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

Arquitetura de software para reuso de componentes

por

EDUARDO KROTH

Dissertação submetida à avaliação,
como requisito parcial para a obtenção do grau de Mestre
em Ciência da Computação

Prof. Dr. Carlos Alberto Heuser
orientador

Porto Alegre, janeiro de 2000

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Kroth, Eduardo
Arquitetura de software para reuso de componentes
por Eduardo Kroth   Porto Alegre : PPGC da UFRGS, 2000.
69 f.: il.

Dissertação (metrado) - Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, RS-BR, 2000. Orientador: Heuser, Carlos A.

1. Reuso de software. 2. Componente. 3. Framework. 4. Contratos de reuso. 5. Arquitetura de software. 6. Padrões de projeto. I. Heuser, Carlos Alberto. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Profa. Wrana Panizzi
Pro-Reitor de Pos-Graduacao: Prof. Franz Rainer Semmelmann
Diretor do Instituto de Informatica: Prof. Philippe Olivier Alexandre Navaux
Coordenadora do PPGC: Profa. Carla Maria Dal Sasso Freitas
Bibliotecaria-Chefe do Instituto de Informatica: Beatriz Regina Bastos Haro

Agradecimentos

Primeiramente, quero agradecer à minha esposa Cláudia e à minha filha Gabriela, pela compreensão, companhia e apoio neste tempo que dediquei ao mestrado. Sei que são momentos que não se recuperam mais, mas espero que eu possa proporcionar bons momentos daqui em diante. A elas, que se transferiram para Porto Alegre em uma época difícil (Gabriela tinha 6 meses de idade), sou eternamente grato. Elas assumiram tudo isto como uma gostosa aventura. Souberam superar estes três anos muito bem.

Quero agradecer aos meus familiares (Rudi, Teresinha e Karla) pelo incentivo dado desde menino, através dos exemplos de perseverança e de dedicação às causas que se comprometem. Mui grato! Aos meus cunhados e a minha querida sogra, agradeço pelo suporte dado a nossa família.

Agradecimentos aos primos de Porto Alegre que nos receberam tão carinhosamente, dando suporte logístico quando necessário. Foi um bom momento para aproximar-nos.

Agradecimentos aos colegas do departamento de Informática, que incentivaram e orientaram-me na caminhada em busca deste título. Grato, também, a UNISC que proporcionou uma bolsa de afastamento durante dois anos. Espero retribuir por muito tempo. Aos alunos bolsistas e já egressos da graduação, que contribuíram com este trabalho através da programação do Software Assistente e das implementações em Java.

Agradecimento a todos funcionários do Instituto de Informática. Aos colegas /97 e /98, também sou grato pelas boas conversas e troca de idéias durante as horas de folga.

Um agradecimento especial ao professor Heuser, pela maneira como orientou os estudos neste curso, ensinando-me como pesquisar, pensar e escrever de uma forma melhor. Nas várias reuniões que realizamos, tive a oportunidade de assimilar o conhecimento transmitido. Se foi da melhor maneira, deixo ao tempo julgar.

Sumário

Lista de Figuras **6**

Lista de Tabelas **7**

Lista de Abreviaturas **8**

Resumo **9**

Abstract **10**

1 Introdução **11**

2 Contratos de reuso **15**

2.1 Exemplo do tipo de relação considerada **15**

2.2 Contratos de reuso **19**

2.3 Contratos de reuso para a aplicação em questão **21**

2.3.1 Tipo de contrato uso 21

2.3.2 Tipo de contrato implementação 24

2.4 Exemplo aplicando os tipos de contratos definidos **26**

3 Arquitetura de Integração **28**

3.1 Padrão de projeto para o tipo de contrato uso **28**

3.1.1 Padrão de Projeto Decorator 30

3.1.2 Padrão de projeto Facade 31

3.1.3 Classe Roteadora: proposta de implementação do tipo de contrato uso 33

3.1.4 Aplicação do tipo de contrato uso em um estudo de caso 35

3.2 Padrão de projeto para o tipo de contrato implementação **36**

3.2.1 Padrão de Projeto Adapter 38

3.2.2 Classe Implementadora: proposta de implementação do tipo de contrato implementação 38

3.2.3 Aplicação do tipo de contrato implementação em um estudo de caso 40

3.3 Visão geral da arquitetura de integração **41**

4 Software Assistente **43**

4.1 Especificação do software assistente **43**

4.1.1 Especificação do caso de uso "Armazenamento de Componentes" 45

4.1.2 Especificação do caso de uso "Seleção de Componentes" 49

4.1.3 Especificação do caso de uso "Construção de uma aplicação" 50

4.1.4 Especificação do caso de uso "Especificação de contratos de reuso" 51

4.1.5 Especificação do caso de uso “Geração da camada de integração” 54

4.1.6 Relatórios do Software Assistente 55

4.2 Modelo de dados do software assistente **57**

5 Conclusões 61
Anexo 1 - Código das classes de integração 63
Bibliografia 66

Lista de Figuras

FIGURA 1.1 – Arquitetura de software em três camadas 13
FIGURA 2.1 - Exemplo de classes de aplicação e componentes..16
FIGURA 2.2 – Exemplo de dependência de um método..19
FIGURA 2.3 – Exemplo de dependência de método entre classes 20
FIGURA 2.4 – Exemplo de contrato de reuso com tipos de contratos..20
FIGURA 2.5 – Exemplo de uso parcial de um componente com renomeação de métodos .22
FIGURA 2.6 – Exemplo de uso de vários componentes por uma aplicação 24
FIGURA 2.7 – Exemplo de implementação de métodos abstratos de componentes 25
FIGURA 2.8 – Aplicação dos novos tipos de contratos no modelo de Viagens..27
FIGURA 3.1 - Modelo de objetos usando tipo de contrato uso..28
FIGURA 3.2 - Modelo de classes de um visualizador de textos..31
FIGURA 3.3 - Estrutura do padrão de projeto Decorator 31
FIGURA 3.4 – Associação entre componentes e aplicação..32
FIGURA 3.5 - Estrutura do padrão de projeto Facade 32
FIGURA 3.6 - Modelo de objetos usando classes roteadoras..33
FIGURA 3.7 – Aplicação de uma classe *roteadora* 36
FIGURA 3.8 – Exemplos de relação de implementação..37
FIGURA 3.9 – Estrutura do padrão de projeto Adapter..38
FIGURA 3.10 – Estrutura do padrão de projeto Adapter..39
FIGURA 3.11 – Exemplo de relação de implementação 40
FIGURA 3.13 – Visão geral do modelo usando arquitetura de integração 41
FIGURA 3.14 – Arquitetura de integração entre componentes e aplicação..42
FIGURA 4.1 - Diagrama de Use Cases 44
FIGURA 4.2 - Interface: Consultar Componentes 45
FIGURA 4.3 - Interface: Estrutura de Componentes 46
FIGURA 4.4 - Interface: Método de Componente..47
FIGURA 4.5 - Interface: Dependência de métodos 48
FIGURA 4.6 - Interface: Seleção de Componentes 49
FIGURA 4.7 - Interface: Método de classe de aplicação 50
FIGURA 4.8 - Interface: Relacionamento entre classe de aplicação e componente............51
FIGURA 4.9 - Interface: Construção da Aplicação 52
FIGURA 4.10 - Interface: Escolha do tipo de classe para implementação 52
FIGURA 4.11 - Interface: implementação com método de outro componente 53
FIGURA 4.12 – Interface: Seleção do diretório das classes da *camada de integração*.......54
FIGURA 4.13 – Lista de classes gerada da *camada de integração* 55
FIGURA 4.14 – Relatório de Componentes 56
FIGURA 4.15 – Relatório de Relação Aplicação e Componentes..56
FIGURA 4.16 – Modelo de dados do software assistente..57
FIGURA A.1 - Código de uma classe *roteadora* 64
FIGURA A.2 - Código de uma classe *implementadora* 65

Lista de Tabelas

TABELA 1.1 - Relação de uso dos métodos da aplicação 17
TABELA 1.2 - Relação de dependência dos métodos de componentes.............................18
TABELA 1.3 - Relação de implementação dos métodos da aplicação 18
TABELA 3.1 – Funcionalidades de uma classe roteadora 34
TABELA 3.2 – Funcionalidades de uma classe implementadora......................................39
TABELA 4.1 – Relação das tabelas do software assistente 58

Lista de Abreviaturas

UML	Unified Moddeling Language
CASE	Computer Aided Software Engineering
MVC	Model-View-Control - "architectural pattern"

Resumo

A dissertação trata do reuso de software. No contexto de reuso, estuda-se o problema de construção de aplicações a partir de componentes de software pré-existentes.

São considerados componentes do tipo “white box”. Esses componentes podem conter métodos concretos, “template” e abstratos, estes últimos exigem uma implementação, quando os componentes são utilizados. Os componentes considerados no trabalho modelam o domínio de problema de uma aplicação.

A dissertação apresenta uma técnica que permite que o construtor de uma aplicação especifique a relação desejada entre as classes da aplicação em construção e um conjunto de componentes pré-existentes. Esta técnica de especificação é suportada por uma notação gráfica e inspirada em uma técnica existente na literatura, chamada de *contratos de reuso*. São apresentados dois novos tipos de contratos de reuso, para modelar as relações entre aplicação e componentes: os contratos denominados *uso* e *implementação.*

Para permitir a implementação dos contratos de reuso, é proposta uma *arquitetura de integração*. Nesta arquitetura o software possui três camadas: a de *componentes*, a de *integração* e a da *aplicação.* A camada de *integração* implementa as relações entre componentes e aplicação. Esta camada pode ser gerada de forma automática, a partir da especificação da relação entre aplicação e componentes através de contratos de reuso.

Usando padrões de projeto existentes na literatura, a dissertação apresenta dois padrões de projeto, destinados a implementar os contratos de reuso. Estes padrões de projeto contém as regras através das quais a camada de integração deve ser construída.

Finalmente, é apresentado um software assistente que gera automaticamente as classes da camada de *integração*. O software assistente possui uma interface que possibilita a definição das relações entre componentes e classes da aplicação, ou seja a especificação de contratos de reuso. Os componentes estão armazenados em um repositório.

Palavras-chave: reuso de software, componente, framework, contratos de reuso, arquitetura de software, padrões de projeto.

TITLE: "SOFTWARE ARCHITECTURE FOR COMPONENTS REUSE"

Abstract

The thesis deals with software reuse. In this context, it studies the problem of developing applications from preexisting software components.

White box components are considered here. *White box components* may contain template methods, that demand an implementation, when the components are used. The components considered in this work are part of the problem domain layer of an application.

The thesis presents a technique that allows specification of the desired relation between the application classes in construction and a set of preexisting components. This specification technique is based on an existing technique, called *reuse contracts*, It is supported by a graphical notation The thesis presents two new types of *reuse contracts* in order to model the relation between application and components. These contracts are called *use* and *implementation* contracts.

To allow the implementation of reuse contracts, an *integration architecture* is proposed. In this software architecture there are three layers: the *components layer*, the *integration layer* and the *application layer*. The integration layer implements the relations between components and application. This layer can be generated automatically from the specification of the relation between application and components, made through *reuse contracts*.

On the basis of design patterns that appear in literature, the thesis presents two design patterns, oriented to implement reuse contracts. These design patterns contain the rules through which the integration layer must be constructed.

Finally, an assistant software is presented which automatically generates the classes of the integration layer. The assistant software offers an interface that allows the definition of the the reuse contracts that specify the relation one intends to implement between his application and a set of existing components. The components are stored in a repository.

Keywords: software reuse, component, framework, reuse contracts, design patterns

1 Introdução

O reuso de software é uma abordagem dentro da Engenharia de Software que enfoca o reaproveitamento sistemático das três fases do desenvolvimento de sistemas: análise, projeto e codificação. Atualmente, o reuso apresenta resultados mais concretos na fase de codificação, devido a existência de bibliotecas de classes e de componentes disponíveis em larga escala. A fase de análise tem uma prática de reuso mais restrita, pois não possui tecnologia suficientemente madura [JGS97].

Uma das técnicas existentes para facilitar o reuso de software é a orientação a objetos [BOO94]. Entretanto, a simples adoção da tecnologia de objetos, sem a existência de um padrão de reuso explícito e um processo de desenvolvimento de software orientado ao reuso, provavelmente não fornecerá o sucesso esperado no reuso em larga escala. Em outras palavras, a simples adoção da tecnologia de objetos não traz vantagens de reuso, fazendo-se necessário um processo específico que forneça o reuso de software [JAC97].

Outra técnica que propõe reuso é a aplicação de “framework”. Um "framework" é uma estrutura de classes interrelacionadas, que constitui uma implementação inacabada, para um conjunto de aplicações de um domínio [TAL95]. Além de permitir a reutilização de um conjunto de classes, um "framework" também minimiza o esforço de desenvolvimento de novas aplicações, pois já contém a definição de arquitetura geradas a partir dele bem como, tem predefinido o fluxo de controle da aplicação.

Um “framework” pode ser classificado de acordo com a visibilidade da sua estrutura interna (“black box” e “white box”) e também, de acordo com sua aplicabilidade (*frameworks horizontais* e *frameworks verticais*) [FAY97].

Frameworks horizontais são "frameworks" que podem ser aplicados a qualquer domínio de problema, pois são desenvolvidos de forma genérica sem preocupação com domínios específicos. Exemplos de *frameworks horizontais* são os que implementam infra-estruturas de comunicação de dados, de interface de usuários e de gerenciamento de dados (persistência de objetos). Em alguns casos, *frameworks horizontais* implementam padrões de arquitetura de software ("architectural patterns"), como por exemplo, o MVC (Model-View-Control) [BUS95].

Frameworks verticais são "frameworks" desenvolvidos com orientação para um domínio de aplicação específico. Para exemplificar, pode-se considerar um "framework" para controle de produção industrial. Neste domínio de problema, existem características comuns a várias indústrias. O "framework" é então construído de forma que atenda as características comuns, deixando as específicas para serem implementadas em aplicações construídas a partir dele.

O "framework" do tipo “black box” disponibiliza somente sua interface (assinatura dos métodos públicos). O modo como é implementada a funcionalidade interna não é visível ao usuário. Este encapsulamento facilita seu uso, pois o usuário não necessita

conhecer o funcionamento interno, mas também, reduz sua flexibilidade, pois não permite que o usuário adapte novas características [JOH88].

O "framework" do tipo *"white box"* tem a sua estrutura interna visível ao usuário. Esta abertura de código permite que o usuário do "framework" possa estudar sua funcionalidade e também possa vir a fazer algumas alterações implementadas externamente à classe do "framework" ("override"*)* [BOS97]. O uso de um "framework white box" requer que o desenvolvedor conheça a funcionalidade interna de suas classes, exigindo um envolvimento maior com a sua estrutura.

Este trabalho trata do reuso de software a nível de domínio de problema e deseja permitir a extensão de software pré-existente portanto usará "frameworks" verticais e do tipo "white box".

A especialização de um "framework" requer que o desenvolvedor da aplicação implemente os métodos abstratos. Como a concepção de um "framework" não permite que sua estrutura seja modificada para atender a um problema específico, o usuário do "framework" deve prover uma adaptação de sub-classes para implementar a especialização. Além de métodos abstratos, o desenvolvedor da aplicação pode necessitar que métodos concretos do "framework" sejam alterados ou novas funções sejam adicionadas a ele.

Um *framework vertical* tem alto custo de desenvolvimento, pois requer um planejamento complexo, demorando para ser construído. Para minimizar este alto custo, propõe-se a divisão de um "framework" em pequenos "frameworks", aqui chamados de "mini-frameworks´, que podem atender problemas menores dentro de um determinado domínio. Estes "mini-frameworks" são equivalentes ao que outros autores chamam de "framelets". [PRE99].

Outro conceito utilizado na área de reuso de software é o de *componente* [BEL98, ORT98]. Um componente de software é definido como uma unidade de composição de software, que possui um conjunto de interfaces e um conjunto de requisitos. Um componente pode ser desenvolvido, adquirido, incorporado ao sistema ou composto por outros componentes de forma independente. Assim como "frameworks", componentes também podem ser classificados segundo a visibilidade de sua estrutura interna em "black box" e "white box" [SZY98].

Os "mini-framework" tratados neste trabalho são em tudo análogos ao componentes "white box" [JOH97]. Por esta razão, passa-se a utilizar o termo componente como sinônimo de "mini-framework".

Assim, este trabalho trata do reuso de componentes pré-construídos para a construção da camada de domínio de problema de uma aplicação. Considera-se que os componentes são do tipo "white-box", isto é, que suas funcionalidades podem ser estendidas pela aplicação e também, que os componentes sejam relativos a um "framework" pequeno, isto é, implementem algumas poucas classes de um domínio de problema.

Um ponto importante a ser tratado neste contexto, é o de como documentar a relação desejada entre uma aplicação que está sendo projetada e um conjunto de componentes existentes [LAM93]. Uma técnica para tal é chamada de *contratos de reuso* [LUC96]. Um contrato de reuso é uma especificação conceitual, abstrata, de como uma classe de aplicação se relaciona com um conjunto de componentes, ou seja, quais propriedades de cada componente são chamadas dentro da aplicação, que propriedades de cada componente são implementadas dentro da aplicação e assim por diante. O termo "contrato" é utilizado para indicar que um contrato de reuso estabelece um compromisso entre um especificador do reuso de um conjunto de componentes (especificador da aplicação) e o implementador que decide como esta relação será efetivamente implementada.

Uma arquitetura de software pode ser dividida em três camadas: camada de *interface homem-máquina*, camada de *domínio do problema* e camada de *gerenciamento de dados* [ALL98] (figura 1.1). Este trabalho trata especificamente da camada de *domínio do problema*.

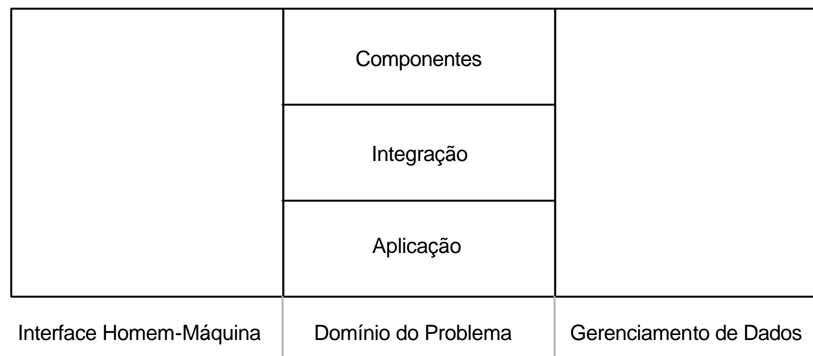

FIGURA 1.1 – Arquitetura de software em três camadas

Uma vez especificada conceitualmente a relação entre componentes e aplicação, é necessário implementá-la. Este trabalho propõe para tal uma arquitetura de software onde a camada de *domínio do problema* é dividida em três outras camadas, a de *componentes*, a da *aplicação* e a de *integração*, que como o nome indica, estabelece a ligação entre as camadas precedentes. A camada de integração, além de separar mais claramente as responsabilidades dentro da arquitetura do software, contém um conjunto de classes que podem ser especificadas automaticamente a partir dos contratos de reuso [SIL96].

Especificamente, no contexto do reuso de software a nível de domínio de problemas, este trabalho apresenta:

- uma técnica, baseada em *contratos de reuso*, para especificar conceitualmente a relação desejada entre uma aplicação e um conjunto de componentes;
- uma arquitetura de software que permite implementar a relação especificada pelos contratos de reuso através de uma *camada de integração*;
- um software assistente que permite gerar automaticamente, a partir dos contratos de reuso, a camada de integração.

O trabalho está organizado como segue.

O capítulo 2 apresenta os *contratos de reuso*. Estes contratos são analisados frente ao problema aqui tratado e dois novos tipos de contratos de reuso são propostos.

O capítulo 3 descreve a camada de integração. Esta descrição é feita com suporte de alguns padrões de projeto [GAM94].

O capítulo 4 apresenta a especificação do software assistente que permite gerar a camada de integração a partir dos contratos de reuso.

2 Contratos de reuso

Este capítulo discute a relação entre um conjunto de componentes pré-existentes e uma aplicação em construção. Apresenta *contratos de reuso* [LUC96], como uma forma de especificar esta relação.

2.1 Exemplo do tipo de relação considerada

Um exemplo para mostrar as relações entre uma aplicação e os componentes que estão sendo considerados neste trabalho é apresentado nesta seção.

A figura 2.1 apresenta um conjunto de classes de componentes e de classes de aplicação. A notação utilizada é a da UML[ERI98]. Os componentes são ExecuçãoPlano, TransaçãoLugar e TransaçãoLinha, representados através de pacotes [DSO99]. Estes componentes, como outros que aparecem nesta dissertação, são inspirados em um conjunto de padrões de análise apresentados em [COA97].

O componente ExecuçãoPlano trata de um problema onde um *plano* é formado por fases, etapas. Este componente é composto pelas classes Plano e ExecuçãoPlano. Por exemplo, a classe Plano pode representar um plano de execução de uma obra, uma linha de produção de uma fábrica ou um planejamento de atividades para algum objetivo. A classe ExecuçãoPlano representa as fases da execução de uma obra, as fases da linha de produção ou as atividades de um planejamento. O método Plano.calculaTotalPlano calcula o custo total de um plano. O método ExecuçãoPlano.calculaDuração calcula a duração de cada fase do plano. Os métodos abstratos ExecuçãoPlano.getDataHoraInicio e ExecuçãoPlano.getDataHoraFim são métodos que, quando implementados, devem retornar um valor referente ao início e ao fim de uma determinada fase.

