FACULDADE DE INFORMÁTICA
Programa de Pós-Graduação em Ciência da Computação
PUCRS – Brasil

http://www.pucrs.br/inf

UM ESTUDO SOBRE O DESENVOLVIMENTO BASEADO EM COMPONENTES

LUCIANA DE ARAUJO SPAGNOLI

KARIN BECKER

TECHNICAL REPORT SERIES

NUMBER 026

MAY, 2003

Contato:

lspagnoli@inf.pucrs.br
http://www.inf.pucrs.br/~lspagnoli

kbecker@inf.pucrs.br
http://www.inf.pucrs.br/~kbecker

Luciana de Araujo Spagnoli é aluna de mestrado do PPGCC-PUCRS, trabalhando ativamente com Desenvolvimento Baseado em Componentes, preocupando-se com as atividades de adaptação e integração de componentes. Também atua como pesquisadora no centro de pesquisa do Convênio DELL/PUCRS.

Karin Becker concluiu seu doutorado em 1993, junto às Facultés Universitaires Notre-Dame de la Paix (Bélgica). Atualmente é professora titular junto à Faculdade de Informática da Pontifícia Universidade Católica do Rio Grande do Sul (FACIN-PUCRS), credenciada para atuação junto ao Programa de Pós-Graduação em Ciência de Computação daquela instituição (PPGCC-PUCRS). Coordena projetos de pesquisa com apoio CNPq, RHAE e FAPERGS nas áreas de reuso, orientação a objetos, *frameworks* e componentes.

Published by Programa de Pós-Graduação em Ciência da Computação - PPGCC/PUCRS

Av. Ipiranga, 6681

90619-900 - Porto Alegre, RS – Brasil

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

API	*Application Programming Interface*
CCM	*CORBA Component Model*
COM	*Component Object Model*
CORBA	*Common Object Request Broker Architecture*
DBC	Desenvolvimento Baseado em Componentes
DCOM	*Distributed Component Object Model*
EJB	*Enterprise JavaBeans*
IIOP	*Internet Inter-ORB Protocol*
J2EE	*Java 2 Platform, Enterprise Edition*
J2SE	*Java 2 Platform, Standard Edition*
JAR	*Java Archive*
JB	*JavaBeans*
JMS	*Java Message Service*
JNDI	*Java Naming and Directory Interface*
OMG	*Object Management Group*
RMI	*Remote Method Invocation*
UML	*Unified Modeling Language*
XML	*Extensible Markup Language*

RESUMO

O Desenvolvimento Baseado em Componentes (DBC) se caracteriza como uma abordagem de desenvolvimento estabelecida pela integração planejada de componentes de software já existentes. Esta abordagem enfatiza o reuso e apresenta vantagens no sentido de possibilitar o aumento da produtividade e qualidade. DBC facilita a manutenção dos sistemas por possibilitar que estes sejam atualizados pela integração de novos componentes e/ou substituição dos componentes existentes. Apesar de todas as potenciais vantagens desta abordagem têm-se também inúmeras questões em aberto e definições ainda sem consenso. Componentes, modelo de componentes e *framework* de componentes são exemplos de termos básicos que freqüentemente são alvo de confusão.

Neste contexto, este relatório procura identificar e discutir algum dos principais termos e conceitos relacionados a DBC. Inicialmente são identificadas e discutidas possíveis definições de componentes, bem como se procura caracterizar o que são modelos e *frameworks* de componentes, e discutir o relacionamento de componentes com a orientação a objetos. Em um segundo momento são discutidas as perspectivas de desenvolvimento relacionadas a DBC (desenvolvimento de componentes e desenvolvimento com componentes), enfatizado o desenvolvimento com componentes e as suas duas principais atividades, adaptação e integração de componentes. Finaliza-se este relatório com a apresentação da tecnologia *Enterprise JavaBeans* e a caracterização desta tecnologia frente ao referencial teórico abordado.

Palavras-chave: componentes, DBC, modelo de componentes, *framework* de componentes, adaptação, integração, *Enterprise JavaBeans*.

ABSTRACT

Component-Based Development (CBD) is a software development approach characterized by the integration of existing components. CBD emphasizes software reuse and its benefits are related to productivity and quality increase. CBD also helps the maintenance of systems by the integration of new components and/or replacement of the existing components. Despite of all advantages, there are many open issues and lack of consensus in CBD. Components, component model, component framework are examples of basic terms for which conflicting definitions exist.

In this sense, this report aim at identifying and discussing some of the main concepts and approaches related to CBD. Initially, some definitions for components are presented and discussed. Component models, frameworks as well as the relationship between components and object-orientation are also addressed in the report. CBD software development approaches are discussed next, focusing on the development with components, and theirs two main activities, namely component adaptation and integration, are detailed. Finally, Enterprise JavaBeans is briefly discussed, and relationship of this technology with CBD concepts is established.

Keywords: components, CBD, component model, component framework, adaptation, integration, Enterprise JavaBeans.

1 INTRODUÇÃO

A preocupação e os esforços empregados para melhorar as práticas de desenvolvimento de software buscando o aumento da produtividade e da qualidade, bem como a redução de custos e esforços, evidenciam novas perspectivas para o desenvolvimento de software.

O reuso é um importante aspecto a ser considerado, pois potencialmente apresenta um positivo impacto na qualidade, produtividade e custo do software. Reconhecer o fato de que muitos sistemas apresentam componentes similares ou até mesmo idênticos, os quais são repetidamente desenvolvidos do zero conduz à tentativa de reusar componentes já existentes [SAM97].

Recentemente tem havido um novo interesse na noção de desenvolvimento de software através da integração planejada de componentes de software já existentes [BRO98]. Componentes e reuso completam-se perfeitamente, a utilização de componentes para construir sistemas de software quase que automaticamente conduz ao reuso, embora o uso de componentes por si só não seja suficiente para garantir o reuso de software [SAM97].

O Desenvolvimento Baseado em Componentes (DBC) surge como uma nova perspectiva de desenvolvimento de software caracterizada pela composição de partes já existentes. Conforme [SZY99], construir novas soluções pela combinação de componentes desenvolvidos e comprados aumenta a qualidade e dá suporte ao rápido desenvolvimento, levando à diminuição do tempo de entrega do produto ao mercado. Os sistemas definidos através da composição de componentes permitem que sejam adicionadas, removidas e substituídas partes do sistema sem a necessidade de sua completa substituição. Com isso, DBC auxilia na manutenção dos sistemas de software, por permitir que os sistemas sejam atualizados através da integração de novos componentes e/ou atualização dos componentes já existentes.

As razões para o atual interesse por DBC, segundo [BRO98], advêm da maturidade das tecnologias que permitem a construção de componentes e a combinação destas para o desenvolvimento de aplicações, bem como o atual contexto organizacional e de negócio que apresenta mudanças quanto a como as aplicações são desenvolvidas, utilizadas e mantidas.

Sabendo da importância do reuso no desenvolvimento de software e dos potenciais benefícios de DBC, este trabalho está centrado em uma revisão bibliográfica que tem por objetivo apresentar um estudo sobre a definição de componentes e de aspectos relacionados, as etapas do desenvolvimento de software com componentes e de uma das tecnologias de componentes. Espera-se com este trabalho contribuir para o entendimento de DBC e suas atividades relacionadas, bem como possibilitar a verificação dos conceitos observados na teoria em uma das tecnologias de componentes.

Este trabalho está organizado em cinco capítulos. O Capítulo 2 apresenta diferentes conceitos e características de componentes, bem como trata da definição de modelo e *framework* de componentes, e do relacionamento dos componentes com orientação a objetos. No Capítulo 3 são apresentadas as perspectivas de desenvolvimento de software em DBC, sendo enfatizada a abordagem de desenvolvimento *com* componentes e detalhamento das atividades de adaptação e composição de componentes. O Capítulo 4 apresenta e caracteriza a tecnologia *Enterprise JavaBeans* no contexto deste trabalho. Finalizando, no Capítulo 5 são apresentadas as considerações finais.

2 COMPONENTES: DEFINIÇÕES E CONCEITOS RELACIONADOS

A recente atenção e interesse que recaem sobre DBC têm proporcionado a difusão de seus benefícios e características. Entretanto, uma grande dificuldade está em conceituar um componente. Neste capítulo é discutida a conceituação de componente, a caracterização de modelo e *framework* de componente e o relacionamento existente entre componentes e orientação a objetos.

2.1 Componentes

Inúmeras definições de componentes e componentes reusáveis são encontradas na literatura. Em [SAM97] é apresentado que componentes podem ser vistos como alguma parte do sistema de software que é identificável e reusável, ou como o estado seguinte de abstração depois de funções, módulos e classes.

A definição apresentada por [SAM97] considera que "componentes de software reusáveis são artefatos autocontidos, facilmente identificáveis que descrevem e/ou executam funções especificas e têm interfaces claras, documentação apropriada e uma condição de reuso definida".

Esta definição abre possibilidade para se ter uma variedade de tipos de componentes (funções, classes, aplicações, subsistemas, entre outros), pois existe a preocupação do autor em que sua definição continue válida mesmo que rupturas tecnológicas levem ao surgimento de novos componentes.

Um melhor entendimento da definição apresentado por [SAM97] pode ser alcançado através de uma discussão mais aprofundada dos termos indicados por sua definição:

- *Autocontido:* característica dos componentes de poderem ser reusáveis sem a necessidade de incluir/depender de outros componentes. Caso exista alguma dependência, então todo o conjunto deve ser visto como o componente reutilizável;

- *Identificação:* componentes devem ser facilmente identificados, ou seja, devem estar contidos em um único lugar ao invés de espalhados e misturados com outros artefatos de software ou documentação. Os componentes são tratados como artefatos por

poderem assumir uma variedade de diferentes formas como, por exemplo, código fonte, documentação e código executável;

- *Funcionalidade:* componentes têm uma funcionalidade clara e específica que realizam e/ou descrevem. Componentes podem realizar funções ou podem ser simplesmente descrições de funcionalidades. Com isso se deseja também caracterizar como componente toda a documentação do ciclo de vida do software, embora esta não inclua a codificação propriamente dita de funcionalidades;

- *Interface:* componentes devem ter uma interface clara, que indica como podem ser reusados e conectados a outros componentes, e devem ocultar os detalhes que não são necessários para o reuso;

- *Documentação:* a existência de documentação é indispensável para o reuso. O tipo de componente e sua complexidade irão indicar a conveniência do tipo de documentação;

- *Condição de reuso:* componentes devem ser mantidos de modo a preservar o reuso sistemático e a condição de reuso compreende diferentes informações como, por exemplo, quem é o proprietário, quem deve ser contatado em caso de problema, qual é a situação de qualidade, entre outras.

Ainda buscando melhor classificar componente, [SAM97] apresenta um conjunto de atributos para componentes:

- *Funcionalidade:* atributo essencial de um componente para que este seja reusado, pois indicará a sua aplicabilidade a um determinado contexto;

- *Interatividade:* considera as diferentes formas de interação entre componentes e a forma como estes se comportam frente a estas interações;

- *Interação:* indica como acontece a interação entre componentes, e entre componentes e seus usuários;

- *Concorrência:* preocupa-se com a concorrência entre componentes e com os aspectos relacionados para que isso se verifique;

- *Distribuição:* atributo relacionado aos fatores necessários para a distribuição de componentes permitindo a manutenção de sua comunicação e a troca de dados;

- *Formas de adaptação:* atributo que indica a preocupação em adaptar o componente ao longo do tempo de forma que ele permaneça reutilizável;

- *Controle de qualidade:* atributo que indica a preocupação em avaliar e garantir a qualidade do componente.

A definição apresentada por [SAM97] indica que componente pode ser qualquer parte do sistema de software que possa ser identificado e reusado. Existe em [SAM97] a preocupação com o reuso em todas as etapas do processo de desenvolvimento de software, por isso sua definição é intencionalmente genérica quanto ao que é um componente, de forma que leve ao maior grau de reuso possível. O reuso sistemático deve ter início independente da tecnologia disponível e, à medida que novas tecnologias forem sendo disponibilizadas e atingirem um certo grau de maturidade, o reuso deve ser estendido de modo a abranger novos tipos de componente [SAM97].

