GERAÇÃO DE COMPONENTES DE NEGÓCIO A PARTIR DE MODELOS DE ANÁLISE

Hugo Vidal Teixeira

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof. Cláudia Maria Lima Werner, D.Sc.

Prof. Regina Maciel Braga Villela, D.Sc.

Prof. Guilherme Horta Travassos, D.Sc.

Prof. Antônio Francisco do Prado, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
MARÇO DE 2003

TEIXEIRA, HUGO VIDAL

Geração de Componentes de Negócio a Partir de Modelos de Análise [Rio de Janeiro] 2003

XI, 131 p., 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2003)

Tese – Universidade Federal do Rio de Janeiro, COPPE

1. Desenvolvimento baseado em componentes
2. Arquitetura de componentes
3. Modelagem de negócio

I. COPPE/UFRJ II. Título (série)

A Deus.

AGRADECIMENTOS

A Deus, simplesmente por tudo.

À minha família, pelo incentivo e carinhos dedicados a mim.

À Renata, minha noiva, pela força, amor e carinhos inestimáveis que me sustentaram e mantiveram meus pensamentos alegres e otimistas.

À prof. Cláudia Werner, minha orientadora, que desde o início corrigia meus erros e me ensinava a andar, principalmente, quando eu já achava que podia voar. Pelos elogios, carinhos e críticas construtivas que hoje me fazem uma pessoa melhor.

À prof. Regina Braga, minha co-orientadora, que me mostrou, assim que ingressei no mestrado, a área de desenvolvimento baseado em componentes, a qual me mantive até o fim. Sem ela, tudo teria sido mais difícil, já que sua dedicação e compreensão contribuíram com um toque especial nas minhas atitudes.

Ao professor Antônio Prado, por ter aceitado participar desta banca e por ter demonstrado muita amizade nos encontros ocorridos em congressos e viagens.

Ao professor Guilherme Travassos, por ter aceitado participar desta banca e, com sua amizade, mostrou-se muito interessado e atencioso comigo.

Aos professores da COPPE, que possuem a competência de manter os cursos da pós-graduação em um nível tão elevado e reconhecido.

Aos amigos do projeto *Odyssey*, em especial ao Alexandre Corrêa, Alexandre Dantas, Aline, Gustavo Veronese, Hamilton, José Ricardo, Leonardo Murta, Marcelo Costa, Márcio Barros, Marcos Lopes, Marcos Mangan, Marcos Kalinowsky, Robson Pinheiro, Sascha Kalinowsky e Sômulo Mafra, pela ajuda e amizade.

À CAPES, pelo apoio financeiro ao desenvolvimento deste trabalho.

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

GERAÇÃO DE COMPONENTES DE NEGÓCIO A PARTIR DE MODELOS DE ANÁLISE

Hugo Vidal Teixeira

Março / 2003

Orientadores: Cláudia Maria Lima Werner
Regina Maciel Braga Villela

Programa: Engenharia de Sistemas e Computação

A arquitetura de componentes representa um objetivo desafiador para o desenvolvimento baseado em componentes (DBC). Apesar das diversas abordagens que tratam deste assunto, muitas dificuldades ainda surgem quando projetamos e montamos componentes visando obter uma solução de software integrada. O problema que focamos neste trabalho é a ausência de regras que poderiam se apoiar em modelos de análise para guiar o arquiteto durante a especificação da arquitetura de uma aplicação. Em outras palavras, essa falta de apoio resulta em uma série de atividades muito trabalhosas e sem qualquer automação.

Para aumentar o nível de abstração das tarefas de especificação da arquitetura e reduzir o esforço de projeto, esse trabalho apresenta uma proposta de geração de componentes de negócio a partir de modelos de análise. Para alcançar esse objetivo, algumas regras e procedimentos foram pesquisados e evoluídos, guiando o arquiteto durante o processo de desenvolvimento. Como os atributos de qualidade interferem nas decisões relacionadas com a arquitetura, um trabalho recente desenvolvido na COPPE/UFRJ foi adaptado para apoiar a avaliação de diferentes estilos arquiteturais e tecnologias.

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

GENERATION OF BUSINESS COMPONENTS FROM ANALYSIS MODELS

Hugo Vidal Teixeira

March / 2003

Advisors: Cláudia Maria Lima Werner
Regina Maciel Braga Villela

Department: Computer and Systems Engineering

Software component architecture represents a challenging goal in component-based development (CBD). Despite the several approaches regarding this subject, difficulties still arise when designing and assembling components to provide an integrated software solution. The problem we focus in this work is the absence of rules that could use analysis models to guide the architect through the specification of an application architecture. In other words, this lack of support results in a series of laborious and non-automated activities.

In order to increase the abstraction level of the architecture specification tasks and reduce designing efforts, this work presents a proposal aimed at generating business components from analysis models. To achieve this goal, some rules and procedures have been researched and evolved, guiding the software architect during the development process. Since quality attributes interfere in architecture-related decisions, a recent work developed at COPPE/UFRJ has been adapted to support the evaluation of architectural styles and technologies.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABELAS

Capítulo 1 – Introdução

1.1 – Motivação

Quando observamos como o mundo da computação evoluiu desde o seu nascimento em meados do século passado, ficamos surpreendidos ao descobrir que aquelas gigantescas máquinas movidas à válvula se transformaram em minúsculos e poderosos computadores que hoje dominam nossas vidas. Nesse período de tempo, passamos dos ambientes monolíticos típicos dos *mainframes* para um mundo orientado a objetos, capaz de distribuir sistemas e bancos de dados, além de efetuar seguras transações à distância.

Mesmo após essa evolução tecnológica e décadas de intensa pesquisa, a produção de software, respeitando os preceitos de qualidade e produtividade, ainda permanece uma promessa insatisfeita. Enquanto muito progresso foi alcançado quanto ao entendimento da mecânica de construção de sistemas a partir de modelos e especificações, a crescente demanda por aplicações mais complexas, maiores e mais baratas ainda representa um problema a ser solucionado (PRESSMAN, 2000) (SAMETINGER, 1997).

Dentre as diversas linhas de pesquisa inerentes à engenharia de software (PRESSMAN, 2000), o **desenvolvimento baseado em componentes (DBC)** tem assumido um papel promissor no combate a esses obstáculos. Sua característica mais marcante tem sido elevar o nível de abstração das atividades envolvidas no desenvolvimento de software, visando obter uma solução para um problema a partir de partes reutilizáveis que possam ser conectadas umas às outras. Essa visão tem se mostrado muito importante por focar os serviços prestados pelos componentes separadamente de seus detalhes de implementação, maximizando a reutilização e evitando o uso de informações menos significativas durante a montagem da arquitetura do software.

Muitas das idéias trazidas pelo desenvolvimento baseado em componentes apresentam conceitos e princípios que se estendem além da orientação a objetos. Esse movimento fez com que novas abordagens fossem desenvolvidas especialmente para essa finalidade, empregando os componentes como o aspecto-chave da arquitetura do software. As abordagens mais conhecidas atualmente são: *UML Components*

(CHEESMAN e DANIELS, 2001), *Catalysis* (D'SOUZA e WILLS, 1999) e *KobrA* (ATKINSON *et al.*, 2000).

No entanto, desde o seu surgimento há aproximadamente uma década, o desenvolvimento baseado em componentes ainda permanece imaturo perante as necessidades dos desenvolvedores de software. Seus principais problemas e desafios ainda impedem que projetos de software sejam realizados com sucesso, ou que consigam atingir os ideais teóricos descritos na literatura (BROWN, 2000) (HERZUM e SIMS, 2000) (WALLNAU *et al.*, 2001a), resultando em perdas de tempo, custos altos e re-trabalho.

Estas dificuldades existem porque as atuais abordagens de DBC procuram enfatizar muito mais suas notações e diagramas do que como, efetivamente, podemos especificar e montar os componentes a partir dos elementos sintáticos modelados na análise do problema. Essa falta de suporte ao arquiteto representa o diferencial que precisa ser preenchido para que as ferramentas utilizadas no desenvolvimento não sejam adotadas apenas como diagramadores inertes.

Além disso, a construção de componentes exige muito trabalho de documentação, sendo necessário criar diversos diagramas diferentes para cada componente especificado. Esta atividade torna-se ainda mais complexa quando precisamos obedecer às restrições (pré e pós-condições) das operações envolvidas.

1.2 – Objetivo

Tendo em vista os problemas e necessidades que ainda impedem que o desenvolvimento baseado em componentes seja adotado em larga escala pelas organizações, buscamos desenvolver neste trabalho um mecanismo de apoio à construção da arquitetura de componentes a partir dos modelos de análise de uma aplicação.

É importante destacar que, na visão do desenvolvimento baseado em componentes, a arquitetura é formada por *especificações de componentes* que não detalham como elas são implementadas por dentro. O foco de cada especificação permanece voltado para um ou mais **serviços**, que são representados pelas interfaces oferecidas por ela. Dentro deste contexto, a geração proposta neste trabalho só está concentrada nesses serviços de negócio (componentes), especificando suas **interfaces** (operações e parâmetros) e **responsabilidades** na arquitetura.

De uma forma geral, o objetivo principal deste trabalho é identificar regras que tornem a elaboração da arquitetura uma atividade menos complexa e um pouco mais automatizada do que se tem visto nas atuais abordagens de DBC, reduzindo o trabalho repetitivo dos projetistas no que diz respeito à especificação de todos os componentes de forma manual, com interfaces carregadas de operações e restrições.

Em qualquer projeto de DBC, é imprescindível que todos os detalhes de funcionamento e conexão dos componentes estejam bem documentados em suas especificações, visto que são eles que tornarão possível o entendimento dos componentes e o fluxo da informação entre suas interfaces. Por outro lado, se para cada software construído o projetista tiver que re-elaborar estas informações básicas sem qualquer apoio tecnológico, muito trabalho árduo prevalecerá durante a montagem da arquitetura.

Para permitirmos que este trabalho possa ser utilizado em muitos projetos de DBC, estudamos o conteúdo das atuais abordagens sobre esse assunto, procurando destacar o que há em comum entre elas. Isso possibilita a extração de idéias que são mais consensuais e independentes de abordagem, provendo técnicas que reduzem a complexidade encontrada no desenvolvimento e especificação de componentes de negócio de uma aplicação. Como possíveis conseqüências, essa independência tende a trazer uma maior aceitação da proposta e ajuda a aumentar suas chances de ser aproveitada por diferentes ferramentas de apoio ao desenvolvimento de software.

O ambiente de desenvolvimento de software utilizado na implementação desse trabalho é o Ambiente *Odyssey* (WERNER *et al.*, 2000), que visa prover uma infra-estrutura de suporte à reutilização baseada em modelos de domínio. A reutilização de software no ambiente *Odyssey* ocorre através dos processos de *Engenharia de Domínio*, cujo objetivo é construir artefatos reutilizáveis voltados para problemas de conhecimento específico, e de *Engenharia de Aplicação*, que tem o objetivo de desenvolver aplicações reutilizando esses artefatos.

Assim como muitos outros ambientes de desenvolvimento de software, o ambiente *Odyssey* não foi construído, originalmente, com todas as características necessárias ao desenvolvimento baseado em componentes. Portanto, discutimos as *mudanças* que precisaram ser feitas nesse ambiente, procurando reduzir, ao máximo, o impacto causado tanto na abordagem proposta, quanto no ambiente de desenvolvimento.

1.3 – Organização

Este trabalho está organizado em seis capítulos. Neste **primeiro** capítulo, de introdução, exibimos a *motivação*, o *objetivo* e a *organização* do trabalho.

No **segundo** capítulo apresentamos os *conceitos* e *princípios* básicos que fazem da criação e reutilização de componentes uma atividade vantajosa para o desenvolvimento de software.

No **terceiro** capítulo aprofundamos nosso conhecimento sobre o desenvolvimento baseado em componentes, estudando seu *processo*, seus modelos de análise e as *abordagens* de DBC que serviram como base para o desenvolvimento deste trabalho.

No **quarto** capítulo exibimos a abordagem proposta, que consiste na *geração* de componentes de negócio utilizando os modelos de análise do software. Além disso, apresentamos um mecanismo de apoio à seleção de estilos arquiteturais e tecnologias de componente, aproveitando o trabalho anteriormente desenvolvido por XAVIER (2001) na COPPE/UFRJ.

No **quinto** capítulo descrevemos a implementação feita no ambiente *Odyssey*, ilustrando as fases e opções disponíveis pelo mecanismo de geração de componentes. Além disso, algumas discussões são levantadas sobre as modificações que precisaram ser feitas nesse ambiente, devido à introdução de alguns conceitos trazidos pela proposta.

No **sexto** capítulo são apresentadas as contribuições e limitações deste trabalho, assim como listados os trabalhos futuros.

Capítulo 2 – Componentes: Princípios e Conceitos

O crescente uso de computadores pessoais, impulsionados pelo advento da Internet, fizeram com que o uso da tecnologia alcançasse uma maior parcela da população mundial, criando novos tipos de mercados e usuários de software (AOYAMA, 1998). Além de telefones celulares que acessam a Internet, *palmtops* que executam aplicativos em Java e geladeiras que se comunicam com supermercados, os sistemas mais modernos, de uma forma geral, estão ficando mais complexos e difíceis de se gerenciar, resultando em um alto custo de desenvolvimento, baixa produtividade e qualidade duvidosa (XIA *et al.*, 2000) (BERTOLINO *et al.*, 2002).

Como reflexo desses problemas, as empresas estão sendo obrigadas a repensar seus processos e metodologias de desenvolvimento de software para atender a necessidade de seus usuários e conquistar novas fatias de mercado. A redução dos custos, a rapidez de entrega e o elevado grau de qualidade são objetivos muito importantes que o desenvolvimento com metodologias tradicionais ainda não conseguiu alcançar de forma satisfatória (AOYAMA, 1998).

Durante muitos anos, a *reutilização de software* teve uma importante participação no combate a essas dificuldades, oferecendo meios de recuperar e aplicar artefatos de antigos projetos em novos desenvolvimentos (SAMETINGER, 1997). Entretanto, os artefatos reutilizados encontravam-se em diferentes níveis de abstração (e.g., *diagramas*, *algoritmos*, *documentos*, etc.) e não possuíam uma forma predefinida que os tornassem mais propícios para a tarefa de reutilização, exigindo muito trabalho para entendê-los e modificá-los segundo as novas necessidades (HAN, 1998).

Por outro lado, a orientação a objetos complementou o quadro de reutilização de software com um conjunto de idéias que rapidamente se consolidaram na indústria de software. Dentre elas, podemos citar o importante conceito de *herança* e outras interessantes propostas, como os *padrões de projeto* (GAMMA *et al.*, 1995) e os *frameworks* (FAYAD e SCHMIDT, 1997).

Apesar das vantagens trazidas por esse conjunto de idéias, a orientação a objetos, por si só, não é suficiente para proporcionar melhorias expressivas na qualidade, prazos e custos dos desenvolvimentos (D'SOUZA e WILLS, 1999). Os verdadeiros benefícios da orientação a objetos só são alcançados através de um bom

gerenciamento do processo de desenvolvimento, fazendo com que suas técnicas, linguagens, métodos e ferramentas só sejam eficientes quando utilizados da forma correta.

Em uma visão mais prática, a crescente demanda por software e os problemas conhecidos do desenvolvimento tradicional exigiram que a reutilização de software sofresse mudanças. Em particular, um passo muito importante precisou ser dado nesse sentido: *reutilizar componentes sem que fosse preciso ler seus códigos* (BROWN, 2000) (D'SOUZA e WILLS, 1999) (HAN, 1998). Dessa forma, o ***Desenvolvimento Baseado em Componentes (DBC)*** surgiu como uma estratégia de montagem de software visando obter resultados satisfatórios na indústria, tentando vender a imagem de que *software* pode ser construído como *hardware*, i.e., conectando componentes por interfaces bem definidas (CHEESMAN e DANIELS, 2001) (VERYARD, 2001) (HALL, 2000).

O novo paradigma trazido pelo desenvolvimento baseado em componentes procura, entre outros objetivos, estabelecer *mercados* de componentes pré-fabricados que possam ser vendidos comercialmente (WALLNAU *et al.*, 2001a). Estes componentes comerciais - também chamados de *Commercial off-the-shelf Components (COTS)* - representam a mais nova tentativa de reduzir drasticamente o tempo de desenvolvimento, podendo ser empregados em diferentes aplicações e reutilizados por diferentes desenvolvedores (HÖRNSTEIN e EDLER, 2002).

Além do objetivo comercial, cada organização pode manter o seu próprio repositório interno de componentes, o qual tende a se tornar cada vez mais abrangente com o passar dos anos. Inicialmente, a montagem dos sistemas pode se tornar lenta devido à dificuldade de se achar e desenvolver cada componente. Porém, o tempo gasto nestas atividades tende a ser reduzido e recompensado conforme a organização amadurece com seus projetos (CRNKOVIC e LARSSON, 2001).

O desenvolvimento baseado em componentes possui muitos detalhes e conceitos que precisam ser estudados separadamente. Portanto, analisaremos neste capítulo os princípios fundamentais e definições relacionadas aos componentes, deixando os aspectos mais metodológicos para o capítulo seguinte. As seções que veremos aqui são:

- **Seção 2.1**: Apresenta o conceito de *componente de software* adotado em DBC, descrevendo suas características e princípios básicos;
- **Seção 2.2**: Apresenta o conceito de *arquitetura de componentes*, suas partes, divisões e estruturas internas.

2.1 – Componentes de Software

A busca por um entendimento comum sobre os conceitos do desenvolvimento baseado em componentes tem sido um forte desafio para a indústria de software. Apesar de existirem diversos modelos e tecnologias disponíveis (e.g. COM+ (KIRTLAND, 1997), EJB (THOMAS, 1998), etc.), pouco sucesso foi obtido quanto a uma padronização ou consenso sobre o conceito básico de componente. Sob o foco deste trabalho, destacaremos nesta seção o que entendemos por componente de software, observando tanto as definições apresentadas por alguns autores quanto as suas características e aspectos fundamentais que possibilitam o desenvolvimento baseado em componentes ser aplicado em projetos de sistemas.

A reutilização de artefatos simples de software (como funções, módulos, classes, etc.) pode ser considerada como o primeiro passo dado na evolução das abordagens de desenvolvimento baseado em componentes (SAMETINGER, 1997). Apesar deste enfoque, uma forma mais apropriada vem sendo desenvolvida para aprimorar as técnicas e abordagens utilizadas, refinando gradativamente as características consideradas imprescindíveis à correta e qualitativa reutilização de componentes. Estas técnicas, também chamadas de *plug & play* (VERYARD, 2001) (BROWN, 2000), procuram manipular componentes de software da mesma maneira que componentes eletrônicos são conectados entre si (HALL, 2000). Para isso, algumas regras básicas de conexão precisam ser obedecidas. Pelo fato de ser muito difícil alcançar uma definição em poucas palavras do que realmente constituem estes componentes, muitos autores procuram analisar separadamente cada um dos seus princípios fundamentais:

1. Um componente é uma parte independente e substituível de um sistema que realiza um serviço na arquitetura (XIA *et al.*, 2000) (SZYPERSKI, 1998);
2. Todo componente implementa uma ou mais interfaces que são separadas da sua implementação interna (PAGE-JONES, 1999) (BROWN, 2000) (WALLNAU *et al.*, 2001a) (CHEESMAN e DANIELS, 2001) (SZYPERSKI, 1998);
3. Todo componente implementa uma ou mais interfaces que são definidas de forma *contratual*: cada operação é descrita em termos de sua *assinatura* (tipos de parâmetros de entrada e saída) e um conjunto de restrições que descrevem o protocolo de comunicação com outros componentes (PAGE-

JONES, 1999) (CHEESMAN e DANIELS, 2001) (BACHMAN *et al.*, 2000) (XIA *et al.*, 2000) (SZYPERSKI, 1998);

4. Nenhum componente é instanciado em múltiplas cópias, onde cada cópia possui o seu próprio estado persistente. Em outras palavras, os componentes não possuem variáveis internas que mudam de valor a cada chamada de operação. Por não possuírem esse estado interno, uma única instância é suficiente para tratar as requisições da arquitetura (PAGE-JONES, 1999) (SZYPERSKI, 1998) (WALLNAU *et al.*, 2001a);
5. Um componente pode ***exigir*** um certo conjunto de serviços (também chamado de dependências de contexto) do ambiente no qual ele é implantado (PAGE-JONES, 1999) (SZYPERSKI, 1998). Esses serviços são, normalmente, infra-estrutura básica como, por exemplo, persistência e distribuição transparentes;
6. Um componente pode ***fornecer*** um certo conjunto de serviços que são exigidos pelo ambiente no qual ele é implantado (PAGE-JONES, 1999). Normalmente, esses serviços obrigatórios são utilizados pelas tecnologias para descobrir quais interfaces o componente realiza (BOX, 1997);
7. Todo componente pode interagir com outros componentes de software (que seguem padrões compatíveis) no mesmo ambiente, para formar unidades de software com capacidades específicas (PAGE-JONES, 1999) (WALLNAU *et al.*, 2001a);
8. Um componente pode ser vendido (ou distribuído) em forma *binária*, ao invés de forma *compilável* (código-fonte) (WALLNAU *et al.*, 2001a) (PAGE-JONES, 1999) (PARRISH *et al.*, 1999);
9. Um componente pode, dependendo da tecnologia empregada, oferecer *publicações* (em tempo de execução ou em tempo de projeto) das suas operações, de forma que outros componentes possam descobri-las e utilizá-las (PAGE-JONES, 1999).

Analisando as características acima, é importante relembrar que um consenso ainda não foi alcançado sobre o assunto, restando divergências que acabam causando diferenças entre as abordagens e tecnologias desenvolvidas. A questão do estado persistente citada no item quatro é um exemplo concreto disso, onde alguns autores (PAGE-JONES, 1999) (SZYPERSKI, 1998) (WALLNAU *et al.*, 2001a) afirmam que

componentes de software não podem possuir um estado persistente, enquanto outros (HERZUM e SIMS, 2000) (BROWN, 2000) e algumas tecnologias – como *JavaBeans* (JAVABEANS, 1997) – os permitem. Para este trabalho, seguiremos a tendência que é mais utilizada na prática: *componentes podem possuir um estado persistente*. Conseqüentemente, eles podem ser instanciados quantas vezes forem necessárias.

Na visão de CHEESMAN e DANIELS (2001), os princípios fundamentais aos quais os componentes aderem são, basicamente, os mesmos princípios que sustentam a orientação a objetos. São eles:

1. Unificação de dados e funções: Um objeto consiste de dados (ou estados) e de funções que processam estes dados. O agrupamento natural destes elementos aumenta a coesão do software.
2. Encapsulamento: Um cliente de um objeto mantém-se isolado de como os dados daquele objeto são armazenados ou como suas funções são implementadas. Esta separação é muito importante para o gerenciamento das dependências e redução do acoplamento do software.
3. Identidade: Cada objeto possui uma identidade única, independente de estado.

No contexto do desenvolvimento de software, as informações citadas acima estabelecem apenas os princípios básicos que sustentam a idéia de montagem de sistemas baseados em componentes. Apesar de muito importantes, elas mostram-se insuficientes quando procuramos fazer uma avaliação mais detalhada das atuais abordagens e tecnologias desta área. Para alcançarmos tal objetivo, é imprescindível que uma análise muito mais precisa das características dos componentes seja elaborada, permitindo que outros aspectos mais decisivos possam ser destacados.

A necessidade de existirem tais fatores reside no fato de que muitos projetos industriais e de pesquisa têm conseguido montar sistemas utilizando componentes (WALLNAU et al., 2001a). Entretanto, as abordagens utilizadas são consideradas *ad hoc* e dificilmente respeitam as fronteiras e serviços dos componentes utilizados (HAN, 1998). Nesses casos, os projetos de desenvolvimento de software que adotam tais propostas estão sujeitos a riscos e outros fatores que poderiam resultar em perdas, contratempos ou insucesso para as equipes.

BROWN (2000) afirma que muitas tecnologias e abordagens atuais de desenvolvimento baseado em componentes, por utilizarem orientação a objetos e

apoiarem a distribuição de processamento, compartilham um número de características que são familiares aos atuais projetistas e engenheiros de sistemas. Apesar disso, outros elementos menos conhecidos precisam ser destacados para que o desenvolvimento baseado em componentes possa ser corretamente exercido. Estes elementos são (BROWN, 2000):

1. Todo componente deve possuir uma especificação: todo componente deve possuir uma descrição abstrata do serviço oferecido para servir como um contrato entre os consumidores e fornecedores do serviço. Este assunto é explorado na seção 2.2.
2. Todo componente pode possuir uma ou mais implementações: Baseados em uma especificação, um componente pode possuir diversas implementações diferentes, possivelmente em diferentes tecnologias. Este assunto também é explorado na seção 2.2.
3. A montagem da arquitetura deve obedecer às restrições de um padrão: Todo componente existe dentro de um ambiente bem definido, também chamado de modelo de componente (*component model)*. Este modelo representa um conjunto de serviços que apóiam o software em desenvolvimento, além de um conjunto de regras que os componentes devem obedecer para tirar proveito destes serviços. Este assunto é explorado na seção 2.4.
4. Uma abordagem de empacotamento: Componentes podem ser agrupados de diferentes formas para fornecer serviços. Estes grupos são chamados de pacotes, e representam unidades de funcionalidade que precisam ser instaladas em um sistema. Para tornar um pacote acessível, alguma forma de registro do pacote precisa ser realizada.
5. Uma abordagem de Implantação: Uma vez que um pacote está instalado em um ambiente, ele precisa ser implantado para funcionar. Isto ocorre através da criação de um executável do componente e pelo início de suas interações com outros componentes.

A partir dos cinco elementos acima, BROWN (2000) ressalta que um entendimento mais profundo sobre componentes pode ser vislumbrado, utilizando três perspectivas que capturam estes elementos:

1. Perspectiva de Empacotamento (*packaging*): um componente é visto como uma unidade de entrega. A perspectiva de empacotamento considera um

componente como sendo um conceito organizacional, focado na identificação de um conjunto de elementos que pode ser reutilizado como uma unidade. A ênfase aqui está na **reutilização** do componente.

2. Perspectiva de Serviço (*Service*): um componente é visto como um fornecedor de funcionalidades. A perspectiva de serviço considera um componente como sendo uma entidade de software que oferece serviços (operações) para seus clientes (consumidores). Ele também enfatiza a noção de contrato entre o fornecedor e o consumidor destes serviços. A ênfase aqui se encontra na **separação** de fornecedores e usuários dos serviços, resultando em uma arquitetura baseada em serviços.
3. Perspectiva de Integridade (*Integrity*): um componente é visto como uma unidade encapsulada. A perspectiva de integridade define um componente como sendo uma implementação de uma unidade encapsulada, i.e., como um conjunto de software que mantém a integridade de seus dados gerenciados e que é independente da implementação de outros componentes. Esse critério é uma condição necessária para que os componentes possam ser facilmente atualizados ou substituídos na arquitetura. A ênfase aqui está na **substituição** e na utilização de componentes como software "*plug and play*".

BROWN (2000) ressalta, ainda, que as diferentes tecnologias de desenvolvimento baseado em componentes tendem a colocar diferentes prioridades sobre estas perspectivas. Por exemplo, tecnologias que empregam *componentes binários* (compilados) enfatizam o **empacotamento**. Tecnologias que apóiam o uso de *componentes de infra-estrutura* (genéricos) destacam o **serviço**. E, tecnologias que trabalham com *componentes registrados* no ambiente de execução (e.g., *ActiveX*, *COM*, etc.) enfatizam a **integridade**. Apesar destas variações, é importante lembrar que quanto mais adepta a abordagem estiver de todas as perspectivas, mais efeitos positivos poderão ser alcançados pelos seus usuários.

2.1.1 – Componentes & Interfaces

Analisando os dados apresentados no início deste capítulo, observamos que todo componente de software deve fornecer serviços para o sistema em que ele é integrado. Como o componente deve proteger todos os seus dados através do encapsulamento, as **interfaces** possuem um papel fundamental para esse tipo de desenvolvimento,

principalmente, por permitir que operações sejam chamadas, mantendo seus detalhes de implementação desconhecidos para os clientes (CHEESMAN e DANIELS, 2000) (D'SOUZA e WILLS, 1999) (BROWN, 2000) (HAN, 1998) (SZYPERSKI, 1998). Em linguagens e sistemas que apóiam o desenvolvimento baseado em componentes, uma interface pode ser implementada por vários componentes e um componente pode implementar diversas interfaces (BACHMAN *et al.*, 2000).

Da mesma forma que componentes eletrônicos são encaixados uns nos outros, componentes de software fornecem e exigem interfaces para funcionamento (CHEESMAN e DANIELS, 2000) (HALL, 2000) (SZYPERSKI, 1998) (YACOUB, 1999a). As interfaces fornecidas representam a porta de entrada para os serviços do componente, e são implementadas por classes internas do mesmo. Por outro lado, as interfaces exigidas são serviços externos que precisam ser disponibilizados ao componente para que seu trabalho seja realizado.

Na visão de BACHMAN *et al.* (2000), a integração de componentes e o desenvolvimento de mercados especializados nestes elementos dependem fundamentalmente da noção que possuímos de interface. BROWN (2000) apresenta algumas características adicionais sobre a utilização de interfaces no desenvolvimento baseado em componente:

1. As interfaces representam o primeiro passo para avaliar se um componente em particular é adequado para um determinado propósito;
2. Um componente pode implementar várias interfaces. Os clientes de um componente podem optar em usar quantas forem apropriadas;
3. Interfaces bem definidas encorajam a competição entre implementadores de componentes. Isto permite que arquitetos de sistemas possam escolher entre possíveis implementações, utilizando como critério alguns requisitos não-funcionais (como custo, qualidade do serviço, confiabilidade).

Nenhuma das situações descritas acima seria possível sem uma documentação apropriada para as interfaces (BROWN, 2000). No contexto do desenvolvimento baseado em componentes, a descrição de uma interface está muito relacionada ao comportamento do componente que a implementa. Para permitir que estas descrições de comportamento sejam flexíveis e possam servir como contratos para possíveis implementadores, não se pode amarrá-las a detalhes de implementação. Por esse e outros motivos, as abordagens de desenvolvimento baseado em componentes promovem

a separação de dois conceitos diferentes: as *especificações* e as *implementações* de componentes.

2.1.2 – Especificações e Implementações de Componentes

Uma das maiores dificuldades das atuais abordagens de reutilização de software é o fato de precisarmos de um detalhado conhecimento sobre um artefato reutilizável para achá-lo, integrá-lo ou adaptá-lo no contexto do sistema (SAMETINGER, 1997) (HAN, 1998). Conseqüentemente, cada uma destas tarefas é dependente da implementação que estamos lidando, tornando muito difícil modificar o artefato sem causar impactos no resto do software. Como solução para este problema, o desenvolvimento baseado em componentes mantém uma separação conceitual entre uma especificação de uma funcionalidade de componente e como ela pode ser implementada (BROWN, 2000) (SZYPERSKI, 1998) (CHEESMAN e DANIELS., 2001) (D'SOUZA e WILLS, 1999).

Antes de prosseguirmos mais adiante, precisamos definir o que entendemos por especificação de componente no contexto do desenvolvimento baseado em componentes. Segundo BROWN (2000), uma especificação de componente é uma descrição abstrata de um ou mais serviços oferecidos para servir como um contrato entre seus consumidores e fornecedores. Sob este ponto de vista, uma especificação deve descrever o comportamento esperado de um componente dentro de certas situações específicas, apresentando seus conjunto de estados, restrições e interações com outras interfaces.

A idéia da separação entre especificações e implementações de componente, também chamada de *Design by Contract* (MEYER, 1997) (CHEESMAN e DANIELS, 2001), apresenta inúmeras vantagens. Entre elas:

1. Mudanças no software podem ser facilmente articuladas através da substituição de implementações sem causar impacto nos usuários do serviço oferecido pelo componente (BROWN, 2000) (CHEESMAN e DANIELS., 2001) (D'SOUZA e WILLS, 1999);
2. O implementador de um componente tem a liberdade de escolher como a implementação do componente será feita, tendo que obedecer apenas às restrições descritas na sua especificação (BROWN, 2000) (CHEESMAN e DANIELS, 2001) (D'SOUZA e WILLS, 1999);

3. A montagem da arquitetura do software torna-se centrada nos **serviços** dos componentes (especificações), não precisando detalhar como estes serviços são implementados (BROWN, 2000) (D'SOUZA e WILLS, 1999);
4. A implementação de um componente depende apenas da sua especificação. Isto permite que a equipe de projetistas adquira alguns dos seus componentes através de terceiros (*third-parties*) (BROWN, 2000).
5. A documentação de uma especificação de componente servirá para todas as suas implementações, reduzindo o tempo gasto no desenvolvimento de manuais e outros documentos de utilização dos componentes.

Para tornar a utilização de especificações de componentes uma tarefa viável, uma descrição muita bem detalhada do comportamento do componente precisa ser feita (BROWN, 2000) (CHEESMAN e DANIELS, 2001) (SZYPERSKI, 1998). Muitos dos atuais modelos de componentes existentes na indústria de software (e.g. *JavaBeans*, *ActiveX*, etc.) focam apenas os aspectos sintáticos (operações, propriedades e eventos) do que os componentes fornecem e possivelmente exigem na sua reutilização. Isto representa apenas uma pequena parte do conteúdo necessário ao entendimento da interface de um componente, onde existe um complexo protocolo de interação embutido.

CHEESMAN e DANIELS (2001) apresentam uma visão mais prática sobre a importância de se ter especificações de componentes claramente detalhadas. Segundo eles, confiar meramente na assinatura das interfaces dos componentes para conectá-los é uma atividade imatura e imprudente. Este tipo de atitude pode levar a problemas mais sérios porque componentes diferentes podem fornecer serviços diferentes utilizando um mesmo conjunto de operações. Assim como existem peças diferentes com encaixes iguais, saber o que cada parte faz é mais importante do que observar seus encaixes. Para isso, precisamos saber como descrever o comportamento de um componente utilizando mais do que operações.

2.1.3 – Descrevendo Comportamento em Especificações de Componente

Pelo fato de componentes de software consistirem de elementos muito abstratos, descrever seus comportamentos de forma clara e precisa tem se mostrado muito difícil na prática. BROWN (2000) afirma que este tipo de descrição tem sido feito de forma precária e muito restrita, concentrando-se basicamente nas assinaturas e outras

descrições informais sobre os componentes. Além de poucos dados, esse tipo de documentação, muitas vezes, apresenta-se focado somente nas implementações desenvolvidas, não separando qualquer aspecto de comportamento que poderia ser utilizado como uma especificação mais abstrata.

Muitas pesquisas na área de componentes e interfaces têm contribuído com um conjunto rico de informações para a descrição de componentes (HAN, 1998) (YACOUB, 1999a) (YACOUB, 1999b) (NING, 1999) (HALL, 2000) (HUBER, 1998) (WIENBERG *et al.*, 1999). Enquanto novas categorias de diagramas e documentos são trazidas destas pesquisas, a grande maioria delas só abrange aspectos relacionados às implementações de componentes. É importante frisar que as atuais abordagens de desenvolvimento baseado em componentes convergiram para a proposta de separação entre especificações e implementações. Qualquer trabalho que não seguir este princípio básico dificilmente conseguirá obter sucessos satisfatórios na indústria de software baseado em componentes.

BROWN (2000) ressalta que a utilização de modelos e diagramas na descrição de especificações de componentes é um passo fundamental para o entendimento preciso do seu comportamento. Segundo ele, as vantagens que podem ser obtidas com esta utilização são:

1. Modelos podem fornecer semântica de comportamento;
2. Especificações com modelos são mais fáceis de se inspecionar quanto a sua utilidade;
3. Componentes que possuem modelos podem ser autodescritivos;
4. O impacto causado pelo uso de especificações fornecidas por terceiros é mais fácil de ser analisado quando existem modelos disponíveis;

O conjunto de modelos e diagramas utilizados para descrever o comportamento dos componentes especificados varia de acordo com cada abordagem. De uma maneira geral, as variações estão relacionadas às notações e representações sintáticas de cada elemento, porém, não apresentando diferenças que violem as características básicas do desenvolvimento baseado em componentes. Estudaremos no próximo capítulo essas variações com mais detalhes.

2.1.4 – Contratos de Componentes

Como visto anteriormente, a utilização de interfaces no desenvolvimento baseado em componentes é uma decisão puramente estratégica que procura isolar os detalhes de implementação de um fornecedor de um serviço de seus clientes. Para que isso possa ser feito, os clientes precisam, em primeiro lugar, assumir que determinadas condições estão sendo cumpridas, para que não haja a possibilidade de erro ou má utilização do serviço. Por outro lado, o projetista de um serviço precisa especificar todas as condições que um cliente precisa obedecer para que este utilize corretamente seus serviços. Assim, podemos dizer que o cliente e o fornecedor são *co-dependentes* (BACHMAN *et al.*, 2000), i.e., o cliente espera que o fornecedor disponibilize o serviço de uma certa forma, e o fornecedor espera que o cliente acesse e use o seu serviço também de uma certa forma.

Apesar da co-dependência também ser um fator importante nas interações entre módulos internos de um sistema, suas implicações para o desenvolvimento baseado em componentes se tornam críticas, devido ao objetivo de se querer substituir os componentes com facilidade. Para alcançar uma solução, a definição de *contrato de interface* assume um papel fundamental para qualquer abordagem, principalmente quando as seguintes premissas são estabelecidas (BACHMAN *et al.*, 2000):

1. Contratos abrangem duas ou mais partes;
2. As partes freqüentemente negociam os detalhes de um contrato antes de se tornarem signatários;
3. Contratos prescrevem normas de comportamento para todos os signatários;
4. Contratos não podem ser alterados, a não ser que as mudanças sejam concordadas por todas as partes.

Pelo fato de interfaces consistirem de um conjunto de operações e assinaturas (parâmetros de entrada e saída, e tipo do retorno), a especificação de contratos de interfaces tem sido feita utilizando-se pré e pós-condições (SZYPERSKI, 1998) (CHEESMAN e DANIELS, 2001) (D'SOUZA e WILLS, 1999) (BACHMAN, 2000) (BROWN, 2000). Uma pré-condição indica o que deve estar verdadeiro no momento em que a operação é chamada, enquanto uma pós-condição define uma condição que deve ser estabelecida entre o início e o fim da execução da operação. Assim, o cliente precisa estabelecer a pré-condição antes de chamar a operação, e o fornecedor, confiando nisso, deve estabelecer a pós-condição antes de retornar para o cliente.

D'SOUZA e WILLS (1999) utiliza duas condições adicionais que ajudam na especificação de uma operação em termos de **concorrência**: *rely* (confiar) e *guarantee* (garantir). A cláusula *guarantee* indica uma condição que precisa ser mantida verdadeira durante toda a execução da operação (i.e., o componente precisa **garantir** que ela é verdadeira nesse período de tempo), enquanto a cláusula *rely* indica o que se espera que esteja verdadeiro durante toda a execução da operação (i.e., o componente **confia** que a condição permanecerá verdadeira durante esse período de tempo).

Os contratos de interface especificados com as condições vistas acima definem apenas como o fornecedor de um serviço e seus clientes devem se comportar para que a interação seja bem sucedida. No contexto de fornecimento de serviços, existem certos aspectos que um componente precisa possuir, mas que não são relevantes para um cliente, como, por exemplo, seus possíveis estados, suas dependências de outras interfaces, etc. Por outro lado, uma interface define tudo que um cliente precisa saber para acessar um serviço, mas nada mais do que isso. Uma interface não precisa especificar, por exemplo, como suas implementações precisam interagir com outros componentes para que seu serviço seja realizado. Isto deve ser feito em uma especificação de componente separada, onde as interfaces fornecidas e exigidas estão inseridas em um contexto de encapsulamento. Estes dois tipos de contrato são fundamentais para o desenvolvimento baseado em componente. São eles (CHEESMAN e DANIELS, 2001) (BACHMAN et al., 2000):

- Contrato de Uso: Contrato entre um serviço de uma interface e seus clientes;
- Contrato de Realização: Contrato entre uma especificação de componente e suas implementações.

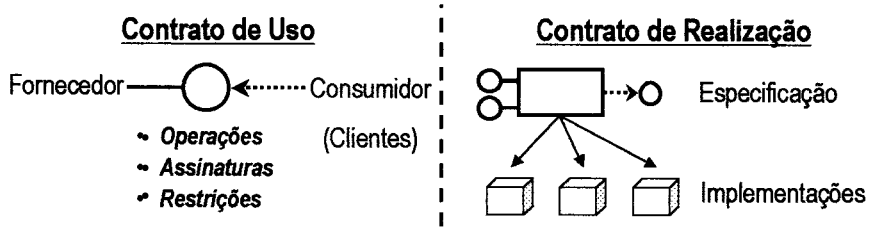


Figura 2.1: Contrato de uso e contrato de realização.

A razão primária para manter estes dois conceitos separados é a facilidade de mudança adquirida: uma mudança na especificação de componente não implica em mudanças no contrato de uso, i.e., nas interfaces do serviço. Isto permite que mudanças na especificação sejam feitas sem exigir uma reavaliação de todos os contratos de uso dos serviços do componente (CHEESMAN e DANIELS, 2001).

2.2 – Arquitetura de Componentes

No contexto do desenvolvimento baseado em componentes, a arquitetura de software assume um papel estratégico para o sucesso de qualquer abordagem de DBC. Isto ocorre porque ela representa um dos primeiros passos para o projeto de qualquer sistema, e produz entradas para a maioria, senão todos, os passos subseqüentes do desenvolvimento. Segundo SZYPERSKI (1998), todo sistema que for complexo o suficiente para possuir regras de projeto e implementação deve precisar de uma arquitetura. Para o autor, esta necessidade serve para criar a base de *independência* e *cooperação* simultânea entre as partes comunicantes, fornecendo uma solução onde o resultado é mais valioso do que a soma de todas as partes.

D'SOUZA e WILLS (1999) apresentam uma definição geral de arquitetura de componentes como: "*A arquitetura de um sistema consiste da(s) estrutura(s) de suas partes (incluindo as partes de software e hardware envolvidas no tempo de execução, projeto e teste), a natureza e as propriedades relevantes que são externamente visíveis destas partes (módulos com interfaces, unidades de hardware, objetos), e os relacionamentos e restrições entre elas*". WALLNAU *et al.* (2001a) ressaltam que todos os aspectos estruturais de um sistema são definidos por sua arquitetura, incluindo como o software é dividido em componentes, quais funcionalidades são mapeadas para esses componentes, e como os componentes interagem entre si.

A construção da arquitetura de componentes é uma atividade influenciada por diversos fatores. Antes da sua definição inicial, é imprescindível definir todos os objetivos pretendidos pelo desenvolvimento, juntamente com suas prioridades (BASS *et al.*, 1998) (BREDEMEYER, 2002). De fato, este conjunto de informações possui uma forte influência sobre todas as decisões envolvidas na escolha do tipo de arquitetura de componentes que será adotada, além de como as atividades de desenvolvimento, compra e adaptação dos componentes será feita.

Apesar de diversos fatores técnicos contribuírem na elaboração da arquitetura (como tecnologia, distribuição de processamentos, camadas, serviços, etc.), grande parte das decisões envolvidas permanecem relacionadas aos requisitos não-funcionais do software em desenvolvimento (SZYPERSKY, 1998) (WALLNAU *et al.*, 2001b) (XAVIER, 2001) (BACHMAN *et al.*, 2000) (BASS *et al.*, 1998).

De uma maneira geral, podemos conceber a montagem da arquitetura como uma atividade voltada para o funcionamento interno do software em construção, onde

componentes e interfaces serão desenvolvidos visando alcançar uma solução integrada para um problema específico de um domínio. Apesar de se tratar de uma fase que se situa em um nível mais baixo de abstração quando comparada a fase anterior do desenvolvimento baseado em componentes – a *modelagem dos requisitos* – a montagem da arquitetura deve ainda permanecer independente de qualquer implementação ou tecnologia, utilizando apenas especificações de componentes para representar os serviços existentes na arquitetura (BROWN, 2000) (CHEESMAN e DANIELS, 2001) (D'SOUZA e WILLS, 1999) (HERZUM e SIMS, 2000) (HALLSTEINSEN e SKYLSTAD, 1999).

A utilização de especificações de componentes no nível arquitetural permite que a implementação e conexão dos componentes possam ser realizadas *simultaneamente* por equipes diferentes. Este paralelismo de atividades ajuda a obter uma boa redução do tempo de desenvolvimento do software, além de simplificar o gerenciamento sobre as equipes envolvidas (AOYAMA, 1998).