O componente TransaçãoLugar trata de um problema onde *transações* acontecem em um *lugar*. Este componente é formado pelas classes Lugar e TransaçãoLugar. Por exemplo, a classe TransaçãoLugar pode representar uma nota fiscal, um contrato assinado, um ponto em uma excursão de viagem. A classe Lugar pode representar o endereço de uma filial, o local da assinatura do contrato ou um ponto de parada em uma excursão de viagens. A classe TransaçãoLugar possui um método abstrato calculaTransação que tem a funcionalidade de retornar o valor da transação. Os métodos da classe Lugar tratam de contar o número de transações para um determinado lugar (contaTransações), efetuar algum cálculo sobre as transações (calculaSobreTrans), somar as transações efetuadas para o determinado lugar (somaTransações) e classificar as transações por alguma ordem (classificaTrans).

O componente TransaçãoLinha trata de um problema onde uma *transação* contém várias *linhas de transação*. Este componente é formado pelas classes Transação e TransaçãoLinha. A classe Transação pode representar um orçamento, um pedido, uma nota fiscal. A classe LinhaTransação pode representar itens de um orçamento, itens de um pedido ou itens de uma nota fiscal. O método Transação.contaLinhas conta as linhas de uma transação. O método Transação.somaLinhas soma os valores das linhas agregadas a

uma transação. O método TransaçãoLinha.calculaLinha efetua a multiplicação da quantidade pelo valor unitário. A obtenção destes valores é a funcionalidade dos métodos abstratos getQuantidade e getValor.

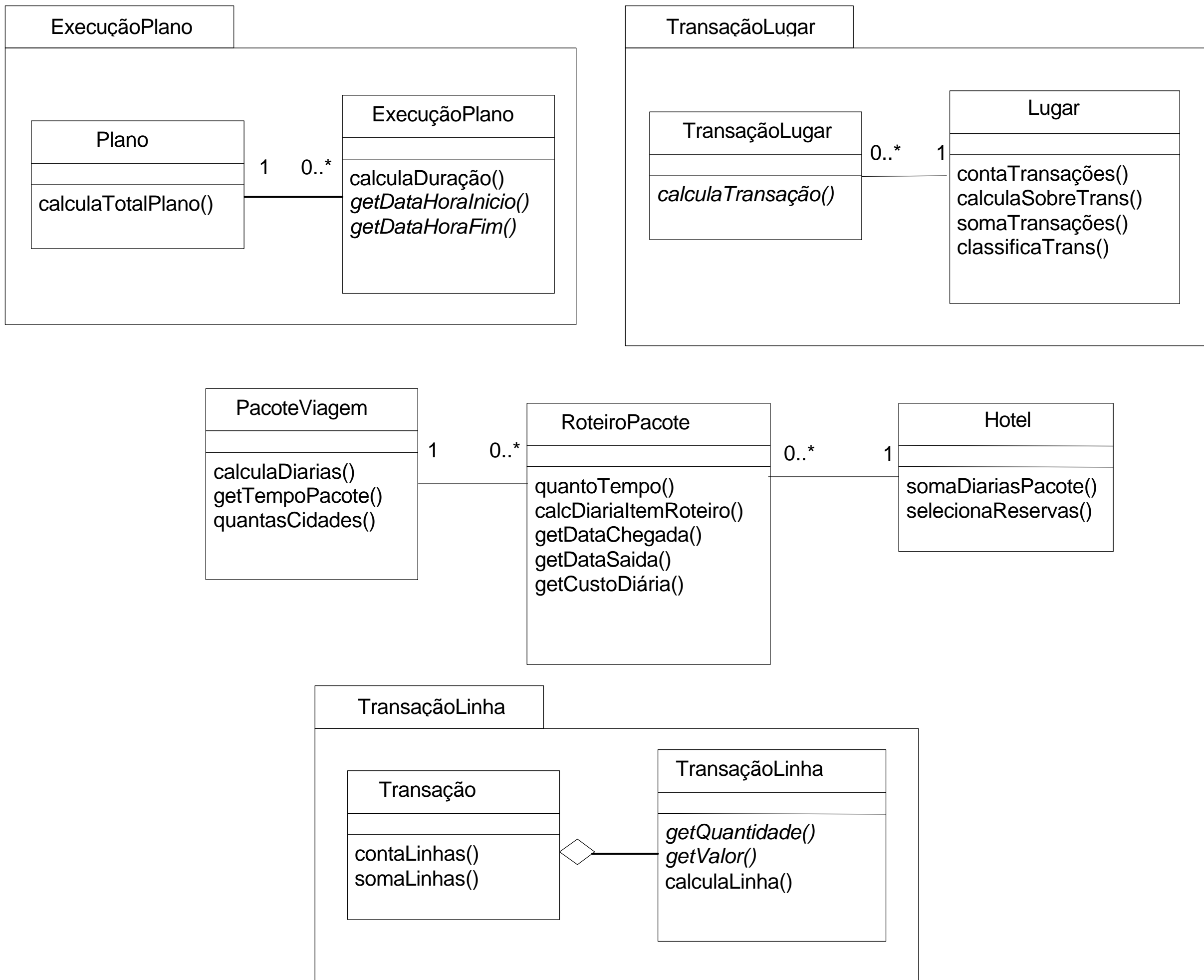

FIGURA 2.1 - Exemplo de classes de aplicação e componentes

As classes PacoteViagem, RoteiroPacote e Hotel são as classes da aplicação que deseja-se construir.

A classe PacoteViagem trata de um determinado pacote de viagens. Um pacote de viagens é composto por RoteiroPacote, classe que representa os pontos que formam o pacote de viagens. Cada ponto da viagem possui um Hotel para hospedagem dos clientes da agência.

A classe de aplicação PacoteViagem possui métodos para calcular a soma das diárias de um pacote (calculaDiarias), para retornar qual é a duração de um determinado pacote (getTempoPacote) e para retornar o número de cidades que compõe o pacote (quantasCidades).

A classe de aplicação RoteiroPacote é composta por métodos concretos para retornar qual a duração daquela etapa da viagem (quantoTempo), para calcular as diárias gastas em um ponto da viagem (calcDiariaItemRoteiro) e para retornar o custo de uma diária (getCustoDiária). Além dos métodos concretos, a classe contém os métodos abstratos para retornar a data de chegada em um determinado ponto do roteiro (getDataChegada) e a data de saída de um determinado ponto do roteiro (getDataSaida).

A classe de aplicação Hotel possui dois métodos, um soma as diárias de um determinado pacote (somaDiariasPacote) e o outro seleciona as reservas de um determinado pacote (selecionaReservas).

Há vários tipos diferentes de relações que se deseja estabelecer entre métodos de classes da aplicação e métodos de componentes.

Por exemplo, deseja-se que o método de aplicação PacoteViagem.calculaDiarias() possua a funcionalidade já implementada pelo método Transação.somaLinhas(), que implementa a soma dos valores das várias linhas de uma transação. A esta relação entre métodos, dá-se o nome de relação de *uso*, e diz-se que o método PacoteViagem.calculaDiarias() usa o método Transação.somaLinhas().

A tabela abaixo mostra os métodos das classes da aplicação que usam a implementação de métodos dos componentes, no caso estudado.

TABELA 1.1 - Relação de *uso* dos métodos da aplicação

RoteiroPacote

quantoTempo()	usa	ExecuçãoPlano.calculaDuração()
calculaDiariaItemRoteiro()	usa	TransaçãoLinha.calculaLinha()

Hotel

somaDiariasPacote()	usa	Lugar.somaTransações()
selecionaReservas()	usa	Lugar.classificaTransações()

PacoteViagem

quantasCidades()	usa	Transação.contaLinhas()
calculaCustoDiarias()	usa	Transação.somaLinhas()
getTempoPacote()	usa	Plano.calculaTotalPlano()

Quando o desenvolvedor da aplicação usar métodos "template" de componente [PRE94], ele deve providenciar a implementação dos métodos abstratos, que participam deste método "template". Para tal é necessário que ele conheça quais os métodos de componente que são referenciados dentro de cada método "template". Esta relação entre métodos de componente é chamada de relação de *dependência*.

A tabela apresentada a seguir relaciona os métodos de componentes que dependem de outros métodos para suas execuções.

TABELA 1.2 - Relação de *dependência* dos métodos de componentes

Lugar

somaTransações()	depende de	TransaçãoLugar.CalculaTransação()
ContaTransações()	depende de	TransaçãoLugar
ClassificaTrans ()	depende de	TransaçãoLugar

Plano

CalculaTotalPlano()	depende de	ExecuçãoPlano.calculaDuração()

ExecuçãoPlano

CalculaDuração()	depende de	ExecuçãoPlano.getDataHoraInicio() ExecuçãoPlano.getDataHoraFim()

Transação

contaLinhas()	depende de	TransaçãoLinha
somaLinhas()	depende de	TransaçãoLinha.calculaLinha()

TransaçãoLinha

calculaLinha()	depende de	getQuantidade() e getValor()

Os métodos abstratos referenciados por métodos "template" que foram escolhidos pelo desenvolvedor da aplicação necessitam ser implementados [MEI96]. É necessário também, especificar que método da aplicação, ou de outro componente, implementa cada método abstrato referenciado.

Por exemplo, o método RoteiroPacote.quantoTempo usa o método ExecuçãoPlano.calculaDuração. Esse método é um método "template" e referencia dois métodos abstratos da mesma classe: getDataHoraInicio e getDataHoraFim. Portanto, esses dois métodos abstratos devem ser implementados caso queira se utilizar o método ExecuçãoPlano.calculaDuração. Este é um caso de método de componente sendo implementado por um método de aplicação.

Também métodos de componente podem ser usados para implementar métodos abstratos de componente. Por exemplo, o método TransaçãoLugar.calculaTransação é usado por um método "template" da classe Lugar. No caso estudado, este método é implementado por outro de componente: TransaçãoLinha.calculaLinha.

Esta relação entre métodos é chamada de relação de *implementação*. A próxima tabela mostra as relações necessárias para a implementação dos métodos abstratos. Um método abstrato de componente pode ser implementado por um método de uma classe da aplicação ou por um método de outro componente.

TABELA 1.3 - Relação de *implementação* dos métodos da aplicação

TransaçãoLugar

calculaTransação()	implementado por componente	TransaçãoLinha.calculaLinha()

ExecuçãoPlano

getDataHoraInicio()	implementado por aplicação	RoteiroPacote.getDataSaida()
getDataHoraFim()	implementado por aplicação	RoteiroPacote.getDataChegada()

TransaçãoLinha

getQuantidade()	implementado por aplicação	RoteiroPacote.quantoTempo()
getValor()	implementado por aplicação	RoteiroPacote.getCustoDiaria()

2.2 Contratos de reuso

Contratos de reuso [MEN96] é uma técnica para representar formalmente a relação entre uma classe de aplicação e componentes. Está sendo desenvolvida por um grupo de pesquisa do Laboratório de Tecnologia da Programação (PROG) da Universidade Vrije de Bruxelas, Bélgica [DHO98]. A notação em questão foi criada em 1996 [LUC96], ainda não baseada em UML. Trabalhos recentes [MEN98a,MEN98b] adaptaram a notação para UML.

Contratos de reuso usam componentes "white-box", pois documentam a relação entre métodos de componentes e de classes da aplicação. Em uma relação entre componentes e aplicação, um método do componente referenciado por um método da aplicação pode depender de outros métodos, inclusive abstratos. Estes referenciados pelo método de componente escolhido devem ser implementados, portanto, *contratos de reuso* tratam de documentar esta dependência entre métodos de componentes. Na notação original, a relação de dependência entre métodos de componentes é denominada de *evolução de componentes*. Neste trabalho, esta relação será denominada de *dependência de métodos*.

A *dependência de métodos* recebe duas formas de notação: uma no modelo de classes e outra através de *diagramas de colaboração* da UML. No modelo de classes, a lista de métodos dependentes é descrita entre colchetes “[ ]” após a assinatura do método. A figura 2.3 mostra o comportamento do método ExecuçãoPlano.calculaDuração que necessita dos métodos getDataHoraInicio e getDataHoraFim para a sua execução.

A *dependência de métodos* é relevante neste trabalho pois apresenta os métodos abstratos utilizados por um método que é *usado* por um método da aplicação. O software assistente utiliza estes métodos abstratos para apresentar ao desenvolvedor da aplicação com o objetivo de que ele possa providenciar a implementação de tais métodos abstratos. A figura 2.2 mostra um exemplo onde o método calculaDuração depende dos métodos getDataHoraInicio e getDataHoraFim. Caso o desenvolvedor da aplicação referenciar o método calculaDuração, ele deve providenciar a implementação dos dois métodos abstratos.

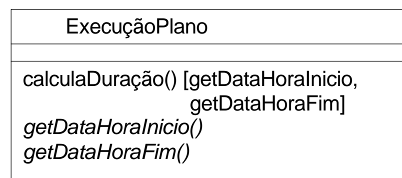

FIGURA 2.2 – Exemplo de dependência de um método

Para métodos que dependem dos métodos que estão em outra classe do componente, a notação deve indicar o nome da classe correspondente. Por exemplo, a figura 2.3 mostra o

método Lugar.somaTransações que depende do método abstrato TransaçãoLugar.calculaTransação.

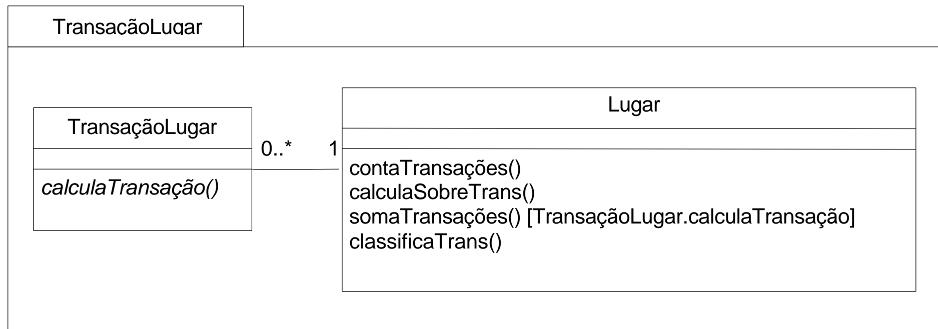

FIGURA 2.3 – Exemplo de dependência de método entre classes

As relações entre componentes e classes de aplicação podem ser classificadas em tipos básicos chamados de *tipos de contratos de reuso*. Um exemplo de tipo de contrato é a extensão, quando a classe cliente define métodos adicionais aos de suas classes provedoras. Já o tipo de contrato concretização estabelece que um método abstrato de uma classe provedora será implementado por um método de uma classe de aplicação. Cada método de uma classe cliente pode estar relacionado de acordo com diferentes tipos de contratos de reuso com uma determinada classe provedora.

São os seguintes os tipos de contratos de reuso definidos originalmente: concretização e abstração; extensão e cancelamento; refinamento e engrossamento [STE96]. Na notação UML, é criado um estereótipo para cada tipo de contrato, sendo este estereótipo indicado junto à associação de dependência.

O tipo de contrato abstração indica que o método concreto do componente torna-se um método abstrato na aplicação. O tipo de contrato concretização é usado quando um método abstrato definido no componente é implementado na classe da aplicação. A figura 2.4 mostra o método Coleção.faz() como um método abstrato, tornando-o um método concreto na classe Conjunto.

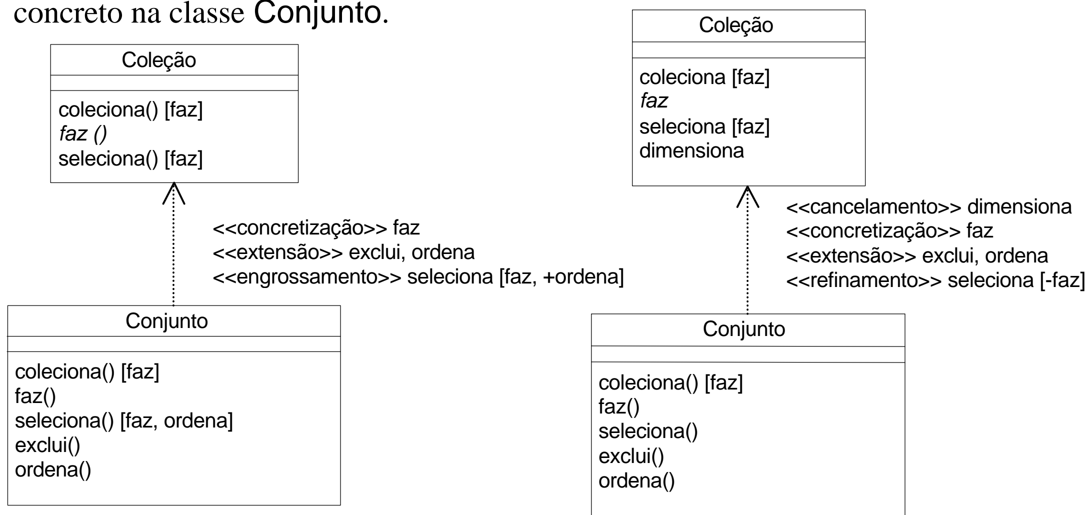

FIGURA 2.4 – Exemplo de contrato de reuso com tipos de contratos

O tipo de contrato extensão indica os métodos que não existem no componente e que são incluídos na classe da aplicação. A figura 2.4 mostra o tipo de contrato extensão, onde os métodos exclui() e ordena() são métodos a serem incluídos na subclasse Conjunto.

O tipo de contrato cancelamento indica os métodos de componente que não são aproveitados no componente. A exclusão de um método ocorre quando o desenvolvedor da aplicação não deseja que ele exista na classe de aplicação. O método Coleção.dimensiona() existente na classe do componente não deve existir na classe da aplicação Conjunto.

O tipo de contrato refinamento indica os métodos a serem incluídos na lista de *métodos referenciados*. O tipo de contrato engrossamento indica os métodos que devem ser retirados da lista de *dependência de métodos*. Estas operações usam o sinal de + (mais) para a adição e de - (menos) para a exclusão de métodos nas estruturas de classes. A figura 2.4 mostra a lista de métodos dependente do método Coleção.seleciona() sendo acrescido pelo método ordena(). Como exemplo de refinamento, o método Coleção.seleciona() tem o método faz() excluído de sua lista de métodos dependentes.

2.3 Contratos de reuso para a aplicação em questão

Para o problema tratado neste trabalho, os contratos de reuso, como apresentados acima, não são suficientes. São necessários outros tipos de relações entre métodos, por isso, essa seção apresenta os tipos de relação entre métodos que são necessários para o problema em consideração, bem como dois tipos de contratos de reuso aplicáveis a este problema.

Cada tipo de contrato, definido nesta seção, é especificado por sua sintaxe e por suas propriedades. A sintaxe especifica a forma de escrita e de leitura de um tipo de contrato e as propriedades descrevem as características de cada tipo de contrato. As funcionalidades de cada tipo de contrato, descritas na seção 2.5, servirão para a construção da arquitetura de integração a ser desenvolvida no próximo capítulo.

2.3.1 Tipo de contrato uso

O tipo de contrato uso trata de métodos da aplicação que referenciam métodos de componentes com o objetivo de usar a implementação do método do componente. Para isto, a funcionalidade do método da aplicação deve ser a mesma que a do método do componente. Existem três propriedades que caracterizam esse tipo de contrato, diferenciando-o de uma simples herança de métodos:

- possibilidade de renomear métodos de um componente quando usado em uma aplicação;
- uso parcial dos métodos de um componente;
- associação de uma classe da aplicação com várias classes de componentes.

A sintaxe do tipo de contrato uso é a seguinte:

uso <método do componente> - <método da aplicação>

onde se lê *<método do componente> é usado por <método da aplicação>*.

Propriedade 1: Possibilidade de renomear métodos de um componente

Esta propriedade permite que o nome de um método da aplicação, que está referenciando um método do componente, não seja o mesmo do método referenciado, dando ao desenvolvedor da aplicação liberdade em escolher outro nome para o método da aplicação.

Esta propriedade provoca uma associação entre aplicação e componente, onde o método da aplicação apenas invoca o método do componente, motivando a funcionalidade 3- *invocar os métodos de componentes*[1].

A figura 2.5 mostra que o desenvolvedor da aplicação define o método Lugar.somaTransações sendo usado pelo método de aplicação Hotel.somaDiariasPacote. Da mesma maneira, o método Lugar.classificaTrans recebe o nome de Hotel.selecionaReservas quando utilizado na classe da aplicação.

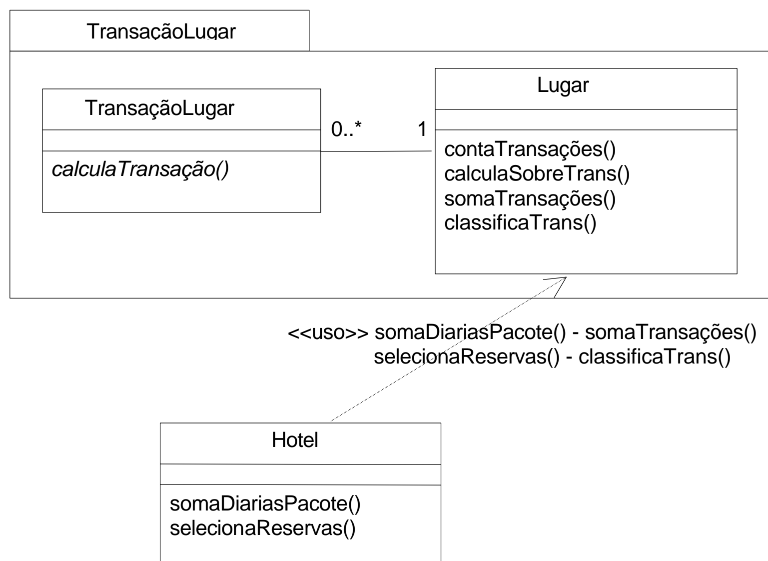

FIGURA 2.5 – Exemplo de uso parcial de um componente com renomeação de métodos

Propriedade 2: uso parcial dos métodos de um componente

[1] As funcionalidades 1,2,3 e 4 necessárias para a implementação do tipo de contrato uso serão definidas no momento oportuno.

