

Componentes Distribuídos

■ Componentes de Software

- Vantagens e Limitações
- Modelos de Componentes
- Interfaces
- Contêineres

- Comunicação
- Desenvolvimento
- Distribuição e Implantação
- Padrões de Mercado

■ Componentes em Java

- JavaBeans
- Java EE

- EJB
- JPA

■ Componentes CORBA

Componentes de Software

- Histórico - Anos 60-70
 - Crise do Software
 - Linguagens Procedurais
 - Métodos e técnicas de desenvolvimento de software não eram adotados
 - Poucos fornecedores de software no mercado



Componentes de Software

- Histórico - Anos 80-90
 - Métodos e técnicas de Engenharia de Software
 - Linguagens Orientadas a Objetos
 - Mercado dinâmico e competitivo
 - Continuidade da crise do software



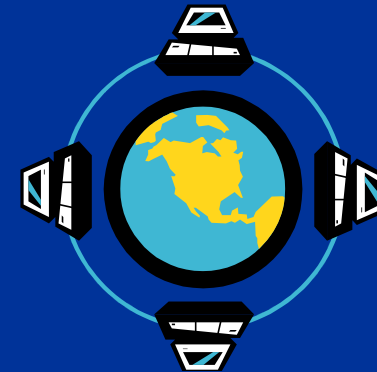
Componentes de Software

■ Panorama Atual



Aplicações são cada vez mais complexas

Necessidade de comunicação pela rede



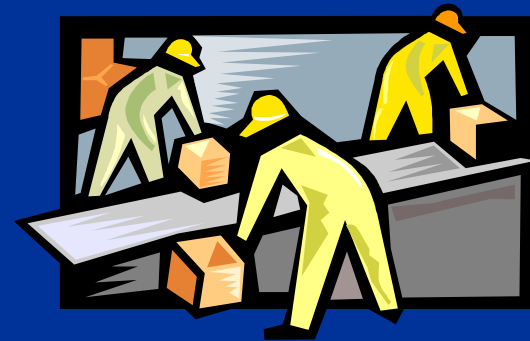
Menos tempo para desenvolvimento

Maior qualidade com custo menor



Componentes de Software

- Outras indústrias passaram por crises similares à que hoje enfrenta a indústria de software
- Na maioria das indústrias, o uso de componentes traz ganhos significativos
 - Indústria eletrônica, automobilística, de máquinas e equipamentos, construção civil, etc.
- Por que não utilizar componentes também para construir software?



Componentes de Software

■ Definições

■ Componente

"Aquilo que entra na composição de alguma coisa. Parte elementar de um sistema."

(Aurélio)

■ Composição

"Formar ou construir de diferentes partes."

(Aurélio)



Componentes de Software

- Definição de Componente de Software
 - *“Unidade de Software independente que encapsula, dentro de si, seu projeto e implementação, e oferece serviços, por meio de interfaces bem definidas, para o meio externo”* (Fonte: I. Gimenes & E. Huzita)
 - *“Unidade de software com interfaces e dependências claramente especificadas, que pode ser implantada independentemente e ser utilizada por terceiros para composição.”* (Fonte: C. Szyperski)

Componentes de Software

- Principais Características
 - Unidades funcionais de software
 - Executam uma atividade específica
 - Interação através de interfaces bem definidas
 - Produzidos e implantados independentemente
 - Distribuídos em código binário
 - Combinados para compor aplicações