Para se promover um entendimento completo sobre os diferentes aspectos envolvidos durante a fase de montagem da arquitetura, a criação de diferentes visões sobre a conexão dos componentes tem-se mostrado uma atividade muito importante na prática (D'SOUZA e WILLS, 1999). Cada visão consiste em um conjunto de modelos focados em um determinado aspecto da arquitetura, omitindo outros detalhes que são menos importantes dentro daquele contexto. Cada visão criada é relevante para uma ou mais fases do processo de desenvolvimento, servindo como insumo para cada passo da construção do software. Dentre elas, podemos citar como as duas mais utilizadas:

- Visão Lógica: Decomposição do software em componentes lógicos, suas estruturas de relacionamento, interfaces e suas dependências comportamentais. O foco desta visão está nas interfaces dos componentes e nos serviços agregados a eles, não descrevendo detalhes de implementação ou tecnologia (CHEESMAN e DANIELS, 2001) (BREDEMEYER, 2002) (D'SOUZA e WILLS, 1999) (HERZUM e SIMS, 2000) (SPARLING, 1999) (HALLSTEINSEN e SKYLSTAD, 1999).
- Visão de Implantação: Conjunto de informações sobre a implantação do software em CPUs distribuídas. O foco desta visão está nos nós distribuídos, seus componentes em execução e protocolos de comunicação da rede utilizados. (D'SOUZA e WILLS, 1999) (BREDEMEYER, 2002) (HERZUM e SIMS, 2000) (SPARLING, 1999).

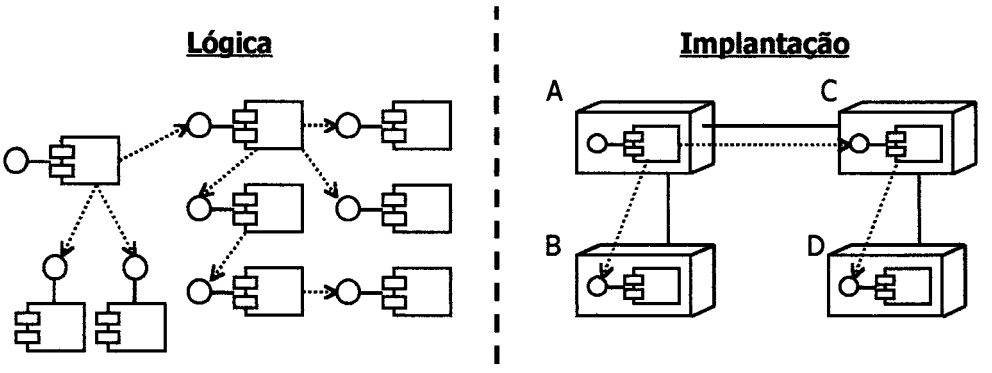


Figura 2.2: Visão lógica e de implantação da arquitetura.

A Figura 2.2 ilustra as visões lógica e de implantação do software. É importante ressaltar que existem outras visões e informações que também precisam ser descritas para que a arquitetura esteja bem documentada, como, por exemplo, informações relacionadas à execução de *threads* e processos concorrentes (D'SOUZA e WILLS, 1999).

Na prática, caso algumas dessas informações estejam relacionadas a apenas um único componente, elas devem ser documentadas diretamente na sua especificação. Nos casos onde mais componentes estão envolvidos, essa descrição deve ser elaborada pelo arquiteto e será anexada à documentação do software (CHEESMAN e DANIELS, 2001) (BROWN, 2000). Seja dentro das especificações de componente ou em documentos específicos da aplicação, o importante é que estas informações estejam descritas em algum lugar no projeto.

2.2.1 – Divisão em Camadas

Segundo alguns autores (JACOBSON *et al.*, 1997) (GRISS, 1997) (XIA *et al.*, 2000) (HERZUM e SIMS, 2000) (D'SOUZA e WILLS, 1999) (SZYPERSKI, 1998) (HALLSTEINSEN e SKYLSTAD, 1999), a arquitetura de software no contexto do desenvolvimento baseado em componentes deve ser divida em camadas com diferentes níveis de abstração. Assim, como visto na Figura 2.3, uma aplicação pode ser decomposta por camadas que acessam camadas inferiores, visando serviços mais básicos na arquitetura. A camada mais alta consiste de componentes específicos de negócio que realizam as regras de negócio do domínio. Abaixo desta camada encontram-se componentes utilitários que prestam serviços genéricos necessários ao desenvolvimento das aplicações. Finalmente, a última camada inclui os componentes de infra-estrutura que se comunicam com as plataformas de execução do software, como o sistema operacional, *drivers* ou dispositivos de hardware.

Na visão de alguns autores (HERZUM e SIMS, 2000) (D'SOUZA e WILLS, 1999) (SPARLING, 1999), as duas camadas superiores constituem a *arquitetura da aplicação* – ou *arquitetura de negócio* (SPARLING, 1999) – enquanto a última camada consiste da *arquitetura técnica* do software. A arquitetura da aplicação é formada por componentes que realizam a lógica de negócio segundo os requisitos especificados. Já a arquitetura técnica corresponde a componentes de suporte que permitem e oferecem facilidades para a execução das aplicações. A seguir, descreveremos com mais detalhes cada um dos três tipos de componentes vistos na Figura 2.3.

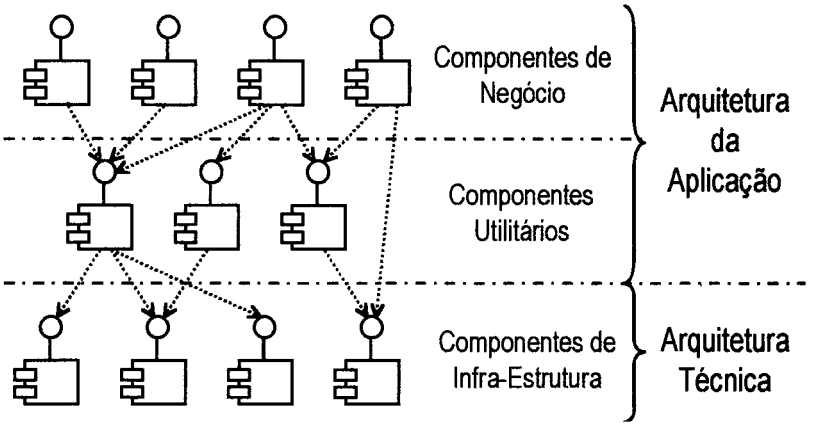


Figura 2.3: Arquitetura em camadas.

2.2.1.1 – Componentes de Negócio

Os componentes de negócio representam os serviços mais importantes de toda a arquitetura. Segundo HERZUM e SIMS (2000), um componente de negócio é uma implementação de um conceito ou processo de negócio autônomo. Ele consiste de todos os artefatos necessários para representar, implementar e implantar um conceito de negócio como um elemento reutilizável em um sistema de informação distribuído. Apesar de se tratar de conceitos e processos autônomos, estes elementos se referenciam dentro do domínio da aplicação, fazendo com que os componentes de negócio reflitam essas dependências e forneçam meios de interagir e colaborar entre si.

Para BROWN (2000), os serviços prestados pelos componentes de negócio implementam algoritmos e regras que são específicas de cada organização, tornando-os caros e representando uma vantagem competitiva sobre outros competidores do mercado. Além disso, estes componentes dificilmente são disponibilizados em repositórios de componentes ou distribuídos livremente, o que encarece seus preços e exige políticas de segurança e controle de acesso dentro das organizações.

Alguns autores, como HERZUM e SIMS (2000), CHEESMAN e DANIELS (2001) e ATKINSON *et al.* (2000), e algumas tecnologias, como *Enterprise Javabeans* (MONSON-HAEFEL, 1999), utilizam duas categorias de componente de negócio em seus trabalhos: *componentes de entidade* e *componentes de processo*. Os componentes de entidade representam os principais conceitos de um domínio, onde os processos de negócio operam. Eles correspondem a conceitos normalmente identificados pela análise orientada a objetos ou pela análise de entidade-relacionamento da aplicação (HERZUM e SIMS, 2000). Exemplos destes componentes podem ser *Fatura*, *Item*, *Cliente*, etc. Os componentes de processo representam os processos e atividades relativas ao domínio da aplicação, executando ou apoiando usuários (humanos ou não) a desempenhar tarefas específicas do negócio.

Comparando os dois tipos de componentes, HERZUM e SIMS (2000) explicam que os componentes de processo são os que mais apresentam diferenças significativas em suas funcionalidades dentro de um mesmo domínio ou setor de negócio. O que acontece é que estes componentes refletem os processos que são específicos de cada organização, apresentando detalhes que, muitas vezes, podem ser considerados como vantagens competitivas sobre as fatias de mercado. Por outro lado, os componentes de entidade tendem a apresentar menos diferenças entre organizações pelo fato de eles representarem conceitos mais conhecidos e divulgados, além de poderem pertencer a mais de um domínio de aplicação.

Segundo os mesmos autores, os componentes de processo evoluem muito mais rápido do que os componentes de entidade, e é essa capacidade de evoluir na velocidade do mercado que é o aspecto mais importante dos sistemas de informação. Normalmente, esse tipo de evolução afeta os processos executados, as abordagens e algoritmos utilizados, as suas formas de utilização, os dados produzidos, e a forma que todos estes são apresentados ao usuário final. No contexto das aplicações, os componentes de processo podem realizar um ou mais casos de uso, possuindo interfaces para executá-los e controlá-los. A identificação dos componentes de processo e suas responsabilidades dentro do contexto de uma aplicação são baseadas, normalmente, na modelagem de processo e nos casos de uso.

2.2.1.2 – Componentes Utilitários

Existem determinados componentes que não pertencem a nenhum domínio específico, mas que são utilizados por inúmeras aplicações de negócio. Esses serviços

destacam-se por serem voltados para atividades genéricas, e que representam os blocos básicos para construção dos sistemas (BROWN, 2000) (HALLSTEINSEN e SKYLSTAD, 1999). Exemplos destes serviços (componentes) são (HERZUM e SIMS, 2000) (BROWN, 2000):

1. *Entidades básicas*: controle de calendário, registro de endereços, controle monetário, etc.
2. *Comércio eletrônico*: Validação e Verificação de informações, formulários eletrônicos de pagamento, acompanhamento de pedidos em tempo-real, etc.
3. *Suporte*: Geradores de Números, Biblioteca de código de erros, Biblioteca de constantes, etc.

Dentre todas as categorias de componentes, os componentes utilitários são os mais reutilizados no contexto de desenvolvimento de sistemas de negócio. HERZUM e SIMS (2000) afirmam que estes tipos de componentes tendem a ser mais fáceis de se modelar e entender porque eles representam conceitos que são autônomos e sem dependências para processos específicos de negócio. Além disso, os requisitos destes componentes tendem, praticamente, a não se modificarem com o passar dos anos, podendo ter interfaces cada vez mais refinadas e precisas para uso geral.

É importante notar que os componentes utilitários dificilmente aparecem na modelagem estática das aplicações. Por exemplo, enquanto o conceito *pagamento* aparece nos modelos de análise, o componente que representa um *formulário para pagamentos* dificilmente aparecerá.

2.2.1.3 – Componentes de Infra-Estrutura

Para que o desenvolvimento de aplicações robustas com componentes não tenha que se preocupar com serviços básicos da arquitetura (como distribuição de objetos, persistência, etc.), utiliza-se, normalmente, componentes de infra-estrutura para suprir estes serviços (HERZUM e SIMS, 2000) (D'SOUZA e WILLS, 1999) (SZYPERSKI, 1998) (HALLSTEINSEN e SKYLSTAD, 1999). A idéia desta camada é reduzir a complexidade técnica existente na utilização de serviços primitivos da arquitetura, reduzindo significativamente o tempo e o custo do desenvolvimento do software (HERZUM e SIMS, 2000). D'SOUZA e WILLS (1999) afirmam que a arquitetura técnica consiste de serviços e funcionalidades que são independentes de domínio e afetam, notavelmente, a viabilidade e os requisitos não-funcionais da aplicação. Os

serviços desta camada incluem, entre outros (HERZUM e SIMS, 2000) (D'SOUZA e WILLS, 1999):

1. Infra-estrutura de comunicação entre componentes distribuídos;
2. Armazenamento de *logs* de eventos referentes à comunicação e exceções;
3. Serviços de acesso a bancos de dados e persistência;
4. Mecanismos de sinalização, tratamento e recuperações relativas às exceções;
5. Ativação, monitoramento e desativação de funcionalidades que dependem de eventos (como, por exemplo, *time-outs,* coleta de lixo, etc.), ou mecanismos de tratamento de falhas;
6. Facilidade quando a portabilidade dos sistemas, assim como a implantação de componentes em diferentes ambientes, seja ela em tempo de projeto, execução ou teste;

Segundo D'SOUZA e WILLS (1999), a implementação da arquitetura técnica deve ser feita antes mesmo da arquitetura da aplicação. A principal razão para isso é o fato de grande parte dos riscos de qualquer projeto virem da infra-estrutura tecnológica. Enquanto, normalmente, as equipes avaliam a complexidade do desenvolvimento baseando-se nos requisitos especificados, pouca importância é dada à implementação da infra-estrutura que apoiará a execução da aplicação.

Para que a arquitetura da aplicação consiga utilizar os serviços prestados pela arquitetura técnica, uma certa padronização precisa ser feita nos componentes de negócio e utilitários (SZYPERSKI, 1998) (HERZUM e SIMS, 2000) (D'SOUZA e WILLS, 1999). Dependendo da tecnologia usada pelo software, a equipe de desenvolvimento pode optar por desenvolver seus próprios componentes e padronizações de infra-estrutura, ou utilizar alguma que já exista no mercado, como o Java Beans (JAVABEANS, 1997) e COM (BOX, 1997). A padronização que é utilizada sobre os componentes para mantê-los compatíveis com a infra-estrutura é também chamada de *modelo de componentes (component model)*. Por outro lado, o software que representa a arquitetura técnica (propriamente dita) também é chamado de *framework de componentes*. Ambos os assuntos estão detalhados no final desta seção.

2.2.2 – Estilos Arquiteturais

Os desenvolvedores de software estão cada vez mais reconhecendo a importância de se explorar conhecimento e experiência sobre a construção de sistemas.

Uma forma de fazer isso é definindo estilos arquiteturais e relacionando-os a grupos de aplicações que possuem determinadas características semânticas em comum (WILE, 2001) (BUSCHMAN *et al.*, 1996).

SHAW e GARLAN (1996) afirmam que a utilização de estilos arquiteturais para uma família de sistemas relacionados pode, em princípio, simplificar drasticamente o processo de construção de uma aplicação, reduzir seus custos de implementação através de uma infra-estrutura reutilizável, e aumentar a integridade do software aplicando análises e verificações específicas para cada estilo.

No contexto do desenvolvimento baseado em componente, os estilos arquiteturais provenientes da orientação a objetos, normalmente, costumam tomar formas de componentes para serem aplicados à arquitetura. Trabalhos de documentação destes esquemas podem ser encontrados na literatura sob a forma de *padrões de interação de componentes* (ESKELIN, 1999), os quais podem ser aplicados em diversos contextos e independente de domínio.

Além destes trabalhos, dois estilos arquiteturais específicos para o desenvolvimento de componentes de negócio foram documentados por HERZUM e SIMS (2000), os quais são denominados por eles de estilo **baseado em tipos** (*type-based*) e de estilo **baseado em instâncias** (*instance-based*). Embora esses autores tenham sido os primeiros a dar um nome para esses estilos, os mesmos podem ser encontrados em outras abordagens de DBC (CHEESMAN e DANIELS, 2001) (BROWN, 2000) (ATKINSON, 2000), o que os tornam mais conhecidos nessa área.

2.2.2.1 – Estilo baseado em Tipos (Type-based)

O estilo *baseado em tipos*, também conhecido como *baseado em serviços*, procura tratar seus componentes como uma representação de um tipo de negócio[1] na arquitetura. A estratégia destes componentes é encapsular todas as instâncias de um tipo de negócio, não fornecendo qualquer acesso às suas informações que não sejam por suas interfaces.

Por serem gerentes de instâncias, os componentes no estilo baseado em tipos precisam oferecer meios de identificar unicamente cada uma de suas instâncias. Para isso, utilizam-se identificadores numéricos (ou alfanuméricos) que são associados a elas, permitindo que elas sejam encontradas e referenciadas na execução do software

---

[1] *Tipo de negócio (business type)* é um conceito que deve ser tratado por uma aplicação. Veremos mais sobre esse assunto no próximo capítulo.

(CHEESMAN e DANIELS, 2001) (BROWN, 2000) (HERZUM e SIMS, 2000). Os números identificadores de cada instância são chamados de *chaves técnicas (technical key)* (MCINNIS, 1999) e, normalmente, são gerados por componentes utilitários da arquitetura. As chaves técnicas possuem este nome porque não fazem parte do domínio da aplicação, constituindo apenas uma solução meramente técnica para o problema.

Pelo fato de cada componente encapsular suas instâncias, não é possível endereçá-las diretamente (utilizando-se, por exemplo, ponteiros) nem através da rede (por exemplo, chamadas remotas). Nesses casos, qualquer operação que precise ser realizada sobre uma ou mais instâncias deve ser feita diretamente para o componente que a gerencia, passando como parâmetros o(s) identificador(es) em questão. Dessa forma, as interfaces deste estilo tendem a ter o seguinte formato:

*Interface.nomeOperacao(**in** id_instancia, **in** ou **out** algum_outro_dado): retorno.*

Quando observamos a presença dos identificadores nas interfaces dos componentes deste estilo, é natural questionarmos a sua necessidade, já que poderíamos utilizar os próprios atributos dos tipos de negócio para identificá-los. Apesar de parecer intuitiva, essa idéia apresenta desvantagens que precisam estar evidentes para o arquiteto. A primeira delas é o aumento da dependência entre os componentes da arquitetura, já que modificar os atributos de um tipo de negócio exigirá mudanças isoladas em outros componentes. Além disso, cada tipo de negócio que possuir mais de um atributo-chave complicará as interfaces que estão relacionadas a ele, principalmente, por introduzir mais parâmetros e restrições de uso.

O gerenciamento de tipos de negócio no estilo baseado em tipos não precisa ter um componente para cada tipo modelado. Existem casos onde um componente torna-se mais adequado quando este gerencia tipos de negócio que estão relacionados dentro do domínio de aplicação. Entretanto, as decisões que envolvem a escolha dos tipos que deverão pertencer a um mesmo componente permanecem vagas na literatura.

Segundo HERZUM e SIMS (2000), este estilo é o mais conservador e livre de riscos dentre todos os conhecidos. Ele possui o apoio das tecnologias atuais mais maduras, como *COM* (BOX, 1997), *JavaBeans* (JAVABEANS, 1997), etc., e representa o passo mais simples que as organizações podem dar, sendo a abordagem mais intuitiva para os desenvolvedores tradicionais voltados para aplicações cliente/servidor.

2.2.2.2 – Estilo baseado em Instâncias (Instance-based)

O estilo *baseado em instâncias* possui características completamente opostas ao estilo baseado em tipos. Neste estilo, cada instância de um tipo de negócio corresponde a uma instância de componente que pode ser endereçada diretamente na arquitetura. Por exemplo, se temos as faturas *F1532* e *F1711* cadastradas no sistema, ambas correspondem a objetos endereçáveis na memória. Isto exige que dois tipos de componentes sejam criados: o *componente-instância*, que representa as instâncias conceituais, e o *componente-coleção*, que gerencia essas instâncias (HERZUM e SIMS, 2000) (CHEESMAN e DANIELS, 2001).

Como cada instância pode ser referenciada em memória, o trabalho dos componentes-coleção é unicamente criar e procurar instâncias para outros componentes da arquitetura. Qualquer operação que precise ser realizada sobre uma determinada instância pode ser feita diretamente em sua interface, livrando o componente-coleção deste serviço. Desta forma, a interface do componente-instância e do componente-coleção possui o seguinte formato:

- Componente-Instância:

  *InterfaceInstancia.create(**in** algum_dado)*

  InterfaceInstancia.getAlgumDado(): algum_dado

  *InterfaceInstancia.setAlgumDado(**in** algum_dado)*

- Componente-Coleção:

  *InterfaceColecao.create(**in** algum_dado): instancia_criada*

  *InterfaceColecao.find(**in** criterio_busca): lista_instancias*

2.2.3 – Comparação dos Estilos

Os dois estilos apresentados possuem características variadas e podem ser empregados em diferentes contextos de aplicação. A primeira diferença que podemos observar sobre os dois casos está relacionada aos seus principais aspectos filosóficos: o estilo baseado em tipos pode ser visto como uma extensão da modularização tradicional feita nos sistemas. Já o estilo baseado em instâncias pode ser visto como uma extensão da visão orientada a objetos a qual estamos mais acostumados ultimamente, parecendo mais natural por utilizarmos conceitos como encapsulamento, polimorfismo, herança, etc.

Apesar da semelhança com a orientação a objetos, HERZUM e SIMS (2000) afirmam que o estilo baseado em instâncias está longe de ser uma abordagem onde os objetos só precisam de interfaces para funcionar como um sistema distribuído. Muito pelo contrário, ele ressalta que para sistemas de grande porte que possuem uma equipe muito grande e exigem um elevado grau de desempenho envolvendo muitos usuários, o trabalho exigido para um bom rendimento com a reutilização, gerenciamento, projeto e implantação dos vários objetos distribuídos está além da capacidade de muitas organizações. Sob as mesmas condições, o estilo baseado em tipos apresentaria uma complexidade substancialmente menor, podendo, inclusive, ser adotado em situações ainda mais complexas.

Como a utilização de cada estilo produz componentes com diferentes níveis de granularidade, as questões de persistência podem se tornar um diferencial nesse assunto. O estilo baseado em instâncias, por tratar de componentes mais simples e soltos pela arquitetura, tende a utilizar mecanismos de persistência transparente e que são fornecidos pelo framework da aplicação (arquitetura técnica), como serialização, por exemplo. Por outro lado, componentes baseados em tipos teriam problemas com esses mesmos mecanismos, principalmente, por serem responsáveis por quantidades muito grandes de dados, perdendo muito desempenho sempre quando toda sua imagem de memória fosse armazenada ou restaurada de uma vez só. Portanto, os mecanismos de persistência do estilo baseado em tipos devem ser fornecidos por algum outro componente da arquitetura, ou implementado pelo próprio componente, o que aumentaria ainda mais sua granularidade. CHEESMAN e DANIELS (2001) sugerem que esses componentes não mantenham objetos em memória, apenas na base de dados. Dessa forma, sempre que um elemento precisasse ser atualizado ou consultado, bastaria acessar diretamente o registro utilizando sua chave técnica (identificador).

Apesar das grandes vantagens do estilo baseado em tipos, é importante reconhecer que componentes baseados em instância podem ser extremamente poderosos quando aplicados em situações corretas, alcançando satisfatoriamente objetivos que envolvem desempenho, encapsulamento, concorrência, utilização e balanceamento de carga da rede. Entretanto, para se conseguir estes objetivos, um esforço considerável deve ser empregado sobre a equipe para se obter maturidade arquitetural e organizacional, reduzindo as chances de falhas ou outros problemas nos projetos (HERZUM e SIMS, 2000).

2.2.4 – Aplicação dos Estilos

Vimos na seção 2.4.1 que a arquitetura da aplicação é composta por duas categorias distintas de componentes: componentes de negócio, que podem ser componentes de entidade ou componentes de processo, e os componentes utilitários. Sobre esta divisão funcional, HERZUM e SIMS (2000) consideram razoável utilizar diferentes estilos arquiteturais para o desenvolvimento de cada um deles, permitindo certos componentes da aplicação serem construídos baseados em tipos e outros baseados em instâncias. A Figura 2.4 ilustra as diferentes categorias e seus estilos apropriados.

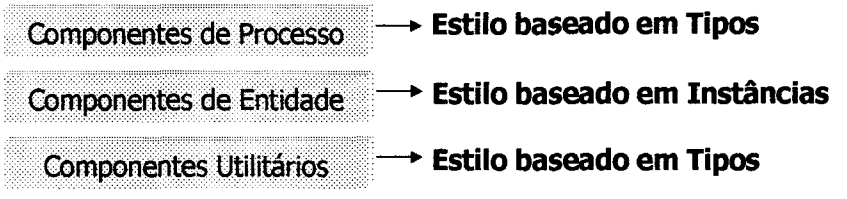


Figura 2.4: Categorias funcionais e seus estilos apropriados.

Quando consideramos o desenvolvimento de componentes de processo, por exemplo, só faz sentido termos o estilo baseado em tipos na arquitetura. Isto acontece porque, neste estilo, os objetos de negócio ficam encapsulados dentro dos componentes que gerenciam as instâncias, o que torna mais complexa a manipulação dessas informações na arquitetura. Conseqüentemente, é muito vantajoso utilizar componentes de processo para reduzir essa complexidade, embutindo, no seu comportamento, essa comunicação com esses componentes que encapsulam os objetos de negócio. No caso do estilo baseado em instâncias, essa complexidade de acesso aos dados é muito menor, já que as instâncias podem ser referenciadas diretamente, tornando desnecessário a existência de componentes intermediários nessa interação.

Ainda na Figura 2.4, os componentes de entidade, por razões bastante intuitivas, exigem um estilo baseado em instâncias na arquitetura. Finalmente, os componentes utilitários tendem a ser baseados em tipos porque, normalmente, os objetos gerenciados por eles possuem uma granularidade baixa, não exigindo que estes sejam endereçáveis através da rede.

2.2.5 – Remoção e Integridade Referencial

Tendo em vista que tipos de negócios possuem dependências entre si, a remoção de instâncias de tipos de negócio na arquitetura levanta uma importante questão sobre a

integridade dos dados existentes: como fazer para que as referências mantidas pelas instâncias dos tipos de negócio permaneçam válidas durante toda a execução do software? HERZUM e SIMS (2000) respondem dizendo que a responsabilidade pertence a cada componente, controlando diretamente seus dados e indiretamente os dados mantidos por outros componentes. Neste último caso, a coordenação é feita através das interfaces dos outros componentes, os quais corrigirão seus dados conforme o esperado.

Para os mesmos autores, quando um banco de dados é utilizado pelos componentes de negócio, a integridade dos dados pode ser deixada sob sua responsabilidade, evitando acessos a outros componentes da arquitetura. Em CHEESMAN e DANIELS (2001), a integridade dos dados é feita pelos componentes de processo, que coordenam as chamadas de remoção entre os componentes responsáveis pelos tipos de negócio. Outra possibilidade para este assunto é utilizar *eventos* que possam ser disparados durante uma remoção, fazendo com que outros componentes reajam a eles (HERZUM e SIMS, 2000).

2.2.5.1 – Comunicação por Eventos

A utilização de eventos em aplicações baseadas em componentes representa um poderoso mecanismo de coordenação e comunicação na arquitetura, o que aumenta a autonomia dos componentes e desacopla suas estruturas. O princípio básico existente por trás desta idéia é que, ao invés de se chamar operações em interfaces predeterminadas, os componentes geram e recebem eventos, reagindo a eles de acordo com o fato acontecido (D'SOUZA e WILLS, 1999) (HERZUM e SIMS, 2000).

Um dos problemas mais comuns no uso de eventos é a dificuldade de se prever a ordem das chamadas que serão desencadeadas quando um deles ocorrer. Segundo HERZUM e SIMS (2000), as aplicações baseadas em componentes podem tirar vantagens de modelos híbridos que utilizam eventos como um meio de comunicação apenas em circunstâncias particulares. Exemplos destas circunstâncias são:

1. Como um mecanismo de *refresh* para atualizar interfaces ou conjuntos de dados que estão sendo freqüentemente modificados;
2. Como um mecanismo de aviso para componentes de interface com o usuário, onde o usuário precisa ser notificado ou consultado sobre alguma informação;
3. Para implementar a integridade referencial entre tipos de negócio;

4. Para implementar eventos reais dos processos de negócio da aplicação, desencadeando ações que serão tratadas pela arquitetura e pelos usuários.

2.2.6 – Modelos de Componentes

Para que a montagem de uma aplicação possa ser bem sucedida, determinadas regras e condições precisam ser respeitadas pelos componentes na arquitetura, de forma que estes consigam interagir no contexto da tecnologia utilizada. Perguntas do tipo "*Como as interfaces são especificadas? Como as interfaces são nomeadas? Existem interfaces obrigatórias na implementação dos componentes?*" (SZYPERSKI, 1998) precisam ser respondidas por todos os componentes de forma igual, para que a integração com os componentes de infra-estrutura ocorra corretamente. Para formatar os componentes dentro de um certo padrão, modelos de componentes como *JavaBeans* (JAVABEANS, 1997), COM (BOX, 1997), EJB (MONSON-HAEFEL, 1999) e CORBA (MOWBRAY e RUH, 1997) foram criados para responder estas questões, definindo como seus componentes devem ser descritos para se comunicarem com a arquitetura técnica do software.

BACHMAN *et al.* (2000) apresenta os objetivos alcançados pela utilização de modelos de componentes no contexto do desenvolvimento de software:

- Composição Uniforme: Dois componentes podem interagir se, e somente se, eles compartilham suposições sobre o que é fornecido e exigido um do outro. Obviamente, algumas destas suposições se referem a aspectos únicos de cada componente (normalmente, o seu serviço prestado). Porém, existem outras suposições que precisam estar padronizadas em todos os componentes, para que as chances de incompatibilidade acidental sejam reduzidas.
- Atributos de Qualidade Apropriados: Como os atributos de qualidade do software são dependentes do estilo arquitetural utilizado (SZYPERSKY, 1998) (WALLNAU *et al.*, 2001b) (XAVIER, 2001) (BACHMAN *et al.*, 2000), padronizar os tipos de componentes usados em um sistema e os seus padrões de interação é uma forma de garantir que uma integração realizada com componentes desenvolvidos por terceiros irá possuir os *atributos de qualidade* desejados. Além dos atributos de qualidade, pode-se obter a *qualidade do serviço* pela especificação de padrões de interação como transacionais, criptografados, etc.

- Implantação de Componentes e Aplicações: O sucesso do desenvolvimento baseado em componentes depende de um mercado robusto que contenha muitos componentes desenvolvidos por terceiros. Para que a composição de componentes seja viável neste contexto, é fundamental que os componentes comprados possam ser exportados do ambiente de desenvolvimento e importados em um ambiente de conexão. Além disso, após conectar os componentes, a aplicação gerada precisa ser implantada no ambiente do cliente, permitindo, quando necessário, que estes componentes possam interagir com outros componentes de outras aplicações. De fato, estas necessidades fazem parte da motivação de frameworks de componentes: eles aproveitam a padronização criada pelos modelos de componentes para fornecer suporte à implantação de componentes e aplicações em tempo de composição e execução.

A implantação de componentes e aplicações descrita acima tem feito com que os modelos de componente encorajem os desenvolvedores a construir software utilizando *late binding*[2] (D'SOUZA e WILLS, 1999) (BACHMAN *et al.*, 2000). Apesar da complexidade do desenvolvimento aumentar quando utilizamos esta técnica, vantagens importantes são conseguidas em troca, como facilidade na substituição dos componentes, integração com componentes feitos por terceiros, fortalecimento de mercados de componentes, etc. (BROWN, 2000) (BACHMAN *et al.*, 2000) (SZYPERSKY, 1998).

Para entendermos melhor como os modelos de componente agem para alcançar os objetivos vistos acima, precisamos observar com mais detalhes os tipos de restrições e acordos que eles podem prescrever. BACHMAN *et al.* (2000) enumeram estes tipos:

- Tipos de um Componente: Um tipo de um componente pode ser definido em termos das interfaces que ele implementa. Se um componente implementa três interfaces X, Y e Z, então ele é do tipo X, Y e Z, e pode assumir o papel de qualquer uma delas em momentos diferentes da execução. Desta forma, um modelo de componente pode exigir que seus componentes implementem uma ou mais interfaces, garantindo que eles sejam de um ou mais tipos.

---

[2] *Late binding* é uma técnica que permite que os componentes da aplicação só sejam conectados em tempo de execução, aumentando a facilidade de substituí-los e implantá-los.

- Esquemas de Interação: Modelos de componente especificam como os componentes são localizados, quais protocolos de comunicação são usados, e como as qualidades de serviço (e.g., segurança e transações) são alcançadas. Um modelo de componente descreve dois tipos de interação:
  - *Interação entre componentes*: A interação entre componentes inclui restrições como: os tipos de componentes que podem ser clientes de outros tipos, o número máximo de clientes permitidos, etc.
  - *Interação com o framework*: A interação com o *framework* inclui restrições sobre gerenciamento de recursos, como: o ciclo de vida de um componente (ativação e desativação), gerenciamento de *threads*, persistência, etc.
- Conexão de Recursos: O processo de composição de componentes está muito relacionado à ligação de componentes com um ou mais recursos disponíveis. Um recurso, neste sentido, pode ser um serviço do *framework* sendo utilizado, ou outro componente implantado no *framework*. Um modelo de componente descreve quais recursos estão disponíveis aos componentes, e como e quando os componentes são ligados a estes recursos. O mecanismo de implantação de componentes no *framework* também está descrito no modelo de componente.

2.2.7 – *Frameworks* de Componentes

Conforme vimos acima, os modelos de componente servem para padronizar, segundo determinadas regras, os componentes que irão compor a arquitetura de um software. Entretanto, para que isto seja possível, é necessário que um suporte de software exista para gerenciar esses componentes de acordo com as regras que eles obedecem. Este suporte – chamado de *Framework de Componentes* – pode ser visto como um pequeno "sistema operacional" que serve para gerenciar os recursos compartilhados pelos componentes, e fornecer todos os mecanismos que permitem a interação entre eles (BACHMAN *et al.*, 2000) (WECK, 1999).

A utilização de *frameworks* de componentes está diretamente relacionada a um dos objetivos principais do desenvolvimento baseado em componentes: a criação de um mercado robusto de componentes (WECK, 1999). Isto acontece porque os mercados de componente exigem padronizações que viabilizem a construção de ferramentas de desenvolvimento, composição e implantação de seus componentes. Além das transições

entre estas três fases, os componentes de software devem ser capazes de estabelecer conexões (em tempo de execução ou composição) com outros componentes desenvolvidos por terceiros (HUGHES, 1999). Alcançar estes objetivos sem qualquer ferramental ou padronização seria praticamente inviável.

SZYPERSKI (1998) afirma que a contribuição mais importante dos *frameworks* de componentes é a imposição parcial de princípios de arquitetura. Isto acontece porque quando um *framework* obriga determinadas instâncias de componente a realizar determinadas tarefas que estão sob o seu controle, o *framework* pode impor algumas políticas estratégicas. Um exemplo concreto disso ocorre quando um *framework* impõe ordem sobre o disparo de eventos *multicast* em componentes, eliminando erros que poderiam ser causados por chamadas desordenadas entre eles.

Apesar das vantagens existentes no uso dos *frameworks* de componentes, HUGHES (1999) ressalta que a presença de vários *frameworks* no mercado[3] torna difícil a escolha do mais adequado no contexto do software que se deseja construir. Além disso, outros desafios ainda permanecem incertos, como a utilização de mais de um *framework* em uma mesma aplicação (SZYPERSKI, 1998), e a complexidade de se construir *frameworks* configuráveis para se adaptarem aos requisitos não-funcionais do software (BACHMAN *et al.*, 2000).

É importante lembrar que, além desses *frameworks de componentes*, a literatura emprega outros significados a essa mesma palavra. No caso da abordagem *Catalysis* (D'SOUZA e WILLS, 1999), por exemplo, os *frameworks* são estruturas parametrizáveis constituídos por artefatos comuns a um mesmo domínio de aplicação. Outro exemplo é a abordagem *KobrA* (ATKINSON *et al.*, 2000), onde os *frameworks* são modelos que descrevem semelhanças e variações entre as aplicações de um mesmo domínio. Melhores detalhes sobre esses *frameworks* estão explicados na seção 3.3.

2.3 – Conclusões

As inovações trazidas pelas atuais tecnologias de software, somadas ao crescente uso da Internet, estão influenciando significativamente a forma com que o desenvolvimento de software se adapta para atender as novas necessidades dos seus

---

[3] Os *frameworks* de componentes atuais, em sua grande maioria, estão embutidos nas próprias tecnologias de componentes, como a Máquina Virtual Java (JVM), e também nos sistemas operacionais (como o *Windows*, que controla a execução de componentes *COM/DCOM/COM+*) (BOX, 1997) (KIRTLAND, 1997). Além desses, outros podem ser encontrados separadamente, como é o caso do *OpenDoc* (SZYPERSKI, 1998).

clientes. Em virtude disso, o conceito de componente como uma unidade encapsulada e substituível está se consolidando e ganhando melhorias a cada pesquisa realizada sobre esse tema.

Nesta perspectiva, estudamos neste capítulo os princípios fundamentais que servem como base para o desenvolvimento baseado em componentes, destacando seus aspectos mais importantes para o estudo deste trabalho. Vimos como os componentes são constituídos e caracterizados, e como eles são dispostos para formar a arquitetura de uma aplicação. Veremos no capítulo seguinte como esse conjunto de informações é usado no desenvolvimento de software.

Capítulo 3 – Desenvolvimento Baseado em Componentes

Estudamos no capítulo anterior os fundamentos básicos do desenvolvimento baseado em componentes, focando seus principais conceitos e definições. Além disso, vimos a organização da arquitetura das aplicações e como os componentes são classificados e estruturados para desempenharem corretamente suas tarefas.

Discutiremos nesse capítulo questões mais avançadas sobre a montagem do software a partir de componentes, destacando os aspectos mais práticos envolvidos nessas atividades. Para isso, iniciaremos com o estudo do processo de DBC e, em seguida, verificaremos como os requisitos podem ser descritos em termos de modelos e diagramas de análise. Por fim, apresentaremos as principais abordagens de DBC que foram estudadas e serviram como base para a realização deste trabalho. As seções deste capítulo estão divididas na seguinte forma:

- **Seção 3.1**: Apresenta o *processo* básico de DBC, destacando as suas principais atividades e tarefas envolvidas;
- **Seção 3.2**: Apresenta como a *modelagem* e a *diagramação dos requisitos* são realizadas em DBC;
- **Seção 3.3**: Apresenta as *abordagens* estudadas neste trabalho, suas variações e peculiaridades.

3.1 – O Processo de Desenvolvimento Baseado em Componentes

Comparado com os processos tradicionais de desenvolvimento de software (PRESSMAN, 2000), o processo do desenvolvimento baseado em componentes apresenta mudanças consideráveis em sua estrutura (BERTOLINO *et al.*, 2002). Ao colocar a reutilização de software em primeiro plano, a construção de uma aplicação pode ser vista como um puro processo de montagem de software que utiliza peças com encaixes e serviços bem definidos (BROWN, 2000).

Apesar das diferentes abordagens de DBC existentes atualmente (descritas no final deste capítulo), BROWN (2000) afirma que o método aplicado por cada uma delas é fundamentalmente o mesmo, não apresentando variações que fujam à idéia de

montagem e integração de software. Isto faz com que, de uma forma ou de outra, todas possuam três passos básicos em seus processos (Figura 3.5): o **entendimento do contexto**, a **definição da arquitetura** e, por último, o **fornecimento da solução**. Na prática, esses passos podem ser executados *top-down* ou *bottom-up*, desenvolvendo aplicações desde o seu início ou partindo-se de artefatos já existentes, como sistemas legados, por exemplo.

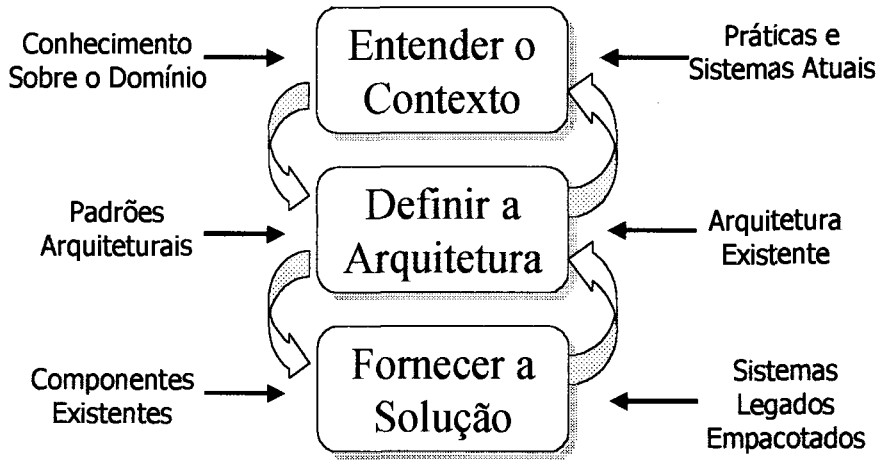


Figura 3.5: Passos básicos do desenvolvimento baseado em componentes.

O *entendimento do contexto* representa as atividades de análise referentes ao software que será construído, representando com modelos e diagramas o comportamento dos objetos dentro do domínio do problema. Um estudo detalhado sobre esta fase está descrito na seção 3.2 deste capítulo.

A *definição da arquitetura* refere-se às atividades de criação, busca e conexão dos componentes para mostrar como o software deve ser integrado internamente. Isto envolve encapsular e distribuir funcionalidades pelos componentes, os quais poderão ser desenvolvidos ou comprados separadamente de terceiros. Os componentes manipulados no nível da arquitetura não apresentam detalhes de implementação, simbolizando, apenas, os *serviços* fornecidos e consumidos entre eles.

Por fim, o *fornecimento da solução* representa a implementação e a montagem física do software, escolhendo tecnologias e infra-estruturas onde a aplicação será executada. Esta fase também envolve os testes e a implantação do software no ambiente do usuário.

AOYAMA (1998) apresenta uma visão um pouco mais detalhada sobre o processo de DBC (Figura 3.6), onde as atividades de aquisição de componentes e análise dos requisitos são iniciadas simultaneamente antes da definição da arquitetura.

Os testes efetuados sobre o sistema são divididos em duas atividades distintas: *testes de integração* (voltados para a conexão dos componentes) e os *testes funcionais* (voltados para as funcionalidades previamente modeladas).

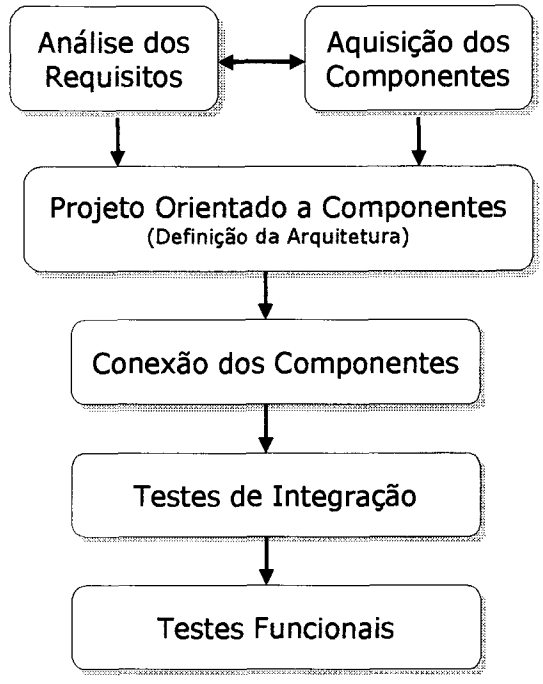


Figura 3.6: Processo proposto por AOYAMA (1998).

A nova forma de construção de software imposta pelo desenvolvimento baseado em componentes originou novos *papéis* para desenvolvedores e gerentes dentro do processo. Isto ocorreu devido à introdução de novas atividades e tarefas que exigem conhecimento especializado de seus executores (SZYPERSKI, 1998). Os principais papéis encontrados na literatura, e que são diferentes daqueles encontrados no desenvolvimento tradicional, são (SZYPERSKI, 1998) (BERGNER *et al.*, 1999):

- **Arquiteto de Componentes**: Sua missão é analisar os requisitos dos sistemas que já existem e dos que serão desenvolvidos, montando e dividindo suas arquiteturas em componentes de software. Seu trabalho envolve a escolha das tecnologias, infra-estrutura e ambientes de implantação, além do monitoramento dos aspectos técnicos durante o desenvolvimento.
- **Implementador de Componentes**: Seu papel é analisar especificações de componente para projetá-las e implementá-las em alguma tecnologia. Como vimos anteriormente, a implementação de componentes pode ser terceirizada ou voltada para mercados especializados em domínios.

- **Montador de Componentes**: Sua missão é selecionar implementações de componentes e tratar todos os aspectos referentes as suas conexões. Isto envolve questões relacionadas a compatibilização de interfaces e criação de adaptadores. Uma importante responsabilidade do montador de componentes é alertar a equipe de desenvolvimento sobre a viabilidade do uso de determinadas tecnologias ou implementações de componentes.

3.2 – Análise e Modelagem de Requisitos

Os atuais sistemas de software e os problemas que eles resolvem são complexos demais para um ser humano entendê-los como um todo. Portanto, é fundamental que modelos sejam construídos visando fornecer meios de entender tanto o domínio do problema quanto os próprios sistemas em diferentes níveis de abstração e variados pontos de vista. A idéia de modelagem vem sendo aplicada, há décadas, em inúmeros ramos da engenharia. A importância dos modelos criados, normalmente, vai além do entendimento do problema, servindo como base para outras atividades, como estimativas de custos e prazos, distribuição de trabalhos e alocação de recursos (ERIKSSON e PENKER, 1998).

A crescente demanda por sistemas maiores e mais complexos serviu como um importante incentivo para a evolução e aperfeiçoamento das técnicas de modelagem nas últimas décadas, partindo do antigo pensamento estruturado até os recentes paradigmas da orientação a objetos. A importância deste último vem sendo fortemente reconhecida devido à analogia que se estabelece entre a forma em que o software é modelado e a visão que temos sobre o mundo real, fazendo com que a modelagem do problema se encaixe melhor na estrutura dos sistemas desenvolvidos.