A propriedade estabelece que o desenvolvedor da aplicação tenha a liberdade de escolher somente os métodos de componente que considera necessários para a classe de aplicação. *Contratos de reuso* usa o tipo de contrato cancelamento para excluir os métodos não desejados. Ao contrário do tipo de contrato cancelamento, o tipo de contrato uso propõe que se informe os métodos de componente escolhidos.

A figura 2.5 apresenta a relação entre a classe de aplicação Hotel e a classe de componente Lugar, onde os seguintes métodos são referenciados, constituindo uma relação de uso:

- Hotel.somaDiariasPacote usa Lugar.somaTransações
- Hotel.selecionaReservas usa Lugar.classificaTrans

Se fosse aplicado o tipo de contrato cancelamento, o desenvolvedor da aplicação deveria informar os métodos que não são usados. No caso do exemplo em questão, os métodos Lugar.contaTransações() e calculaSobreTrans() devem ser cancelados na relação com a classe Hotel.

Propriedade 3: associação de uma classe de aplicação com várias classes de componentes

Na construção de uma classe de aplicação, o desenvolvedor da aplicação pode ter a necessidade de escolher mais do que um componente para associar a uma aplicação. Esta necessidade ocorre quando a implementação dos métodos da classe de aplicação estão em componentes diferentes por isso, esta propriedade permite que uma classe de aplicação possa se referenciar com vários componentes.

Esta propriedade é necessária pois a solução para a construção de uma classe da aplicação não está, obrigatoriamente, armazenada em um só componente. A propriedade requer que objetos sejam instanciados nos correspondentes objetos associados, motivando as funcionalidades *1 (instanciar e destruir objetos de componentes)* e *2 (possuir referência para os objetos dos componentes relacionados)*.

A figura 2.6 apresenta a classe da aplicação RoteiroPacote associando-se com duas classes de componentes: ExecuçãoPlano e TransaçãoLinha.

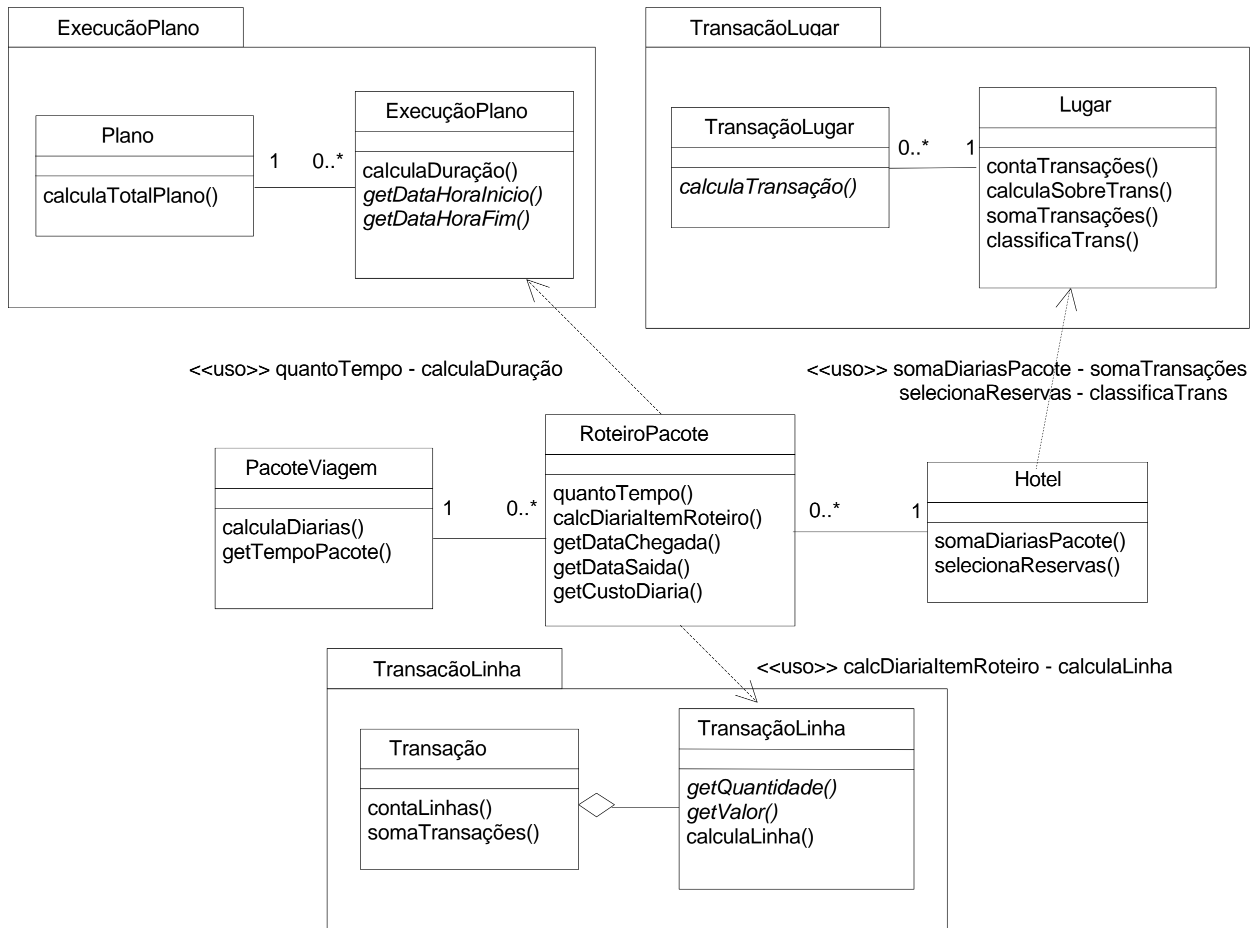

FIGURA 2.6 – Exemplo de uso de vários componentes por uma aplicação

2.3.2 Tipo de contrato implementação

O tipo de contrato implementação trata da implementação de métodos abstratos de componentes. Este tipo de contrato é aplicado quando um método "template" de componente é referenciado e, consequentemente, os métodos abstratos da *lista de dependência* devem ser implementados. O desenvolvedor da aplicação deve providenciar a implementação do método abstrato, que pode estar em um método da aplicação ou em método de outro componente.

A sintaxe do tipo de contrato é a seguinte:

impl <método do componente> - <método da aplicação>,

onde se lê *<método do componente> é implementado por <método da aplicação>.*

O tipo de contrato implementação apresenta duas propriedades: (i) referência de um método abstrato a um método concreto e (ii) possibilidade de nomes diferentes entre métodos abstratos e concretos.

Propriedade 1: Referência de um método abstrato a um método concreto

A implementação de um método abstrato de componente possui duas opções: (i) usar um método da aplicação ou (ii) usar um método de outro componente. A propriedade 1 diz que a relação implementação deve indicar qual o método concreto que fornece a implementação de um método abstrato de componente.

Por exemplo, a figura 2.7 mostra o método ExecuçãoPlano.getDataHoraInicio() necessitando de uma implementação e o desenvolvedor da aplicação especificando o método RoteiroPacote.getDataSaida() para tal.

Propriedade 2: Possibilidade de nomes diferentes entre métodos abstratos e concretos

A propriedade especifica que o método de implementação não necessita ter o mesmo nome do método do componente. O exemplo citado na propriedade anterior ilustra esta propriedade.

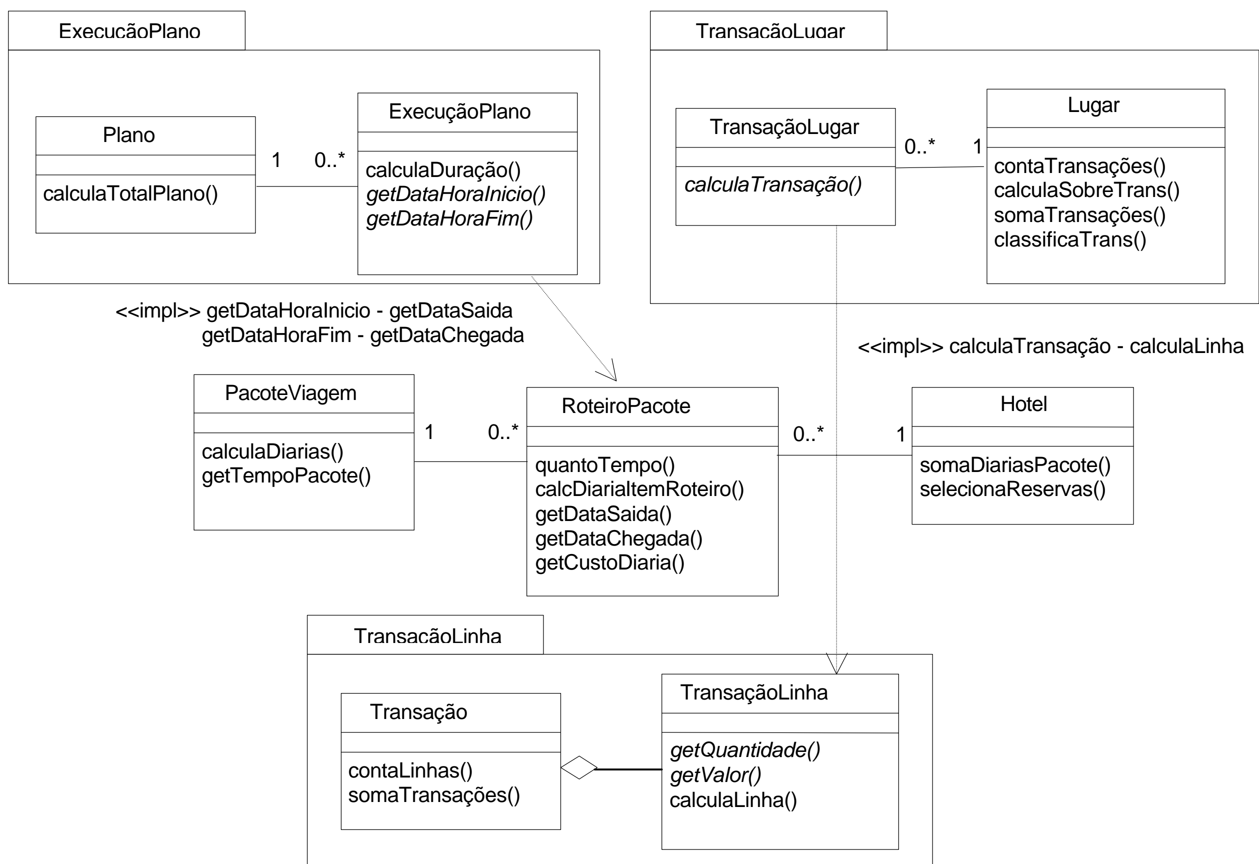

FIGURA 2.7 – Exemplo de implementação de métodos abstratos de componentes

O desenvolvedor da aplicação, quando escolhe outro componente para usar a implementação de um método, deve observar se as especificações de funcionalidade são satisfatórias. No exemplo da figura 2.7, o desenvolvedor da aplicação estipula que os métodos abstratos da classe ExecuçãoPlano usem as implementações feitas na classe de aplicação RoteiroPacote. Os métodos abstratos da classe ExecuçãoPlano são requeridos pelo método ExecuçãoPlano.calculaTransação e são métodos que retornam conteúdo de atributos necessários para o método chamador.

O método TransaçãoLugar.calculaTransação é um método abstrato neste componente, porém possui componentes semelhantes que implementam este método com uma funcionalidade semelhante. Deste modo, o desenvolvedor da aplicação escolhe outro componente para que possa referenciar a sua implementação.

2.4 Exemplo aplicando os tipos de contratos definidos

Na seção 2.1 foram descritas, de maneira textual, as relações existentes entre a aplicação de uma agência de viagens e um conjunto de componentes. Nesta seção, as relações definidas entre a aplicação e os componentes são apresentadas através dos novos tipos de contratos uso e implementação. Cada tipo de contrato usa um estereótipo da associação de dependência (UML) para representar sua notação, assim como definido pelos *contratos de reuso.*

Os tipos de contrato implementação são apresentados em dependências partindo da classe do componente em direção a uma classe da aplicação, ou a uma outra classe de componente. A associação de dependência é usada pois ela não altera a estrutura interna do componente, servindo somente de referência entre as classes. Outros tipos de associação (por exemplo, herança, associação unidirecional ou agregação) não são indicadas para este caso, pois alteram a estrutura interna do componente.

A figura 2.8 mostra o modelo de classes usando os tipos de contratos para especificar as relações existentes entre componentes e classes de aplicação. Do modo como estão representados os tipos de contratos uso e implementação, um gerador de código de uma ferramenta CASE (por exemplo, Rational Rose [RAT99]) não produz nenhuma linha de código para atender tais associações de dependência. A partir deste modelo representado na figura, o trabalho conduz para uma forma de implementação dos tipos de contratos.

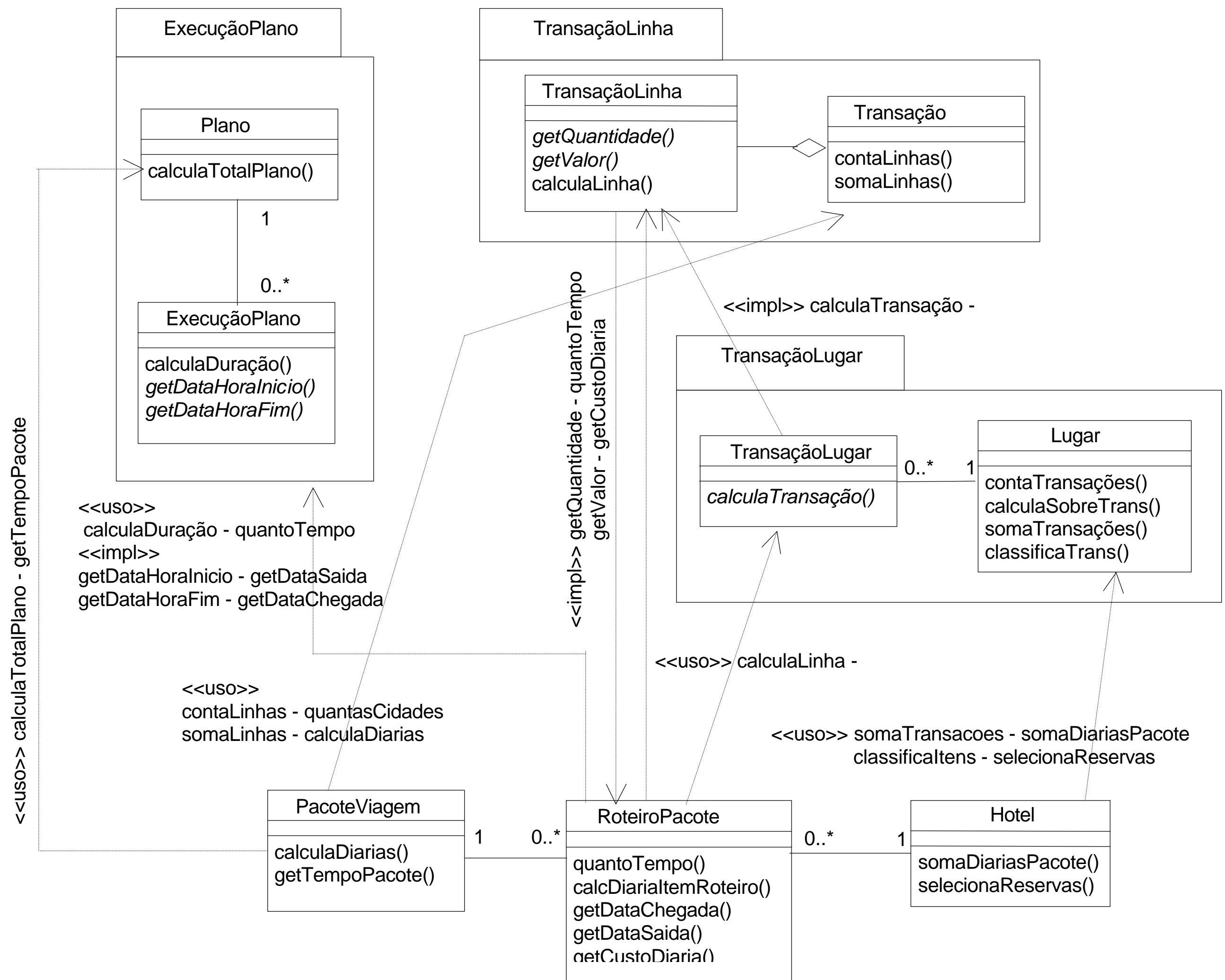

FIGURA 2.8 – Aplicação dos novos tipos de contratos no modelo de Viagens

3 Arquitetura de Integração

Este capítulo apresenta uma arquitetura de classes para a implementação dos tipos de contratos de reuso propostos no capítulo anterior. A construção das classes da *arquitetura de integração* é baseada na lista de funcionalidades especificadas para cada tipo de contrato de reuso. Foram utilizados alguns padrões de projeto ("design patterns") conhecidos da literatura [GAM94].

O capítulo está organizado da seguinte forma: a seção 3.1 trata do tipo de contrato uso, apresentando as funcionalidades definidas para sua implementação, juntamente com uma breve explicação dos padrões de projeto Decorator e Facade. Após, será apresentada a estrutura de uma classe *roteadora* como padrão para implementação do tipo de contrato uso. A seção 3.2 apresenta as funcionalidades definidas para a implementação do tipo de contrato implementação. O padrão de projeto Adapter é explicado para fundamentar o padrão de implementação da classe *implementadora*. Por fim, na seção 3.3, é apresentado o modelo de classes do estudo de caso em questão (sistema de agência de viagens) com as classes *roteadoras* e *implementadoras*.

Nas especificações de cada funcionalidade, é apresentado o código correspondente escrito em Java [SUN99].

3.1 Padrão de projeto para o tipo de contrato uso

A estrutura de implementação do tipo de contrato uso pode ser considerada como um padrão de projeto. O padrão de projeto segue as seguintes funcionalidades, além das características apresentadas na seção 2.3.1: (i) instanciar e destruir os objetos de componentes associados; (ii) possuir referência para os objetos dos componentes relacionados; (iii) invocar os métodos de componentes; e (iv) atualizar as referências exigidas pela aplicação. Os exemplos citados a seguir referem-se ao modelo apresentado na figura 3.1.

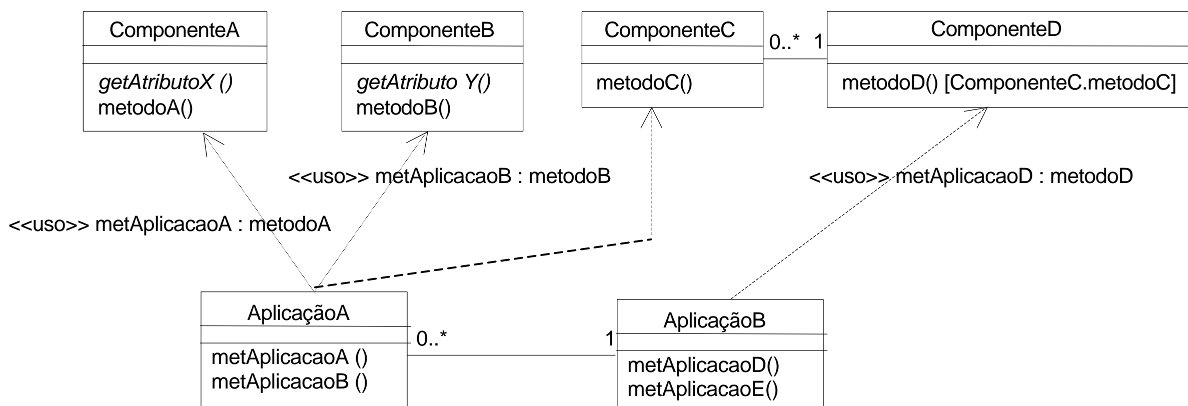

FIGURA 3.1 - Modelo de objetos usando tipo de contrato uso

Funcionalidade1: instanciar e destruir os objetos de componentes associados

Todo objeto da aplicação que é instanciado deve providenciar a instanciação de todos os componentes associados a ele através da relação uso. Quando o objeto da aplicação for destruído, todos os objetos de componentes associados devem ser igualmente destruídos. Pode-se dizer que o objeto da aplicação gerencia a "vida" dos objetos de componentes associados, determinando sua instanciação e sua destruição.

No exemplo em questão, observa-se que a classe de aplicação AplicaçãoA está se relacionando com as classes ComponenteA, ComponenteB e ComponenteC através do tipo de contrato uso. Isto implica em dizer que, toda vez que se instanciar um objeto da classe da aplicação AplicaçãoA, deve-se instanciar os objetos correspondentes a partir das classes dos componentes relacionados.

