Uma Extensão do Fluxo de Análise e Projeto do RUP com suporte a Desenvolvimento Baseado em Componentes

Eduardo Santana de Almeida
esa2@cin.ufpe.br

Universidade Federal de Pernambuco
Centro de Informática

Resumo

Atualmente, muitas empresas têm adotado ou estão em processo de adoção do Rational Unified Process (RUP). Dentre os fatores que justificam este processo, destacam-se: a utilização da notação Unified Modeling Language (UML), sendo um padrão utilizado por grande parte da comunidade de desenvolvimento de software; a variedade de ferramentas que oferecem suporte a sua execução; e, por fim, a amplitude do processo, permitindo cobrir grande parte das atividades relacionadas com o desenvolvimento de software. Entretanto, cada vez mais, as empresas estão buscando métodos, processos e ferramentas que incrementem o nível de reutilização nos seus projetos. Contudo, no RUP, o suporte à abordagem de componentes é encorajado, porém esse suporte ainda é muito influenciado e restringido pela notação UML. Assim, este artigo apresenta uma proposta de extensão do RUP com suporte a Desenvolvimento Baseado em Componentes, que procura orientar tanto o desenvolvimento, como a reutilização de componentes de um domínio do problema.

Palavras-chaves: RUP, Desenvolvimento Baseado em Componentes (DBC), Reutilização.

1. Introdução

Uma das razões mais convincentes para a adoção de abordagens de desenvolvimento de software baseado em componentes, com ou sem objetos, é a premissa de reutilização. A idéia é construir software de componentes já existentes primariamente pela montagem e substituição de suas partes interoperáveis. Estes componentes abrangem desde controles de interfaces de usuário, como *listboxes* e HTML browsers, até componentes para persistência em banco de dados e distribuição. As implicações de redução de tempo de desenvolvimento e melhoria da qualidade do produto tornam esta abordagem muito atrativa [Szyperski, 1998].

A reutilização é uma variedade de técnicas que visa reaproveitar ao máximo o trabalho de análise, projeto e implementação já concluído. O objetivo é não reinventar a mesma idéia cada vez que um novo produto tiver que ser desenvolvido, mas sim organizar o trabalho já realizado e implantá-lo imediatamente em um novo contexto. Deste modo, mais produtos podem ser entregues em menores tempos, os custos referentes às tarefas de manutenção são reduzidos, uma vez que as melhorias realizadas em um segmento do projeto refletir-se-ão em todos os projetos nos quais este está sendo utilizado e, por fim, tem-se uma melhoria de qualidade, visto que os componentes reutilizados já foram bem testados.

Para que a reutilização possa ser efetiva, deve-se considerá-la em todas as fases do processo de desenvolvimento do software. Portanto, o Desenvolvimento Baseado em Componentes (DBC) deve oferecer métodos, técnicas e ferramentas que suportem desde a identificação e especificação dos componentes, do domínio do problema, até o seu projeto e

implementação em uma linguagem orientada a componentes. Além disso, o DBC deve empregar inter-relações entre componentes já existentes, previamente testados, visando reduzir a complexidade e o custo de desenvolvimento do software [Werner, 2000].

Atualmente, muitas empresas têm adotado ou estão em processo de adoção do Rational Unified Process (RUP). Dentre os fatores que justificam este processo, destacam-se: a utilização da notação Unified Modeling Language (UML), sendo um padrão utilizado por grande parte da comunidade de desenvolvimento de software; a variedade de ferramentas que oferecem suporte a sua execução; e, por fim, a amplitude do processo, permitindo cobrir grande parte das atividades relacionadas com o desenvolvimento de software. Entretanto, cada vez mais, as empresas estão buscando métodos, processos e ferramentas que incrementem o nível de reutilização nos seus projetos. Contudo, no RUP, o suporte à abordagem de DBC é encorajado, porém esse suporte ainda é muito influenciado e restringido pela notação UML. A visão do método no conceito de componente ainda é muito limitada, sendo voltada mais ao nível físico de pacotes de código de programação, baseando-se em componentes UML e diagramas de implantação. Isto é comprovado pela definição do RUP de componente como "uma não trivial, quase independente, e substituível parte de um sistema que realiza uma função clara no contexto de uma arquitetura bem definida". RUP utiliza o conceito da UML de subsistema para a proposta de modelagem dos componentes, mas não detalha como esse processo deve ser realizado [Boertin, 2001]. Além disso, o processo não define claramente, como adicionar requisitos não funcionais, como, por exemplo, distribuição e persistência em banco de dados e como se dá o desenvolvimento das aplicações reutilizando os componentes previamente construídos, deixando esta tarefa totalmente a cargo do desenvolvedor.

Assim, este artigo apresenta uma proposta de extensão do RUP, com suporte a DBC, a fim de suprir as limitações atuais referentes à reutilização. Por questões de limitações de escopo, apenas o aspecto estático do RUP foi considerado.

O artigo está organizado em seis seções. A primeira seção contém esta introdução. A seção 2 apresenta brevemente o RUP. A seção 3 descreve o método Catalysis discutindo alguns de seus princípios. A seção 4 descreve a proposta de extensão propriamente dita. A seção 5 apresenta os trabalhos relacionados e, finalmente, a seção 6 apresenta as conclusões e perspectivas de trabalhos futuros.

2. Rational Unified Process (RUP)

O Rational Unified Process (RUP) [Jacobson, 2001] é o processo de engenharia de software da Rational Software Corporation. O processo é iterativo, orientado a objetos, controlado e suportado através de ferramentas, podendo ser aplicado a uma variedade de projetos de desenvolvimento de software. O RUP alcançou grande popularidade na indústria de software, especialmente, entre usuários das ferramentas Rational que oferecem suporte à modelagem e à implementação.

Segundo os autores [Jacobson, 2001], o RUP representa mais do que um simples processo de desenvolvimento de software capaz de transformar requisitos do usuário em um sistema de software. Ele é um *framework* de processo genérico, capaz de ser especializado para diferentes áreas de aplicação, tipos de organização, níveis de competência e tamanhos de projeto.