Nas últimas décadas, inúmeras técnicas orientadas a objetos surgiram na literatura, como o método de *Booch* (BOOCH, 1994), *OMT* (RUMBAUGH *et al.*, 1991), *OOSE/Objectory* (JACOBSON *et al.*, 1992), e *Fusion* (COLEMAN *et al.*, 1994). Dentre todas, a linguagem *UML* (PAGE-JONES, 1998) (ERIKSSON e PENKER, 1998), por ser a unificação da notação dos três primeiros acima, pôde ser considerada uma das mais importantes notações já criadas, servindo como base para a maioria das novas abordagens que surgiram após a sua criação.

No desenvolvimento baseado em componentes não foi diferente. As notações e diagramas da *UML* estão presentes, praticamente, em todas as abordagens estudadas durante a realização deste trabalho (as abordagens estão descritas em mais detalhes no

final deste capítulo). Apesar desta significativa influência, cada abordagem procurou estender e ajustar a *UML* de acordo com suas próprias necessidades, criando interessantes variações que tornaram diversificado o desenvolvimento de aplicações a partir de componentes. A ênfase desta seção está no estudo destas tendências, onde analisaremos e destacaremos importantes aspectos da modelagem que são fundamentais para o entendimento dos demais capítulos.

Por terem suas raízes na orientação a objetos, as abordagens de desenvolvimento baseado em componentes dividem sua modelagem de análise em duas partes distintas: a modelagem *estática* e a *dinâmica*. Particularmente, esta divisão revelou-se muito importante para este trabalho e, por isso, estudaremos cada uma delas separadamente.

3.2.1 – Modelagem Estática

A modelagem estática de uma aplicação representa, de uma forma geral, um conjunto de termos que descreve o **conhecimento** do software sobre um determinado domínio de problema. Os termos apresentam relacionamentos entre si que retratam como a aplicação deve manipulá-los durante a sua execução (D'SOUZA e WILLS, 1999) (BROWN, 2000). Alguns autores (CHEESMAN e DANIELS, 2001) (BROWN, 2000) (ATKINSON *et al.*, 2000) sugerem que o primeiro diagrama estático de uma aplicação seja um mapa de conceitos com seus relacionamentos. O objetivo deste diagrama é servir como um glossário de termos que serão utilizados no desenvolvimento, procurando, o mais cedo possível, igualar o vocabulário da equipe para eliminar ambigüidades e redundâncias na comunicação.

Na medida em que o conhecimento sobre o software é aprofundado, um diagrama mais detalhado precisa ser definido. A idéia pregada pelas abordagens de DBC consiste em identificar todos os **tipos de negócio** (*business types*) que o novo sistema deverá tratar, refinando e detalhando o mapa conceitual em um diagrama de dados bastante preciso (CHEESMAN e DANIELS, 2001) (BROWN, 2000) (ATKINSON *et al.*, 2000) (D'SOUZA e WILLS, 1999) (MCINNIS, 2000).

Para que nosso entendimento esteja completo, precisamos compreender melhor o significado do termo **tipo** (*type*) para a área de DBC. Embora exista um consenso entre os autores envolvidos com esse tema, podemos analisar como alguns deles explicam, com suas próprias palavras, o significado atribuído a esse conceito:

- *"Tipos são classes no nível de especificação. Eles descrevem apenas informações, não software"* (CHEESMAN e DANIELS, 2001).

- *"Um tipo é um objeto de interesse do domínio que é descrito pelo seu comportamento. Tipos são independentes de implementação"* (D'SOUZA e WILLS, 1999) (BROWN, 2000).
- *"Um tipo descreve o papel que um objeto assume em uma interação"* (MCINNIS, 2000).

O ***Diagrama de Tipos de Negócio*** (ou *business type diagram – BTD*) é fundamental para qualquer aplicação baseada em componentes porque ele representa como os dados serão gerenciados persistentemente pelo sistema, além de ser o último diagrama elaborado antes da montagem e definição da arquitetura de componentes. BROWN (2000) afirma que este diagrama é muito semelhante ao tradicional diagrama de entidade-relacionamento (*E-R*), sendo que as *entidades* são chamadas de *tipos*, e os *relacionamentos*, chamados de *associações*.

Segundo o mesmo autor, as abordagens anteriores ao desenvolvimento baseado em componentes costumavam tratar o gerenciamento dos dados persistentes da aplicação como simples bases de dados. Já em DBC, o objetivo tornou-se completamente diferente, uma vez que a persistência dos dados pode ser feita de diferentes formas e distribuída por diferentes componentes na arquitetura (HERZUM e SIMS, 2000) (CHEESMAN e DANIELS, 2001). Portanto, a modelagem dos tipos de negócio deve ser vista, simplesmente, como um conjunto de informações conceituais que serão manipuladas pelo software independente da forma de armazenamento.

Os diagramas de tipos de negócio possuem poucas diferenças de notação entre as abordagens de DBC estudadas. Com uma pequena diferença no método *Catalysis*[4] (D'SOUZA e WILLS, 1999), todos apresentam este diagrama como uma abstração do diagrama de classes da *UML*, onde as "classes" são os *tipos de negócio* e as "associações" representam as restrições estáticas que existem entre os elementos do domínio (Figura 3.7) (CHEESMAN e DANIELS, 2001) (ATKINSON *et al.*, 2000) (BROWN, 2000).

---

[4] O método *Catalysis* difere das outras abordagens porque ele emprega o conceito de *tipo* além da modelagem de requisitos, utilizando-os também para representar elementos da arquitetura (como especificações de componente, por exemplo). A principal diferença que podemos observar nesta abordagem, é que, quando um tipo representa um elemento da arquitetura, outros tipos podem ser desenhados em seu interior para representar o conhecimento que aquele elemento possui sobre o domínio. O mais importante é que isso não ocorre na notação da etapa inicial de análise, onde apenas o modelo de negócio está sendo criado. Nesta etapa, a notação utilizada é semelhante aos diagramas de tipos de negócio das outras abordagens. Discutiremos melhor esse assunto na seção 3.3.2.

Da mesma forma que classes possuem *atributos*, alguns autores (CHEESMAN e DANIELS, 2001) (D'SOUZA e WILLS, 1999) (BROWN, 2000) também os utilizam dentro dos tipos de negócio (Figura 3.7). Por serem informações no nível de análise, esses atributos servem apenas para descrever os dados conhecidos e armazenados pelos tipos. Os atributos dos tipos podem ser de tipos primitivos (e.g., *string*, *inteiro*, etc.) ou estruturas de dados pré-definidas (e.g., *endereço*, etc.), além de não possuírem distinção quanto à visibilidade, sendo todos públicos.

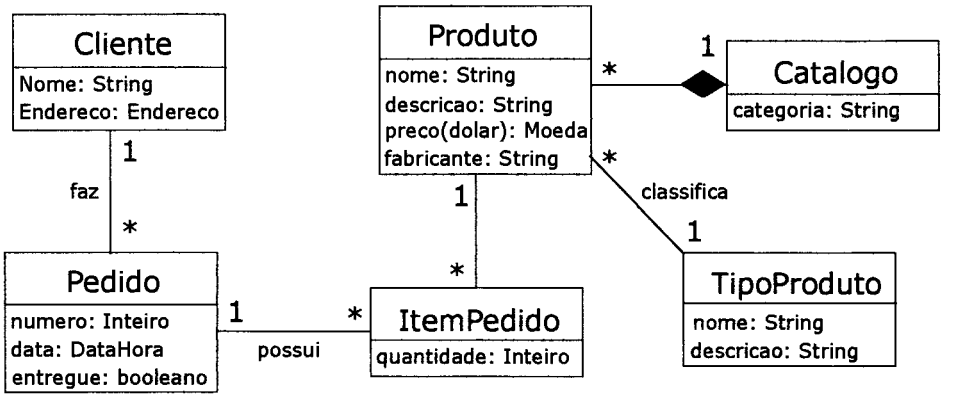


Figura 3.7: Exemplo de diagrama de tipos de negócio.

Apesar de possuírem atributos, os tipos de negócio não costumam possuir *métodos* em DBC. Como vimos acima, os tipos de negócio são apenas conceitos do domínio em que as aplicações estão interessadas. Eles, por si só, não possuem meios de realizar operações dentro na arquitetura. Esta tarefa, como foi visto na seção 2.1, é delegada às interfaces dos componentes (CHEESMAN e DANIELS, 2001).

Por outro lado, existem casos onde um tipo de negócio pode possuir atributos cujo valor depende diretamente de outras informações externas, exigindo a utilização de **atributos parametrizados** (CHEESMAN e DANIELS, 2001) (D'SOUZA e WILLS, 1999). Um atributo parametrizado pode ser visto como um atributo unicamente voltado para *consultas* no tipo de negócio, não sendo possível atribuir valores específicos a eles. Um exemplo desses atributos pode ser visto na Figura 3.7, onde o *preço* de um *produto* depende diretamente da cotação do dólar. A fórmula de cálculo de um atributo parametrizado, normalmente, é especificada utilizando-se a linguagem *OCL* (*Object Constraint Language*), que é uma sintaxe semi-formal adequada para especificação.

3.2.2 – Modelagem Dinâmica

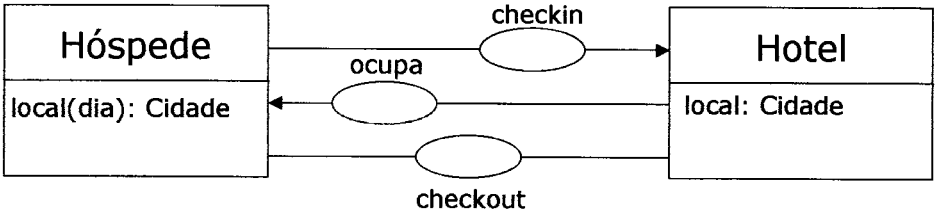
A modelagem estática representa apenas uma parte de todos os aspectos que precisam ser modelados para que o problema do domínio seja entendido. Para completar

esse conhecimento, a modelagem dinâmica é utilizada para explicar a mecânica que existe por trás dos objetos identificados na modelagem estática, detalhando os papéis e as formas de colaboração entre eles. É importante relembrar que a modelagem dinâmica feita na fase de levantamento dos requisitos trata, assim como na modelagem estática, apenas do *conhecimento* do software sobre o domínio em questão. Os aspectos relacionados ao comportamento interno da aplicação (como estados ou interações entre objetos, por exemplo) só serão detalhados na fase de definição da arquitetura, descrita mais adiante (BROWN, 2000).

Muitos dos modelos e diagramas utilizados para descrever os aspectos dinâmicos do domínio são baseados na *UML*. Em particular, os **casos de uso** são os mais utilizados pelas abordagens, pois especificam detalhadamente uma seqüência de ações que o sistema deve tratar para realizar o processo de negócio (CHEESMAN e DANIELS, 2001) (BROWN, 2000) (ATKINSON *et al.*, 2000) (D'SOUZA e WILLS, 1999). Outros diagramas também são empregados para descrever processos de negócio. A abordagem *KobrA* (ATINKSON *et al.*, 2000), por exemplo, utiliza os diagramas de atividades e seqüência para complementar as informações dos casos de uso.

Por outro lado, algumas abordagens adaptaram o contexto de outros diagramas da *UML,* procurando refletir melhor a utilização de tipos de negócio em DBC. Como exemplo, podemos citar as *colaborações* no *Catalysis* (D'SOUZA e WILLS, 1999), que são abstrações dos diagramas de colaboração tradicionais da *UML*. Assim como os casos de uso, seu objetivo é descrever as ações envolvidas entre os tipos de negócio do domínio, apresentando seus *papéis* e *regras* da interação (BROWN, 2000).

Um exemplo de *colaboração* do *Catalysis* está ilustrado na Figura 3.8, onde o tipo *hóspede* colabora com o tipo *hotel* utilizando ações direcionadas e não-direcionadas. A regra da ação *checkout* está documentada em *OCL*, especificando que a localização do hóspede um dia depois da ação é diferente da localização do hotel.




```
inv effect Hospede::checkout_trigger
    post: self.local(agora + 1 dia) <> Hotel.local
```

Figura 3.8: Colaboração do Catalysis.

3.3 – Abordagens Existentes na Literatura

Apresentamos nesta seção as abordagens estudadas para a realização deste trabalho. O objetivo aqui é fornecer uma visão geral sobre as idéias e conceitos descritos na literatura, mostrando variações e particularidades de cada uma delas. Como atualmente existem poucas abordagens para DBC na literatura, utilizamos aquelas que são mais conhecidas e que possuem uma documentação completa para a sua utilização. São elas: *UML Components*, *Catalysis* e *KobrA*.

3.3.1 – UML Components

A abordagem *UML Components* (CHEESMAN e DANIELS, 2001) é um trabalho de extensão da UML para apoiar o desenvolvimento baseado em componentes. Os autores ressaltam que, apesar de muito utilizada ultimamente, a UML original (ERIKSSON e PENKER, 1998) possui uma concepção de componentes com deficiências que impedem a sua adequada aplicação em projetos de software. As razões para isso estão associadas ao fato de que quando ela foi criada e adotada como um padrão para modelagens orientadas a objetos, os componentes ainda estavam começando a surgir no mercado. Assim, sua visão de componentes permaneceu como conceitos de implementação, não oferecendo qualquer suporte aos tão importantes aspectos de especificação e montagem de software a partir de componentes reutilizáveis.

A estratégia desta abordagem concentra-se, principalmente, em considerar os componentes como uma visão arquitetural da análise realizada sobre um problema. Para isso, os componentes adotam os princípios da orientação a objetos (unificação de dados e funções, encapsulamento e identidade), porém, estendendo-os para se adequarem ao papel da interface e das especificações independentes de tecnologia.

O processo descrito pelo *UML Components* é uma adaptação do *RUP (Rational Unified Process)*, onde apenas algumas atividades foram incluídas ou modificadas, como a especificação, o fornecimento e a montagem (as quais substituem, respectivamente, os processos de análise, projeto e implementação originais do RUP).

As adaptações feitas pela abordagem aproveitam as técnicas fornecidas pela própria UML, como a utilização de estereótipos e *OCL*. Além disso, novos diagramas foram criados para suprir as necessidades de modelagem (Tabela 3.1). Os outros

diagramas tradicionais da UML, como o diagrama de casos de uso, também são contextualizados para o desenvolvimento baseado em componentes.

Tabela 3.1: Diagramas utilizados pela abordagem *UML Components*.

	Diagrama	Descrição
Análise	*Business Concept Model Diagram*	Modelo conceitual das informações que existem no domínio do problema.
	Business Type Diagram	Diagrama com os tipos de negócio (e seus relacionamentos) que precisam ser mantidos pelo software.
	Use Case Diagram	Diagrama de casos de uso do software.
Especificação da Arquitetura	*Interface Specification Diagram*	Definição precisa sobre as ações de uma interface, possuindo um modelo de informação, a especificação das operações e as invariantes.
	Component Specification Diagram	Diagrama que descreve as interfaces fornecidas e exigidas de uma especificação de componente.
	Component Architecture Diagram	Diagrama da arquitetura de componentes.
	Interface Responsibility Diagram	Diagrama que mostra o conhecimento de uma interface sobre os elementos de negócio.
	Component Interaction Diagram	Diagrama de interação entre uma especificação de componente e suas interfaces em tempo de execução.

Os diagramas apresentados na Tabela 3.1 abordam a fase de **análise** e **especificação da arquitetura de componentes** do software (a fase de *implementação dos componentes* não está presente porque esta pode ser feita com os diagramas originais da própria *UML*).

O trabalho de especificação de uma arquitetura nesta abordagem merece ser destacado pela quantidade de informação que precisa ser elaborada manualmente pela equipe de desenvolvimento. Além de ser necessário criar todas as operações de todas as interfaces sem o apoio de qualquer regra, o arquiteto precisa elaborar cada um dos cinco diagramas descritos na Tabela 3.1. Dentre eles, o mais trabalhoso é, sem dúvida, o *Component Interaction Diagram*, pois temos que criar um para cada operação especificada.

3.3.2 – Catalysis

A abordagem *Catalysis* (D'SOUZA e WILLS, 1999) é um dos mais completos trabalhos sobre desenvolvimento baseado em componentes já realizados, unificando os conceitos de objetos, *frameworks* e tecnologias de componente. Suas técnicas e métodos oferecem meios precisos para definir interfaces, componentes e conectores que se

relacionam pela arquitetura, mantendo, inclusive, uma rastreabilidade entre as especificações, os objetivos do negócio e o código-fonte dos componentes. Seus princípios são baseados em três conceitos fundamentais:

- *Colaboração*: conjunto de ações entre objetos que atuam com diferentes papéis entre si. As colaborações representam a forma que os objetos se relacionam para alcançarem um objetivo comum, podendo ser generalizadas, isoladas ou compostas para descrever o comportamento do software.
- *Tipos*: são os *tipos* descritos na seção 3.2.1, os quais definem o comportamento externamente visível de um objeto e são completamente independentes de implementação. A notação dos *tipos* no *Catalysis* possui uma pequena particularidade em relação aos outros métodos (conforme destacado anteriormente): eles podem possuir outros *tipos* em seu interior, os quais representam o seu conhecimento sobre o domínio (Figura 3.9).
- *Refinamento*: é o relacionamento entre duas descrições em diferentes níveis de detalhe sobre um mesmo item. Por exemplo, uma implementação de componente é um refinamento de uma especificação com um preciso mapeamento entre as duas visões. A utilização do refinamento faz com que a modelagem seja vista de uma forma mais apropriada e com detalhes suficientes para a realização das atividades, sem perder detalhes relevantes.

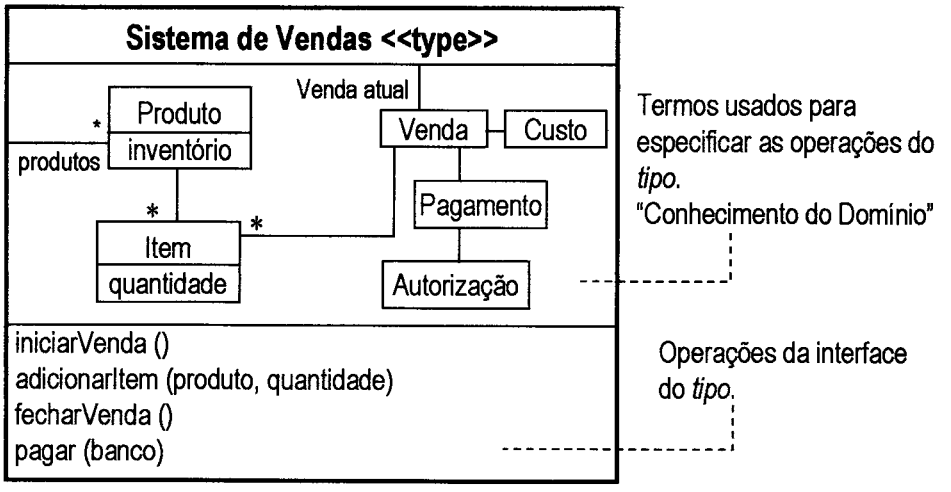
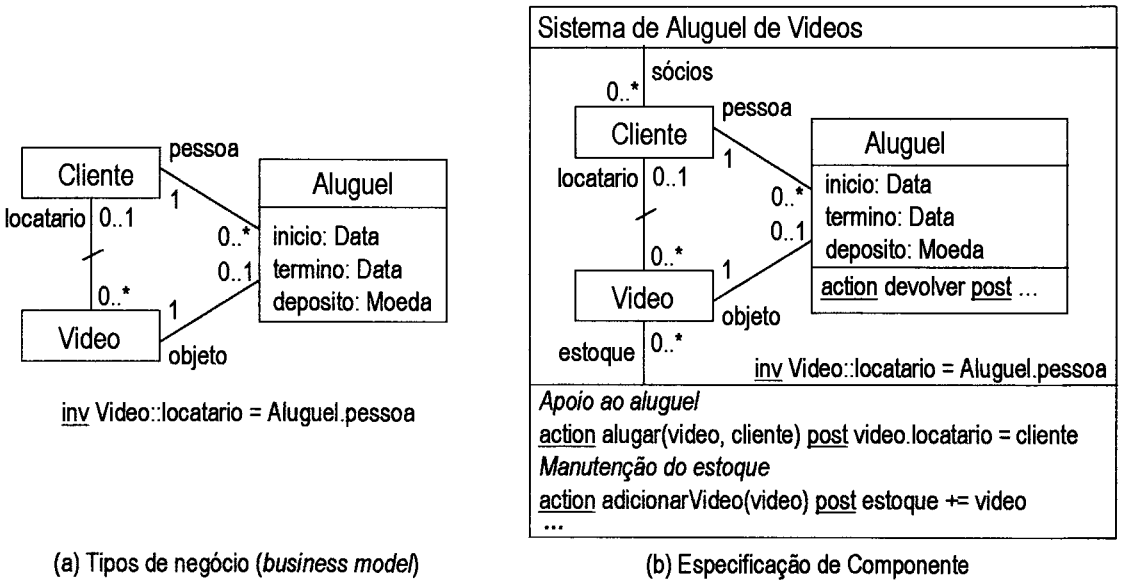


Figura 3.9: Exemplo de tipo de negócio no *Catalysis*.

Para apresentarmos um pouco mais sobre essa abordagem, utilizaremos um exemplo para demonstrar como os requisitos são modelados e aplicados na especificação dos componentes da arquitetura. O exemplo que utilizaremos é o tradicional sistema de aluguel de fitas de vídeo, cuja modelagem de negócios pode ser

vista na Figura 3.10a. Repare que, embora tenhamos falado que um *tipo* possa conter outros em seu interior (representando o seu conhecimento dentro do domínio), os *tipos de negócio*, especificamente, não costumam atender a esta regra. Isto acontece porque eles representam conceitos simples (atômicos) do domínio e possuem, no máximo, atributos que os caracterizam.

Após concluir o modelo de negócio da aplicação, os componentes – que também são *tipos* para essa abordagem – são especificados conforme visto na Figura 3.10b. Ao contrário dos *tipos de negócio*, esses elementos arquiteturais representam, normalmente, vários conceitos ao mesmo tempo, o que torna necessário incluir em seu interior um modelo que explique quais dos *tipos de negócio* são conhecidos e como eles se relacionam no contexto do componente.



(a) Tipos de negócio (*business model*)

(b) Especificação de Componente

Figura 3.10: Tipos de negócio e especificações de componentes no Catalysis.

Para ajudar na reutilização do conhecimento adquirido durante o desenvolvimento, um outro aliado muito importante é aplicado nessa abordagem: os *frameworks*. Apesar de ter o mesmo nome dos *frameworks* descritos na seção 2.2.8 do capítulo anterior, estes elementos, aqui, são modelos genéricos que possuem um conjunto de tipos, propriedades, restrições e colaborações que podem ser reutilizadas para instanciar muitas aplicações.

A idéia de um *framework* é modelar genericamente contextos que são independentes de domínio, mas que as aplicações podem precisar para funcionar, como aluguel de itens, alocação de recursos, estoque de produtos, etc. Desta forma, qualquer

aplicação que precise de alguma destas funcionalidades pode reutilizar os *frameworks* sem precisar remodelar tudo novamente.

3.3.3 – KobrA

A abordagem *KobrA* (ATKINSON *et al.*, 2000) foi, originalmente, criada com o objetivo de tornar os componentes de um sistema de software o foco do processo de desenvolvimento. Além disso, seus princípios adotam uma estratégia de linha de produtos para a criação, manutenção e implantação de componentes. Este método utiliza uma visão de componentes que é semelhante a "*módulos binários*", os quais são descritos em uma representação baseada na UML que expressa suas características e relacionamentos. Esta forma de atuação não só faz com que as atividades de análise e projeto se tornem orientadas a componentes, como também permite que a estrutura e o comportamento dos sistemas possam ser descritos independente de tecnologia.

Por tratar de linha de produtos, a abordagem *KobrA* utiliza o conceito de *framework* genérico e reutilizável que é utilizado para instanciar aplicações. Cada *framework* oferece uma descrição genérica sobre todos os elementos de software que uma família de aplicações deve possuir, incluindo, também, as partes que variam entre elas. Para instanciar as aplicações, um modelo de decisão específico para a família é utilizado para selecionar as funcionalidades desejadas. O resultado é uma aplicação com a mesma forma e estrutura do *framework*, porém, sem os elementos genéricos ou sem relação direta com ela.

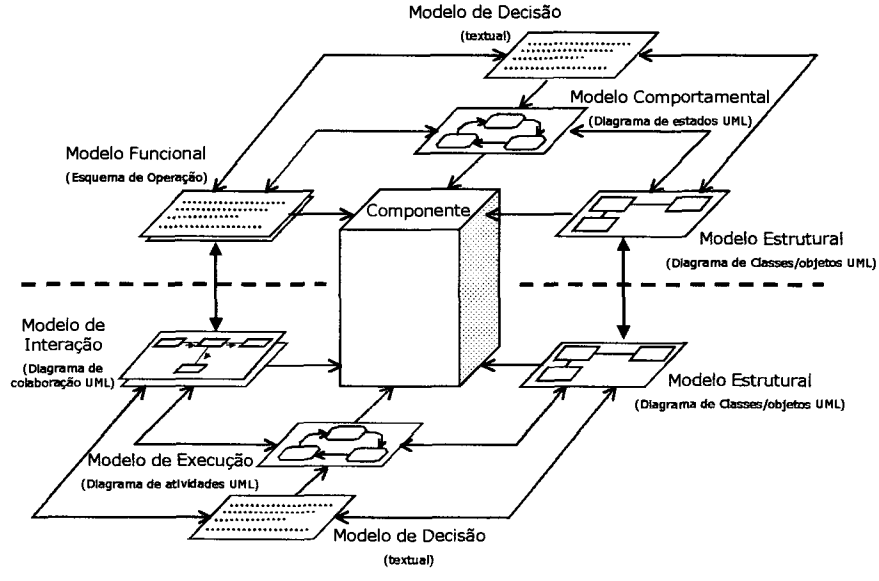


Figura 3.11: Especificações e Realizações de um Componente na abordagem KobrA.

Cada componente nesta abordagem é divido em duas partes (Figura 3.11): a **especificação**, que descreve as características externamente visíveis do componente (i.e., seus requisitos), e a **realização**, que descreve como o componente satisfaz estas características em termos de interações com outros componentes, descrevendo, inclusive, seu projeto interno. O *framework* consiste, portanto, de um conjunto de especificações e realizações inter-relacionadas com um detalhado controle de consistência e rastreabilidade.

3.4 – Conclusões

Dentre as diferentes perspectivas de engenharia de software existentes atualmente, o desenvolvimento baseado em componentes se destaca por possuir um processo focado na reutilização e no desenvolvimento de componentes de software. Essa visão traz novas esperanças aos desenvolvedores que estão cada vez mais pressionados em termos de prazos, custos e recursos permitidos aos seus projetos.

Visando entender melhor essas novas tendências, estudamos neste capítulo como o processo de desenvolvimento baseado em componente é caracterizado, destacando seus objetivos e atividades executadas. Estudamos, ainda, como a modelagem de análise é feita pelas abordagens, descrevendo suas principais diferenças e semelhanças. Finalmente, estudamos as três abordagens de DBC mais conhecidas, apresentando suas estruturas, propósitos e modelos empregados no desenvolvimento de software.

A proposta que se inicia no capítulo seguinte emprega grande parte dessas idéias trazidas pela literatura, buscando aproveitar esse conteúdo que já foi aprimorado por diversas pesquisas dessa área. Em uma análise final, podemos adiantar que as seguintes semelhanças entre as abordagens de DBC possuem maior destaque para este trabalho:

a. A utilização de **tipos de negócio** (com **atributos**) e seus diagramas (BTDs) para modelar aspectos estáticos do software (na abordagem *KobrA*, esses diagramas são chamados de *Context Realization Class Diagram*);
b. A utilização de **casos de uso** para modelar aspectos dinâmicos do software (Na abordagem *Catalysis*, os casos de uso equivalem às *ações* presentes nas colaborações);

Considerando o processo de DBC (Figura 3.5), a proposta se enquadra na atividade de *definição da arquitetura*, onde os componentes são especificados independente de tecnologia.

Capítulo 4 – Geração de Componentes de Negócio

4.1 – Introdução

O desenvolvimento e reutilização de componentes de software são atividades que exigem muito trabalho de suas equipes. Podemos observar isso pela quantidade de documentação que precisa ser gerada, tanto para se especificar componentes e interfaces, quanto para se elaborar modelos que descrevem a arquitetura. É certo que algumas destas tarefas podem até ser executadas com o apoio de algumas ferramentas comerciais[5] disponíveis para as equipes, visto que muitos conceitos do desenvolvimento baseado em componentes estão presentes em outras abordagens baseadas na orientação a objetos (SZYPERSKI, 1998). Entretanto, quando analisamos essas ferramentas de modelagem disponíveis no mercado, percebemos que a sua grande maioria não é especificamente *centrada* em componentes (HERZUM e SIMS, 2000). Isto torna a realidade de se desenvolver e trabalhar com componentes muito mais árdua, aumentando seus riscos e desestimulando sua adoção em projetos de software.

Quando observamos esses aspectos sob o ponto de vista dos desenvolvedores de componentes, somos capazes de entender a importância de termos meios rápidos e eficientes de se construir e especificar componentes para o mercado. Idealmente, a construção de componentes deveria ser feita por equipes que não precisassem se tornar *gurus* nesses aspectos técnicos. Assim, ferramentas automatizadas de construção de software estão se revelando cada vez mais fundamentais atualmente (BROWN, 2000).

Para avançarmos nesse objetivo, é essencial que o desenvolvimento de componentes se torne tecnicamente mais simples. Para isso, o custo por componente individual precisa ser reduzido, e isso não dependerá somente da disponibilidade de tecnologias apropriadas, mas, principalmente, de um conjunto de princípios arquiteturais que apóie o desenvolvimento destes componentes.

HERZUM e SIMS (2000) destacam que um resultado muito atrativo para a indústria seria uma ferramenta de modelagem que apoiasse a *criação*, *especificação* e *implementação* de componentes a partir da perspectiva de análise de negócio. Essa

---

[5] Como o Rational Rose (RATIONAL, 2003), ModelMaker (MODELMAKER, 2003), etc.

proposta tiraria a preocupação com detalhes internos dos componentes, permitindo que as organizações se concentrassem somente nos seus *serviços* e *aspectos funcionais*.

Apesar de estarmos discutindo sobre tendências e necessidades do desenvolvimento baseado em componentes, podemos nos perguntar e refletir porque esses aspectos ainda persistem quando existem abordagens específicas sobre esse assunto na literatura, conforme apresentado no capítulo anterior.

O problema das atuais abordagens de DBC sobre esse assunto, segundo a visão abordada neste trabalho, pode ser vista como uma conseqüência da introdução de novas idéias sobre a maneira com a qual estávamos acostumados a desenvolver software. Isto fez com que as abordagens de DBC se preocupassem mais com seus conceitos e notações, do que como efetivamente deveríamos aplicá-los para obter bons resultados na prática.

Voltando nosso foco para a montagem da arquitetura de componentes, as abordagens de DBC apresentam deficiências com relação à presença inexpressiva de regras ou princípios que conduzam as decisões da equipe de desenvolvimento durante esta fase. Em termos gerais, isso significa que as abordagens delegam praticamente todas as decisões envolvidas na construção da arquitetura para os projetistas e arquitetos de software, como se fosse uma tarefa simples. Essa forma de trabalho revela-se inadequada por exigir da equipe decisões que demandam tempo e conhecimentos que nem sempre são intuitivos.

Foi pensando nesse problema que a proposta que descrevemos nesse capítulo procura trazer algumas contribuições para a especificação de componentes de negócio em projetos de software. A proposta deste trabalho parte do princípio de que é possível construir esses componentes a partir dos elementos sintáticos definidos na modelagem de negócio. Apesar das regras e mecanismos ajudarem muito a equipe de desenvolvimento, é importante lembrar que o papel do arquiteto continua sendo muito importante no projeto da arquitetura. Isto porque ainda contamos com suas decisões para especificar os componentes, porém em um nível de escolha um pouco mais simples e menos trabalhoso.

A proposta está apresentada da seguinte forma: na **seção 4.2** detalhamos como é possível gerar e conectar os componentes de negócio nos dois estilos arquiteturais vistos no capítulo anterior. Na **seção 4.3** descrevemos uma possível maneira de escolhermos entre os dois estilos utilizados, aproveitando o trabalho desenvolvido por XAVIER (2001). Finalmente, concluímos o capítulo na **seção 4.4**.

4.2 – Geração e Conexão em Diferentes Estilos Arquiteturais

A proposta de geração e conexão de componentes de negócio que apresentamos neste capítulo resultou de um estudo detalhado envolvendo as técnicas e métodos de desenvolvimento baseado em componentes que foram descritos no capítulo anterior. Este estudo concentrou-se, principalmente, em identificar e avaliar um mecanismo que pudesse facilitar o trabalho da equipe de desenvolvimento, propondo regras que reduzissem a complexidade das tarefas relacionadas aos componentes de negócio na arquitetura.

Como vimos anteriormente, o desenvolvimento baseado em componentes inicia a construção dos componentes e montagem da arquitetura logo após a fase de análise de requisitos. Isto significa uma repentina transição de modelos em alto nível de abstração para uma detalhada arquitetura com componentes e interfaces. Por não haver passos intermediários nesse procedimento, o único caminho capaz de gerar componentes de negócio sem alterar o processo de DBC consiste em traduzir elementos sintáticos presentes nos modelos de análise em artefatos arquiteturais, como especificações de componentes, interfaces, métodos, parâmetros, etc.

Por existirem, até o momento de término deste trabalho, dois estilos arquiteturais para componentes de negócio na literatura (estilo **baseado em tipos** e **baseado em instâncias**), este trabalho abrange estes estilos com dois conjuntos diferentes de regras e mecanismos. A diferença entre os estilos exige que essa separação seja feita, permitindo que cada um deles seja gerado dentro de suas próprias características e particularidades. De uma forma ou de outra, o arquiteto do software terá as duas opções a sua disposição, tendo apenas que escolher entre uma delas (Figura 4.12).

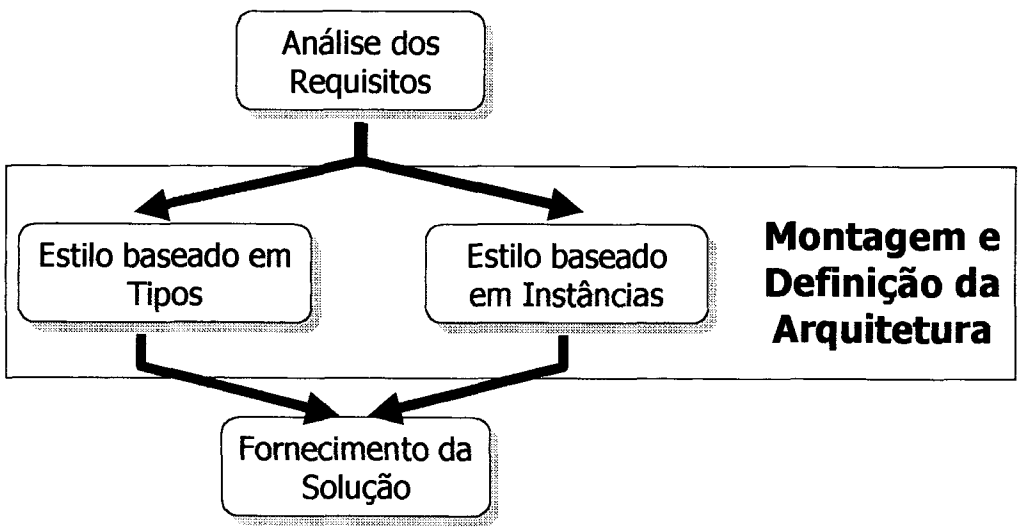


Figura 4.12: Dois estilos arquiteturais para componentes de negócio.

É importante lembrar que o DBC trabalha apenas com *especificações de componentes* durante a definição e montagem da arquitetura do software. Portanto, a geração proposta por este trabalho concentra-se apenas nos **serviços**, **responsabilidades** e **interfaces** dos componentes de negócio, não oferecendo suporte ou meios de implementá-los em tecnologias específicas. Como vimos no capítulo anterior, caso a organização não deseje implementar essas especificações, é possível terceirizar esse desenvolvimento.

Como a modelagem dos requisitos do software possui muitos elementos inter-relacionados, além de gerar as especificações de componente, é possível *conectá-los* segundo essas associações presentes na modelagem. Entretanto, a conexão dos componentes que propomos nesse capítulo depende diretamente do estilo arquitetural (para componentes de negócio) que estamos lidando, possuindo total compatibilidade com seus princípios e regras.

Apresentamos a seguir como a geração e conexão desses componentes podem ser realizadas nos dois estilos estudados, descrevendo e observando cada uma das regras que compõem esse mecanismo. Além desses aspectos, discutiremos também outros importantes pontos referentes à remoção e integridade referencial na arquitetura. Independente do estilo usado, o conjunto de componentes gerados é capaz de atender a todos os requisitos especificados, podendo, inclusive, ser reutilizado em mais de uma aplicação.

4.2.1 – Estilo Baseado em Tipos

Como apresentado no segundo capítulo, o *estilo baseado em tipos* é caracterizado por componentes que gerenciam instâncias de tipos de negócio (também chamados de ***componentes gerentes de instâncias***), encapsulando-as para que ninguém mais tenha controle sobre essas informações na arquitetura. Assim, qualquer operação que precise ser realizada sobre uma determinada informação de negócio deve ser feita através da interface do seu componente responsável, o qual irá pessoalmente acessar as instâncias desejadas e operar sobre elas. É importante lembrar que a identificação das instâncias é feita por *chaves técnicas* da arquitetura, que são números identificadores que não se repetem na execução do sistema.

As chaves técnicas na arquitetura de componentes possuem um papel muito importante para o estilo baseado em tipos, uma vez que elas desacoplam os componentes e oferecem mais segurança no controle das suas informações. Por outro

lado, sua utilização **polui** bastante as interfaces dos componentes gerentes de instâncias criados, inserindo muitos *Ids* que tornam difícil sua manipulação por parte da equipe de desenvolvimento.

Foi pensando nesse problema que empregamos, neste trabalho, uma outra categoria de componente para apoiar a utilização dos componentes gerentes de instância na arquitetura - os ***componentes de processo***. Vimos no segundo capítulo que o propósito dos componentes de processo é controlar as atividades de negócio apoiadas pela aplicação, gerenciando os recursos envolvidos nesse trabalho, como, por exemplo, itens de negócio, pessoas, máquinas, etc.

Apesar desta abrangência, a literatura aproveita muito pouco o potencial dos componentes de processo no desenvolvimento (HERZUM e SIMS, 2000) (CHEESMAN e DANIELS, 2001), não os utilizando para apoiar os componentes gerentes de instâncias criados, além de deixá-los, muitas vezes, misturados nas funcionalidades destes últimos. Esta forma de trabalho, além de aumentar a granularidade dos componentes desenvolvidos, revela-se pouco vantajosa por não eliminar as chaves técnicas das suas interfaces.

No contexto deste trabalho, todos os componentes de processo, além de apoiarem a execução das atividades de negócio, são utilizados para *facilitar o manejo* das chaves técnicas dos componentes gerentes de instância. Como as informações necessárias para a execução dos processos de negócio são gerenciadas pelos componentes gerentes de instâncias, os componentes de processo podem acessar estes últimos transparecendo a complexidade das chaves técnicas para seus clientes. Dessa forma, eles agem como "*tradutores*" para os outros componentes da arquitetura, os quais desejam executar e obter informações de negócio, mas não precisam entender como essas informações são mantidas. A Figura 4.13 ilustra esse funcionamento.

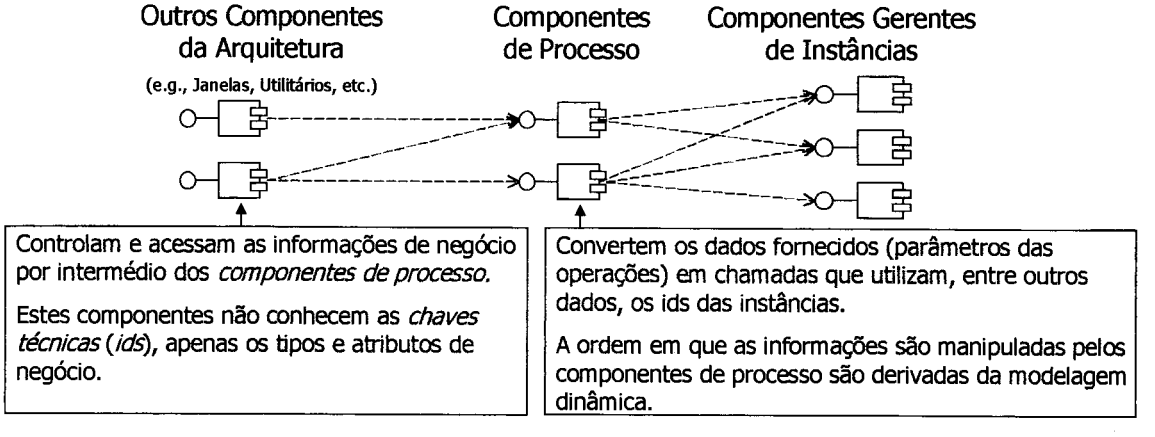


Figura 4.13: Acesso aos componentes de processo.

Tendo em vista a estratégia acima, apresentamos, a seguir, como os dois tipos de componentes podem ser gerados e conectados para formar a arquitetura baseada em tipos. Apresentamos, inicialmente, os componentes gerentes de instâncias, para que, posteriormente, possamos utilizá-los na geração dos componentes de processo.

4.2.1.1 – Geração de Componentes Gerentes de Instâncias

A geração dos componentes gerentes de instância é feita utilizando-se os *diagramas de tipos de negócio* criados na fase de análise de requisitos (vide capítulo anterior). Estes diagramas, por representarem a modelagem estática do software, descrevem como as informações de negócio precisam ser mantidas e estruturadas durante a execução dos sistemas. Deste modo, a forma com que os componentes gerentes de instâncias são construídos *depende* diretamente de como essas informações estão modeladas.

Como o próprio nome diz, os componentes gerentes de instâncias *gerenciam* as instâncias dos tipos de negócio na arquitetura, encapsulando-as e controlando o acesso aos seus dados. Como existem tipos de negócio muito relacionados entre si, é possível que um mesmo componente gerencie grupos destes elementos ao mesmo tempo na arquitetura (BROWN, 2000) (CHEESMAN e DANIELS, 2001). Essa união de serviços é uma estratégia voltada para a redução da complexidade do desenvolvimento, uma vez que, em sistemas de grande porte, manter um componente gerente de instâncias para cada tipo de negócio é algo quase impraticável.

Além da redução da complexidade, a utilização de componentes responsáveis por grupos de tipos de negócio possui outras vantagens importantes. Uma delas diz respeito a ***adequabilidade*** dos componentes. Agrupar tipos certos em componentes certos pode fazer a diferença durante a montagem da arquitetura, tornando estes últimos *menos acoplados* e *mais propícios à reutilização*. Isto acontece porque cada componente passa a ser visto como uma peça bem elaborada dentro da arquitetura, prestando um serviço que é mais interessante para o domínio do que um simples gerente de objetos.

No contexto deste trabalho, portanto, o primeiro passo que devemos executar consiste em dividir a arquitetura em grupos de tipos de negócio que serão administrados por componentes separadamente. A forma usada para agrupar os tipos precisa ser cuidadosamente estudada, para que, assim, os componentes resultantes tenham mais chances de serem aproveitados em outros projetos. Vimos, anteriormente, que as atuais

abordagens não oferecem regras ou conselhos que ajudem o arquiteto durante esse agrupamento, exigindo-lhes experiência e dando margem a erros.

Para esta tarefa de divisão em grupos, desenvolvemos um conjunto de regras que servem como ferramenta de apoio para as decisões do arquiteto do software. Essas regras utilizam os elementos presentes na notação do *diagrama de tipos de negócio (ou BTD – Business Type Diagram)*, para indicar tipos que devem ser mantidos juntos em um mesmo componente.

Regras de Agrupamento de Tipos de Negócio

A primeira regra que deve ser aplicada ao BTD para iniciar o processo de agrupamento dos tipos de negócio está relacionada ao conceito de *herança*. Utilizamos esse relacionamento comum da orientação a objetos para formar os primeiros grupos que servirão como base para a aplicação das outras regras. Em linhas gerais, a ***regra da herança*** atua transformando todas as estruturas de herança existentes no BTD em grupos inseparáveis de tipos de negócio (Figura 4.14).

