no colapso de componentes (Seção 3.2.2) devem considerar que unir um pacote B a um outro pacote C é o mesmo que unir um pacote C ao pacote B, gerando apenas uma sugestão;
- As sugestões de reestruturação baseadas no colapso de componentes (Seção 3.2.2) só podem ser geradas se os pacotes envolvidos já não estiverem dentro de um mesmo pacote.

Dentre os aspectos onde o planejamento do estudo poderia ser aprimorado, considerando-se o seu estágio naquele momento, destacaram-se:

- A recuperação do modelo do sistema através de alguma ferramenta de engenharia reversa pode ser realizada antes de iniciar o estudo de caso, evitando que problemas na obtenção dos modelos de entrada afetem o resultado da abordagem.

5.3 Estudo de Viabilidade da Geração de Componentes e Interfaces

5.3.1 Objetivo Global

O propósito deste estudo é verificar a capacidade da estratégia de geração de componentes e interfaces em apoiar a obtenção de um conjunto de componentes e interfaces a partir de um modelo de projeto OO recuperado através de engenharia reversa e reestruturado conforme a estratégia de reestruturação da abordagem ROOSC. De forma que consiga obter um modelo de componentes em conformidade com aspectos semânticos e com os princípios de DBC encontrados na literatura e apresentados no Capítulo 2.

5.3.2 Objetivo do Estudo

Analisar a estratégia de geração de componentes e interfaces

Com o propósito de caracterizar

Com respeito à capacidade da estratégia de geração de componentes e interfaces proposta em gerar componentes e interfaces relevantes sob aspectos semânticos e princípios de DBC

Do ponto de vista do engenheiro de software

No contexto do engenheiro de software com conhecimento do domínio da ferramenta ArchTrace (MURTA*, et al.*, 2006)

5.3.3 Planejamento do Estudo

5.3.3.1 Questões

A fim de avaliar os resultados produzidos pela estratégia de geração de componentes e interfaces da abordagem ROOSC, o paradigma GQM (*Goal Question Metric*) de BASILI *et al.* (1994) é adotado. Tendo como base os princípios de DBC adotados nesta dissertação e o conhecimento do domínio por parte do engenheiro de software, elaboraram-se algumas questões para apoiar a avaliação dos resultados obtidos a partir da estratégia de reestruturação:

Q1: Qual o percentual de componentes gerados que são relevantes?

M1:

$$\textbf{PercentComponentesRelev} = \frac{\text{ComponentesRelevantes}}{\text{TotalComponentes}}$$

Visando calcular percentual dos componentes gerados que são relevantes, os seguintes valores são considerados:

ComponentesRelevantes: total de componentes que representam um conceito ou um conjunto de funcionalidades coesas do domínio, sob o ponto de vista do engenheiro de software.

TotalComponentes: total de componente gerados.

Q2: Qual o percentual de interfaces providas relevantes?

M1:

$$\textbf{PercentInterfacesProvidasRelev} = \frac{\text{InterfacesProvidasRelevantes}}{\text{TotalInterfacesProvidas}}$$

Visando calcular o percentual de interfaces providas geradas que são relevantes, os seguintes valores são considerados:

InterfacesProvidasRelevantes: total de interfaces que representam realmente serviços providos pelo componente sob o ponto de vista dos clientes deste componente.

TotalInterfacesProvidas: total de interfaces providas geradas.

Q3: Qual o percentual de interfaces requeridas relevantes?

M3:

$$\textbf{PercentInterfacesRequeridasRelev} = \frac{\text{InterfacesRequeridasRelevantes}}{\text{TotalInterfacesRequeridas}}$$

A fim de obter a precisão das interfaces geradas em relação as interfaces que o engenheiro de software esperava obter, os seguintes valores são considerados:

InterfacesRequeridasRelevantes: número de interfaces relevantes que representam realmente serviços os quais o componente necessita.

TotalInterfacesRequeridas: total de interfaces providas geradas.

5.3.3.2 Seleção do Contexto

O estudo será realizado em ambiente acadêmico sobre uma única aplicação. Esta aplicação foi selecionada pelo fato do especialista da aplicação estar disponível para apoiar o estudo. Como a estratégia de geração de componentes e interfaces recebe como entrada um modelo de projeto OO reestruturado através da estratégia de reestruturação da abordagem ROOSC, a aplicação selecionada é a mesma ferramenta utilizada no primeiro estudo (Seção 5.2).

5.3.3.3 Seleção dos Participantes

A seleção do participante deve considerar a importância do engenheiro de software possuir algum conhecimento sobre o domínio da aplicação e sobre DBC. Neste estudo, conseguiu-se um engenheiro de software que é também um especialista do domínio e conhece DBC, portanto ele aplicou a estratégia de geração de componentes e interfaces. Além da participação na aplicação da estratégia, o engenheiro de software ajudou a avaliar os componentes e interfaces geradas. Como a aplicação utilizada, neste estudo, é a mesma que foi utilizada no primeiro estudo (Seção 5.2), o engenheiro de software selecionado como participante foi o mesmo engenheiro que participou do

primeiro estudo.

5.3.3.4 Seleção de Variáveis

As variáveis independentes deste estudo são:

- A experiência do participante sobre DBC;
- O conhecimento do participante sobre o domínio;
- O modelo de projeto OO reestruturado;
- O conjunto de cenários de caso de uso do sistema;
- O conjunto de rastros de execução de cenário de caso de uso;

A variável dependente deste estudo é o modelo de componentes, no nível arquitetural, resultante.

5.3.3.5 Definição das Hipóteses

Hipótese nula (H0): A estratégia de geração de componentes e interfaces não apóia a geração de componentes que estejam em conformidade com aspectos semânticos do ponto de vista do engenheiro de software com conhecimento do domínio e com os princípios de DBC.

5.3.3.6 Validade do Estudo

A seguinte ameaça à validade do estudo foi identificada:

Validade Interna

Viés do Modelo Reestruturado: se o engenheiro de software não tiver reestruturado o modelo de projeto baseando-se no seu conhecimento do domínio da aplicação e nos princípios de DBC adotados nesta dissertação, ele pode ter gerado um modelo de projeto que não possua pacotes que sejam bons candidatos a componente (ex.: com baixo acoplamento, alta coesão e genérico o bastante para ser aplicado a novos contextos). Isto pode levar a um modelo de componentes com muitas dependências e uma quantidade excessiva de interfaces, acarretando uma avaliação ruim do modelo de componentes resultante.

Viés do Modelo de Componentes: se o engenheiro de software não tiver conhecimento do domínio da aplicação e dos princípios de DBC adotados nesta dissertação, ele pode ter dificuldade em definir quais componentes gerar e sobre como avaliar o modelo de componentes gerado. Por isso, o engenheiro de software convidado foi o próprio autor da aplicação em questão, que possui também conhecimento sobre DBC.

Validade de Conclusão

Refere-se a habilidade de concluir corretamente os relacionamentos entre o tratamento e o resultado do experimento.

Tratamento da hipótese nula: Como o estudo experimental foi realizado com apenas um participante e este participante possui um vasto conhecimento do domínio e de DBC, as conclusões do estudo podem ser influenciadas de tal modo que a estratégia de geração de componentes e interfaces pareça viável.

Validade de Construção

Refere-se aos relacionamentos entre a teoria e a conclusão, de tal modo que o tratamento reflita a causa bem e o resultado reflita o efeito bem.

Projeto do estudo: O pesquisador pode planejar o estudo visando obter os resultados esperados.

