Uma arquitetura de software para reuso de componentes

Eduardo Kroth
Universidade de Santa Cruz do Sul
Departamento de Informática
kroth@dinf.unisc.br

Carlos Alberto Heuser
UFRGS/Informática
Porto Alegre RS - Brasil
heuser@inf.ufrgs.br

Resumo

O artigo trata do reuso de software, mais especificamente do problema de construção de aplicações a partir de componentes de software pré-existentes. Estes componentes possuem métodos concretos e "template" que oferecem uma funcionalidade específica para ser usada na aplicação em construção. Este artigo apresenta uma técnica que permite que o construtor de uma aplicação especifique a relação desejada entre as classes da aplicação e um conjunto de componentes pré-existentes. Nesta relação, identificam-se dois tipos de contratos de reuso: *uso* e *implementação*. O tipo de contrato *uso* estabelece regras para o uso de métodos de um componente e o tipo de contrato *implementação* estabelece regras para a implementação de métodos abstratos de um componente. Para permitir a implementação dos tipos de contratos, é proposta uma *arquitetura de integração*, formada por componentes, classes da aplicação e classes de integração. As classes de integração possuem especificações como padrões de projeto. O artigo descreve a arquitetura de aplicações que utilizam componentes e mostra como a arquitetura de integração deve ser construída.

Palavras chave: reuso de software, componentes, frameworks, padrões de projeto, contratos de reuso

Abstract

The paper addresses the software reuse, more specifically the problem of developing applications from pre-existing software components. These components have concrete and template methods which offer a specific functionality to be used in the application development. This paper presents a technique that allows the application developer to specify the expected relation between the application classes and a set of pre-existing components. In this relation, we can identify two kinds of reuse: use and implementation. The "use" contract establishes the rules for the use of component methods by application methods. The "implementation" contract establishes the rules for the implementation of abstract methods of a component. In order to allow the implementation of the contracts, a *integration architecture* is proposed, formed by components, application classes and integration classes. The integration classes are specified as design patterns. The paper describes the development of applications that use components and shows the way the integration architecture must be constructed.

Keywords: software reuse, components, frameworks, design patterns, reuse contracts

1. Introdução

A simples adoção da tecnologia de objetos, sem a existência de um padrão de reuso explícito e de um processo de desenvolvimento de software orientado ao reuso não fornece o sucesso esperado para produção de software em larga escala [JAC97]. Uma das técnicas de reuso de software para um determinado domínio de problema é a de implementar as funcionalidades comuns em *frameworks* [JOH88]. Um *framework* destinado à construção de aplicações em um domínio de problema específico é chamado de *framework vertical*, em oposição a um *framework horizontal* destinado a prover uma infra-estrutura comum a muitos domínios de problema (interface visual, comunicações, persistência,...) [FAY97]. O problema envolvido no uso de frameworks, especialmente no uso de frameworks verticais é que estes, por implementarem todo um domínio do problema, tendem a ser grandes, complexos, difíceis de entender e difíceis de usar. Para resolver este problema, várias soluções têm sido propostas, como o uso de ferramentas gráficas de visualização [MEU97], o uso de contratos [COD97] ou o uso de padrões de projeto (*design patterns*) [ODE97].

O presente artigo propõe uma alternativa diversa, a de dividir um framework em um conjunto de *mini-frameworks*, cada um consistindo de um pequeno conjunto de classes interrelacionadas. Cada um destes mini-frameworks pode ser tratado como um *componente* de software [SZY98].

Usando uma arquitetura de software em três camadas (interface homem máquina, domínio do problema e gerenciamento de dados), este artigo concentra-se na camada de domínio do problema para desenvolver sua explanação. Dentro da proposta deste trabalho, a camada de domínio do problema é dividida em três camadas distintas de acordo com o interesse de cada uma delas [SIL96]. Das três camadas, uma é genérica e é formada por componentes que oferecem soluções para pequenos problemas específicos. A outra camada é composta por componentes de uma aplicação qualquer. A camada intermediária faz a implementação das relações existentes entre classes da aplicação e de componentes.