O processo de desenvolvimento utilizando o RUP é dividido em quatro fases: concepção, elaboração, construção e transição, e um número arbitrário de iterações.

Adicionalmente, o RUP é estruturado através de um conjunto de *workflows*, os quais agrupam diferentes tipos de atividades.

O RUP não introduz nenhum novo conceito de modelagem, pois o mesmo se apóia na utilização da Unified Modeling Language (UML) para estas tarefas, porém introduz novos conceitos, como artefatos, atividades e *workflows*. Esses conceitos capturam abstrações, como: “quem” deve fazer “o quê”, “como” e “quando” [Boertin, 2001].

O conceito chave do RUP é a definição de atividades (*workflows*) realizadas do começo ao fim do ciclo de vida de desenvolvimento, como a identificação dos requisitos, análise, projeto, implementação e teste. Uma das principais vantagens do RUP é prover uma oportunidade para o desenvolvimento de sistemas, de forma iterativa e incremental, o que é visto, atualmente, como uma das melhores práticas de desenvolvimento [Jacobson, 2001].

3. Catalysis

Na tentativa de melhorar o processo de desenvolvimento de software orientado a objetos, pesquisadores desenvolveram vários métodos, destacando-se o OMT [Rumbaugh, 1991] e o de Booch [Booch, 1994]. Esses métodos, embora tenham contribuído continuamente para produzir software com melhor qualidade, apresentavam problemas de rastreamento de requisitos, semântica para verificar consistência, níveis de granularidade e refinamento e distinção entre modelo do domínio, modelo do sistema e arquitetura [D’Souza, 1999]. A evolução desses métodos resultou em um método integrado de técnicas para construir sistemas distribuídos orientado a objetos, denominado Catalysis [D’Souza, 1999].

Catalysis é uma iniciativa de pesquisa desenvolvida na Universidade de Brighton, Inglaterra, por D’Souza e Wills [D’Souza, 1999]. Conforme os autores, o método Catalysis é um método de desenvolvimento baseado em componentes completo, cobrindo todas as fases do desenvolvimento de um componente a ser utilizado no desenvolvimento de uma dada aplicação, desde a especificação até sua implementação.

Catalysis é baseado em um conjunto de princípios [D’Souza, 1999] para o desenvolvimento de software. Dentre esses princípios, pode-se destacar: a abstração, precisão, refinamento, componentes “plug-in” e algumas leis de reutilização.

O princípio abstração orienta o desenvolvedor na busca de aspectos essenciais do sistema, dispensando detalhes que não são relevantes para o contexto do sistema. O princípio precisão tem como objetivo descobrir erros e inconsistências na modelagem. Refinamentos sucessivos de transições de uma fase para outra são aplicados, buscando-se obter artefatos cada vez mais precisos e propícios à reutilização. O princípio componentes “plug-in” suporta a reutilização de componentes, a fim de construir outros. Por fim, a principal lei de reutilização do método Catalysis é não reutilizar código sem reutilizar modelos de especificações desses códigos.

O processo de desenvolvimento de software utilizando Catalysis é dividido em três níveis: Domínio do Problema, Especificação dos Componentes e Projeto Interno dos Componentes. Estes níveis correspondem às fases tradicionais do ciclo de vida do software: elicitação, análise e modelagem, e projeto dos requisitos. Assim como no ciclo tradicional de desenvolvimento de software, em Catalysis, pode-se retornar ao nível anterior para remover inconsistências com o domínio do problema e com a especificação [D’Souza, 1999].

No nível Domínio do Problema é dada ênfase no entendimento do problema, especificando-se “o quê” o sistema deve atender para solucionar o problema. Em seguida,

na Especificação dos Componentes, é descrito o comportamento do sistema de uma forma não ambígua e, no último nível, Projeto Interno dos Componentes, define-se como serão implementados os requisitos especificados para os componentes [D'Souza, 1999].

4. Uma Extensão do Fluxo de Análise e Projeto do RUP com suporte a Desenvolvimento Baseado em Componentes (DBC)

De modo a estender o fluxo de análise e projeto do RUP com suporte a DBC, tivemos que adicionar algumas diretrizes especificas, como, por exemplo, adaptar atividades do fluxo de análise e projeto, como também realizar a adição de uma nova atividade e de um novo perfil de trabalhador, neste caso, no fluxo de projeto.

A adaptação consiste, basicamente, na utilização de artefatos (*Modelo de Tipos*, *Framework de Modelos* e *Modelo de Aplicação do Framework*) e princípios do método Catalysis (ver seção 3), assim como o suporte a reutilização, permitindo tanto o desenvolvimento "para reutilização", quanto "com reutilização". A Figura 1 apresenta a proposta de extensão, onde estão identificadas as atividades adaptadas e adicionadas, assim como o perfil do novo trabalhador.