5.3.4 Operação do Estudo

5.3.4.1 Preparação

O engenheiro de software participante do estudo realiza a preparação dos dados de entrada, a fase de geração de componentes e interfaces do sistema a partir do modelo de projeto reestruturado através da estratégia de reestruturação da abordagem ROOSC e a avaliação do modelo de componentes.

Nesta etapa, o engenheiro de software recebe o material do estudo, contendo os formulários de: consentimento (Anexo B/B.1), avaliação do modelo de componentes (Anexo B/B.2), roteiro de geração de componentes e interfaces (Anexo B/B.3) e caracterização (Anexo B/B.4), recebendo ainda uma apresentação presencial sobre a utilização da ferramenta GenComp (*Generating Components*), que apóia a estratégia de geração de componentes e interfaces da abordagem ROOSC, sobre o preenchimento dos formulários e roteiro de realização do estudo de caso.

Durante a preparação dos dados, o engenheiro utiliza como uma das entradas para a estratégia o modelo de projeto OO do sistema reestruturado por ele em estudo de caso anterior. Para completar o conjunto de entradas da estratégia, o pesquisador executou o sistema alvo, especificando os seus cenários de casos de uso.

Os cenários de caso de uso foram definidos avaliando-se itens do menu principal e menus pop-up das aplicações. Opções de menu semanticamente equivalentes, como Salvar e Salvar Como, originaram apenas um cenário de caso de uso. Cenários de

exceção não foram considerados, embora cenários disparados de formas não convencionais (ex: clique em um link na interface) tenham sido considerados. Uma vez definidos os cenários de casos de uso, os rastros de execução das aplicações foram coletados com o apoio da ferramenta Tracer (CEPEDA e VASCONCELOS, 2006; VASCONCELOS, 2007).

Durante esta etapa, um treinamento presencial foi mostrado ao engenheiro de software, como dito anteriormente, explicando a respeito da utilização da ferramenta GenComp que apóia toda a estratégia de geração de componentes e interfaces. Este treinamento utiliza como exemplo a reestruturação da mesma aplicação utilizada no primeiro estudo.

5.3.4.2 Operação

Uma vez preparados os artefatos de entrada (i.e. modelo de projeto do sistema alvo reestruturado, cenários de caso de uso e rastros de execução), o engenheiro de software realizou a geração de componentes e interfaces do sistema ArchTrace. Esta etapa foi realizada com o apoio da ferramenta GenComp descrita no Capítulo 4 (Seção 4.4). A atividade de geração é uma atividade semi-automatizada e requer a tomada de decisão por parte do engenheiro. O modelo de componentes foi acompanhado, visualizando-o no próprio ambiente Odyssey para posterior avaliação do restante do formulário de avaliação.

Toda a geração foi realizada conforme o roteiro apresentado ao engenheiro de software. A cada passo da geração, o engenheiro de software avaliou algumas questões do formulário de avaliação sobre o modelo de componentes. Após o material ter sido preenchido, o material foi entregue para o pesquisador. Quanto às dúvidas que surgiram durante a geração (ex.: a respeito da ferramenta), o engenheiro de software entrou em contato com o pesquisador, que esteve presente no local do estudo para o esclarecimento de dúvidas.

5.3.5 Análise dos Resultados Obtidos

Durante o planejamento desse estudo, foi definido como objetivo principal avaliar a viabilidade da estratégia de geração de componentes e interfaces que faz parte da abordagem ROOSC. Essa viabilidade, conforme os questionamentos estabelecidos, é avaliada através da análise dos valores dos percentuais estabelecidos e através da análise das questões subjetivas respondidas pelo engenheiro do sistema alvo.

É importante ressaltar que o engenheiro de software foi instruído a avaliar o modelo de componentes sob o ponto de vista de componentes que representem conceitos ou funcionalidades do domínio, de acordo com o conhecimento do domínio do engenheiro.

Qual o percentual de componentes gerados que são relevantes?

A resposta a esta pergunta foi calculada com base na métrica descrita na questão Q1. A questão avalia os componentes resultantes sob o aspecto funcional e semântica destes componentes para serem reutilizados no mesmo domínio do sistema alvo. O percentual de componentes relevantes atingido foi calculado extraindo-se o valor de "ComponentesRelavantes" da resposta da questão Q2 do formulário 1 (Anexo B/B.2), i.e. a quantidade de componentes que o engenheiro julga representar uma funcionalidade ou conceito do domínio. O valor de "TotalComponentes" foi extraído da resposta da questão Q1 do formulário 1 (Anexo B/B.2), i.e. a quantidade de componentes resultantes após a geração de componentes e interfaces. A Tabela 5.5 apresenta os valores obtidos e o percentual de componentes relevantes calculado.

Tabela 5.5: Percentual de Componentes Relevantes.

Sistema	ComponentesRelevantes	TotalComponentes	Percentual de Componentes Relevantes
ArchTrace	12	12	100%

Qual o percentual de interfaces providas relevantes?

A resposta a esta pergunta foi calculada com base na métrica descrita na questão Q2. A questão avalia se as interfaces providas são relevantes. O percentual de interfaces providas, consideradas relevantes, atingido foi calculado extraindo-se o valor de "InterfacesProvidasRelevantes" da resposta da questão Q3 do formulário 1 (Anexo B/B.2), i.e. a quantidade de interfaces providas que realmente deveriam estar sendo providas pelos componentes que as provêem sob o ponto de vista dos clientes destes componentes. O valor de "TotalInterfacesProvidas" foi extraído da resposta da questão Q4 do formulário 1 (Anexo B/B.2), i.e. a quantidade de interfaces providas geradas ao final da geração de componentes e interfaces. A Tabela 5.6 apresenta os valores obtidos e o percentual de interfaces providas relevantes calculado.

Tabela 5.6: Percentual de interfaces providas relevantes.

Sistema	InterfacesProvidas-Relevantes	TotalInterfacesProvidas	Percentual de interfaces providas relevantes
ArchTrace	46	52	88%

Qual o percentual de interfaces requeridas relevantes?

A resposta a esta pergunta foi calculada com base na métrica descrita na questão Q3. A questão avalia se as interfaces providas são relevantes. O percentual de interfaces requeridas, consideradas relevantes, atingido foi calculado extraindo-se o valor de "InterfacesRequeridasRelevantes" da resposta da questão Q5 do formulário 1 (Anexo B/B.2), i.e. a quantidade de interfaces requeridas que realmente deveriam estar sendo requeridas pelos componentes que as requerem sob o ponto de vista da necessidade destes serviços nestes componentes. O valor de "TotalInterfacesRequeridas" foi extraído da resposta da questão Q6 do formulário 1 (Anexo B/B.2), i.e. a quantidade de interfaces requeridas geradas ao final da geração de componentes e interfaces. A Tabela 5.7 apresenta os valores obtidos e o percentual de interfaces requeridas relevantes calculado.

Tabela 5.7: Percentual de interfaces requeridas relevantes.

Sistema	InterfacesRequeridasRelevantes	TotalInterfacesRequeridas	Percentual de interfaces requeridas relevantes
ArchTrace	25	31	81%

Além das questões formuladas na Seção 5.3.3.1, foram analisadas questões subjetivas que não poderiam ser expressas em números. As questões e suas respectivas respostas estão sumarizadas na Tabela 5.8. As respostas comentadas serão detalhadas a seguir. É importante considerar que algumas questões foram comentadas para a melhoria de algumas características do apoio ferramental e este tipo de detalhamento de resposta que for relevante somente como lição aprendida para aperfeiçoamento do ferramental de apoio, será discutido na Seção 5.3.5.2.