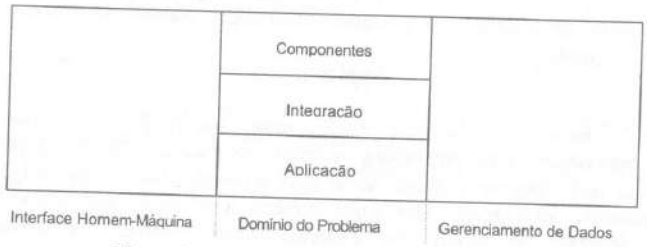

Figura 1 – Arquitetura de software em três camadas

O artigo apresenta uma arquitetura de software que permite acoplar as classes de uma aplicação a um conjunto de componentes como os descritos acima. Além disto, o trabalho descreve um software assistente que permite construir de forma automática as classes da camada de integração.

O artigo está organizado da seguinte forma. O capítulo 2 descreve as relações existentes entre uma aplicação e componentes. O capítulo 3 trata da notação de contratos de reuso como forma de

representar as relações entre aplicações e componentes. Neste capítulo, é apresentado dois novos tipos de contratos. Os dois novos tipos de contratos recebem propriedades que motivam a geração de uma *arquitetura de integração* que é desenvolvida no capítulo 4. O capítulo 5 descreve um software assistente como uma ferramenta de apoio ao desenvolvedor de aplicações no uso de componentes.

2. Relações entre uma aplicação e componentes

O exemplo descrito a seguir mostra as possíveis relações entre uma aplicação e os componentes. A figura 2 apresenta classes de componentes e classes de uma aplicação para agência de viagens. Os componentes são inspirados em um conjunto de padrões de análise descritos em [COA97]. Os componentes possuem métodos concretos, "template" e abstratos. Os métodos abstratos estão identificados pela escrita em itálico. Os métodos concretos e os métodos "template" estão disponíveis para que o desenvolvedor da aplicação relacione com os métodos da aplicação que possuam funcionalidades idênticas. Na figura 2, os componentes estão dentro de "pacotes" cujos nomes identificam o componente.

As classes da aplicação apresentadas na figura 2 tratam de um sistema de agência de viagens. Estas classes são desenvolvidas pelo desenvolvedor da aplicação, ator responsável pela definição de relações entre a aplicação e os componentes.
Uma aplicação pode ser construída através da combinação de vários componentes. A relação entre uma aplicação e os vários componentes que ela utiliza apresentam as seguintes características:

- uma classe da aplicação pode não necessitar de todos os métodos que aparecem em uma classe de um componente,
- uma classe de aplicação pode utilizar ou implementar métodos de diferentes componentes,
- um método abstrato de um componente pode ser implementado por um método concreto da aplicação ou por um método concreto de outro componente, e
- os métodos da aplicação podem ter nomes diferentes dos nomes que possuem nos componentes.

Usando o modelo de objetos da figura 2, pode-se estabelecer algumas relações entre a aplicação e os componentes. Por exemplo, o desenvolvedor da aplicação estabelece que o método PacoteViagem.calculaDiarias() possui a mesma funcionalidade que o método Transação.somaLinhas(), assim como o método RoteiroPacote.quantoTempo() em relação ao método ExecuçãoPlano.calculaDuração(). A estas relações entre métodos, dá-se o nome de relação de *uso*, e diz-se que o método PacoteViagem.calculaDiarias() usa o método Transação.somaLinhas(), e assim por diante.

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.