Componentes de Software

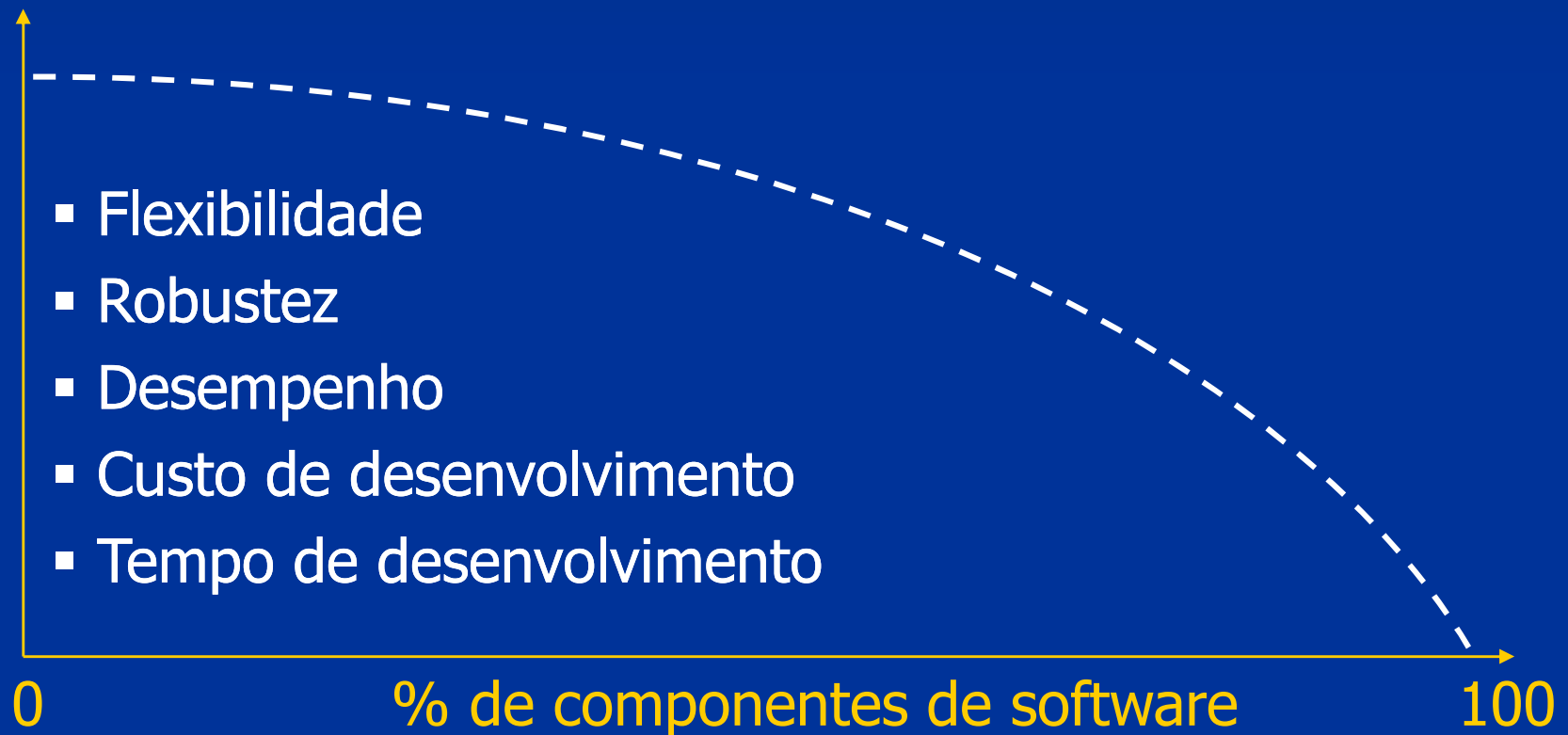
- Características Adicionais
 - Em geral são desenvolvidos usando os métodos, técnicas, linguagens e ferramentas tradicionais
 - Parte do processo de desenvolvimento pode ser automatizada
 - Configurados de acordo com a necessidade do usuário
 - Podem ser facilmente atualizados e reutilizados em outras aplicações

Componentes de Software

- Perspectivas de Mercado
 - A tecnologia de componentes ainda está amadurecendo
 - Com isso os componentes ainda são usados em poucas aplicações
 - Interfaces gráficas
 - Processadores de texto e planilhas
 - Navegadores e servidores Web (plug-ins)
 - Aplicações de comércio eletrônico
 - Integração com BDs e sistemas legados

Vantagens e Limitações

■ Utilização de componentes de software



Vantagens e Limitações

- Desenvolvimento de Sistemas
 - A complexidade necessária para executar uma tarefa é encapsulada por um componente
 - O sistema pode ser construído a partir de componentes reutilizados ou fornecidos por terceiros
 - Componentes são testados individualmente, melhorando a confiabilidade da aplicação
- Custo
 - Diluído entre os usuários do componente
 - Maior competitividade no mercado

Vantagens e Limitações

- Reutilização de Componentes
 - Outras unidades de software podem ser reutilizadas
 - Ex.: Classes, Bibliotecas, Pacotes, ...
 - No entanto, estas unidades geralmente são:
 - Dependentes de linguagem/ambiente
 - De granularidade inadequada
 - Difíceis de combinar
 - Pouco flexíveis