A questão Q7 (Anexo B/B2) obteve resposta "Parcialmente", pois o engenheiro de software questionou a reutilização de alguns componentes que possuem funcionalidades muito específicas para o sistema alvo. Componentes voltados para funcionalidades de interface gráfica, infra-estrutura ou utilidades, segundo o engenheiro, são mais fáceis de ser reutilizados em novos contextos. Tal relato não é uma preocupação para esta avaliação, pois o objetivo da estratégia de geração de componentes e interfaces da abordagem ROOSC é possibilitar a reutilização de partes do sistema alvo em novos desenvolvimentos sem se preocupar com a especificidade do projeto do sistema alvo.

Tabela 5.8: Resposta das questões de avaliação subjetiva (Anexo B/B2).

Questões	Sim	Não	Parcialmente
Q7 - Os componentes gerados representam elementos, que apóiem a manutenção e a reutilização de partes da aplicação? Esta questão deve ser respondida olhando o modelo de componentes gerado.			X
Q8 - O modelo de componentes poderia ser utilizado para apoiar a reengenharia da aplicação do paradigma de Orientação a Objetos para componentes?	X		
Q9 - Em algum momento você desejou reestruturar o modelo de projeto OO para gerar componentes (via análise estática) diferentes dos que foram gerados?		X	
Q10 – As sugestões de componentes são de fácil compreensão? Se a resposta for Não ou Parcialmente, favor indicar o(s) tipo(s) de sugestão que precisam de melhorias.	X		
Q11 – Em algum momento você desejou obter alguma interface que não foi gerada? Em caso afirmativo informe quais as interfaces desejadas?	X		
Q12 – Em algum momento você desejou ter alguma informação, para apoiar a tomada de decisão sobre que componentes gerar, e esta informação não foi fornecida pela estratégia? Se a resposta for Sim ou Parcialmente, favor indicar quais as informações desejadas.	X		

A questão Q11 (Anexo B/B2) obteve resposta “Sim”, porque algumas funcionalidades de comunicação providas (ex: *sockets*) não geraram dependências por não terem sido mapeadas para o modelo de projeto do sistema alvo. Além disto, duas interfaces que pareciam fazer sentido semanticamente não foram geradas para os componentes gerados durante a análise estática. Tais relatos não inviabilizam a estratégia de geração de componentes e interfaces, porque as dependências que não são representadas no modelo de projeto ou nos rastros de execução não podem ser descobertas e tratadas pela estratégia. É importante ressaltar que estes artefatos são considerados variáveis independentes para o estudo e interferem no modelo de componentes resultante que é uma variável dependente.

A questão Q12 (Anexo B/B2) obteve resposta “Sim”, pois o engenheiro sentiu necessidade de visualizar os componentes gerados via análise estática durante a geração de componentes via análise dinâmica e poder vinculá-los aos cenários diretamente, ao invés de sempre ter que associar os cenários ou grupos de cenários a rastros de execução. Este relato é considerado uma importante contribuição para aprimorar futuramente a formação de componentes de sistema, não caracterizando qualquer falha na estratégia, pois os rastros continuariam a apoiar a determinação dos componentes de sistemas e suas interfaces, gerados via análise estática, que atendem a um determinado cenário ou grupo de cenários.

As demais questões não foram comentadas ou possuem comentários referentes ao ferramental de apoio. Aquelas que forem referentes ao ferramental de apoio serão comentadas nas lições aprendidas relatadas na Seção 5.3.5.2.

5.3.5.1 Considerações em Relação à Hipótese Nula

O resultado da questão Q1, mostrado ao longo da Seção 5.3.5 , indicou que 100% dos componentes resultantes representam funcionalidades ou conceitos do domínio. Este resultado pode ser observado pela avaliação da questão Q8 (Anexo B/B2), presente na Tabela 5.8, e reforça ainda mais o fato de que a avaliação da questão Q7 (Anexo B/B2) , presente na Tabela 5.8, não inviabiliza a estratégia de geração de componentes e interfaces. É importante ressaltar que a estratégia gera os componentes com uma funcionalidade específica mesmo que esta funcionalidade presente neles faça parte de um domínio muito específico ou não muito reutilizável, ainda sob o ponto de vista do domínio.

O resultado das questões Q2 e Q3 apresentam um grande percentual de interfaces relevantes, tanto providas quanto requeridas. Isto caracteriza um aspecto do resultado da estratégia que é a geração das interfaces, de modo que faça sentido sob o ponto de vista das necessidades de serviços e provimento dos mesmos nos componentes. Apesar da questão Q11 (Anexo B/B2), presente na Tabela 5.8, sugerir a ausência de algumas interfaces, estas não deveriam ser identificadas pela estratégia, se os artefatos de entrada não indicam as dependências que gerariam estas interfaces.

Apesar da questão Q12 (Anexo B/B2), presente na Tabela 5.8, indicar algumas informações que poderiam ter sido providas, estas informações não se caracterizaram como um empecilho a geração de componentes e interfaces e sim como uma possibilidade de flexibilizar a geração de componentes de sistema, possibilitando a associação direta do componente, gerado via análise estática, que atende ao cenário ou grupo de cenários.

Desta forma, os resultados da estratégia de geração de componentes e interfaces possibilitam observar um possível apoio desta estratégia na geração de componentes e interfaces, conforme os princípios de DBC adotados nesta dissertação.

5.3.5.2 Lições Aprendidas

Este estudo permitiu obter alguns comentários do engenheiro de software, onde consta alguns pontos de melhoria com relação à estratégia de reestruturação e ao seu

ferramental de apoio (GenComp). Apesar da ferramenta GenComp não ter sido avaliada, neste estudo, a sua utilização, durante o estudo, pode trazer algum *feedback* para o seu aperfeiçoamento.

O aspecto onde a estratégia de geração de componentes e interfaces poderia ser aprimorada, considerando-se o seu estágio naquele momento, é:

- A estratégia deve considerar a possibilidade da indicação do componente, gerado via análise estática, que atende a um determinado cenário ou grupo de cenários.

Dentre os aspectos onde a GenComp poderia ser aprimorada, considerando-se o seu estágio naquele momento, destacaram-se:

- A definição do nível hierárquico a ser considerado para a geração de componentes, via análise estática, poderia ser realizada utilizando-se a estrutura de árvores;
- As sugestões para geração de componentes, via análise estática, devem mostrar o nome acompanhado do *namespace* completo do pacote, facilitando a identificação de pacotes com o mesmo nome em *namespaces* distintos;
- As associações de rastros a cenários poderiam ser realizadas através de tabelas ao invés de caixas de diálogo, possibilitando que o engenheiro possa verificar os arquivos associados antes de confirmar a geração das interfaces com base nos rastros. Além disto, esta nova forma possibilita uma visão de todos os rastros que se está atribuindo.

5.4 Considerações Finais

Neste capítulo, foram apresentados os estudos experimentais realizados no contexto desta dissertação. Os estudos permitiram observar a possibilidade da abordagem ROOSC poder apoiar a reengenharia de software OO para componentes com base em métricas. Além disso, os estudos se mostraram fundamentais para o seu refinamento ao longo deste processo de pesquisa. Foi possível perceber, ao longo da realização desses estudos, a importância em se avaliar uma tecnologia, método ou processo na Engenharia de Software antes que o mesmo seja transferido para um contexto industrial.

Conforme apresentado no Capítulo 3, o processo proposto nesta dissertação engloba duas estratégias, a saber: reestruturação do modelo de projeto visando obter

bons candidatos a componentes com base nos princípios de DBC apresentados nesta dissertação, e a geração de componentes e interfaces propriamente. Neste contexto, foram conduzidos dois estudos, um para verificar a viabilidade de cada estratégia.