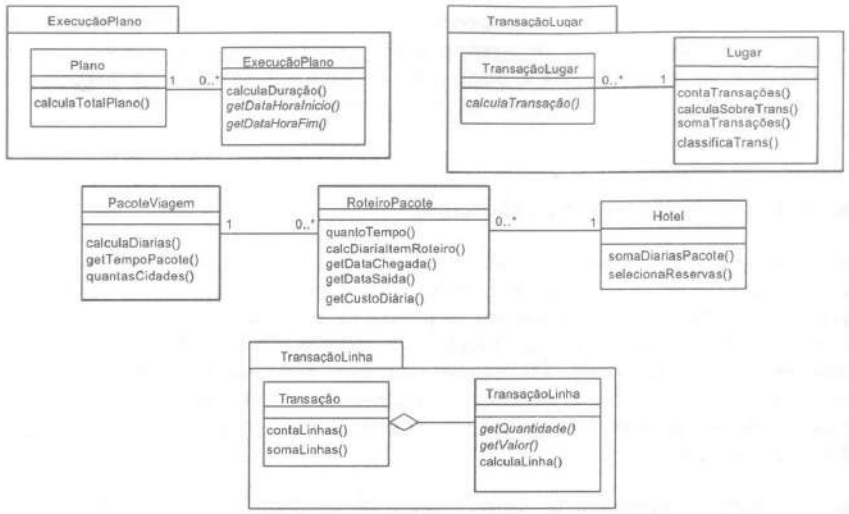

Figura 2 – Exemplo de classes de aplicação e de componentes

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*. Um método abstrato de componente pode ser implementado por um método de uma classe da aplicação ou por um método de outro componente.

3. Usando contratos de reuso para representar as relações

Este artigo usa os conceitos da notação de *contratos de reuso* para representar as relações de *uso* e de *implementação* estabelecidas entre uma aplicação e os componentes.

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 Livre 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. As relações entre componentes e classes de aplicação podem ser classificadas em tipos básicos chamados de *tipos de contratos de reuso*. 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 presente trabalho apresenta dois novos tipos de contratos. Cada tipo de contrato citado neste capítulo é especificado por uma sintaxe e por propriedades. A sintaxe descreve a forma de escrita e de leitura de um tipo de contrato. As propriedades relatam as características de cada tipo de contrato. Os exemplos citados referem-se à figura 3.

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: (i) possibilidade de renomear métodos de um componente quando usado em uma aplicação; (ii) uso parcial dos métodos de um componente e, (iii) 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. Por exemplo, o método ExecuçãoPlano.calculaDuração() é usado pelo método RoteiroPacote.quantoTempo() e não possuem o mesmo nome.

Propriedade 2: uso parcial dos métodos de um componente. 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. Na figura 3, a classe de aplicação Hotel não utiliza todos os métodos da classe de componente Lugar.

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 de 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 relacionar 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. Por exemplo, a classe de aplicação RoteiroPacote está relacionada com as classes de componentes TransaçãoLinha, ExecuçãoPlano e TransaçãoLugar.

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 3 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. 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 3, 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.

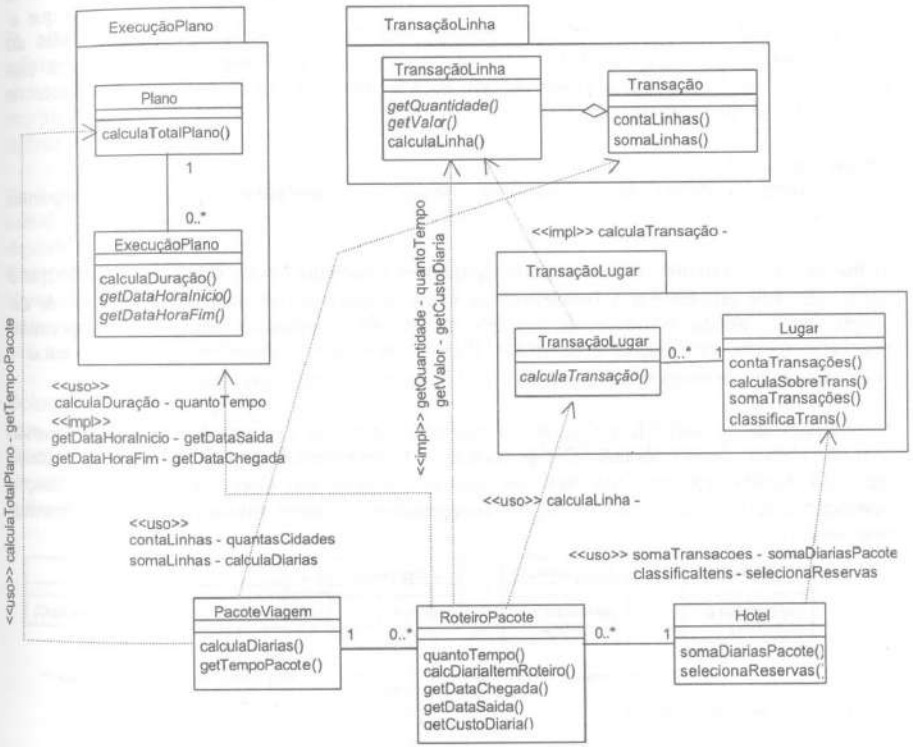

Figura 3 – Aplicação dos tipos de contratos no sistema de Viagens

4. Arquitetura de integração

Este capítulo apresenta uma arquitetura de classes para a implementação dos tipos de contratos de reuso propostos no capítulo anterior. A arquitetura de integração é necessária pois a notação de