Em contrapartida à definição de [SAM97], as duas definições apresentadas a seguir não são tão genéricas quanto ao que é componente, principalmente por considerarem componente como o estado seguinte de abstração depois de funções, módulos e classes.

A segunda definição a ser considerada é a apresentada por [BRO97], na qual componente é caracterizado como *"conjunto independente de serviços reutilizáveis"*. Esta definição apresenta dois elementos principais, *"independente"* e *"serviços reusáveis"*, que podem ser considerados independentemente.

O primeiro elemento é o conceito de *"serviços reusáveis"*, que implica que o componente provê habilidades que muitos outros componentes podem acessar. Tendo o conhecimento do que o componente faz, qualquer potencial usuário do componente pode se preocupar com o papel do componente na solução completa da aplicação e não em como os serviços providos pelo componente são realizados. Essa característica indica a necessidade da existência de uma especificação que apresente o que o componente faz e como se comporta quando os serviços são usados. Os serviços indicados pela especificação devem ser providos por algum desenvolvedor que implementa o componente, expressando-o em termos de código e dados, buscando atentamente satisfazer a especificação. Em geral, a implementação pode ser realizada em diferentes linguagens de programação ou plataformas. Um componente pode ser substituído por outro em uma aplicação, desde que ambos atendam à mesma especificação.

O segundo elemento da definição de [BRO97] é *"independente"*, o qual indica a ausência de vínculo do componente com o contexto em que ele pode ser usado. A expectativa de que os componentes cooperem entre si para completar uma solução não deve estar vinculada à

existência de dependências entre eles, pois os componentes não devem ser desenvolvidos com dependências fixas entre si.

A definição de [BRO97], por si só, não explicita diferentes características de componente. Entretanto, estas características são identificadas e apresentadas quando do detalhamento dos dois elementos da definição. Embora a definição de [BRO97] mantenha a preocupação com reutilização, o escopo de abrangência do que é um componente fica restrito a um único tipo de artefato que é o de componentes de código.

A terceira definição de componente a ser considerada é a apresentada por [SZY99], pela qual "componente de software é uma unidade de composição com interfaces contratualmente especificadas e apenas explícitas dependências de contexto. Componente de software pode ser usado independente e combinado com outras partes".

Esta definição busca compreender as propriedades dos componentes de serem unidades com atuação independente, de poderem ser combinados a outras partes e de não apresentarem persistência de estado. Entretanto, cada uma destas propriedades apresenta inúmeras implicações.

A propriedade de ser uma unidade de atuação independente exige que o componente seja bem separado do ambiente e dos demais componentes. Também não deve existir a perspectiva de se ter acesso a detalhes de construção do componente. Logo, o componente precisa encapsular sua implementação e interagir com o ambiente através de interfaces bem definidas. Para que o componente possa ser combinado com outros componentes ele precisa ser suficientemente autocontido. É necessário que o componente apresente uma especificação indicando o que é exigido para seu reuso e o que ele provê. Finalmente, de modo que o componente não apresente nenhuma persistência de estado é necessário que ele não possa ser diferenciado de suas cópias. Ou seja, os componentes não devem manter dados que são específicos da aplicação em que são utilizados, de modo que cópias de um mesmo componente sejam idênticas.

Ainda buscando melhor elucidar a conceituação de [SZY99], têm-se que a necessidade dos componentes de especificarem suas dependências de contexto diz respeito à necessidade deles de indicarem o que o ambiente em que serão acionados deve prover para que eles funcionem.

A definição apresentada por [SZY99] expressa explicitamente importantes propriedades que caracterizam os componentes, o que não se verifica na definição de [BRO97], por estas características estarem implícitas aos elementos que a compõem. Entretanto, igualmente à

definição de [BRO97], a definição de [SZY99] se mantém mais restrita ao tipo de componente, considerando apenas os componentes de código, quando comparada à definição apresentada por [SAM97].

As três definições apresentadas ([SAM97][BRO97][SZY99]), em conjunto com suas respectivas explicações, abordam de diferentes formas aspectos e características comuns dos componentes. A Tabela1 apresenta um quadro comparativo das características de componente a partir das definições e considerações de cada um dos autores.

Tabela 1 - Características dos componentes

	[SAM97]	**[BRO97]**	**[SZY99]**
Identificação	artefato identificável	conjunto independente	unidade de composição
Funcionalidade	funções específicas	serviços reusáveis	serviços providos
Independência	autocontido	independente	autocontido
Interação	interface	serviços	interface
Tipo de artefato	qualquer tipo	componente de código (fonte ou executável)	componente de código (fonte ou executável)
Documentação com a qual se preocupa	apropriada para o reuso	especificação dos serviços	especificação das interfaces
Preocupação com reuso	condição de reuso	×	×

A aplicação e uso de tecnologias de componentes para o desenvolvimento de software está diretamente relacionado a visão de componentes como componentes de código. Nesta visão é desconsiderada a possibilidade de se ter componentes como artefatos presentes em todas as fases do ciclo de desenvolvimento, e os componentes são considerados apenas artefatos da fase de implementação. Por ser a visão que recebe maior ênfase atualmente ela é considerada, neste trabalho, como sendo o aspecto mais maduro de DBC.

Neste sentido, as definições de [BRO97] e [SZY99] ressaltam este aspecto mais maduro de DBC, onde se preocupam exclusivamente com componentes como sendo a abstração seguinte a funções, módulos e classes. Em contrapartida, a definição de [SAM97] trata componentes não apenas como componentes de códigos, mas também como todo possível tipo de componente independente de forma. A conceituação de [SAM97] ressalta melhor uma situação de DBC quando o foco não estiver apenas no reuso de componentes de código, e sim de componentes em todas as fases do processo de desenvolvimento de software.

2.2 Modelo e *Framework* de componentes

Existe em DBC um relativo consenso quanto à impossibilidade de separar componentes e arquitetura de software. Segundo [WER00], um componente não pode ser visto de forma completamente independente dos outros componentes com os quais se relaciona e de seu ambiente. Desta forma, a arquitetura de software assume um importante papel por ser a partir dela que é possível especificar de forma mais detalhada como se dá a interconexão entre os componentes.

Conforme [BAC02], os componentes podem ser vistos segundo duas distintas perspectivas: *implementações* e *abstrações arquiteturais*. Vistos como *implementações*, os componentes representam artefatos de software que podem ser disponibilizados e usados para compor grandes sistemas/subsistemas. Por outro lado, vistos como *abstrações arquiteturais*, os componentes expressam regras de projeto que impõem um modelo padrão de coordenação para todos os componentes. Estas regras de projeto têm a forma de um *modelo de componente*, ou de um conjunto de padrões e convenções com as quais os componentes devem estar em conformidade.

Existe na literatura uma certa confusão relacionada aos termos *modelo de componentes* e *framework de componentes*. Conforme [BAC02], existem diferenças quanto a como estas categorias são nomeadas e não quanto às categorias propriamente ditas. Existe um relativo consenso na literatura quanto ao uso do nome modelo de componentes para identificar o conjunto de padrões e convenções com as quais os componentes devem estar em conformidade. Entretanto, a infra-estrutura que dá suporte a estes modelos é encontrada na literatura tanto identificada como *framework* de componentes [BAC02] como infra-estrutura de componentes [BRO00]

Segundo [BAC02], um *modelo de componente* representa um elemento da arquitetura do sistema na qual são definidos os padrões e convenções impostas aos componentes do sistema, de modo a descrever a função de cada um e como eles interagem entre si. Com isso busca-se expressar restrições de projeto arquitetural ou global.

Já *framework de componente*, conforme [BAC02], é a implementação de serviços que dão suporte ou reforçam o modelo de componentes. A função do *framework* é gerenciar os recursos compartilhados pelos componentes e prover um mecanismo que possibilite a comunicação (interação) entre eles. A infra-estrutura oferecida pelos *frameworks* de componentes impõem restrições e regras no projeto e implementação, as quais devem ser consideradas pelo modelo de componentes.

2.2.1 Modelo de componentes

Segundo [BAC02], não existe ainda um consenso sobre o que deve ou pode estar incluído em um modelo de componentes, mas se espera através dele definir os seguintes padrões e convenções:

- *Tipos de componentes:* definidos em termos das interfaces que implementam, onde cada interface em um componente corresponde a um tipo. Caso um componente implemente as interfaces A e B, então ele é do tipo A e B, o que lhe garante uma capacidade polimórfica em relação a estes tipos. Isto permite que estes diferentes tipos de componentes desempenhem diferentes papéis no sistema, bem como participem de diferentes formas de interação.

- *Formas de interação:* definição da forma de interação entre componentes e entre componentes e o *framework* de componentes, através da especificação de como os componentes são localizados, o protocolo de comunicação usado e como a qualidade dos serviços é alcançada. A classe de interação entre componentes compreende restrições quanto aos tipos de componentes que podem ser clientes de outros tipos, o número de possíveis clientes simultâneos e outras restrições topológicas. A classe de interação entre componentes e *frameworks* inclui definições relacionadas a gerenciamento de recursos, como o ciclo de vida de um componente (ativação, desativação), formas de gerência, persistência e assim por diante. As formas de interação podem dizer respeito a todos os tipos de componentes ou apenas a tipos particulares.

- *Definição de recursos:* a composição dos componentes é realizada pela ligação dos componentes a um ou mais recursos, onde um recurso pode ser tanto provido por um *framework* de componentes como por algum componente utilizado no *framework*. O modelo de componentes descreve quais recursos estão disponíveis a cada componente e como e quando eles são associados a estes recursos. Em contrapartida, o *framework* vê os componentes como recursos a serem gerenciados.

Complementar a definição de [BAC02], a Tabela 2 apresenta sucintamente o conjunto de convenções definidas pelo modelo de componentes estabelecidas em [HEI01].

Tabela 2 - Elementos definidos pelo modelo de componentes

Item	Descrição
Interfaces	Especificação do comportamento e propriedades
Nomeação	Nomes globais únicos para as interfaces e componentes

Metadados	Informações sobre os componentes, interfaces e seu relacionamento
Interoperabilidade	Comunicação e troca de dados entre componentes de diferentes origens, implementados em diferentes linguagens
Customização	Interfaces que possibilitam a customização dos componentes
Composição	Interfaces e regras para combinar componentes no desenvolvimento de aplicações e para substituir e adicionar componentes as aplicações já existentes
Suporte a evolução	Regras e serviços para substituir componentes ou interfaces por versões mais novas
Empacotamento e utilização	Empacotar implementações e recursos necessários para instalar e configurar componentes

2.2.2 *Framework* de componentes

O *framework* de componente representa a base sobre a qual estes padrões e convenções do modelo de componentes são empregados. Com isso, *framework* e modelo de componentes podem ser considerados dois conceitos complementares e fortemente relacionados. As definições estabelecidas pelo modelo de componentes devem ser suportadas pelo *framework*, bem como o *framework* deve respeitar e regular as definições estabelecidas pelo modelo de componentes [BAC02]. Além disso, o uso dos *framework* de componentes possibilita que os desenvolvedores de componentes e aplicações não precisem se preocupar em implementar em suas aplicações inúmeros serviços complexos para troca de mensagens, passagem de dados e ligação dos componentes.

O conceito definido por [BAC02] como *framework* de componentes é identificado em [BRO00] como *infra-estrurura de componentes*, sendo estabelecido também como tipicamente responsável por ao menos uma destas categorias de serviços: empacotamento, distribuição, segurança, gerenciamento de transações e comunicação assíncrona. A Tabela 3 apresenta uma breve definição dos serviços da infra-estrutura de componentes definidos em [BRO00]

Tabela 3 - Categorias de serviços presentes na infra-estrutura de componentes

Item	Descrição
Empacotamento	- definição de uma forma padrão que possibilite a infra-estrutura de componentes saber quais serviços o componente disponibiliza e a assinatura dos métodos que invocam estes serviços; - definição de uma forma padrão para solicitações de serviços a componentes externos;
Distribuição	- ativação e desativação das instâncias dos componentes; - gerenciar a alocação das instâncias para processos remotos; - prover uma transparência de localização, em que o cliente não precise saber onde está a instância do componente que fornece o serviço, bem como a

	instância do componente não precise ter conhecimento da origem das solicitações que recebe;
Segurança	- serviços de controle de acesso, além de serviços que proporcionem conexões seguras quando da transmissão de informações; - níveis de isolamento para garantir segurança e conexões confiáveis entre os componentes;
Gerenciamento de transação	- prover o gerenciamento de transações distribuídas, de modo a controlar e coordenar as interações complexas com os componentes, visando garantir a consistência dos dados;
Comunicação assíncrona	- suporte a comunicação assíncrona entre componentes, que normalmente ocorre através de alguma forma de enfileiramento das solicitações

O relacionamento entre componentes, modelo e *framework* de componentes pode ser identificado na Figura 1, adaptada de [CRN02].