Vale ressaltar que outros estudos ainda deveriam ser conduzidos, a fim de propiciar novas melhorias na abordagem ROOSC e nos seus sub-produtos, sendo listados alguns que seriam de grande valia, como se segue:

- Estudos de viabilidade com novos participantes, uma vez que o estudo já está planejado;
- Estudos de observação, ainda no ambiente acadêmico, junto a um estudo sobre o apoio das métricas sugeridas nesta dissertação;
- Estudos de observação com sistemas no contexto da indústria para verificar a usabilidade em ambientes não acadêmicos;
- Avaliação de uma tarefa de implementação guiada pelo projeto de componentes gerado pela abordagem ROOSC;
- Verificação da abordagem ROOSC como um todo, aplicando-a em um único estudo para a criação do modelo de componentes e verificação do esforço necessário.

Convém ressaltar que foram realizados os estudos necessários para sustentar a teoria defendida nesta dissertação acerca de uma nova forma de realizar a reengenharia de sistemas OO para componentes.

É importante ressaltar, mais uma vez, que TRAVASSOS *et al.* (2002) destacam a importância da experimentação e validação para novos métodos técnicas, linguagens e ferramentas, onde a experimentação representa o centro do processo científico, consistindo na única forma de se verificar uma nova teoria. Deste modo, os sistemas escolhidos para estes estudos foram no âmbito acadêmico, pois ao envolver a indústria em estudos de viabilidade pode ser muito custoso pela falta de garantia de retorno. Os estudos realizados são uma prova de conceito que não permite generalizar os resultados a ponto de utilizá-los na indústria.

Capítulo 6 Conclusão

6.1 Epílogo

Esta dissertação apresentou uma abordagem de reengenharia de software orientado a objetos (OO) para componentes, denominada ROOSC. ROOSC é composta de 2 estratégias, a saber: reestruturação de agrupamentos de classes (i.e. pacotes), visando à obtenção de bons candidatos a componentes, de acordo com os princípios de DBC, apresentados nesta dissertação; e a geração de componente e interfaces propriamente, visando gerar componentes em conformidade com os princípios de DBC apresentados nesta dissertação.

Conforme apresentado, a necessidade por softwares cada vez mais complexos, aliado a escassez de tempo e recursos, à busca por maior qualidade nos produtos de software e à existência de sistemas legados ricos em conhecimento, faz com que a indústria busque novas soluções. Estas soluções devem permitir manter os softwares mais facilmente e reutilizar o conhecimento presentes, no software legado. No caso do software OO, a migração para o paradigma de DBC pode trazer maior reusabilidade e manutenibilidade. Neste contexto, as abordagens de reengenharia para componentes provêem suporte a grande parte do processo de reengenharia, mas não agrupam algumas características importantes para este tipo de técnica (ex: diretrizes para formação do projeto de componentes e interfaces).

Uma vez que os sistemas legados costumam ser complexos, mal projetados e não possuem documentação de apoio atualizada, o maior esforço de pesquisa e desenvolvimento nesta dissertação se concentrou na atividade de reestruturação, com foco na re-organização dos agrupamentos de classes guiada por métricas. Como resultado, obteve-se um conjunto de métricas e reestruturações. Finalmente, a estratégia de geração de componentes e interfaces, que representa a última fase da abordagem proposta, envolve a transformação dos pacotes do modelo de projeto reestruturado em componentes e suas respectivas interfaces, gerando ainda os componentes de sistema.

6.2 Contribuições

Esta dissertação apresenta as seguintes contribuições, destacadas com base nos pontos em aberto nas abordagens existentes que foram apresentadas no Capítulo 2.

1) *Uma abordagem sistemática de reengenharia de software OO para componentes:* a abordagem proposta, ROOSC, envolve etapas bem definidas, executadas através de processos com critérios, métodos e técnicas estabelecidos para apoiar a geração de componentes a partir de um projeto de sistema OO. Cobrindo desde uma estratégia para a formação do projeto interno dos componentes, até uma estratégia para a formação de interfaces e geração de componentes que vai além de seu projeto interno. Estas regras para a formação de componentes e interfaces podem ser utilizadas independentemente de linguagem de implementação do sistema alvo ou de uma possível re-implementação.
2) *Um conjunto de métricas associado a um conjunto de reestruturações para a formação de projeto interno OO para componentes*: um conjunto de métricas para projeto OO de componentes foi sugerido e a ele foi associado um conjunto de reestruturações. Este conjunto de métricas e reestruturações é utilizado para a formação do projeto dos pacotes candidatos a componentes.
3) *Apoio ferramental integrado a um ambiente de reutilização de software*: foi desenvolvido apoio ferramental à abordagem ROOSC, cobrindo a estratégia de reestruturação e a estratégia de geração de componentes e interfaces. O ambiente Odyssey apresenta uma infra-estrutura de apoio à reutilização, permitindo que os modelos gerados por ROOSC sejam reutilizados em desenvolvimentos futuros.
4) *Estudos experimentais:* estudos experimentais foram realizados no contexto desta dissertação, gerando um planejamento que pode ser re-aproveitado em novos estudos e obtendo resultados que contribuirão para a melhoria da abordagem proposta.

6.3 Limitações

Esta seção apresenta as limitações em relação a abordagem proposta, ROOSC:

1) *Necessidade de conhecimento do domínio por parte do engenheiro e software*: embora a abordagem permita que o engenheiro de software tome as suas decisões, fornecendo-lhe informações, é imprescindível que ele tenha algum conhecimento do domínio para discernir o que é bom apenas estruturalmente e o que representa algo que faz sentido no domínio da aplicação.

2) *Versão da UML*: em 2007, quando esse trabalho de pesquisa foi iniciado, a versão da especificação da UML utilizada pelo ambiente Odyssey era a 1.4 e ainda é atualmente. A partir desse momento, outras versões surgiram, estando atualmente a linguagem na versão 2.2. Entretanto, o ferramental de apoio da ROOSC ainda trabalha com a versão 1.4 da UML, que é a versão utilizada pelo ambiente Odyssey, atualmente.
3) *Esforço na avaliação do modelo de componentes gerado*: a avaliação do modelo de componentes gerado pode exigir grande esforço, quando o modelo de projeto reestruturado ou o número de cenários são extensos. Além do número de componentes resultantes poder ser extenso, o número de interfaces também pode ser e isto dificulta a leitura do modelo.
4) *Generalidade da ROOSC*: embora generalidade tenha sido um requisito buscado nesta dissertação, para uma abordagem de reengenharia de software OO para componentes, observa-se que, assim como outras abordagens da literatura, ROOSC atinge uma generalidade parcial. A abordagem é voltada à reengenharia de sistemas OO e o seu ferramental apóia a análise de sistemas escritos em Java.

6.4 Trabalhos Futuros

Algumas oportunidades de melhorias e possibilidades de extensão desta pesquisa foram identificadas ao longo do trabalho, podendo ser tratadas através de trabalhos futuros, como se segue:

1) *Refinamento do modelo de componentes*: o modelo de componentes ainda pode ser refinado através de métricas para agrupamento de componentes, métricas que avaliem a complexidade do componente, reusabilidade (CHO, *et al.*, 2001), entre outras.
2) *Geração de código a partir do modelo de componentes gerado*: embora algumas abordagens de reengenharia busquem a re-implementação dos sistemas em alguma tecnologia de componentes (PRADO, 2005), a abordagem proposta nesta dissertação tem maior ênfase no projeto dos componentes. Apesar desta ênfase, há a necessidade de gerar código, reutilizando código do sistema original, a fim de verificar a efetiva reutilização dos componentes gerados pela ROOSC.