Vantagens e Limitações

- Manutenção de Software
 - Componentes podem ser substituídos e atualizados independentemente do restante da aplicação
 - Manutenção é feita pelo fornecedor do componente

Vantagens e Limitações

- Limitações no Desenvolvimento
 - Excesso de generalização pode levar a ineficiência
 - Nem sempre é possível configurar um componente para atender a todas as necessidades do usuário

Vantagens e Limitações

- Compatibilidade entre Componentes
 - Interfaces e/ou modelos de componentes podem ser incompatíveis, impedindo a sua composição
 - É raro encontrar componentes de fabricantes diferentes que sejam intercambiáveis
 - Mecanismos para interação entre componentes precisam ser padronizados

Modelos de Componentes

- Um **modelo de componentes** define:
 - A forma como componentes são vistos externamente
 - Aspectos da estrutura interna dos componentes
 - Os serviços que o suporte de execução deve fornecer aos componentes
 - O mecanismo usado para comunicação entre os componentes
 - O processo de desenvolvimento, de distribuição e de implantação de componentes

Modelos de Componentes

- Componentes fornecem **serviços** a terceiros através de suas **interfaces**
 - Um serviço é uma atividade bem definida executada pelo componente
 - A interface de um componente define como terceiros podem solicitar os serviços do componente
- Instâncias de um mesmo componente são gerenciadas por um componente **Home**
 - Gerencia o ciclo de vida e localiza instâncias

Modelo de Componentes

- Componentes podem ter **dados de estado**
 - *Stateless* : componentes sem estado
 - *Stateful* : componentes com estado
- Dados de estado podem ser:
 - Internos: uso privado do componente
 - Externos: obtidos através da interface
- O estado do componente pode ser:
 - Volátil: perdido ao reiniciar o componente
 - Persistente: mantido em memória persistente (HD, BD, etc.) e restaurado ao reiniciar

Modelos de Componentes

- Componentes podem interagir com o **suporte de execução** – o servidor de aplicação – através de:
 - ***Containers*** : para usar serviços do suporte de execução
 - ***Interfaces de callback*** : para receber chamadas do suporte de execução (*upcalls*)

Modelos de Componentes

- Os **mecanismos de comunicação** (local ou remota) devem ser padronizados para que componentes possam interagir
- O processo de **desenvolvimento, distribuição e implantação** de componentes segue regras bem definidas, com base em:
 - Metodologias de desenvolvimento
 - Compiladores e geradores de código
 - Servidores de componentes
 - Mecanismos de distribuição
 - etc.

Interfaces

- Os serviços fornecidos pelo componente são disponibilizados através de uma ou mais interfaces claramente definidas
- Na interface do componente são descritos:
 - Propriedades: as características configuráveis dos componentes, que serão levadas em conta durante a sua execução
 - Métodos: rotinas chamadas por terceiros para solicitar os serviços fornecidos pelo componente

Interfaces

- Outras informações podem constar da interface do componente
 - Número de versão e de série
 - Linguagem de programação
 - Dependências de outros componentes/serviços
 - Mecanismos de comunicação/invocação
 - etc.
- Conhecendo a interface de um componente, é possível utilizá-lo para compor aplicações

Interfaces

- Descrição das interfaces de componentes
 - Na linguagem de programação, nos modelos que utilizam somente uma linguagem
 - Em uma linguagem de descrição de interface (IDL), nos modelos multi-linguagem
- De posse da descrição da interface, é possível gerar o código não-funcional
 - Utilização dos mecanismos de comunicação
 - Interação com serviços e com o suporte de execução
 - etc.