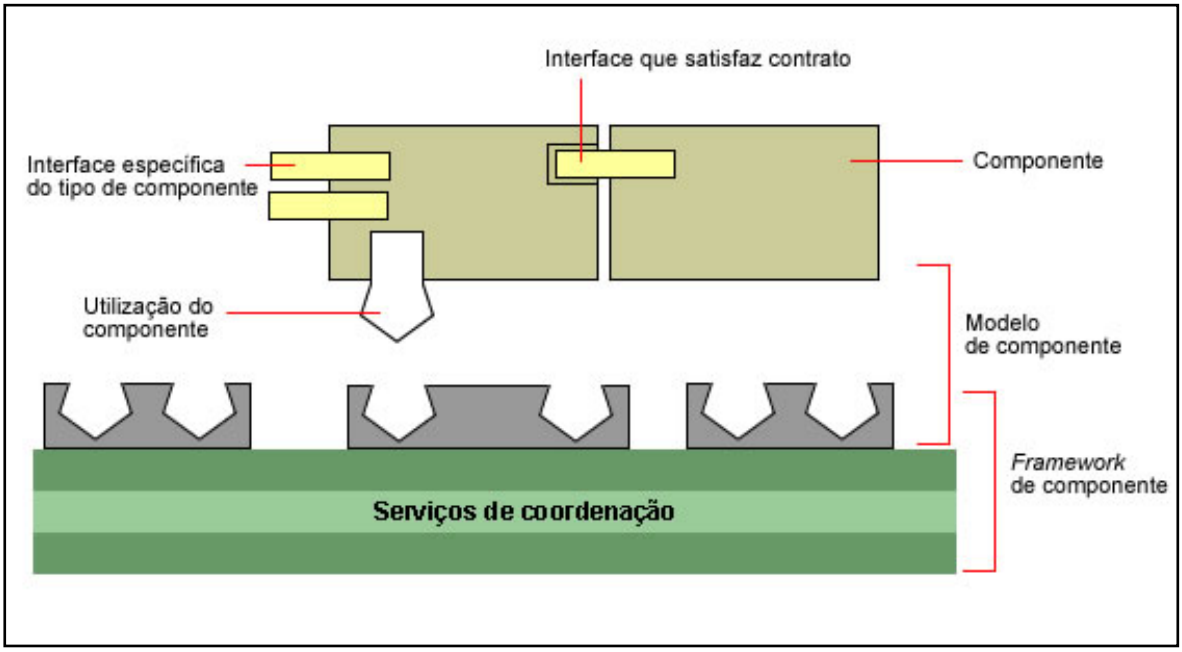

Figura 1 – Relacionamento entre componente, modelo e *framework* de componentes

A Figura 1 permite identificar o *framework* de componentes como a infra-estrutura de suporte a comunicação e ligação dos componentes, através do fornecimento de serviços de coordenação. Já o modelo de componentes identifica as definições quanto a tipos de componentes, formas de interação e recursos necessários. Os componentes definidos para executarem determinadas funcionalidades, devem estar em conformidade com as definições do modelo de componentes, pertencendo a um dos possíveis tipos e respeitando as formas de interação definidas, além de se utilizarem dos serviços disponibilizados. Por fim tem-se que a utilização dos componentes caracteriza a ligação entre o modelo e *framework* de componentes.

Os principais modelos de componentes disponíveis atualmente são: CCM (CORBA *Component Model*), do OMG (*Object Management Group*); DCOM (*Distributed Component Object Model*) e COM/COM+ (*Component Object Model*), da Microsoft; e *JavaBeans* e *Enterprise JavaBeans* (EJB), da Sun Microsystems. Um estudo mais abrangente do modelo EJB é apresentado no Capítulo 4. Estes modelos de componentes apresentam diferentes implementações comerciais ou de domínio público que caracterizam os seus respectivos *framework* de componentes. Para o modelo EJB, por exemplo, existem produtos comerciais como WebShere Application Server (IBM) e BEA WebLogic (BEA Systems) ou de domínio público como JBoss (Tektel) e Jonas (Evidian).

2.3 Orientação a objetos e modelagem

Freqüentemente DBC é considerado o passo seguinte após a programação orientada a objetos. Com isso, não é surpreendente que componentes sejam freqüentemente relacionados com objetos e que algumas vezes o termo componente seja usado como sinônimo de objeto [HEI01]. Entretanto, os conceitos de componentes e objetos são independentes, embora muitos *frameworks* de componentes sejam baseados em conceitos da orientação a objetos.

Justamente pelo fato dos *frameworks* de componentes fazerem uso da tecnologia de objetos é que parece estranho que DBC possa ser independente desta tecnologia. Conforme apresentado em [BRO98], a tecnologia de objetos é útil e um conveniente ponto de partida para DBC. Entretanto, ela não expressa todo o grau de abstração necessária para DBC e é possível realizar o desenvolvimento de componentes sem empregá-la. Sendo assim, a tecnologia de objetos não é nem necessária, nem suficiente para DBC [BRO98].

Contudo, uma das tendências exploradas tem sido a combinação destas duas técnicas. A maior granularidade dos componentes em relação às classes favorece a modularidade e reuso buscados pela modelagem de objetos. Em contrapartida, a rigorosa disciplina de especificação da modelagem de objetos auxilia o projeto baseado em interfaces e a possibilidade de substituição pretendida pelos componentes [KOB00]. Algumas das sinergias de componentes e modelagem de objetos são exploradas por Kobryn [KOB00] através da investigação de como a notação padrão de modelagem de objetos, UML (*Unified Modeling Language*), dá suporte aos *frameworks* de componentes EJB e COM+.

Durante sua investigação, Kobryn [KOB00] apresenta como componentes e o padrão de arquitetura de *frameworks* de componentes podem ser modelados através da UML, seguido da

aplicação da notação aos *frameworks* EJB e COM+. Embora os exemplos apresentados levem Kobryn a concluir que a especificação UML 1.3 consegue modelar muitos dos aspectos de componentes e *frameworks*, ele também identifica algumas questões importantes, entre elas:

- Necessidade de refinamento da semântica dos componentes que atualmente é vaga e apresenta sobreposições com a semântica de classes e subsistemas;
- Atualização da semântica dos componentes deve ser pretendida de modo que se modele componente desde o início do ciclo de vida e não apenas nos diagramas de implementação;
- Definição de um modelo de gerenciamento de construções que de suporte ao desenvolvimento de grandes sistemas baseados em componentes e *frameworks*;
- Melhor especificação de como deve ser a composição de componentes com as diferentes entidades que realizam ou implementam suas interfaces;
- Necessidade de se ter UML *Profiles* padronizados para tecnologias de componentes específicas, como EJB, COM+ e CORBA (*Common Object Request Broker Architecture*).

Ao final, Kobryn [KOB00] ainda apresenta que se tem tido muitas razões para acreditar que UML irá se desenvolver junto aos componentes de modo a atender suas necessidades especiais, e que não será surpresa caso no futuro se tenha UML como uma linguagem de modelagem baseada em componentes, ao invés de orientada a objetos.

Considerando as atuais tendências no desenvolvimento de software e as inúmeras solicitações de melhorias da UML, a OMG vem realizando uma extensa revisão na UML, a partir da qual será disponibilizada a UML 2.0. Conforme [KOB02] a UML 2.0 oferece a possibilidade de resolver a maioria dos problemas comumente associados a UML 1.x, entre os quais estão: tamanho excessivo, complexidade desnecessária, semântica imprecisa, implementações não padronizadas, customização limitada, suporte inadequado para o desenvolvimento baseado em componentes, e incapacidade de intercâmbio de modelos de diagramas. Ainda segundo [KOB02], a UML 2.0 ainda deve tornar o conceito de componente um conceito chave que se desenvolva durante todo o ciclo de vida software, ao invés de apenas uma tardia lembrança da fase de implementação, como muitas vezes apresenta-se na UML 1.x.

3 DESENVOLVIMENTO DE SOFTWARE E COMPONENTES

As características particulares apresentadas por DBC indicam a necessidade de alterações no ciclo de vida de desenvolvimento de software de modo a incorporar atividades próprias desta abordagem. Existem duas perspectivas: *desenvolvimento de componentes* e *desenvolvimento com componentes*. Este capítulo trata do desenvolvimento de software com componentes através do estudo das atividades envolvidas, com ênfase as atividades de adaptação e composição de componentes.

3.1 Perspectivas de desenvolvimento

O desenvolvimento de software a partir de DBC pode considerar o *desenvolvimento de componentes* ou o *desenvolvimento com componentes*. A primeira perspectiva engloba as atividades envolvidas na concepção e implementação de um componente, devendo existir a preocupação em gerar a documentação necessária para que o componente possa ser posteriormente reutilizado. A segunda perspectiva considera a existência de componentes e relaciona às atividades necessárias para o desenvolvimento de software pela composição destes. Esta segunda perspectiva é apresentada mais detalhadamente na Seção 3.2.

As atividades relacionadas ao desenvolvimento de componentes envolvem atividades de especificação e implementação do componente, onde métodos/técnicas/processos como *Catalysis* [DSO98] e UML *Components* [CHE01] podem ser empregadas para ajudar na qualidade do processo de desenvolvimento.

3.2 Desenvolvimento com componentes

O desenvolvimento de software com componentes apresenta um conjunto de atividades relacionadas que levam à implementação de uma aplicação através da composição de componentes já existentes. Desta forma, é importante para esta perspectiva que já existam componentes de software implementados e que estejam disponíveis para serem selecionados e reutilizados na composição dos sistemas.

As atividades essenciais para o desenvolvimento com componentes, segundo [BRO97], são: 1) encontrar componentes com potencial de serem usados no desenvolvimento da aplicação; 2) selecionar componentes que atendam aos requisitos de uma aplicação específica; 3) realizar adaptações; 4) realizar a composição dos componentes; e 5) atualizar os componentes.

A Figura 2, adaptada de [BRO97], ilustra as cinco atividades propostas em [BRO97] e cada uma delas é brevemente caracterizada a seguir.

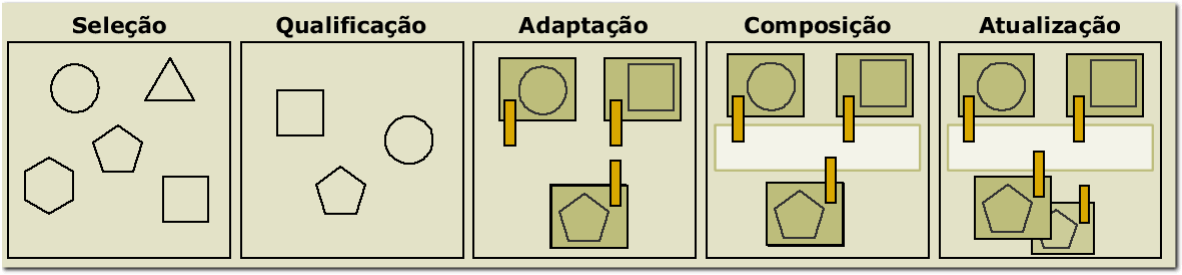

Figura 2 - Atividades do desenvolvimento *com* componentes

- *Seleção:* esta primeira atividade consiste em pesquisar e selecionar componentes que apresentam potencial de serem usados no desenvolvimento do sistema [BRO97]. Os componentes podem ser procedentes de diferentes origens, incluindo componentes já desenvolvidos para outros projetos e componentes desenvolvidos e comercializados por diferentes empresas. Logo, é necessário realizar um processo de investigação das propriedades e da qualidade do componente, embora nesta etapa saiba-se pouco das características dos componentes. As informações disponíveis podem dizer respeito simplesmente ao nome do componente, seus parâmetros e uma breve descrição do ambiente de execução. O objetivo desta atividade é gerar uma relação de componentes candidatos, os quais serão avaliados na próxima etapa para a determinação daqueles que realmente serão utilizados. As dificuldades intrínsecas desta atividade de seleção correspondem à definição do que pode ser visto como um componente por parte da aplicação e, com isso, qual componente, entre os existentes, é o mais apropriado para ser utilizado. Por fim, caso nenhum candidato a componente tenha sido selecionado nesta etapa, passa-se a ter a necessidade de desenvolver o componente ao invés de reutilizar um já existente, partindo assim para o processo de desenvolvimento de componentes.