3) *Realização de novos estudo:* após a realização dos estudos experimentais desta dissertação, pode-se planejar a realização de novos estudos de viabilidade considerando sistemas com escalas maiores e, posteriormente, realizar estudos na indústria com sistemas reais.
4) *Teste da abordagem:* o sistema original poderia ser re-gerado com os componentes definidos, verificando-se a manutenção de seu comportamento.
5) *Acompanhamento da evolução do sistema:* a abordagem ROOSC poderia ser utilizada para o acompanhamento da evolução do sistema, através de um mecanismo de armazenamento histórico das métricas. Dessa forma, de tempos em tempos poderia se verificar se a estrutura do sistema está se degradando.

Referências Bibliográficas

ARAÚJO, M. A. P. e TRAVASSOS, G. H., 2006, "Experimentais em Manutenção de Software: Observando Decaimento de Software Através de Modelos Dinâmicos", In: Simpósio Brasileiro de Qualidade de Software / Workshop de Manutenção de Software Moderna, 2006, pp. Vila Velha/ES.

BARROCA, L., GIMENES, I. M. S. e HUZITA, E. H. M., 2005, "Desenvolvimento Baseado em Componentes". In: *Desenvolvimento Baseado em Componentes: Conceitos e Técnicas*,Ciência Moderna, pp.

BASILI, V., CALDIEIRA, G. e ROMBACH, H., 1994, Goal Question Metrics Paradigm, Encyclopedia of Software Engineering. John Wiley & Sons.

BLOIS, A. P. B., 2006, Uma Abordagem de Projeto Arquitetural Baseado em Componentes no Contexto de Engenharia de Domínio. COPPE, UFRJ,

BOJIC, D. e VELASEVIC, D., 2000, "A Use-Case Driven Method of Architecture Recovery for Program Understanding and Reuse Reengineering", In: 4th European Software Maintenance and Reengineering Conference, pp. 23-31, Zuriq, Swiss.

BOOCH, G., RUMBAUGH, J. e JACOBSON, I., 1998, The Unified Modeling Language User Guide. 1st Addison-Wesley.

BRAGA, R., 2000, Busca e Recuperação de Componentes em Ambientes de Reutilização de Software. COPPE, UFRJ, Rio de Janeiro, Brasil.

BROWN, A. W., 2000, Large-Scale Component-Base Development. 1st edition Prentice Hall.

CARNEY, D., 1997, Assembling Large Systems from COTS Components: Opportunities, Cautions, and Complexities.

CEPEDA, R. S. V. e VASCONCELOS, A. P. V., 2006, "Tracer e Phoenix: Ferramentas Integradas para a Engenharia Reversa de Modelos Dinâmicos de Java para UML", In: XX Simpósio Brasileiro de Engenharia de Software, XIII Sessão de Ferramentas, pp. 25-30, Florianópolis, SC, Brasil.

CHARDIGNY, S., SERIAI, A., OUSSALAH, M. e TAMZALIT, D., 2008, "Extraction of Component-Based Architecture from Object-Oriented Systems", Working IEEE/IFIP Conference on Software Architecture (WICSA) 2008, Vancouver, BC, Canada.

CHEESMAN, J. e DANIELS, J., 2001, UML components: a Simple Process for Specifying Component-based Software. Addison-Wesley Longman Publishing.

CHIDAMBER, S. R. e KEMERER, C. F., 1994, "A metrics suite for object oriented design", IEEE Transaction on Software Engineering, 20, pp. 476–493.

CHIKOFSKY, E. J. e CROSS, J. H. I., 1990, Reverse Engineering and Design Recovery: a Taxonomy. *IEEE Software, v. 7, n. 1*. pp. 13-17.

CHO, E. S., KIM, M. S. e KIM, S. D., 2001, "Component Metrics to Measure Component Quality", In: Eighth Asia-Pacific Software Engineering Conference (APSEC'01), pp. 419, Macau.

CRNKOVIC, I., CHAUDRON, M. e LARSSON, S., 2006, " Component-Based Development Process and Component Lifecycle", In: Software Engineering Advances, International Conference pp. 44-44,

D'SOUZA, D. e WILLS, A., 1999, Objects, Components and Frameworks with UML – The Catalysis Approach. Addison-Wesley Publishing Company.

DANTAS, A. R., 2001, Um Sistema de Críticas para a UML. UFRJ, Rio de Janeiro, RJ, Brasil.

DANTAS, A. R., CORREA, A. L. e WERNER, C. M. L., 2001, "Oráculo: Um Sistema de Críticas para a UML", In: XV Simposio Brasileiro de Engenharia de Software - SBES, Caderno de Ferramentas, pp. 398-403, Rio de Janeiro, RJ , Brasil.

DANTAS, C. R., 2005, Odyssey-WI: Uma Abordagem para a Mineração de Rastros de Modificação de Modelos em Repositórios Versionados. UFRJ, Rio de Janeiro, RJ, Brasil.

DING, L. e MEDVIDOVIC, N., 2001, "Focus: a Light-Weight, Incremental Approach to Software Architecture Recovery and Evolution", Working IEEE/IFIP Conference on Software Architecture (WICSA'01), Amsterdam, Holland.

DISESSA, A. A., AZEVEDO, F. e PARFANES, O., 2004, "in component computing: A synthetic review", Interactive Learning Enviroments, pp. 109-159.

FAVRE, J. M., CERVANTES, H., SANLAVILLE, R., DUCLOS, F. e ESTUBLIER, J., 2001, Issues in Reengineering the Architecture of Component-Based Software.

FONTANETTE, V., GARCIA, V. C., BOSSONARO, A. A., PEREZ, A. B. e PRADO, A. F., 2002, "Reprojeto de Sistemas Legados Baseado em Componentes de Software", XXVIII Conferencia Latinoamericana de Informática (InfoUYclei), Montevideo, Uruguai.

GAMMA, E., HELM, R., JOHNSON, R. e AL, E., 2000, Padrões de Projeto - Soluções Reutilizáveis de Software Orientado a Objetos. Bookman.

GANESAN, D. e KNODEL, J., 2005, "Identifying Domain-Specific Reusable Components from Existing OO Systems to Support Product line Migration", In: Workshop on Reengineering towards Product Lines, pp. 27–36, Pittsburgh, USA.

GIMENES, I. M. S., BARROCA, L., HUZITA, E. H. M. e CARNIELLO, A., 2000, "O Processo de Desenvolvimento Baseado em Componentes Através de Exemplos". In: Ceretta, R. ERI 2000,VIII Escola de Informática da SBC-Sul, pp. 147-178.

GONÇALVES, J. J., OLIVEIRA, T. M. A. e OLIVEIRA, K. M., 2006, "Métricas de Reusabilidade para Componentes de Softwares", VI Workshop de Desenvolvimento Baseado em Componentes (WDBC 2006), pp. 14-21.

IEEE, 1993, "IEEE Standard Glossary of Software Engineering Terminology", pp.

IEEE, C.-T., 2004, Reengineering & Reverse Engineering Terminology. Washington.

JPROFILE, 2008,

KNODEL, J., 2004, "On Analyzing the Interfaces of Components", Workshop in Software Reengineering,, Bad Honnef, Germany.

LEE, E., LEE, B., SHIN, W. e WU, C., 2003, "A Reengineering Process for Migrating from an Object-oriented Legacy System to a Component-based System", In: the 27th Annual International Conference on Computer Software and Applications, pp. 336,

LEE, J. K., JUNG, S. J. e KIM, S. D., 2001, "Component Identification Method with Coupling and Cohesion", In: 8th Asia–Pacific Software Engineering, pp. 79–86, Macau, China.