Funcionalidade 2: possuir referência para os objetos dos componentes relacionados

O objeto da aplicação deve possuir uma referência para cada objeto instanciado nos componentes para que tenha condições de invocar os métodos relacionados (especificação 3). Por exemplo, a classe de aplicação AplicaçãoA deve ter referências para as classes de componentes ComponenteA, ComponenteB e ComponenteC.

Funcionalidade 3: invocar os métodos de componentes

O método da aplicação indicado na relação de uso deve ser codificado para que referencie o método correspondente no objeto de componente. Por exemplo, o corpo do método AplicaçãoA.metAplicaçãoA deve somente referenciar o método ComponenteA.metodoA.

Funcionalidade 4: atualizar as referências exigidas pela aplicação

Quando um método do componente referenciar um método de outro componente, logo uma referência deve ser instanciada entre as classes. Caso a classe referenciada não possua uma classe de aplicação associada, deve-se providenciar uma classe da aplicação que se relacione a esta classe somente para instanciar e destruir objetos dela.

Por exemplo, um dos métodos da aplicação (AplicaçãoB.metAplicaçãoD) referencia um método de componente (ComponenteD.metodoD). O método referenciado, por sua vez, depende do método ComponenteC.metodoC. A dependência do método ComponenteD.metodoD é por um método que está em outra classe, portanto, necessita que a classe ComponenteC seja instanciada e tenha relação com alguma classe da aplicação. Sendo assim, o desenvolvedor da aplicação indica que a classe de aplicação AplicaçãoA faz referência para a classe ComponenteC somente para atender a dependência de métodos definida acima.

3.1.1 Padrão de Projeto Decorator

Um problema existente na implementação do tipo de contrato uso que pode ser resolvido, parcialmente, pelo padrão de projeto Decorator é o seguinte: existe a necessidade de estender algumas funcionalidades de um componente para uma classe de aplicação, de modo que esta extensão não se propague para outras classes de aplicação que estejam associadas com tal componente.

Originalmente, o padrão de projeto Decorator trata de estender funcionalidades de um componente para uma classe especializada sem que outras classes especializadas deste componente sejam modificadas. A idéia deste padrão de projeto é criar uma classe intermediária (Decorator) que possua os métodos desejados para extensão referenciando os métodos correspondentes da classe do componente. Desta forma, os métodos que referenciam métodos do componente estão em uma classe separada da classe da aplicação.

Por exemplo, considera-se um modelo de classes que represente um visualizador de textos. Há uma classe VisãoTexto que possui propriedades comuns para a exibição de um texto. Outras características de visualização são proporcionadas, tais como, barra de rolagem e borda de texto, entre outras. As características adicionais não são atendidas pela classe VisãoTexto, mas considere, então, que estas características devam ser agregadas a classe VisãoTexto. Para isto, o padrão Decorator apresenta a seguinte solução:

Cria-se uma classe abstrata que possui os métodos abstratos e comuns do visualizador de textos (ComponenteVisual). Desta classe, herda-se a classe que implementa os métodos abstratos (VisãoTexto). Do outro lado, especializa-se uma classe (Decorator) que se responsabiliza pela agregação de novas funcionalidades à classe comum. A partir da classe intermediária Decorator, outras classes podem ser especializadas conforme as características adicionais que sejam incorporadas ao visualizador de textos. A figura 3.1 mostra duas classes especializadas de Decorator: DecoratorRolagem e DecoratorBorda. O método de visualização destas classes incrementa ("override") o método de visualização do texto.

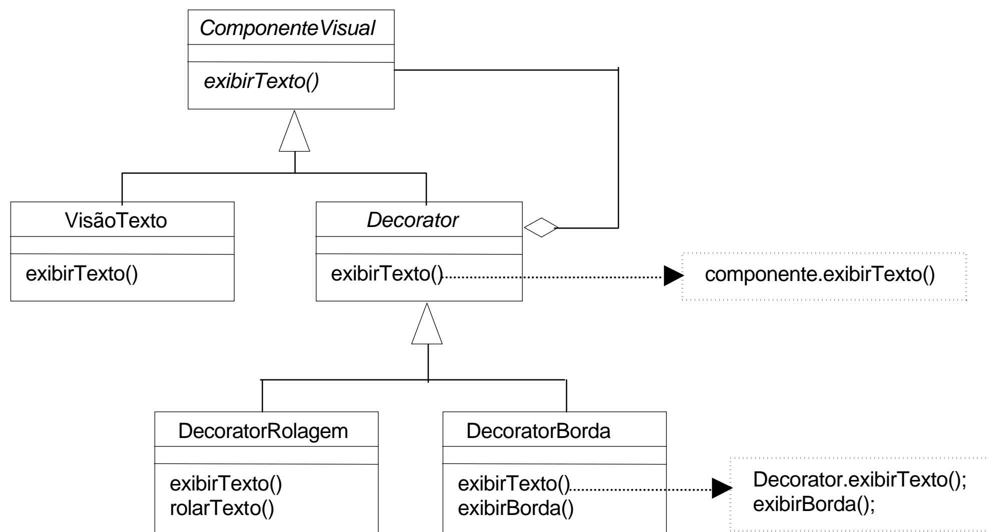

FIGURA 3.1 - Modelo de classes de um visualizador de textos

A figura 3.2 mostra o modo como compor um objeto de VisãoTexto com objetos de DecoratorRolagem e DecoratorBorda.

FIGURA 3.2 - Modelo de classes de um visualizador de textos

A figura 3.3 mostra a estrutura do padrão Decorator. A classe Component é uma super-classe da qual herdam as classes ConcreteComponent que contem as funcionalidades comuns e Decorator. A classe Decorator possui um método comum que referencia o mesmo método de sua super-classe (operation). As classe que implementam as funcionalidades adicionais são classes especializadas da classe Decorator e possuem o método comum (operation) sendo incrementado ("override") com métodos que implementam as funcionalidades adicionais (addedBehavior).

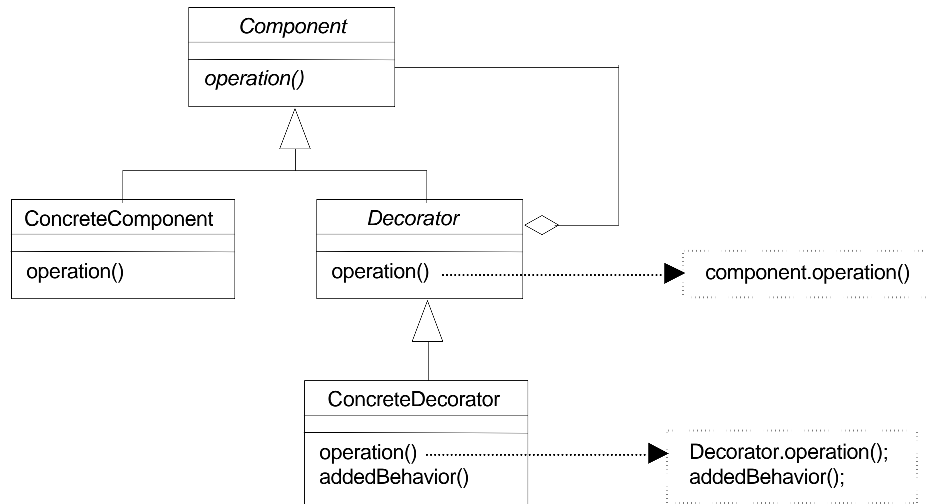

FIGURA 3.3 - Estrutura do padrão de projeto **Decorator**

A idéia aproveitada do padrão de projeto Decorator é a de isolar os métodos que referenciam métodos de componentes em uma estrutura específica. O padrão de projeto Decorator tem outras características que não interessam na solução do problema em questão, portanto, não serão aqui descritas.

3.1.2 Padrão de projeto Facade

Outro problema existente na implementação do tipo de contrato uso e que pode ser resolvido pelo padrão de projeto Facade é o seguinte. Uma classe de aplicação pode associar-se com vários componentes e portanto, a classe de aplicação deve possuir uma referência para cada componente associado e, conter os métodos que estão indicados no tipo de contrato uso.

O padrão de projeto Facade trata de várias associações existentes entre classes de aplicação e classes de componentes, proporcionando uma interface unificada entre

aplicação e componentes. O padrão Facade proporciona uma interface de alto nível que facilita o uso de um subsistema[GAM94].

A proposta do padrão de projeto Facade é criar uma classe que exerça o papel de interface entre as classes da aplicação e os componentes. Para isto, as classes da aplicação devem associar-se à classe Facade, onde cada método da aplicação invoca um método desta classe, que possui referências para as classes que formam o componente e é formada por métodos que invocam os correspondentes métodos dos componentes.

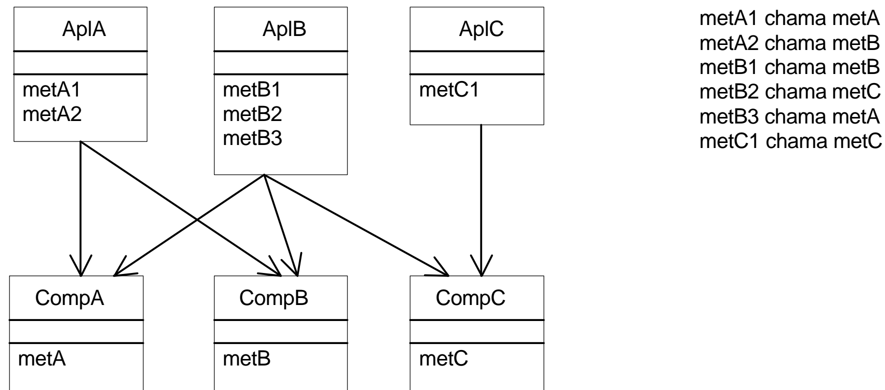

FIGURA 3.4 – Associação entre componentes e aplicação

Usando um exemplo da figura 3.4, observa-se que a classe AplB tem 3 referências, uma para cada classe de componente necessária para a invocação de seus métodos (metB1, metB2 e metB3). Com a classe Facade intermediando as classes de aplicação e as de componente (figura 3.5), a classe AplB somente necessita de uma referência para a classe Facade e esta, por sua vez, faz referencia a cada classe interna do componente (CompA, CompB e CompC). Os métodos da classe Facade possuem uma chamada para cada método homônimo de uma classe do componente. Exemplificando, Facade.metB chama CompB.metB .

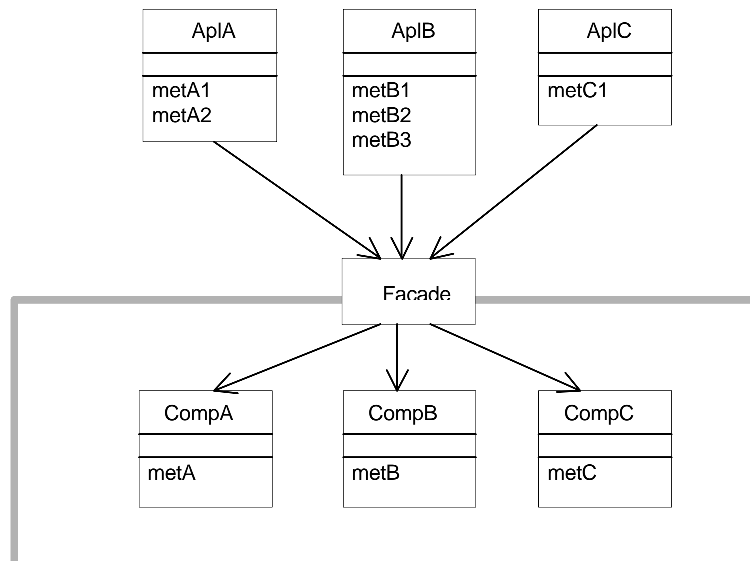

FIGURA 3.5 - Estrutura do padrão de projeto **Facade**

O padrão de projeto Facade contribui com a função de distribuir as chamadas de métodos a várias classes distintas de componentes a partir de uma só classe.

3.1.3 Classe Roteadora: proposta de implementação do tipo de contrato uso

A combinação dos padrões apresentados leva a um novo padrão destinado a implementação do tipo de contrato uso. Este padrão baseia-se em uma classe chamada classe *roteadora*. A classe roteadora separa as funcionalidades descritas nesta seção das demais funcionalidades próprias de uma classe de aplicação.

A classe *roteadora* é uma classe abstrata construída para cada classe da aplicação que possua um contrato de reuso com classes de componentes. Uma classe *roteadora* também pode ser chamada de "wrapper". "Wrapper" é um tipo de classe que centraliza a distribuição de métodos referenciados a vários componentes distintos [ALL98, HOL93] e serve para adaptar propriedades de um componente modificando, assim, sua estrutura externa.

A nomenclatura de uma classe *roteadora* tem a seguinte estrutura: o nome da classe da aplicação acrescida do sufixo Rot. Por exemplo, a figura 3.6 apresenta a classe *roteadora* AplicaçãoRot para atender as referências da classe de aplicação Aplicação. A classe de aplicação possui métodos referenciados para os componentes ComponenteA, ComponenteB e ComponenteC.

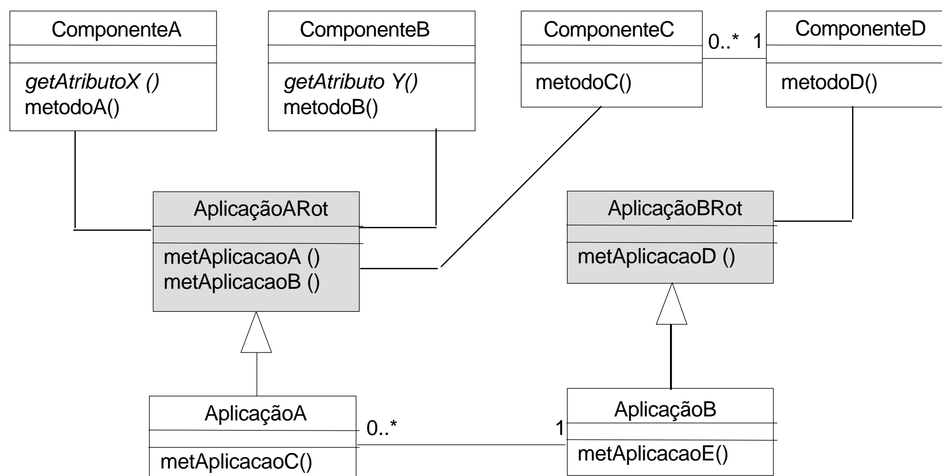

FIGURA 3.6 - Modelo de objetos usando classes roteadoras

A seguir, são apresentadas, de forma resumida e estruturada, as especificações de uma classe *roteadora*. Para cada funcionalidade, é referenciado o padrão de projeto que originou a solução, além de uma especificação resumida, um exemplo e um código aplicado ao exemplo.

TABELA 3.1 – Funcionalidades de uma classe *roteadora*

Funcionalidade	*instanciar objetos de componentes associados*
Origem	padrão de projeto Decorator
Especificação	quando um objeto da aplicação é instanciado, deve-se também instanciar objetos para as classes de componentes associadas
Exemplo	a classe Aplicacao tem contrato de reuso com dois componentes: ComponenteA, ComponenteB e ComponenteC
Código	(ver código abaixo)

```
public abstract class AplicacaoRot
...
public AplicacaoRot()
{
 ComponenteA objComponenteA = new ComponenteA();
 ComponenteB objComponenteB = new ComponenteB();
 ComponenteC objComponenteC = new ComponenteC();
}
```

Funcionalidade	*destruir objetos de componentes associados*
Origem	padrão de projeto Decorator
Especificação	quando um objeto da aplicação é destruído, deve-se também destruir os objetos instanciados nas classes de componentes associadas
Exemplo	quando o objeto objAplicacao é destruído, os objetos objComponenteA, objComponenteB e objComponenteC também devem ser destruídos
Código	(ver código abaixo)

```
public abstract class AplicacaoRot
...
public void finalize() throws  Throwable
{
 super.finalize();
 objComponenteA=null;
 objComponenteB=null;
 objComponenteC=null;
}...
```

Funcionalidade	*possuir referência para os objetos dos componentes associados*
Origem	padrão de projeto Decorator e Facade
Especificação	para cada objeto instanciado a partir de um componente associado, o objeto da aplicação deve possuir a devida referência.
Exemplo	o objeto objAplicacao possui referência para os objetos objComponenteA , objComponenteB e objComponenteC

Código	`public abstract class AplicacaoRot`
	`...`
	`{`
	` private ComponenteA objComponenteA;`
	` private ComponenteB objComponenteB;`
	` private ComponenteC objComponenteC;`
	`}`
	`...`

Funcionalidade	*invocar os métodos de componentes*
Origem	padrão de projeto Facade
Especificação	para cada método indicado pelo tipo de contrato uso
Exemplo	o método metAplicacaoA referencia o método metodoA

Código	`public abstract class AplicacaoRot`
	`...`
	`public void metAplicacaoA()`
	`{`
	` return objComponenteA.metodoA();`
	`}`
	`...`

Funcionalidade	*atualizar as referências exigidas pelas classes da aplicação*
Origem	solução desenvolvida neste trabalho
Especificação	quando um objeto da aplicação é instanciado, deve-se instanciar objetos para as classes de componentes associadas ao objeto da aplicação
Exemplo	a classe de aplicação AplicacaoA possui referência para o componente ComponenteC somente para atender a dependência explícita que o componente ComponenteD possui do ComponenteC, na execução do método métodoD
Código	`conforme o modo de implementar associações do programador`

As funcionalidades apresentadas mostram a forma como uma classe *roteadora* é construída. Deste modo, pode-se considerar que a construção de uma classe *roteadora* pode ser feita automaticamente. Para isto, um software deve possuir os *contratos de reuso* definidos entre componentes e classes da aplicação.

3.1.4 Aplicação do tipo de contrato uso em um estudo de caso

O estudo de caso em questão (agência de viagens) apresenta uma lista de relações de uso entre componentes e aplicação (seção 2.1).

A figura 3.7 mostra a implementação da relação de uso das classes de aplicação RoteiroPacote e Hotel. Para estas classes, são geradas as respectivas classes *roteadoras*

RoteiroPacoteRot e HotelRot. Os métodos definidos na relação de uso estão nas classes *roteadoras*, separando-se da classe da aplicação.

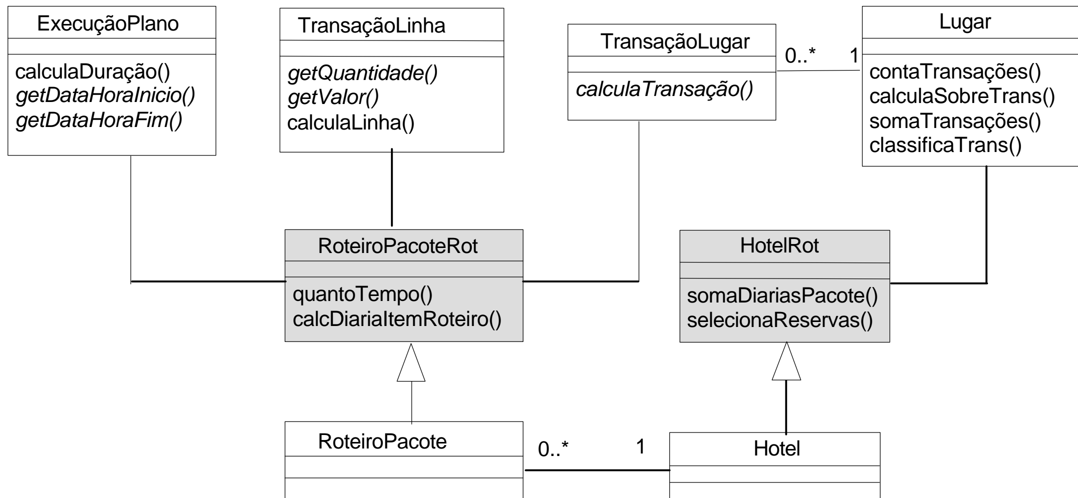

FIGURA 3.7 – Aplicação de uma classe *roteadora*

O problema apresentado na funcionalidade 4 (*atualizar as referências exigidas pelas classes da aplicação*) onde uma classe de aplicação refere-se a um componente para atender uma outra associação, pode ser observado no exemplo em questão.

O método Lugar.somaDiariasPacote usa a implementação do método de componente Lugar.somaTransações. Este método de componente, por sua vez, depende do método TransaçãoLugar.calculaTransação. Portanto, a classe TransaçãoLugar deve ser instanciada para atender a execução do método referenciado. Na relação de uso, a classe de componente TransaçãoLugar não possui nenhuma referência aos seus métodos, logo não deve receber referência de uma classe *roteadora*. Mas, como a classe Lugar tem um método referenciado por um método de aplicação e este método referenciando depende de um método de TransaçãoLugar logo, deve ser referenciado por uma classe de aplicação correspondente. No caso, a classe de aplicação correspondente é a classe RoteiroPacote.

3.2 Padrão de projeto para o tipo de contrato implementação

A implementação do tipo de contrato implementação também pode ser considerada como um padrão de projeto. A implementação do padrão de projeto segue as seguintes funcionalidades, além das características apresentadas na seção 2.3.2.

Para a implementação do tipo de contrato implementação, são necessárias as seguintes especificações: (i) possuir referência para os objetos relacionados e (ii) métodos abstratos referenciam os métodos concretos relacionados para implementação. Os exemplos descritos a seguir referem-se ao modelo apresentado na figura 3.8.

Especificação 1: possuir referência para os objetos relacionados

Quando um componente possui relação de implementação, o objeto instanciado deste componente deve referenciar os objetos onde se encontram os métodos concretos relacionados.