- *Qualificação:* esta etapa consiste em garantir que o componente candidato executa as funcionalidades necessárias, ajusta-se ao estilo arquitetural definido para o sistema e apresenta qualidades que são necessárias para a aplicação (desempenho, confiabilidade, usabilidade) [PRE01]. Segundo [BRO97], são verificas as interfaces que podem ser fonte

de conflito e identifica-se a sobreposição entre componentes. A qualificação geralmente envolve uma análise detalhada de toda a documentação ou especificação disponível, discussão com os desenvolvedores dos componentes e usuários, e teste de execução do componente em diferentes cenários [BRO97]. Ainda segundo [BRO97], quanto mais rigorosas forem as descrições dos componentes disponíveis, mais a atividade de qualificação pode ser reduzida a validar o funcionamento através de experiências e do uso temporário do componente.

- *Adaptação*: atividade que busca corrigir potenciais fontes de conflito entre os componentes selecionados e qualificados para compor o sistema [BRO97]. Esta é uma atividade importante pois, segundo [BOS99], na maioria das vezes o reuso de um componente está relacionado a alguma forma de adaptação para que ele atenda aos requisitos da aplicação. Um estudo mais detalhado desta atividade é apresentado na Seção 3.3.

- *Composição*: nesta etapa é realizada a composição dos componentes através de uma infra-estrutura comum [BRO97]. Para uma composição consistente dos componentes, a infra-estrutura deve prover a ligação dos componentes e um conjunto de convenções conceituais compartilhadas pelos componentes. A Seção 3.4 apresenta um estudo mais detalhado desta atividade.

- *Atualização*: como última atividade tem-se a atualização dos componentes, onde versões antigas serão substituídas ou novos componentes, com comportamento e interface similares, serão incluídos. Esta atividade tem importantes implicações, pois sem um cuidadoso planejamento, uma alteração em um componente pode provocar inúmeras repercussões inesperadas em muitos outros componentes do sistema [BRO97].

As cinco atividades apresentadas descrevem possíveis etapas no desenvolvimento de software com componentes. Entretanto, não existe uma ordem para sua realização ou mesmo a obrigação de sua realização. As etapas necessárias, bem como a importância e o esforço empregados em cada uma correspondem a uma decisão relacionada ao processo de desenvolvimento com componentes específicos a uma aplicação.

3.3 Adaptação de componente

Adaptação de componente é o processo de alteração de um componente para o uso em uma aplicação específica. Pesquisas relacionadas ao reuso de software, segundo [BOS99], têm

mostrado que raramente um componente é reutilizado como foi originalmente desenvolvido e que geralmente necessita de alguma forma de alteração para se adequar à arquitetura da aplicação ou aos demais componentes.

Os projetistas de componentes não têm condições de prever todos os possíveis usos que um componente pode assumir. Desta forma, é necessário que os desenvolvedores das aplicações tenham mecanismos para adaptar facilmente um componente sem ter o conhecimento do código fonte [HEI99a].

As técnicas de adaptação de componentes podem ser classificadas, segundo [BOS99], como técnicas de caixa-branca (*white-box*) e caixa-preta (*black-box*). As técnicas de caixa-branca definem que a adaptação de um componente reutilizável ocorre através de mudanças, substituição ou exclusão de partes da sua especificação interna. Em contrapartida, as técnicas de caixa-preta partem do reuso do componente como ele é, adaptando apenas as interfaces do componente. Assim, esta segunda técnica exige unicamente o entendimento das interfaces do componente e não de sua especificação interna.

Embora *adaptação*, *customização* e *evolução* sejam termos empregados para indicar mudanças relacionadas a componentes, estes termos apresentam algumas diferenças. Segundo [HEI99a], *evolução* diz respeito a mudanças realizadas nos componentes por seus projetistas e *adaptação* corresponde a alterações realizadas por um desenvolvedor de aplicação para um uso possivelmente bastante diferente do componente em uma aplicação. Adaptação também é diferenciada de customização em [HEI99a], que apresenta *customização* como a seleção de uma alternativa de uso entre um conjunto pré-estabelecido de possibilidades como, por exemplo, parametrização. Com isso, tanto adaptação quanto evolução podem ser vistas como formas de estender as propriedades de um componente.

3.3.1 Requisitos para adaptação

Existe na literatura ([HEI99a][BOS99][WEI01]) uma relação de requisitos que usualmente devem ser observados pelas técnicas de adaptação de componentes. Estes requisitos podem ser empregados para avaliar e selecionar uma técnica, bem como podem servir como base para a definição de novas técnicas. A seguir são discutidos requisitos para adaptação apresentados em [BOS99].

- *Caixa-preta:* O componente e sua adaptação são duas entidades separadas, o que exige que a técnica de adaptação não tenha acesso à implementação interna do componente, estando o acesso limitado às interfaces do componente.
- *Transparência:* A adaptação de um componente deve ser tão transparente quanto possível. Neste contexto, transparência indica que tanto a aplicação que usa o componente adaptado como o componente propriamente, desconsideram a existência da adaptação entre eles. Em [HEI99a] este requisito é apresentado subdividido como:
    - Homogêneo: a aplicação que usa o componente adaptado deve usá-lo da mesma forma de como usava o componente original.
    - *Conservador*: aspectos do componente original que não tenham sido adaptados devem ser acessados sem um esforço explícito pelo componente adaptado, ou seja, o componente original permanece atuando em conjunto ao componente adaptado, provendo as atividades que não tiverem sido adaptadas.
    - *Ignorante*: o componente original não deve ter conhecimento da existência do componente adaptado.
- *Componível:* As técnicas de adaptação devem ser facilmente compostas com o componente para o qual é aplicada, ou seja, não devem ser necessárias redefinições do componente. O componente adaptado deve poder ser composto com outros componentes assim como era sem as adaptações. Desta forma, uma adaptação deve poder ser composta com outra adaptação e, com isso, possibilitar que um componente seja adaptado usando diferentes tipos de adaptações.
- *Reusável:* Uma técnica de adaptação consiste geralmente de uma parte genérica (adaptação) e outra específica (componente). As tradicionais técnicas de adaptação apresentam como problema o fato de nem a parte genérica, nem a parte específica serem reusáveis por não se conseguir separá-las. Com isso a adaptação tem que ser implementada novamente sempre que necessária, ao invés de reutilizada.
- *Configurável:* Para que a técnica seja útil e reusável ela deve prover suficiente formas de configuração da adaptação, de modo que esta possa ser aplicada a diferentes componentes.

Complementando a relação apresentada em [BOS99], a seguir são apresentados dois requisitos adicionais apresentados em [HEI99a].

- *Independente de framework:* A técnica de adaptação não deve ser dependente do *framework* de componentes do qual o componente faz parte. Com isso, uma técnica de adaptação deve funcionar igualmente com componentes que seguem os modelos CCM, COM e EJB, por exemplo.

- *Independente de linguagem de programação:* A técnica de adaptação deve ser independente da linguagem usada para implementar o componente, podendo ser empregada da mesma forma em diferentes linguagens.

3.3.2 Técnicas de adaptação

As técnicas de adaptação muitas vezes estão relacionadas a características específicas do paradigma de programação, ou mesmo das próprias linguagens de programação, podendo também ser análogas a determinados padrões de projeto. Em [BOS99], [HEI99b] e [WEI01] são descritas diferentes técnicas de adaptação, das quais quatro são apresentadas a seguir.

Alteração de código

Uma primeira alternativa que pode ser considerada é a modificação explícita e direta do código do componente. Esta técnica apresenta-se como totalmente caixa-branca, permitindo ao desenvolvedor da aplicação realizar toda e qualquer adaptação que considerar pertinente no componente. A grande desvantagem de uma técnica como esta é a necessidade de um conhecimento detalhado do funcionamento interno do componente.

Herança

Esta técnica de adaptação caixa-branca está relacionada ao uso do mecanismo de herança da orientação a objetos. De acordo com a linguagem orientada a objetos, todos os aspectos internos, ou apenas parte, estarão disponíveis ao componente adaptado [BOS99]. Por exemplo, em Smalltalk todos os métodos e variáveis de instância são disponibilizados às subclasses, enquanto que em C++ e Java o uso das palavras-chave *private* e *protected* é que determinarão o que será disponibilizado às subclasses. Segundo [BOS99], a vantagem importante desta técnica é que o código continua a existir em um mesmo local. Em contrapartida, a grande desvantagem é a necessidade de se ter um conhecimento funcional detalhado da superclasse para definir as subclasses necessárias na adaptação do componente.

Wrapping

O funcionamento desta técnica, que pode ser descrita pelo padrão *Adapter* [GAM00], está baseado na definição de um componente, denominado de *wrapper*, que encapsula o componente

original e atua como filtro para as requisições recebidas, determinando o comportamento do componente como desejado. A principal aplicação desta técnica de caixa-preta está relacionada à solução de interfaces incompatíveis. Uma importante desvantagem desta técnica, segundo [BOS99], é o fato de poder resultar em uma considerável sobre carga de implementação, caso toda a interface do componente precise ser controlada pelo *wrapper*, incluindo aquelas que não precisam ser adaptadas.

Proxy

Nesta técnica de caixa-preta é criado um componente, denominado *proxy*, que intermedia a comunicação com um componente original, definindo como devem ser atendidas as requisições endereçadas ao componente. Esta técnica, que pode ser descrita pelo padrão de projeto *Proxy* [GAM00], difere-se da técnica *Wrapping* por não encapsular o componente original, e sim posicionar um *proxy* entre o componente original e as requisições, de modo a filtrá-las, mantendo a possibilidade de acesso direto ao componente original.

Em [BOS99], [HEI99a], [HEI99b] e [WEI01] estão relacionadas diferentes técnicas de adaptação (caixa-branca e caixa-preta), bem como são apresentadas tabelas comparativas das técnicas e requisitos com as quais estão em conformidade.

3.4 Composição de componentes

Composição é o termo usado em DBC para definir como os sistemas são formados. Segundo [BAC02], não existem diferenças entre os termos composição e integração, entretanto composição é mais amplamente usado na literatura relacionada a DBC.

Os componentes são compostos de modo que possam interagir e, segundo [BAC02], verifica-se dois tipos de entidades que podem ser compostas: componentes e *frameworks* de componente. Desta forma, [BAC02] apresenta três principais classes de interação nos sistemas baseados em componentes:

- *Componente–Componente:* composição que possibilita interações entre componentes e define as funcionalidades da aplicação. O compromisso que especifica estas interações pode ser classificado como contrato em nível de aplicação.

- *Framework–Componente:* composição que possibilita interações entre os *frameworks* de componentes e seus componentes. Estas interações habilitam os *frameworks* a

gerenciarem os componentes e o compromisso que especifica estas interações pode ser classificado como contrato em nível de sistema.

- *Framework–Framework:* composição que possibilita interações entre *frameworks* e permite a composição de componentes definidos em diferentes *frameworks*. O compromisso que especifica estas interações pode ser classificado como contrato de interoperação.