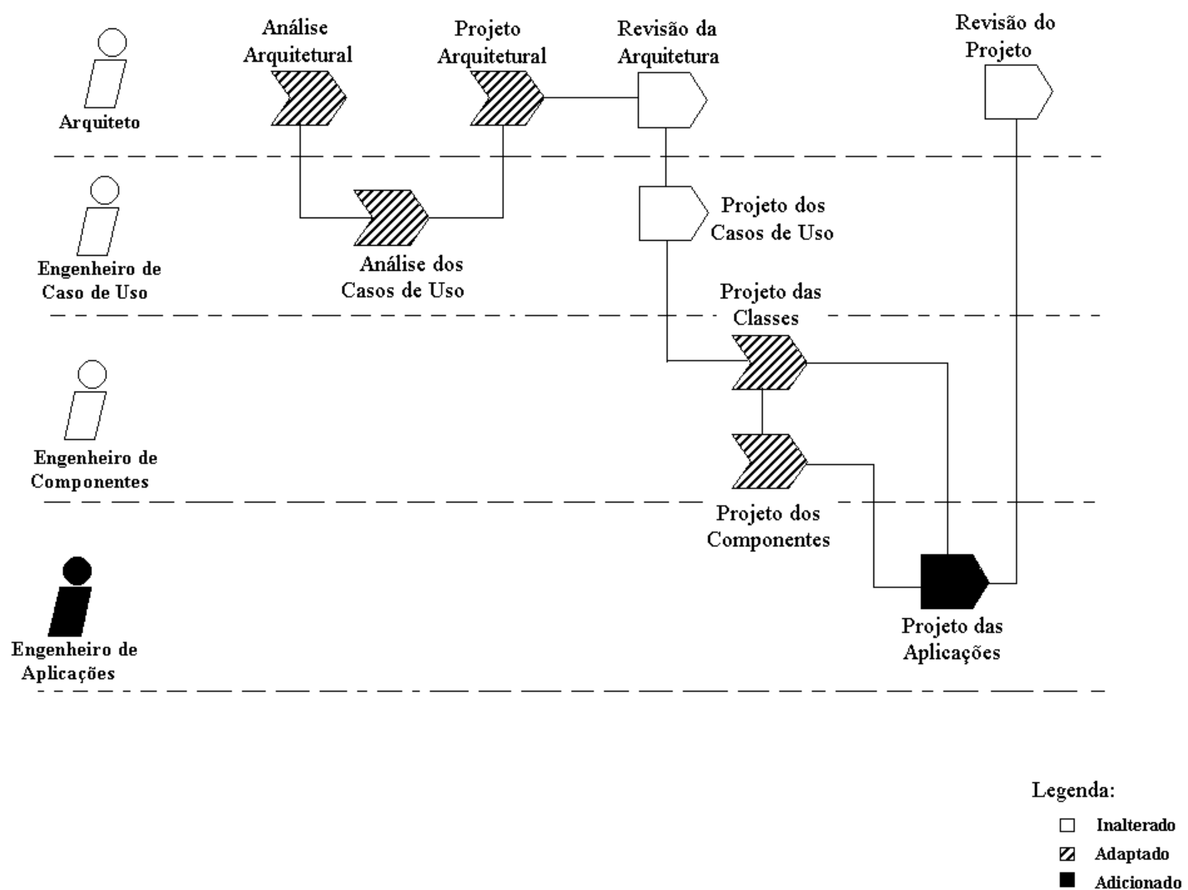

Figura 1. Extensão do Fluxo de Análise e Projeto do RUP com suporte a DBC

Segue-se uma apresentação detalhada de cada atividade do fluxo de análise e projeto estendido. As atividades que não sofreram modificações (Projeto dos Casos de Uso,

Revisão da arquitetura e do projeto) não serão consideradas, uma vez que seu fluxo de atividades manteve-se inalterado.

4.1 Análise Arquitetural

O objetivo da análise arquitetural é identificar os pacotes de análise, os tipos óbvios do domínio e os requisitos especiais comuns. A Tabela 1 apresenta uma visão geral desta atividade.

Tabela 1. Visão geral da atividade Análise Arquitetural

Objetivo (s)
➢ Entender o vocabulário inicial do domínio
Passos
1. Identificar Pacotes de Análise
2. Identificar Tipos óbvios do Domínio
3. Identificar Requisitos especiais comuns

Esta atividade foi adaptada com a utilização do conceito de tipos, oriundo do método Catalysis, de modo a iniciar a inserção de artefatos propícios à reutilização. O modelo original do RUP trabalhava com a utilização de classes de análise, assim, mesmo considerando essas classes, como entidades básicas de análise, optamos pela utilização dos tipos, o qual introduz um novo foco no processo: a orientação ao domínio. William Frakes [1994], um dos maiores pesquisadores na área de reutilização, advoga que o conceito chave em reutilização é o domínio, o qual pode ser definido como uma área de aplicação, ou mais formalmente, um conjunto de sistemas que compartilham decisões de projeto.

O resultado mais significativo desta atividade é a introdução dos princípios de reutilização, particularmente, a noção de domínio. Com base nos tipos iniciais identificados será especificada a arquitetura abstrata do domínio, conforme será apresentada na próxima seção.

4.2 Análise dos Casos de Uso

Uma vez identificado os tipos básicos, a análise dos casos de uso inicia-se. O objetivo principal desta atividade é especificar a arquitetura do domínio, com base nas informações do fluxo de eventos dos casos de uso, e através da realização da análise do domínio[1]. A proposta da utilização da análise do domínio é para que a informação usada no desenvolvimento do sistema, para um determinado domínio, seja identificada, capturada e organizada com a proposta de ser reutilizada. A Tabela 2 apresenta uma visão geral desta atividade.

---

[1] Para a realização da análise do domínio não foi definido nenhum método específico, como, por exemplo, o Feature Oriented Domain Analysis (FODA) [Kang, 1990] ou o Organization Domain Modelling (ODM) [Simos, 1996], porém, este processo pode ser adicionado posteriormente.

Tabela 2. Visão Geral da atividade Análise dos Casos de Uso

Objetivo (s)
➢ Especificar a arquitetura do domínio
Passos
1. Identificar Tipos do domínio
2. Descrever as colaborações entre os tipos
3. Descrever atributos e associações dos tipos
4. Qualificar mecanismos de análise
5. Criar o Modelo de Tipos
6. Criar o Modelo de Frameworks
7. Criar o Modelo de Aplicação do Framework

A adaptação desta atividade, também foi realizada com a adição de artefatos do método Catalysis. O método considera que descrições de mais alto nível de abstração, como especificações e projetos, também podem ser particionadas e reutilizadas, ao contrário da convencional reutilização de código. Esses elementos reutilizáveis de mais alto nível são chamados de *Framework de Modelos*.

Um *Framework de Modelos* possui a estrutura de um pacote genérico, que contém alguns tipos e suas características. Esses tipos e características podem ser definidos por meio de *placeholders,* representados pelos sinais de <>*,* que consistem em definições genéricas a serem instanciadas conforme a aplicação do domínio [D'Souza, 1999].

Quando um *Framework de Modelos* é aplicado, os tipos com *placeholders* são substituídos, para que se possa obter uma versão do *framework* especializado para a aplicação desejada. O artefato disponibilizado a partir dessa aplicação do *Framework de Modelos* é o *Modelo de Aplicação do Framework*. A Figura 2 mostra o *Framework de Modelos e o Modelo de Aplicação do Framework*, onde os tipos genéricos foram substituídos pelos tipos específicos, no caso, *Customer*, *ServiceOrder*, *Employee* e *Task*.