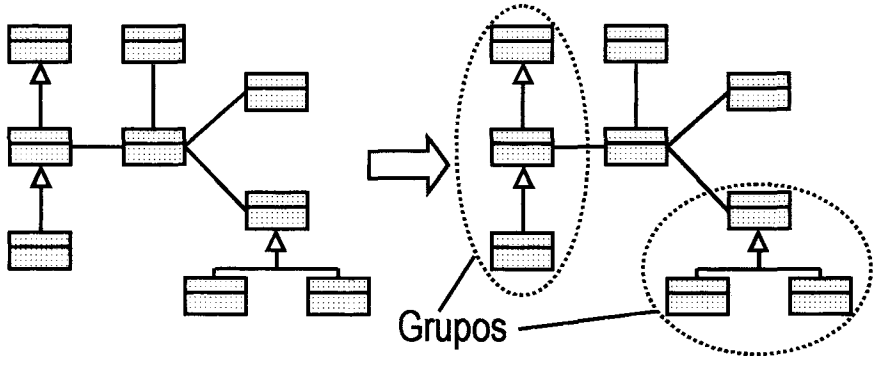


Figura 4.14: Aplicação da regra da herança.

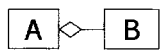
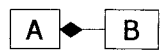
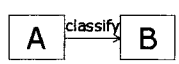
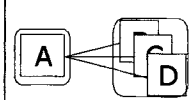
Dentre todas as regras que apresentamos nesse trabalho, esta é a única que permanece inviolável para todos os casos. O motivo por trás desta decisão se baseia em um princípio fundamental dos componentes de software: *todo componente deve ser autocontido* (SZYPERSKI, 1998) (BROWN, 2000) (D'SOUZA e WILLS, 1999). Portanto, não faz sentido um componente controlar um tipo de negócio sem conhecer os seus *supertipos* e *subtipos*.

Após a aplicação da *regra da herança*, outras regras são usadas para alcançar os tipos de negócio que ainda se encontram sem grupos no BTD. Como o objetivo deste trabalho é *apoiar* as decisões sobre a construção dos componentes, as regras restantes são tratadas apenas como *sugestões* para a continuação do agrupamento, podendo ou não ser adotadas pelo arquiteto do software.

As regras que são aplicadas após a regra da herança estão descritas na Tabela 4.2. Antes de aplicá-las, cada tipo de negócio que tenha ficado sem grupo é inserido em um novo grupo vazio, permanecendo sozinho enquanto o arquiteto decide o que fazer com ele. Para guiar as ações do arquiteto, as regras aplicadas mostram como os tipos de negócio no BTD tendem a permanecer juntos de outros tipos durante a fase de montagem da arquitetura.

Em sua essência, as regras da Tabela 4.2 descrevem relacionamentos que carregam uma semântica mais forte entre os tipos de negócio modelados, permitindo ao arquiteto identificar pontos onde a concepção de um componente é mais favorável dentro do contexto de negócio. Na área de banco de dados existem regras semelhantes que contribuíram para o levantamento desse trabalho (CHEN, 1976) (NAVATHE, 1999).

Tabela 4.2: Regras para formação de candidatos ao agrupamento.

Regra	Visualização	Descrição	Papel no Agrupamento
Agregação		Um tipo de negócio A (todo) agrega tipos de negócio B (partes).	B é um candidato FORTE para fazer parte do grupo de A.
Composição		Um tipo de negócio A (todo) é composto por tipos de negócio B (partes).	B é um candidato FORTE para fazer parte do grupo de A.
Associação Classificadora		Um tipo de negócio A classifica um outro tipo de negócio B (Estereotipo «*classify*»).	A é um candidato FRACO para fazer parte do grupo de B.
Isolamento Relacional		Todas as associações de um tipo de negócio A (que se encontra sozinho em seu grupo) são para tipos de negócio que estão dentro de um mesmo grupo (B, C e D).	A é um candidato FRACO para fazer parte do grupo com o qual está se relacionamento (grupo de B, C e D).

A atividade de agrupamento funciona da seguinte forma: cada regra indica um tipo de negócio como sendo um *candidato* a entrar em um determinado grupo. A decisão de introduzir o tipo no grupo indicado pela regra depende exclusivamente da vontade do arquiteto do software. Para facilitar essa decisão, dois tipos de candidatos são destacados: os *candidatos fortes* e os *fracos*.

Os *candidatos fortes* são aqueles que apresentam um relacionamento mais forte com outros tipos de negócio no BTD (no caso, através de agregação ou composição).

Todo candidato forte representa um conselho bastante recomendado que os arquitetos devem aproveitar. Separar candidatos fortes de seus grupos indicados pode desfavorecer a arquitetura e criar componentes complexos e com muitas dependências. Apesar desses riscos, podem existir casos onde o melhor é deixá-los separados, cabendo a equipe de desenvolvimento avaliar sobre a situação.

Os *candidatos fracos* representam um conselho um pouco menos enfático do que os candidatos fortes no agrupamento. Seu objetivo é indicar uma oportunidade de melhorar os grupos com o intuito de deixá-los mais completos e coesos. Caso o arquiteto não deseje incluir o candidato fraco no grupo indicado, o impacto sobre a arquitetura não é tão acentuado, o que ainda mantém os componentes bem elaborados.

Dentre as regras apresentadas na Tabela 4.2, apenas uma exigiu uma adaptação da *UML* para se adequar melhor ao contexto deste trabalho: a *associação classificadora.* Essa associação é uma variação das associações tradicionais, focada em uma situação que costuma ocorrer freqüentemente em sistemas de negócio. Esta situação ocorre quando um determinado tipo de negócio **classifica** outro no contexto do sistema. Para destacar esta regra utilizamos uma seta com o estereotipo *«classify»*, aproveitando o mecanismo de extensão oficial permitido pela UML. Um exemplo desta regra está ilustrado na Figura 4.15.

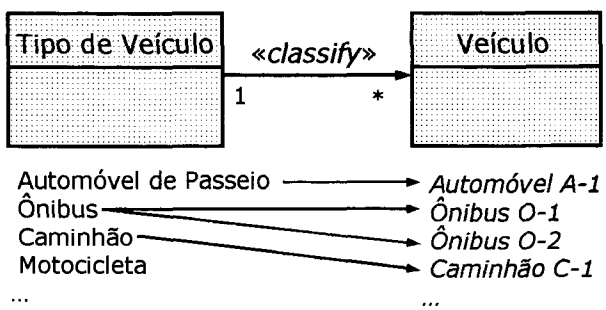


Figura 4.15: Exemplo de associação classificadora.

O agrupamento dos tipos de negócio na arquitetura não dependem somente das sugestões oferecidas pelas regras da Tabela 4.2. Outros fatores muito importantes também influenciam essa divisão, exigindo do arquiteto mais cuidado e planejamento durante esta atividade. Alguns dos aspectos identificados neste trabalho que precisam ser levados em consideração são:

- Busca de Componentes: Vimos no capítulo anterior que o processo de desenvolvimento baseado em componentes prevê a busca de componentes em repositórios. Dependendo dos resultados dessa busca, a forma com que os tipos de negócio são agrupados é afetada. Por exemplo, se encontrarmos

em um repositório um componente que gerencia um conjunto específico de tipos de negócio para o domínio que estamos lidando, teríamos apenas o trabalho de criar outros componentes para gerenciar os tipos que ficaram de fora, considerando que em muitos casos não podemos alterar o componente encontrado no repositório (SZYPERSKI, 1998) (BACHMAN *et al.*, 2000).

- Desempenho: Criar muitos componentes que só gerenciam um tipo de negócio pode tornar a arquitetura lenta e com muitas dependências. Isto acontece porque os tipos de negócio são dependentes entre si, exigindo que os componentes de processo acessem muitos outros componentes durante a execução de uma única operação.
- Distribuição: A distribuição dos componentes pela rede também afeta o agrupamento dos tipos de negócio. Isto acontece porque diferentes nós podem precisar apenas de alguns dos tipos gerenciados pela arquitetura, exigindo que o arquiteto balanceie os componentes de acordo com essas variações. Manter um componente que gerencia muitos tipos, onde cada tipo é exigido em um ponto diferente da rede, pode tornar muito ineficiente o acesso aos dados e a distribuição do processamento.

Além do agrupamento dos tipos de negócio no BTD, as especificações dos componentes gerentes de instâncias necessitam de mais dois ingredientes para se tornarem completas: os *tipos de dados armazenados* e as *interfaces de acesso ao serviço*. A seguir, descrevemos cada um desses tópicos e, no final desta seção, apresentamos em exemplo completo sobre a geração dos componentes gerentes de instâncias para o domínio de hotelaria.

Geração dos Tipos de Dados Armazenados

Todos os componentes gerentes de instância que são desenvolvidos precisam armazenar e controlar corretamente as informações dos tipos de negócio que lhes foram destinados. Para isso, é fundamental documentarmos uma estrutura de dados que descreva quais campos cada instância precisa manter para existir segundo as regras de negócio do software. Além de facilitarem o entendimento sobre os componentes, essas estruturas auxiliam tanto a tarefa de elaboração quanto a implementação das operações nas interfaces.

Para gerarmos as estruturas de dados de cada tipo de negócio, precisamos observar como as informações estão modeladas no BTD. Como estamos utilizando o estilo baseado em tipos, é inevitável que o primeiro campo de toda estrutura seja a sua chave técnica, isto é, um campo *Id* para identificação única. Para permitir mais flexibilidade sobre este campo, utilizamos nesta proposta dois possíveis tipos de dados para ele: *INTEIRO* ou *STRING*. Assim, o arquiteto pode escolher, além da numeração única global (GUID), qualquer outro formato para a identificação das instâncias (como, por exemplo, *A01-0001*, *X01-01-001*, *F11202-0*, etc.). Qualquer que seja esse formato, o arquiteto só terá o trabalho de especificar um único componente (se já não possuir um pronto) que gere os *Ids* para as instâncias, reutilizando-o em todos os gerentes criados.

Os outros campos da estrutura são deduzidos diretamente a partir do BTD. Primeiramente, cada *atributo* do tipo de negócio deve ser trazido para a estrutura, garantindo o correto mapeamento dessas informações para o software. No caso de *atributos parametrizados*, os campos devem ser marcados como *somente-leitura* (*read-only*), lembrando, assim, que esses valores são calculados e não devem existir operações nas interfaces que os atribuam valor. Para tipos de negócio que herdam de outros tipos, todos os atributos dos *supertipos* devem ser trazidos para a sua estrutura, deixando-a auto-suficiente e completa para o gerenciamento.

Além dos atributos, as associações entre as instâncias precisam ser guardadas corretamente. Para isso, precisamos observar cautelosamente quais tipos se relacionam com o tipo que estamos lidando, verificando as multiplicidades e a navegabilidade entre eles. Apesar do uso de multiplicidades ser unânime em todas as abordagens estudadas (veja o capítulo 3), o uso da navegabilidade no BTD não é tão comum assim. Dentre todas, apenas o método *Catalysis* (D'SOUZA e WILLS, 1999) permite que este direcionamento entre as associações seja explorado na modelagem (no caso, em *OCL*).

Sobre este assunto, é importante lembrar que as abordagens deixam grande parte das decisões de projeto para o arquiteto, inclusive as questões relacionadas à navegabilidade entre os tipos. Por não se tratar de um problema específico do domínio, a navegabilidade é vista como uma decisão voltada para a implementação do software, podendo possuir diferentes direções para diferentes contextos. Como neste trabalho estamos concentrados na geração dos componentes de negócio, é fundamental termos essa navegabilidade exposta no BTD. Caso contrário, não seríamos capazes de gerar os artefatos propostos.

Deste modo, utilizamos as multiplicidades e a navegabilidade para criarmos os campos que representam as associações entre os tipos de negócio. Como no estilo baseado em tipos não é permitido manter um endereçamento direto entre as instâncias, todos os campos criados são referências para as chaves técnicas das outras estruturas, isto é, para seus *Ids*. Existem três possibilidades de campos criados a partir das associações (Figura 4.16):

A. Navegável para 1 (uma) instância: Quanto um tipo de negócio é navegável para somente uma instância de outro tipo de negócio, o campo criado em sua estrutura é uma simples variável para guardar o Id desta instância. O tipo desta variável pode ser *inteiro* ou *string*, conforme a decisão do arquiteto em relação ao formato da chave técnica da arquitetura. Nos casos onde a navegabilidade permite zero instâncias (0..1), o campo da estrutura deve permitir o valor nulo.

B. Navegável para * (inúmeras) instâncias: Quando um tipo de negócio é navegável para inúmeras instâncias de outro tipo de negócio, utilizamos em sua estrutura um campo que representa uma lista de *Ids* para guardar estas instâncias. Como estamos fazendo uma especificação independente de tecnologia, não entramos em detalhes sobre como esta lista será implementada no componente. Esta decisão será feita pelo arquiteto quando a arquitetura for mapeada para a implementação, podendo ele utilizar um componente já implementado ou fornecido pela própria linguagem de programação utilizada. O único ponto que deve ser documentado junto à especificação do componente é o formato que deve ser armazenado pela lista (*inteiro* ou *string*).

C. Navegável para um número fixo (k, onde k > 1) instâncias: Quando um tipo de negócio é navegável para um número fixo de instâncias (onde este número é maior do que um), também utilizamos uma lista para guardar os *Ids* das instâncias. Porém, neste caso temos que deixar explícito na especificação do componente que a capacidade da lista é fixa, não podendo guardar mais instâncias do que o permitido.

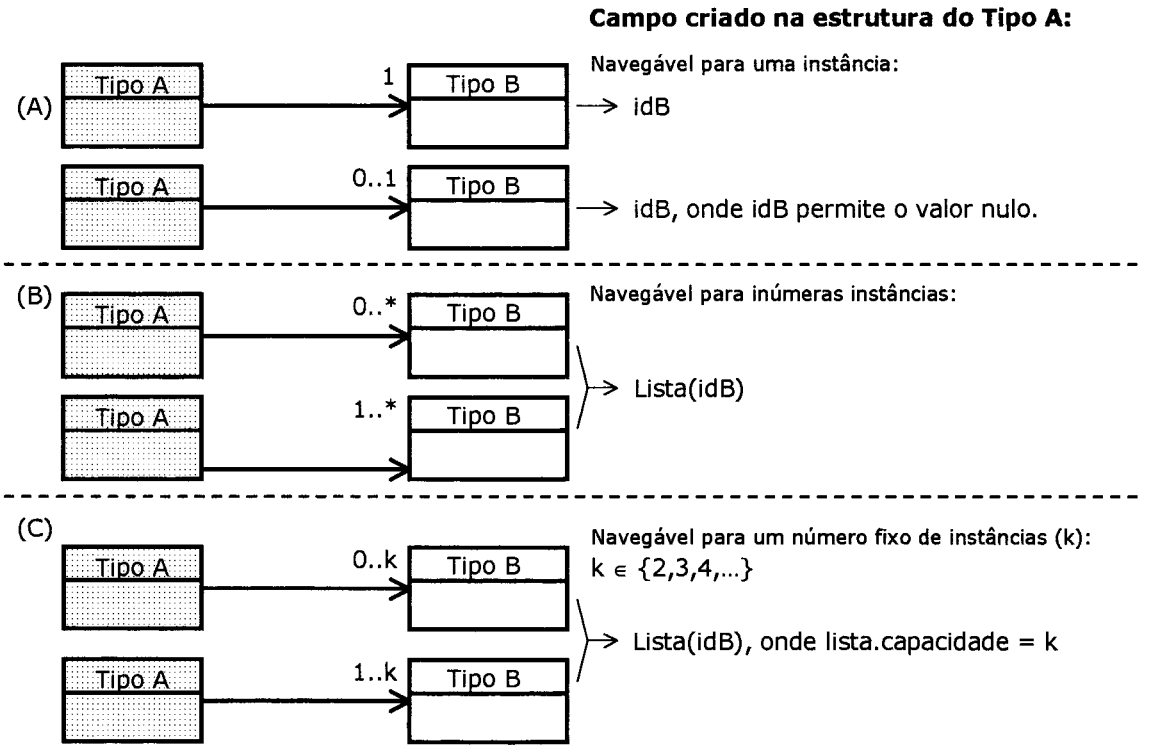


Figura 4.16: Campos criados a partir das associações.

Geração das Interfaces

Após a criação das estruturas de dados que serão usadas para armazenar as instâncias dos tipos de negócio nos componentes gerentes de instâncias, podemos iniciar a elaboração das *interfaces* que acessam e manipulam essas informações. O serviço prestado pelas interfaces destina-se ao gerenciamento completo dos tipos de negócio, permitindo que as instâncias possam ser criadas e removidas, e seus atributos consultados e alterados.

Para gerarmos as interfaces de gerenciamento dos tipos, devemos consultar as suas estruturas de dados para saber quais operações podem ser criadas. Apesar de termos quase todas as informações que precisamos presentes nessas estruturas, outras informações ainda precisam ser fornecidas pela equipe para suprir alguns parâmetros que não são dedutíveis a partir do BTD. Essas informações auxiliares estão relacionadas somente ao negócio que estamos lidando, podendo ser obtidas em reuniões e entrevistas com os especialistas no domínio. Mostraremos com mais detalhes que informações são estas quando explicarmos a criação de cada operação mais adiante.

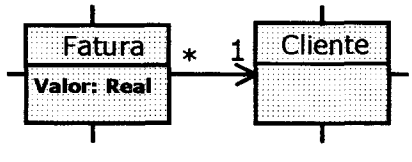
Por tratarmos de componentes responsáveis pelo gerenciamento das instâncias dos tipos de negócio, utilizamos o sufixo *Manager* para complementar o nome de cada interface criada. A letra *"I"* também é utilizada como prefixo por ser comum tanto em

sistemas orientados a objetos quanto no desenvolvimento baseado em componentes. Portanto, cada interface é nomeada da seguinte forma: *I<TipoNegócio>Manager*. Como exemplos, podemos citar *IFaturaManager*, *IReservaManager*, etc.

Apesar de algumas abordagens (CHEESMAN e DANIELS, 2001) e tecnologias (MOWBRAY e RUH, 1997) permitirem que as operações de suas interfaces contenham parâmetros de entrada e saída, este trabalho utiliza somente os parâmetros de entrada nas operações geradas. O único parâmetro de saída utilizado é o retorno da operação. Como pretendemos especificar componentes independentes de tecnologia, não podemos utilizar recursos que só estejam presentes em algumas delas.

A primeira operação que criamos na interface de gerenciamento de um tipo de negócio é aquela responsável pela **criação** das instâncias em tempo de execução. Como a criação de um tipo de negócio depende de diversas informações que variam de domínio para domínio, a geração desta operação exige que esses parâmetros sejam levantados com o especialista durante a fase de análise dos requisitos. Existem dois possíveis conjuntos de informações que podem ser usados como parâmetros de criação: os atributos do tipo de negócio e as chaves técnicas de outros tipos de negócio com os quais o tipo criado mantém associações. Apresentamos na Tabela 4.3 os detalhes e um exemplo desta operação.

Tabela 4.3: Operação de Criação de Instâncias.

Operação:	`create<TipoNegócio>`
Parâmetros:	Os parâmetros podem ser atributos do tipo de negócio criado ou *Ids* de outras instâncias. Neste último caso, só deve ser permitido utilizar instâncias de tipos de negócio que possuem associação com o tipo criado.
Retorno:	*Id* da instância criada.
Exemplo:	Este exemplo mostra um método para criar faturas, onde o primeiro parâmetro é o valor da fatura e o segundo, o *Id* da instância do cliente. O método retorna o *Id* da fatura criada (supondo que o arquiteto tenha escolhido a chave técnica como número *inteiro*).  `IFaturaManager.createFatura(valor: Real, id_cliente: Integer): Integer`

A segunda operação gerada para todos os tipos de negócio é a de **exclusão** de instâncias. Ao contrário da operação de criação, esta operação só possui um único parâmetro que é igual para todos os casos: o Id da instância que será apagada. O retorno

desta operação é um valor *booleano* que indica se a instância foi excluída com sucesso. No contexto deste trabalho, não criamos operações para consultar ou manipular instâncias que já foram excluídas pelo componente. Portanto, é recomendado que este método *exclua fisicamente* as instâncias da memória. Apresentamos na Tabela 4.4 os detalhes e um exemplo desta operação.

Tabela 4.4: Operação de Exclusão de Instâncias.

Operação:	`delete<TipoNegócio>`
Parâmetros:	*Id* da instância que será apagada.
Retorno:	*Booleano* que indica se a instância foi excluída com sucesso.
Exemplo:	Este exemplo mostra um método para excluir faturas. `IFaturaManager.deleteFatura(id_Fatura: Integer): Boolean`

Para que seja viável a utilização de chaves técnicas para identificar as instâncias na arquitetura, precisamos ter operações que convertam *valores de negócio* em *Ids*. Além das operações de consulta que veremos adiante, precisamos ter uma operação que seja capaz de retornar uma única instância para um certo conjunto de valores. Assim como nos bancos de dados relacionais (NAVATHE, 1999), precisamos saber como identificar cada tipo de negócio unicamente no domínio em que estamos trabalhando. Para isso, precisamos contar com a ajuda do especialista no domínio, o qual deve fornecer essa informação.

É importante lembrar que o levantamento dessa informação não envolve somente a escolha de alguns atributos como sendo a "*chave primária*" do tipo de negócio. Existem casos onde um tipo só é identificável unicamente quando o identificador de outro tipo está presente. Por exemplo, imagine uma aplicação que precise cadastrar *navios turísticos*, onde cada navio possui *cabines* para seus passageiros. Como poderíamos busca o *Id* da *cabine 101* sem sabermos em qual navio ela se encontra? Desta forma, estes detalhes não podem faltar durante a geração dos componentes.

Uma vez que estas informações de identificação única já foram levantadas, podemos gerar a operação que retorna a instância referente a esses dados. Nos casos onde o tipo de negócio necessita de outro para se identificar, utilizamos o *Id* deste último como parâmetro de entrada para a operação. Apresentamos na Tabela 4.5 os detalhes e um exemplo desta operação.

Da mesma forma que precisamos de uma operação para buscar uma instância em particular, muitos são os casos onde precisamos de uma lista com **todas** as instâncias

existentes. Para realizar este objetivo, geramos um método que retorna uma lista com todos os *Ids* das instâncias gerenciadas. Em particular, esta é a única operação no estilo baseado em tipos que requer o nome do tipo de negócio escrito no *plural*. Esta exigência é necessária para manter uma correta concordância de número no nome da operação, cujo formato é: *getAll<TipoNegócio (no plural)>*. Apresentamos mais detalhes desta operação na Tabela 4.6.

Tabela 4.5: Operação que busca o Id de uma instância.

Operação:	`getId<TipoNegócio>`
Parâmetros:	Os parâmetros podem ser atributos do tipo de negócio criado ou *Ids* de outras instâncias. Neste último caso, só deve ser permitido utilizar instâncias de tipos de negócio que possuem associação com o tipo criado.
Retorno:	*Id* da instância procurada. Caso esta não exista, a operação retorna nulo.
Exemplo:	Este exemplo mostra uma operação para buscar o *Id* de uma cabine de passageiros em um navio. O primeiro parâmetro é o número da cabine e o segundo, o *Id* do navio onde ela se encontra. `ICabineManager.getIdCabine(numero: Integer, id_Navio: Integer): Integer`

Tabela 4.6: Operação de acesso a todas as instâncias.

Operação:	`getAll<TipoNegócio (no plural)>`
Parâmetros:	Não há parâmetros.
Retorno:	Esta operação retorna uma lista com os *Ids* de todas as instâncias existentes para um tipo de negócio.
Exemplo:	Este exemplo mostra uma operação que retorna uma lista com os *Ids* de todas as faturas existentes no momento. `IfaturaManager.getAllFaturas(): List`

Além das operações acima, precisamos de outras capazes de consultar e alterar os atributos das instâncias dos tipos de negócio. Basicamente, temos que observar a estrutura de dados criada para o tipo, gerando métodos de *leitura* e *escrita* para cada campo existente (menos para o campo *id*). Com exceção dos campos que são *listas* (caso B e C da Figura 4.16), as operações de leitura e escrita seguem, respectivamente, o seguinte formato: *get<Campo><TipoNegócio>* e *set<Campo><TipoNegócio>*. A Tabela 4.7 e a Tabela 4.8 descrevem, respectivamente, cada uma dessas operações.

Os campos que são *listas* seguem um formato diferente, já que temos que criar operações que inserem e removem elementos nestas listas. Essas operações estão descritas na Tabela 4.9 e na Tabela 4.10, respectivamente. Para completar esse quadro, a Tabela 4.11 apresenta a operação utilizada para buscar todos os elementos de uma lista.

Tabela 4.7: Operação de leitura de um campo de uma instância.

Operação:	`get<Campo><TipoNegócio>`
Parâmetros:	O único parâmetro é o *Id* da instância consultada.
Retorno:	Esta operação retorna o valor do campo consultado.

Exemplo:	(1) Este exemplo mostra uma operação para consultar o campo *valor* de uma fatura.
	`IFaturaManager.getValorFatura(id_Fatura: Integer): Real`
	(2) Este exemplo mostra uma operação para consultar o Id do cliente de uma fatura.
	`IFaturaManager.getIdClienteFatura(id_Fatura: Integer): Integer`

Tabela 4.8: Operação de escrita em um campo de uma instância.

Operação:	`set<Campo><TipoNegócio>`
Parâmetros:	O primeiro parâmetro é o *Id* da instância alterada. O segundo é o novo valor do campo.
Retorno:	Esta operação retorna um valor *booleano* indicando se houve sucesso na alteração.

Exemplo:	(1) Este exemplo mostra uma operação para alterar o campo *valor* de uma fatura.
	`IFaturaManager.setValorFatura(id_Fatura: Integer, valor: Real): Boolean`
	(2) Este exemplo mostra uma operação para alterar o Id do cliente de uma fatura.
	`IFaturaManager.setIdClienteFatura(id_Fatura: Integer, id_Cliente: Integer): Boolean`

Tabela 4.9: Operação de inserção de elementos em um campo que é uma lista.

Operação:	`add<TipoNavegável><TipoNegócio>`
Parâmetros:	O primeiro parâmetro é o *Id* da instância que possui a lista. O segundo, o *Id* da instância que será inserida na lista.
Retorno:	Esta operação retorna a posição da lista onde houve a inserção.
Exemplo:	Este exemplo mostra uma operação para inserir cabines em um navio. `INavioManager.addCabineNavio(id_Navio: Integer, id_Cabine: Integer): Integer`

Tabela 4.10: Operação de remoção de elementos em um campo que é uma lista.

Operação:	`remove<TipoNavegável><TipoNegócio>`
Parâmetros:	O primeiro parâmetro é o *Id* da instância que possui a lista. O segundo, o *Id* da instância que será removida na lista.
Retorno:	Esta operação retorna um valor *booleano* indicando se houve sucesso na remoção.
Exemplo:	Este exemplo mostra uma operação para remover cabines de um navio. `INavioManager.removeCabineNavio(id_Navio: Integer, id_Cabine: Integer): Boolean`

Tabela 4.11: Operação que retorna todos os elementos de um campo que é uma lista.

Operação:	`get<TipoNavegável (no Plural)><TipoNegócio>`
Parâmetros:	O único parâmetro é o *Id* da instância que é dona da lista.
Retorno:	Esta operação retorna uma lista com todos os elementos da lista.
Exemplo:	Este exemplo mostra uma operação para pegar todas as cabines de um navio. `INavioManager.getCabinesNavio(id_Navio: Integer): List`

As operações apresentadas até este momento formam o conjunto mínimo para se trabalhar com as instâncias dos tipos de negócio na arquitetura. No entanto, para tornar os componentes mais práticos e interessantes para a aplicação, precisamos de métodos que busquem conjuntos de instâncias segundo alguns critérios de busca. Como solução, podemos utilizar os campos das estruturas dos tipos de negócio para gerar essas operações, onde o objetivo é retornar todas as instâncias que possuam um determinado valor para o campo. Utilizamos o seguinte formato para essas operações: *find<TipoNegócio>By<Campo>*. Apresentamos na Tabela 4.12 detalhes mais completos e exemplos desses métodos de busca.

Tabela 4.12: Operação de busca de instâncias segundo um determinado campo.

Operação:	`find<TipoNegócio>By<Campo>`
Parâmetros:	O parâmetro é o valor do campo que as instâncias retornadas devem possuir.
Retorno:	Esta operação retorna uma lista com os *Ids* das instâncias que obedecem ao critério de busca.

Exemplo:	(1) Este exemplo mostra uma operação para buscar os *Ids* de todas as faturas com um determinado valor.
	`IFaturaManager.findFaturaByValor(valor: Real): List`
	(2) Este exemplo mostra uma operação para buscar os *Ids* de todas as faturas de um determinado cliente.
	`IFaturaManager.findFaturaByCliente(id_Cliente: Integer): List`

Exemplo

Para facilitar a compreensão das regras descritas nesta seção, apresentamos um exemplo de geração de componentes gerentes de instâncias para o domínio de *hotelaria*. Apesar da sua simplicidade, este exemplo mostra-se suficiente por ajudar a captar detalhes importantes e responder questões que, eventualmente, ainda não estejam claras para o leitor. A modelagem dos requisitos descrita a seguir foi aproveitada do exemplo existente em (CHEESMAN e DANIELS, 2001), servindo apenas como base para a aplicação das regras deste trabalho.

O sistema que exemplificaremos é um software voltado para o cadastro de *clientes*, *hotéis*, *quartos* e *tipos de quarto*. Além destes elementos, o sistema deve ser capaz de fazer reservas pela Internet e oferecer consultas aos seus usuários. Os principais casos de uso deste software estão ilustrados na Figura 4.17, onde dois atores diferentes operam o sistema. A função do *administrador* é manter a base de dados atualizada para que os *hóspedes* possam consultá-la e fazer suas reservas.

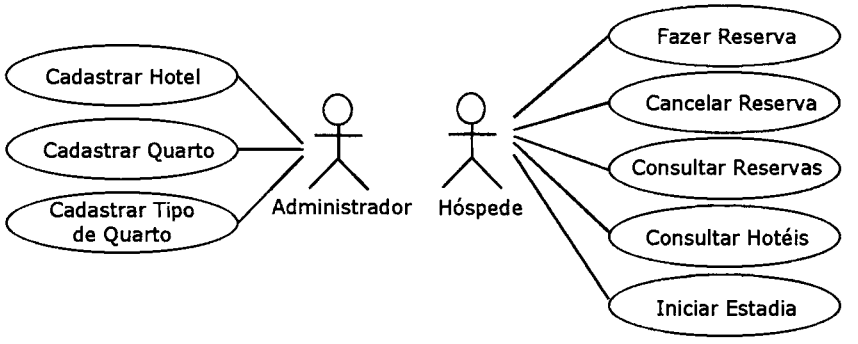


Figura 4.17: Casos de uso do sistema de hotelaria.

Para dar início a geração dos componentes, temos que observar o diagrama de tipos de negócio (Figura 4.18) e aplicar as regras de agrupamento apresentadas no início desta seção. Inicialmente, aplicamos a *regra da herança* e descobrimos que os tipos *Cliente* e *Hóspede* devem permanecer unidos em um mesmo grupo (Figura 4.19a).

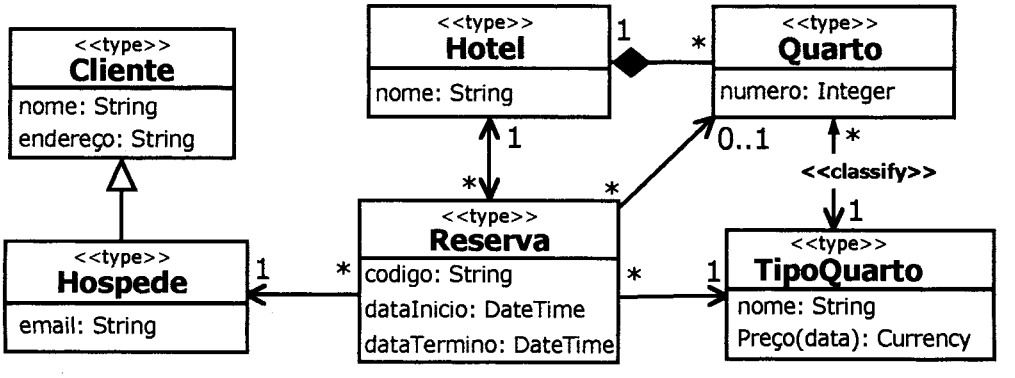


Figura 4.18: Diagrama de tipos de negócio (BTD) do sistema de hotelaria.

Para continuarmos com a aplicação das regras, criamos um grupo para cada tipo de negócio que tenha ficado sem nenhum grupo (Figura 4.19b). Após este passo, utilizamos as regras descritas na Tabela 4.2 para destacar os possíveis candidatos a permanecerem juntos, contando com a decisão do arquiteto para decidir quais deles, efetivamente, serão integrados. No exemplo, dois são indicados (Figura 4.19c): o *Quarto,* que é um forte candidato para ser gerenciado pelo componente que gerencia os *Hotéis,* e o *Tipo de Quarto,* que é um candidato fraco para ser gerenciado pelo componente que gerencia os *Quartos.*

A decisão de escolher o destino dos candidatos normalmente não é trivial. Para a realização deste exemplo, poderíamos optar em deixar os tipos *Hotel, Quarto* e *Tipo de Quarto* juntos em um mesmo grupo. Da mesma forma, também teríamos a opção de deixá-los separados em componentes diferentes. A vantagem de acatarmos as sugestões das regras é a oportunidade de construirmos componentes mais interessantes e completos para o domínio. Contudo, continuaremos este exemplo atendendo apenas ao

candidato forte sugerido, i.e., mantendo o *Quarto* junto do *Hotel*. Não incluiremos o *Tipo de Quarto* para não sobrecarregar muito esse componente. A Figura 4.19d ilustra os componentes gerentes de instâncias criados.

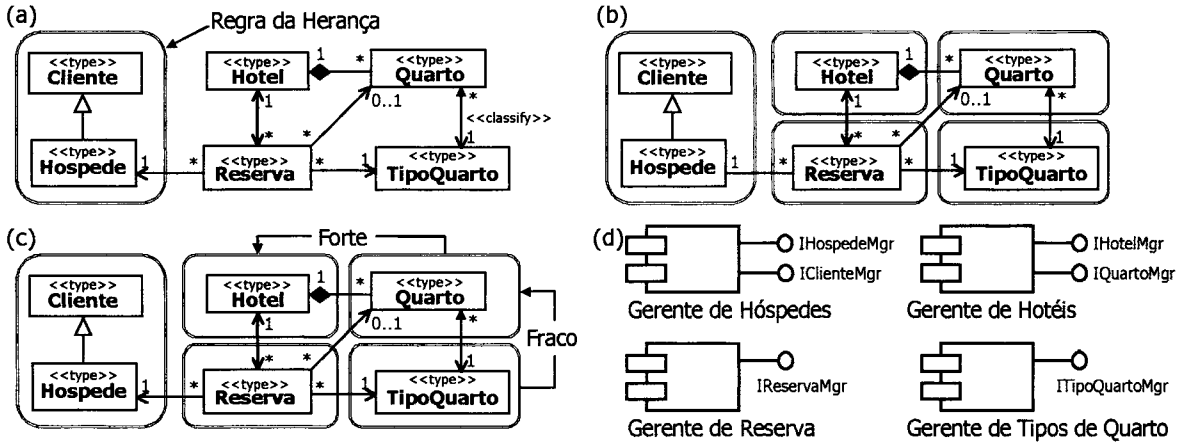


Figura 4.19: Agrupamento dos tipos de negócio de hotelaria.

Após o agrupamento dos tipos de negócio, temos que providenciar as *estruturas de dados* e as *interfaces* de acesso aos serviços dos componentes. Como vimos anteriormente, uma estrutura de dados é gerada para cada tipo de negócio, possuindo um campo *Id* (chave técnica), seus atributos (próprios e herdados) e dependências (navegações). Apresentamos na Figura 4.20 todas as estruturas de dados geradas para os tipos de negócio do nosso exemplo.

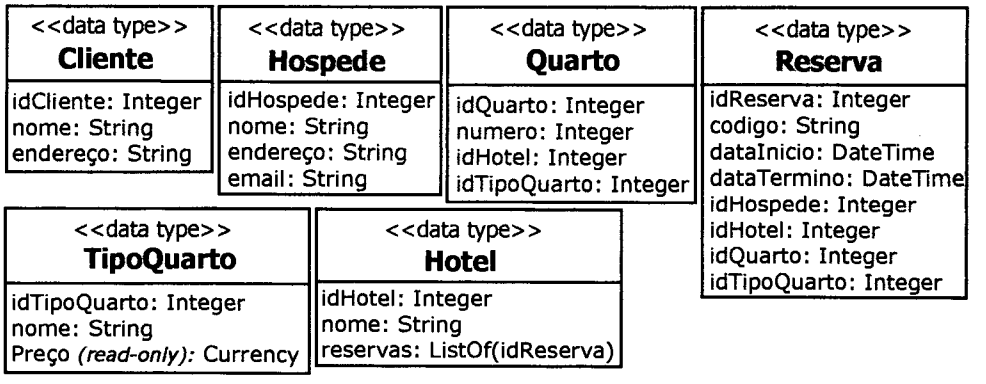


Figura 4.20: Estruturas de dados dos tipos de hotelaria.

Para a criação das interfaces, as estruturas de dados guiam a construção da maior parte das operações de acesso e controle das instâncias. Além de algumas operações de controle geral (como criação, exclusão, busca de *Ids*, etc.), todas as outras operações são baseadas no conteúdo das estruturas. Apresentamos na Figura 4.21 as operações geradas para a interface de gerenciamento de reservas. Para este exemplo, consideramos que a criação de uma reserva requer, além das datas de estadia, um cliente, um hotel e um tipo de quarto.

Figura 4.21: Interface para o gerenciamento de reservas .

4.2.1.2 – Geração de Componentes de Processo

Dentro do estilo baseado em tipos, a utilização dos componentes gerentes de instâncias apresenta algumas dificuldades devido a grande quantidade de chaves técnicas existentes nas assinaturas das suas operações. Além desse problema, o simples fato de lidarmos com componentes que apenas *armazenam* as instâncias faz com que os aspectos dinâmicos do negócio fiquem sob a responsabilidade de outros componentes na arquitetura.

Como uma forma de combater esses problemas e, ao mesmo tempo, evitar que a responsabilidade de execução dos processos de negócio fique espalhada irregularmente entre os outros componentes da aplicação, utilizamos os *componentes de processo* como um meio de unificar esses serviços na arquitetura. Basicamente, isto significa que todos os componentes que precisam realizar atividades de negócio devem utilizar esses serviços, sem se preocupar com o que está além desta fronteira.

A estratégia funciona da seguinte forma: cada componente de processo oferece interfaces com operações diretamente relacionadas à execução dos processos de negócio. A chamada de uma destas operações desencadeia uma série de chamadas nas

interfaces dos componentes gerentes de instâncias, trocando e buscando informações conforme o necessário. Para facilitar o uso destes serviços, as interfaces dos componentes de processo *não* utilizam as chaves técnicas das estruturas de dados, apenas os parâmetros voltados para o contexto de negócio (como atributos, por exemplo).

Para gerarmos as especificações dos componentes de processo com informações claras e completas, temos que ser capazes de criar tanto essas *operações que apóiam os processos* quanto a *seqüência de chamadas* que cada uma produz nos componentes gerentes de instâncias. Para isso, precisamos de meios eficientes que não sobrecarreguem a equipe com muito trabalho árduo e repetitivo, como, por exemplo, ficar *garimpando* cada operação para saber em que ordem elas podem ser chamadas para realizar um determinado objetivo.

Considerando a complexidade envolvida nessa atividade, a construção destas especificações deve levar em conta aspectos mais abstratos para se tornar prática e viável. Como estamos preocupados em *apoiar* a execução dos processos de negócio, temos que observar a modelagem dinâmica do software e empregá-la na especificação dessas operações. Para esse objetivo, optamos nesse trabalho em utilizar os **diagramas de casos de uso** da UML.

Como vimos no capitulo 2, além dos diagramas de caso de uso, a modelagem dinâmica também pode ser feita com os *diagramas de colaboração* da abordagem *Catalysis* (D'SOUZA e WILLS, 1999). A decisão que nos motivou a escolher os casos de uso, ao invés destes últimos, foi baseada em alguns critérios fundamentais que precisamos esclarecer.

Primeiro, a *UML* permite que os casos de uso sejam detalhados em **passos** que descrevem as ações envolvidas na execução do software. Na verdade, esses passos demonstram como a aplicação deve apoiar e interagir com o processo de negócio, revelando o que, de fato, estamos interessados em obter. No caso das colaborações, a ausência de seqüências ou informações encadeadas similares aos passos faz com que estas se tornem menos interessantes para o nosso objetivo.

Segundo, as colaborações não são tão descritivas quanto os casos de uso. Isto porque elas são compostas por ações que apenas descrevem as regras de relacionamento entre os tipos de negócio, o que não deixa tão claro *como* os usuários devem interagir com o software. Apesar de serem documentados em linguagem natural, os passos dos casos de uso explicam esses importantes detalhes e exibem informações manipuladas

durante a execução do processo, o que contribui com a geração das operações que precisamos.

Por fim, vale lembrar que os casos de uso são muito mais usados e conhecidos pelos desenvolvedores de software do que as colaborações do *Catalysis*. Ainda assim, ressaltamos fortemente que os diagramas de colaboração não foram utilizados nesta proposta porque não se adequaram ao objetivo pretendido. Além disso, não discutimos nem questionamos o seu valor perante o desenvolvimento baseado em componentes.

Para que seja possível utilizarmos os passos dos casos de uso na geração dos componentes de processo, precisamos de um mecanismo que entenda as ações descritas em cada um deles. Como interpretar a linguagem natural ainda é uma tarefa complexa na prática, desenvolvemos, nesta proposta, uma forma de expor este conhecimento utilizando uma linguagem baseada em princípios básicos da orientação a objetos.

Segundo LARMAN (2001), existem cinco operações básicas para expressar qualquer ação realizada sobre elementos conceituais de um sistema:

- Criação e exclusão de instâncias;
- Modificações de atributos;
- Associações formadas e desfeitas.

Apesar de estarmos interessados em desenvolver componentes, estas operações retratam exatamente a forma com que as instâncias dos tipos de negócio são articuladas pelo software. Isto permite que elas sejam utilizadas e combinadas para especificar cada um dos passos dos casos de uso, deixando claro as ações que precisamos para gerar os componentes de processo. A Tabela 4.13 apresenta cada uma dessas ações e como elas podem ser usadas nessa especificação.

Além dessas ações tradicionais, precisamos dispor de outras que complementem a geração dos componentes de processo. Primeiramente, temos que lembrar que todo software deve ser capaz de interagir com seus usuários, permitindo que dados sobre as instâncias sejam *mostrados* e *lidos* pela sua interface. Embora esta proposta não aborde a construção desses componentes de interfaces (e.g., *janelas*, *páginas WEB*, etc.), temos que preparar os componentes de processo para funcionarem e cooperarem corretamente com eles. Para isso, utilizamos as ações *Enter* e *Show* para especificar que um atributo de um tipo de negócio deve ser lido ou mostrado na tela, respectivamente. A Tabela 4.14 apresenta mais detalhes sobre estas ações.

Tabela 4.13: Ações para especificar os passos dos caso de uso.

Ação	Descrição	Exemplos
Create (Tipo)	Ação para **criar** uma instância de um tipo de negócio.	(1) *Create (Fatura)* (2) *Create (Pedido)*
Delete (Tipo)	Ação para **excluir** uma instância de um tipo de negócio.	(1) *Delete (Fatura)* (2) *Delete (Pedido)*
Update (Tipo.Atributo)	Ação para **atualizar** um atributo de um tipo de negócio.	(1) *Update (Fatura.Valor)* (2) *Update (Cliente.Saldo)*
Connect (Tipo, Tipo)	Ação para **conectar** dois tipos de negócio que possuem uma associação definida no BTD. Para ser possível conectá-los, o primeiro deve ser navegável para o segundo.	*Connect (Fatura, Cliente)* (Onde fatura é navegável para cliente)
Disconnect (Tipo, Tipo)	Ação para **desconectar** dois tipos de negócio que possuem uma associação definida no BTD. Para ser possível desconectá-los, o primeiro deve ser navegável para o segundo.	*Disconnect (Fatura, Cliente)* (Onde fatura é navegável para cliente)

Tabela 4.14: Ações para entrada e saída de dados.

Ação	Descrição	Exemplos
Enter (Tipo.Atributo)	Ação para indicar que um atributo de um tipo de negócio deve ser **fornecido** pelo usuário através da interface.	(1) *Enter (Fatura.Valor)* (2) *Enter (Pedido.Codigo)*
Show (Tipo.Atributo)	Ação para indicar que um atributo (campo) de uma instância deve ser **mostrado na tela**.	(1) *Show (Fatura.Valor)* (2) *Show (Cliente.Nome)*

O desenvolvimento de soluções de software, muitas vezes, introduz mudanças na execução dos processos de negócio das organizações. Tarefas como *imprimir relatórios*, *validar cartões de crédito* ou *controlar robôs à distância* são alguns exemplos dessas transformações trazidas pelas novas tecnologias. Para descrever funcionalidades deste tipo, utilizamos uma ação especial que pode ser usada nesses casos mais específicos. Os detalhes desta ação (denominada *Custom*) estão apresentados na Tabela 4.15.

Tabela 4.15: Ação para especificar casos mais específicos.