Contratos de Reuso serve somente para o do diagrama de objetos, não sendo diretamente implementada em linguagens de programação. A implementação dos novos tipos de contratos é apresentada a seguir na forma de padrões de projeto.

4.1 Padrão de projeto roteador

Contexto

O tipo de contrato uso existente entre métodos de uma aplicação e de componentes apresenta propriedades que requerem implementação. O tipo de contrato uso exige que a classe da aplicação que usar este tipo de contrato para relacionar-se com classes de componentes, deve proporcionar a implementação de certas funcionalidades apresentadas pelo tipo de contrato. A implementação destas funcionalidades afeta a estrutura original da classe de aplicação. Para que a estrutura original da classe de aplicação não seja alterada, sugere-se que as funcionalidades do tipo de contrato uso sejam implementadas em uma classe especificamente construída para este fim. Esta classe deve ser uma classe abstrata e construída individualmente para cada classe de aplicação que possuir o tipo de contrato *uso*.

Problemas

A implementação da relação de uso entre aplicação e componentes apresenta os seguintes requisitos:

a) 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.

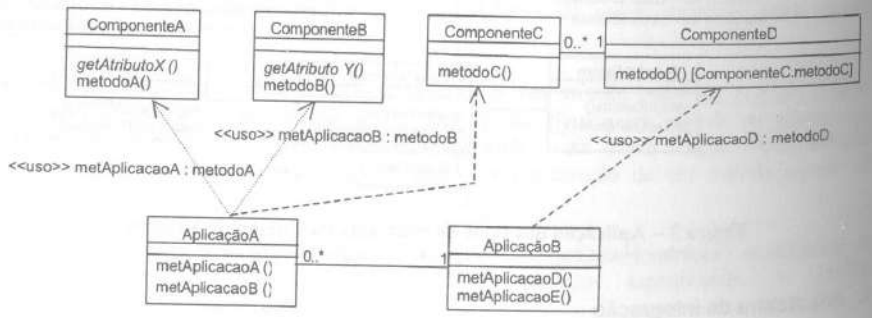

Figura 4 – Modelo de objetos usando tipo de contrato uso

b) 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.

c) 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.

d) 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.

Solução

Para construir a solução da classe roteadora utilizou-se dois outros padrões de projetos, Decorator e Facade[GAM94]. A classe roteadora é uma classe abstrata e existe para cada classe da aplicação que possua um contrato de reuso com classes de componentes. A classe da aplicação herda as propriedades de sua correspondente classe roteadora.

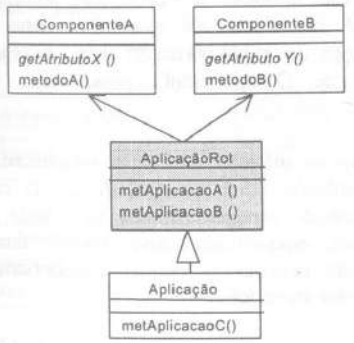

Figura 5 – Modelo de objetos usando classes roteadoras

A nomenclatura das classes roteadoras tem a seguinte ordem: nome da classe da aplicação acrescida do sufixo Rot. Usando o exemplo da figura 5, observa-se que a classe roteadora criada para atender a associação entre Aplicação (aplicação) com ComponenteA e ComponenteB (componentes) é denominada de AplicaçãoRot.

Uma classe roteadora deve possuir as funcionalidades de instanciação e de destruição dos componentes implementadas nos seus correspondentes métodos de mesma funcionalidade. A classe roteadora deve, também, possuir referências para todos os componentes relacionados pelo tipo de contrato uso. Os métodos estabelecidos pelo tipo de contrato uso devem estar na classe roteadora, contendo a invocação do método correspondente no componente.