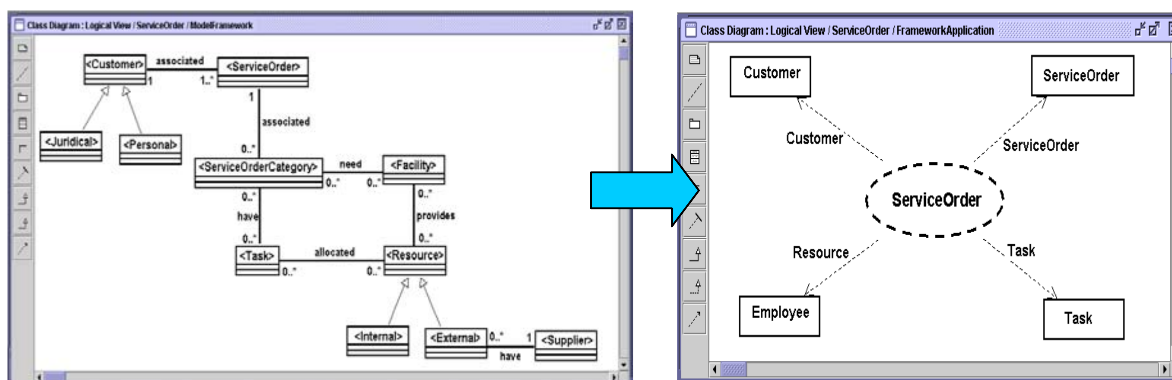

Figura 2. Framework de Modelos e Modelo de Aplicação do Framework

A atividade de análise dos casos de uso tem início com a especificação do *Modelo de Tipos* (com base nas informações dos casos de uso e análise do domínio), identificando atributos e operações dos tipos de objetos, sem se preocupar com a implementação. Ainda neste passo, pode-se utilizar o dicionário de dados, para especificar cada tipo encontrado, e

a *Object Constraint Language* (OCL) [OMG, 2001], para detalhar o comportamento dos objetos, sem ambigüidades.

Uma vez identificados e especificados, os tipos são agrupados em *Framework de Modelos*. *Framework de Modelos* são especificados em um alto nível de abstração, estabelecendo um esquema genérico que pode ser importado, com substituições e extensões, de modo a gerar aplicações específicas [D'Souza, 1999]. Adicionalmente concebido como um *Framework de Modelos*, este se torna um artefato reutilizável da fase de análise. Assim, podem-se especificar esses *frameworks* num alto nível de abstração e oferecer um processo para refiná-los até o nível de um conjunto de componentes interoperáveis [D'Souza, 1999].

Em resumo, os passos realizados pelo engenheiro de caso de uso, compreendem a especificação do:

a) Modelo de Tipos;
b) Framework de Modelos;
c) Modelo de Aplicação do Framework;
d) Modelos de Interações, representados pelos diagramas de colaboração, baseados nos casos de uso.

Uma vez especificados, esses modelos são utilizados na próxima atividade do fluxo de análise e projeto para obter o projeto arquitetural.

4.3 Projeto Arquitetural

A partir dos modelos especificados na atividade anterior, especialmente, o modelo de tipos, o projeto arquitetural inicia-se. O objetivo principal desta atividade é projetar a arquitetura do domínio, refinando os tipos especificados em classes de projeto, com suas interfaces, visando a posterior materialização em componentes de software (`Tipo->Classe->Futuro Componente`). A Tabela 3 apresenta uma visão geral desta atividade.

Tabela 3. Visão Geral da atividade Projeto Arquitetural

Objetivo (s)
➢ Projetar a arquitetura do domínio
Passos
1. Identificar classes de projetos
2. Identificar atributos e relacionamentos
3. Projetar interfaces
4. Criar Modelo de Classes

A adaptação desta atividade consistiu basicamente, do deslocamento do processo de identificação dos subsistemas para uma fase posterior (Projeto dos Componentes). A principal razão para este deslocamento foi devido a limitação da visão de componente do RUP, sendo voltada mais ao nível físico de pacotes de código de programação, baseando-se em componentes UML e diagramas de implantação. Assim, conforme será apresentado na seção 4.5, este processo passou a ser mais coerente e sistemático.

4.4 Projeto das Classes

Nesta atividade, o foco está inserido no projeto de uma particular classe, isto é, quais operações e requisitos não funcionais são importantes e como essas classes colaboram entre si. A Tabela 4 apresenta uma visão geral desta atividade.

Tabela 4. Visão Geral da atividade Projeto das Classes

Objetivo (s)
➢ Projetar internamente as Classes
Passos
1. Identificar suporte a requisitos não funcionais
2. Projetar a estrutura interna das classes
3. Refinar as colaborações entre as classes

A adaptação desta atividade consistiu em explicitar o projeto dos requisitos não funcionais, neste caso, distribuição, baseando-se em estruturas reutilizáveis. Assim, a partir das classes identificadas na atividade anterior aplica-se o padrão *Distributed Adapters Pattern* (DAP) [Alves, 2001], a fim de realizar o suporte a distribuição. A seção seguinte apresenta uma visão geral deste padrão.

4.4.1 Distributed Adapters Pattern (DAP)

O *Distributed Adapters Pattern* (DAP) [Alves, 2001] foi desenvolvido com a proposta de refinar as camadas de distribuição numa arquitetura distribuída. O DAP, atualmente, é uma combinação do *Facade*, *Adapter* e *Factory* design patterns [Gamma, 1995].

O DAP é um padrão que está inserido no contexto de comunicação remota entre dois componentes, oferecendo os seguintes benefícios [Alves, 2001]:

- um componente pode acessar serviços remotos, oferecidos por outros componentes;
- os componentes são independentes de *Application Programming Interfaces* (API) de comunicação;
- as modificações no código dos componentes, para oferecer suporte à comunicação, são minimizadas;
- as mudanças no mecanismo de comunicação se tornam uma tarefa facilitada, minimizando o impacto no código de negócio.