Os diferentes tipos de contratos apresentados anteriormente são descritos por [BAC02] como as seguintes formas de composição:

- *Utilização do componente:* os componentes devem ser empregados através de um *framework* antes de serem compostos e executados. A utilização envolve a definição da interface que o componente deve implementar de modo que o *framework* possa gerenciar seus recursos, implementando desta forma o contrato em nível de sistema.

- *Utilização do framework: frameworks* podem ser utilizados dentro de outros *frameworks*, o que corresponde à concepção apresentada em [SZY99] na qual os *frameworks* podem apresentar diferentes camadas. O contrato é análogo ao contrato de utilização do componente.

- *Composição simples:* componentes utilizados dentro de um mesmo *framework* podem ser combinados. A composição expressa funcionalidades específicas dos componentes e da aplicação, implementando assim um contrato em nível de aplicação. Os mecanismos necessários para esta interação são providos pelo *framework*.

- *Composição heterogênea:* composição de componentes através de diferentes *frameworks*, suportada pelos *frameworks* em camadas. Devem ser estabelecidos compromissos primeiramente entre os componentes e seus *frameworks*, e posteriormente é definida a composição. Neste caso, além da implementação de contratos em nível de sistema, também é necessário a implementação de contratos ponte de modo a possibilitar a interação de modelos de componentes bastante genéricos.

- *Extensão de framework:* os *frameworks* podem ser tratados como componentes e, desta forma, serem compostos como outros componentes. Esta forma de composição normalmente permite a parametrização do comportamento do *framework* através de seus conectores (*plug-ins*). Contratos de conectores padrão para serviços providos são cada vez mais comuns em *frameworks* comerciais.

- *União de componentes:* forma de composição em que são compostos componentes e outras uniões parciais de componentes, as quais podem conter um ou mais componentes. Estas composições entre componentes implementam contratos em nível de aplicação.

Observando as tecnologias de componentes mais representativas tem-se claro que embora definam formas genéricas de composição, não existe um consenso de quais tipos de composição devem ser suportados pelos modelos e *frameworks* de componentes [BAC02].

Além da apresentação da definição das formas de composição, em [BAC02] também é abordado como a composição ocorre através da discussão do tempo em que se dá a associação dos recursos. Essencialmente, dois componentes são compostos quando os recursos providos por um componente passam a ser acessíveis pelos outros componentes, os componentes clientes. Conforme [BAC02], podemos pensar no tempo de associação como uma linha de tempo que tem de um lado a associação prévia e de outro lado a associação tardia. Observando o processo de desenvolvimento da perspectiva de componentes de software, a associação prévia necessita que os desenvolvedores de componentes tomem algumas decisões que possibilitarão efetivamente associar algum recurso ao componente, ou mais especificamente regras que indicarão como essa associação vai ocorrer. A associação tardia significa exatamente o oposto, pois os desenvolvedores de componentes não devem tomar nenhuma decisão que regule futuras associações.

A associação tardia, entretanto, exige a imposição de restrições adicionais por parte do *framework* de componentes aos desenvolvedores de componente. Estas restrições correspondem a como devem ser apresentados os serviços dos componentes e como os componentes podem interagir quando seus serviços forem utilizados. Esta maior complexidade na definição dos componentes é o preço pago para possibilitar a associação tardia. Entretanto, os *frameworks* que possibilitam esse tipo de associação oferecem maior flexibilidade para a substituição de componentes, e para a integração com outras partes e componentes comerciais.

4 TECNOLOGIA DE COMPONENTES

O desenvolvimento de sistemas baseados em componentes, bem como o desenvolvimento dos próprios componentes de código é viabilizado através do uso das diferentes tecnologias de componentes. Este capítulo apresenta e caracteriza a tecnologia EJB, bem como a posiciona frente ao referencial teórico discutido nos capítulos anteriores.

4.1 Enterprise JavaBeans

A especificação J2EE (*Java 2 Platform, Enterprise Edition*) definida pela Sun Microsystems, fornece uma abordagem baseada em componentes para o projeto, desenvolvimento, composição e utilização de aplicações corporativas [BOD02]. A plataforma J2EE define um conjunto de APIs (*Application Programming Interface*) entre as quais está EJB, implementada através dos pacotes e subpacotes `javax.ejb`.

Entretanto EJB não é apenas uma API e caracteriza-se como uma arquitetura de componentes para o desenvolvimento e utilização de aplicações corporativas baseadas em componentes distribuídos [DEM02]. Considerando uma arquitetura cliente/servidor, os componentes EJB são componentes distribuídos posicionados do lado servidor acessados por diferentes aplicações cliente. Neste sentido, os componentes EJB são desenvolvidos para realizarem típicas operações do lado servidor, como executarem algoritmos de diferentes complexidades, ou coordenarem e executarem grandes volumes de transações. Utilizando EJB é possível desenvolver aplicações distribuídas escaláveis e seguras sem ter que se preocupar em implementar todos os procedimentos necessários para prover a distribuição.

EJB é definido, segundo [DEM02], como sendo uma arquitetura de componentes. Ao mesmo tempo, a literatura relacionada a componentes estabelece EJB como sendo um modelo de componentes. Desta forma, o que se tem é que a arquitetura EJB compreende entre as suas definições a especificação de um modelo de componentes flexível que fornece um conjunto de convenções, definições e regras para a implementação de componentes do lado servidor utilizando a linguagem Java. A arquitetura EJB corresponde ao conjunto total de definições relacionadas a EJB, enquanto o modelo de componentes corresponde apenas a um subconjunto destas definições.

Em [DEM02] são apresentadas definições quanto aos tipos de componentes EJB, a infra-estrutura que dá suporte a utilização destes componentes, o relacionamento entre a infra-estrutura e os componentes, e o relacionamento entre os componentes. Estas definições são realizadas em termos de contratos que devem ser obedecidos. A partir disso é esperado que as aplicações possam fazer uso de componentes EJB de diferentes origens, pois estes componentes obrigatoriamente terão de estar em conformidade com as definições estabelecidas (modelo de componentes).

Compreender EJB está centrado no entendimento das três principais entidades da arquitetura: *bean*, *container* EJB e servidor EJB. Cada uma destas entidades é apresentada a seguir e o relacionamento entre elas é ilustrado na Figura 3.

- *Enterprise bean:* ou apenas *bean*, é o nome atribuído aos componentes EJB, que caracterizam-se como componentes distribuídos hospedados nos *containers* EJB [MON00].

- *Container* EJB: é local onde os *beans* são disponibilizados para serem utilizados e tem por função gerenciar todos os aspectos do *bean* durante a execução deste, incluindo: acesso remoto, segurança, persistência, transações, concorrência e acesso a recursos [MON00]. Os *beans* não podem funcionar fora do *container*, embora a existência deste deva ser transparente para o *bean*. O *container* tem por função intermediar tanto o acesso dos clientes ao *bean*, como a relação entre *bean* e servidor EJB. Segundo [MON01], *container* é mais um conceito do que uma entidade realmente existente, pois embora a especificação EJB defina o modelo de componentes em termos de responsabilidade do *container*, não existe uma clara distinção entre *containers* e servidores EJB.

- Servidor EJB: embora não exista uma clara distinção em relação aos *containers* EJB, o papel do servidor é apresentar diferentes *containers*, um para cada tipo de *bean*, e prover serviços que permitam que os *containers* gerenciem os *beans*. Segundo [HEI01], muitos dos produtos comerciais, implementações do servidor e/ou *container* EJB, têm desenvolvido o servidor e o *container* como um único produto. Atualmente, existem diferentes possibilidades de produtos comerciais e de domínio público para servidores/*containers* EJB. Como exemplos de produtos comerciais têm-se o WebSphere Application Server da IBM e o BEA WebLogic Server da BEA Systems. Já na categoria de domínio público, existem: jBoss da Tektel, JOnAS da Evidian e OpenEJB (EJB Container Provider) da Intalio.

A Figura 3 ilustra o relacionamento existente entre *bean*, *container* EJB e servidor EJB. Embora a figura apresente apenas um *bean* e um *container* EJB, é possível que mais de um *bean* esteja hospedado em um mesmo *container*, bem como um servidor EJB pode disponibilizar diferentes *containers*.

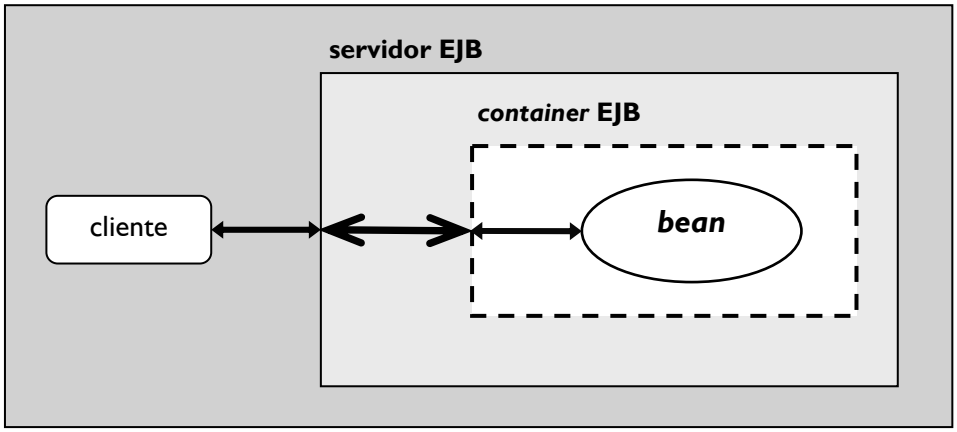

Figura 3 - Principais conceitos da arquitetura EJB

Enquanto os componentes EJB definem os serviços que serão expostos a seus clientes através de suas interfaces, os *containers* correspondem ao ambiente onde estes componentes são disponibilizados. Os servidores EJB aparecem como responsáveis por definir a infra-estrutura que permite que os *containers* gerenciem os *beans*. Desta forma, o papel do servidor EJB é prover diferentes serviços, bem como possibilitar o funcionamento e acesso aos *containers*. Alguns dos serviços providos pelo servidor EJB são apresentados a seguir:

- *Serviço de transações:* transação é uma unidade de atividade ou um conjunto de atividades que são executadas juntas. Por sua característica atômica, todas as tarefas têm de ser completadas para que a transação seja finalizada com sucesso. Os servidores EJB gerenciam transações automaticamente. A simples declaração dos atributos de transação por parte do desenvolvedor, quando o *bean* é disponibilizado, indicam ao servidor EJB como gerenciar o *bean* em tempo de execução. Entretanto, EJB não apresenta um mecanismo que permita ao *bean* gerenciar as transações diretamente, caso seja necessário [MON01].

- *Serviço de segurança:* os servidores EJB podem prover diferentes tipos de serviços de segurança. Embora muitos servidores EJB forneçam comunicação segura (i.e. ocorre um isolamento da comunicação ou criptografia) e algum mecanismo de autenticação (garantir acesso apenas a usuários autorizados), apenas o controle de acesso é especificado no modelo de componentes EJB. Segundo [MON01], um mecanismo de autenticação possivelmente será especificado em uma versão subseqüente da especificação EJB [DEM02], entretanto comunicação segura provavelmente nunca será

especificada por não estar relacionada apenas a definições da arquitetura EJB e dos protocolos de objetos distribuídos. EJB especifica que o controle de acesso é realizado através da indicação de identidades seguras a todas as aplicações clientes que fizerem acesso ao servidor. Quando um *bean* tiver um método invocado ele deverá verificar se quem esta fazendo a invocação está autorizado a realizá-la.

- *Serviço de nomes:* diferenciando-se apenas em como são implementados, os serviços de nome provêem um mecanismo para a localização de objetos ou recursos distribuídos. Este serviço deve tanto permitir o registro de nomes associados a objetos distribuídos, como possibilitar obter a referência de um objeto através do fornecimento do seu nome. EJB provê o serviço de nomes a partir da utilização da API JNDI (*Java Naming and Directory Interface*) que permite encontrar e acessar *beans* sem saber sua localização na rede.