Interfaces

- Exemplo de Interface em Java

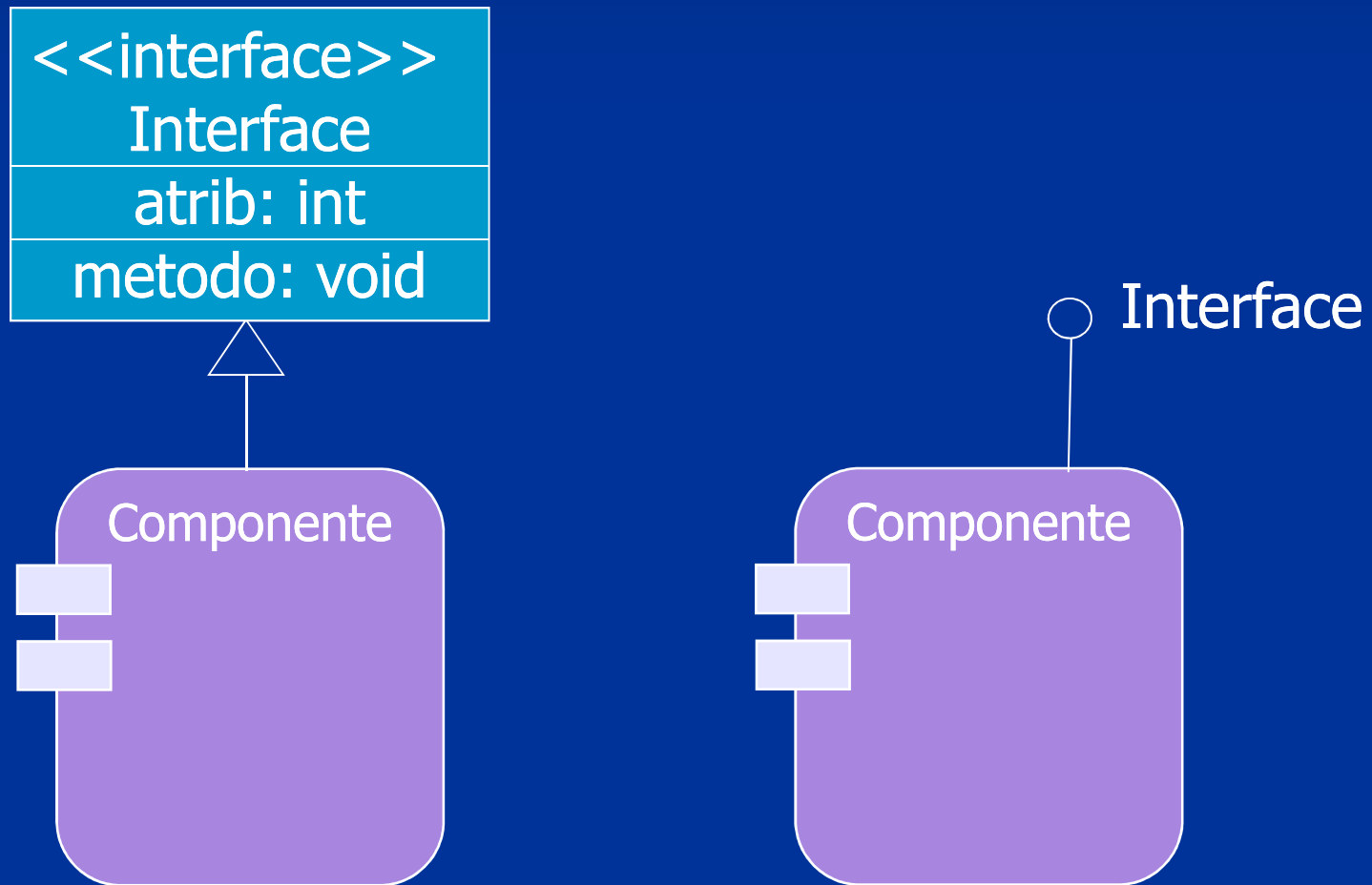
```
public interface Supermercado {  
    public float getPreco (long codigo);  
    public long getCodigo (String nome);  
    public String getNome (long codigo);  
}
```

- Exemplo de Interface em IDL

```
interface Supermercado {  
    float getPreco (in long codigo);  
    long getCodigo (in string nome);  
    string getNome (in long codigo);  
}
```