A técnica adotada pelo DAP, para oferecer todas estas funcionalidades mencionadas acima, é introduzir um par de objetos adaptadores, visando conseguir um melhor desacoplamento dos componentes dentro de uma arquitetura distribuída. Os adaptadores, basicamente, encapsulam a API, que é necessária para permitir o acesso distribuído ou remoto para objetos de negócio. Deste modo, a camada de negócio de uma aplicação torna-se independente em relação à camada de distribuição e as mudanças nesta camada não causam impactos.

No padrão, existem dois tipos de adaptadores: **Source Adapter** e **Target Adapter**. Em uma típica interação, um objeto que possua a interface do usuário em uma máquina solicita serviços de um *source adapter* localizado na mesma máquina. O *source adapter*, em seguida, solicita os serviços de um correspondente *target adapter*, residido em uma

máquina remota. Finalmente, *o target adapter* solicita serviços de um objeto *Facade*, co-localizado com o *target adapter*. A Figura 3 [Alves, 2001a] mostra este exemplo.

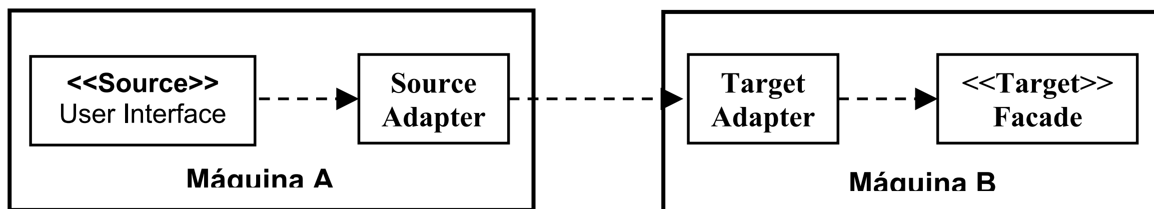

Figura 3. Um exemplo do DAP.

4.4.2 Aplicando o DAP

A partir do modelo de classes, são identificadas as classes que serão distribuídas. A Figura 4 mostra o projeto das classes após a aplicação do DAP.

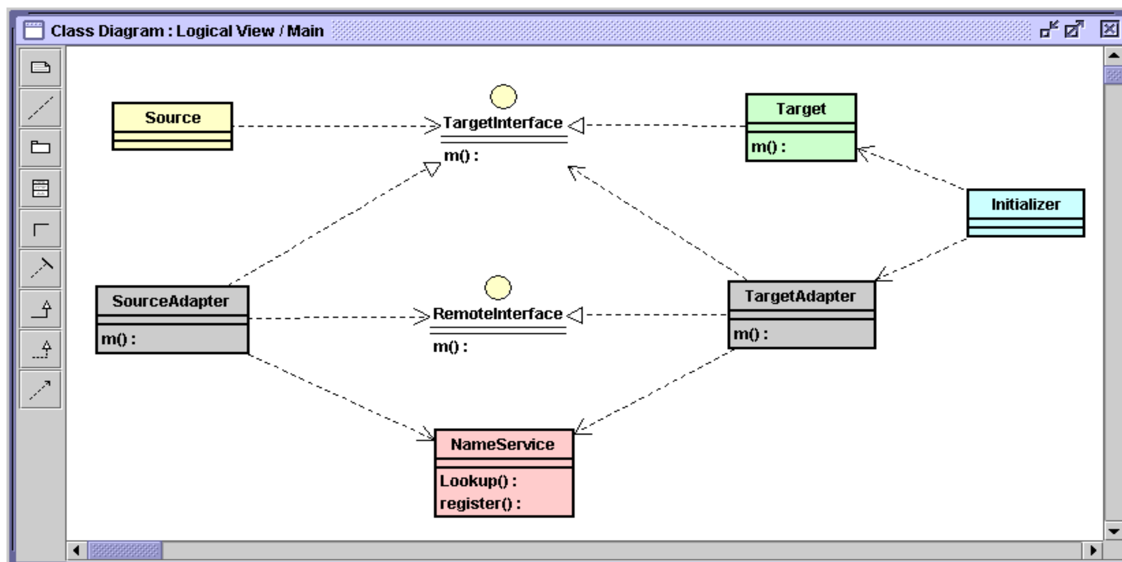

Figura 4. Estrutura do Padrão DAP.

As classes *Source* e *Target* abstraem os componentes de negócio. A interface *TargetInterface* abstrai o comportamento da classe *Target* em um cenário distribuído. De qualquer modo, esta interface e as classes, *Source* e *Target*, não possuem código de comunicação. Estes três elementos constituem uma camada independente de distribuição.

Os principais elementos do padrão de distribuição são *SourceAdapter* e *TargetAdapter*. Estes elementos estão ligados a uma API específica de distribuição e encapsulam os detalhes de comunicação. *SourceAdapter* é um adaptador que isola a classe *Source* do código de distribuição. Este reside na mesma máquina que o *Source* e trabalha como um *proxy* para o *TargetAdapter*. Este último reside em outra máquina, isolando a classe *Target* do código de distribuição. Como *SourceAdapter* e *TargetAdapter*, usualmente, residem em máquinas diferentes, e não interagem diretamente: *TargetAdapter* implementa *RemoteInterface*, da qual *SourceAdapter* depende [Alves, 2001].

A classe *NameService* possui operações para registrar e efetuar *lookup* em um objeto remoto; ambos adaptadores usam esta classe, a qual representa um serviço genérico de nomes e é comum em muitas plataformas de distribuição. A classe *Initializer* reside na mesma máquina que as classes *Target* e *TargetAdapter*, e é responsável por criar objetos *TargetAdapter* e *Target*.

Uma vez projetado as classes, o projeto dos componentes inicia-se.

4.5 Projeto dos Componentes

Nesta atividade, projetam-se internamente os componentes, juntamente com outros requisitos não funcionais, destacando-se: tolerância a falhas, *caching* e persistência. A Tabela 5 apresenta uma visão geral desta atividade.