Ação	Descrição	Exemplos
Custom (String)	Ação para descrever casos onde as outras são insuficientes ou inadequadas. O parâmetro desta ação é uma *string* qualquer que indica a atividade realizada.	(1) *Custom (Imprimir_Relatorio)* (2) *Custom (Validar_Cartao)*

Utilizando cada uma das ações acima, somos capazes de especificar os passos dos casos de uso em uma linguagem mais simples de ser processada. Para isso, cada passo pode conter uma ou mais ações que expressam o seu efeito sobre o software. Apresentamos na Figura 4.22 um exemplo dessa especificação para o caso de uso *"Fazer Reserva"* do sistema de hotelaria citado na seção anterior.

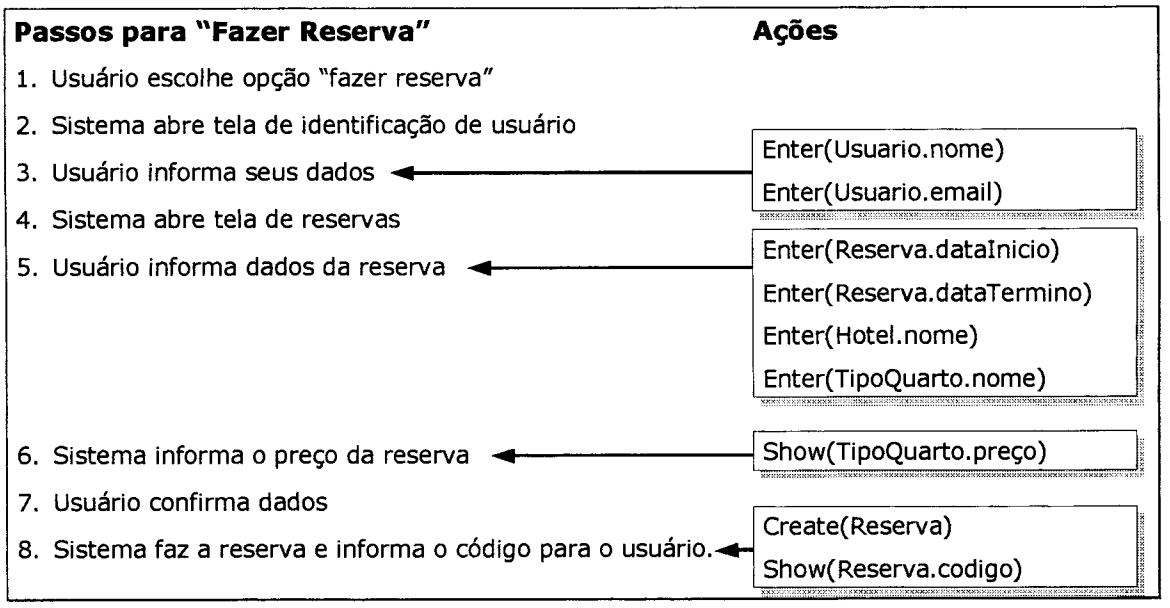


Figura 4.22: Especificação do caso de uso "Fazer Reserva"

Observando esse exemplo, percebemos que apenas quatro dos oito passos foram especificados com as ações estudadas. Isto acontece porque existem passos que estão fora do escopo dos componentes de negócio gerados por esta proposta. Normalmente, a grande maioria desses passos fica sob a responsabilidade dos componentes de interface da aplicação, onde janelas são abertas, botões são pressionados, etc. Para esses casos, não é necessário utilizar as ações, desde que as informações de negócio não estejam envolvidas.

Após a especificação dos casos de uso, podemos iniciar a geração dos componentes de processo da arquitetura. Como possuímos agora informações estruturadas e separadas em ações, podemos utilizar um **algoritmo** que as percorre para construir as interfaces e as seqüências de chamadas sobre os componentes gerentes de instância. A seguir, descrevemos detalhadamente como esse algoritmo funciona, mostrando seus princípios e condições necessárias para um correto funcionamento.

4.2.1.3 – Algoritmo para Realização de Casos de Uso

A geração dos componentes de processo seria muito trabalhosa se não tivéssemos um mecanismo que nos ajudasse com essa atividade. De um lado, temos os

casos de uso especificados com ações que detalham o seu comportamento sobre o software (seção anterior). Do outro, precisamos de componentes de processo com operações que realizam chamadas sobre os componentes gerentes de instâncias, obedecendo a lógica dos processos de negócio. A Figura 4.23 ilustra essa situação.

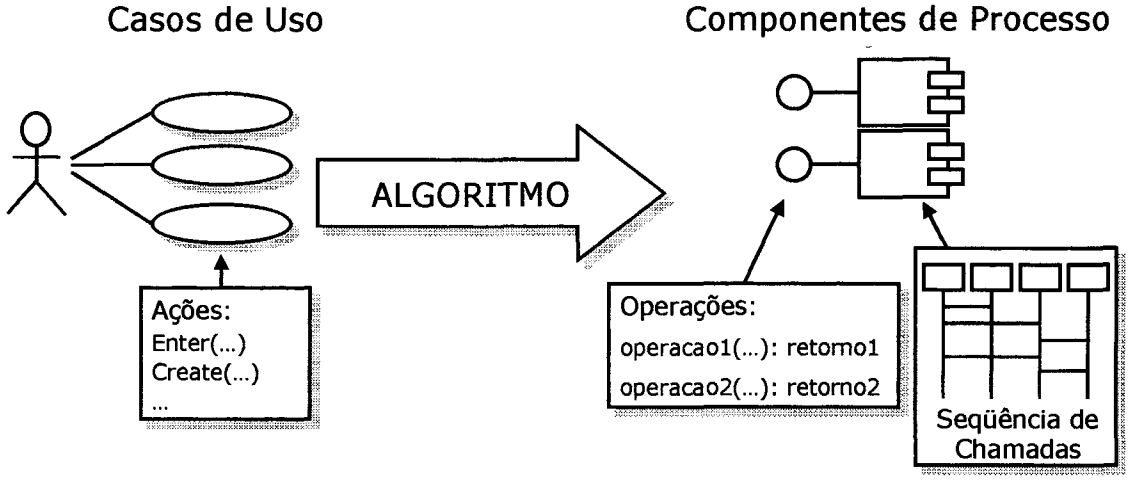


Figura 4.23: Conversão de casos de uso em componentes de processo.

Para preenchermos esse espaço e ajudarmos nessa transformação, desenvolvemos um algoritmo capaz de analisar um caso de uso e tentar elaborar possíveis interações que reflitam o seu objetivo na arquitetura. O seu funcionamento básico consiste em percorrer a seqüência de ações do caso de uso e, com isso, gerar todas as possíveis seqüências de chamadas que o realize. Por convenção, criamos uma interface para cada caso de uso, onde são inseridas todas as operações necessárias a sua execução.

Da mesma forma que utilizamos ações para especificar os casos de uso do software, precisamos ter algum tipo de informação associada às operações dos componentes gerentes de instâncias para indicar a sua utilidade. Isto é necessário porque o algoritmo precisa saber quais operações realizam determinada ação no sistema, utilizando-as conforme suas necessidades.

Para não termos que utilizar o nome das operações para saber o que elas fazem, cada uma recebe uma *ação* que indica a sua responsabilidade no componente. É importante lembrar que essas ações são geradas automaticamente com cada operação, não exigindo que a equipe interfira nesse procedimento. Apresentamos na Tabela 4.16 cada uma dessas operações (apresentadas anteriormente na *seção 4.2.1.1*) com suas respectivas ações.

Tabela 4.16: Ações associadas às operações de gerenciamento de um tipo de negócio.

Operação	Ação	Exemplo
create<TipoNegocio> (Tabela 4.3)	Create(TipoNegocio)	createFatura(...): Integer Ação: Create(Fatura)
delete<TipoNegocio> (Tabela 4.4)	Delete(TipoNegocio)	deleteFatura(...): Boolean Ação: Delete(Fatura)
getId<TipoNegocio> (Tabela 4.5)	Getter(TipoNegocio.id)	getIdFatura(...): Integer Ação: Getter(Fatura.id)
getAll<TipoNegocio> (Tabela 4.6)	Getter(TipoNegocio.All)	getAllFaturas(): List Ação: Getter(Fatura.All)
get<Campo> <TipoNegocio> (Tabela 4.7)	*Se o <Campo> for um atributo do tipo de negócio:* Getter(TipoNegocio.Campo)	*<Campo> é um atributo:* getValorFatura(...): Real Ação: Getter(Fatura.Valor)
	Se o <Campo> for uma associação do tipo de negócio: Getter(TipoNegocioAssociado.id)	*<Campo> é uma associação:* getIdClienteFatura(...): Integer Ação: Getter(Cliente.id)
set<Campo> <TipoNegocio> (Tabela 4.8)	*Se o <Campo> for um atributo do tipo de negócio:* Setter(TipoNegocio.Campo)	*<Campo> é um atributo:* setValorFatura(...): Boolean Ação: Setter(Fatura.Valor)
	Se o <Campo> for uma associação do tipo de negócio: Connect(TipoNegocio-TipoNegocioAssociado)	*<Campo> é uma associação:* setIdClienteFatura(...): Boolean Ação: Connect(Fatura-Cliente)
add<TipoNavegavel> <TipoNegocio> (Tabela 4.9)	Connect(TipoNegocio-TipoNegocioNavegavel)	addCabineNavio(...): Integer Ação: Connect(Navio-Cabine)
remove<TipoNavegavel> <TipoNegocio>	Disconnect(TipoNegocio-TipoNegocioNavegavel)	removeCabineNavio(...): Boolean Ação: Disconnect(Navio-Cabine)
find<TipoNegocio>By <Campo>	-	-

Agora que conhecemos o que cada operação dos componentes gerentes de instâncias faz, podemos utilizá-las para realizar os casos de uso do software. Essencialmente, o objetivo do algoritmo é percorrer cada caso de uso, ação por ação, tentando encontrar operações nas interfaces dos componentes gerentes de instâncias que possam ser chamadas para executá-lo.

Apesar de parecer simples, temos que lembrar que uma operação só pode ser chamada quando todos os seus parâmetros estiverem *disponíveis* para o algoritmo. Para que um parâmetro se torne disponível, é necessário que haja uma ação *Enter* no caso de uso, indicando que o usuário entrará esse dado em algum momento. Após este instante, este valor poderá ser usado em qualquer operação que o necessite.

Caso exista alguma operação que não possa ser chamada porque um de seus parâmetros não foi informado pelo usuário, o algoritmo deve tentar buscá-lo nas outras operações das interfaces. Para isso, ele deve procurar quais operações retornam aquele valor (identificadas pela ação `Getter(ValorProcurado)`), verificando se ele dispõe de todos os seus parâmetros. Caso algum parâmetro esteja faltando, todo o ciclo se repete até que a seqüência de chamadas esteja completa ou incapaz de continuar. A Figura 4.24 demonstra esse procedimento com um exemplo.

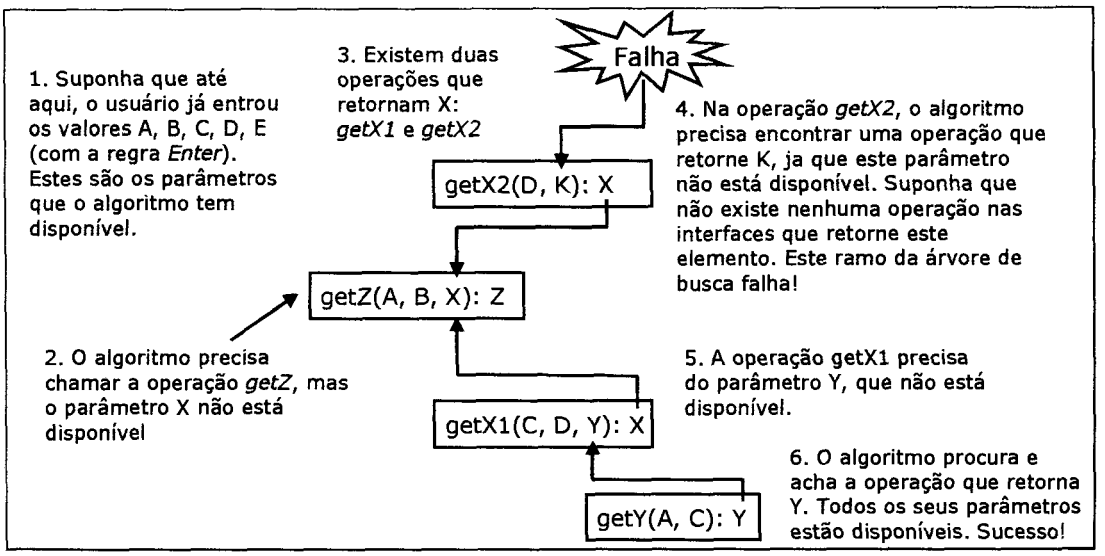


Figura 4.24: Seqüência de chamadas realizadas pelo algoritmo.

No exemplo da Figura 4.24, o algoritmo precisa do parâmetro X para chamar a operação *getZ*. Existem duas operações nas interfaces que retornam este valor: *getX1* e *getX2*. Para chamar a *getX2*, o parâmetro K precisa ser encontrado. Mas, como não existem operações que retornam este valor, este ramo da árvore de busca termina sem sucesso. Por outro lado, a operação *getX1* é bem sucedida, pois todos os seus parâmetros são alcançáveis a partir dos parâmetros disponíveis.

Existem dois pontos muito importantes que o algoritmo precisa implementar para funcionar corretamente. Primeiro, é fundamental que ele liste *todas* as possíveis seqüências que realizam um caso de uso, dando ao arquiteto a opção de escolher entre elas. Esta situação ocorre sempre quando um parâmetro precisa ser buscado e muitas são as operações que podem ser empregadas na sua busca. Um exemplo disso pode ser visto na Figura 4.25, onde a árvore de busca produz quatro seqüências diferentes de chamadas a partir da operação *getA*. É importante lembrar que a ordem de leitura das chamadas na árvore deve ser feita em *profundidade*, devido a necessidade de buscar os parâmetros não disponíveis antes de se chamar cada operação.

O segundo ponto que precisa ser tratado é o *acúmulo de conhecimento* que ocorre ao longo das chamadas de operação. Em outras palavras, o algoritmo precisa controlar e armazenar todos os parâmetros que são retornados pelas operações enquanto a árvore de busca é montada. Estes parâmetros se tornam disponíveis para que futuras chamadas possam usá-los caso necessário.

Como exemplo, podemos ver na Figura 4.25 que o conjunto inicial de parâmetros (*F, X, W, U*) ganhou novos elementos após a montagem de cada seqüência. Observando a seqüência 3, por exemplo, o parâmetro *T* foi buscado apenas UMA vez (com a operação *getT*) e foi aproveitado em DUAS operações: *getS2* e *getA*. Caso esse acúmulo de conhecimento não fosse feito, o algoritmo perderia muito tempo buscando parâmetros que já foram pegos anteriormente, tornando-o lento e ineficiente.

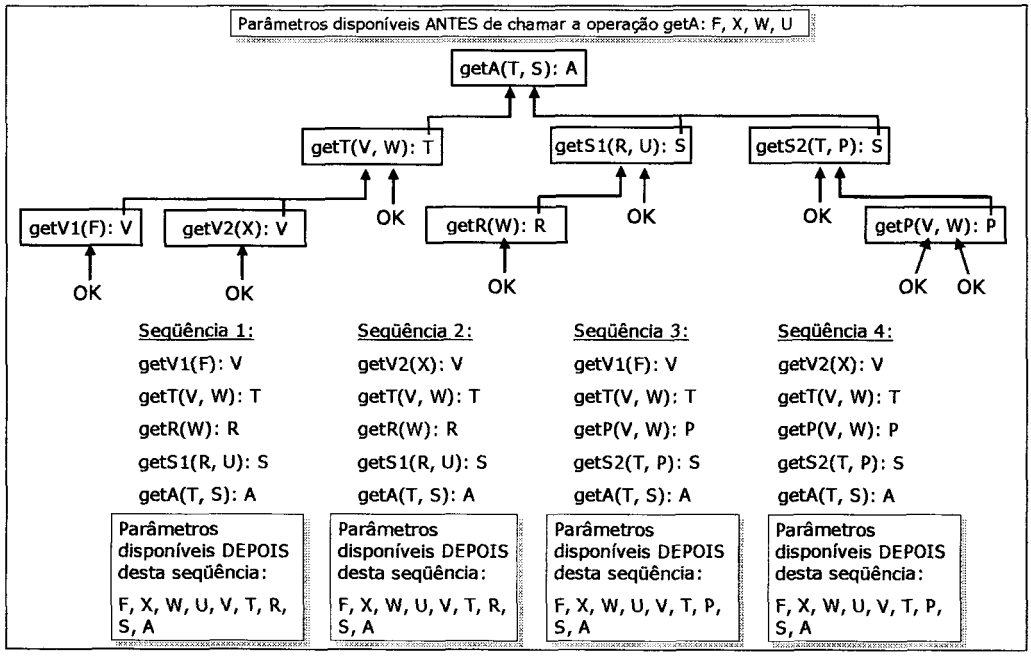


Figura 4.25: Exemplo de árvore de busca com diferentes seqüências.

As questões relacionadas à árvore de chamadas representam apenas o início do mecanismo que propomos neste trabalho. Para torná-lo útil e completo, precisamos esclarecer como as ações de um caso de uso são transformadas em uma especificação de componente de processo capaz de satisfazê-lo.

Vimos até agora que a regra *Enter* serve para informar quais atributos de negócio o usuário deve entrar através da interface do sistema. Como os componentes de processo não são responsáveis por essa interação com os usuários, tudo que essa regra nos oferece é o conhecimento que pode ser passado ao componente de processo durante a execução do caso do uso. Embora essa influência seja pequena, a regra *Enter* é a mais importante porque sem ela é impossível saber quais operações podemos chamar.

A regra *Show* é aquela que indica que um determinado valor precisa ser informado na tela do sistema. Como quem irá informar este valor é outro componente da arquitetura (e.g., janelas, etc.), temos que providenciar uma operação na interface do componente de processo que permita que este outro componente busque esse valor.

Para especificarmos esta operação e gerarmos as chamadas necessárias nos componentes gerentes de instância, precisamos utilizar o algoritmo que monta a árvore de chamadas estudada. Isto é feito passando para ele o **objetivo da busca** (no caso, o atributo que precisa ser mostrado na tela) e **todos os parâmetros disponíveis** até o momento (informados pela regra *Enter*).

Após a sua invocação, ele retornará todas as possíveis seqüências de chamadas que podem ser feitas para se buscar o elemento em questão. O arquiteto deve decidir qual das seqüências é a mais adequada e os parâmetros da operação do componente de processo são os parâmetros disponíveis que foram *efetivamente* utilizados nas chamadas. Como estamos tratando de uma operação que busca um valor, utilizamos o prefixo *get* para nomeá-la, seguido do nome do atributo retornado. A Figura 4.26 esclarece melhor esse procedimento.

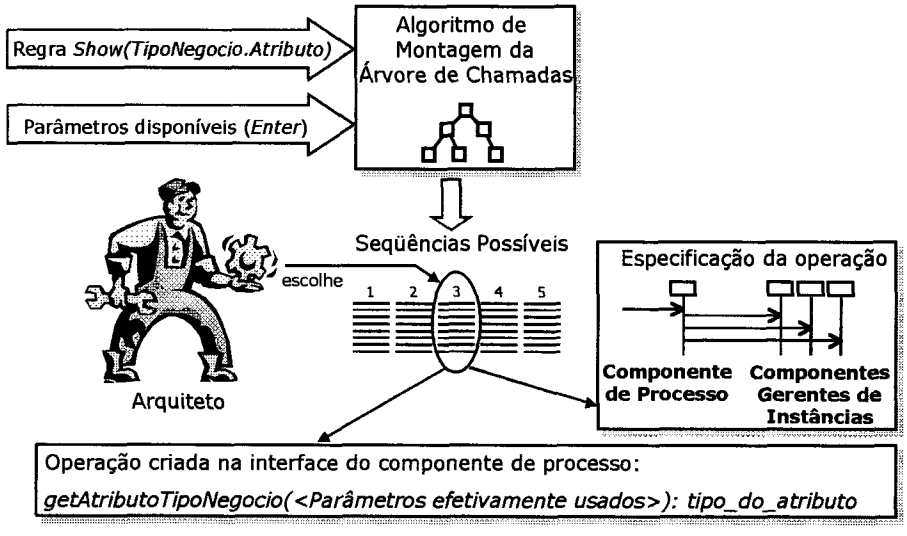


Figura 4.26: Especificação de uma operação a partir da regra *Show*.

Quando mais de uma seqüência de chamadas são geradas, a decisão do arquiteto é importante porque o algoritmo encontrou duas ou mais operações com características semelhantes e não sabe qual delas usar no caso de uso. Por exemplo, se temos duas interfaces diferentes, onde cada uma possui uma operação que realiza a ação *Getter(Fatura.valor)*, o algoritmo teria que gerar duas seqüências diferentes porque ele não entende a diferença entre elas. Assim, espera-se que o arquiteto escolha a mais adequada, observando em qual interface está a operação que ele deseja utilizar.

Para a regra *Create*, o procedimento é muito semelhante ao da regra *Show*. Primeiramente, o algoritmo busca todas as operações que realizam essa ação nos componentes gerentes de instâncias. Normalmente, só existe uma operação com este objetivo na arquitetura, que é a que propomos antes na Tabela 4.3. No entanto, se o arquiteto tiver especificado outras operações e associado esta ação a elas, o algoritmo as utilizará também.

Como as operações de *criar* instâncias possuem parâmetros, precisamos montar a árvore de chamadas para todos aqueles que não estão disponíveis para o algoritmo. Isto significa que cada parâmetro pode resultar em variadas seqüências de chamadas, exigindo, mais uma vez, que o arquiteto decida sobre a melhor delas. Caso a busca de todos esses parâmetros seja bem sucedida, a operação é especificada unindo-se as seqüências geradas para cada um deles, o que permite, ao final, que a operação de criação seja chamada. A Figura 4.27 demonstra essa especificação.

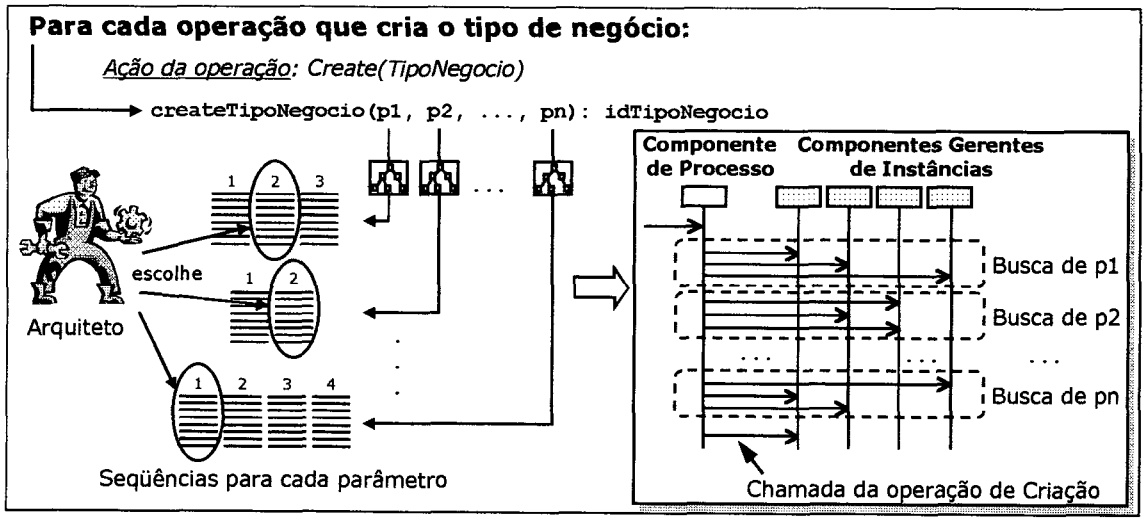


Figura 4.27: Especificação de operações de criação de instâncias.

A estratégia adotada para as ações *Delete*, *Update*, *Connect*, *Disconnect* e *Custom* é exatamente a mesma da ação *Create*. Contudo, duas observações precisam ser feitas para complementar essa descrição:

1. Para a ação *Disconnect*, também utilizamos as operações que realizam a ação *Connect* dos campos que não são listas, passando o valor nulo como parâmetro;
2. Para a ação *Custom*, o arquiteto precisa criar (em qualquer interface) uma operação responsável pela sua execução, associando a ela a mesma ação e o mesmo parâmetro utilizado no caso de uso. Isto fará com que o algoritmo ache esta operação e a chame durante a montagem da especificação.

A respeito desse assunto, o impacto sobre a geração dos componentes de processo é **amortizado** pelos componentes que interagem diretamente com os usuários do software. Na verdade, tarefas como *verificar preenchimentos de formulários, exibir mensagens, habilitar ou desabilitar funções*, etc. são de responsabilidade desses componentes que só acessam os componentes de processo quando *todas* as informações já foram devidamente coletadas.

A única influência que os componentes de processo podem sofrer quanto aos fluxos alternativos é a necessidade de informar aos outros componentes da arquitetura se suas operações obtiveram êxito durante a sua chamada. Isto pode ser feito especificando-se um *retorno adequado* para cada operação, elaborando códigos que indicam a existência de erros, inconsistências ou outros tipos de problema mais específicos.

Considerações sobre o Desempenho do Algoritmo

O desempenho do algoritmo apresentado é uma questão que precisamos destacar com mais cuidado nesse trabalho. Embora não estejamos discutindo como a implementação deve ser realizada internamente, é importante perceber que a tarefa de gerar todas as possíveis seqüências de chamadas para um caso de uso pode se tornar crítica para alguns casos.

Como a execução do algoritmo vasculha as operações das interfaces procurando por aquelas que realizam determinadas ações na arquitetura, limitar o número de interfaces disponíveis para a busca do algoritmo é uma estratégia que pode ajudar muito o tempo de resposta para o usuário. Neste caso, somente as interfaces dos componentes gerentes de instâncias e de outros componentes de interesse do arquiteto devem ser selecionadas, evitando acessos indesejados durante essa busca.

Além disso, quando consideramos a geração dos componentes de processo a partir dos casos de uso da aplicação, podemos deduzir que o número de seqüências geradas tende a ser normalmente baixo, o que facilita a decisão do arquiteto na escolha da mais adequada. A razão por traz desta *heurística* é bastante simples: partindo do princípio de que, no estilo baseado em tipos, um tipo de negócio só pode ser gerenciado por um único componente, as operações que *criam* e *apagam* suas instâncias, além das que *modificam* seus atributos e associações, só serão encontradas na interface deste componente. Essa coesão, somada ao fato de que apenas uma operação basta (na maioria dos casos) para cada uma dessas ações sobre um tipo de negócio, contribui para

Como existem muitos passos e ações dentro de cada caso de uso, podem existir situações onde não seja possível completar uma seqüência válida de chamadas (seja por falta de entrada de dados, inconsistência na modelagem, inexistência de alguma operação específica, etc.). Nestes casos, o algoritmo deve ser capaz de indicar os passos exatos a partir dos quais não foi possível continuar, alertando o arquiteto para que ele inspecione e re-avalie os motivos do problema.

Para exemplificarmos esse mecanismo de realização de casos de uso e facilitarmos o seu entendimento, apresentamos na Figura 4.28 a especificação do componente de processo gerado a partir do caso de uso "*Fazer Reserva*" que foi detalhado anteriormente na Figura 4.22. Neste exemplo, a interface *ICliente* representa os componentes da arquitetura que desejam utilizar esse serviço, tendo a sua disposição três operações básicas: *getPrecoTipoQuarto*, *createReserva*, *getCodigoReserva*. Essas operações foram geradas na interface *IFazerReserva*, que é realizada pelo componente de processo.

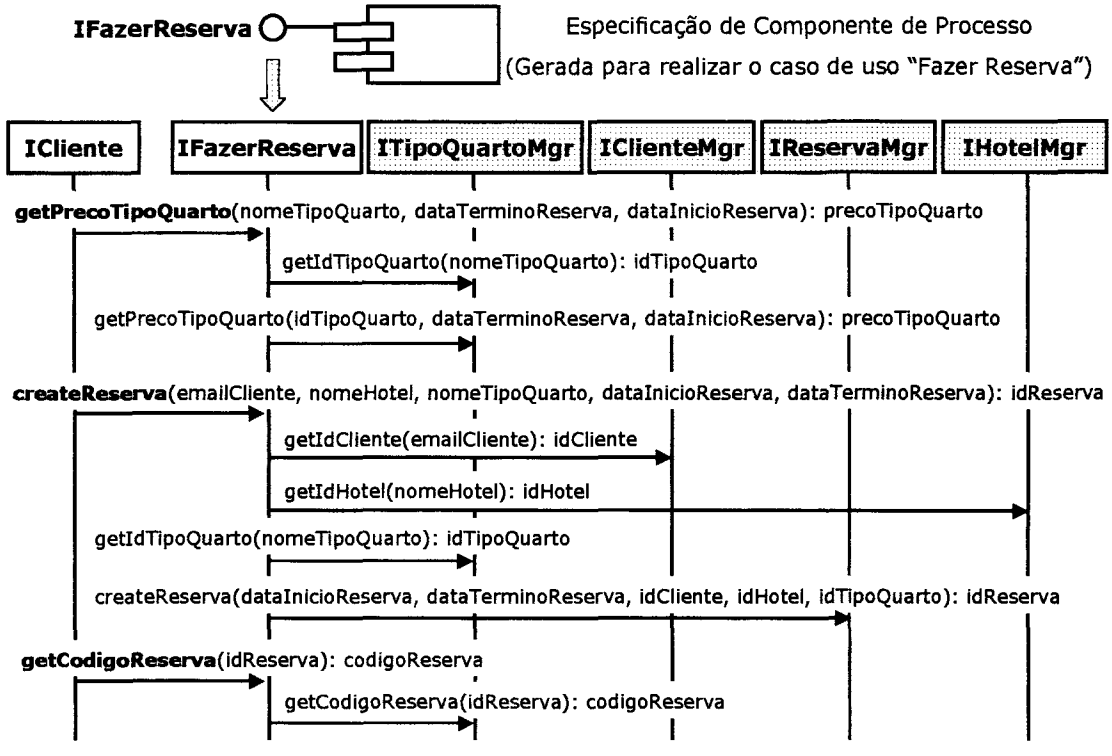


Figura 4.28: Especificação do componente responsável pelo caso de uso "Fazer Reserva"

Como este trabalho utiliza a definição de casos de uso proposta pela *UML*, não podemos nos esquecer que, além do cenário de sucesso do caso de uso (fluxo principal), existem situações que nem sempre ocorrem conforme o esperado. Esses casos, também chamados de *fluxos alternativos*, descrevem como o sistema deve proceder nessas situações onde há riscos de problemas.

a redução do número de seqüências e, de uma maneira geral, beneficiam a utilização da proposta. É importante ressaltar que não estamos **garantindo** que isso sempre ocorrerá, pois podem existir situações onde a execução do algoritmo pode resultar em um número elevado de seqüências de chamadas. Nestes casos, o arquiteto deve refinar as ações associadas às operações, utilizando *strings* mais específicas e livre de ambigüidade.

4.2.1.4 – Remoção e Integridade Referencial

Embora o estilo baseado em tipos não utilize endereçamento direto entre as suas instâncias, vimos que elas podem possuir dependências (*navegações*) entre si. Isto significa que a remoção de instâncias não pode ser feita simplesmente apagando-as fisicamente dentro dos componentes que as gerenciam. Na prática, temos que tomar muitos cuidados durante esse procedimento, porque outras instâncias correm o risco de ter sua integridade referencial violada se não forem adequadamente tratadas.

Como o gerenciamento das instâncias dos tipos de negócio não é feito por um único componente na arquitetura, precisamos elaborar uma estratégia que permita que cada componente seja *avisado* sobre as remoções efetuadas. Isto lhes garante a oportunidade de trata-las e evita que problemas maiores ocorram mais tarde.

Para especificarmos este recurso de forma eficiente na arquitetura, temos que difundir a informação de cada remoção somente para os componentes que realmente estão interessados nela. Para isso, precisamos conectá-los observando tanto as dependências entre os tipos de negócio no BTD, quanto à forma que eles foram agrupados para formar os componentes gerentes de instâncias.

Analisando esses dois fatores, podemos assumir que quando um tipo de negócio ***A*** depende de outro tipo ***B*** (i.e., é navegável para o tipo ***B***), o componente que gerencia ***A*** é um dos interessados no evento que o gerente de ***B*** emite durante uma remoção. Se ***A*** e ***B*** não foram agrupados em um mesmo gerente de instâncias, então o gerente de ***A*** deve ser conectado ao gerente de ***B*** para que este o informe sobre a exclusão de suas instâncias. A Figura 4.29 ilustra esse princípios.

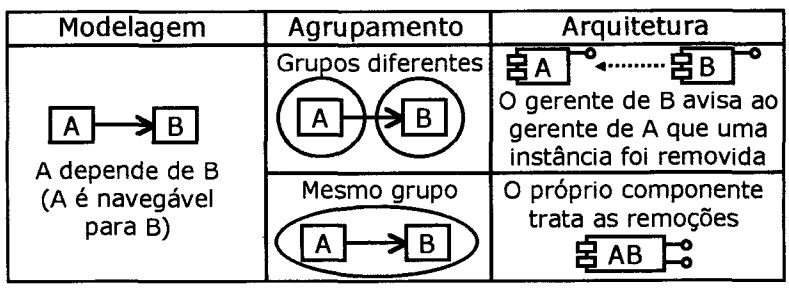


Figura 4.29: Efeito da dependência entre tipos na conexão dos seus gerentes.

Agora que sabemos quais componentes deverão estar conectados entre si, precisamos encontrar um padrão de interação que possa ser aplicado a cada dupla para que a mensagem da remoção possa fluir entre eles. Dentre as muitas soluções que podem ser empregadas, optamos neste trabalho por utilizar um padrão de interação que consideramos simples e que se revelou eficaz através da linguagem Java (JAVABEANS, 1997) e pela literatura (ESKELIN, 1999): *os Listeners*.

O princípio de funcionamento deste padrão é bastante simples: todo componente interessado em um determinado **evento** pode se registrar como "*ouvinte*" na interface do componente que o emite. Deste modo, sempre que o evento ocorrer, o componente *emissor* avisa essa notícia a todos os seus ouvintes registrados, passando para eles, inclusive, valores que eles precisam saber para reagirem a ela.

No contexto da arquitetura, todo componente emissor deve possuir em sua interface uma *operação* para registrar os ouvintes interessados nos seus eventos. Como estamos tratando aqui de eventos relacionados à remoção de instâncias na arquitetura, utilizamos o nome *addListenerFor<TipoNegocio>Removal* para nomeá-la. O único parâmetro desta operação é um objeto da interface *I<TipoNegocio>RemovalListener*, que deve ser realizada por todos os componentes ouvintes do evento.

A interface *I<TipoNegocio>RemovalListener* é necessária porque ela contém o método que todos os ouvintes devem implementar para tratar a ocorrência do evento. Assim, quando o componente emissor desejar dispará-lo, ele percorre todos os ouvintes chamando esta operação e passando para ela o *Id* da instância que foi removida. O procedimento executado pelos ouvintes deve ser especificado pelo arquiteto. Na grande maioria dos casos, todas as instâncias que dependem da instância removida são apagadas também, podendo desencadear um processo em cascata na arquitetura.

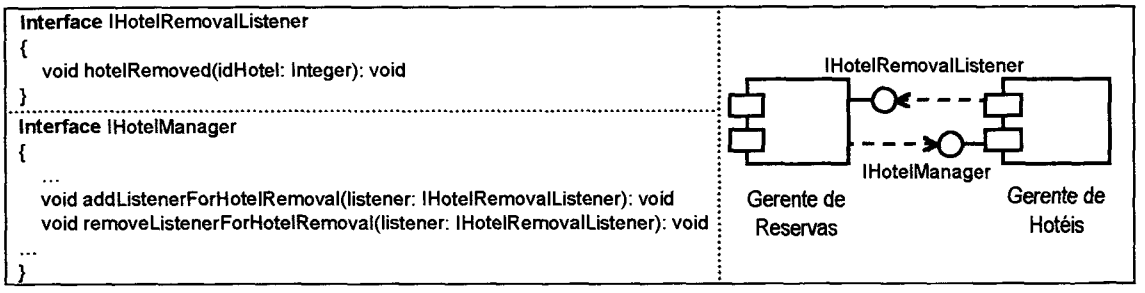


Figura 4.30: Controle de remoção no sistema de hotelaria.

Voltando para o sistema de hotelaria iniciado na seção 4.2.1.1, percebemos pela Figura 4.18 que o tipo *Reserva* é navegável para o tipo *Hotel*. Isto significa que o componente que gerencia as reservas é interessado no evento de remoção de hotéis, já

que muitas reservas podem ser afetadas com esse acontecimento. Apresentamos na Figura 4.30 a conexão desses componentes segundo o padrão estudado. Todas as outras navegações desse sistema seguem o mesmo estilo.

4.2.2 – Estilo baseado em Instâncias

O estilo baseado em instâncias representa a segunda alternativa para a geração de componentes de negócio neste trabalho. Agora, ao invés de construirmos componentes que gerenciam e controlam o acesso às instâncias dos tipos de negócio, especificaremos uma arquitetura mais solta e flexível, capaz de operar e referenciar seus objetos diretamente em memória.

Conforme vimos no capítulo 2, este estilo utiliza dois tipos de componentes para cada tipo de negócio existente no software: os *componentes de entidade* e os *componentes-coleção* (Figura 4.31). Os *componentes de entidade* são aqueles que são utilizados para formar as instâncias, servindo como um "*molde*" para elas. Para não deixar esses objetos *perdidos* na memória, todas as instâncias são criadas pelo *componente-coleção*, o qual mantêm suas referências em uma lista e oferece serviços de busca para os outros componentes da arquitetura.

A geração dessa dupla de componentes deve ser realizada para cada tipo de negócio do software. Estudaremos agora os detalhes desse procedimento e, logo em seguida, discutiremos alguns problemas e soluções para as questões relacionadas à remoção de instâncias neste estilo.

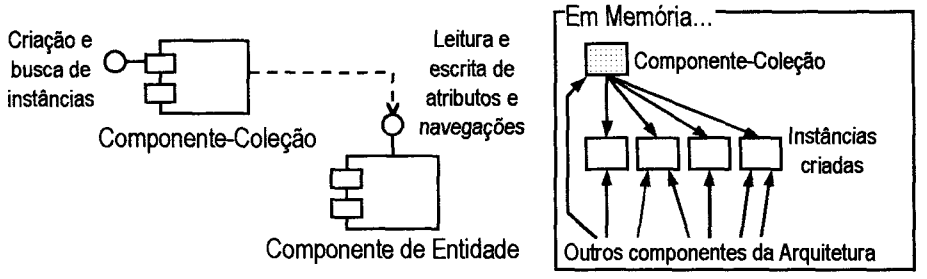


Figura 4.31: Componentes de entidade e coleções.

4.2.2.1 – Geração dos Componentes de Entidade

Os componentes de entidade são representações fiéis dos tipos de negócio na arquitetura. Para especificá-los, precisamos montar suas interfaces a partir dos atributos e navegações presentes no *diagrama de tipos de negócio* (*BTD*) da aplicação. Cada interface criada é nomeada no formato *I<TipoNegocio>* (e.g., *IFatura*, *IProduto*, etc.) e possui todas as operações necessárias a manipulação de uma instância.

Como a criação das instâncias dos tipos de negócio é feita pelos componentes-coleção, não especificaremos uma operação na interface dos componentes de entidade para criá-los. Esta decisão foi tomada, principalmente, porque a existência desta operação não garante que ela seja sempre chamada quando uma nova instância for criada, lembrando que a maioria das tecnologias não permite *construtores* em interfaces. Deixaremos as operações relacionadas à criação de instâncias para a seção seguinte, onde a geração dos componentes-coleção está descrita.

Começaremos agora a especificar as operações que acessam e manipulam os dados de cada instância. Para isso, utilizaremos os atributos dos tipos de negócio para gerar as operações de leitura e escrita destes valores. A Tabela 4.17 e a Tabela 4.18 mostram, respectivamente, os detalhes e um exemplo destas operações. Para tipos de negócio que herdam de outros tipos, a interface do seu componente de entidade deve ter essas operações para todos os atributos herdados, garantindo, assim, que estes componentes sejam mais independentes e autocontidos.

Tabela 4.17: Operação de leitura de um atributo de uma instância.

Operação:	`get<Atributo><TipoNegocio>`
Parâmetros:	Não há parâmetros.
Retorno:	Esta operação retorna o valor do atributo consultado.
Exemplo:	Este exemplo mostra uma operação para consultar o atributo *valor* de uma fatura. `IFatura.getValorFatura(): Real`

Tabela 4.18: Operação de escrita em um atributo de uma instância.

Operação:	`set<Atributo><TipoNegocio>`
Parâmetros:	O parâmetro é o novo valor do atributo.
Retorno:	Não há retornos.
Exemplo:	Este exemplo mostra uma operação para alterar o campo *valor* de uma fatura. `IFatura.setValorFatura(valorFatura: Real): void`

Para finalizarmos a geração dos componentes de entidade, precisamos especificar as operações relacionadas às navegações dos tipos de negócio. Como estamos tratando de um estilo que não utiliza *chaves técnicas* (*ids*), essas operações lidam diretamente com as interfaces dos componentes navegáveis, acessando-os como se fossem *"ponteiros"*. Assim, para os casos onde a navegabilidade ocorre para somente uma única instância (1 ou 0..1), basta criarmos um método de leitura e escrita para esta referência. A Tabela 4.19 e a Tabela 4.20 mostram essas operações.

Tabela 4.19: Operação de leitura de uma navegação de uma instância.

Operação:	`get<TipoNavegavel><TipoNegocio>`
Parâmetros:	Não há parâmetros.
Retorno:	Esta operação retorna uma referência para o componente de entidade do tipo navegável.
Exemplo:	Este exemplo mostra uma operação para consultar o cliente de uma fatura. `IFatura.getClienteFatura(): ICliente`

Tabela 4.20: Operação de escrita em uma navegação de uma instância.

Operação:	`set<TipoNavegavel><TipoNegocio>`
Parâmetros:	O parâmetro é a referência para o componente de entidade do tipo navegável.
Retorno:	Não há retornos.
Exemplo:	Este exemplo mostra uma operação para alterar o cliente de uma fatura. `IFatura.setClienteFatura(cliente: ICliente): void`

Para os casos onde a navegabilidade ocorre para mais de uma instância (*inúmeras* [*] ou *multiplicidade fixa* [k > 1]), temos que criar operações que inserem e removem essas referências navegáveis. Estas operações estão apresentadas na Tabela 4.21 e na Tabela 4.22, respectivamente. Para buscar todas as referências navegáveis do componente, utilizamos a operação apresentada na Tabela 4.23.

Tabela 4.21: Operação de inserção de referências de uma navegação.

Operação:	`add<TipoNavegável><TipoNegócio>`
Parâmetros:	O único parâmetro é a instância para a qual o componente será navegável.
Retorno:	Esta operação retorna a posição da lista interna do componente de entidade onde houve a inserção.
Exemplo:	Este exemplo mostra uma operação para inserir cabines em um navio. `INavio.addCabineNavio(cabine: ICabine): Integer`

Tabela 4.22: Operação de remoção de elementos em um campo que é uma lista.

Operação:	`remove<TipoNavegável><TipoNegócio>`
Parâmetros:	O único parâmetro é a instância que não será mais navegável.
Retorno:	Esta operação retorna um valor *booleano* indicando se houve sucesso na remoção.
Exemplo:	Este exemplo mostra uma operação para remover uma cabine de um navio. `INavio.removeCabineNavio(cabine: Icabine): Boolean`

Tabela 4.23: Operação que retorna todos os elementos de um campo que é uma lista.

Operação:	`get<TipoNavegável (no Plural)><TipoNegócio>`
Parâmetros:	O único parâmetro é o *Id* da instância que é dona da lista.
Retorno:	Esta operação retorna uma lista com todos os elementos da lista.
Exemplo:	Este exemplo mostra uma operação para pegar todas as cabines de um navio. `INavioManager.getCabinesNavio(id_Navio: Integer): List`

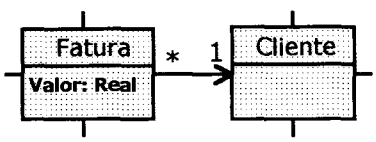
4.2.2.2 – Geração dos Componentes-Coleção

Os componentes-coleção assumem um papel fundamental no apoio aos componentes de entidade neste tipo de arquitetura. Suas interfaces – nomeadas no formato *I<TipoNegocio>Collection* – oferecem operações tanto para a fabricação das instâncias dos tipos de negócio quanto para a realização de buscas pela arquitetura. Na prática, esses serviços revelam-se muito úteis por contribuírem com a organização e coleta das instâncias que, sem eles, permaneceriam dispersas e sem controle em memória.