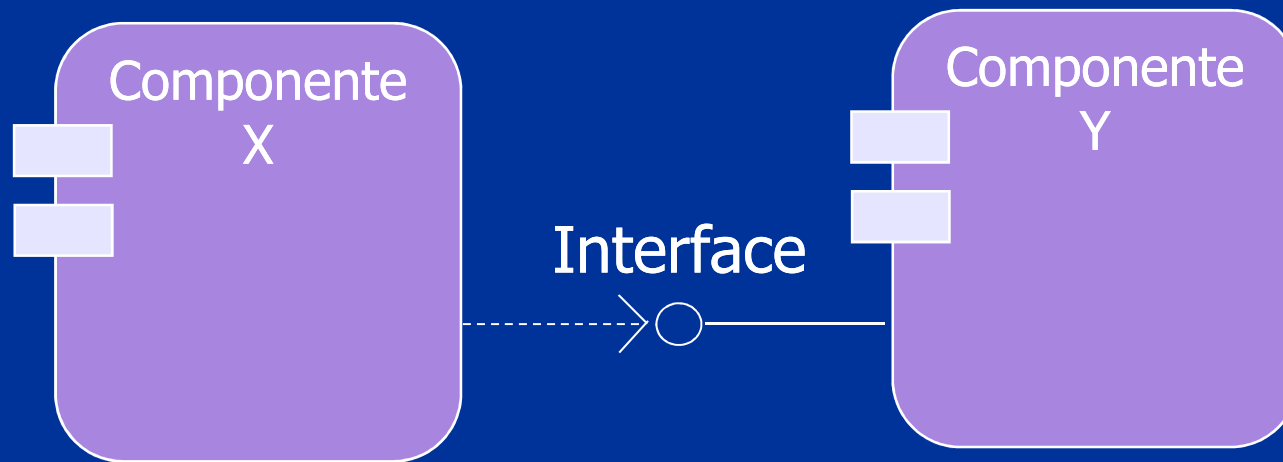
Interfaces

- Representação em UML de interfaces



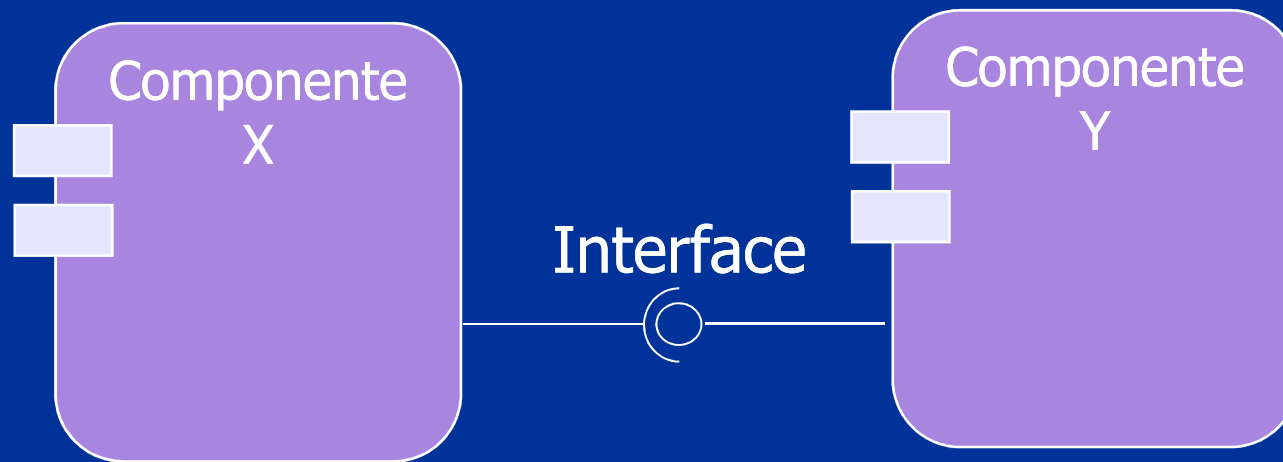
Interfaces

- Representação em UML 1.x da interconexão entre componentes
 - Componente X usa a Interface de Y



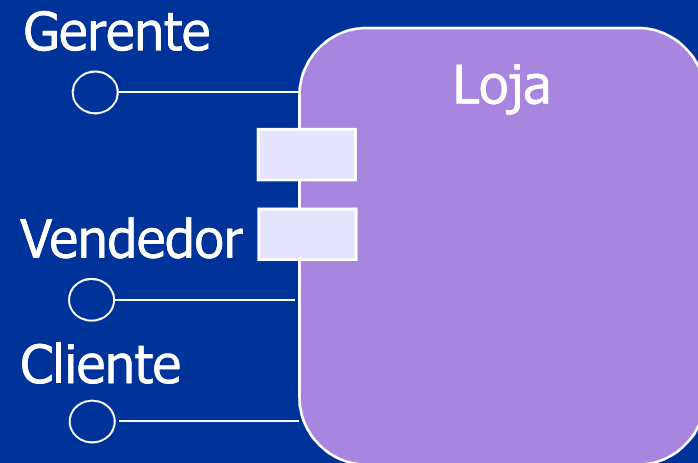
Interfaces

- Representação em UML 2.0 da interconexão entre componentes
 - Componente X usa a Interface de Y