Tabela 5. Visão Geral da atividade Projeto dos Componentes

Objetivo (s)
➢ Projetar internamente os Componentes
Passos
1. Projetar o Modelo de Componentes
2. Projetar outros requisitos não funcionais
3. Refinar as colaborações entre as classes

A adaptação desta atividade consistiu em definir uma sistemática para o mapeamento direto entre classes e componentes, além de definir como outros requisitos não funcionais são projetados. Esta atividade, no modelo atual do RUP, encontrava-se bastante vaga, em virtude do processo não ter, inicialmente, a orientação à reutilização.

Assim, com base no modelo de classes com a estrutura do padrão DAP, o modelo de componentes é projetado. As classes definidas anteriormente (adaptadores, classes de serviços distribuídos e classes de negócio) são mapeadas para os componentes correspondentes, conforme mostra a Figura 5. Os componentes *Source* e *CustomerTarget* abstraem as regras de negócio do domínio do problema. A interface *ICustomer* abstrai o comportamento do componente *CustomerTarget* num cenário distribuído. Tanto esta interface como os componentes, *Source* e *CustomerTarget*, não possuem código de distribuição. Esses três elementos constituem uma camada independente de distribuição, conforme é mostrado na parte superior da Figura. Os demais componentes estão conectados a uma específica API de distribuição e encapsulam os detalhes de comunicação.

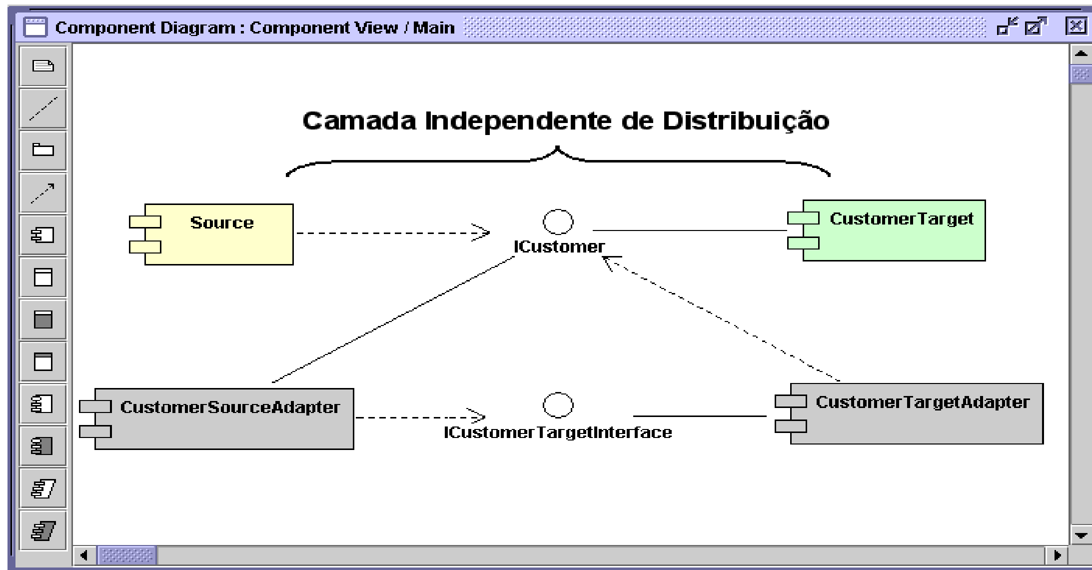

Figura 5. Projeto do Modelo de Componentes

Uma vez projetados os componentes, podem-se projetar outros requisitos não funcionais.

4.5.1 Outros requisitos não funcionais

Os adaptadores, anteriormente apresentados, tratam com detalhes básicos de distribuição e ocultam estes detalhes do código de negócio e de interface. Estes podem, também, adicionar outros requisitos não funcionais, mantendo a integridade da camada independente de distribuição. Nesta seção, será mostrado como os adaptadores podem realizar a adição destes requisitos, que podem ser úteis para projetar componentes distribuídos [Alves, 2001a].

i. Tolerância a Falhas. O componente *SourceAdapter* não possui mecanismo de tolerância a falhas. Assim, se ocorrer um erro de comunicação ou se o servidor estiver indisponível, este, simplesmente, gera uma exceção de comunicação. No entanto, este componente pode, adicionalmente, implementar mecanismos de tolerância a falhas.

Se o componente *SourceAdapter* recebe uma exceção remota quando interage com o *TargetAdapter*, ele pode implementar a política de tentar contactar o *TargetAdapter* novamente durante um certo período de tempo, ou tentar contactar outro *TargetAdapter*. Esta política pode ser implementada pelo *SourceAdapter* de maneira transparente à camada independente de distribuição.

***ii. Caching*.** Algumas operações podem retornar uma considerável quantidade de dados, entretanto somente parte deles será útil em determinado momento. Enviar todos os dados de uma só vez para o cliente pode não ser desejável, uma vez que pode causar um impacto negativo de performance na rede. Uma possível solução é enviar um *cache* com parte dos dados necessários e transferir mais dados à medida em que uma falha ocorra.

Um componente *SourceAdapter* pode implementar este comportamento de *caching*. Quando uma operação de consulta retorna muitas entradas, parte destas são utilizadas para

inicializar um *SourceAdapter*. Um cliente, por exemplo, um *Servlet*, pode obter as entradas deste adaptador.

Quando uma falha ocorre num *SourceAdapter*, este contacta o *TargetAdapter*, a fim de obter mais entradas. Este comportamento de *caching* é implementado pelo *SourceAdapter* de forma totalmente transparente ao cliente.

iii. Persistência dos Dados. A fim de facilitar a persistência dos dados, pode-se reutilizar os componentes do *Framework Persistence* [Sobral, 2001], que utiliza os princípios dos padrões apresentados em [Yoder, 1998], para mapeamento de objetos de negócio para banco de dados relacional.