4.2 Padrão de projeto implementador

Contexto

O tipo de contrato implementação é usado quando métodos abstratos de um componente são invocados por outros métodos do componente. Os métodos abstratos, portanto, devem ser implementados em uma classe especializada para que não altere a estrutura original do componente. O tipo de contrato implementação indica o método concreto para a implementação do método abstrato. Este método concreto pode estar na estrutura de uma classe de aplicação ou na estrutura de um outro componente. Sendo assim, um componente relacionado com classes de aplicação pode possuir métodos abstratos implementados em diferentes classes, tanto da aplicação como de outros componentes, fato que deve ser considerado neste problema.

Problemas

Para a implementação do tipo de contrato implementação, são necessárias as seguintes especificações:

a) 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.

b) 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.

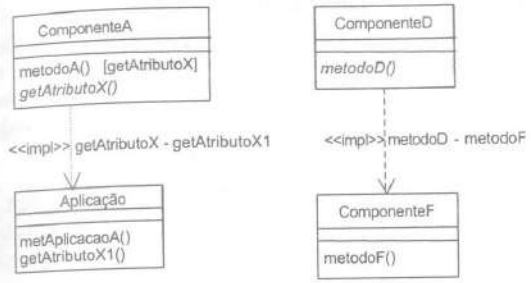

Figura 6 – Exemplos de relação de implementação

Solução

Para a implementação do tipo de contrato implementação, baseou-se no padrão de projeto Adapter [GAM94]. Para cada tipo de contrato implementação que associe uma classe de componente e uma classe responsável pela implementação, é gerada uma classe implementadora. A classe implementadora é resultado deste artigo e se posiciona, no modelo de objetos, como uma subclasse de uma classe de componente. 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 classe do componente é especializada pela classe implementadora, ou seja, o objeto instanciado na camada do componente é o objeto da classe implementadora.

Usando a figura 6 como exemplo, a relação de implementação estabelece que o método ComponenteA.getAtributoX é implementado por AplicaçãoA.getAtributoX1 e o método ComponenteD.metodoD é implementado por ComponenteF.metodoF.

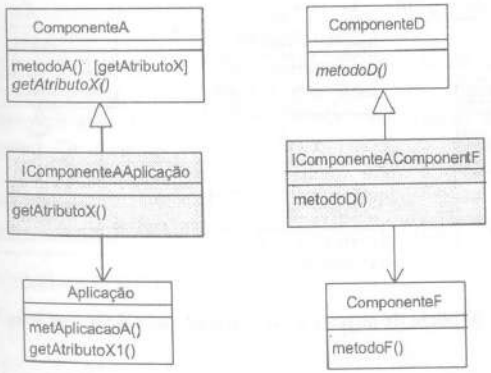

Figura 7 – Modelos de objetos usando classes implementadoras

4.3 Estudo de caso usando a arquitetura de integração

Usando o estudo de caso apresentado na figura 3 - sistema de agência de viagens usando alguns componentes pré-existentes - pode-se gerar um modelo de objetos com a camada de integração promovendo a implementação das relações de uso e de implementação. Para as relações de uso, são construídas as classes roteadoras. Na figura 8, as classes roteadoras e implementadoras estão em destaque com cor cinza apenas para identificá-las das outras classes.

Na figura 8, as classes roteadoras são: HotelRot, RoteiroPacoteRot e PacoteViagemRot. Cada classe roteadora é uma classe abstrata e sua estrutura é herdada pela classe correspondente da aplicação. As classes roteadoras estão referenciadas aos componentes estabelecidos pelas relações de uso.