- *Persistência:* persistir os componentes é garantir que o comportamento e os dados associados sejam armazenados de alguma forma. Este serviço está diretamente relacionado ao conceito de *Container-managed persistence*, abordado mais adiante. Cada fabricante define o mecanismo que irá garantir ao servidor EJB realizar persistência, entretanto estes mecanismos devem suportar os métodos EJB *callback* e transações. Os métodos EJB *callback* são métodos que permitem ao servidor/*container* EJB realizar notificações ao *bean*, sendo que estas notificações correspondem ao aviso do acontecimento de algum determinado evento [MON01].

Após esta descrição inicial da arquitetura EJB e apresentação dos principais conceitos relacionados, as próximas seções irão tratar das demais características de EJB. Finalizando este capítulo, a Seção 3.2 apresenta uma caracterização da tecnologia EJB frente ao referencial teórico abordado nos capítulos anteriores.

4.1.1 Tipos de componentes EJB

A arquitetura EJB define três tipos de *beans*. A justificativa para a existência destes diferentes tipos, segundo [ROM02], está relacionada ao fato da Sun Microsystems não ser a única empresa envolvida na definição da arquitetura EJB. Como as várias empresas envolvidas lidam com clientes que necessitam de diferentes sistemas distribuídos, essa flexibilização foi estipulada de modo a atender diferentes necessidades. Entretanto, embora esta flexibilização possibilite que estas diferentes necessidades sejam atendidas, a definição de diferentes tipos de *beans*

potencialmente aumenta o tempo de aprendizagem da tecnologia EJB, bem como pode possibilitar que exista um uso incorreto dos diferentes tipos.

Os tipos de *beans* definidos são: *session bean*, *entity bean* e *message-driven bean*. Todos os *beans*, independente do seu tipo, são localizados no lado servidor, entretanto *session* e *entity bean* são componentes baseados em RMI[1] (*Remote Method Invocation*) acessados usando protocolos de objetos distribuídos. Já o *message-driven bean* (definido apenas a partir da versão 2 da especificação EJB) é um componente assíncrono que recebe e processa mensagens, e está baseado no serviço JMS[2] (*Java Message Service*) [MON01]. Cada um dos tipos de *beans* é apresentado mais detalhadamente nas próximas subseções.

4.1.1.1 Session bean

Este tipo de *bean* representa extensões das aplicações cliente e é responsável por executar processos ou tarefas [MON01] que podem ser tanto acessos ao banco de dados, como execução de diferentes operações. Ou seja, os *session beans* implementam a lógica, as regras de negócio, algoritmos e fluxos de trabalho [ROM02].

Estes *beans* são instanciados pelo *container* EJB quando da solicitação de um cliente, e apresentam um período de vida curto, geralmente relativo apenas à execução do procedimento solicitado pelo cliente. Além disso, os *session beans* representam objetos em memória e, por isso, não sobrevivem à ocorrência de problemas no servidor de aplicação [ROM02].

Os *session beans* podem ser caracterizados como *stateful* ou *stateless*:

- *Stateful:* a utilização de um *session bean* pode estar relacionada a inúmeras invocações de métodos do *bean* por parte do cliente, de modo a realizar algum processo de negócio. O *session bean stateful* se caracteriza por suas instâncias manterem o seu estado consistente entre as diferentes chamadas de métodos feitas pelos clientes. Manter o estado consistente representa atualizar os valores das suas variáveis enquanto o cliente estiver conectado, sendo que estas variáveis representam o estado de um cliente em particular. Com isso, as variáveis alteradas na execução de um método terão seu estado atualizado e novas utilizações farão acesso a este novo estado. Um exemplo de utilização para este tipo de *bean* pode ser um *site* de *e-commerce*, em que os clientes podem adicionar produtos à cesta de compras. A cada nova adição de produto o cliente

[1] RMI é forma nativa da linguagem Java de prover a comunicação entre objetos distribuídos. É através do uso da API Java RMI que EJB suporta a distribuição dos componentes. Também existe o RMI-IIOP (*Internet Inter-ORB Protocol*), que é uma extensão de RMI usada para integração de EJB com CORBA. Em RMI-IIOP, se faz uso do protocolo IIOP para a comunicação, a partir do mecanismo RMI de distribuição.

[2] JMS também é uma API da especificação J2EE, a qual possibilita a comunicação através de mensagens.

está executando uma nova interação, devendo haver a atualização e manutenção do estado entre estas diferentes interações.

- *Stateless:* este tipo de *bean* representa a interação com clientes através de uma única requisição, ou seja, o trabalho do *bean* encerra quando ele responde à requisição recebida. Como não existe uma conversação (ciclo de requisições-respostas), não existe também a necessidade de haver a manutenção do estado do *bean*. Após a execução do método, o *container* EJB pode optar por destruir o *bean* ou recriá-lo, preparando-o para novas invocações. Esta é uma decisão específica do algoritmo implementado pelo *container* EJB. Um exemplo de utilização para este tipo de *bean* pode ser um componente para verificação de cartão de crédito, onde o componente recebe um conjunto de parâmetros (*e.g.* número do cartão, data de expiração, nome do titular, valor da compra) e de acordo com a existência ou não de crédito emite uma única resposta. Após responder a uma solicitação o *bean* está pronto para receber novas solicitações e não armazena nenhuma informação sobre as solicitações já respondidas.

4.1.1.2 Entity bean

Este tipo de componente representa *beans* que sabem tornar seus dados permanentemente persistentes em dispositivos de armazenamento, como banco de dados e sistemas legados [ROM02].

Entity beans são completamente diferentes de *session beans*, pois não executam tarefas complexas ou implementam o fluxo de trabalho da aplicação, mas representam dados, informações simples ou complexas, que são guardadas nos dispositivos de armazenamento. Os *entity beans* representam objetos do negócio (*e.g.* produtos, clientes) e geralmente cada um destes *beans* possui um mapeamento para uma tabela de banco de dados relacional, e cada uma de suas instâncias representa uma linha da tabela. *Entity* e *session beans* também se diferenciam quanto ao seu tempo de vida, os *entity beans* tem um ciclo de vida geralmente muito mais longo que um *session bean*, já que não dependem da duração da sessão do cliente e sim da existência dos dados no banco de dados [ROM02].

Como os *entity beans* representam dados armazenados, deve existir uma sincronização periódica entre o *bean* e o dispositivo em que eles são armazenados. A partir disso, existem duas formas para definir a persistência dos dados em EJB, as quais se diferenciam quanto ao responsável por implementá-la.

- *Bean-managed persistence*: a persistência deve ser garantida pelos próprios *beans*, onde o desenvolvedor do componente deverá se responsabilizar por escrever todo o código necessário para incluir, salvar, carregar e encontrar os dados. O momento da realização da sincronização dos dados é definido pelo *container* EJB, o qual dispara os procedimentos escritos pelo desenvolvedor do componente para realizar a persistência.

- *Container-managed persistence*: a persistência dos dados é realizada pelo *container* EJB, isentando os desenvolvedores de *bean* de escrever o código necessário para implementá-la. Neste caso, tanto o momento da execução da sincronização, como a sincronização propriamente dita, é definido pelo *container*. Assumindo-se a existência dessa possibilidade, os servidores EJB devem prover mecanismos para garantir a persistência automaticamente. Existem diferenças entre os produtos (servidores EJB comerciais ou de domínio público) quanto ao nível de suporte, onde alguns fornecem sofisticados mapeamentos de objeto para banco de dados relacionais, enquanto outros são bastante limitados.

4.1.1.3 Message-driven bean

Este terceiro tipo de *bean* é especificado apenas a partir da versão 2 da arquitetura EJB. *Message-driven bean* é um tipo de *bean* bastante diferente dos dois anteriores, os quais representam componentes do lado servidor que são baseados em RMI, ou seja, são componentes distribuídos que podem ter seus métodos invocados por clientes espalhados pela rede. Os *message-driven beans* são componentes que recebem mensagens enviadas por qualquer cliente JMS [ROM02], o que define que o acesso a estes *beans* não ocorre através da invocação de métodos, mas sim através do envio de mensagens. Dizer que se trata de componentes assíncronos está relacionado ao fato de que os clientes JMS que enviam as mensagens não aguardam retorno do *bean* para continuarem executando, pois estes *beans* não apresentam nenhum tipo de retorno.

4.1.2 Estrutura e implementação dos componentes EJB

O desenvolvimento dos componentes EJB corresponde à codificação de um conjunto de arquivos, tendo-se a definição de diferentes classes e interfaces. Os diferentes arquivos gerados implementam o funcionamento do componente e as interfaces necessárias para a sua execução, bem como as interfaces que identificam os serviços que estes componentes disponibilizam a seus clientes. Em virtude dos diferentes tipos de beans, existe a possibilidade de se ter diferentes cenários no desenvolvimento com componentes EJB em que diferentes classes e interfaces devem

ser codificadas. As subseções a seguir detalham partes, classes e interfaces que definem a estrutura do *bean* e são implementados no decorrer do desenvolvimento de componentes EJB.

4.1.2.1 Classe enterprise bean

A primeira parte do componente, chamada de classe *enterprise bean*, corresponde a implementação propriamente dita do componente. Esta parte é implementada através de uma típica classe Java que apresenta a codificação da lógica que estabelece as funcionalidades do componente.

A implementação da classe *enterprise bean* é diferente para cada um dos tipos de *beans* [ROM02], em virtude dos diferentes fins para os quais eles se propõem. Enquanto que nos *session beans* esta classe apresenta a lógica relacionada ao processo de negócio, nos *entity beans* ela descreve a lógica relacionada aos dados e nos *message-drive beans*, a lógica orientada às mensagens.

A especificação EJB estabelece algumas interfaces que a classe *enterprise bean* deve implementar, o que obriga o componente a expor métodos que todos os *beans* devem prover, de modo a estar em conformidade com o que foi estabelecido no modelo de componentes EJB [ROM02]. O *container* EJB faz chamadas a estes métodos tanto para gerenciar o *bean* como para alertá-lo de eventos significativos. Estes métodos não são visíveis aos clientes do *bean*, pois apenas o *container* está habilitado a chamá-los.

Independente de tipo, todas as classes *enterprise bean* têm de implementar a interface `javax.ejb.EnterpriseBean`. A implementação desta interface corresponde apenas a uma indicação de que a classe é de fato uma classe *enterprise bean*. Considerando os diferentes tipos de *beans*, cada um dos tipos possui uma interface específica que estende a interface `javax.ejb.EnterpriseBean`. Desta forma, a interface `javax.ejb.EnterpriseBean` nunca precisa ser implementada diretamente na classe *enterprise bean*, já que o *bean* irá implementar a interface correspondente ao seu tipo [ROM02]. Cada tipo apresenta a sua respectiva interface, bem como esta interface irá forçar o componente a implementar um determinado conjunto de métodos, através dos quais o *container* se comunica e gerencia o *bean*.

4.1.2.2 Remote interface

A *remote interface* expõe os métodos que poderão ser acessados pelas aplicações externas ao *container* EJB. Esta *interface* relaciona quais métodos podem ser acessados e como eles podem ser acessados. A partir da especificação desta *interface*, o *container* gera os EJB *objects*, que

implementam a *remote interface* e correspondem a visão que o usuário tem do *bean*. Desta forma, quando um método de um *bean* é invocado, esta solicitação é intermediada pelo *container* através do EJB *object*, que sabe os métodos do *bean* que o cliente pode invocar por implementar a *remote interface*. Embora exista essa intermediação, o processamento da chamada do método é realizado pelo próprio *bean*, que retorna o resultado ao EJB *object*, que por sua vez informa o resultado ao cliente.

4.1.2.3 Home interface

A *home interface* define os métodos relativos ao ciclo de vida de um *bean*, originando os métodos para a criação, localização e remoção de *beans*. A partir da especificação desta *interface* o *container* gera *home object*, que implementa a *home interface*, e tem por função criar, localizar e remover os EJB *objects*.

A Figura 4 ilustra o relaciona entre as interfaces *remote* e *home* e os respectivos objetos, EJB *object* e *home object*, criados pelo *container* e que possibilitam a interação com o *bean*.