Esse *framework* é composto, basicamente, de quatro componentes específicos e de outros componentes pertencentes à biblioteca da linguagem *Java*. A Figura 6 mostra os componentes específicos e suas dependências. O componente *ConnectionPool*, através de sua interface *IConnectionPool*, realiza a gerência e conexão com o banco de dados utilizado na aplicação. O componente *DriversUtil*, com base em definições *eXtensible Markup Language* (XML) [W3C, 2001], contém as informações dos drivers de banco de dados suportados, disponibilizadas através de sua interface *IDriverUtil*. O componente *TableManager* gerencia o mapeamento de um objeto para tabelas do banco de dados, disponibilizando seus métodos pela interface *ITableManager*. O componente persistente da estrutura *FacadePersistent*, através de sua interface *IPersistentObject*, disponibiliza os valores que devem ser adicionados ao banco de dados, passando parâmetros ao componente *TableManager*.

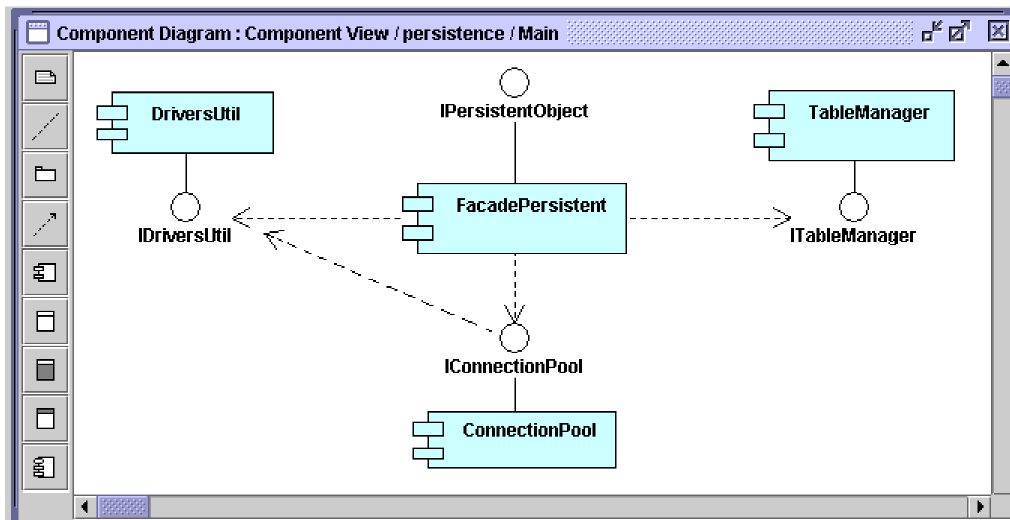

Figura 6. Framework Persistence

Após o projeto dos componentes do domínio do problema, passa-se para a atividade de projeto das aplicações, que corresponde ao desenvolvimento com reutilização. A extensão proposta neste trabalho, não considerou o projeto de concorrência, porém, este requisito não funcional também poderia ser adicionado nesta atividade.

4.6 Projeto das Aplicações

Após a disponibilização dos componentes para reutilização, deve existir uma maneira sistemática e progressiva de realizar o desenvolvimento das aplicações reutilizando estes artefatos. O modelo atual do RUP não define este processo, deixando esta tarefa totalmente a cargo do desenvolvedor. Deste modo, foi definida a inclusão desta nova atividade, Projeto das Aplicações, juntamente com a adição do perfil de um novo trabalhador: o engenheiro de aplicações. A Tabela 6 apresenta uma visão geral desta atividade.

Tabela 6. Visão geral da atividade Projeto das Aplicações

Objetivo (s)
➢ Projetar as Aplicações reutilizando os componentes
Passos
1. Identificar componentes
2. Consultar componentes
3. Selecionar componentes
4. Compor aplicações

As seções seguintes apresentam uma visão detalhada de cada passo.

4.6.1 Identificar Componentes

O primeiro passo durante o projeto das aplicações, consiste em identificar os componentes necessários com base nos requisitos das aplicações. Inicialmente, num alto nível de abstração, o engenheiro de aplicações define a decomposição dos requisitos em componentes. Mesmo com a utilização de padrões no projeto dos componentes e separando as funcionalidades em componentes específicos (distribuição, *caching*, persistência), o trabalho de decomposição das aplicações em componentes é extremamente complexo, e, muitas vezes, este processo pode não ser realizado diretamente, sendo necessário a decomposição em componentes de granularidade ainda menor, de modo a gerenciar esta complexidade.

Assim, técnicas efetivas de engenharia de requisitos podem ser utilizadas, a fim de auxiliar o processo de identificação dos componentes necessários à aplicação.

4.6.2 Consultar Componentes

O passo de consulta dos componentes é realizado de modo a buscar componentes disponíveis que possam ser utilizados no projeto da aplicação. Este passo, para tornar-se mais efetivo, deve ser suportado por um repositório [Sametinger, 1997] que contém uma coleção de componentes disponíveis para reutilização podendo ser, por exemplo, agrupados por domínios de aplicações.

Diversas técnicas podem ser utilizadas para auxiliar na busca por componentes, desde mecanismos baseados em palavras chaves ou introspecção computacional, como mecanismos mais complexos baseados em ontologias [Braga, 2000] e agentes de software [Ye, 2002].

4.6.3 Selecionar Componentes

Uma consulta por componentes, normalmente, resulta numa lista de componentes disponíveis. Conseqüentemente, esta lista tem que ser analisada de modo a permitir uma reutilização efetiva por parte das aplicações. Dependendo dos componentes disponíveis, um conjunto de diferentes ações pode ser necessário:

i. Se o conjunto de componentes, resultante da consulta, contém um componente que satisfaça os requisitos da aplicação, a reutilização "perfeita" é realizada. Porém, esta taxa de sucesso imediato não é muito provável, mesmo no contexto de linhas de produto.

ii. O caso mais provável é o componente não atender todos os requisitos necessários para o projeto da aplicação. Neste caso, um processo de adaptação é necessário.

iii. Por fim, caso nenhum componente possa ser encontrado, a atividade de projeto dos componentes é então realizada, de modo a atender os requisitos da aplicação e tornar este artefato passível de uma posterior reutilização.

4.6.4 Compor Aplicações

O último passo do projeto das aplicações consiste na sua composição. Após a identificação e seleção dos componentes necessários, o engenheiro da aplicação projeta as dependências e estabelece os contrato entre os componentes.