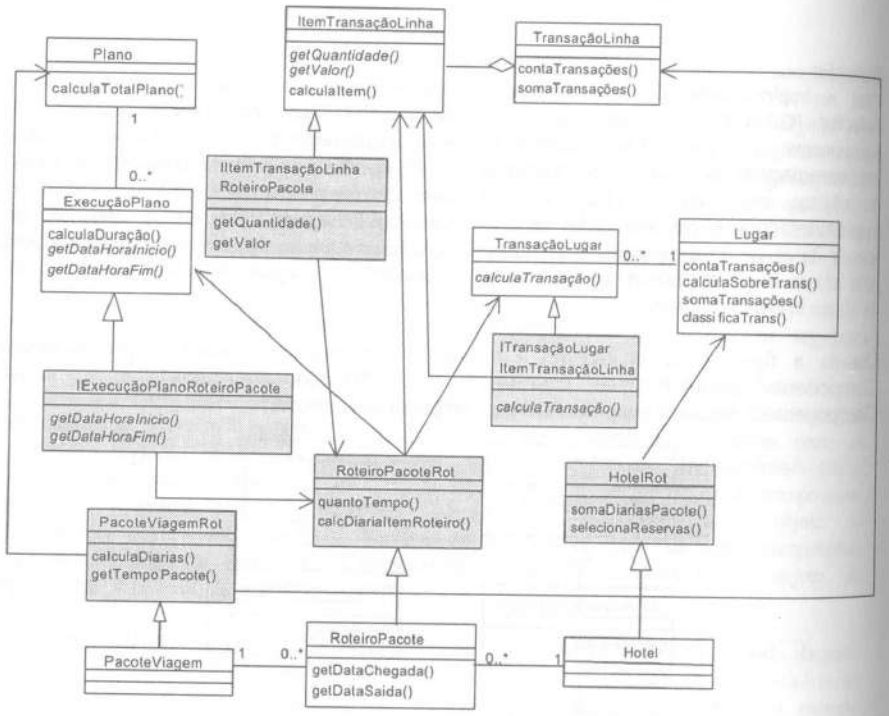

Figura 8 – Modelo de objetos com a arquitetura de integração formada

As classes implementadoras correspondem às relações de implementação estabelecidas entre classes de componentes e classes da aplicação ou de outros componentes. Cada classe implementadora é construída para cada combinação de classe de componente com sua

correspondente classe de implementação. As classes implementadoras construídas para o estudo de caso são: IItemTransaçãoLinhaRoteiroPacote, ItransaçãoLugarItemTransaçãoLinha e IExecuçãoPlanoRoteiroPacote.

4 Software Assistente

A geração das classes que compõe a camada de integração pode ser feita por um software específico, pois as regras de construção das classes roteadoras e implementadoras estão bem definidas. O software assistente é considerado como uma ferramenta de apoio ao desenvolvedor de aplicações [KRO99]. O software assistente deve basear-se nos componentes pré-existentes armazenados em um repositório específico, nas classes de aplicação que se deseja usar e nos contratos de reuso estabelecidos entre classes de aplicação e de componentes.

Os componentes são armazenados em uma estrutura que possibilita a busca dos componentes e a identificação do conjunto de métodos disponíveis ao desenvolvedor da aplicação. Neste software assistente, um componente pode ser especificado na forma de padrões de projeto: contexto, problema e solução.

A estrutura de armazenamento de métodos trata de registrar as propriedades do método, tais como: função, tipo de retorno, nível de acesso, abstrato ou não, parâmetros usados e a lista de métodos invocados pelo método em questão. A lista de dependência de métodos invocados é importante, principalmente, quando existem métodos abstratos invocados. Os métodos abstratos invocados requerem que o desenvolvedor da aplicação providencie uma implementação. Neste caso, é usado o tipo de contrato implementação.

As estruturas das classes de uma aplicação são informadas ao software assistente para que o desenvolvedor da aplicação possa estabelecer as relações de uso e de implementação. As relações estabelecidas entre classes de uma aplicação e componentes são armazenadas no software assistente para que sirvam ao módulo de geração da camada de integração.

A especificação de relações de uso e de implementação é feita em uma interface composta pelas classes da aplicação e pelos componentes escolhidos pelo desenvolvedor da aplicação. Nesta interface, o desenvolvedor da aplicação estabelece os métodos da aplicação que usam métodos de componentes. Os métodos de componentes escolhidos são avaliados pelo software assistente, indicando quais os métodos abstratos requerem implementação. Os métodos abstratos selecionados são informados ao desenvolvedor da aplicação, sendo este responsável pela indicação de métodos concretos que sirvam de implementação a estes métodos abstratos.