LEMOS, G. S., RECCHIA, E. L., PENTEADO, R. e BRAGA, R., 2003, "Padrões de Reengenharia auxiliados por Diretrizes de Qualidade de Software", The Third Latin American Conference on Pattern Languages of Programming - SugarLoafPLoP 2003, Porto de Galinhas, PE, Brasil.

LOPES, M. A. M., 2005, Um Mecanismo para Apoio à Percepção Aplicado a Modelos de Software Compartilhados. UFRJ, Rio de Janeiro, RJ, Brasil.

MAIA, N. E. N., 2006, Uma Abordagem para a Transformação de Modelos. UFRJ, Rio de Janeiro, RJ, Brasil.

MANGAN, M. A. S., 2006, Uma Abordagem para o Desenvolvimento de Apoio à Percepção em Ambientes Colaborativos de Desenvolvimento de Software. UFRJ, Rio de Janeiro, Brasil.

MARTIN, R., 2000, Design Principles and Design Patterns.

MELO JR, C. R. S., 2005, Metrictool: Uma Ferramenta Parametrizável para Extração de Métricas de Projetos Orientados a Objetos. UFRJ, Rio de Janeiro, Brasil.

MILER, N., 2000, A Engenharia de Aplicações no Contexto da Reutilização baseada em Modelos de Domínio. COPPE, UFRJ, Rio de Janeiro, Brasil.

MOURA, A. M., VASCONCELOS, A. e WERNER, C., 2008a, "Uma Estratégia de Reestruturação de Modelos baseada em Métricas para Apoiar a Reengenharia de Software Orientado a Objetos para Componentes", INFOCOMP Journal of Computer Science, pp. 1-10.

MOURA, A. M. M., VASCONCELOS, A. P. V. e WERNER, C. M. L., 2008b, "ORC: Uma Ferramenta de Reestruturação de Modelos baseada em Métricas para Apoiar a Reengenharia de Software Orientado a Objetos para Componentes", In: II Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software - Sessão de Ferramentas, pp. 33-40, Porto Alegre : PUC RS.

MOURA, A. M. M., VASCONCELOS, A. P. V. e WERNER, C. M. L., 2008c, "Uma Abordagem de Reengenharia de Software Orientado a Objetos para Componentes apoiada por Métricas", In: XXII Simpósio Brasileiro de Engenharia de Software - XIII Wokshop de Teses e Dissertações em Engenharia de Software, 2008, pp. 19-24, Campinas.

MOURA, A. M. M., VASCONCELOS, A. P. V. e WERNER, C. M. L., 2008d, "Uma Estratégia de Reestruturação de Modelos baseada em Métricas para Apoiar a Reengenharia de Software Orientado a Objetos para Componentes", Workshop de Manutenção Moderna de Software (WMSWM 2008), Florianópolis, Santa Catarina, Brasil.

MURTA, L. G. P., 1999, FRAMEDOC: Um Framework para a Documentação de Componentes Reutilizáveis. UFRJ, Rio de Janeiro, RJ, Brasil.

MURTA, L. G. P., 2002, Charon: Uma Máquina de Processos Extensível Baseada em Agentes Inteligentes. UFRJ, Rio de Janeiro, RJ, Brasil.

MURTA, L. G. P., 2006, Gerência de Configuração no Desenvolvimento Baseado em Componentes. UFRJ, Rio de Janeiro, RJ, Brasil.

MURTA, L. G. P., VAN DER HOEK, A. e WERNER, C. M. L., 2006, "ArchTrace: Policy-Based Support for Managing Evolving Architecture-to-Implementation Traceability Links", In: International Conference on Automated Software Engineering (ASE), pp. 135-144, Tokyo, Japan.

MURTA, L. G. P., VASCONCELOS, A. P. V., BLOIS, A. P. B., LOPES, M. A. M., MELO JR., C. R. e WERNER, C. M. L., 2004, "Run-Time Variability through Component Dynamic Loading", In: Sessão de Ferramentas, XVIII Simpósio Brasileiro de Engenharia de Software, pp. 67-72, Brasília, Brasil.

NIERSTRASZ, O., DUCASSE, S. e DEMEYER, S., 2005, "Object-oriented Reengineering Patterns An Overview", 4th International Conference GPCE, Tallinn, Estonia.

ODYSSEY, 2008, Odyssey: Infra-Estrutura de Reutilização baseada em Modelos de Domínio. Disponível em: http://reuse.cos.ufrj.br/odyssey.

PRADO, A. F., 2005, "Reengenharia de Software Baseado em Componentes". In: Desenvolvimento Baseado em Componentes: Conceitos e Técnicas,Ciência Moderna, pp.

PREMERLANI, W. J. e BLAHA, M. R., 1994, "An Approach for Reverse Engineering of Relational Databases", Communications of the ACM, v. 37, n. 5, pp. 42-49.

PRESSMAN, R., 2006, Software Engineering: A Practitioner's Approach. 6th edition McGraw-Hill Science/Engineering/Math.

RICHNER, T. e DUCASSE, R., 1999, "Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information", In: International Conference on Software Maintenance (ICSM'99), pp. 13-22, Oxford, UK.

RIVA, C. e RODRIGUEZ, J. V., 2002, "Combining Static and Dynamic Views for Architecture Reconstruction", In: Sixth European Conference on Software Maintenance and Reengineering (CSMR'02), pp. 47-56, Budapeste, Hungary.

SAMETINGER, J., 1997, Software engineering with reusable components. New York, NY, USA, Springer-Verlag New York, Inc.

SDMETRICS, 2007, Sdmetrics Tool. Disponível em: http://www.sdmetrics.com.

SILVA, I. A., 2005, GAW: Um Mecanismo Visual de Percepção de Grupo Aplicado ao Desenvolvimento Distribuído de Software. UFRJ, Rio de Janeiro, RJ, Brasil.

SOMMERVILLE, I., 1995, Software Engineering. 5a Edição Addison-Wesley.

SPAGNOLI, L. e BECKER, K., 2003, Um estudo sobre o Desenvolvimento Baseado em Componentes. In: Faculdade de Informática, PUCRS.

TEIXEIRA, H. V., 2003, Geração de Componentes de Negócio a partir de Modelos de Análise. UFRJ, Rio de Janeiro, Brasil.

TRAVASSOS, G. H., GUROV, D. e AMARAL, E. A. G. G., 2002, Introdução à Engenharia de Software Experimental. In: Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ.

VASCONCELOS, A., 2007, Uma Abordagem de Apoio à Criação de Arquiteturas de Referência de Domínio Baseada na Análise de Sistemas Legados. COPPE/UFRJ, Rio de Janeiro, Brasil.

VERNAZZA, T., GRANATELLA, G., SUCCI, G., BENEDICENTI, L. e MINTCHEV, M., 2000, "Defining metrics for software components", pp.

VERONESE, G. O. e NETTO, F. J., 2001, ARES: Uma Ferramenta de Auxílio à Recuperação de Modelos UML de Projeto a partir de Código Java. UFRJ, Rio de Janeiro.

VITHARANA, P., JAIN, H. e ZAHEDI, F., 2004, Strategy-Based Design of Reusable Business Components.

WANG, X., SUN, J., YANG, X., HUANG, C., HE, Z. e MADDINENI, S. R., 2006, "Reengineering standalone C++ legacy systems into the J2EE partition distributed environment", In: 28th International Conference on Software Engineering, pp. 525 - 533, Shanghai, China.

WANG, X., YANG, X., SUN, J. e CAI, Z., 2008, "A New Approach of Component Identification Based on Weighted Connectivity Strength Metrics", . Information Technology Journal (7), 1, pp. 56-62.