Por exemplo, o objeto de ComponenteA possui uma relação de implementação com o objeto de AplicaçãoA, portanto o objeto do componente possui uma referência para tal objeto da aplicação. No outro exemplo, o objeto de ComponenteD possui uma relação de implementação com o objeto de outro componente, ComponenteF. Logo o objeto de ComponenteD possui uma referência ao objeto de ComponenteF.

Especificação 2: métodos abstratos referenciam os métodos concretos relacionados

Cada método abstrato existente na relação de implementação deve ser especializado. O código desta especialização deve conter uma referência ao método concreto definido na relação.

Por exemplo, o método ComponenteA.getAtributoX quando especializado, deve possuir uma referência ao método AplicaçãoA.getAtributoX1. No outro exemplo, o método ComponenteD.metodoD possui uma referência para o método ComponenteF.metodoF.

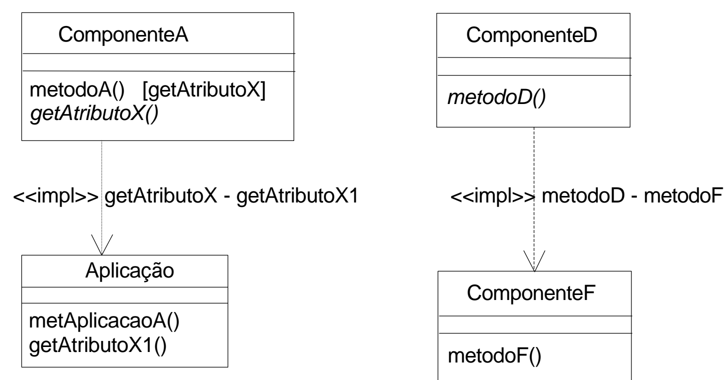

FIGURA 3.8 – Exemplos de relação de **implementação**

A implementação do tipo de contrato de implementação baseou-se em um padrão de projeto definido por Gamma [GAM94]: Adapter. A seguir, descreve-se a estrutura deste padrão de projeto e suas contribuições para o problema estudado neste trabalho.

3.2.1 Padrão de Projeto Adapter

Um problema encontrado na implementação do tipo de contrato implementação é o seguinte. Um método abstrato de um componente deve ser implementado para que satisfaça um método "template" de componente. A implementação deste método abstrato pode permitir que o método concreto não tenha o mesmo nome do método abstrato e o método concreto pode estar em uma classe de aplicação ou em outro componente. Para resolver este problema, o padrão de projeto Adapter apresenta uma solução.

O padrão de projeto Adapter converte a interface de um componente em outras interfaces de cliente conforme sua necessidade de adaptação. O padrão Adapter permite que componentes sejam especializados para que métodos abstratos sejam implementados em outras classes. Esta característica atende o problema encontrado na implementação do tipo de contrato implementação.

O padrão de projeto Adapter é usado em casos onde existem uma classe abstrata (Target) e a sua correspondente classe de interface (Adapter), sendo que essas duas não possuem a mesma interface de outra classe (Adaptee). Considera-se que na classe concreta (Adaptee) esteja implementado um método de interface diferente da classe abstrata (Target). Assim, a classe Adapter serve de intermediação entre as duas classes, fazendo o papel de "adaptador" do método abstrato. A figura 3.9 apresenta a estrutura do padrão de projeto Adapter.

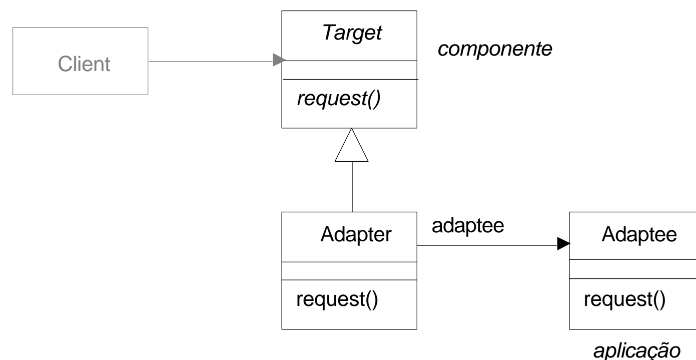

FIGURA 3.9 – Estrutura do padrão de projeto **Adapter**

A contribuição do padrão de projeto Adapter para o tipo de contrato de implementação está no fato de que a classe Target (componente) não pode sofrer modificações para atender as necessidades de uma aplicação Client. Sendo assim, cria-se uma classe intermediária (Adapter) que possui uma referência para a implementação específica necessária. Esta implementação específica está na classe Adaptee (aplicação ou outro componente).

3.2.2 Classe Implementadora: proposta de implementação do tipo de contrato implementação

A classe *implementadora* é apresentada como uma proposta de implementação do tipo de contrato implementação. A classe *implementadora* possui a mesma funcionalidade da

classe Adapter, pois especializa um determinado componente para que métodos abstratos sejam implementados. O método concreto da classe *implementadora* que está implementando um método abstrato do componente apenas possui uma referência para um outro método concreto que possui a implementação propriamente dita.

A geração de classe *implementadora* segue a relação de implementação. Para cada associação de componente com classe de aplicação através da relação de implementação, gera-se uma classe *implementadora.* O nome de uma classe *Implementadora* é composto pelo sufixo I (de implementadora) acrescido da conjunção dos nomes da classe do componente e da classe da aplicação.

A figura 3.10 mostra as classes *implementadoras* para as relações de implementação especificadas na figura 3.8.

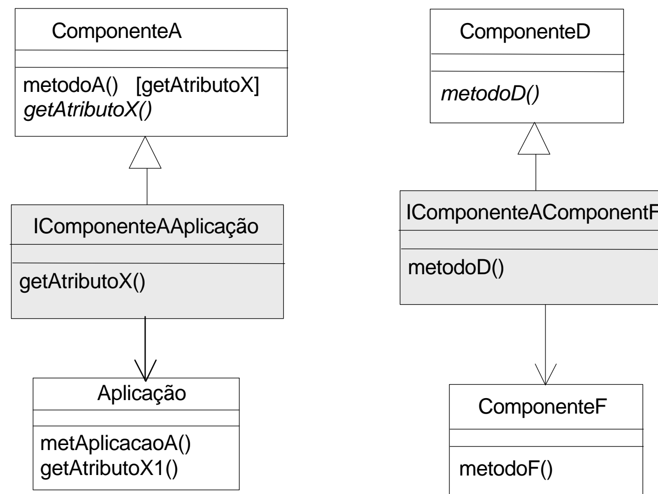

FIGURA 3.10 – Estrutura do padrão de projeto **Adapter**

A seguir, estão especificadas as funções definidas para a classe Implementadora. Os exemplos referenciam-se à figura 3.10.

TABELA 3.2 – Funcionalidades de uma classe *implementadora*

Funcionalidade	*possuir referência para o objeto que possui o(s) método(s) concreto(s)*
Origem	padrão de projeto Adapter
Especificação	cada classe implementadora possui uma referência para o objeto do método concreto referenciado
Exemplo	ComponenteA.getAtributoX() sendo implementado por Aplicacao.getAtributoX1(). No código abaixo, considera-se que a classe Aplicacao tem uma classe roteadora

Código	`public class IcomponenteAAplicacao extends ComponenteA`
	`{`
	` private AplicacaoRot aplicacao;`
	`}`
	`...`

Funcionalidade	*invocar os métodos concretos*
Origem	padrão de projeto Adapter
Especificação	para cada método indicado pelo tipo de contrato uso, é necessário ter uma invocação para o método concreto
Exemplo	ComponenteA.getAtributoX() invocando o método Aplicacao.getAtributoX1()
Código	`public class IcomponenteAAplicacao ... public void getAtributoX() { return aplicacao.getAtributoX1(); } ...`

```
public class IcomponenteAAplicacao
...
public void getAtributoX()
{
 return aplicacao.getAtributoX1();
}
...
```

3.2.3 Aplicação do tipo de contrato implementação em um estudo de caso

A aplicação do tipo de contrato implementação é feita no modelo de classes de uma agência de viagens (seção 2.1). O desenvolvedor da aplicação define *contratos de reuso* para a implementação de alguns métodos abstratos de componentes.

A figura 3.11 mostra métodos do componente ExecuçãoPlano sendo implementados pela classe de aplicação RoteiroPacote. Cria-se a classe *implementadora* IexecuçãoPlanoRoteiroPacote para implementar a relação de implementação dos métodos abstratos getDataHoraInicio e getDataHoraFim. Os métodos existentes na classe *implementadora* possuem referência para os métodos getDataSaida e getDataChegada, respectivamente.

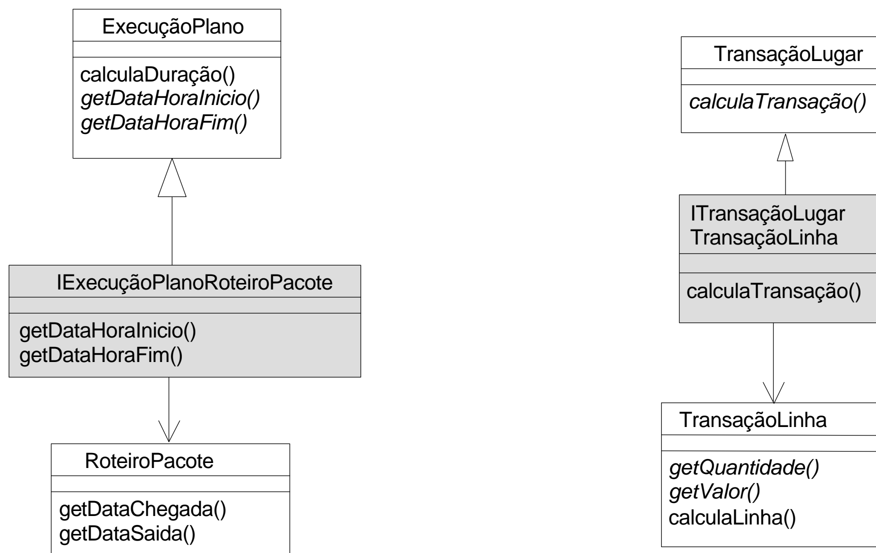

FIGURA 3.11 – Exemplo de relação de **implementação**

A figura 3.11 mostra um método abstrato de componente TransaçãoLugar.calculaTransação(), sendo implementado pelo método TransaçãoLinha.calculaLinha. O desenvolvedor da aplicação determina a relação de implementação a um outro componente que possui um método de mesma especificação.

3.3 Visão geral da arquitetura de integração

A arquitetura de integração resultante do uso dos padrões descritos acima, resulta em um software composto por várias camadas (Figura 3.13). A camada mais interna, a de componentes, é referenciada somente pela camada de integração e esta, por sua vez, é referenciada somente pelas classes da aplicação. Outras classes que venham a compor o software (interface homem/máquina,...) somente farão referência a camada mais externa de aplicação.

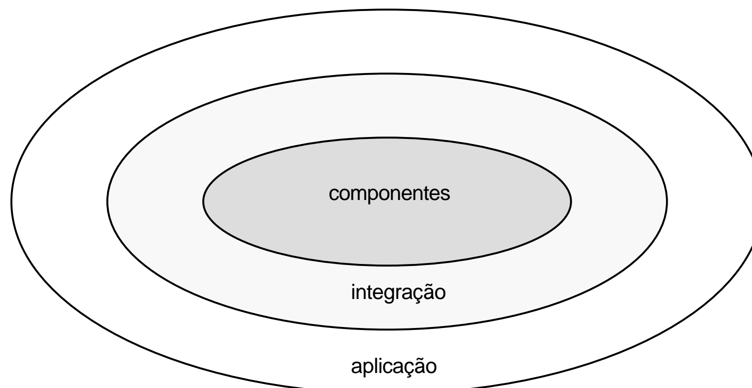

FIGURA 3.13 – Visão geral do modelo usando arquitetura de integração

A figura 3.14 mostra o modelo de classes resultante da implementação dos *contratos de reuso* definidos na relação entre componentes e classes de aplicação. As classes com destaque em cinza são as classes que formam a camada de *integração*. Estas camadas são geradas com base nas relações de uso e de implementação definidas para o problema em questão.

As classes *roteadoras* são classes abstratas geradas para classes da aplicação que possuem relação com componentes. Uma classe *roteadora* possui associação de agregação com as respectivas classes de componentes e possui os métodos da aplicação que referenciam métodos de componentes. Desta forma, as funcionalidades da relação de uso ficam separadas da estrutura original da classe da aplicação.

As classes *implementadoras* são classes especializadas (subclasses) dos componentes que necessitam de implementação de métodos abstratos, portanto, a instanciação de um objeto de componente ocorre da classe *implementadora*. As classes *roteadoras* que referenciam uma classe de componente devem, neste caso, referenciar a classe *implementadora* correspondente. Por exemplo, a classe RoteiroPacoteRot possui referência para as classes *implementadoras* IexecuçãoPlanoRoteiroPacote, ItransaçãoLinhaRoteiroPacote e ItransaçãoLugarTransaçãoLinha, pois estas classes possuem especialização da classe de componente.

O Anexo 1 apresenta o código da classe RoteiroPacoteRot e o código da classe IExecuçãoPlanoRoteiroPacote.

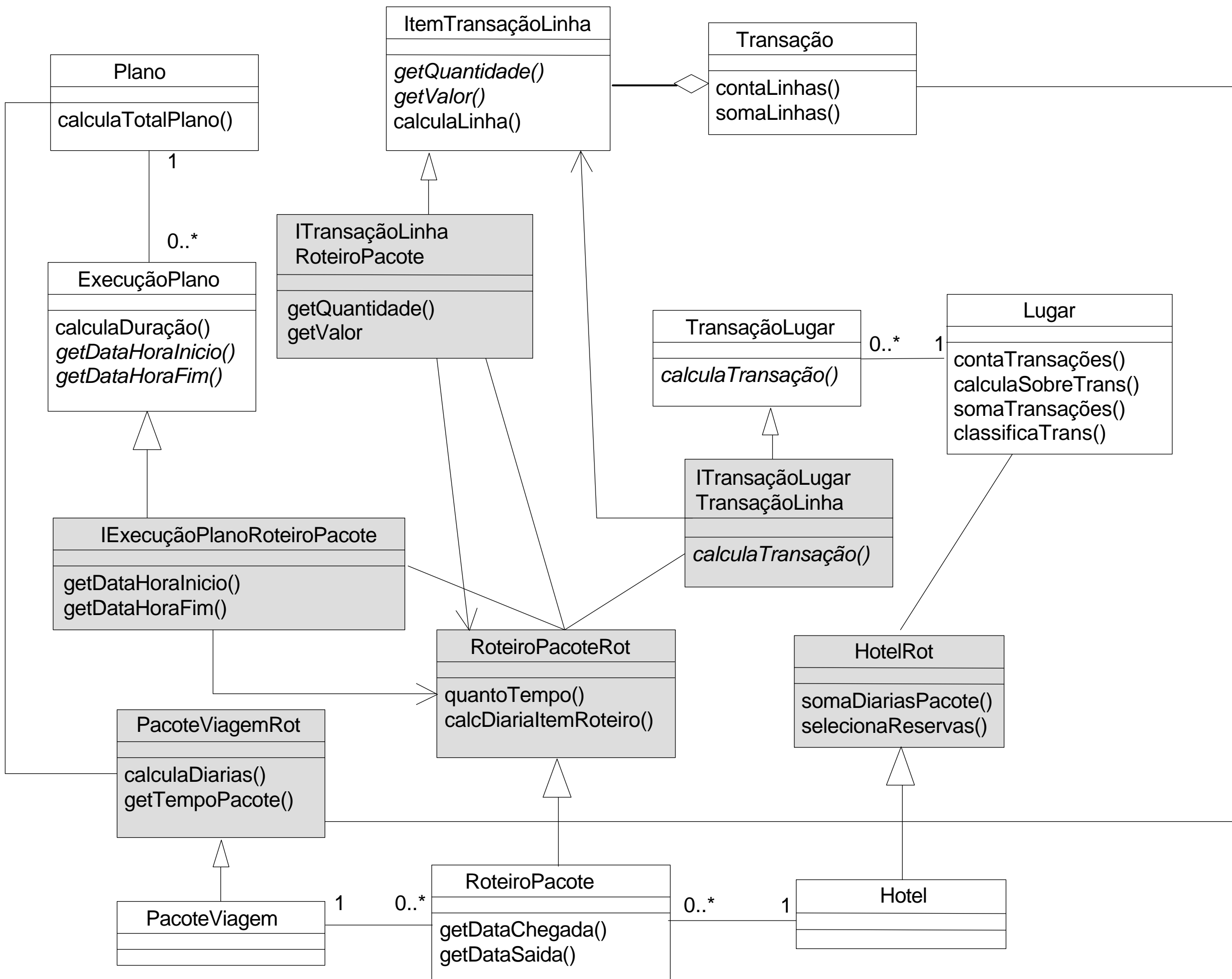

FIGURA 3.14 – Arquitetura de integração entre componentes e aplicação

4 Software Assistente

Este capítulo apresenta o software assistente como uma ferramenta de apoio ao desenvolvedor de aplicações. O objetivo do software assistente é o de dados um conjunto de :

- componentes armazenados em um repositório, e
- classes de aplicação que se deseja usar, e
- contratos de reuso que estabelecem as relações entre as classes da aplicação e as classes de componentes

gerar automaticamente as classes da camada de integração.

O software assistente deve apresentar os componentes existentes em um repositório de componentes ao desenvolvedor para que ele possa construir uma aplicação, através da definição dos contratos de reuso entre componentes e classes de aplicação. A idéia é a de que o usuário somente necessite codificar as classes da aplicação, sendo as demais geradas pelo software assistente.

Durante a descrição do software assistente, existem dois *atores*[2] :

- Desenvolvedor de componentes - ator responsável pelo desenvolvimento do componente.
- Desenvolvedor de aplicação - ator responsável pela construção de aplicações e pela relação entre componentes e classes de aplicação.

O capítulo está organizado da seguinte maneira. A seção 4.1 descreve as especificações dos módulos que formam o software. A seção 4.2 mostra o modelo de dados projetado para armazenar os dados de componentes, da aplicação e da *camada de integração.*

4.1 Especificação do software assistente

O software assistente deve suportar a técnica de reuso apresentada nos capítulos anteriores. Para isto, considera-se que uma técnica baseada em reuso necessita tratar com diferentes temas em relação aos componentes [RUG97]:

- *representação do componente:* trata da especificação da estrutura lógica de um componente, por exemplo, o contexto e o problema solucionado pelo componente;
- *classificação do componente:* trata da organização adequada dos componentes em um repositório para que tenha condições de recuperá-lo mais facilmente;
- *seleção do componente:* trata da recuperação eficiente e flexível dos componentes do repositório;
- *customização do componente:* trata da adaptação dos componentes recuperados para uma nova aplicação e,

---

[2] quando o texto tratar de *usuário do software*, considera-se desenvolvedor de componente ou desenvolvedor de aplicação.

- *composição do componente:* preocupa-se com a integração dos componentes reusáveis para formar uma nova aplicação.

Este capítulo descreve as especificações do software assistente através da notação UML, aplicando *casos de uso reais* ("real use cases") [LAR97]. O software assistente apresenta os seguintes casos de uso:

- Armazenamento de componentes - Os componentes devem estar armazenados de modo que o desenvolvedor da aplicação possa utilizar os métodos dos componentes.

- Seleção de componentes - O software deve dispor de um mecanismo de pesquisa de componentes onde resulte uma lista conforme as necessidades específicas do desenvolvedor de aplicação.

- Construção de aplicação - O software deve auxiliar na definição das relações de uso e de implementação entre componentes e classes de aplicação. As relações definidas devem ser armazenadas para futuros processos

- Geração da camada de integração – O software deve gerar as classes de *integração* conforme as relações de uso e de implementação definidas pelo desenvolvedor da aplicação. As regras de geração estão especificadas nas funcionalidades de cada padrão de projeto descrito no capítulo anterior.

O diagrama de *use cases* mostrado na figura 4.1 descreve o software assistente para uso de componentes.

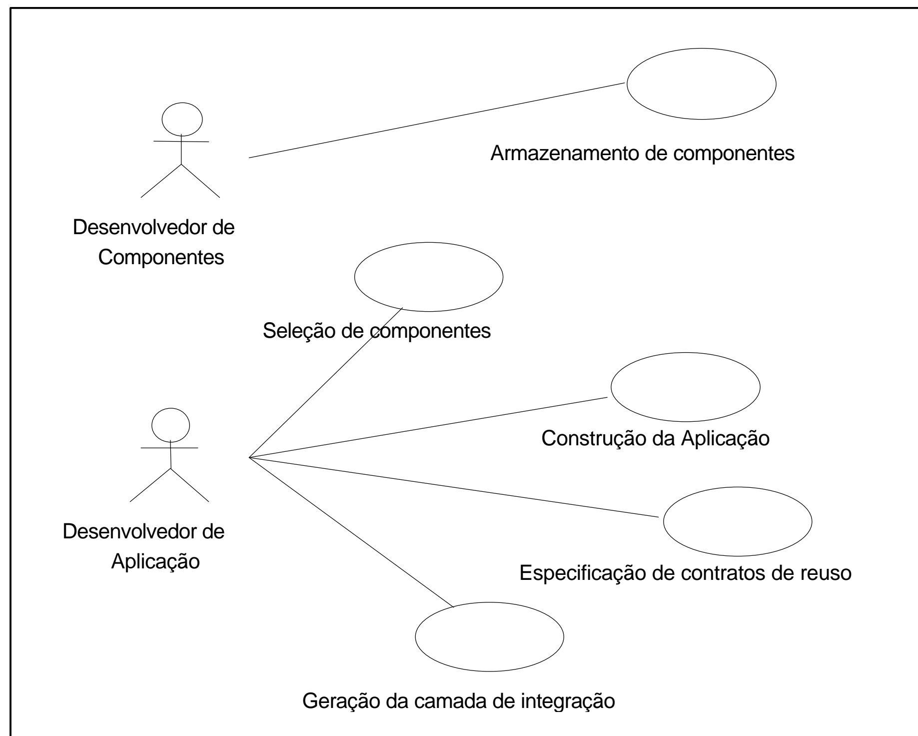

FIGURA 4.1 - Diagrama de Use Cases

4.1.1 Especificação do caso de uso "Armazenamento de Componentes"

A proposta deste caso de uso é armazenar a estrutura de um componente em um repositório específico de componentes.