Para que os componentes-coleção possam fabricar instâncias de um tipo de negócio na arquitetura, precisamos gerar uma operação em suas interfaces para essa finalidade. Assim como no estilo baseado em tipos, é fundamental que o arquiteto informe todos os parâmetros exigidos nessa operação, já que eles são imprevisíveis e variam de negócio para negócio.

Após criar uma instância, a operação de criação retorna uma referência para o objeto criado, permitindo que outros componentes o enderecem. Vale lembrar que essa referência também é guardada em uma lista interna do componente-coleção, para que ele consiga acessá-la posteriormente. Apresentamos na Tabela 4.24 os detalhes e um exemplo desta operação.

Tabela 4.24: Operação de criação de instâncias.

Operação:	`create<TipoNegocio>`
Parâmetros:	Os parâmetros podem ser atributos do tipo de negócio criado ou *Ids* de outras instâncias. Neste último caso, só deve ser permitido utilizar instâncias de tipos de negócio que possuem associação com o tipo criado.
Retorno:	Esta operação retorna uma referência para o componente de entidade recém criado.
Exemplo:	Este exemplo mostra uma operação para criar faturas, onde o primeiro parâmetro é o valor da fatura e o segundo, a referência da instância do cliente.  `IFaturaCollection.createFatura(valor: Real, cliente: ICliente): IFatura`

A primeira operação de busca que é gerada para o componente-coleção é aquela que só pode retornar uma única instância para um determinado conjunto de dados. Para isso, temos que especificar (com a ajuda do especialista) as informações que são necessárias para identificar unicamente os tipos de negócio da aplicação. Como essa

discussão já foi levantada anteriormente com a geração da operação vista na Tabela 4.5 do estilo baseado em tipos, não a descreveremos novamente aqui. A Tabela 4.25 apresenta esta operação com o mesmo exemplo utilizado anteriormente.

Tabela 4.25: Operação que busca uma instância em particular.

Operação:	`get<TipoNegocio>`
Parâmetros:	Os parâmetros podem ser atributos do tipo de negócio criado ou referências para outras instâncias. Neste último caso, só deve ser permitido utilizar instâncias de tipos de negócio que possuem associação com o tipo criado.
Retorno:	Referência para a instância procurada. Caso esta não exista, a operação retorna nulo.
Exemplo:	Este exemplo mostra uma operação para buscar a instância de uma cabine de passageiros em um navio. O primeiro parâmetro é o número da cabine e o segundo, a referência para o navio onde ela se encontra. `ICabineCollection.getCabine(numero: Integer, navio: INavio): ICabine`

As operações de busca que finalizam a geração dos componentes-coleção são baseadas nos atributos e navegações dos tipos de negócio do software. Estas operações são bastante parecidas com aquelas descritas na Tabela 4.12, sendo, porém, que essas não utilizam chaves técnicas (*ids*). O retorno dessas operações consiste de uma lista de referências para as instâncias que se encontram dentro do critério de busca aplicado. A Tabela 4.26 apresenta essas operações com mais detalhes.

Tabela 4.26: Operação de busca de instâncias em memória.

Operação:	`find<TipoNegócio>By<Atributo (ou TipoNavegavel)>`
Parâmetros:	O parâmetro é o valor do atributo (ou referência para a instância navegável) que as instâncias retornadas devem possuir.
Retorno:	Esta operação retorna uma lista com as referências das instâncias que satisfazem ao critério de busca.

Exemplo:	(1) Este exemplo mostra uma operação para buscar todas as faturas com um determinado valor.
	`IFaturaCollection.findFaturaByValor(valor: Real): List`
	(2) Este exemplo mostra uma operação para buscar todas as faturas de um determinado cliente.
	`IFaturaCollection.findFaturaByCliente(cliente: ICliente): List`

4.2.2.3 – Remoção e Integridade Referencial

Um dos problemas mais críticos do estilo baseado em instâncias é a forma de tratar suas remoções em tempo de execução. Como cada instância deste estilo é um objeto livre na arquitetura, ele pode ser referenciado por inúmeros outros componentes

simultaneamente. Caso uma determinada instância seja *desalocada* inadvertidamente durante a execução do software, os efeitos que podem ocorrer são imprevisíveis e possivelmente perigosos à integridade dos dados da aplicação.

Considerando que rastrear todos os componentes que referenciam uma determinada instância é algo fora de cogitação, este trabalho aplicou uma solução viável para contornar esse problema. Ao invés de excluirmos as instâncias fisicamente da memória, utilizaremos um campo para indicar a sua **remoção lógica**.

Apesar de não ser a melhor solução, os resultados obtidos são considerados satisfatórios por vários motivos. Primeiro, a remoção lógica reduz a probabilidade de erros causados durante a execução, já que os objetos não são desalocados da memória. Segundo, a única preocupação que os outros componentes da arquitetura terão quando utilizarem uma instância é saber se ela, naquele momento, não está como "*excluída*". Esta verificação é bastante simples e não sobrecarrega a especificação desses outros componentes.

Para viabilizarmos esta solução, precisamos incluir duas operações na interface de todos os componentes de entidade criados. Ambas estão detalhadas, respectivamente, na Tabela 4.27 e na Tabela 4.28.

Tabela 4.27: Operação que altera o estado de exclusão de uma instância.

Operação:	`setDeleted`
Parâmetros:	Valor *booleano* que indica se a instância está excluída. Este parâmetro existe para que o programador tenha a flexibilidade de desativar ou reativar instâncias em tempo de execução.
Retorno:	Nenhum valor é retornado.
Exemplo:	Este exemplo mostra a operação para excluir uma fatura: `IFatura.setDeleted(deleted: boolean): void`

Tabela 4.28: Operação que verifica se uma instância está excluída.

Operação:	`IsDeleted`
Parâmetros:	Não há parâmetros.
Retorno:	Valor *booleano* que indica se a instância está excluída.
Exemplo:	Este exemplo mostra a operação que é usada para saber se uma fatura foi excluída: `IFatura.isDeleted(): boolean`

4.3 – Decisão e Escolha do Melhor Estilo

Os dois estilos arquiteturais estudados constituem diferentes oportunidades para o desenvolvimento de sistemas de negócio. Cada um possui suas próprias vantagens e

desvantagens que os tornam mais adequados para alguns contextos e piores para outros. Na prática, isto significa que escolher entre eles não é tão simples quanto parece, exigindo uma certa experiência da equipe de desenvolvimento.

A primeira influência que afeta essa escolha é o conjunto de requisitos não-funcionais da aplicação. Para que seja possível satisfazê-los, é fundamental conhecer como cada um dos estilos se comportam em relação a eles, revelando ao arquiteto seus pontos fortes e fracos. Apesar de ajudarem muito, esse conhecimento não é suficiente e representa apenas uma parte do problema que estamos analisando.

Para completarmos esse quadro, mais um importante aspecto precisa ser considerado: a *tecnologia*. Embora a montagem da arquitetura em DBC seja independente dela, sua influência sobre os requisitos não-funcionais da implementação não é desprezível. Em muitos casos, os efeitos são tão fortes que acabam inviabilizando determinados projetos que, em outras tecnologias, seriam perfeitamente aceitos.

Analisando esses fatores, precisamos de um mecanismo adequado que nos ajude a escolher entre os dois estilos utilizados neste trabalho. Para isso, precisamos combinar corretamente as características de cada um deles com as diversas tecnologias de componente existentes no mercado. Isto representa a base mínima para que qualquer inferência possa ser feita sobre o assunto.

Para alcançar esse objetivo, utilizamos o trabalho descrito por XAVIER (2001), que apresenta uma proposta de seleção de padrões arquiteturais utilizando atributos de qualidade desejáveis para um software. Apesar de não estar totalmente relacionada ao tema desta tese, este trabalho consegue atender satisfatoriamente às suas necessidades de apoio a decisão.

Em sua pesquisa, XAVIER utiliza onze características arquiteturais de qualidade (obtidas da norma ISO/IEC 9126-1) para avaliar padrões arquiteturais orientados a objetos (BUSCHMANN *et al.*, 1996). O objetivo dessa avaliação é analisar a influência que cada padrão exerce sobre as características de qualidade da arquitetura, revelando o quanto elas são beneficiadas ou desfavorecidas por ele, considerando aspectos de esforço e custo de projeto. Para isso, a avaliação é feita utilizando-se uma classificação que expressa, em vários níveis, essa relação:

1. Muito Bom (MB): Característica principal do padrão utilizado, sendo sua aplicação completamente favorável;
2. Bom (B): A utilização do padrão é favorável, mas exige algum esforço de projeto para alcançar o benefício almejado;

3. Médio (M): A utilização do padrão se aplica a situações em que o esforço gasto justifica o benefício alcançado;
4. Ruim (R): A utilização do padrão é injustificável na maioria das situações;
5. Muito Ruim (MR): A utilização do padrão é altamente desfavorável;
6. Desconhecido (D): Quando não há menção sobre esta relação de esforço.

A Tabela 4.29 apresenta a avaliação inicial realizada sobre esses padrões, a qual foi baseada na própria definição fornecida por BUSCHMANN *et al.* (1996), e complementada com descrições de SHAW *et al.* (1995), CLEMENTS (1996) e BOSCH (2000).

Tabela 4.29: Avaliação dos padrões arquiteturais para as características de qualidade.

Padrões Arquiteturais / (ISO/IEC 9126-1) Características Arquiteturais	Camadas	Pipes & Filters	BlackBoard	Broker	MVC	PAC	Microkernel	Reflection
Interoperabilidade	B	D	D	B	D	M	M	D
Segurança de Acesso	MB	MB	M	B	B	M	B	D
Maturidade	M	R	MR	M	M	M	B	D
Tolerância a Falhas	M	R	MB	R	M	M	R	R
Recuperabilidade	M	MR	R	R	D	R	R	D
Operacionalidade	B	D	D	D	MB	MB	D	D
Comportamento em relação ao tempo	M	B	MR	MR	M	R	M	R
Comportamento em relação aos recursos	R	R	MR	MR	M	R	M	R
Modificabilidade	B	M	MB	B	B	R	MB	MB
Testabilidade	MB	R	MR	R	B	MR	M	D
Adaptabilidade	MB	MR	D	B	B	D	B	B

Com estas informações armazenadas em uma base, XAVIER desenvolveu um cálculo capaz de estimar, em termos numéricos, quais padrões são os mais indicados para um determinado objetivo pretendido pelo projetista. Para isso, é preciso indicar cada uma das características de qualidade como sendo Importante (IM), Desejável (DJ) ou Irrelevante (IR) para o software que se pretende construir. Os resultados são obtidos transformando-se a base de dados com a avaliação (Tabela 4.29) e o objetivo do projetista em vetores normalizados no $\mathbf{R}^{11}$, onde as distâncias entre eles são calculadas e comparadas umas com as outras. Assim, os padrões mais próximos do objetivo são os mais apropriados para o projeto. A Figura 4.32 ilustra esse procedimento.

Para utilizar a proposta de XAVIER neste trabalho, adaptamos seu contexto para que fosse possível apoiar a decisão sobre os **estilos** e as **tecnologias de componentes** existentes. Dessa forma, ao invés de padrões arquiteturais, a avaliação agora é feita para cada combinação possível entre esses dois elementos, mantendo os mesmos critérios e

conceitos anteriores. Como exemplo, um possível preenchimento para esta avaliação pode ser vista na Tabela 4.30, onde as tecnologias *Enterprise Java Beans* (*EJB*), *JavaBeans* e *COM* estão avaliadas nos dois estilos arquiteturais estudados.

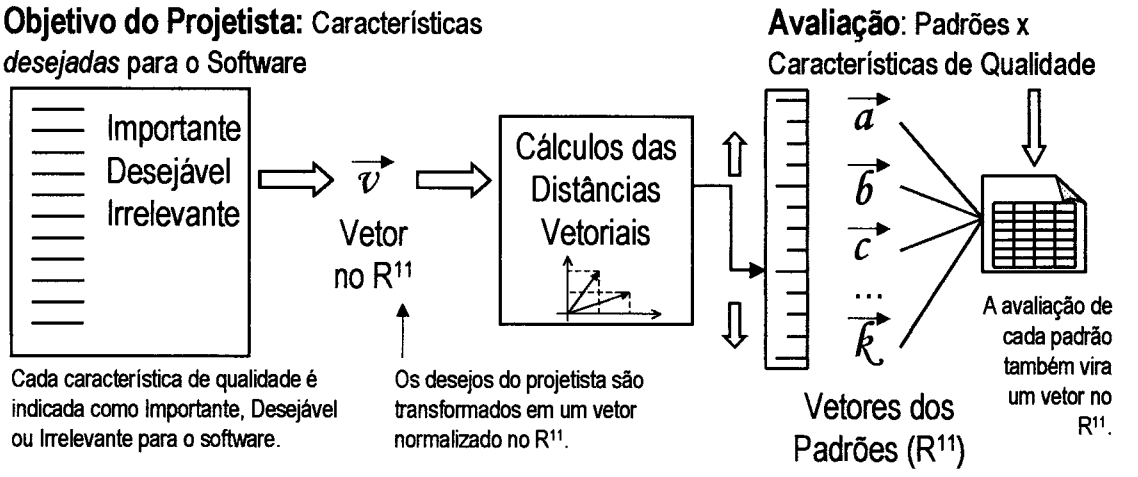


Figura 4.32: Proposta de instanciação de padrões (XAVIER, 2001).

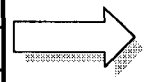
Tabela 4.30: Avaliação dos estilos e tecnologias para as características de qualidade.

Estilos & Tecnologias / (ISO/IEC 9126-1) Características Arquiteturais	EJB - Estilo baseado Tipos	EJB - Estilo baseado Instâncias	JavaBeans - Estilo baseado Tipos	JavaBeans - Estilo baseado Instâncias	COM - Estilo baseado Tipos	COM - Estilo baseado Instâncias
Interoperabilidade	B	B	B	B	B	D
Segurança de Acesso	MB	R	MB	R	B	M
Maturidade	MB	M	MB	M	MB	R
Tolerância a Falhas	MB	MR	B	M	B	R
Recuperabilidade	D	MR	B	R	M	MR
Operacionalidade	M	B	R	B	M	MB
Comportamento em relação ao tempo	B	MR	B	M	M	MB
Comportamento em relação aos recursos	B	MR	B	R	M	B
Modificabilidade	MB	M	MB	M	MB	M
Testabilidade	MB	R	MB	R	B	R
Adaptabilidade	MB	MB	MB	MB	M	M

Os cálculos originais propostos por XAVIER não precisaram ser adaptados para funcionarem com a nova forma de avaliação (Tabela 4.30). Assim, quando o arquiteto desejar saber quais as melhores combinações de estilo e tecnologia para o seu software, basta que ele informe a relevância de cada uma das onze características de qualidade indicadas. Um exemplo desse procedimento e os resultados obtidos (considerando a avaliação da Tabela 4.30) podem ser vistos na Figura 4.33.

Requisitos desejados:

Interoperabilidade	Desejável
Segurança de Acesso	Irrelevante
Maturidade	Importante
Tolerância a Falhas	Irrelevante
Recuperabilidade	Desejável
Operacionalidade	Importante
Comportamento em relação ao tempo	Desejável
Comportamento em relação aos recursos	Irrelevante
Modificabilidade	Importante
Testabilidade	Desejável
Adaptabilidade	Importante



Estilos e Tecnologias mais indicados (na ordem):

1. JavaBeans [Estilo baseado em Instâncias]
2. EJB [Estilo baseado em Instâncias]
3. COM [Estilo baseado em Tipos]
4. EJB [Estilo baseado em Tipos]
5. JavaBeans [Estilo baseado em Tipos]
6. COM [Estilo baseado em Instâncias]

Figura 4.33: Estilos e tecnologias sugeridos para um conjunto de requisitos.

Existe uma questão muito importante que precisa ser esclarecida sobre a utilização e adaptação da proposta de XAVIER neste trabalho. A avaliação feita por esse autor sobre os padrões arquiteturais (Tabela 4.29) foi inicialmente baseada na literatura e, posteriormente, refinada por especialistas que, com sua experiência, aumentaram o grau de confiabilidade dessas informações. No contexto do nosso trabalho, o preenchimento da nova avaliação (Tabela 4.30) contou, quase que exclusivamente, com a experiência e conhecimentos adquiridos durante o desenvolvimento desta pesquisa, o que não significa que essa avaliação esteja absolutamente correta. A extração de informações na literatura foi inviabilizada porque os fabricantes das tecnologias praticamente não abordam essas questões. O procedimento recomendado é que a equipe configure permanentemente essa avaliação conforme ela avança em seus projetos, reduzindo cada vez mais as incertezas sobre seus resultados.

4.4 – Conclusão

A especificação da arquitetura de componentes proposta neste capítulo representa um passo para o desenvolvimento de software baseado em componentes, principalmente por definir regras independente de abordagens e que *se baseiam em princípios básicos de DBC*. De uma forma geral, este trabalho se apóia nos conceitos comuns presentes nas abordagens estudadas, visando aproveitá-los na identificação de um conjunto de regras e na construção de mecanismos que se revelam vantajosos durante a montagem da arquitetura de software.

A aplicação deste trabalho em projetos de desenvolvimento de software possibilita a conquista de algumas vantagens interessantes para seus usuários. A primeira delas é a **padronização** dos sistemas que são desenvolvidos. Por empregarmos os mesmos conjuntos de operações na geração das interfaces, uma padronização natural tende a ocorrer com os componentes ao longo dos anos. Isto reduz significativamente o esforço envolvido na *documentação*, *implementação* e *entendimento* dos componentes.

Outra vantagem importante que possivelmente podemos obter é o **treinamento** de pessoas menos qualificadas. Como o desenvolvimento baseado em componentes ainda é desconhecido por muitos desenvolvedores, as regras e mecanismos apresentados servem como um guia para a construção e montagem da arquitetura, mesmo quando não houver uma ferramenta que auxilie essa tarefa.

Voltando nosso foco para a geração das interfaces dos componentes, algumas sugestões precisam ser lembradas para que o máximo de proveito seja tirado deste trabalho. Em primeiro lugar, é altamente recomendado que **todas** as operações propostas para os componentes sejam sempre inseridas nas suas interfaces, independente do negócio em que estejamos trabalhando. O fato de um sistema não usar uma operação não implica que outro também não a usará. Temos que manter em mente que quanto mais completos os componentes forem, mais chances de reutilização eles terão no futuro.

Do mesmo modo, qualquer outra operação que o arquiteto achar importante construir deve ser vista como uma excelente oportunidade para melhorar o componente de negócio, deixando-o cada vez mais apropriado aos contextos de reutilização. Todo este esforço para manter os componentes completos e adequados ajuda a *reduzir* o tempo gasto com adaptações e modificações, o que favorece bastante o cronograma do desenvolvimento.

Veremos no próximo capítulo como a proposta pode ser implementada em um ambiente de reutilização de software, discutindo suas interfaces com o usuário, problemas e questões que foram levantadas durante o desenvolvimento deste trabalho.

Capítulo 5 – Ferramenta para Geração de Componentes de Negócio em um Ambiente de Reutilização

No capítulo anterior, estabelecemos as regras e o mecanismo de geração de componentes de negócio a partir dos requisitos especificados para o software. Para tornar essa proposta viável na prática, o próximo passo é fornecer uma implementação que auxilie a equipe de desenvolvimento durante a elaboração da arquitetura, reduzindo o *tempo* e o *esforço* gasto com a especificação e conexão dos componentes e interfaces.

Como as idéias contidas neste trabalho representam apenas parte do processo de desenvolvimento de aplicações, manter uma implementação isolada não seria muito útil na prática. Nesse sentido, escolhemos a infra-estrutura *Odyssey* (WERNER *et al.*, 2000) como um meio de contextualizar essas idéias, aproveitando suas funcionalidades e o suporte oferecido aos seus usuários.

Este capítulo está dividido em quatro seções. Na seção 5.1, apresentamos o ambiente *Odyssey* e o seu papel no desenvolvimento de software. Na seção 5.2, apresentamos o protótipo implementado, discutindo, ainda, o seu impacto na estrutura do ambiente *Odyssey*. Na seção 5.3, analisamos alguns detalhes técnicos dessa implementação, como suas divisões internas e classes. Finalmente, fazemos as conclusões na seção 5.4.

5.1 – A Infra-Estrutura *Odyssey*

A infra-estrutura *Odyssey* (WERNER *et al.*, 2000) é um ambiente de desenvolvimento de software que oferece ferramentas para apoiar a construção de aplicações através da *reutilização* (Figura 5.34). Seu enfoque principal consiste em especificar e estruturar *modelos de domínio* que possam ser reutilizados posteriormente pela equipe de desenvolvimento. A construção destes modelos – também chamada de *desenvolvimento para reutilização* – representa as atividades de **engenharia de domínio (ED)**, que seguem um processo próprio denominado *Odyssey-DE* (BRAGA, 2000).

O conhecimento contido nos modelos de domínio abrange desde elementos conceituais até artefatos de projeto e implementação (como, por exemplo, classes e diagramas de seqüência). Isso possibilita que o processo de **engenharia de aplicação**

(EA) os reutilize durante novos desenvolvimentos, buscando reduzir o tempo e o esforço gasto no projeto. Para conduzir a equipe, essa forma de trabalho – também conhecida como *desenvolvimento com reutilização* – é executada segundo um processo, chamado *Odyssey-AE* (MILLER, 2000).

De um modo geral, a infra-estrutura *Odyssey* é composta por um conjunto de ferramentas que procuram apoiar as etapas definidas pelo *Odyssey-DE* e *Odyssey-AE*. Dentre elas, podemos citar ferramentas de *planejamento para captura de conhecimento de domínios* (ZOPELARI, 1998); *documentação de artefatos* (MURTA, 1999); *especificação e instanciação de arquiteturas específicas de domínios* (XAVIER, 2001); *navegador inteligente* (BRAGA, 2000); *gerador de código executável* (WERNER *et al.*, 2000); *ferramentas para modelagem e acompanhamento de processos* (MURTA, 2002); *verificação de consistência de modelos UML* (DANTAS, 2001), apoio à *engenharia reversa* (VERONESE e NETTO, 2001) e *suporte a padrões no projeto de software* (DANTAS et al., 2002). Todo o ambiente foi implementado utilizando-se a linguagem JAVA.

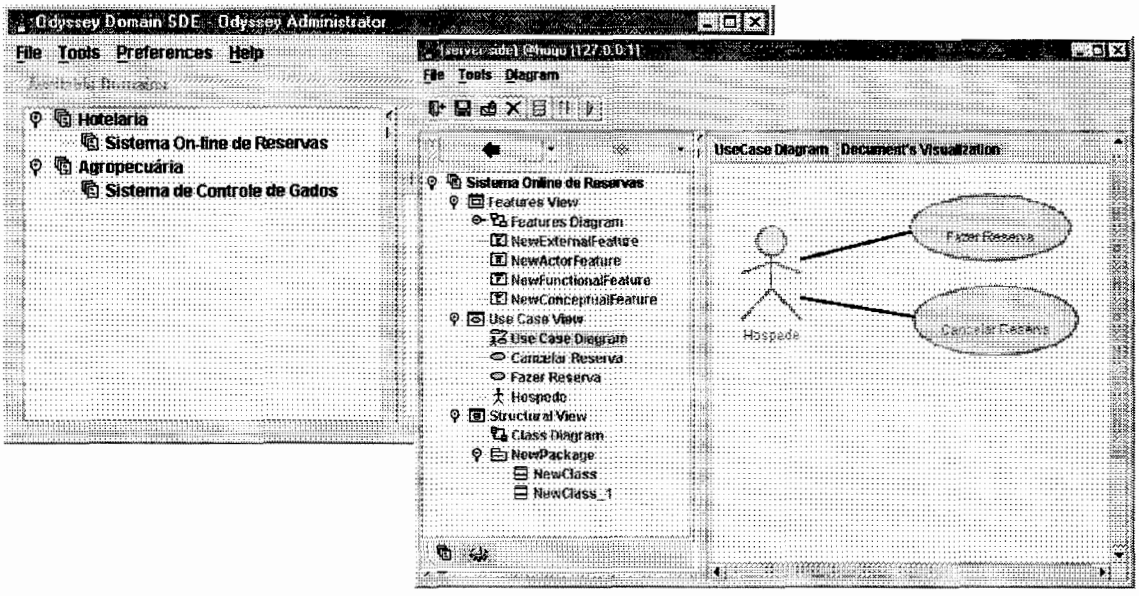


Figura 5.34: Infra-Estrutura *Odyssey*.

5.1.1 – Reutilização de Artefatos no Ambiente *Odyssey*

Como a reutilização de artefatos é um aspecto-chave para o ambiente *Odyssey*, estudamos, nessa seção, um pouco mais sobre essa funcionalidade, procurando expor determinados detalhes que precisam ser entendidos no contexto deste trabalho. Mais adiante, nas seções 5.2.2 e 5.2.6, voltaremos a esse tema, mostrando como a proposta de geração de componentes interfere neste mecanismo.

Em primeiro lugar, a reutilização dos artefatos de um domínio em novas aplicações é uma estratégia que está diretamente relacionada à estrutura interna que armazena esses artefatos dentro do ambiente *Odyssey*. Como os elementos utilizados para modelar um domínio variam entre diferentes níveis de abstração, o ambiente mantém uma rastreabilidade entre os elementos semânticos para contextualizá-los e relacioná-los dentro do domínio (MILLER, 2000) (BRAGA, 2000).

A estrutura mantida pelo ambiente *Odyssey* é caracterizada por níveis decrescentes de abstração, onde os elementos mais abstratos, situados na parte de cima da estrutura, possuem *rastros* para os elementos menos abstratos, situados mais abaixo. Para MILLER (2000), os elementos mais abstratos do domínio são os ***contextos***, os quais representam seus subdomínios e fronteiras com outros domínios. Logo abaixo deles, encontram-se as ***features***[6] (KANG *et al.*, 1990), as quais representam as funcionalidades e conceitos do domínio. A partir das *features*, muitos outros elementos são rastreáveis, como, por exemplo, casos de uso, classes, atores, diagramas, etc. (veremos esta estrutura completa quando estudarmos a sua modificação na seção 5.2.6).

A criação de uma nova aplicação está relacionada a essa estrutura porque, quando o usuário do ambiente seleciona alguns *contextos* e *features* relacionados ao seu interesse (MILLER, 2000), é possível *recortar* o modelo do domínio, trazendo todos os elementos que são rastreáveis a partir deles (Figura 5.35). Nesse procedimento, suas características e relacionamentos são mantidos da mesma forma que foram especificados no modelo do domínio, permitindo ao desenvolvedor que, posteriormente, ajustes possam ser feitos para refletir outros detalhes da aplicação.

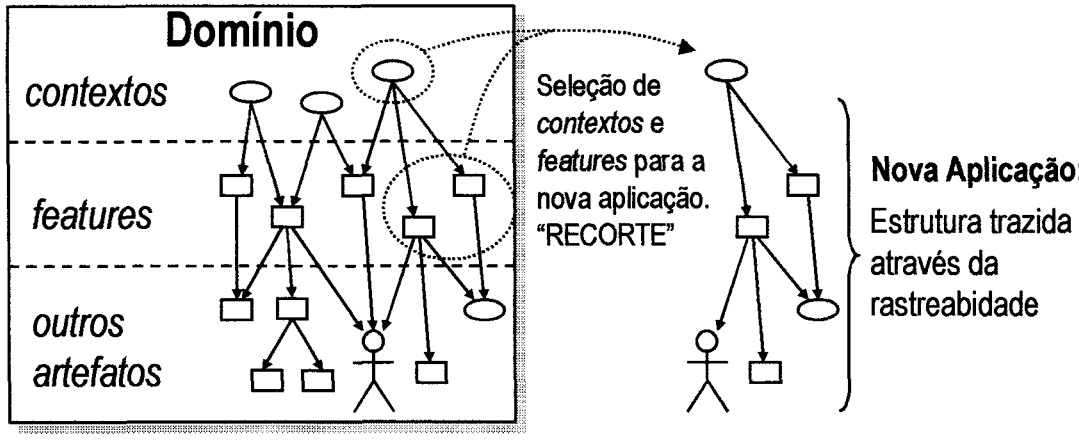


Figura 5.35: Reutilização de artefatos no Odyssey.

[6] *Features* são características essenciais do domínio e representam uma forma de captura da informação semântica do mesmo (SIMOS e ANTHONY, 1998).

5.2 – A Implementação da Ferramenta

Agora que já apresentamos a infra-estrutura *Odyssey* e como a reutilização de artefatos é feita no desenvolvimento de software, podemos iniciar a descrição da implementação feita neste trabalho. Primeiramente, explicamos na **seção 5.2.1** como a modelagem de negócios foi implementada no ambiente, discutindo suas características e influências mais relevantes. Na **seção 5.2.2**, abordamos algumas importantes questões que relacionam a engenharia de domínio e a geração de componentes, e que foram alvo de muitas discussões neste trabalho. Nas **seções 5.2.3 e 5.2.4**, detalhamos, respectivamente, como a geração de componentes é realizada no estilo baseado em tipos e baseado em instâncias, segundo a proposta apresentada no capítulo anterior. Na **seção 5.2.5,** demonstramos a funcionalidade de seleção de estilos e tecnologias a partir dos requisitos não-funcionais da aplicação e, finalmente, na **seção 5.2.6,** detalhamos como a reutilização dos modelos de domínio foi adaptada para que os componentes gerados pudessem participar da seleção de artefatos durante a instanciação de aplicações.

5.2.1 – Modelagem de Negócio

Apresentamos nesta seção como a modelagem de negócio foi implementada na infra-estrutura *Odyssey*, descrevendo suas funcionalidades e requisitos necessários à geração de componentes de negócio. Como estamos tratando de um ambiente que, originalmente, não foi projetado seguindo todos os princípios que hoje caracterizam o desenvolvimento baseado em componentes, precisamos esclarecer, inicialmente, algumas decisões que foram tomadas durante a sua modificação, e que resultaram em algumas mudanças estruturais relevantes.

5.2.1.1 – Considerações Iniciais

Conforme vimos no capítulo anterior, a proposta de geração de componentes desenvolvida neste trabalho utiliza *diagramas de tipos de negócio*, que são os diagramas estáticos adotados pelas abordagens de DBC estudadas. Ao escolhermos a infra-estrutura *Odyssey* como alvo para essa implementação, uma importante discussão é levantada devido à introdução de *tipos de negócio* em um ambiente focado em *features*.

Antes precisamos realizar uma comparação entre esses dois conceitos, procurando por indícios que favoreçam ou impeçam a substituição de um pelo outro. Em termos gerais, a Tabela 5.31 apresenta suas principais diferenças, introduzindo a

idéia de que estamos tratando de elementos que possuem objetivos e contextos diferentes na prática.

Além das diferenças apresentadas na Tabela 5.31, podemos nos apoiar em outro motivo para reforçar o fato de que as *features* não são adequadas para substituir tipos de negócio no desenvolvimento de sistemas: a sua abrangência. Enquanto os tipos de negócio descrevem conceitos que são instanciados por uma aplicação, as *features* abrangem **famílias de aplicações**, descrevendo suas *funcionalidades*, *entidades*, *restrições*, *opções* e *fronteiras com outros domínios* (MILLER, 2000) (BRAGA, 2000). Caso estas fossem utilizadas no lugar dos tipos de negócio, teríamos que limitar e adaptar essa modelagem.

Tabela 5.31: Comparação entre *features* e tipos de negócio.

	Features (KANG *et al.*, 1990) (MILLER, 2000)	**Tipos de Negócio** (CHEESMAN et al., 2001) (BROWN, 2000)
Foco	Domínio.	Aplicação.
Objetivo	Descrevem as características gerais das aplicações de um domínio, destacando as suas semelhanças e diferenças.	Descrevem como o software que será construído deve tratar os conceitos (entidades), seus atributos e relacionamentos em tempo de execução.
Escopo da Análise de Requisitos	Descrevem conceitos que, nem sempre, são tratados pelas aplicações. Normalmente, algumas features são utilizadas apenas para complementar o entendimento sobre o problema.	Descrevem apenas os conceitos que precisam ser tratados pelas aplicações.
Objetos & Instâncias	Os objetos / instâncias de negócio que as aplicações criam e manipulam em tempo de execução não são descritos pelas features. Utilizam-se diagramas de análise próprios para isso (e.g., diagrama de classes, objetos, etc. da *UML*).	Especificam quais tipos de instância os componentes de negócio da arquitetura precisam gerenciar.
Aspectos Sintáticos	Não possuem atributos. Também não possuem *navegação* nas associações.	Possuem atributos (que, inclusive, podem ser parametrizados) e podem ter *navegação* nas associações.

No contexto da infra-estrutura *Odyssey*, todas essas diversidades citadas foram suficientes para manter as *features* e os *tipos de negócio* **coexistindo** dentro do mesmo ambiente. Para isso, mantivemos os tipos de negócio em uma *visão separada (Business View)*, permitindo que estes possam ser utilizados de forma independente pela equipe de desenvolvimento. A Figura 5.36a ilustra essa divisão no *Odyssey*.

Para evitar que os tipos de negócio ficassem soltos e descontextualizados em relação as *features* dentro do ambiente, identificamos a necessidade de relacioná-los para aumentar a coesão e integridade da modelagem. Essa *rastreabilidade* é permitida somente para as ***features conceituais*** do domínio, principalmente, porque são elas as responsáveis pela descrição dos conceitos do software, já que os tipos de negócio também são conceitos em um nível diferente de abstração. A Figura 5.36b demonstra essa ligação dentro da estrutura do *Odyssey*. Para os casos de uso, uma rastreabilidade semelhante existe para as *features funcionais* (BRAGA, 2000).

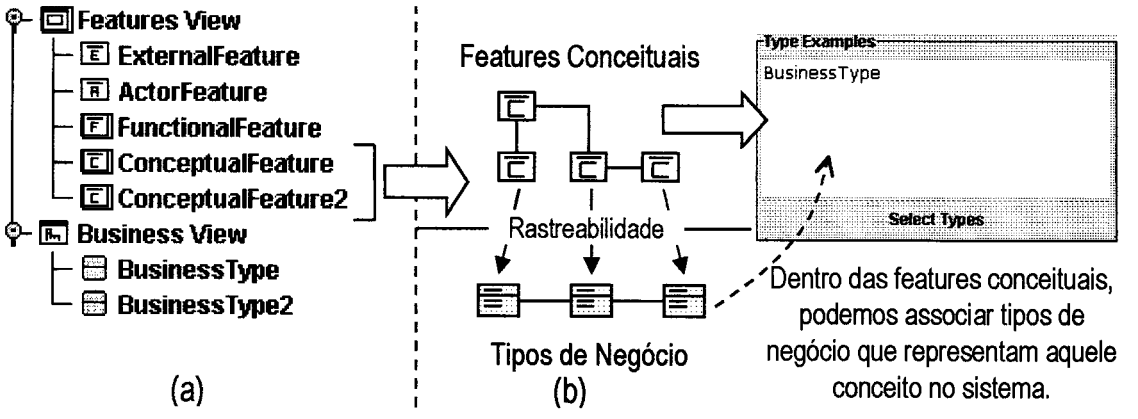


Figura 5.36: Visão de features e de negócios.

5.2.1.2 – Diagramas de Tipos de Negócio

Os diagramas de tipos de negócio implementados no *Odyssey* seguem o mesmo padrão *UML* das abordagens de DBC estudadas. Nele são modeladas as associações, multiplicidades e navegações entre os tipos de negócio, que são usadas na geração dos componentes de negócio. A Figura 5.37 apresenta a modelagem do sistema de hotelaria exemplificado no capítulo anterior.

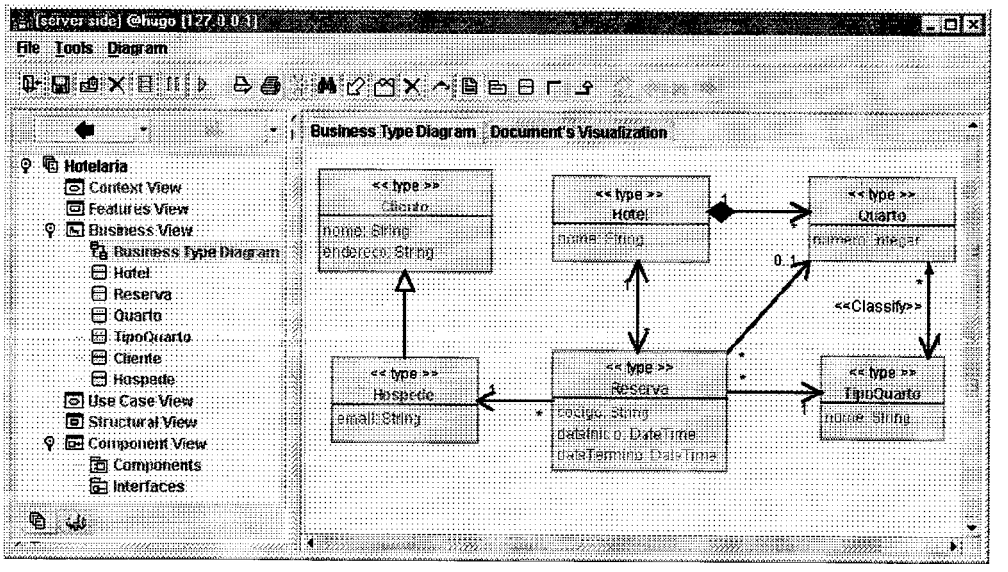


Figura 5.37: Diagrama de Tipos de Negócio no *Odyssey*.

A janela de edição de um tipo de negócio possui todos os campos necessários a sua completa especificação. Dentre eles, podemos citar o campo *plural*, as *dependências de identificação única* e os *parâmetros de criação* (destacados na Figura 5.38).

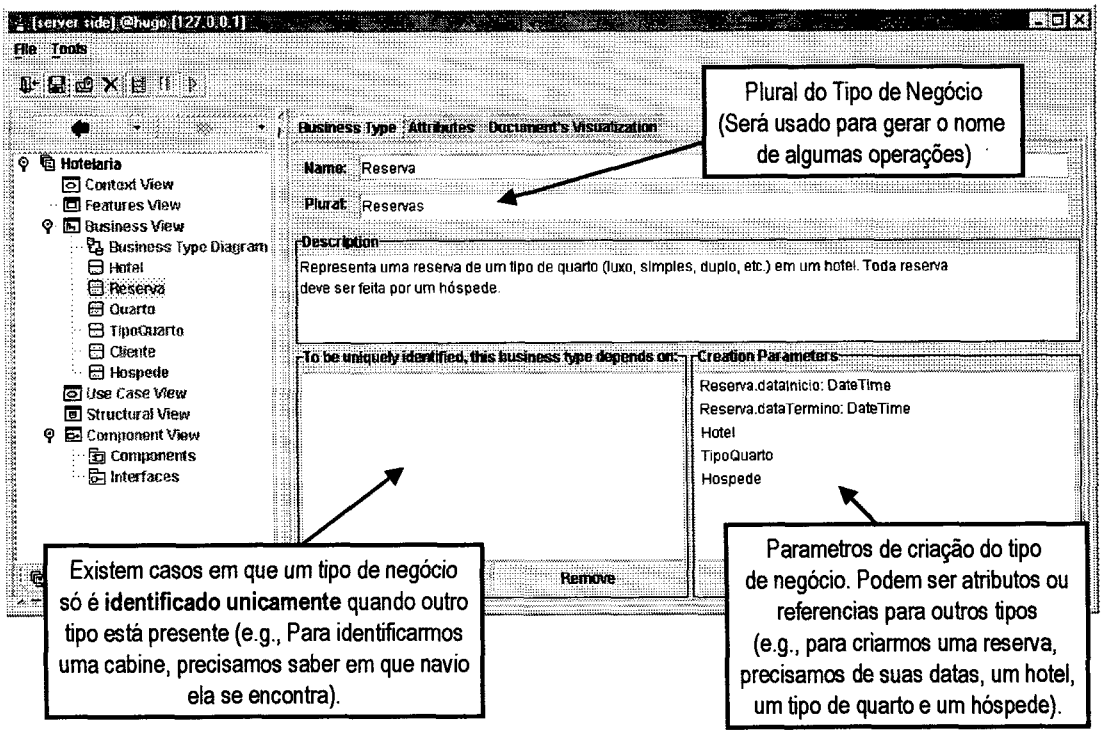


Figura 5.38: Janela de edição de um tipo de negócio.

Para editarmos os atributos de um tipo de negócio, utilizamos a janela vista na Figura 5.39, onde definimos seu nome e tipo. Além disso, podemos defini-los como chave de negócio ou se eles possuem parâmetros (atributos parametrizados).

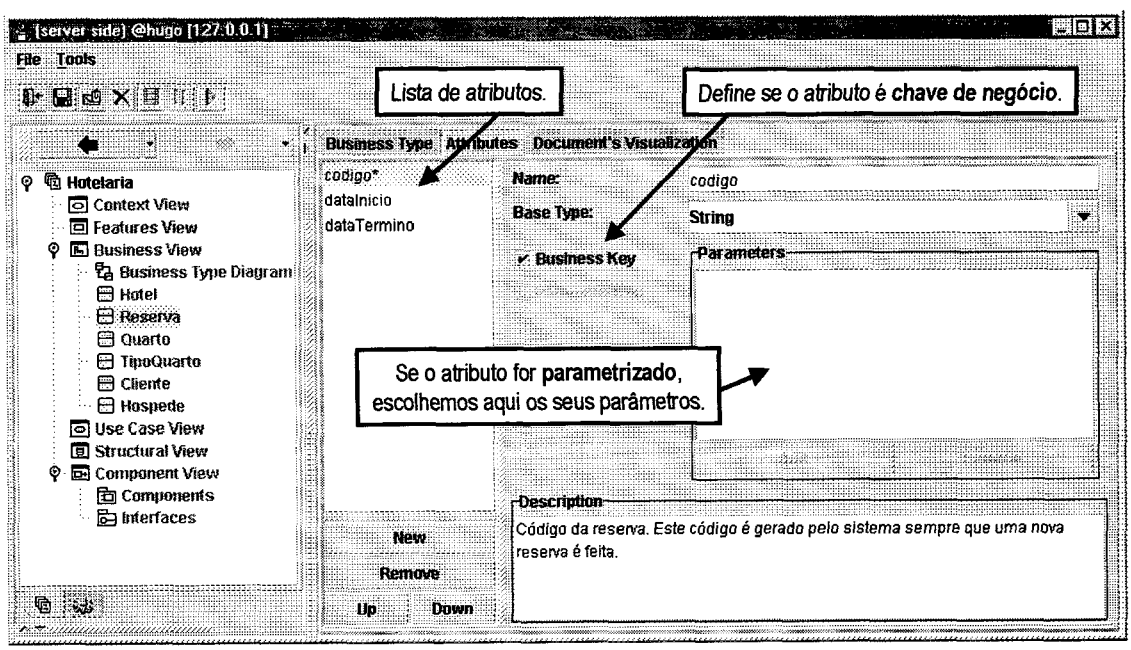


Figura 5.39: Janela de edição dos atributos de um tipo de negócio.

5.2.1.3 – Casos de Uso

Embora os casos de uso já existissem antes da implementação desta proposta, tivemos que incluir na sua especificação uma nova janela para edição dos seus passos internos. Conforme vimos no capítulo anterior, cada passo pode possuir uma ou mais ações que descrevem – em um nível mais baixo de abstração – como eles se comportam em relação aos tipos de negócio do software. A Figura 5.40 apresenta esta janela.

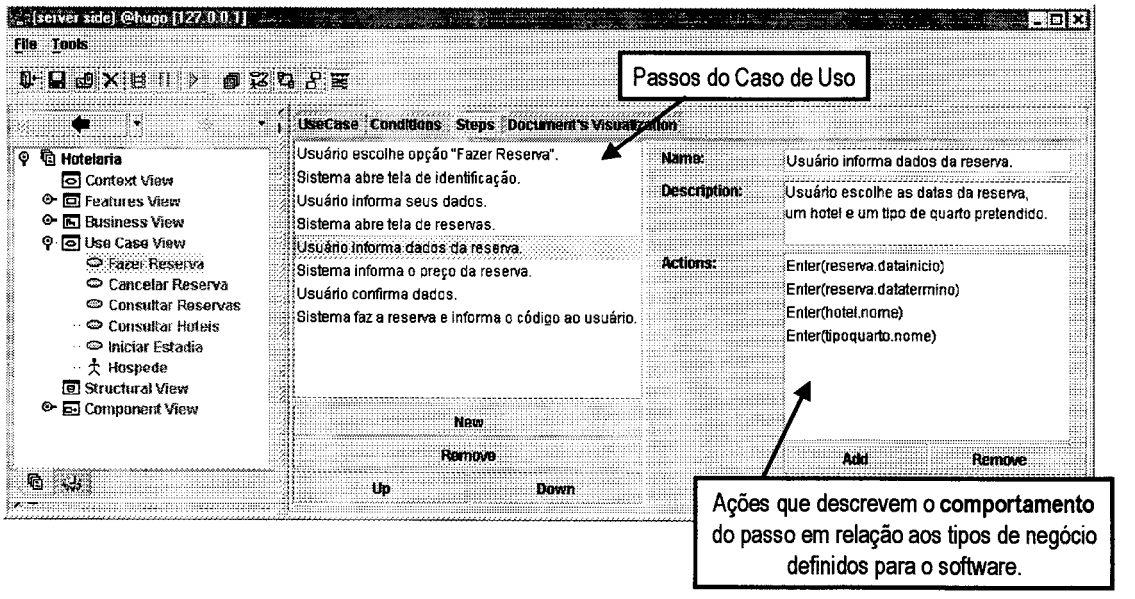


Figura 5.40: Especificação dos passos de um caso de uso.