5 Conclusões e Trabalhos Futuros

Este artigo apresenta (i) uma técnica para especificação de relações entre componentes e classes de aplicação usando uma notação gráfica, (ii) uma arquitetura de software para a implementação das relações definidas e (iii) 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*.

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. 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.

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.

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.

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 estas.

Agradecimentos

Os autores agradecem o apoio da FAPERGS na participação deste evento e aos revisores deste artigo que contribuíram com sugestões.

Referências

[BOS97] BOSCH, J. Adapting Object-Oriented Components. Jyväskylä, Finland - Ed. Springer **Proceedings of the ...** ECOOP'97 Workshops, junho 1997.

[COA97] COAD, P. et al. **Object models: strategies, patterns and applications**. New Jersey, Ed. Prentice Hall, 1997

[COD97] CODIENE, W. et al. From custom applications to domain-specific frameworks. **Communications of the ACM**, 40(10): 71-77.

[DHO98] D'HONDT, T.; et al. **Reuse Contracts** Disponível em http://progwww.vub.ac.be/prog/pools/rcs/index.html (18 out 1999)

[FAY97] FAYAD, M.; SCHIMIDT, D. Object-Oriented Application Frameworks. **Communications of the ACM**. 40(10):71-77

[GAM94] GAMMA, E. et al. **Design Patterns: Elements of Reusable Object-Oriented Software**. Massachusetts, Addison Wesley Publishing Company, 1994.

[JAC97] JACOBSON, I. et al. **Software Reuse- architecture process and organization for business success**. Ed. Addison-Wesley, New York, 1997.

[JOH88] JOHNSON, R.; FOOTE, B. Designing Reusable Classes. **Journal of Object-Oriented Programming**, junho/julho 1988, volume 1 número 2.p32-35

[KRO99] KROTH, E. et al. Software Assistente no Uso de Componentes. **Proceedings of the** XII – Simpósio Brasileiro de Engenharia de Software – Sessão de Ferramentas, outubro 1999.

[LUC96] LUCAS, C. **Documenting Reuse and Evolution with Reuse Contracts**. Tese de doutorado, Departamento de Ciência da Computação, Universidade Vrije, Bruxelas, Bégica. 1996

[MEI96] MEIJER, T. et al. Class Composition in FACE, a Framework Adaptive Composition Environment. **Proceedings of the...** ECOOP'96 Workshop Reader , julho 1996.

[MEN96] MENS, K. et. al. Reuse Contracts: Managing Evolution in Adaptable Systems. **Proceedings of the...** ECOOP'96 Workshop on Adaptability in Object-Oriented Software Development, 1996

[MEN98a] MENS, K. et. al. Supporting Disciplined Reuse and Evolution of UML Models. **Proceedings of the...** UML'98 Workshop, Mulhouse, França, junho 1998.

[MEN98b] MENS, K. et. al. Giving Precise Semantics to Reuse and Evolution in UML. **Proceedings of the...** ICSE'98 International Workshop on Principles of Software Evolution, Kyoto, Japan, 1998.

[MEU97] MEUSEL, M. Czarecki; Kopf, W. A model for structuring user documention of object-oriented frameworks using patterns and hypertext. **Proceedings of the ...** ECOOP'97 (LNCS 1241), pp.498-510, Springer-Verlag, 1997.

[ODE97] ODENTHAL, G.; Quiebel-Cirkel,K. Using Patterns for Design and Documentation. **Proceedings of the...** ECOOP'97 (LNCS 1241), pp. 511-529, Springer-Verlag, 1997.

[SIL96] SILVA, A. R. et. al. Three-Layered Framework with Separation of Concerns. **Proceedings of the...** OOPSLA'96 Workshop on Exploration of Framework Design Principles, Califórnia, EUA, outubro, 1996

[STE96] Steyaert, P. et al. Reuse Contracts: Managing the Evolution of Reusable Assets. **Proceedings of the...** OOPSLA'96 Conference on Object-Oriented Programming, Systems, Languages and Applications, ACM SIGPLAN Notices, vol. 31, nº. 10, outubro 1996, pp. 268-285

[SZY98] SZYPERSKI, C. **Component Software**. Harlow, Reino Unido, Addison Wesley Publishing Company,1998