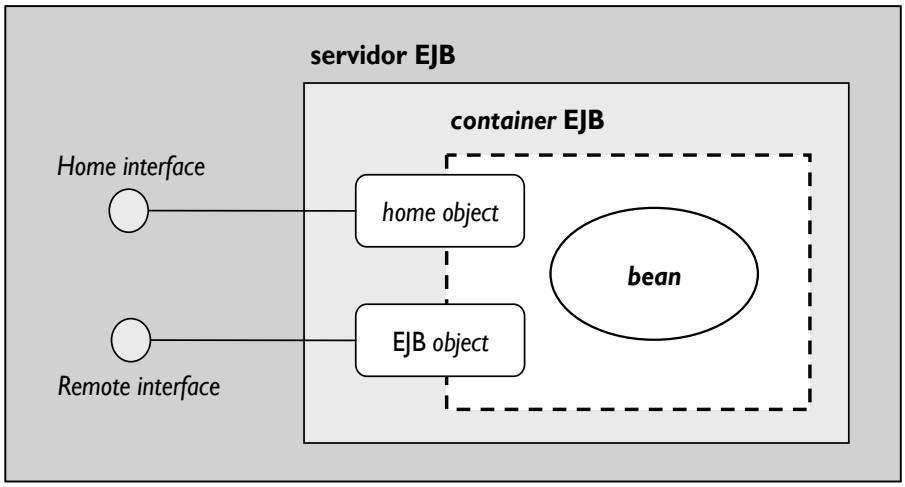

Figura 4 - Interfaces e objetos relacionados a implementação de *beans*

4.1.2.4 Local interface

A *local interface* foi definida a partir da versão 2 da especificação EJB e seu uso é opcional. Esta interface foi definida com o propósito de diminuir a sobrecarga de comunicação entre os *beans* quando do uso da *remote interface*. A *local interface* só pode ser usada entre componentes que estão no mesmo *container*, pois a diminuição da sobrecarga foi estabelecida através da retirada das atividades relativas ao protocolo de distribuição. Esta interface é equivalente a *remote interface*, apenas com o diferencial que trata apenas componentes locais a um mesmo *container*.

4.1.2.5 Descritor

O desenvolvedor de *beans* pode delegar ao *container*/servidor EJB muitas responsabilidades, como, por exemplo, o uso de serviços de transação, ao invés de implementá-los. Neste sentido, de forma a informar ao *container* as necessidades e características do *bean*, o desenvolvedor do componente deve definir um descritor (*deployment descriptor*) que relacione todas as definições que regulam o funcionamento do *bean* no *container*. A partir da versão 2 da arquitetura EJB este descritor está definido na forma de um arquivo XML (*Extensible Markup Language*), que tanto pode ser escrito pelo desenvolvedor diretamente, como pode ser gerado através do uso de ferramentas fornecidas pelos *containers* EJB. Quando o componente é disponibilizado em um *container*, este descritor será lido automaticamente pelo *container*, de modo que o *container* saiba como gerenciar o *bean* em tempo de execução.

4.1.2.6 Empacotamento

A finalização do desenvolvimento de um *bean* ocorre com o empacotamento do conjunto de arquivos implementados em um arquivo JAR (*Java Archive*). Os JAR são arquivos compactados similares aos arquivos ZIP. O empacotamento dos arquivos e geração do arquivo JAR pode ser realizado por uma ferramenta específica para este propósito. A Figura 5, adaptada de [ROM02], ilustra a definição dos arquivos JAR.

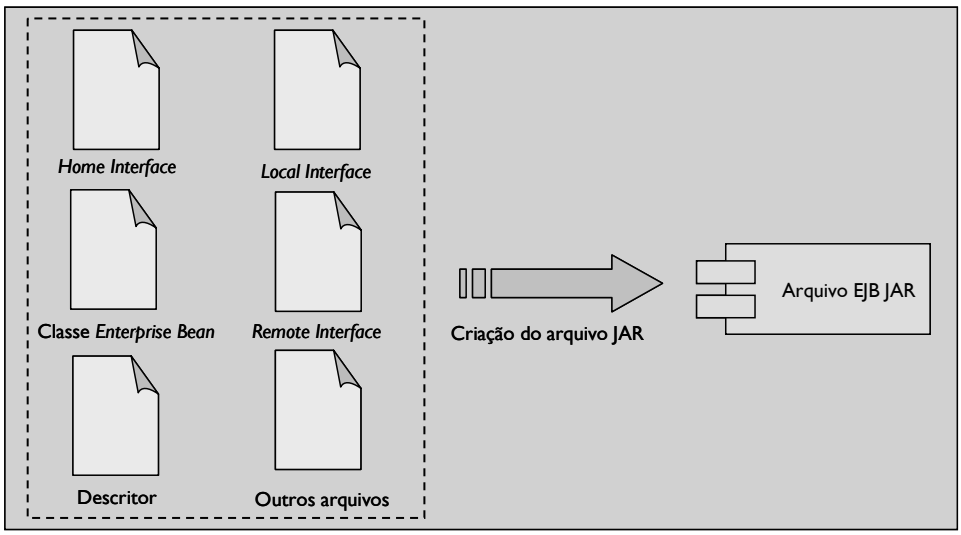

Figura 5 - Arquivos utilizados para gerar um componente EJB

Após o *bean* ter sido empacotado, a sua utilização ainda depende da disponibilização deste componente em um container EJB. O servidor/*container* EJB se responsabiliza por descompactar o arquivo e realizar a carga do *bean*, viabilizando a utilização deste *bean*.

4.1.3 Papéis relacionados à arquitetura EJB

O documento de especificação de EJB [DEM02], além do conjunto de definições relacionadas aos componentes EJB, também apresenta definições de papéis exercidos no desenvolvimento de aplicações com EJB. Com isso, EJB propõem a participação de seis diferentes papéis para o desenvolvimento e utilização das aplicações. A garantia de que o produto gerado por cada um dos papéis é compatível, é estabelecido pelas definições da arquitetura EJB, as quais devem ser respeitadas pelos diferentes papéis. A Tabela 4 apresenta os seis diferentes papéis propostos pela especificação EJB [DEM02], juntamente com uma breve caracterização quanto a suas responsabilidades e atividades.

Tabela 4 - Papéis definidos na arquitetura EJB

Papel	Descrição
Desenvolvedor de *bean* (*Enterprise Bean Provider*)	- Responsável por desenvolver os *enterprise beans*, tendo por função codificar as classes que implementam os métodos de negócio do *bean*. - Este desenvolvedor não se preocupa em programar serviços como transações, segurança e distribuição, pois delega ao *container* a responsabilidade por estas atividades. - Também pode exercer o papel de Desenvolvedor de aplicação.
Desenvolvedor de aplicação (*Application Assembler*)	- Papel responsável por combinar diferentes *beans*, ou *beans* e outros componentes, em grandes unidades de aplicação. Ou seja, tem por função desenvolver sistemas a partir dos componentes desenvolvidos pelo Desenvolvedor de *bean*. - Papel executado por um especialista do domínio, o qual define aplicações que usam *beans*.
Disponibilizador (*Deployer*)	- Este papel obtém os *beans* produzidos pelo Desenvolvedor de *bean* e combinados pelo Desenvolvedor de aplicação, disponibilizando-os em um ambiente operacional, que inclui servidor e *container* EJB. - Para executar este papel, o Disponibilizador utiliza ferramentas fornecidas pelo fabricante do servidor/*container* EJB.
Fabricante de servidor EJB (*EJB Server Provider*)	- Papel exercido geralmente por empresas fabricantes de outros tipos de sistemas, como banco de dados e sistemas operacionais. - A função deste papel é desenvolver os servidores EJB. - A atual arquitetura EJB assume que o fabricante do servidor e *container* é o mesmo.
Fabricante de *container* EJB (*EJB Container Provider*)	- Este papel deve prover as ferramentas necessárias que possibilitem a disponibilização dos *beans*, bem como o suporte em tempo de execução para a utilização das instâncias dos *beans*.
Administrador de sistema (*System Admistrator*)	- Este papel é responsável pela configuração e administração dos recursos computacionais e da infra-estrutura de rede, incluindo o servidor e *container* EJB. - A arquitetura EJB não define contratos para a gerência e administração do sistema.

4.1.4 Comparando *JavaBeans* e EJB

Embora a especificação de EJB o caracterize como uma arquitetura, considera-se que um subconjunto das definições dessa arquitetura caracterize como o modelo de componentes EJB. Desta forma, a Sun Microsystems além do modelo de componentes EJB, também apresenta a

definição de um segundo modelo, o *JavaBeans* (JB). Ambos os modelos são baseados na linguagem Java, e apesar de *JavaBeans* e Enterprise *JavaBeans* compartilharem o nome "*JavaBeans*" eles são dois modelos de componentes completamente diferentes, inclusive por apresentarem propósitos bastante distintos [ROM02][ROD99]. Embora muitos documentos já tenham identificado EJB como sendo uma extensão de JB com a inclusão de funcionalidades corporativas, conforme [ROD99] isso não passa de uma informação incorreta. Buscando identificar as diferenças entre estes dois modelos de componentes, a Tabela 5 apresenta a caracterização dos dois modelos segundo alguns aspectos discutidos em [RAJ98], [ROD99] e [ROM02].

Tabela 5 - Características de JB e EJB

Aspecto	JB	EJB
Componente	Componente que pode ser manipulado visualmente	Componente do lado servidor acessados remotamente
Tipos	Não apresenta (JB 1.0)	*Session bean, entity bean* e *message-driven bean*
Visibilidade	Podem ser objetos visuais ou não	Não são objetos visuais
Implementação	Classes Java que apresentam métodos `get`/`set`	Classes Java que implementam interfaces específicas (`javax.ejb`)
Aplicações desenvolvidas	Baseadas em eventos (lado cliente)	Distribuídas (lado servidor)
Persistência	Sim	*Stateful session bean*
Ambiente de execução	Não é necessário	Necessário

Enquanto o modelo JB caracteriza-se pela definição de componentes clientes, que podem potencialmente apresentar uma interface gráfica, embora não seja obrigatório, o modelo EJB define componentes do lado servidor que não apresentam interface gráfica. EJB representa um maior nível de complexidade em relação a JB, em virtude do tipo de componente que define. Os componentes JB podem ser implementados utilizando-se apenas a plataforma J2SE (*Java 2 Platform, Standard Edition*), enquanto os componentes EJB necessitam da plataforma J2EE, a qual tem por base a plataforma J2SE. Além disso, a utilização do componente EJB exige que este componente seja disponibilizado no servidor/*container* EJB, o qual fornece aos *beans* serviços como distribuição, persistência, segurança, entre outros. Em contrapartida, os componentes JB não precisam de um ambiente de execução especial.

Com isso é possível identificar que a similaridade entre estes dois modelos de componentes está apenas no fato de terem sido definidos pela Sun, compartilharem o nome "*JavaBeans*" e serem baseados na tecnologia Java, pois seus propósitos e estruturas são completamente distintos.

4.2 Caracterização da tecnologia EJB frente ao referencial teórico

Os *enterprise beans* apresentam-se como componentes de código, os quais implementam serviços e podem ser reutilizados. Embora a reutilização possa ocorrer, não existe nenhuma preocupação na arquitetura EJB no sentido de gerar uma documentação apropriada que favoreça o posterior reuso do *bean*. EJB caracteriza-se por apresentar três tipos distintos de componentes, os quais estão voltados para a disponibilização de diferentes tipos de serviços. A definição quanto aos tipos de *bean* existentes, a sua forma de interação e quais recursos estes componentes tem acesso é definido pelo modelo de componentes EJB.

Em contrapartida a literatura relacionada a DBC, que trata EJB como um modelo de componentes, a especificação oficial da tecnologia a caracteriza como uma arquitetura. Estas diferenças de nomenclatura acabam por dificultar a identificação na tecnologia dos conceitos descritos no referencial teórico. Embora seja possível visualizar as definições do modelo de componentes como parte das definições da arquitetura EJB, não se tem na especificação EJB definições explicitas quanto ao modelo e *framework* de componente.