5.2.2 – Considerações sobre a Engenharia de Domínio

A implementação da proposta de geração de componentes de negócio no ambiente *Odyssey* traz algumas discussões referentes à **engenharia de domínio** que não podem ser ignoradas ao longo deste trabalho. Para entendermos do que se trata, é importante lembrar que uns dos principais objetivos do *Odyssey* é reutilizar artefatos pré-elaborados do domínio nas aplicações que são desenvolvidas.

Essa questão, especificamente a elaboração de artefatos reutilizáveis no nível do domínio, revelou-se um ponto complexo para este trabalho, principalmente, por tratar de *artefatos genéricos* que podem ser empregados em uma ou mais aplicações do domínio. Como a proposta apresentada no capítulo anterior utiliza a modelagem referente a *uma aplicação específica* para gerar seus componentes de negócio, implementar este mecanismo no nível do domínio pode gerar muitas discussões.

Embora muitas aplicações dentro de um mesmo domínio compartilhem características muito semelhantes, não podemos assumir que seus componentes internos

sejam sempre idênticos entre si. Existem detalhes que variam de aplicação para aplicação, e que são adotados de acordo com cada objetivo pretendido. Muitos desses detalhes são influenciados tanto por aspectos *não-funcionais* (como vimos na seção 4.3), quanto por descrições mais sutis, como, por exemplo, a navegabilidade entre as instâncias ou atributos específicos de algumas aplicações.

Uma solução trivial para este problema seria não implementar a geração de componentes no nível do domínio, deixando-a somente no nível das aplicações. Além de pouco vantajosa, essa solução não disponibilizaria componentes para as novas aplicações aproveitarem, o que inibiria a reutilização e se tornaria incoerente com a proposta básica do desenvolvimento baseado em componentes.

Para permitirmos que as aplicações de um domínio compartilhem componentes e incentivem essa reutilização, adotamos uma solução simples para esse problema. No nível do domínio, a geração existe para que a equipe de desenvolvimento possa montar um *repositório de componentes (biblioteca)*, realizando diversas gerações com diferentes características (variando navegações, estilos arquiteturais, etc.). Cada geração é armazenada em um **pacote** que guarda – em sua descrição – essas características e o objetivo daquele conjunto de componentes.

Visando exemplificar esse procedimento, utilizamos, a seguir, o nível do domínio para realizar a geração de componentes de negócio nos dois estilos arquiteturais conhecidos. Ainda sabendo que a mesma geração pode ser feita diretamente nas aplicações, faremos no domínio para que, ao final, possamos mostrar como suas aplicações podem aproveitar estes componentes durante o seu desenvolvimento.

5.2.3 – Geração de Componentes no Estilo baseado em Tipos

Conforme vimos no capítulo anterior, dois tipos de componente são gerados no estilo baseado em tipos: os *componentes gerentes de instâncias* e os *componentes de processo*. A seguir, apresentamos como essa implementação é feita, destacando, para cada um deles, seus passos e atividades principais.

5.2.3.1 – Geração dos Componentes Gerentes de Instâncias

Para apresentarmos a geração dos componentes gerentes de instâncias, utilizamos o diagrama de tipos de negócio de hotelaria apresentado na Figura 5.37.

Como sabemos, a geração destes componentes é iniciada com o agrupamento dos tipos de negócio utilizando as regras da Tabela 4.1.

Acessando esta funcionalidade através do menu *Tools* ▸ *Component Architecture* ▸ *Styles* ▸ *Type-based* ▸ *Generate Type Managers*, a janela de agrupamento é aberta (Figura 5.41a). Do lado esquerdo, temos uma árvore com os grupos pré-formados pelas regras, onde são indicados os candidatos fortes (**R** – *Recommended*) e os candidatos fracos (**S** – *Suggested*) de cada grupo. Ao clicarmos em um dos grupos na árvore, seus detalhes são apresentados do lado direito da janela. Como as regras de agrupamento são sugestões para o arquiteto, utilizamos *checkboxes* para que ele possa selecionar os candidatos que ele preferir.

Quando a *checkbox* é marcada, o candidato passa a fazer parte do grupo que está selecionado. Com isso, todos os candidatos que, antes, faziam parte do grupo do candidato, são trazidos, agora, para o grupo em que ele foi inserido. No exemplo da Figura 5.41b, o *Tipo de Quarto* foi trazido (como candidato) para o grupo do *Hotel*, porque ele é sugerido a permanecer no mesmo grupo onde o *Quarto* está.

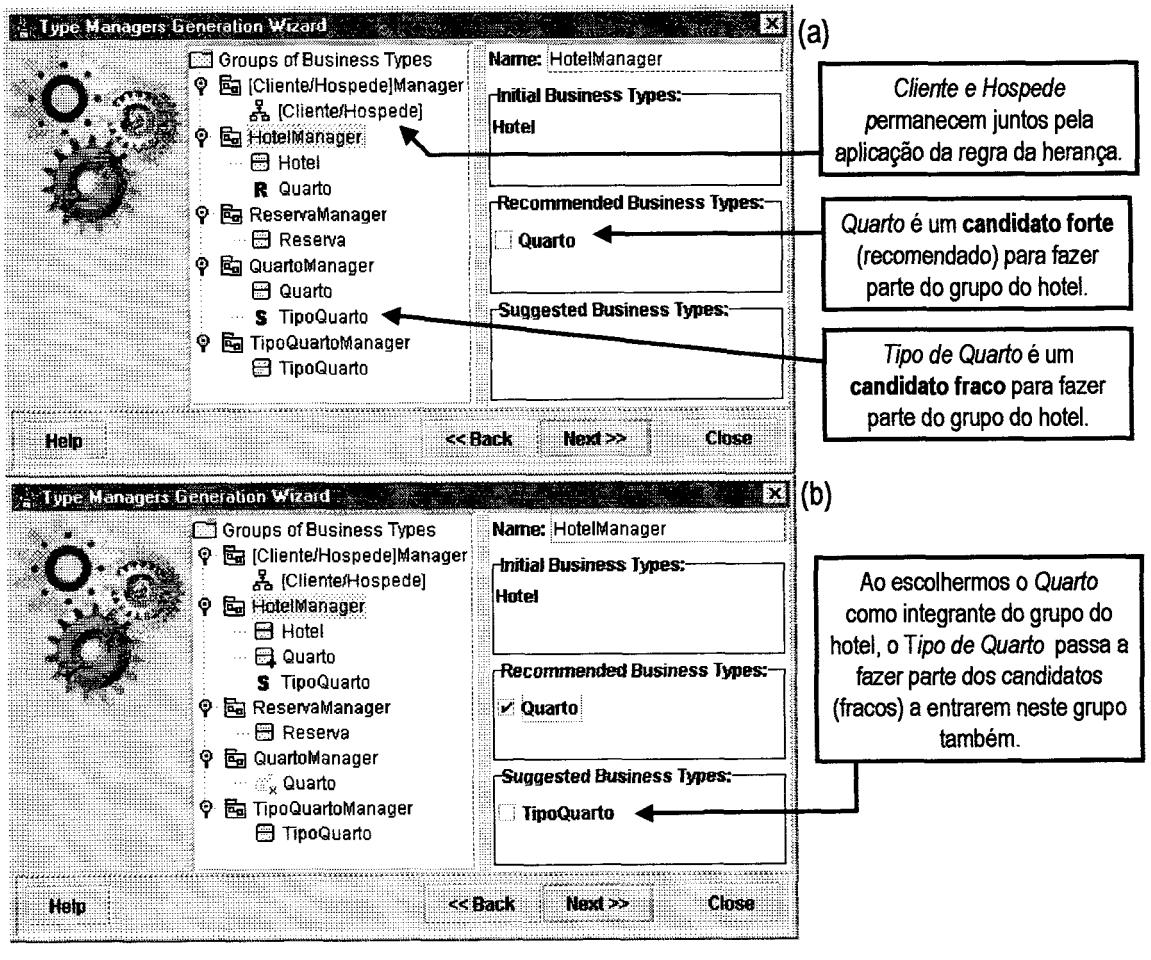


Figura 5.41: Janela de agrupamento (exemplo de hotelaria).

A implementação dessa janela de agrupamento é bastante complexa devido aos cuidados que ela deve tomar durante as ações do usuário. Os principais pontos que devem ser lembrados são:

- Se o arquiteto resolver mudar de idéia durante o agrupamento e querer voltar aos grupos que ele tinha anteriormente, a janela deve ser capaz de retroceder e desfazer qualquer agrupamento, devolvendo candidatos que foram levados e reposicionando-os nos seus devidos lugares;
- Quando um candidato é escolhido para fazer parte de um grupo qualquer (marcando sua *checkbox* nesse grupo), todos os outros grupos onde ele também era sugerido não podem mais acolhê-lo, já que ele só pode estar em um grupo por vez;
- Quando um candidato é levado para outro grupo (como ocorreu na Figura 5.41b), a janela deve verificar se aquele candidato já está sendo sugerido no grupo, evitando que duas *checkboxes* fiquem disponíveis para o mesmo candidato ao mesmo tempo.

Após o agrupamento dos tipos de negócio, uma janela de preferências é aberta, para que o arquiteto escolha algumas opções sobre a arquitetura, como o *tipo de dado da chave técnica* e o *tratamento de remoções de instâncias* (visando manter a integridade referencial dos dados da aplicação, conforme descrito na seção 4.2.1.4). Esta janela está detalhada na Figura 5.42.

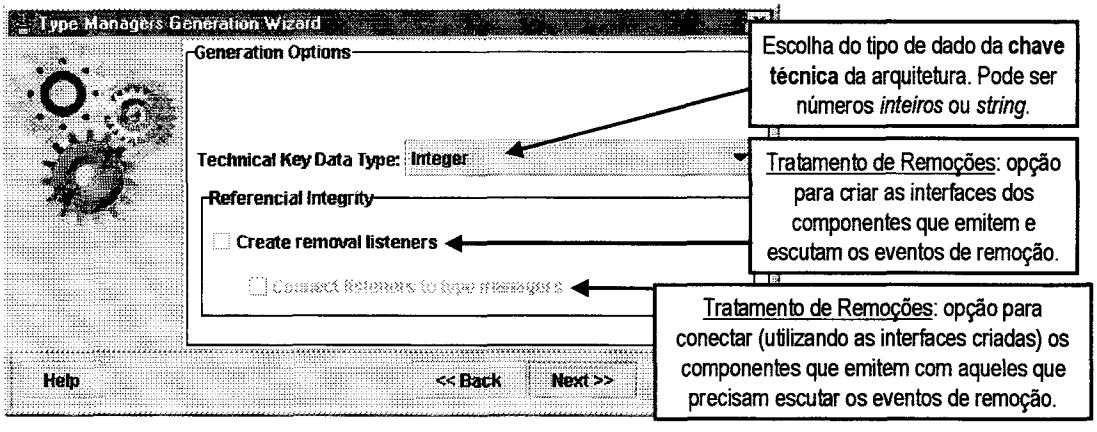


Figura 5.42: Janela de opções sobre a arquitetura.

Para concluir a geração dos componentes, a janela vista na Figura 5.43 é utilizada para que o arquiteto defina o nome dos **pacotes** onde os componentes e as interfaces serão inseridos. Além disso, um pequeno relatório com algumas características da geração é montado, visando documentar o conteúdo destes pacotes.

Caso deseje, o arquiteto pode editar esse relatório para complementá-lo com mais informações.

Após concluir o procedimento, os componentes e as interfaces são gerados conforme visto na Figura 5.44. Cada operação das interfaces é carregada com suas informações semânticas, que descrevem, entre outras coisas, a **ação** que ela efetua sobre os tipos de negócio. Esses dados serão utilizados na geração dos componentes de processo descritos a seguir.

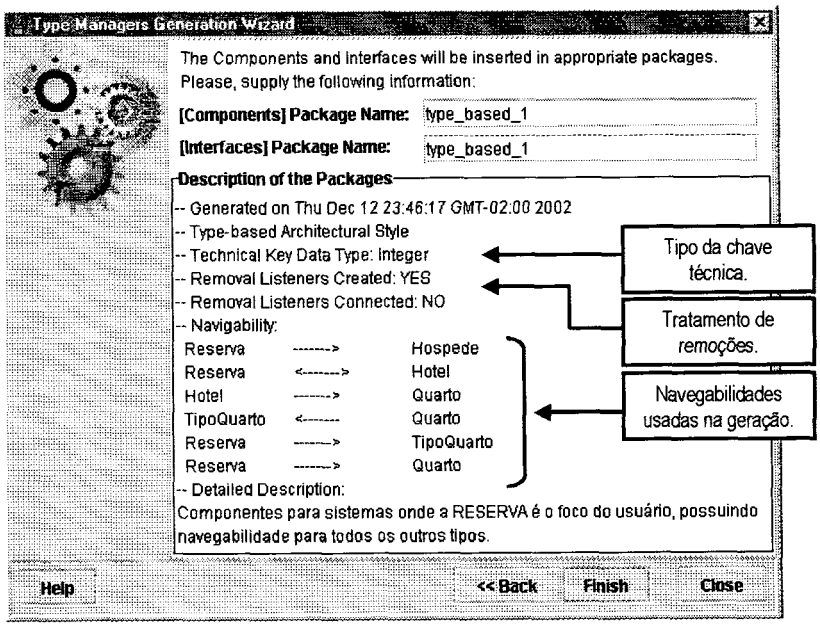


Figura 5.43: Informações sobre os pacotes onde os componentes e as interfaces serão inseridos.

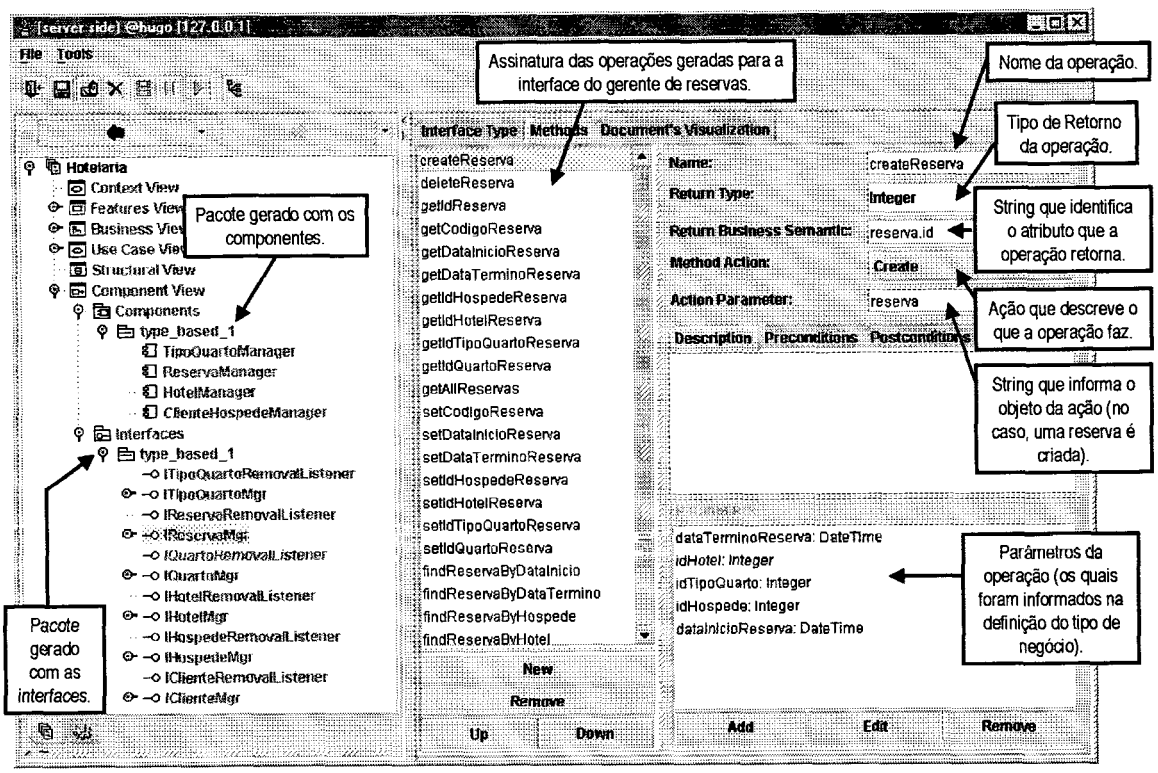


Figura 5.44: Componentes e interfaces gerados.

5.2.3.2 – Geração dos Componentes de Processo

A geração dos componentes de processo utiliza a especificação dos casos de uso como um guia para acessar os componentes gerentes de instância criados. Para isso, precisamos definir quais casos de uso um componente realiza, associando-os conforme apresentado na Figura 5.45 (painel *Use Case Realization*).

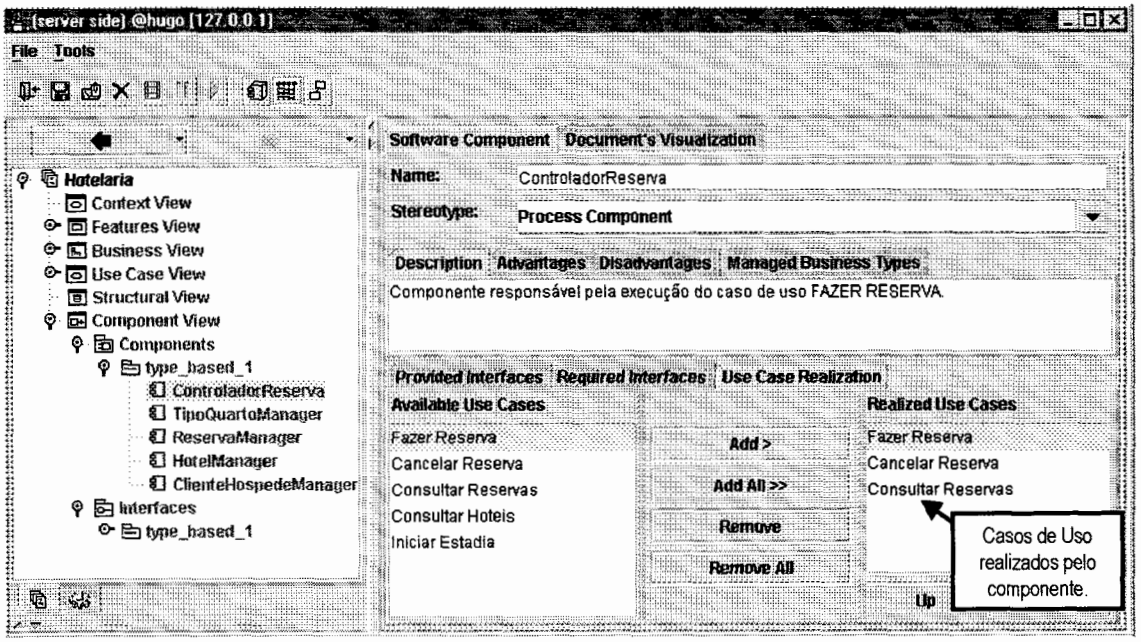


Figura 5.45: Componentes de processo realizam casos de uso.

A geração é iniciada pelo menu *Tools* ▸ *Component Architecture* ▸ *Styles* ▸ *Type-based* ▸ *Generate Interactions From Use Case Realization*, onde um *wizard* é aberto. Inicialmente, o arquiteto precisa escolher qual componente será especificado pelo algoritmo. Após escolhermos o componente de processo desejado (no caso da Figura 5.45, o componente que utilizamos é o *ControladorReserva*), precisamos selecionar quais dos casos de uso associados a ele estamos interessados em gerar (Figura 5.46a).

O passo seguinte consiste em escolher apenas as interfaces que necessariamente o algoritmo precisa utilizar (Figura 5.46b). Essa seleção foi adotada por questões de melhoria de desempenho do algoritmo, já que, na maioria dos casos, muitas interfaces da arquitetura não possuem relação com a execução dos processos de negócio. As interfaces que, normalmente, precisam ser selecionadas são aquelas que possuem métodos para a manipulação das instâncias dos tipos de negócio. Em caso de dúvida, o arquiteto pode escolher quantas interfaces ele quiser, o que influenciará apenas o tempo que o algoritmo leva para gerar as seqüências.

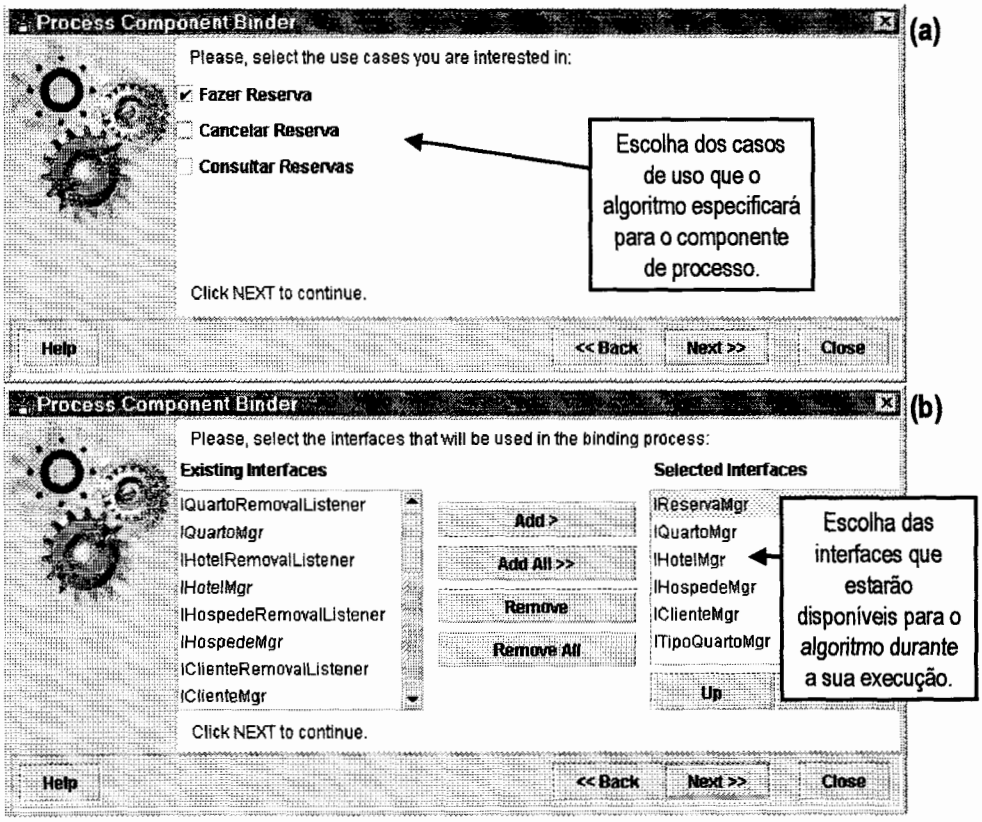


Figura 5.46: Passos para a geração de um componente de processo.

Ao concluir a geração, o resultado obtido pode ser visto na Figura 5.47, onde o diagrama de interação é gerado dentro do componente de processo, descrevendo como um cliente (representado pela interface *IClient*) deve utilizar esse serviço na arquitetura. Caso algum erro ocorra durante a geração, o erro é informado para que o arquiteto procure uma solução para o problema.

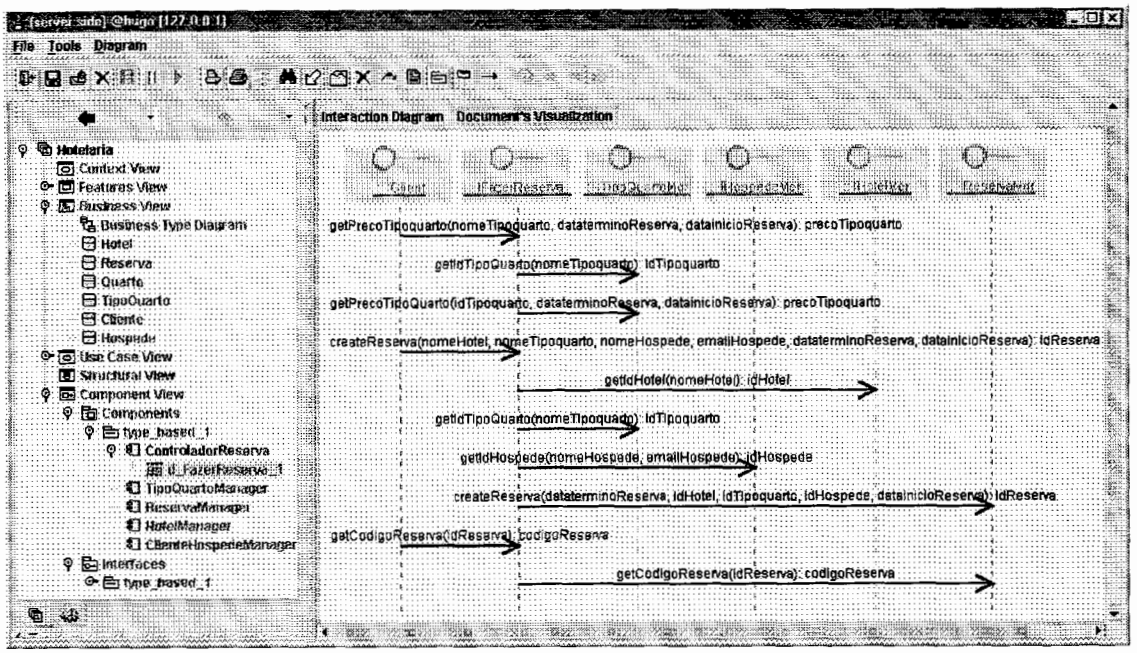


Figura 5.47: Diagrama gerado para o caso de uso "Fazer Reserva".

5.2.4 – Geração de Componentes no Estilo baseado em Instâncias

Conforme vimos no capitulo anterior, a geração de componentes de negócio no estilo baseado em instâncias consiste em criar, para cada tipo de negócio, um *componente de entidade* e um *componente-coleção* na arquitetura. Utilizando o mesmo diagrama de hotelaria da Figura 5.37, essa geração resulta nos componentes e interfaces vistos na Figura 5.48.

Para realizar essa geração, o arquiteto deve acessar o menu *Tools* ▸ *Component Architecture* ▸ *Styles* ▸ *Instance-based* ▸ *Generate Instance Managers* e a única informação que ele deve fornecer é o **nome** e a **descrição** dos pacotes onde os componentes serão inseridos (essa janela é semelhante a apresentada na Figura 5.43, porém, o relatório montado contém os detalhes do estilo baseado em instâncias).

Mesmo não havendo componentes de processo neste estilo, todas as operações são carregadas com as **ações** que elas efetuam sobre os tipos de negócio modelados. Esse preenchimento foi feito para permitir que futuros trabalhos também utilizem estes dados para saber o papel de cada operação no contexto de negócio.

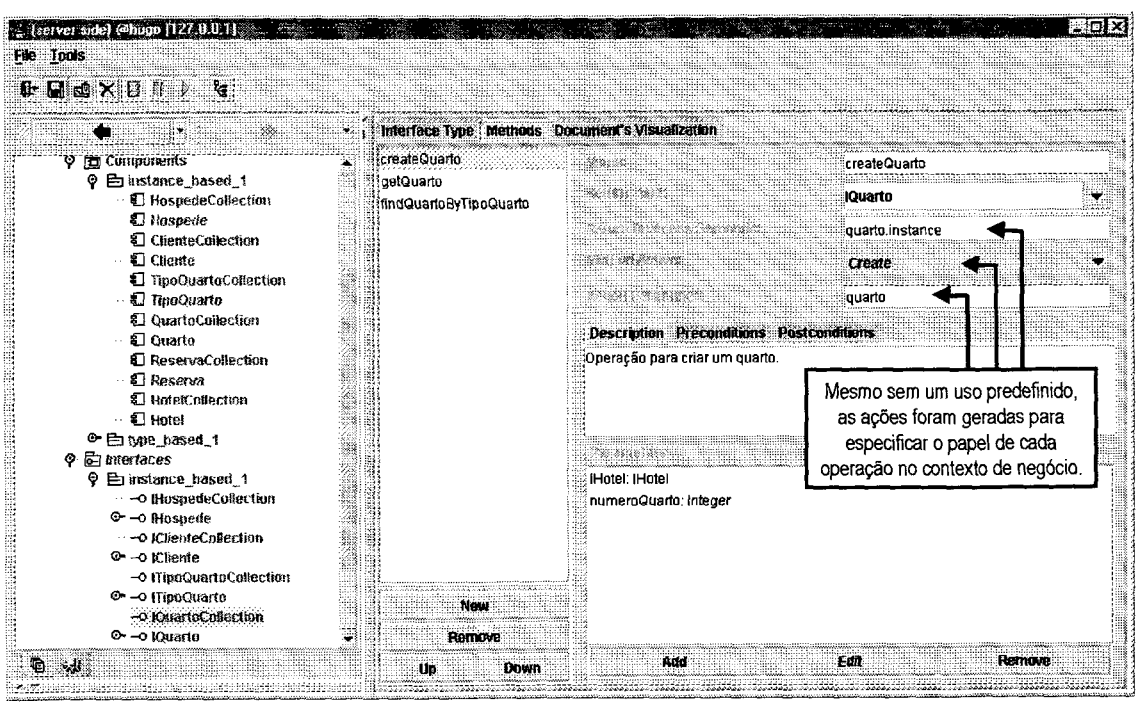


Figura 5.48: Componentes de hotelaria no estilo baseado em instancias.

5.2.5 – Decisão e Escolha dos Estilos e Tecnologias

Para apoiarmos a decisão do arquiteto durante a escolha dos estilos e tecnologias que podem ser empregadas no desenvolvimento das aplicações, implementamos o mecanismo que se baseia no trabalho de XAVIER (2001) e foi adaptado conforme visto

no capitulo anterior. Inicialmente, precisamos cadastrar todas as tecnologias de componente em que estamos interessados, utilizando a janela vista na Figura 5.49 (menu *Tools* ▸ *Component Architecture* ▸ *Technology* ▸ *Add/Remove Component Technologies*).

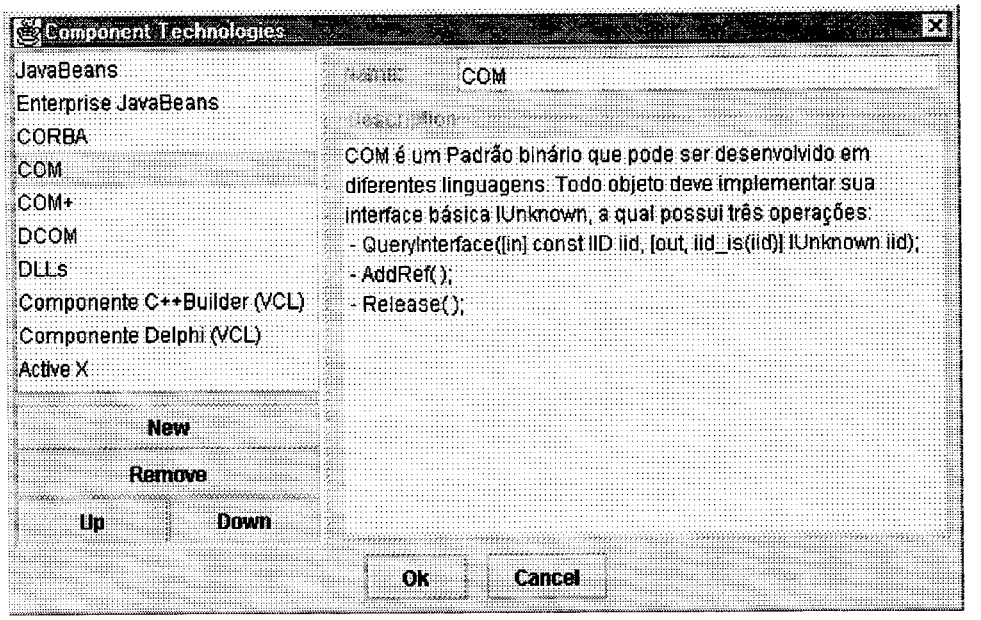


Figura 5.49: Janela de cadastro de tecnologias de componente.

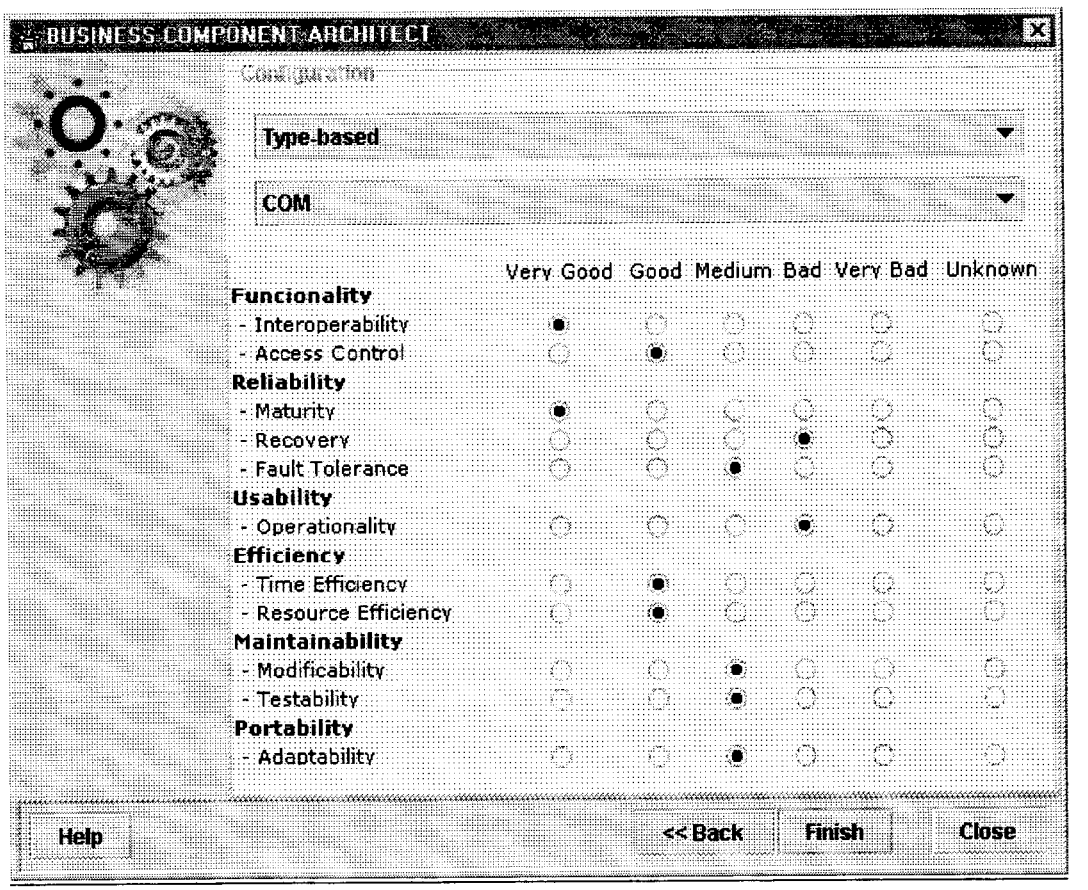


Figura 5.50: Janela de avaliação das tecnologias nos estilos arquiteturais.

Com as tecnologias disponíveis, o próximo passo consiste em avaliá-las para cada um dos dois estilos arquiteturais abordados na geração de componentes (estilo baseado em tipos e o baseado em instâncias). Esta avaliação é feita na janela da Figura 5.50 (menu *Tools* ▸ *Component Architecture* ▸ *NonFunctional Requirements & Technology Evaluation Support*), onde os onze atributos de qualidade devem ser conceituados separadamente. É importante lembrar que a coleta destas informações pode ser realizada com os especialistas nas tecnologias, permitindo, também, que refinamentos contínuos sejam feitos após o término de cada projeto.

Para saber qual o estilo e a tecnologia mais indicados para uma aplicação em particular, o arquiteto precisa expor seus desejos para cada um dos onze atributos de qualidade utilizados. Isto é feito indicando-os como Importante, Desejável ou Irrelevante para o software (Figura 5.51).

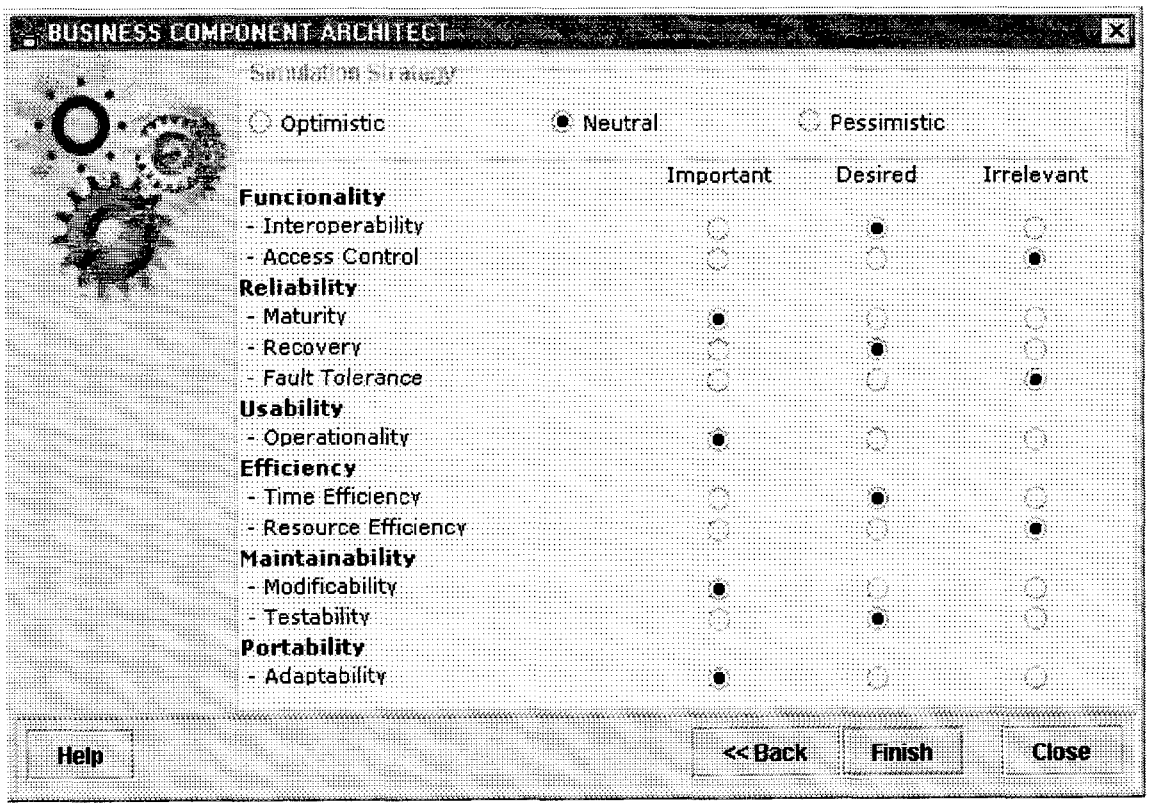


Figura 5.51: Atributos de qualidade desejados para o software.

Utilizando a base de conhecimento da Tabela 4.29 (onde apenas as tecnologias *EJB*, *JavaBeans* e *COM* estão avaliadas), apresentamos na Figura 5.52 os resultados obtidos segundo os desejos da Figura 5.51. Para cada linha retornada, temos um valor que indica a *distância vetorial* da tecnologia e do estilo até o objetivo pretendido. A visão geral dada por esses resultados permite que o arquiteto decida sobre o melhor caminho a ser seguido no desenvolvimento.

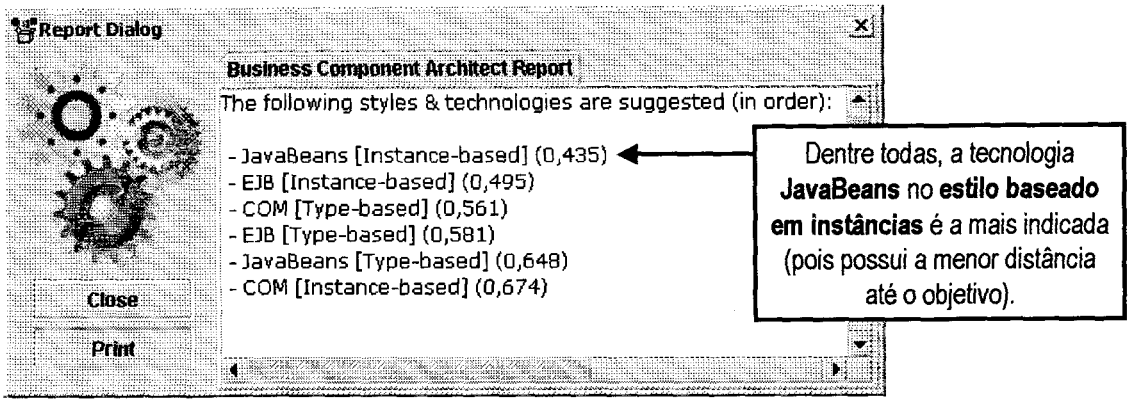


Figura 5.52: Resultados obtidos para os desejos do arquiteto.

5.2.6 – Reutilização de Artefatos em Novas Aplicações

Conforme vimos na seção 5.1.1, uma das estratégias mais importantes da infra-estrutura *Odyssey* é a reutilização dos artefatos de um domínio no desenvolvimento de novas aplicações (MILLER, 2000) (BRAGA, 2000). Como originalmente não existiam *componentes* e *tipos de negócio* em sua estrutura semântica, explicamos, a seguir, como esse mecanismo de reutilização foi *adaptado* para aproveitar esses novos elementos durante a instanciação de aplicações.

Para compreendermos melhor esse procedimento, precisamos observar como a seleção dos artefatos é realizada dentro do ambiente *Odyssey* e, ainda, como suas estruturas internas interferem nessa atividade. Vimos no início deste capítulo que, ao iniciar a criação de uma aplicação, o primeiro conjunto de elementos que deve ser selecionado pelo projetista contém os ***contextos*** relacionados ao objetivo do software, os quais definem o escopo do domínio, os relacionamentos com outros domínios e os principais atores envolvidos nas suas atividades. A Figura 5.53 mostra a janela usada nesta seleção.

A seleção dos contextos encontra-se em primeiro lugar porque estes elementos são os mais abstratos dentre todos os existentes na estrutura semântica utilizada pelo *Odyssey*. Essencialmente, cada contexto representa um subconjunto do domínio que é mapeado para outros elementos da modelagem, formando uma estrutura em árvore que está retratada na Figura 5.54. Como vimos anteriormente, a rastreabilidade desta estrutura é considerada a chave para o sucesso da reutilização dos artefatos, porque ela permite que um ***recorte completo*** seja feito, concentrando-se apenas nas camadas mais superiores da árvore.

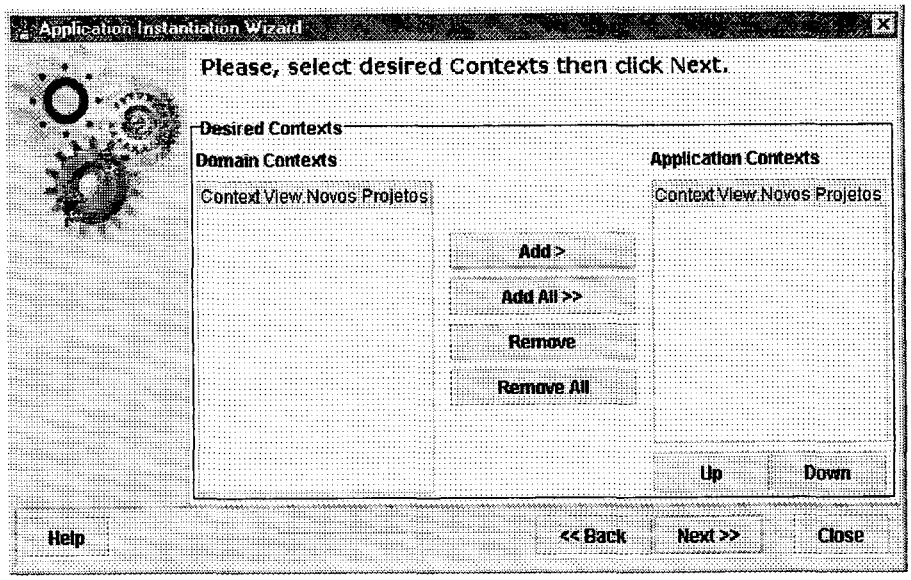


Figura 5.53: Janela de seleção de contextos.