O armazenamento de componentes, para que haja uma pesquisa sobre os mesmos, requer que suas características também sejam armazenadas [RIC98]. As características de um componente devem descrever as suas funcionalidades de forma que auxilie o desenvolvedor de aplicações no momento da pesquisa e da escolha.

Existem várias formas de documentar um componente. A forma escolhida para este software assistente baseia-se na representação de padrões ("patterns") definida em [COP97, GAM94, JOH92]. Os atributos *contexto, problema, solução* e *exemplos* descrevem as funcionalidades gerais de um componente. Uma lista de *palavras-chave* contém palavras que podem identificar uma funcionalidade de um componente. Além destes atributos, o software assistente armazena um diagrama de classes do componente.

O caso de uso "Armazenamento de Componentes" é dividido em quatro partes: (i) cadastro das características do componente, (ii) estrutura do componente, (iii) especificação de um método do componente e (iv) lista de dependência de métodos.

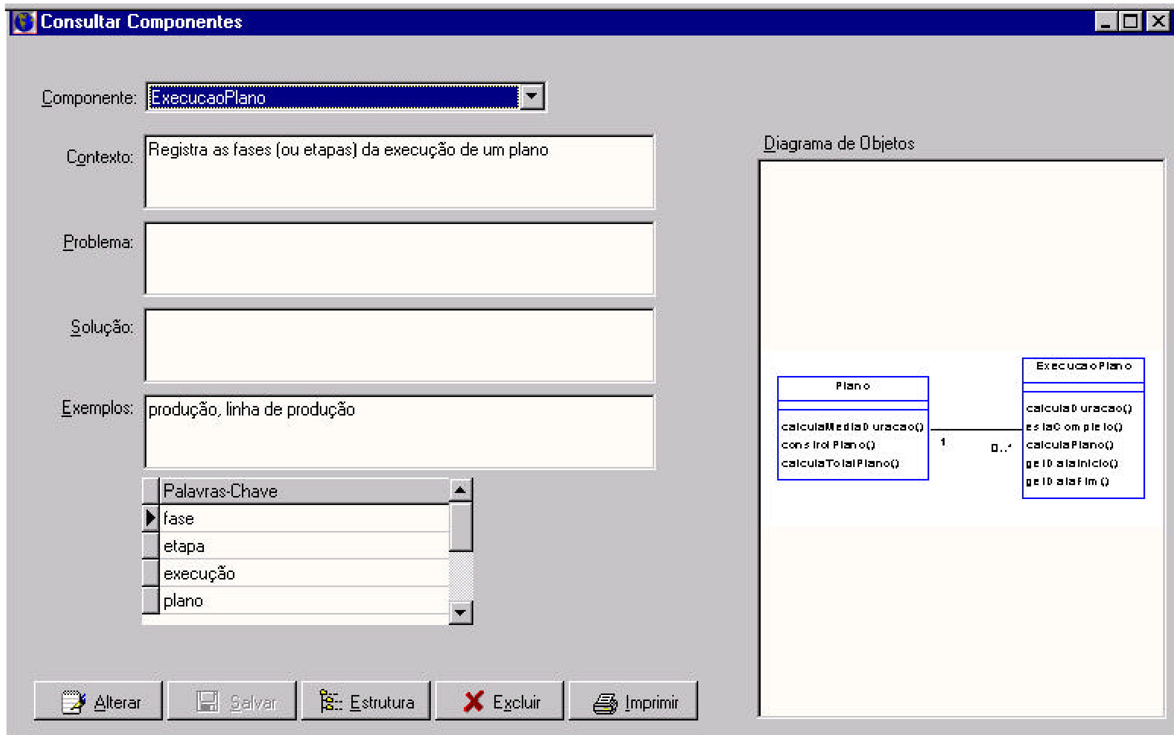

FIGURA 4.2 - Interface: Consultar Componentes

Cadastro de características do componente

A figura 4.2 mostra a interface de cadastro de características do componente.

O desenvolvedor de componentes entra com informações referentes ao componente tais como nome, contexto, problema, solução e exemplos (A). O campo *Nome* é o nome de referência do componente. Usa-se este nome em todos os outros módulos do sistema.

Como recurso para pesquisa de componentes, o software assistente adota a técnica de palavras-chave. As palavras-chave são palavras definidas para identificar o componente. As palavras-chave são utilizadas no caso de uso "Seleção de Componentes". O desenvolvedor de componentes tem uma lista (B) para informar as palavras-chave.

Para completar as características do componente, o desenvolvedor de componentes pode incluir[3] uma figura (C) representando o modelo de classes do componente.

Com a interface de características do componente preenchida, o desenvolvedor de componentes parte para cadastrar a estrutura de um componente (D).

Estrutura do componente

Este módulo do caso de uso trata de manipular a estrutura de um componente. A figura 4.3 mostra a interface da Estrutura do Componente.

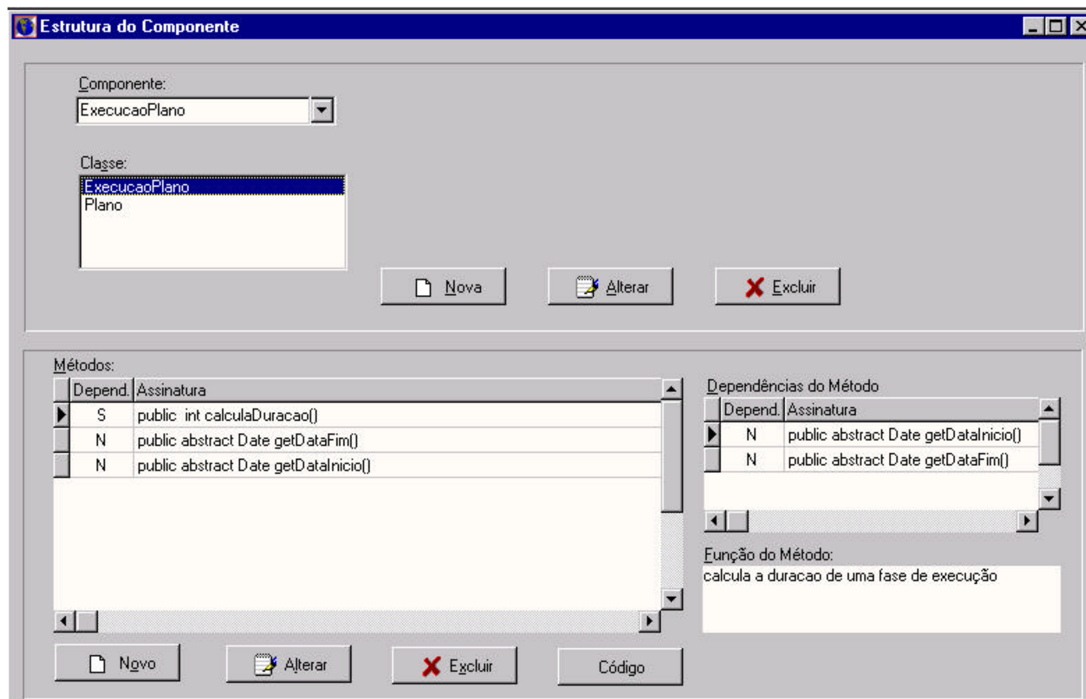

FIGURA 4.3 - Interface: Estrutura de Componentes

---

[3] Na versão atual do software, a figura deve ser importada de alguma ferramenta CASE

O componente tratado neste trabalho não possui atributos, somente métodos. Os métodos de uma classe são listados em (B), onde exibe a assinatura completa do método e um indicador de existência de métodos dependentes ou não. Se caso o método em destaque tiver métodos dependentes, estes são mostrados em (C). O cadastro de métodos é ativado pelos botões Novo e Alterar (E), e está descrito no próximo módulo.

Caso o usuário do software deseja visualizar o código do componente em questão, é necessário ativar o botão Código (D).

Especificação de um método de componente

Este módulo do caso de uso trata de manipular um método de componente. A figura 4.4 mostra a interface da Método de Componente.

Um método de uma classe de componente é descrito da seguinte forma. A *função do método* é um campo textual que expressa as funcionalidades do método.

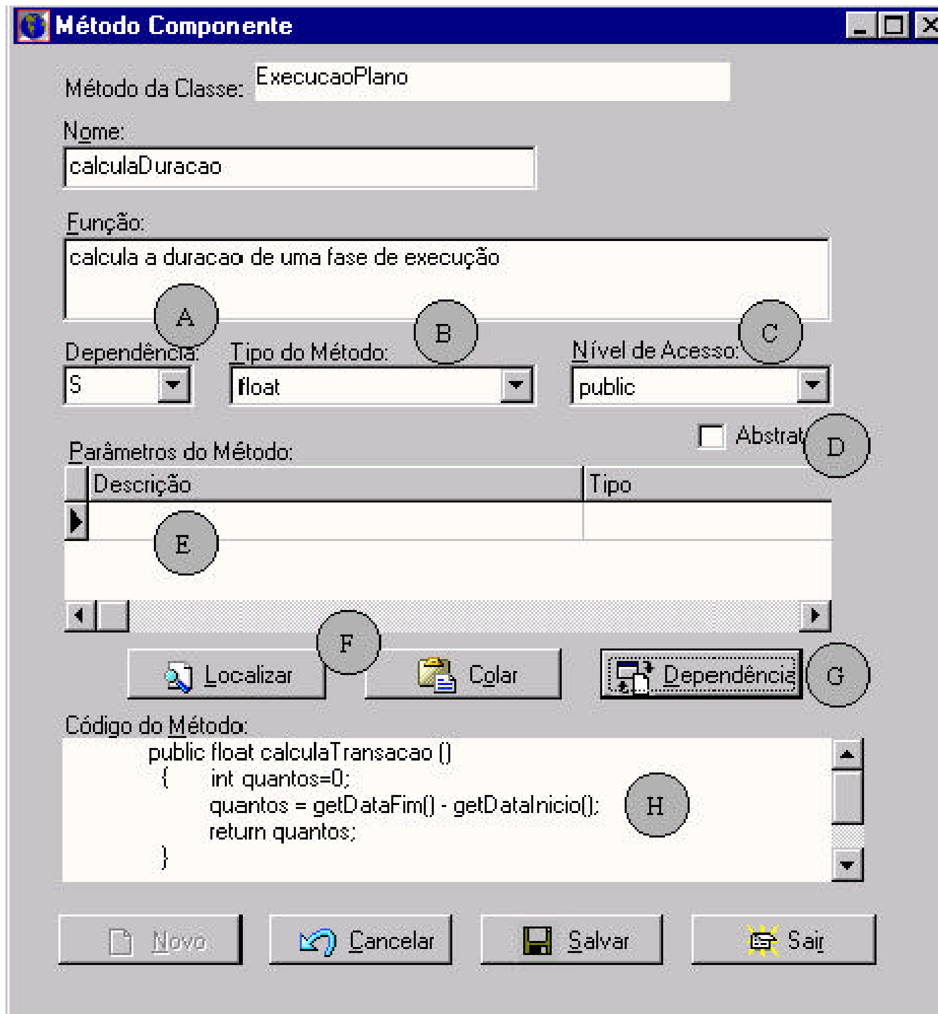

FIGURA 4.4 - Interface: Método de Componente

Um método deve possuir um *tipo de retorno* como resultado de sua execução (B). Este tipo de retorno faz parte de sua assinatura. O desenvolvedor de aplicação observa o tipo de retorno do método para comparar com o retorno proposto no método da aplicação. O *nível de acesso* do método (C) pode assumir os valores de "public", "protected" e "private".

Caso o método seja abstrato, isto deve ser informado (D). Esta informação distingue os métodos que necessitam de implementação dos que não necessitam. Quando um método é abstrato, obrigatoriamente, ele não pode possuir métodos dependentes.

Caso o método tenha *parâmetros* a serem usados em sua assinatura, estes são informados em (E). A lista de parâmetros é observada pelo desenvolvedor da aplicação quando usar um determinado método.

O desenvolvedor de componentes pode incluir o *código do método* em questão. O código do método auxilia o desenvolvedor de componentes a analisar as funcionalidades do método. Ele pode ser copiado de seu código fonte a partir dos botões Localizar e Colar (F).

Um método de componente pode possuir uma *lista de métodos dependentes* (A). A lista de métodos dependentes é apresentada no próximo módulo e a interface desta opção é ativada pelo botão (G).

Lista de dependência de métodos

Este módulo do caso de uso trata da lista de métodos dependentes a um determinado método. A lista de métodos dependentes é importante pois o desenvolvedor da aplicação necessita conhecer quais métodos dependentes, escolhido na relação de uso, devem ser implementados.

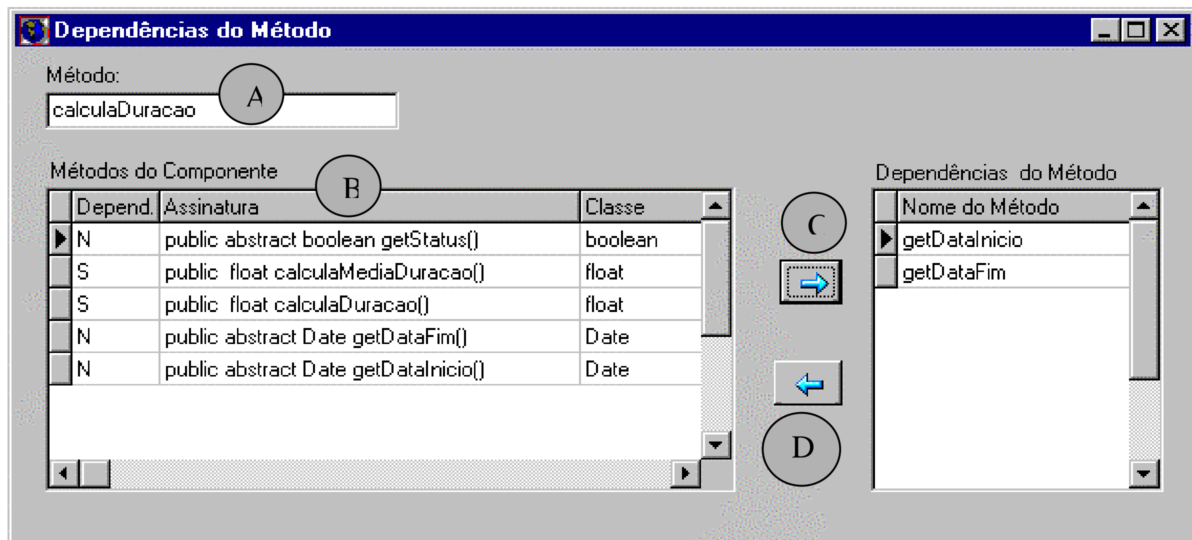

FIGURA 4.5 - Interface: Dependência de métodos

A figura 4.5 mostra a interface da Dependência de métodos. Por exemplo, o desenvolvedor de componentes deve definir os métodos dependentes do método calculaDuração (A) a partir da lista dos métodos do componente (B). Usando o botão (C), o desenvolvedor de componentes indica que os métodos getDataInicio e getDataFim são métodos de que o método calculaDuração depende. Para desfazer uma dependência, use o botão (D).

4.1.2 Especificação do caso de uso "Seleção de Componentes"

A proposta deste caso de uso é selecionar componentes conforme palavra-chave informada pelo usuário do software. A figura 4.6 apresenta a interface de Seleção de componentes.

A seleção de componentes é um dos assuntos mais discutidos na área de reuso de software. Alguns autores atribuem a falta de mecanismos de seleção de componentes como um dos fatores que conduzem a não aplicação de reuso [ISA96, MIL95, WOO88].

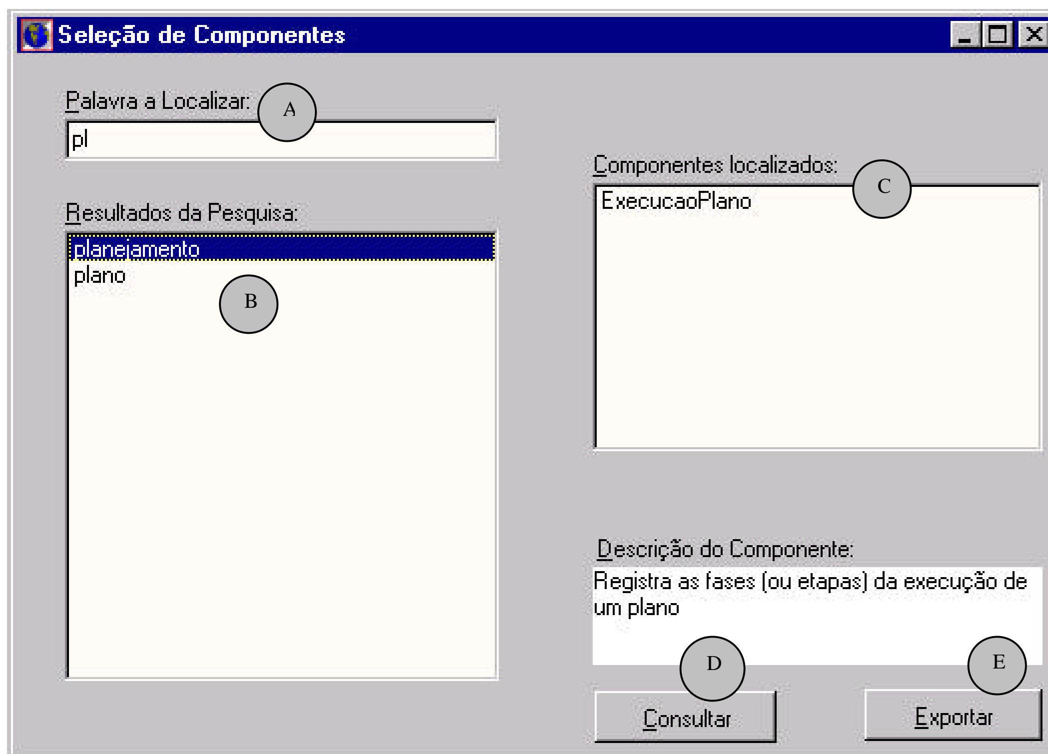

FIGURA 4.6 - Interface: Seleção de Componentes

Este módulo deve providenciar uma interface que permita ao usuário do software entrar com palavras que ele considera possíveis de identificar um componente. Por exemplo, alguns componentes tratam de acompanhar, gerenciar, tratar planejamento de negócio. O usuário informa a expressão "plan" e o software deve responder uma lista de componentes que possuem esta expressão entre suas palavras-chave.

A interface de Seleção de componentes é formada por um campo onde o usuário entra com a expressão desejada (A). Uma lista de palavras-chave surge em (B) conforme o avanço da digitação da expressão desejada. Na lista de componentes localizados (C), exibe-se a lista de componentes que possuem a palavra-chave informada.

Caso o usuário do software deseje consultar a estrutura do componente selecionado, é necessário usar o botão Consultar (D). Para exportar o nome do componente para a interface de origem, use o botão Exportar (E).

4.1.3 Especificação do caso de uso "Construção de uma aplicação"

A proposta deste caso de uso é construir uma aplicação definindo relações de uso e de implementação com componentes pré-existentes. A figura 4.7 mostra a interface de Construção de uma aplicação.

Este caso de uso trata de auxiliar o desenvolvedor da aplicação no que tange a definição das relações entre componentes e a aplicação em construção (A). No primeiro momento, sugere-se que o desenvolvedor da aplicação tenha definido as classes e os métodos que formam sua aplicação. Usando o botão Novo e Alterar (B), o desenvolvedor da aplicação chama a interface de Cadastro de Classes da Aplicação.

FIGURA 4.7 - Interface: Método de classe de aplicação

Para cada classe da aplicação, o desenvolvedor da aplicação deve informar a lista de métodos. A figura 4.8 mostra a interface de cadastro de métodos da aplicação. Esta interface é semelhante ao cadastro de métodos de componente, apenas variando as funcionalidades referente a lista de métodos dependentes, lista que o software assistente não necessita conhecer.

4.1.4 Especificação do caso de uso "Especificação de contratos de reuso"

A proposta deste caso de uso é relacionar os métodos da aplicação com métodos de componentes, usando os conceitos dos *contratos de reuso* uso e implementação. A figura 4.8 mostra a interface de Construção de uma aplicação.

Uma condição para a execução deste caso de uso é que as classes de aplicação já devem possuir seus métodos. Para cada classe da aplicação, o desenvolvedor seleciona os componentes necessários para construir a relação de uso. A figura 4.8 mostra a interface onde uma classe de aplicação relaciona classes de componentes. Os componentes selecionados estão listados em (E) e a classe do componente usado na relação está em (F). (figura 4.9).