WARDEN, R., 1992, Re-engineering - A Pratical Methodology With Commercial Applications. Chapman & Hall, London, England.

WASHIZAKI, H. e FUKAZAWA, Y., 2005, A Technique for Automatic Component Extraction from Object-Oriented Programs by Refactoring. Elsevier North-Holland, Inc., Amsterdam, The Netherlands, The Netherlands.

WERNER, C. M. L., 2005, Reuse: Técnicas e Ferramentas de Apoio à Reutilização de Software. Projeto Integrado de Pesquisa do CNPQ,

WERNER, C. M. L. e BRAGA, R. M. M., 2005, "A Engenharia de Domínio e o Desenvolvimento Baseado em Componentes". In: Desenvolvimento Baseado em Componentes: Conceitos e Técnicas,Ciência Moderna, pp.

WERNER, C. M. L., MATTOSO, M. e BRAGA, R., 1999, "Odyssey: Infra-Estrutura de Reutilização Baseada em Modelos de Domínio", Sessão de Ferramentas do XIII Simpósio Brasileiro de Engenharia de Software, Florianópolis, SC, Brasil.

WOHLIN, C., RUNESON, P., HÖST, M. e AL, E., 2000, Experimentation in Software Engineering: an Introduction. Kluwer Academic Publishers,

XAVIER, J. R., 2001, Criação e Instanciação de Arquiteturas de Software Específicas de Domínio no Contexto de uma Infra-Estrutura de Reutilização. UFRJ, Rio de Janeiro, Brasil.

Anexo A: Instrumentação do Estudo Experimental de Viabilidade da Estratégia de Reestruturação

A.1: Formulário de Consentimento para Apoio no Estudo de Viabilidade da Estratégia de Reestruturação de Modelos OO Baseada em Métricas

Apoio no Estudo de Viabilidade da Abordagem de Reestruturação de Modelos OO Baseada em Métricas

Eu declaro ter mais de 18 anos de idade e que concordo em participar de um estudo conduzido por Ana Maria da Mota Moura, como parte das atividades de mestrado no Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ. Este estudo visa verificar a viabilidade em se utilizar a estratégia de reestruturação de modelos OO baseada em métricas, para apoiar a reengenharia de software OO para componentes.

PROCEDIMENTO

Este estudo está dividido em três etapas: primeiramente, o engenheiro de software, participante do estudo, recupera a arquitetura da aplicação através de alguma abordagem de engenharia reversa; em seguida, o próprio engenheiro de software aplica a estratégia de reestruturação de modelos OO baseada em métricas para reestruturar o modelo OO da aplicação; por fim, o engenheiro de software avalia o modelo reestruturado, preenchendo o formulário de avaliação. Com base nos resultados, o pesquisador será capaz de verificar se a estratégia é viável e, em caso positivo, verificar pontos da estratégia que necessitam de melhoria para refiná-la.

BENEFÍCIOS, LIBERDADE DE DESISTÊNCIA

Eu entendo que sou livre para realizar perguntas a qualquer momento ou solicitar que qualquer informação relacionada à minha pessoa não seja incluída no estudo. Eu entendo que participo de livre e espontânea vontade com o único intuito de contribuir para o avanço e desenvolvimento de técnicas e processos para a Engenharia de Software.

EQUIPE: **PESQUISADOR RESPONSÁVEL**
Ana Maria da Mota Moura
Programa de Engenharia de Sistemas e Computação - COPPE/UFRJ

PROFESSORES RESPONSÁVEIS
Profa. Cláudia M.L. Werner
Programa de Engenharia de Sistemas e Computação - COPPE/UFRJ
Profa. Aline Pires Vieira de Vasconcelos
CEFET Campos

Nome (em letra de forma):_____

Assinatura: _____ Data:_____

A.2: Formulários de Avaliação do Modelo Reestruturado

Formulário1 ***Avaliação do Modelo Reestruturado*** **Identificação:**
Aplicação: _____ **Engenheiro de Software: _____** **Hora início: _____:_____ Hora fim: _____:_____**
***Questões*:**
Q1. Quantos pacotes resultaram da reestruturação do modelo? _____
Q2. Dentre os pacotes resultantes da reestruturação do modelo, candidatos a componentes, quantos você julga que possuem uma funcionalidade específica e bem definida? _____
Q3. Quantas sugestões de reestruturações fortemente recomendadas existiam antes da reestruturação do modelo? _____
Q4. Quantas sugestões de reestruturações fortemente recomendadas restaram após a finalização da reestruturação do modelo? _____
Q5. Quantas e quais métricas foram utilizadas para apoiar a seleção das reestruturações aplicadas independentemente da utilização do conhecimento do domínio? _____

Formulário 2:

Avaliação do Modelo Reestruturado

Este é um formulário de perguntas subjetivas, as quais devem ser respondidas com um dos valores do conjunto {"Sim", "Não", "Parcialmente"}, seguido de uma justificativa.

Q6. Os elementos (pacotes) do modelo reestruturado representam possíveis componentes, que apóiem a manutenção e a reutilização de partes da aplicação? Esta questão deve ser respondida olhando a estrutura de pacotes recuperada.

_____

_____

_____

_____

_____

_____

Q7. O modelo reestruturado poderia ser utilizado para apoiar a reengenharia da aplicação do paradigma de Orientação a Objetos para componentes?

_____

_____

_____

_____

_____

Q8. Em algum momento você desejou realizar alguma reestruturação que não foi sugerida?

_____

_____

_____

_____

_____

_____

Q9. As métricas e seus *templates* são de fácil compreensão? Se a resposta for Não ou Parcialmente, favor indicar as métricas que precisam de melhorias.

_____
_____
_____
_____
_____
_____

Q10. As sugestões de reestruturações e os *templates* das reestruturações são de fácil compreensão? Se a resposta for Não ou Parcialmente, favor indicar as sugestões de reestruturações que precisam de melhorias.

_____
_____
_____
_____
_____
_____

Q11. Em algum momento você desejou ter alguma informação (métrica), para apoiar a tomada de decisão sobre que reestruturação aplicar, e esta informação não foi fornecida pela abordagem? Se a resposta for Sim ou Parcialmente, favor indicar as informações (métricas) desejadas.

_____
_____
_____
_____
_____
_____

A.3: Roteiro de Instruções para a Realização do Estudo

Roteiro para a realização de estudo utilizando a abordagem de reestruturação de modelos baseada em métricas para apoiar a reengenharia de software OO para componentes:

1. Preencher o formulário de Caracterização de Participante e assinar o formulário de Consentimento.
2. Recuperar o modelo de projeto da aplicação utilizando a ferramenta ARES, no ambiente Odyssey.
3. Ler o formulário de avaliação 1 e 2.
4. Para a questão 3 do formulário 1, responder em números o que se pede.
5. Iniciar a marcação do tempo de estudo.
6. Reestruturar o modelo de projeto com o apoio da ferramenta ORC.
7. Para a questão 5 do formulário 1, responder em números o que se pede.
8. Para as questões 8 a 11 do formulário 2, responder {“Sim”, “Não”, “Parcialmente”} seguido de uma justificativa.
9. Retornar ao passo 6 até que todas as reestruturações fortemente recomendadas tenham sido aplicadas e não se deseje aplicar qualquer outra reestruturação opcional.
10. Finalizar a marcação do tempo de estudo.
11. Para as questões 1,2 e 4 do formulário 1, responder em números o que se pede.
12. Para as questões 6 e 7 do formulário 2, responder {“Sim”, “Não”, “Parcialmente”} seguido de uma justificativa.