Por fim, são realizados os testes e os processos de documentação e implantação, que estão fora do escopo deste artigo.

5. Trabalhos Relacionados

Ambler [2002] apresenta uma proposta de extensão do RUP através da adição de um novo fluxo: Gerenciamento de Infra-estrutura (*Infrastructure Management*). O autor justifica que a fim de se obter os benefícios da reutilização, deve-se contar com uma estratégia de gerenciamento de reutilização efetiva, definida através de um conjunto de sete princípios de reutilização a serem incorporados ao utilizar o processo.

Moraes [2002] descreve o *framework* ARCADE (*ARChitecture-based Analisys and DEsign*), baseado em arquitetura de software, para extensão do fluxo de análise e projeto do RUP visando à reutilização. O *framework* estende o RUP com a adição de duas novas atividades (Projetar conectores e Selecionar componentes), e adapta as já existentes utilizando desenvolvimento baseado em componentes, padrões de projeto e *middleware*.

A principal diferença da nossa proposta, em relação aos trabalhos mencionados, está na clara definição das atividades para realizar tanto o desenvolvimento para reutilização, quanto com reutilização. Além de oferecer um suporte sistemático ao projeto dos requisitos não funcionais e considerar artefatos reutilizáveis não somente como código.

6. Conclusão e Trabalhos Futuros

Neste artigo, apresentamos uma proposta de extensão do RUP com suporte a desenvolvimento baseado em componentes, que integra princípios e artefatos do método Catalysis para apoiar tanto o desenvolvimento, como a reutilização de componentes de um domínio do problema.

A extensão proposta viabiliza a analise e o projeto de aplicações e componentes de uma forma sistemática e incremental, contribuindo como uma tentativa de considerar a reutilização nos projetos que utilizam o RUP.

Como trabalhos futuros, pretendemos desenvolver um estudo de caso a fim de refinar o processo de extensão. Além disso, dois aspectos não considerados neste artigo merecem uma posterior investigação: o impacto causado nos fluxos de implementação e teste devido à extensão proposta, e, sobretudo, a análise do aspecto dinâmico do RUP.

Referências Bibliográficas

[Alves, 2001] Alves, V., Borba, P. Distributed Adapters Pattern (DAP): A Design Pattern for Object-Oriented Distributed Applications**.** In *SugarLoafPlop'2001, The First Latin American Conference on Pattern Languages of Programming.*

[Ambler, 2002] Ambler, S. 2002. *Strategic Reuse Management and the Rational Unified Process (RUP)*. Disponível em http://flashline.com. Consultado em 05/06/2003

[Boertin, 2001] Boertin N, Steen M, Jonkers H. Evaluation of Component-Based Development Methods. In *EMMSAD'2001, Sixth CAiSE/IFIP8.1 International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design*.

[Booch, 1994] Booch, G. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, 1994.

[Braga, 2000] Braga, R. *Busca e Recuperação de Componentes em Ambientes de Reutilização de Software*. Dissertação de Doutorado, Universidade Federal do Rio de Janeiro.

[D'Souza, 1999] D'Souza DF, Wills AC. *Objects, Components, and Frameworks with UML, The Catalysis Approach*, Addison-Wesley. USA, 1999.

[Frakes, 1994] Frakes, W., B., Isoda, S. Success Factors of Systematic Software Reuse. *IEEE Software*, Sep, 1994.

[Gamma, 1995] Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Jacobson, 2001] Jacobson, I.; et. al. *The Unified Software Development Process*.

[Kang, 1990] Kang, K. et. al. Feature-Oriented Domain Analysis (FODA) – Feasibiity Study, *SEI Technical Report*.

[Moraes, 2002]Moraes, M. *Um Framework de Análise e Projeto baseado em Arquitetura de Software***.** Dissertação de Mestrado, Universidade Federal de Pernambuco.

[OMG, 2001] *Object Management Group*. Unified Modeling Language 1.4 specification. Disponível site Object Management Group, URL: http://www.omg.org/technology/documents/formal/uml.htm. Consultado em 01/12/2002.

[Rational, 1999] Rational. *Object-Oriented Analysis and Design using the UML*, Student Manual, Vol. I. Rational University, 1999.

[Rational, 1999a] Rational. *Object-Oriented Analysis and Design using the UML*, Student Manual, Vol. II. Rational University, 1999.

[Rumbaugh, 1991] Rumbaugh, R. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[Sametinger, 1997] Sametinger, J. *Software Engineering with Reusable Components*. Springer-Verlag, 1997.

[Simos, 1996] Simos, M. Organization Domain Modeling (ODM): Domain Engineering as a Co-Methodology to Object-Oriented Techniques**.** *Fusion Newsletter*, v.4 1996, Hewlett-Packard Laboratories, pp 13-16.

[Sobral, 2001] Sobral, D. *Framework para Comércio Eletrônico, via Internet Móvel, Mediado por Agentes de Software*. Dissertação de Mestrado, Universidade Federal de São Carlos.

[Szyperski, 1998] Szyperski C. *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley. USA, 1998.

[Werner, 2000] Werner, C., M., L., Braga, R., M. Desenvolvimento Baseado em Componentes, XIV Simpósio Brasileiro de Engenharia de Software, *Minicursos e Tutoriais*, João Pessoa/PB, 2000.

[W3C, 2001] *W3C - World Wide Web Consortium*. Extensible Markup Language (XML) 1.0 Second Edition. Disponível site W3C - World Wide Web Consortium, URL: http://www.w3.org/TR/2000/REC-xml-2000-10-06. Consultado em 10/07/2001.

[Ye, 2002] Ye, Y., Fischer, G. Supporting Reuse by Delivering Task-Relevant and Personalized Information, In *24th International Conference on Software Engineering. Orlando*, USA, 2002.

[Yoder, 1998] Yoder, J., W., Johnson, R., E., Wilson, Q., D. Connecting Business Objects to Relational Databases**,** In *Pattern Languages of Programs (PLoP)*, Monticello, Illinois, USA, 1998.