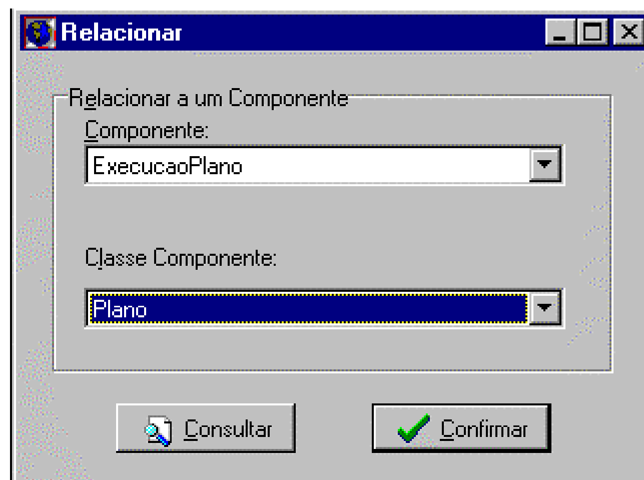

FIGURA 4.8 - Interface: Relacionamento entre classe de aplicação e componente

Para auxiliar o desenvolvedor da aplicação na definição dos métodos da relação de uso, o software assistente exibe a lista de métodos da aplicação (G) e a lista de métodos do componente selecionado (H). Visualizando as duas listas de métodos, o desenvolvedor da aplicação indica o método da aplicação e o método do componente desejado para usar sua implementação. Para confirmar a relação de uso entre os métodos indicados, usa-se o botão (J).

Ao definir uma relação de uso entre dois métodos, da aplicação e do componente, o software assistente deve consistir as seguintes características entre os dois métodos:

- o método do componente não pode ser abstrato,
- o tipo de retorno dos dois métodos devem ser iguais e
- a lista de parâmetros dos dois métodos devem ser iguais.

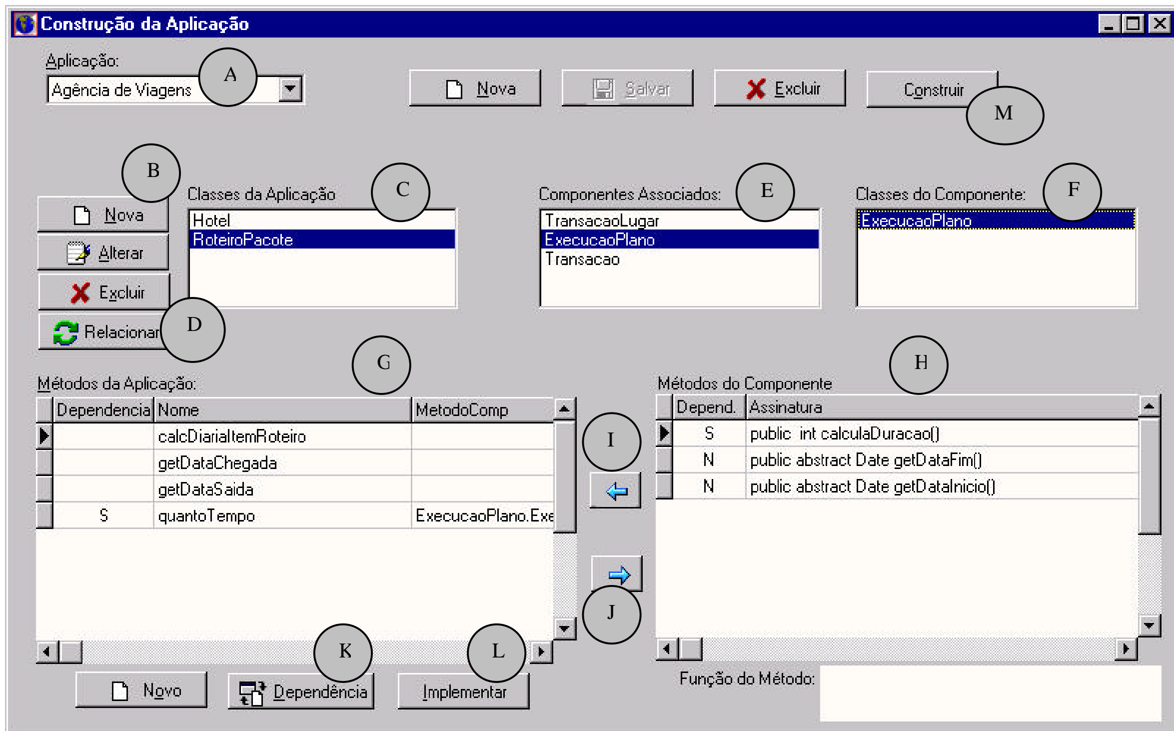

FIGURA 4.9 - Interface: Construção da Aplicação

Caso alguma inconsistência seja detectada, o software assistente deve cancelar a operação e emitir aviso ao desenvolvedor da aplicação.

A definição da relação de implementação entre métodos de componentes e métodos da aplicação acontece quando um método de componente (H) é usado por um método da aplicação e este método possui métodos dependentes (coluna *Depend* em (H)). Neste caso, o desenvolvedor da aplicação seleciona um método com dependência e inicia o processo de definição da implementação do método abstrato, usando inicialmente o botão (L).

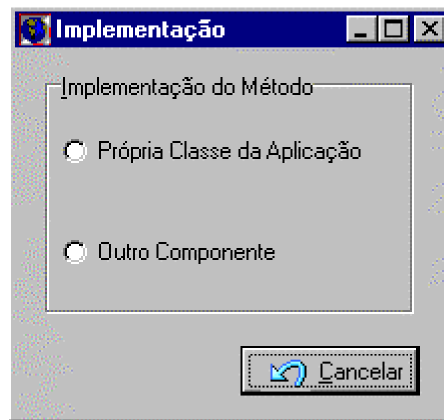

FIGURA 4.10 - Interface: Escolha do tipo de classe para implementação

A figura 4.10 mostra a primeira interface da relação de implementação, onde o desenvolvedor da aplicação informa qual o tipo de classe a ser usada na implementação do método abstrato: *própria classe da aplicação* ou *classe de outro componente*.

Caso a escolha do tipo de classe for a *própria classe da aplicação*, o software assistente mostra a lista de métodos disponíveis na classe da aplicação em questão. Desta lista, o desenvolvedor da aplicação escolhe o método para implementação. O software assistente consiste o tipo de retorno e os parâmetros do método escolhido com as mesmas características do método abstrato do componente.

Caso a escolha do tipo de classe for de *outro componente*, o software assistente mostra uma lista de componentes associados a classe da aplicação em questão e, a partir de um componente escolhido pelo desenvolvedor da aplicação, mostra a lista de classes do componente escolhido. A partir da definição da classe de componente para a implementação, o software exibe a lista de métodos desta classe. Então, o desenvolvedor da aplicação escolhe o método de outro componente para a implementação do método abstrato em questão. O software assistente consiste o tipo de retorno e os parâmetros do método escolhido com as mesmas características do método abstrato do componente.

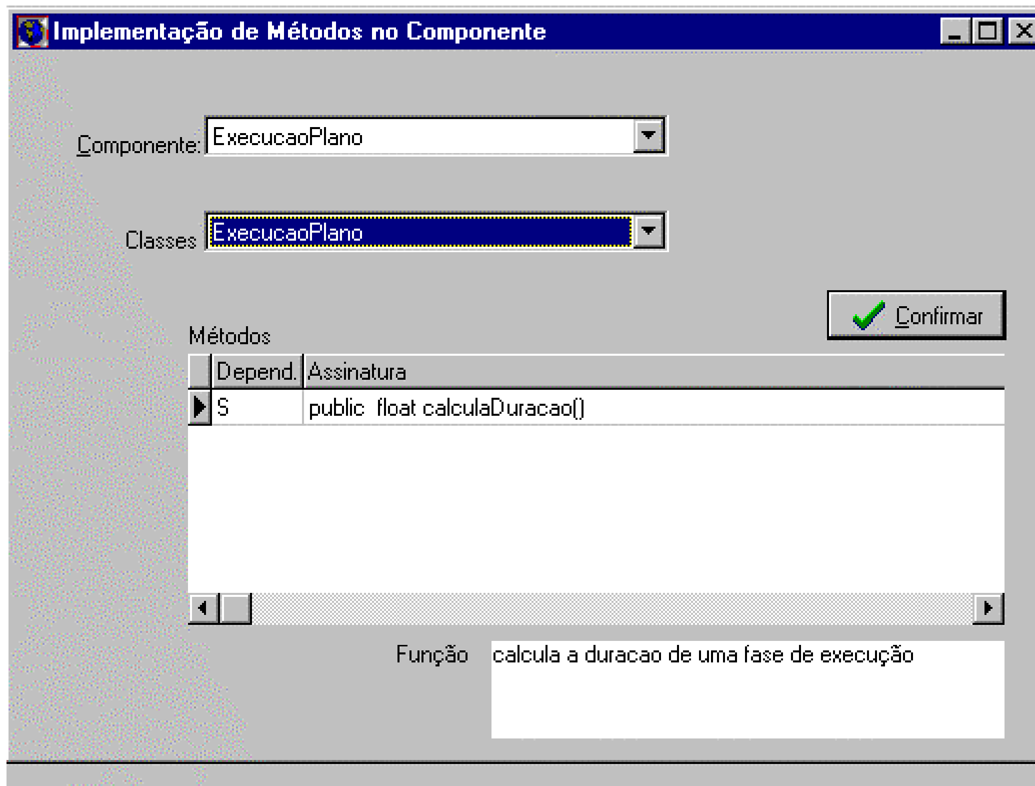

FIGURA 4.11 - Interface: implementação com método de outro componente

4.1.5 Especificação do caso de uso "Geração da camada de integração"

A proposta deste caso de uso é de gerar as classes da *camada de integração* conforme as relações de uso e de implementação estabelecidas pelo desenvolvedor da aplicação. Estas relações estão armazenadas no software assistente. A proposta de geração de classes automaticamente representa uma economia de esforços na fase de codificação. Esta geração automática ocorre pois existem regras pré-definidas de construção de classes [ALT97, BUD96]. A figura 4.9 mostra a interface para a ativação do processo de geração da *camada de integração*.

A geração das classes da *camada de integração* deve ser executada quando o desenvolvedor da aplicação já definiu todas as relações de uso e de implementação necessárias para a aplicação em questão. Quando concluída a definição das relações, usa-se o botão (M) para executar a geração das classes da *camada de integração.*

O processo de geração das classes da *camada de integração* solicita ao desenvolvedor da aplicação, o diretório do sistema operacional onde devem ser geradas as classes da *camada de integração*. A figura 4.12 mostra a interface para escolha do diretório.

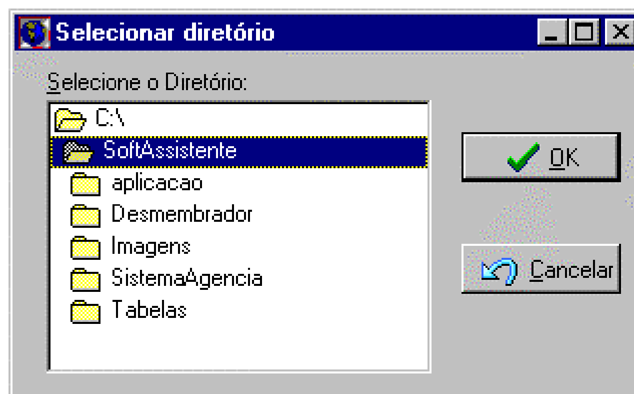

FIGURA 4.12 – Interface: Seleção do diretório das classes da *camada de integração*

A geração de classes da *camada de integração* é dividida em duas partes. Uma é responsável pela geração das classes *roteadoras* e a outra, pela geração das classes *implementadoras.*

A geração das classes *roteadoras* segue os seguintes passos:

- selecionar as classes e os métodos da aplicação que possuem relação de uso,
- selecionar os métodos de componente usados na relação de uso,
- gerar o código de uma classe *roteadora* para cada classe da aplicação que possuir relação de uso,
- gerar um método da classe *roteadora* para cada método da aplicação que possui relação de uso, com uma referência para o método do componente associado.

A geração das classes *implementadoras* segue os seguintes passos:

- selecionar as classes de componente relacionadas na aplicação em questão e que possuem relação de implementação,
- selecionar as classes que implementam os métodos abstratos,
- gerar o código de uma classe *implementadora* para cada associação de classe de componente e sua correspondente classe de implementação (pode ser uma classe da aplicação ou uma classe de outro componente),
- gerar um método da classe *implementadora* para cada método abstrato que possui relação de implementação, tendo uma referência para o método concreto definido na relação.

Ao concluir a geração das classes da *camada de integração*, o software assistente mostra a lista de classes geradas. A figura 4.13 apresenta um exemplo dessa lista de classes geradas.

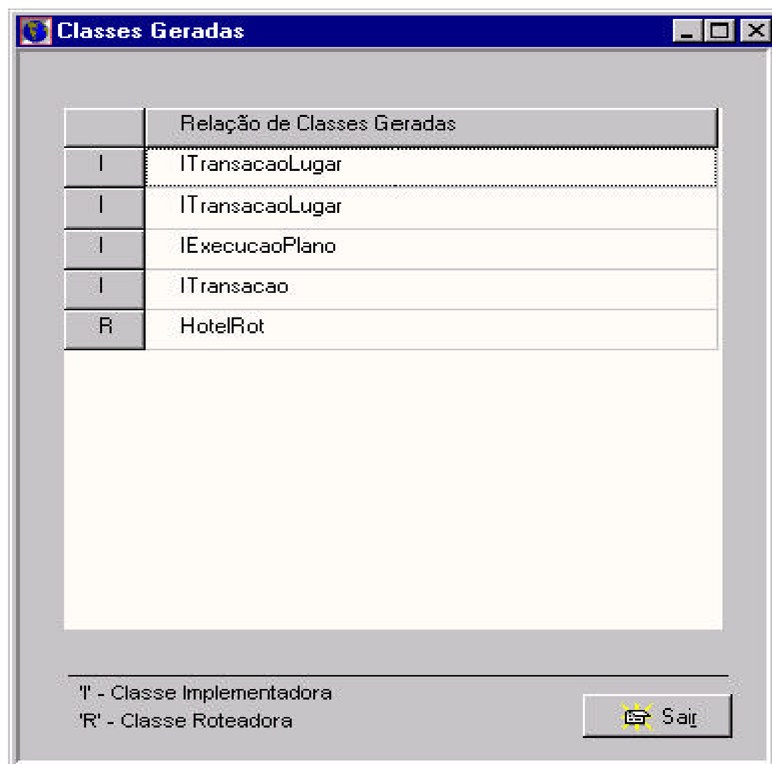

FIGURA 4.13 – Lista de classes gerada da *camada de integração*

4.1.6 Relatórios do Software Assistente

O software assistente oferece dois relatórios para o usuário do software: *relatório de componentes* e *relatório da relação entre componentes e aplicação*.

O relatório de componentes apresenta a estrutura cadastrada dos componentes cadastrados no software assistente. A figura 4.14 mostra um exemplo do relatório de componentes.

```
                    Relatório de Componentes

Componente: ExecuçãoPlano
Contexto:
Problema:
Solução:
Exemplos:
Palavras-chave: fase, etapa, plano, planejamento

Classe: Plano
Métodos:
      Public float calculaTotalPlano()
            depende de ExecuçãoPlano.calculaDuracao()

Classe: ExecuçãoPlano
Métodos:
      Public float calculaDuracao()
            depende de abstract Date getDataInicio()
                         abstract Date getDataFim()
      Public abstract Date getDataInicio()
      Public abstract Date getDataFim()
```

FIGURA 4.14 – Relatório de Componentes

O relatório da relação entre aplicação e componentes mostra as relações de uso e de implementação estabelecidas para uma determinada aplicação. A figura 4.15 apresenta um exemplo do relatório para a aplicação Agência de Viagens.

Relatório da Relação Aplicação - Componentes

Aplicação: Agência de Viagens

Classe: RoteiroPacote

Método	**relação**	Método relacionado
quantoTempo	usa	ExecuçãoPlano.calculaDuracao
calculaDiariaItemRoteiro	usa	TransacaoLinha.calculaLinha
getDataChegada		
getDataFim		

Classe: Hotel

Método	**relação**	Método relacionado
somaDiariasPacote	usa	Lugar.somaTransacoes
selecionaReservas	usa	Lugar.classificaTransacoes

Classe: ExecuçãoPlano

Método	**relação**	Método relacionado
getDataHoraInicio	implementa	RoteiroPacote.getDataSaida
getDataHoraFim	implementa	RoteiroPacote.getDataChegada

FIGURA 4.15 – Relatório de Relação Aplicação e Componentes

4.2 Modelo de dados do software assistente

Esta seção apresenta o modelo de dados do Software Assistente. O modelo de dados trata de armazenar os componentes, as relações de uso e de implementação entre métodos de aplicação e métodos de componentes e alguns dados das classes da aplicação.

O modelo de dados está projetado no modelo relacional. Algumas notações cabem ser destacadas: "PK" indica chave primária ("primary key"), "FK" indica chave estrangeira ("foreign key"), o par de cardinalidades representa a cardinalidade mínima (opcional ou obrigatória) e a cardinalidade máxima ("um" ou "muitos").

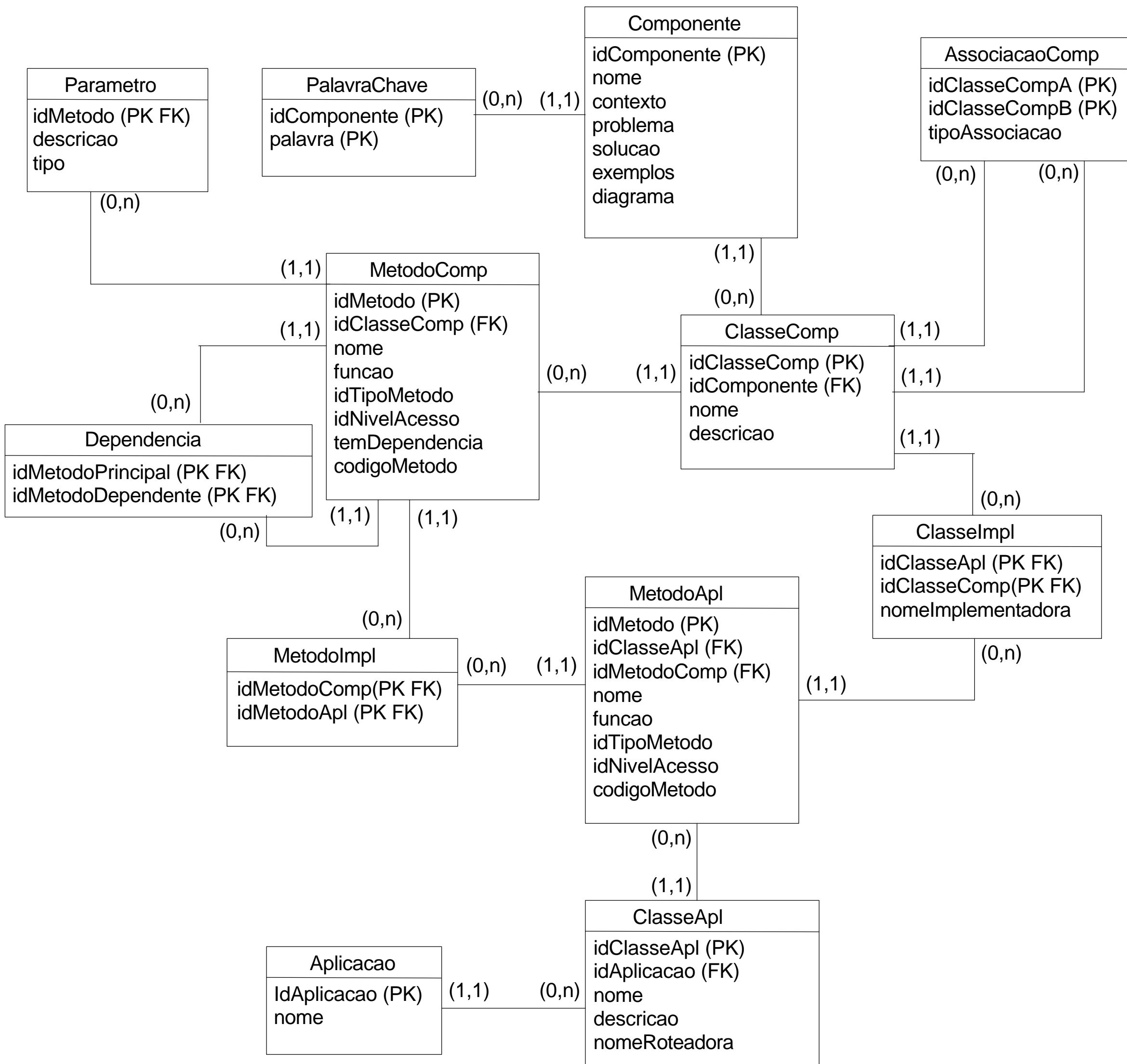

FIGURA 4.16 – Modelo de dados do software assistente

A seguir, é apresentado o dicionário de dados do software assistente.