A.4: Questionário de Caracterização do Participante

Nome _____

Nível (Ms.c/D.Sc.): _____

Experiência em Desenvolvimento de Software

Qual é a sua experiência com desenvolvimento de software na prática? Por favor, inclua na sua resposta o tempo de experiência relevante em desenvolvimento.

_____

_____

_____

Por favor, indique o grau de sua experiência nesta seção seguindo a escala de 5 pontos abaixo:

1 = nenhum

2 = estudei em aula ou em livro

3 = pratiquei em 1 projeto em sala de aula

4 = usei em 1 projeto na indústria

5 = usei em vários projetos na indústria

Experiência em DBC

- Experiência especificando componentes 1 2 3 4 5
- Experiência utilizando algum método de DBC 1 2 3 4 5
- Experiência utilizando métricas de componentes 1 2 3 4 5

Experiência no Domínio

- Experiência no domínio da aplicação 1 2 3 4 5

Experiência em Reutilização de Software

- Experiência (prática) com reutilização de software 1 2 3 4 5

Experiência em Orientação a Objetos

- Experiência no desenvolvimento de projeto OO 1 2 3 4 5
- Experiência utilizando métricas de projeto OO 1 2 3 4 5

Anexo B: Instrumentação do Estudo Experimental de Viabilidade da Estratégia de Geração de Componentes e Interfaces

B.1: Formulário de Consentimento para Apoio no Estudo de Viabilidade da Estratégia de Geração de Componentes e Interfaces

Apoio no Estudo de Viabilidade da Abordagem de Geração de Componentes e Interfaces

Eu declaro ter mais de 18 anos de idade e que concordo em participar de um estudo conduzido por Ana Maria da Mota Moura, como parte das atividades de mestrado no Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ. Este estudo visa verificar a viabilidade em se utilizar a estratégia de geração de componentes e interfaces, para apoiar a reengenharia de software OO para componentes.

PROCEDIMENTO

Este estudo está dividido em duas etapas: primeiramente, o engenheiro de software aplica a estratégia de geração de componentes e interfaces para gerar os componentes e interfaces da aplicação; por fim, o engenheiro de software avalia o modelo de componentes, preenchendo o formulário de avaliação. Com base nos resultados, o pesquisador será capaz de verificar se a estratégia é viável e, em caso positivo, verificar pontos da estratégia que necessitam de melhoria para refiná-la.

BENEFÍCIOS, LIBERDADE DE DESISTÊNCIA

Eu entendo que sou livre para realizar perguntas a qualquer momento ou solicitar que qualquer informação relacionada à minha pessoa não seja incluída no estudo. Eu entendo que participo de livre e espontânea vontade com o único intuito de contribuir para o avanço e desenvolvimento de técnicas e processos para a Engenharia de Software.

EQUIPE: **PESQUISADOR RESPONSÁVEL**
Ana Maria da Mota Moura
Programa de Engenharia de Sistemas e Computação - COPPE/UFRJ

PROFESSORES RESPONSÁVEIS
Profa. Cláudia M.L. Werner
Programa de Engenharia de Sistemas e Computação - COPPE/UFRJ
Profa. Aline Pires Vieira de Vasconcelos
CEFET Campos

Nome (em letra de forma):_____

Assinatura: _____ Data:_____

B.2: Formulários de Avaliação do Modelo de Componentes

Formulário 1 ***Avaliação do Modelo de Componentes*** **Identificação:**
Aplicação: _____ **Engenheiro de Software: _____** **Hora início: _____:_____ Hora fim: _____:_____**
***Questões*:**
Q1. Quantos componentes resultaram da aplicação da estratégia? _____
Q2. Dentre os componentes resultantes quantos você julga que representam uma funcionalidade ou um conceito do domínio? _____
Q3. Quantas interfaces providas representam realmente serviços providos pelo componente sob o ponto de vista dos clientes deste componente? _____
Q4. Quantas interfaces resultaram da aplicação da estratégia? _____
Q5. Quantas interfaces requeridas representam realmente serviços que estes componentes necessitam? _____
Q6. Quantas interfaces requeridas resultaram da aplicação da estratégia? _____

Formulário 2:

Avaliação do Modelo de Componentes

Este é um formulário de perguntas subjetivas, as quais devem ser respondidas com um dos valores do conjunto {"Sim", "Não", "Parcialmente"}, seguido de uma justificativa.

Q7. Os componentes gerados representam elementos, que apóiem a manutenção e a reutilização de partes da aplicação? Esta questão deve ser respondida olhando o modelo de componentes gerado.

_____

_____

_____

_____

_____

Q8. O modelo de componentes poderia ser utilizado para apoiar a reengenharia da aplicação do paradigma de Orientação a Objetos para componentes?

_____

_____

_____

_____

_____

Q9. Em algum momento você desejou reestruturar o modelo de projeto OO para gerar componentes (via análise estática) diferentes dos que foram gerados?

_____

_____

_____

_____

_____

_____

Q10. As sugestões de componentes são de fácil compreensão? Se a resposta for Não ou Parcialmente, favor indicar o(s) tipo(s) de sugestão que precisam de melhorias.

_____
_____
_____
_____
_____

Q11. Em algum momento você desejou obter alguma interface que não foi gerada? Em caso afirmativo informe quais?

_____
_____
_____
_____
_____

Q10. Em algum momento você desejou ter alguma informação, para apoiar a tomada de decisão sobre que componentes gerar, e esta informação não foi fornecida pela estratégia? Se a resposta for Sim ou Parcialmente, favor indicar as informações desejadas.

_____
_____
_____
_____
_____

B.3: Roteiro de Instruções para a Realização do Estudo

Roteiro para a realização de estudo utilizando a abordagem de reestruturação de modelos baseada em métricas para apoiar a reengenharia de software OO para componentes:

13. Preencher o formulário de Caracterização de Participante e assinar o formulário de Consentimento.
14. Ler os formulários de avaliação 1 e 2.
15. Iniciar a marcação do tempo de estudo.
16. Gerar componentes via análise estática com o apoio da ferramenta GenComp.
17. Para a questão 9 do formulário 2, responder o que se pede.
18. Gerar componentes via análise dinâmica com o apoio da ferramenta GenComp.
19. Finalizar a marcação do tempo de estudo.
20. Para as questões do formulário 1, responder em números o que se pede.
21. Para as questões do formulário 2, responder {“Sim”, “Não”, “Parcialmente”} seguido de uma justificativa. Reveja a resposta da questão 9 do formulário 2 baseando-se no modelo completo de componentes.

B.4: Questionário de Caracterização do Participante

Nome _____

Nível (Ms.c/D.Sc.): _____

Experiência em Desenvolvimento de Software

Qual é a sua experiência com desenvolvimento de software na prática? Por favor, inclua na sua resposta o tempo de experiência relevante em desenvolvimento.

_____

_____

_____

_____

_____

_____

Por favor, indique o grau de sua experiência nesta seção seguindo a escala de 5 pontos abaixo:

1 = nenhum

2 = estudei em aula ou em livro

3 = pratiquei em 1 projeto em sala de aula

4 = usei em 1 projeto na indústria

5 = usei em vários projetos na indústria

Experiência em DBC

- Experiência especificando componentes e interfaces 1 2 3 4 5
- Experiência utilizando algum método de DBC 1 2 3 4 5

Experiência no Domínio

- Experiência no domínio da aplicação 1 2 3 4 5

Experiência em Reutilização de Software

- Experiência (prática) com reutilização de software 1 2 3 4 5
- Experiência (prática) com reengenharia de software 1 2 3 4 5