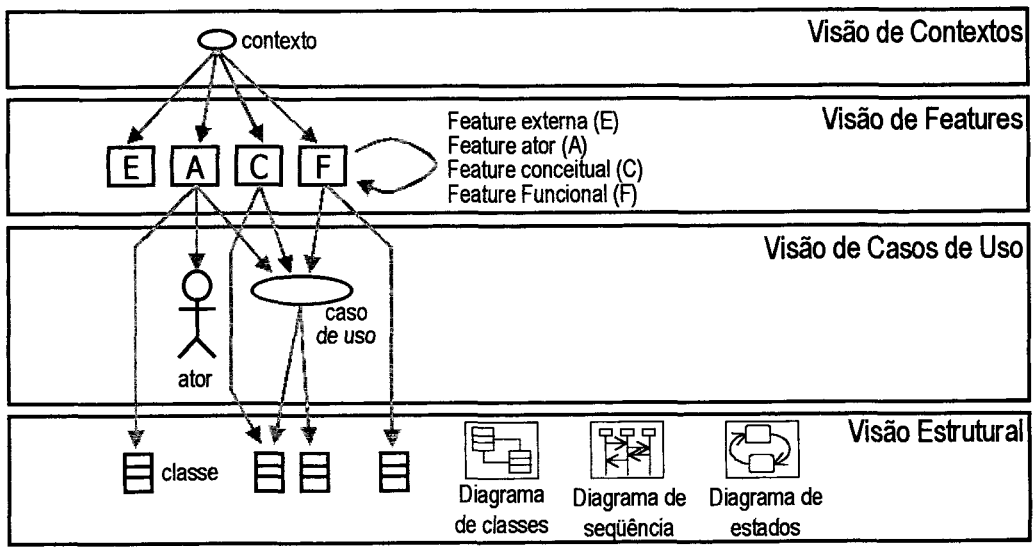


Figura 5.54: Navegabilidade entre os elementos semânticos do *Odyssey*.

Observando a Figura 5.54, percebemos que os contextos do domínio são rastreáveis para as *features*, que podem ser *externas* (E), *atores* (A), *conceituais* (C) e *funcionais* (F). Desta forma, o recorte proposto por MILLER (2000) se estende até o nível das *features*, carregando todos os artefatos que forem rastreáveis a partir delas (casos de uso, classes, etc.). A janela de seleção das *features* pode ser vista na Figura 5.55.

Agora que a proposta de MILLER (2000) está esclarecida sobre a instanciação das aplicações, podemos discutir como a introdução de tipos de negócio e componentes influenciou nesse procedimento. Em primeiro lugar, temos que lembrar que na *seção 5.2.1* explicamos que as *features conceituais* passaram a ser rastreáveis para os tipos de negócio no ambiente. Além dessa ligação, não podemos nos esquecer que os

componentes de negócio gerados com esta proposta foram concebidos a partir dos tipos de negócio e casos de uso modelados. Conseqüentemente, todo componente gerado deve manter uma rastreabilidade para os elementos que o originaram. A nova árvore de rastreabilidade do ambiente *Odyssey* está ilustrada na Figura 5.56.

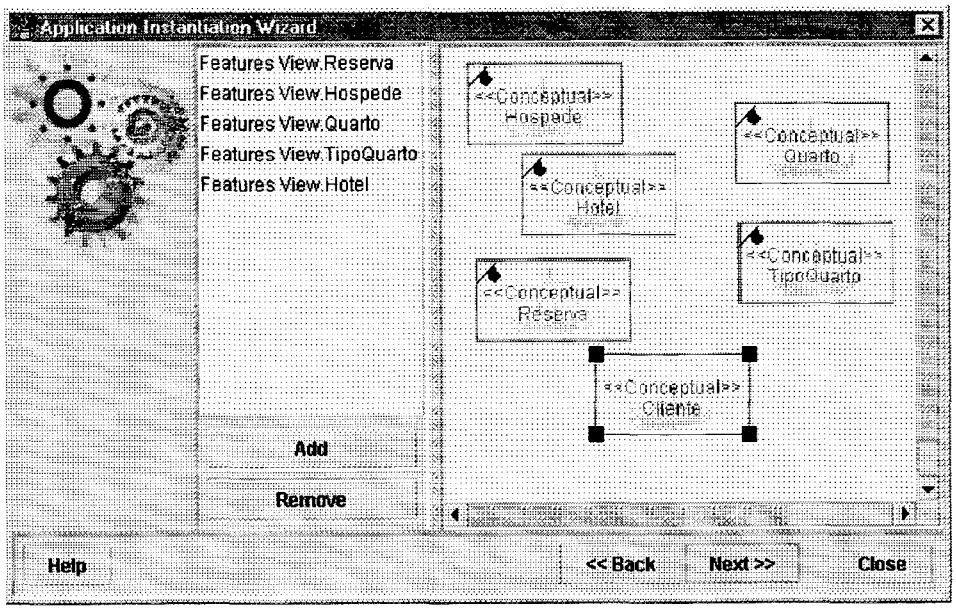


Figura 5.55: Janela de seleção de features.

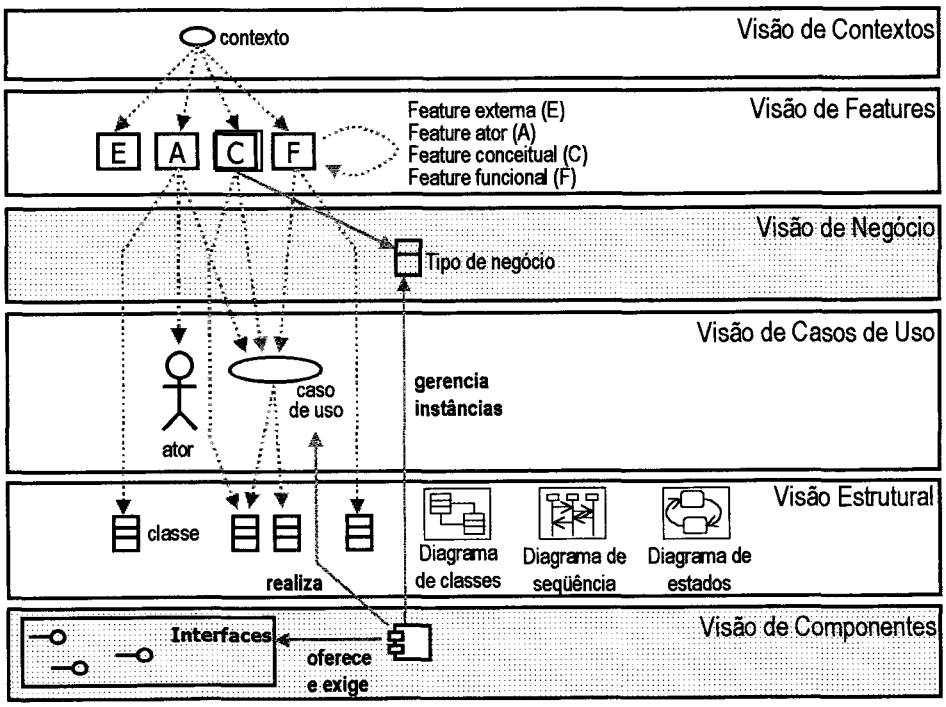


Figura 5.56: Navegabilidade contendo componentes e tipos de negócio.

Observando pelo ponto de vista dos tipos de negócio, a instanciação de aplicações não sofre muitas mudanças com a nova estrutura, pois as *features conceituais* são navegáveis para esses elementos. Isto permite que eles sejam levados como se fossem outros artefatos quaisquer, como classes ou casos de uso, por exemplo. Por outro

lado, a questão que existe por trás dos componentes é mais complicada e merece mais atenção.

Em sua essência, a proposta que desenvolvemos nesta pesquisa assume que os componentes gerados devem trabalhar **juntos** em uma mesma aplicação. Esse é um dos motivos que fizeram com que conectássemos e empacotássemos esses componentes de negócio, montando um "**kit**" que pode ser empregado em diferentes projetos. Isso significa que não podemos carregar os componentes para as aplicações da mesma forma que os outros artefatos o são. Caso contrário, componentes provenientes de diferentes gerações se misturariam entre si, formando um conjunto incompatível de elementos na arquitetura.

Visando obter uma solução plausível para esse problema, permitimos que o projetista, após selecionar os *contextos* e as *features* desejados para a aplicação, escolha os **pacotes** de componente que foram gerados e desenvolvidos no domínio. Nesse caso, só são mostrados os pacotes que possuem componentes relacionados com os tipos de negócio e casos de uso que são rastreáveis a partir das *features* pré-selecionadas.

A janela de seleção de pacotes está ilustrada na Figura 5.57. Ao apertar o botão finalizar (*finish*) da janela, a aplicação é criada, reutilizando todos os artefatos do domínio que estão relacionados com o conjunto de *contextos*, *features* e *pacotes* escolhidos. A Figura 5.58 apresenta a aplicação resultante, contendo os componentes da geração realizada no estilo baseado em tipos no domínio de hotelaria.

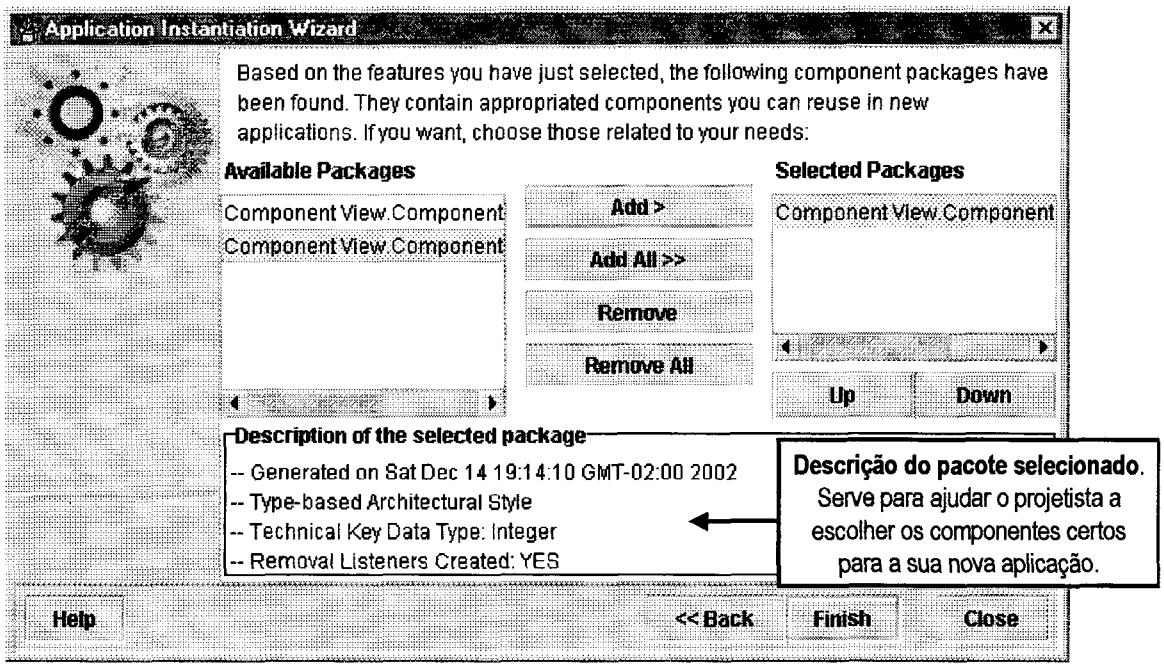


Figura 5.57: Seleção dos pacotes de componentes.

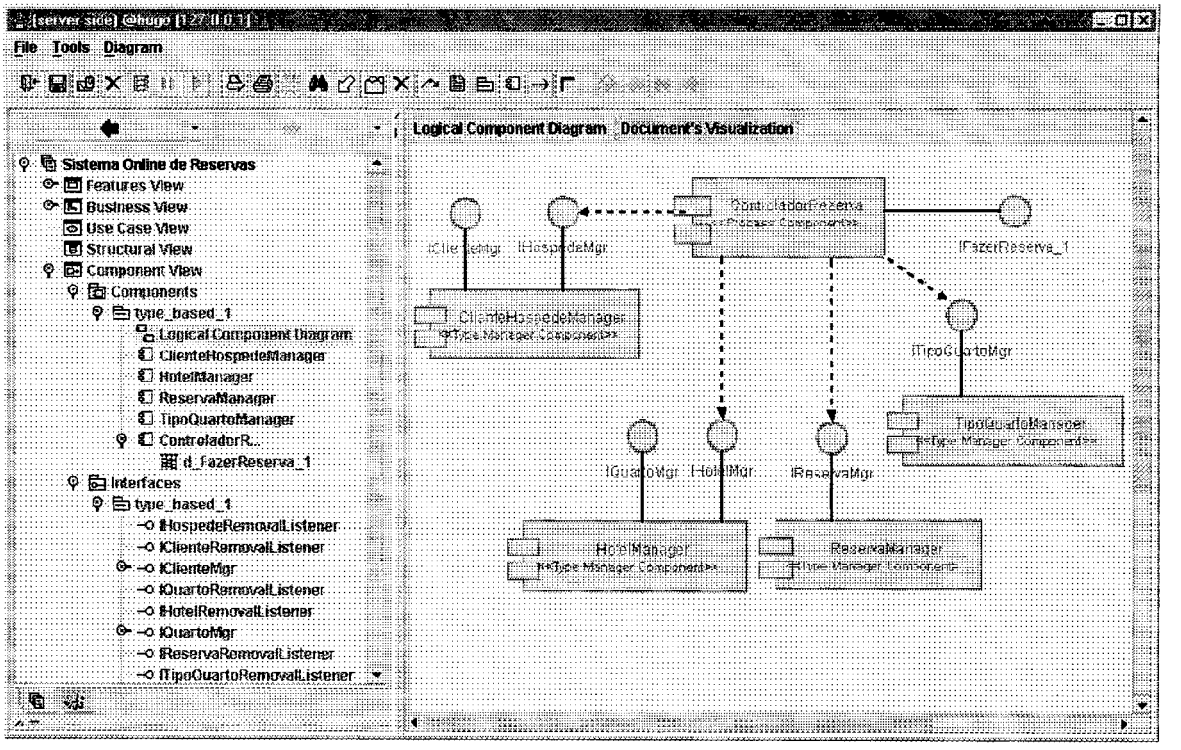


Figura 5.58: Aplicação com os componentes reutilizados do domínio.

5.3 – Detalhamento Técnico da Implementação

A ferramenta apresentada neste capítulo é dividida conforme visto na Figura 5.59. O ambiente *Odyssey* centraliza suas funcionalidades no *Gerente da Arquitetura de Componentes*, o qual organiza e distribui o acesso pelos diferentes serviços da proposta. Os responsáveis pela execução das regras e mecanismos são:

- Gerador de Componentes no Estilo Baseado em Tipos: acessa os tipos de negócio do domínio / aplicação modelado pelo usuário no ambiente *Odyssey* e gera os componentes gerentes de instância segundo o agrupamento realizado pelo arquiteto. Parte.
- Gerador de Componentes de Processo: acessa os passos dos casos de uso que precisam ser tratados pelo componente de processo para gerar as seqüências de chamadas entre as interfaces dos componentes gerentes de instâncias na arquitetura.
- Gerador de Componentes no Estilo Baseado em Instâncias: acessa os tipos de negócio do domínio / aplicação modelado pelo usuário no ambiente *Odyssey* e gera os componentes de entidade e coleção para cada um deles.
- Suporte a Avaliação de Estilos e Tecnologias: responsável pelo cadastro das tecnologias de componentes e suas respectivas avaliações quanto aos

atributos de qualidade nos dois estilos arquiteturais. Sua principal funcionalidade é utilizar estes dados para apoiar e evoluir as decisões do arquiteto durante o desenvolvimento de novas aplicações.

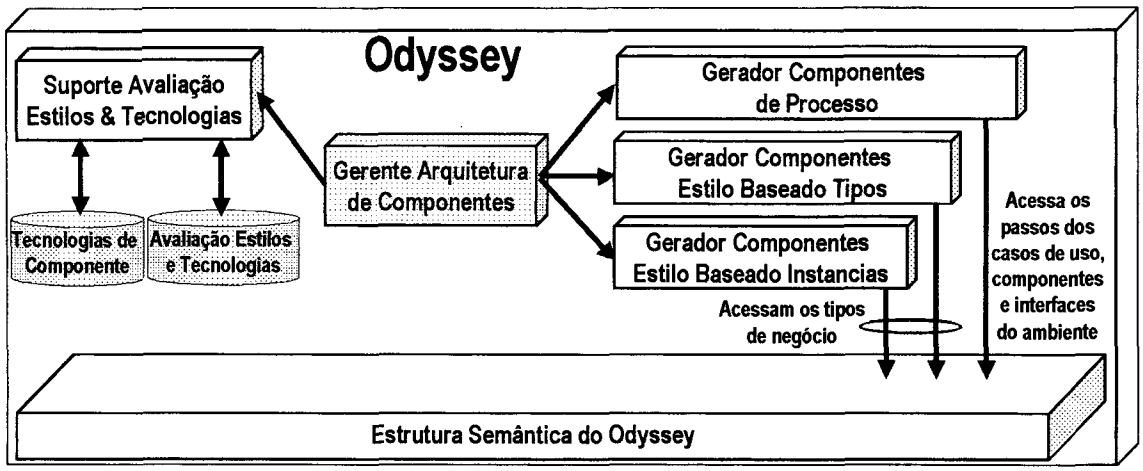


Figura 5.59: Estrutura usada na implementação da proposta.

Devido a complexidade da implementação destas funcionalidades, apresentamos na Figura 5.60 um diagrama de classes simplificado, contendo apenas as classes mais importantes da ferramenta. Idealmente, seria interessante se tivéssemos essas funcionalidades implementadas em componentes que pudessem ser conectados aos ambientes de desenvolvimento de software interessados na proposta. Entretanto, não fizemos dessa forma por exigir um trabalho de reestruturação muito grande do ambiente *Odyssey*, o que se encontra fora do escopo desta pesquisa.

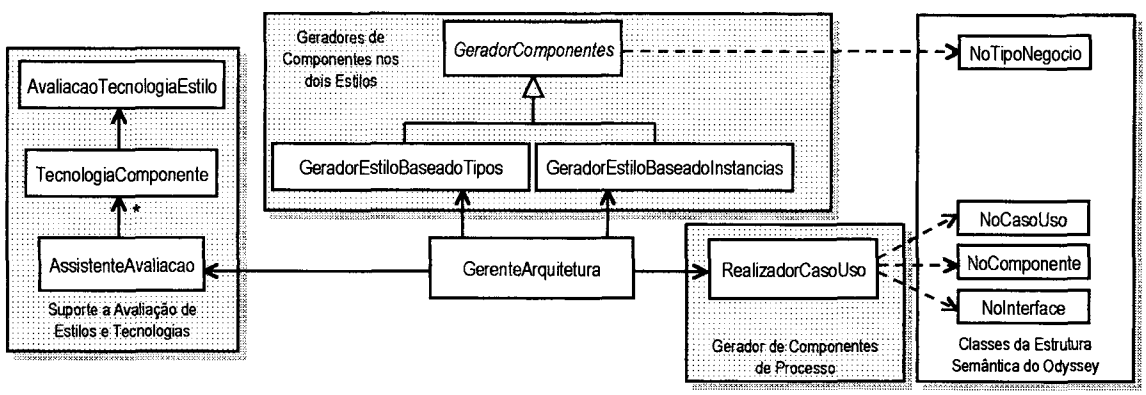


Figura 5.60: Diagrama de classes da implementação.

5.4 – Conclusões

Apresentamos neste capítulo a implementação da proposta de geração e conexão de componentes de negócio que foi realizada no ambiente *Odyssey*, aproveitando sua infra-estrutura de reutilização e desenvolvimento de software. Alguns importantes

aspectos foram discutidos, como o impacto causado pela introdução de tipos de negócio no ambiente e seu comportamento em relação às *features* do método FODA (KANG *et al.*, 1990).

Para exemplificar os mecanismos implementados, utilizamos o mesmo exemplo de hotelaria discutido no capítulo anterior, gerando os componentes nos dois estilos arquiteturais estudados. Também foi apresentado o mecanismo de suporte a decisão que ajuda a equipe de desenvolvimento na escolha dos estilos e tecnologias que podem ser utilizados em uma aplicação.

Considerando os dois níveis de modelagem existentes no ambiente *Odyssey* (domínio e aplicação), discutimos os propósitos da geração para cada um deles segundo o papel que eles assumem no desenvolvimento de software. Além disso, mostramos como uma aplicação pode reutilizar os componentes gerados no domínio seguindo a seleção de artefatos utilizada pelo processo de Engenharia de Aplicação do *Odyssey* (*Odyssey-AE*).

Capítulo 6 – Conclusões e Trabalhos Futuros

6.1 – Visão Geral

Desenvolvemos neste trabalho uma proposta para a geração de componentes de negócio, abordando dois estilos arquiteturais conhecidos da literatura (HERZUM *et al.*, 2000) (CHEESMAN e DANIELS, 2001) (BROWN, 2000) (ATKINSON, 2000): o *estilo baseado em tipos* e o *estilo baseado em instâncias*. Para cada um deles, especificamos as responsabilidades e interfaces de seus componentes e como eles devem ser utilizados pela equipe de desenvolvimento na arquitetura. Além disso, adaptamos um trabalho de pesquisa desenvolvido na COPPE/UFRJ (XAVIER, 2001) para apoiar a escolha entre os estilos e as tecnologias de componente que podem ser usadas na implementação do software, oferecendo ao arquiteto uma forma de avaliar e refinar suas decisões conforme ele avança em seus projetos.

Conforme dito inicialmente, a motivação para este trabalho surgiu das dificuldades e dúvidas encontradas durante a montagem de arquiteturas de componentes na prática, seguindo as atuais abordagens de DBC (e.g., *UML Components*, *Catalysis*, etc.), as quais deixam importantes decisões para o projetista, e não oferecem apoio que possa ser retirado diretamente dos modelos de análise do software. Focando esse problema, desenvolvemos um conjunto de *regras* e *mecanismos* que consegue traduzir os elementos sintáticos presentes nessa modelagem para gerar os componentes de negócios de uma aplicação, reduzindo o nível de complexidade das decisões deixadas para o arquiteto.

A implementação dessas idéias aproveitou o contexto do projeto *Odyssey* (WERNER *et al.*, 2000) desenvolvido na COPPE/UFRJ. Por não possuir, inicialmente, todas as características básicas do DBC, esse ambiente precisou ser modificado para se adequar a esses princípios de desenvolvimento.

6.2 – Contribuições e Benefícios

As principais *contribuições* trazidas por esta pesquisa são:

1. A definição de um conjunto de regras e mecanismos capazes de traduzir elementos sintáticos presentes nos modelos de análise em especificação de componentes independentes de tecnologia. Esta capacidade de geração de

componentes abrange dois estilos arquiteturais para componentes de negócio (*estilo baseado em tipos* e *estilo baseado em instâncias*), definindo suas responsabilidades e elaborando as suas respectivas interfaces na arquitetura. A geração se resume em:

- Estilo baseado em tipos: regras para *agrupar* os tipos de negócio e montar as *estruturas de dados* e *interfaces* dos componentes gerentes de instâncias; um *algoritmo* para especificar os componentes de processo utilizando a descrição dos casos de uso.
- Estilo baseado em instâncias: regras para transformar cada tipo de negócio em uma dupla de componentes: os componentes de entidade e os componentes-coleção.

2. A implementação de uma ferramenta para a geração de componentes de negócio no ambiente *Odyssey* (WERNER *et al.*, 2000), oferecendo suporte a geração de componentes de negócio segundo as regras e mecanismos desenvolvidos.

Dentro desta pesquisa, podemos identificar contribuições mais pontuais que merecem ser destacadas, como:

1. A definição de um algoritmo capaz de especificar, para um determinado componente, as seqüências de chamadas que podem ser efetuadas na arquitetura para se executar um caso de uso no software. O algoritmo utiliza informações de negócio (no formato de *strings*) associadas aos passos do caso de uso para encontrar, nas interfaces dos componentes, operações que as realizam. Quando algum problema é detectado, o algoritmo avisa ao arquiteto para que ele inspecione a sua causa.
2. A especificação de componentes de processo com a capacidade de abstrair, para seus clientes, a dificuldade inerente à manipulação das chaves técnicas no estilo baseado em tipos. Isto permite que os componentes de processo facilitem a execução das atividades de negócio e mantenham esses detalhes internos transparentes para os outros componentes da arquitetura.
3. A identificação de fatores que influenciam o agrupamento dos tipos de negócio em componentes gerentes de instâncias no estilo baseado em tipos. Os fatores identificados e analisados são: busca de componentes em repositórios, desempenho e distribuição da aplicação.

4. A definição de uma técnica que permite apoiar as decisões do arquiteto durante a escolha dos estilos arquiteturais e tecnologias de componentes disponíveis. Esta técnica utiliza uma base de informações preenchida pelo próprio arquiteto, o qual pode evoluí-la e refiná-la conforme ele avança em seus projetos. Esta técnica é uma adaptação de um trabalho anteriormente desenvolvido na COPPE/UFRJ por XAVIER (2001).

Considerando as contribuições acima, alguns *benefícios* podem ser vislumbrados com a utilização da abordagem apresentada. São eles:

- Agilidade e rapidez: a utilização das regras e mecanismos propostos faz com que o trabalho da equipe se torne mais eficiente e menos repetitivo ao longo do desenvolvimento, ajudando a reduzir o tempo gasto com atividades braçais e a se concentrar em outros detalhes mais importantes, como o agrupamento dos tipos de negócio ou o desenvolvimento de outros componentes para o software.
- Independência de abordagem: Os elementos sintáticos utilizados na confecção das regras e mecanismos deste trabalho foram apoiados em princípios comuns das atuais abordagens de DBC, o que permite que eles sejam empregados independente de qual delas foi adotada para o projeto.
- Treinamento: Como o desenvolvimento orientado a objetos (sem componentes) ainda é muito comum atualmente, muitos desenvolvedores precisam de treinamento para conseguir especificar corretamente uma arquitetura de software baseada em componentes (serviços). Embora este trabalho aborde apenas uma parcela deste conhecimento, as idéias apresentadas podem ser utilizadas neste sentido, auxiliando o aprendizado dos integrantes de uma equipe.
- Identificação antecipada de erros de análise: Como os componentes de negócio são gerados a partir dos modelos de análise, qualquer problema presente nesses modelos pode resultar em uma arquitetura deficiente para o projeto. Se a geração for feita no estilo baseado em tipos, o arquiteto pode identificar problemas ocorridos nesta modelagem antes mesmo da implementação dos componentes, o que reduz, significativamente, a quantidade de re-trabalho da equipe. Isso é possível utilizando-se o mecanismo de *geração de componentes de processo* (apresentado na seção 4.2.1.3), o qual é capaz de indicar para o arquiteto onde existem *falhas* ou *ausência de informações* nestes modelos de análise.

- Padronização: Como empregamos os mesmos conjuntos de operações na geração das interfaces dos componentes de negócio, uma padronização ocorre com esses componentes ao longo dos anos. Isso contribui com uma redução do esforço envolvido com a *documentação*, *implementação* e *entendimento* dos componentes.

6.3 – Limitações e Trabalhos Futuros

Algumas limitações e extensões puderam ser detectadas ao longo dessa pesquisa, as quais esperamos tratar em trabalhos futuros. A seguir, explicamos cada tópico separadamente, sugerindo melhorias e analisando possíveis conseqüências para o desenvolvimento de software.

6.3.1 – Preenchimento de Informações Dinâmicas

A geração de componentes de processo (no estilo baseado em tipos) exige que os casos de uso do software contenham bastantes detalhes referentes à manipulação dos tipos de negócio, que acaba tornando o seu preenchimento lento e cansativo. Um esforço poderia ser gasto nesse sentido, investigando formas mais rápidas e adequadas para a captura dessas informações dinâmicas do software. Em princípio, poder-se-ia tentar pesquisar formas de utilizar os próprios diagramas da UML ou elaborar um diagrama específico para essa finalidade.

6.3.2 – Agrupamento Configurável

Dentro do estilo baseado em tipos, o agrupamento de tipos de negócio poderia ser melhorado se fosse possível configurar, pelo ambiente, as regras que são aplicadas aos diagramas de tipos de negócio. Isto poderia ser feito utilizando uma linguagem interpretada (*script*) que teria à sua disposição variáveis e operações de acesso aos elementos sintáticos desse tipo de diagrama. Isto representaria uma boa oportunidade de evoluir a geração de componentes conforme a experiência da equipe aumentasse.

6.3.3 – Interfaces Configuráveis

Uma das limitações contida neste trabalho é a inflexibilidade sobre a geração das interfaces dos componentes de negócio, as quais são montadas utilizando-se um conjunto fixo de operações pré-estabelecidas. Uma possibilidade de melhoria seria a

criação de modelos de interface (*templates*) que permitissem a criação e configuração das suas operações e parâmetros gerados.

Para ser completa, essa funcionalidade também poderia permitir a configuração das ações (*strings*) que explicam o que a operação realiza no contexto de negócio (e.g., se cria/exclui uma instância de um tipo de negócio, se conecta duas instâncias, etc.). Todas essas características deixariam a geração dos componentes mais flexível e capaz de se adaptar melhor aos tipos de sistemas desenvolvidos.

6.3.4 – Suporte a Fabricação de Componentes

Mesmo entre diferentes domínios, a implementação dos componentes gerados com essa proposta não varia muito para cada caso. Um componente que gerencia instâncias de um tipo de negócio possui algoritmos que independem se o tipo pertence ao domínio *A* ou *B*. Da mesma forma, sua documentação, seus modelos internos, seus casos de teste, seus manuais, etc. também seguem um estilo padronizado que pode ser reaproveitado para cada aplicação desenvolvida.

Nesse contexto, se tivéssemos um ferramental que permitisse que esses modelos, códigos e documentos pudessem ser elaborados e reutilizados de forma prática e sem muito esforço, a geração dos componentes alcançaria proporções ideais para o estabelecimento de uma fábrica de componentes. Em termos práticos, isto significa que as partes (componentes) de uma aplicação poderiam ser rapidamente especificadas, implementadas e documentadas, necessitando de um esforço adicional apenas naqueles tipos de componentes que jamais foram desenvolvidos antes.

6.3.5 – Geração de Componentes Complementares

Além dos componentes de negócio gerados com a proposta deste trabalho, outros componentes precisam existir para que a aplicação se torne completa para o seu usuário final. Visando agilizar esse desenvolvimento, precisamos de mais mecanismos que gerem componentes com outras responsabilidades na arquitetura. Os componentes mais adequados para esse objetivo são os componentes de *interface com o usuário* (UI), os quais podem ser janelas, páginas WWW, etc.

A geração desses componentes UI exigirá, com certeza, a presença de informações que não foram levantadas ao longo deste trabalho devido às suas diferentes necessidades. Entretanto, na seção 4.2.1.2 (que descreve a geração de componentes de processo a partir dos casos de uso), observamos que nem todos os passos dos casos de

uso foram preenchidos com ações, já que a maioria deles tratava, justamente, das questões de interação com o usuário.

Nesta linha de raciocínio, poderíamos criar novas ações que fossem específicas para essas interações homem-máquina, servindo de insumo para um mecanismo especialmente preparado para montar protótipos de interface para a aplicação. Dependendo do estilo arquitetural empregado, estes componentes seriam especificados para trabalhar com os componentes gerados com este trabalho, completando e integrando seus papéis na arquitetura.

Além dos componentes UI, outros componentes também podem fazer parte desta geração, como, por exemplo, componentes para *armazenamento* dos dados do software. Embora seja possível utilizar serviços de armazenamento oferecidos pelas tecnologias de componente (como, por exemplo, *serialização* do Java), a geração de componentes de armazenamento específicos para uma aplicação pode ser vantajosa, já que muitas organizações preferem modelar e desenvolver suas próprias bases de dados em seus projetos.

Em uma análise final, é importante salientar a necessidade de aplicação da proposta em projetos reais e mais complexos, onde a complexidade presente nos modelos possa representar um fator decisivo na validação das regras e mecanismos apresentados. Além disso, vantagens adicionais podem ser obtidas, como a identificação de novas regras ou, mesmo, a aquisição de idéias que possam aprimorar, cada vez mais, as técnicas de geração de componentes de software. Nesse sentido, projetos na área de *Agropecuária* e *Educação à Distância* estão sendo desenvolvidos na Universidade Federal de Juiz de Fora (UFJF) e pretendem, entre outros objetivos, aplicar a proposta com essa finalidade.

Capítulo 7 – Referências Bibliográficas

AOYAMA, M., 1998, New age of software development: How component-based software engineering changes the way of software development. *In Proceedings of the First International Workshop on Component-Based Software Engineering*, Kyoto, Japão, Abril.

ATKINSON, C.; BAYER, J.; MUTHIG, D., 2000, Component-based Product Line Development – The KobrA Approach, *In Proceedings of 1st Software Product Line Conference (SPLC1)*, Denver, pp. 289-309, Estados Unidos.

BACHMAN, F; BASS, L.; BUHMAN, C.; COMELLA-DORDA, S.; LONG, F.; ROBERT, J.; SEACORD, R.; WALLNAU, K., 2000, *Volume II: Technical Concepts of Component-Based Software Engineering*, SEI Technical Report CMU/SEI-2000-TR-008.

BASS, L.; CLEMENTS, P.; KAZMAN, R., 1998, Software Architectures in Practice, Addison-Wesley.

BERGNER, K.; RAUSCH, A.; SIHLING, M.; VILBIG, A., 1998, *A Componentware Methodology Based on Process Patterns*, Technical Report TUM-I9823, Instituto de Informática, Universidade Técnica de Munique.

BERTOLINO, A.; POLINI, A., 2002, Re-Thinking the Development Process of Component-based Software, *In Proceedings of the 9th IEEE Conference and Workshop on Engineering of Computer-based Systems*, Lund, Suécia, pp. 7-10, Abril.

BOOCH, G., 1994, *Object-Oriented Analysis and Design with Applications*, Redwood City, CA, Benjamin Cummings, 1994.

BOX, D., 1997, *Essential COM*, Addison-Wesley.

BRAGA, R. M. M., 2000, Busca e Recuperação de Componentes em Ambientes de Reutilização de Software. Tese de D.Sc. COPPE/UFRJ. Rio de Janeiro, Dezembro.

BREDEMEYER, Bredemeyer Consulting, Software Architecting, página da Internet http://www.bredemeyer.com/howto.htm (acesso em 01/06/2002).

BROWN, A., 2000, *Large-Scale Component-Based Development*, Prentice Hall PTR.

BUSCHMANN, F.; MEUNIER, R.; ROHNERTH.; SOMMERLAD P.; STAL, M., 1996, A System of Patterns – Pattern-Oriented Software Architecture, John Wiley & Sons.

CHEESMAN, J., DANIELS, J., 2001, *UML Components – A Simple Process for*

Specifying Component-Based Software, Addison-Wesley.

CHEN, P. P., 1976, The Entity-Relationship Model: Toward the Unified View of Data, *ACM Transactions of Database Systems (TODS)*, Volume 1, Número 1, pp. 9-36, Março.

COLEMAN, D.; ARNOLD P.; BODOFF, S.; DOLLIN, C.; GILCHRIST, H.; HAYES, F.; JEREMES, P., 1994, *Object-Oriented Development: The Fusion Method*, Prentice-Hall.

CRNKOVIC, I.; LARSSON, M., 2001, Challenges of Component-based Development, *Journal of Software Systems*, Dezembro.

D'SOUZA, D., WILLIS, A. C., 1999, *Objects, Components, And Frameworks With UML: The Catalysis Approach*, 1ed., Massachusetts, Addison Wesley.

DANTAS, A. R., 2001, Oráculo: Um Sistema de Críticas para a UML. Projeto Final de Curso, DCC/IM/UFRJ, Rio de Janeiro.

DANTAS, A. R.; VERONESE, G.; CORREA, A.; XAVIER, J. R.; WERNER, C., 2002, Suporte a Padrões no Projeto de Software, Caderno de Ferramentas do XVI Simpósio Brasileiro de Engenharia de Software (SBES'2002), Gramado, Rio Grande do Sul, Outubro.

ERIKSSON, H.; PENKER, M., 1998, *UML Toolkit*, John Wiley & Sons.

ESKELIN, P., 1999, Component Interaction Patterns, *in Proceedings of Pattern Languages of Programs (PloP'1999)*, Monticello, Estados Unidos, Agosto.

FAYAD, M. E.; SCHMIDT D. C., 1997, *Object-Oriented Application Frameworks*, CACM, vol. 40, N° 10, pp. 32-38, Outubro.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J., 1995, *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley.

GRISS, M. L., 1997, Software Reuse: Architecture, Process, and Organization for Business Success, *Proceedings of the Eighth Israeli Conference on Computer Systems and Software Engineering*, Israel, pp. 86-98, Junho.

HALL, P., 2000, Educational Case Study – What is the model of an ideal component? Must it be an object?, *International Workshop on Component-Based Software Engineering*, Limerick, Irlanda, Junho.

HALLSTEINSEN, S.; SKYLSTAD, G., 1999, The Magma Approach to CBSE, *International Workshop on Component-Based Software Engineering*, Los Angeles, Estados Unidos, pp.181-186, Maio.

HAN, J., 1998, A Comprehensive Interface Definition Framework for Software

Components, *Asia-Pacific Software Engineering Conference*, Taipei, Taiwan, Dezembro.

HERZUM, P.; SIMS, O., 2000, *Business Component Factory: A Comprehensive Overview of Component-Based Development for the Enterprise*, OMG Press.

HÖRNSTEIN, J.; EDLER, H., 2002, Test Reuse in CBSE Using Built-In Tests, *In Proceedings of the 9th IEEE Conference and Workshop on Engineering of Computer-based Systems*, Lund, Suécia, pp. 11-14, Abril.

HUBER, F.; RAUSCH, A.; RUMPE, B., 1998, Modeling Dynamic Component Interfaces, *Proceeding of the 26th International Conference on Technology of Object-Oriented Languages and Systems*, Santa Barbara, Estados Unidos, pp. 58-70, Agosto.

HUGHES, E., 1999, From Frameworks to Plug and Play Components, apresentação disponível em http://199.94.100.1/technology/domis/briefs/cmpnts-tutorial/index.htm (acesso em 01/05/2002).

JACOBSON, I.; CHRISTERSON, M.; JOHNSSON, P.; ÖVERGAARD, G, 1992, *Object-Oriented Software Engineering*, Addison-Wesley.

JACOBSON, I.; GRISS, M.; JONSSON, P., 1997, Software Reuse, Addison-Wesley.

JAVABEANS, 1997, *JavaBeans API Specification – version 1.01*, Sun Microsystems, Mountain View, CA, Estados Unidos.

KANG, K., COHEN, S., HESS, J. *et al.*, 1990, *Feature-Oriented Domain Analysis (FODA) - Feasibility Study*, SEI Technical Report CMU/SEI-90-TR-21.

KIRTLAND, M., 1997, Object-Oriented Software Development Made Simple with COM+ Runtime Services, *Microsoft Systems Journal*, Novembro.

KRUCHTEN, P., 2000, *Rational Unified Process: An Introduction*, 2nd edition., Addison-Wesley.

LARMAN, C., 2001, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and The Unified Process*, 2nd edition, Prentice-Hall.

MCINNIS, K., 1999, Component Object Identity and Persistence, artigo disponível em http://www.cbd-hq.com (acesso em 01/05/2002).

MCINNIS, K., 2000, Type Modeling for Portable Components, artigo disponível em http://www.cbd-hq.com (acesso em 01/05/2002).

MEYER, B., 1997, *Object-Oriented Software Construction*, Second Edition, Prentice-Hall.

MILLER, N., 2000, A Engenharia de Aplicações no Contexto da Reutilização Baseada

em Modelos de Domínio. Tese de M.Sc., COPPE/UFRJ, Rio de Janeiro, Brasil.

MODELMAKER, 2003, Ferramenta *ModelMaker* for Borland Delphi 7, ModelMaker Tools, disponível em www.modelmakertools.com.br (acesso em 10/01/2003).

MONSON-HAEFEL, R., 1999, *Enterprise JavaBeans*, O'Reilly Press.

MOWBRAY, T. J.; RUH, W. A., 1997, *Inside Corba: Distributed Object Standards and Applications*, Addison-Wesley.

MURTA, L. G. P., 1999, FRAMEDOC: Um FrameWork para a Documentação de componentes Reutilizáveis, Projeto Final de Curso, DCC/IM/UFRJ, Novembro.

MURTA, L. G. P., 2002, Charon: Uma Máquina de Processos Extensível Baseada em Agentes Inteligentes, Tese de M.Sc., COPPE / UFRJ, Rio de Janeiro, Brasil, Março.

NAVATHE, S. B.; ELMASRI, R. A., 1999, *Fundamental of Database Systems*, Addison-Wesley Publishing, 3rd Edition.

NING, J. Q., 1999, A Component Model Proposal, *International Workshop on Component-Based Software Engineering*, Los Angeles, Estados Unidos, pp. 13-16, Maio.

PAGE-JONES, M., 1999, *Fundamentals of Object-Oriented Design in UML*, Addison-Wesley Pub Co.

PARRISH, A.; DIXON, C.; HALE, D., 1999, Component-Based Software Engineering: A Broad-Based Model is Needed, *International Workshop on Component-Based Software Engineering*, Los Angeles, Estados Unidos, pp. 43-46, Maio.

PRESSMAN, R. S., 2000, *Software Engineering – A Practitioners Approach*, 5th ed., McGraw Hill.

RATIONAL, 2003, Ferramenta Rational Rose, disponível em http://www.rational.com/products/rose (acesso em 10/01/2003).

RUMBAUGH, J.; BLAHA, M.; PREMERLANI, W.; EDDY, F.; LORENSEN, F., 1991, *Object-Oriented Modeling and Design*, Prentice-Hall.

SAMETINGER, J., 1997, *Software Engineering with Reusable Components*, Springer-Verlag.

SHAW, M.; GARLAN, D., 1996, Software Architecture: Perspectives on an Emerging Discipline, Prentice-Hall.

SIMOS, M.; ANTHONY, J., 1998, Weaving the Model Web: A Multi-Modeling Approach to Concepts and Features in Domains Engineering, *In Proceedings of the 5th International Conference on Software Reuse (ICSR-5)*, ACM/IEEE, pp. 94-102,

Victoria, Canadá, Junho.

SPARLING, M., 1999, Business Component Architectures, artigo disponível em http://www.cbd-hq.com (acesso em 01/05/2002).

SZYPERSKI, C., 1998, *Component Software: Beyond Object-Oriented Programming*, ACM Press / Addison Wesley Longman.

VERONESE, G. O., NETTO F. J., 2001, Uma Ferramenta de Apoio a Recuperação de Projetos no Ambiente Odyssey. Projeto Final de Curso, DCC/IM/UFRJ, Abril.

VERYARD, R., 2001, *The Component Based Business: Plug and Play*, Springer-Verlag.

WALLNAU, K. C.; HISSAM, S. A.; SEACORD, R. C., 2001a, *Building Systems from Commercial Components*, SEI Series in Software Engineering, Addison-Wesley.

WALLNAU, K.; STAFFORD, J.; HISSAM, S.; KLEIN, M., 2001b, On the Relationship of Software Architecture to Software Component Technology, *Proceedings of the 6th International Workshop on Component-Oriented Programming (WCOP6), in conjunction with the European Conference on Object-Oriented Programming (ECOOP)*, Budapest, Hungary, Junho.

WECK, W., 1997, Independently Extensible Component Frameworks. Special Issues in Object-Oriented Programming, *Workshop Reader of the 10th European Conference on Object-Oriented Programming ECOOP'96*, Linz, July, M. Mühlhäuser, ed., dpunkt.verlag, Heidelberg, pp. 177-183.

WERNER, C.; BRAGA, R.; MATTOSO, M.; *et alli.*, 2000, Odyssey: Estágio Atual, Caderno de Ferramentas do XV Simpósio Brasileiro de Engenharia de Software (SBES'2000), João Pessoa, Paraíba, pp. 366-369, Outubro.

WIENBERG, A.; MATTHES, F.; BOGER, M., 1999, Modeling Dynamic Software Components in UML, *In Proceedings of the Second International Conference on the Unified Modeling Language (UML'99)*. LNCS 1723, pages 204-219, Springer-Verlag.

WILE, D. S., 2001, Ensuring General-Purpose and Domain Specific Properties Using Architectural Styles, *Proceedings of the 4th International Congress on Software Engineering (Workshop on Component-based Software Engineering)*, Toronto, Canada, Maio.

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

XIA, C.; LYU, M. R.; WONG, K., 2000, Component-Based Software Engineering: Technologies, Quality Assurance Schemes, and Risk Analysis Tools, *Proceedings of the*

7th Asia-Pacific Software Engineering Conference (APSEC'2000), Singapura, Dezembro.

YACOUB, S.; AMMAR, H., MILI, A., 1999a, A Model for Classifying Component Interfaces, *International Workshop on Component-Based Software Engineering*, Los Angeles, Estados Unidos, pp.133-138, Maio.

YACOUB, S.; AMMAR, H., MILI, A., 1999b, Characterizing a Software Component, *International Workshop on Component-Based Software Engineering*, Los Angeles, Estados Unidos, pp. 151-157, Maio.

ZOPELARI, M., 1998, Uma Proposta de Sistemática para Aquisição de Conhecimento no Contexto de Análise de Domínio, Tese de M.Sc., COPPE/UFRJ, Rio de Janeiro, Brasil, Novembro.