TABELA 4.1 – Relação das tabelas do software assistente

Tabela	Aplicação	
Descrição	aplicações desenvolvidas usando componentes	
Chave	**Atributo**	**Descrição**
PK	idAplicação	identificador de uma aplicação
	nome	nome de uma aplicação

Tabela	AssociaçãoComp	
Descrição	associações entre classes de componentes ("A" associa "B")	
Chave	**Atributo**	**Descrição**
PK	idClasseCompA	identificador da classe de componente A
PK	idClasseCompB	identificador da classe de componente B

Tabela	ClasseApl	
Descrição	classes existentes em uma aplicação	
Chave	**Atributo**	**Descrição**
PK	idClasseApl	identificador único de uma classe de aplicação
FK	idAplicação	identificador da classe de aplicação correspondente
	nome	nome da classe de aplicação
	descrição	descrição das características de uma classe de aplicação
	nomeRoteadora	nome da classe roteadora correspondente a uma classe de aplicação

Tabela	ClasseComp	
Descrição	classes existentes em um componente	
Chave	**Atributo**	**Descrição**
PK	idClasseComp	identificador único de uma classe de componente
FK	idComponente	identificador da classe de componente correspondente
	nome	nome de uma classe de componente
	descrição	descrição das características de uma classe de componente

Tabela	ClasseImpl	
Descrição	classes *implementadoras*	
Chave	**Atributo**	**Descrição**
PK FK	idClasseComp	identificador de uma classe de componente
PK FK	idClasseApl	identificador de uma classe de aplicação
	nomeImplementadora	nome da classe *implementadora*

Tabela	Componente	
Descrição	componentes pré-existentes	
Chave	**Atributo**	**Descrição**
PK	idComponente	identificador único de um componente
	nome	nome de um componente
	contexto	resumo das funcionalidades de um componente
	problema	descrição dos problemas tratados por um componente
	solução	descrição da solução apresentada em um componente
	exemplos	exemplos de aplicação que podem usar este componente
	diagrama	figura ”bitmap” que representa o modelo de classes do componente

Tabela	Dependência	
Descrição	relação de métodos dependentes	
Chave	**Atributo**	**Descrição**
PK FK	idMetodoPrincipal	identificador do método principal
PK FK	idMetodoRef	identificador do método referenciado pelo método principal

Tabela	MetodoApl	
Descrição	métodos de uma classe de aplicação	
Chave	**Atributo**	**Descrição**
PK	idMetodoApl	identificador único de um método de aplicação
FK	idClasseApl	identificador da classe de que o método pertence
FK	idMetodoComp	identificador do método estabelecido na relação de *uso*
	nome	nome do método
	função	descrição das funcionalidades do método
	tipoMetodo	tipo de retorno do resultado do método
	nivelAcesso	nível de acesso do método {“public”, ”private”, ”protected}
	codigoMetodo	código fonte do método

Tabela	MetodoComp	
Descrição	métodos de uma classe de componente	
Chave	**Atributo**	**Descrição**
PK	idMetodoComp	identificador único de um método de componente
FK	idClasseComp	identificador da classe de que o método pertence
	nome	nome do método
	função	descrição das funcionalidades do método
	tipoMetodo	tipo de retorno do resultado do método

	nivelAcesso	nível de acesso do método {“public”, ”private”, ”protected}
	temDependencia	indicador da existência de métodos dependentes
	codigoMetodo	código fonte do método

Tabela	MetodoImpl	
Descrição	relação de *implementação* entre componente e aplicação	
Chave	**Atributo**	**Descrição**
PK FK	idMetodoComp	identificador do método abstrato do componente
PK FK	idMetodoApl	identificador do método de implementação

Tabela	PalavraChave	
Descrição	lista de palavras-chave de um componente	
Chave	**Atributo**	**Descrição**
PK	idComponente	identificador do componente
PK	palavra	palavra-chave

Tabela	Parâmetro	
Descrição	lista de parâmetros de um método de componente	
Chave	**Atributo**	**Descrição**
PK FK	idMetodoComp	identificador do método do componente
	descricao	descrição do parâmetro
	tipo	tipo de retorno do parâmetro

.

5 Conclusões

A dissertação apresenta uma técnica de reuso composta por :

- uma técnica para especificação de relações entre componentes e classes de aplicação usando uma notação gráfica;
- uma arquitetura de software para implementação das relações definidas; e
- uma ferramenta de suporte à implementação destas relações.

A técnica utiliza *componentes de software* como objetos de reuso. Os componentes usados são desenvolvidos para a camada de *domínio do problema*. Estes componentes devem possibilitar a visualização da estrutura interna, portanto, são usados componentes do tipo "white box". Desta forma, a camada de *domínio do problema* está formada por componentes, por classes da aplicação e por uma camada de *integração*.

As relações entre componentes e classes de aplicação são classificadas em *uso*, *dependência* e *implementação*.

A relação de *uso* é aplicada quando o desenvolvedor da aplicação deseja usar a funcionalidade de um método de componente como implementação de um método de classe da aplicação. Esta relação considera que nem todos os métodos de componente devam ser usados por classes de aplicação e que os nomes dos métodos da aplicação não necessitam ser os mesmos do componente. Além disto, uma classe de aplicação pode se utilizar de vários componentes. A relação de *dependência* especifica os métodos que um outro determinado método depende para sua execução. Entre estes métodos dependentes, pode haver métodos abstratos que requerem implementação. A implementação de métodos abstratos é responsabilidade do desenvolvedor da aplicação e ela pode ser feita por métodos da aplicação ou por métodos de outros componentes. Para esta relação, método abstrato sendo implementado por outro método, é denominada de relação de *implementação*.

É necessária uma notação para documentar as relações apresentadas acima. A notação usada no trabalho é *Contratos de Reuso*.

A implementação dos contratos de reuso é feita através de um conjunto de classes, as quais formam a camada de *integração*. Para cada tipo de contrato, é apresentada uma classe específica..

Para cada classe da aplicação que possui relação de uso com componentes, é gerada uma *classe roteadora*. A *classe roteadora* é responsável pela instanciação e destruição de objetos dos componentes relacionados. A *classe roteadora* é formada pelos métodos definidas na relação de *uso*. Cada método definido em uma relação de *uso* é codificado na *classe roteadora* e possui uma referência para o método correspondente do componente. A construção da *classe roteadora* foi baseada em dois padrões de projetos conhecidos da literatura, Decorator e Facade [GAM94].

Uma *classe implementadora* é gerada para cada relação de *implementação* existente entre componente e a classe que provê a implementação do método. A *classe implementadora* é uma especialização do componentes e possui uma referência para a classe provedora da implementação. A formação deste padrão de projeto foi baseada em algumas características do padrão de projeto Adapter [GAM94].

Para a geração automática das classes da *camada de integração*, o trabalho apresenta um software assistente. A *camada de integração* é gerada a partir da relação estabelecida entre componentes e classes de aplicação e especificada pelos contratos de reuso. O software assistente apresenta uma interface textual que permite ao desenvolvedor da aplicação, especificar as relações de *uso* e de *implementação* existentes entre métodos de componentes e métodos da aplicação.

Os componentes pré-existentes estão catalogados em um repositório. A partir de um mecanismo de recuperação de componentes usando palavras-chave, o desenvolvedor da aplicação escolhe os componentes adequados para usar na construção da aplicação. O software assistente mostra uma interface que possibilita ao desenvolvedor da aplicação, estabelecer as relações de *uso* e de *implementação* existentes entre os componentes escolhidos e as classes da aplicação em construção.

Na versão atual do software assistente, os contratos de uso são especificados de forma textual. Como trabalho futuro, fica a idéia de adaptar o software assistente ao uso da notação gráfica dos contratos de reuso. Dentro da idéia de uso de notação gráfica, pode-se estudar a implementação dos contratos de reuso, usando os estereótipos de UML, em uma ferramenta CASE existente (por exemplo, Rational Rose).

Não foi avaliada a implementação dos componentes como aqui propostos em ambientes comerciais para construção de software baseado em componentes (Microsoft/COM, OMG/Corba, Java/RMI). Assim, outro trabalho futuro poderia ser a implementação da camada de *integração* e da camada de *componentes* usando arquiteturas como aquelas mencionadas.

Um outro trabalho proposto é aperfeiçoar a indexação e a recuperação de componentes de acordo com as necessidades do usuário. O trabalho apresentado aqui mostra um mecanismo de recuperação de componentes simples, não adequado para catalogo com volume significativo de componentes.

Anexo 1 - Código das classes de integração

Este anexo apresenta um exemplo de código fonte em Java de uma classe *roteadora* e de uma classe *implementadora*, como gerados pelo software assistente.

A classe *roteadora* RoteiroPacoteRot possui associações com três classes *implementadoras*: IExecucaoPlanoRoteiroPacote, ItransacaoLinhaRoteiroPacote e ITransacaoLugarTransacaoLinha. Possui, também, uma instanciação de objeto para cada uma destas três classes *implementadoras*, além, de uma referência para os objetos instanciados.

A aplicação desenvolvida utiliza o padrão ObjetoAssociação para tratar de associações entre duas classes [FAV98]. Este padrão possui uma estrutura que atende as funcionalidades específicas de qualquer tipo de associação.

```
//   ARQUIVO :  RoteiroPacoteRot.java
import java.util.date;
import ObjetoAssociacao;
import IExecucaoPlanoRoteiroPacote;
import ITransacaoLinhaRoteiroPacote;
import ITransacaoLugarTransacaoLinha;

public abstract class RoteiroPacoteRot extends ObjetoAssociacao
{
private IExecucaoPlanoRoteiroPacote objIExecucaoPlanoRoteiroPacote;
private ITransacaoLinhaRoteiroPacote objITransacaoLinhaRoteiroPacote;
private ItransacaoLugarTransacaoLinha objITransacaoLugarTransacaoLinha;

// CONSTRUTOR DA CLASSE
   public RoteiroPacoteRot()
   {
     IExecucaoPlanoRoteiroPacote objIExecucaoPlanoRoteiroPacote =
               new IExecucaoPlanoRoteiroPacote();
     ITransacaoLinhaRoteiroPacote objITransacaoLinhaRoteiroPacote =
               new ITransacaoLinhaRoteiroPacote ;
     ItransacaoLugarTransacaoLinha objITransacaoLugarTransacaoLinha =
               new ItransacaoLugarTransacaoLinha ;
   }

// METODOS DA CLASSE
   public float quantoTempo()
       {
        return  objIExecucaoPlanoRoteiroPacote.calculaDuracao();
       }

   public float calcDiariaItemRoteiro()
       {
        return objITransacaoLugraTransacaoLinha.calculaTransacao();
       }

// METODOS PARA RETORNO DE OBJETOS INSTANCIADOS

   public IExecucaoPlano$IexecucaoPlanoRoteiroPacote
```

```
          getExecucaoPlanoExecucaoPlano()
        {
         return  objIExecucaoPlanoRoteiroPacote;
        }

// METODO PARA DISSOCIAR OBJETOS RELACIONADOS A OUTRAS CLASSES
   public void dissociar(String nome)
        {
         super.dissociar(nome);
         objIExecucaoPlanoRoteiroPacote.dissociar(nome);
         objITransacaoLinhaRoteiroPacote.dissociar(nome);
         objITransacaoLugarTransacaoLinha.dissociar(nome);
        }

// METODO DE DESTRUICAO DOS OBJETOS INSTANCIADOS
   public void finalize() throws  Throwable
        {
         super.finalize();
         objIExecucaoPlanoRoteiroPacote = null;
         objITransacaoLinhaRoteiroPacote = null;
         objITransacaoLugarTransacaoLinha = null;
        }
}
```

FIGURA A.1 - Código de uma classe *roteadora*

A classe *implementadora* IExecucaoPlanoRoteiroPacote faz o papel de implementação entre o componente ExecucaoPlano e a classe *roteadora* RoteiroPacote. Caso a classe da aplicação não possuir uma classe *roteadora*, a associação é diretamente com a classe da aplicação.

```
//   ARQUIVO :  IExecucaoPlanoRoteiroPacote.java
import java.util.date;
import ExecucaoPlano;
import RoteiroPacoteRot;

public class IExecucaoPlanoRoteiroPacote extends ExecucaoPlano
{
public ExecucaoPlano()
{}

     public class  IExecucaoPlanoRoteiroPacote extends
ExecucaoPlano$ExecucaoPlano
      {
          private RoteiroPacoteRot roteiropacote;

          public IExecucaoPlanoRoteiroPacote( ExecucaoPlano$
execucaoplano,String nomePapel, String papelInverso,RoteiroPacoteRot
RoteiroPacote )
          {
               super( ExecucaoPlano,nomePapel,papelInverso);
               this.roteiropacote=roteiropacote;
          }
```

```
/* Função do Método: retorna data inicial da fase do plano */
          public  Date getDataInicio()
          {
            return roteiropacote.getDataChegada();
          }

/* Função do Método: retorna data final da fase  */
          public  Date getDataFim()
          {
               return roteiropacote.getDataSaida();
          }
      }
}
```

FIGURA A.2 - Código de uma classe *implementadora*

Bibliografia

[ALL 98] ALLEN, P.; FROST, S. **Component-Based Development for Enterprise Systems-applying the Select perspective**. England: Cambridge Press, 1998.

[ALT 97] ALTMEYER, J. et al. Application of a Generator-Based Software Development Method Supporting Model Reuse. In: INTERNATIONAL CONFERENCE ON ADVANCED INFORMATION SYSTEM ENGINEERING, CAISE, 9., 1997, Barcelona. **Proceedings...** Berlin: Springer-Verlag, 1997. (Lecture Notes in Computer Science, v.1250).

[BEL 98] BELLUR, U. The role of components and standards in software reuse. In: WORKSHOP ON COMPOSITIONAL SOFTWARE ARCHITECTURES, 1998, California. **Proceedings...** California: [s.n.], 1998.

[BOO 94] BOOCH, G. **Object-oriented analysis and design:** with applications. 2. ed. Redwood City: the Benjamin Cummings, 1994.

[BOS 97] BOSCH, J. Adapting Object-Oriented Components. In: ECOOP, 1997, Jyväskylä. **Proceedings...** [S.l.]: Springer-Verlag, 1997.

[BUD 96] BUDINSKY, F.J; VLISSIDES, J.M. Automatic code generation from design patterns. **IBM Systems Journal**, New York, v. 35, n. 2, 1996.

[BUS 95] BUSCHMANN, F. et al. **A System of Patterns.** Chichester: John Wiley & Sons, 1995.

[COA 97] COAD, P. et al. **Object models**: strategies, patterns and applications. New Jersey: Prentice Hall, 1997.

[COP 97] COPLIEN, J. Idioms and Patterns as Architectural Literature. **IEEE Software Special Issue on Objects, Patterns and Architectures**, [S.l.:[s.n], 1997.

[DHO 98] D'HONDT, T. et al. **Reuse Contracts.** Disponível por http://progwww.vub.ac.be/prog/pools/rcs/index.html (18 out. 1999).

[DSO 99] D'SOUZA, D.; WILLS, A.C. **Objects, Components and Frameworks with UML - the Catalysis approach**. [S.l.]: Addison-Wesley, 1999.

[ERI 98] ERIKSON, H.; PENKER, M. **UML Toolkit**. [S.l.]: John Wiley, 1998.

[FAV 98] FAVA, L.; KROTH, E. **Implementação de Padrões de Análise no Desenvolvimento de Software.** Santa Cruz do Sul: Unisc, 1998. Trabalho de Conclusão do curso de Ciência da Computação.

[FAY 97] FAYAD, M.; SCHIMIDT, D. Object-Oriented Application Frameworks. **Communications of the ACM**, New York, v.40, n.10, p71-77, Oct. 1997.

[GAM 94] GAMMA, E. et al. **Design Patterns**: Elements of Reusable Object-Oriented Software. Massachusetts: Addison Wesley Publishing Company, 1994.

[HOL 93] HÖLZLE, U. Integrating independently-developed components in object-oriented languages. In: EUROPEAN CONFERENCES OBJECT-ORIENTED PROGRAMMING, ECOOP, 7., 1993, Kaiserlautern. **Proceedings...** Kaiserslautern: [s.n.], 1993.

[ISA 96] ISAKOWITZ, T. et al. Supporting Search for Reusable Software Objects. **IEEE Transactions on Software Engineering**, New York, v. 22, n. 6, June 1996.

[JAC 97] JACOBSON, I. et al. **Software Reuse- architecture process and organization for business success**. New York: Addison-Wesley, 1997.

[JGS 97] JAPAN GUIDE/SHARE. **A Domain Model for Reuse in Analysis Phase**. Disponível por http://www.guide.org/pubs/jgsindex.htm (25 jul. 1999).

[JOH 88] JOHNSON, R.; FOOTE, B. Designing Reusable Classes. **Journal of Object-Oriented Programming**, [S.l.], v.1, n.2, p32-35, June/July 1998.

[JOH 97] JOHNSON, R. Frameworks = (components + patterns). **Communications of the ACM**, New York, v.40, n.10, p39-42, Oct. 1997.

[JOH 92] JOHNSON, R. Documenting frameworks using patterns. **SIGPLAN Notices**, New York, v.27, n.10, p.63-76, oct. 1992. Trabalho apresentado na Conference on Object-Oriented Programming System, OOPSLA, 1992.

[LAM 93] LAMPING, J. Typing the specialisation interface. **SIGPLAN Notices**, New York, p.435-451, 1993. Trabalho apresentado na Conference on Object-Oriented Programming System, OOPSLA, 1993.

[LAR 97] LARMAN, C. **Applying UML and Patterns**: An Introduction to Object-Oriented Analysis and Design . New Jersey: Prentice-Hall, 1997.

[LUC 96] LUCAS, C. **Documenting Reuse and Evolution with Reuse Contracts**. Bruxelas, Bégica: Departamento de Ciência da Computação, Universidade Livre, 1996. Tese de doutorado.

[MEI 96] MEIJER, T. et al. Class Composition in FACE, a Framework Adaptive Composition Environment. In: EUROPEAN CONFERENCES OBJECT-ORIENTED PROGRAMMING, ECOOP, 1996, Linz, AT. **Proceedings...** Berlin: Springer-Verlag, 1996.

[MEN 96] MENS, K. et al. Reuse Contracts: Managing Evolution in Adaptable Systems. In: EUROPEAN CONFERENCE OBJECT-ORIENTED PROGRAMMING, Workshop on Adaptability in Object-Oriented Software Development, ECOOP, 1996. **Proceedings...** [S.l. : s.n], 1996.

[MEN 98a] MENS, K. et. al. Supporting Disciplined Reuse and Evolution of UML Models. In: WORKSHOP ON UNIFIED MODELLING LANGUAGE, UML, 1998, Mulhouse, France. **Proceedings...** [S.l. : s.n.], 1998.

[MEN 98b] MENS, K. et. al. Giving Precise Semantics to Reuse and Evolution in UML. In: INTERNATIONAL WORKSHOP ON PRINCIPLES OF SOFTWARE EVOLUTION, 1998, Kyoto, Japan. **Proceedings...** Kyoto, Japan: [s.n.], 1998.

[MIL 95] MILI H. et al. Reusing Software: Issues and Research Directions. **IEEE Transactions on Software Engineering**, New York, v.21, n.6, June 1995.

[ORT 98] ORTEGA, J.; ROBERTS, G. The Concept of Software Structure and its Relations with Software Architecture and Software Patterns. In: EUROPEAN CONFERENCES OBJECT-ORIENTED PROGRAMMING, ECOOP, 1998, Bruxelas. **Proceedings...** Berlin: Springer-Verlag, 1998.

[PRE 94] PREE, W. **Design Patterns for Object-Oriented Software Development**. Workingham: Addison-Wesley, 1994.

[PRE 99] PREE, W.; SIKORA,H. **Creation and Reuse of flexible bean architectures in business applications.** Disponível por http://members.magnet.at/it-transfer/23_executive2.html em (20 nov. 1999).

[RIC 98] RICHNER, T. Describing Framework Architectures: more than Design Patterns. In: EUROPEAN CONFERENCES OBJECT-ORIENTED PROGRAMMING, ECOOP, 1998, Bruxelas. **Proceedings...** Berlin: Springer-Verlag, 1998.

[RUG 97] RUGGIA, R; AMBROSIO, A. P. A Toolkit for Reuse in Conceptual Modelling. In: INTERNATIONAL CONFERENCE ON ADVANCED INFORMATION SYSTEM ENGINEERING, CAISE, 9., 1997, Barcelona. **Proceedings...** Berlin: Springer-Verlag, 1997. (Lecture Notes in Computer Science, v.1250).

[RAT 99] RATIONAL SOFTWARE CORPORATION. **Rational Rose**, produto. Disponível por http://www.rational.com/products/rose/ (16 maio 1999).

[SIL 96] SILVA, A. R. et al. Three-Layered Framework with Separation of Concerns. **SIGPLAN Notices**, New York, 31, n.10, Oct. 1996. Trabalho apresentado na Conference on Object-Oriented Programming System, OOPSLA, Workshop on Exploration of Framework Design Principles, 1996.

[STE 96] STEYAERT, P. et al. Reuse Contracts: Managing the Evolution of Reusable Assets. **SIGPLAN Notices**, New York, v.31, n.10, p.268-285, Oct. 1996. Trabalho apresentado na Conference on Object-Oriented Programming System, OOPSLA, 1996.

[SUN 99] SUN MICROSYSTEMS. **Java**. Disponível por http://www.java.sun.com, (25 out. 1999).

[SZY 98] SZYPERSKI, C. **Component Software**. Harlow, Reino Unido: Addison Wesley Publishing Company,1998.

[TAL 95] TALIGENT. **Leveraging object-oriented frameworks**. Taligent Inc. white paper. Disponível por http://www.taligent.com (18 mar. 1999).

[WOO 88] WOOD, M.; SOMMERVILLE, I. **An Information Retrieval System for Software Components**. Glasgow-UK: Universidade de Strathclyde, 1988.