Considerando o referencial teórico, os *frameworks* têm papel de reforçar e suportar as definições do modelo de componentes, bem como prover uma forma de interação dos componentes. A partir do estudo da tecnologia EJB é possível caracterizar as implementações de servidores/*container* como sendo os seus *frameworks* de componentes, os quais são disponibilizados pelos produtos comerciais e/ou de domínio público que implementam esta tecnologia (e.g. WebShere e JBoss). Desta forma, os *frameworks* de componentes EJB seguem as definições da arquitetura EJB (modelo de componentes) e implementam os serviços disponibilizados aos *beans* (e.g transações, segurança, persistência, entre outros).

Ainda considerando os conceitos de modelo e *framework* de componentes, é possível verificar que a tentativa de caracterizar a tecnologia EJB frente ao referencial teórico não é uma atividade trivial. A caracterização apresenta uma série de dificuldades, principalmente pelas diferenças de nomenclatura e por não existir uma correlação entre o definido no referencial com o estabelecido na tecnologia.

A relação dos *enterprise beans* com a orientação a objetos pode ser verificada através do uso da linguagem de programação Java na sua implementação. Os *beans* correspondem a um conjunto de classes e interfaces Java que são empacotadas sob a forma de um arquivo JAR e disponibilizadas em um servidor/*container* EJB.

No trabalho de [KOB00], apresentado no Capítulo 2, verificou-se um conjunto de deficiências da UML para a modelagem de aplicações baseadas em componentes, as quais se espera que sejam minimizadas na especificação UML 2.0. Complementar as definições para a modelagem de componentes existentes na UML, a modelagem de aplicações EJB pode contar ainda com a especificação UML *profile for* EJB [UML01], que identifica um conjunto padrão de extensões relacionadas às características da arquitetura EJB.

As atividades de adaptação e composição de componentes não são tratadas pela especificação da arquitetura EJB, entretanto, de uma maneira geral, se considera possível a realização das duas atividades com os componentes EJB, sendo necessário apenas respeitar as características e definições da tecnologia.

A Tabela 6 resume as considerações quanto à caracterização da tecnologia EJB frente as tópicos do referencial teórico discutidos nesta seção.

Tabela 6 - Caracterização de EJB frente ao referencial teórico

Referencial Teórico	Tecnologia EJB
Componente	componente de código empacotado sob a forma de um arquivo JAR
Reutilização	não apresenta preocupação com a reutilização
Modelo de componente	não faz referencia direta, embora parte das definições estabelecidas na arquitetura EJB possa ser vistas como o modelo de componentes EJB
***Framework* de componente**	não faz referencia direta, embora as implementações de servidor/*container* possam ser vistas como *framework* de componentes
Adaptação	não trata especificamente de questões relacionadas à adaptação. É possível a aplicação de diferentes técnicas, apenas devendo preocupação em selecionar uma técnica que não viole as definições da arquitetura EJB
Composição	a arquitetura EJB determina que a composição de componentes será sempre mediada pelo *container*. A partir disso ainda se pode considerar a possibilidade de ocorrência da composição *componente-componente*, *componente-framework* e *framework-framework*, esta última em especifico na utilização de *session* e *entity beans* para a comunicação de componentes EJB e CORBA, pelo uso do protocolo de distribuição RMI-IIOP
Modelagem	utilização da UML quanto à modelagem de componentes e emprego das extensões da UML *profile for* EJB
Orientação a objetos	implementação dos componentes EJB através do uso da linguagem de programação Java

5 CONSIDERAÇÕES FINAIS

A revisão bibliográfica sobre DBC apresentada neste relatório teve inicio no estudo de diferentes definições de componentes. No Capítulo 1 foram apresentadas e discutidas três definições que retratam duas diferentes visões de componentes. Conforme [SAM97], os componentes podem ser vistos como alguma parte do sistema de software que é identificável e reusável, ou como o estado seguinte de abstração depois de funções, módulos e classes. Foram apresentadas e discutidas as definições para componentes de [SAM97], [BRO97] e [SZY99], sendo elaborado um quadro comparativo com os elementos destas três definições.

As definições discutidas ilustram as duas visões de componentes, sendo possível observar que as definições de componentes que os consideram apenas como componentes de código representam o aspecto mais maduro de DBC atualmente, pois tratam da visão que tem recebido mais atenção. Apesar desta atual realidade, é esperado que DBC avance e os componentes passem a serem vistos segundo a definição apresentada por [SAM97], em que componente pode ser visto como alguma parte do sistema de software que é identificável e reusável, possibilitando se ter reuso em todas as fases de desenvolvimento de software, ao invés de unicamente na fase de implementação, como ocorre hoje em dia.

O Capítulo 1 ainda apresentou os conceitos de modelo e *framework* de componente, e qual o papel destes para os componentes. Como último tópico do Capítulo 1, investigou-se a relação de componentes e orientação a objetos, permitindo verificar que a tecnologia de objetos não é nem necessária, nem suficiente para DBC. Embora se tenha essa independência, nada impede que atuem em conjunto, inclusive esta combinação é uma das tendências exploradas. Finalizando o capítulo, foi apresentado o estudo de Kobryn [KOB00] quanto a utilização da notação padrão de modelagem de objetos, UML, para a modelagem dos *frameworks* de componentes..

A partir do estudo de componentes e aspectos relacionados, no Capítulo 2 foram abordadas as perspectivas de desenvolvimento de software relacionadas a DBC. Foram apresentadas possíveis atividades do processo de desenvolvimento *com* componentes, sendo discutidas detalhadamente as atividades de adaptação e composição de componentes. A adaptação é uma atividade de extrema importância, considerando que os componentes não são desenvolvidos prevendo todos os seus possíveis usos, e que para o reuso deste componente pode ser necessária alguma forma de adaptação. Igualmente, a atividade de composição também é

bastante importante, pois é ela que possibilita o desenvolvimento de uma aplicação pela integração de diferentes componentes.

Como terceira etapa deste trabalho, o Capítulo 4 apresentou um estudo da tecnologia de componentes EJB. Além da apresentação da tecnologia e de seus principais conceitos, EJB também foi caracterizado frente aos tópicos discutidos nos capítulos anteriores deste relatório, o que possibilitou contextualizar o referencial teórico abordado no trabalho e verificar a dificuldade em identificar os conceitos da literatura nas tecnologias de componentes existentes.

Em contrapartida a existência de inúmeras motivações para a abordagem de DBC, também existe inúmeras questões em aberto. Este relatório buscou apresentar uma revisão bibliográfica que possibilitasse uma contextualização inicial de DBC. Sendo assim, além de caracterizar-se como um primeiro estudo sobre DBC, este relatório técnico é um ponto de partida para a continuidade das pesquisas nesta área no âmbito do Programa de Pós-Graduação em Ciência da Computação da PUCRS.

REFERÊNCIAS BIBLIOGRÁFICAS

[BAC02] BACHMANN, Felix *et al*. Volume II: Technical Concepts of Component-Based Software Engineering – 2nd Edition. On-line. Capturado em Out. 2002. Disponível na Internet http://www.sei.cmu.edu/pub/documents/00.reports/pdf/ 00tr008.pdf

[BOD02] BODOFF, Stephanie *et al*. **The J2EE Tutorial**. Boston: Addison-Wesley, 2002. 491p.

[BOS99] BOSCH, Jan. Superimposition: a component adaptation technique. **Information and Software Technology**, v.41, n.5, p.257-273, Mar. 1999.

[BRO97] BROWN, Alan W.; Short, Keith. On Components and Objects: The Foundation of Component-Based Development. In: INTERNATIONAL SYMPOSIUM ON ASSESSMENT OF SOFTWARE TOOLS AND TECHNOLOGIES (SAST 97), 5, 1997, Pittsburgh, PA. **Proceedings...** Pittsburgh, PA: IEEE Press, 1997. p.112-121.

[BRO98] BROWN, Alan W.; WALLNAU, Kurt C. The Current State of CBSE. **IEEE Software**, v.15, n.5, p.37-46, Sep. 1998.

[BRO00] BROWN, Alan W. Large-scale component-based development. Upper Saddle River, NJ: Prentice Hall, 2000. 286p.

[CRN02] CRNKOVIC, Ivica; LARSSON, Magnus. **Building reliable component-based software systems**. Norwood, MA: Artech House, 2002. 413p.

[CHE01] CHEESMAN, John; DANIELS, John. **UML Components**: a simple process for specifying component-based software. Boston: Addison-Wesley, 2001. 176p.

[DEM02] DEMICHIEL, Linda G. Enterprise JavaBeans Specification, Version 2.1. On-line. Capturado em Set. 2002. Disponível na Internet http://java.sun.com/products/ejb/docs.html

[DSO98] D'SOUZA, Desmond Francis; WILLS, Alan Cameron. **Objects, Components and Frameworks with UML**: The Catalysis Approach. Reading: Addison-Wesley, 1998. 785p.

[GAM00] GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. **Padrões de Projeto**. Porto Alegre: Bookman, 2000. 364p.

[HEI99a] HEINEMAN, George T. An Evaluation of Component Adaptation Techniques. In: INTERNATIONAL WORKSHOP ON COMPONENT-BASED SOFTWARE ENGINEERING, 2, 1999, Los Angeles. On-line. Capturado em Jan. 2002. Disponível na Internet http://www.sei.cmu.edu/cbs/icse99/papers/13/13.htm

[HEI99b] HEINEMAN, George T; OHLENBUSCH, Helgo M. An Evaluation of Component Adaptation Techniques (Technical Report). On-line. Capturado em Abr. 2002.

Disponível na Internet http://www.cs.wpi.edu/~heineman/classes/cs562/pdf/HO99.pdf

[HEI01] HEINEMAN, George T.; COUNCILL, William T. **Component-Based Software Engineering**: putting the pieces together. New York: Addison-Wesley, 2001. 818p.

[KOB00] KOBRYN, Cris. Modeling components and frameworks with UML. **Communications of the ACM**, v.43, n.10, p.31-38, Oct. 2000.

[KOB02] KOBRYN, Cris. Will UML 2.0 be a agile or awkward?. **Communications of the ACM**, v.45, n.1, p.107-110, Jan. 2002.

[MON00] MONSON-Haefel, Richard; ROHALY, Tim. Enterprise JavaBeans Technology Fundamentals. On-line. Capturado em Jul. 2002. Disponível na Internet http://developer.java.sun.com/developer/onlineTraining/EJBIntro/EJBIntro.html

[MON01] MONSON-Haefel, Richard. **Enterprise JavaBeans**. Sebastopol, CA: O'Reilly, 2001. 567p.

[PRE01] PRESSMAN, Roger S. **Software Engineering**: a practioner's approach. New York: McGraw-Hill, 2001. 860p.

[RAJ98] RAJ, Gopalan Suresh. Enterprise JavaBeans. On-line. Capturado em Out. 2002. Disponível na Internet http://my.execpc.com/~gopalan/java/ejb.html

[ROD99] RODRIGUES, Lawrence; RAJ, Gopalan Suresh. JavaBeans vs Enterprise JavaBeans. **Java Developer's Journal**, v.4, n.8, p.22-30, Aug. 1999.

[ROM02] ROMAN, Ed; AMBLER, Scott; JEWELL, Tyler. **Mastering Enterprise JavaBeans**. 2.ed. New York: John Wiley & Sons, 2002. 639p.

[SAM97] SAMETINGER, Johannes. **Software Engineering with Reusable Components**. New York: Springer, 1997. 271p.

[SZY99] SZYPERSKI, Clemens. **Component Software**: beyond object-oriented programming. Harlow: Addison-Wesley, 1999. 411p.

[UML01] UML Profile For EJB. On-line. Capturado em jul. 2002. Disponível da Internet http://jcp.org/aboutJava/communityprocess/review/jsr026/

[WEI01] WEISS, Gerson Mizuta. Adaptação de componentes de software para o desenvolvimento de sistemas confiáveis. São Paulo, 2001. Dissertação (Mestrado em Ciência da Computação) – Instituto de Computação, UNICAMP, 2001.

[WER00] WERNER, Claudia M. L.; BRAGA, Regina M. M. Desenvolvimento Baseado em Componentes. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE – MINICURSOS/TUTORIAIS, 14, 2000, João Pessoa. **Anais...** João Pessoa: UFRJ, CEFET-PB, 2000. p. 297-329.