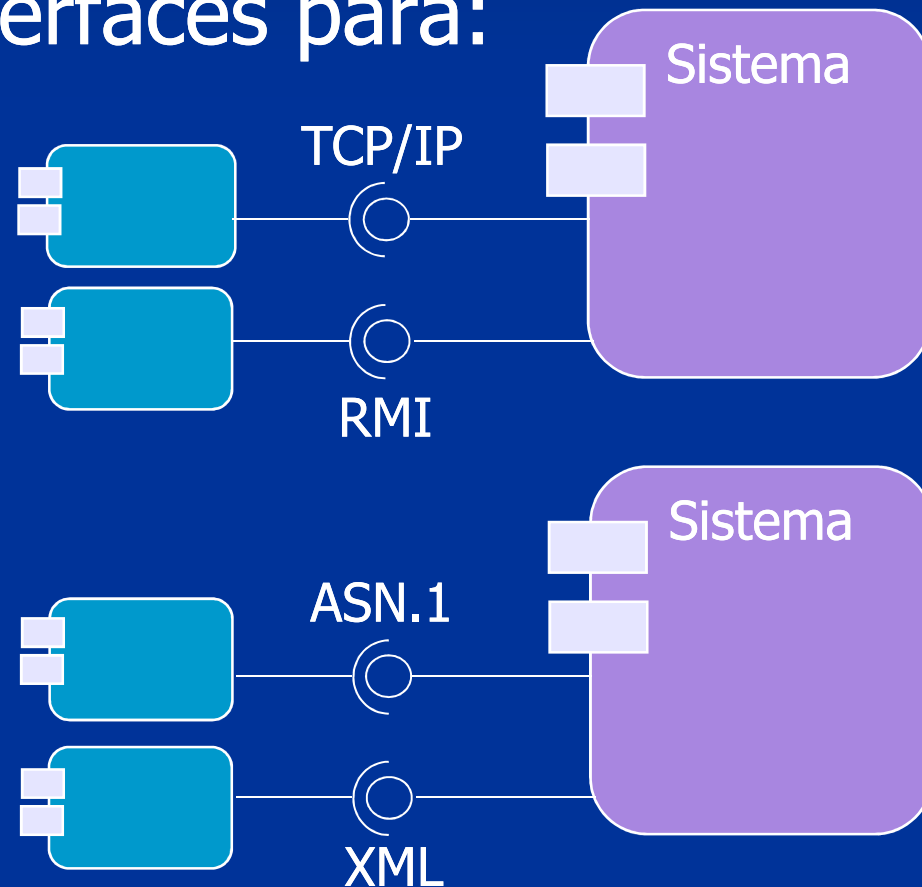
Interfaces

- Várias interfaces podem ser suportadas por um mesmo componente
- Interfaces diferentes podem ser fornecidas para cada tipo de usuário



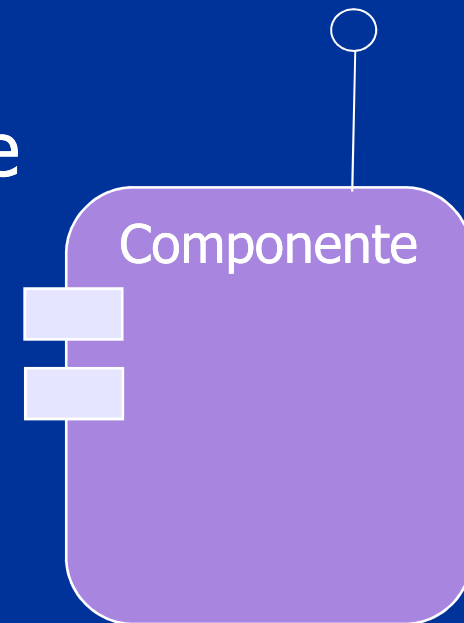
Interfaces

- Um componente também pode ter diferentes interfaces para:
 - Suportar diferentes protocolos de comunicação
 - Se comunicar usando vários formatos de dados
 - etc.



Interfaces

- Uma interface padrão suportada por todos os componentes permite inspecionar suas características internas
 - Descobrir a(s) interface(s) suportada(s) pelo componente
 - Obter informações como número de versão, de série, chave pública, etc.
 - Descobrir o(s) mecanismo(s) de comunicação aceito(s) pelo componente

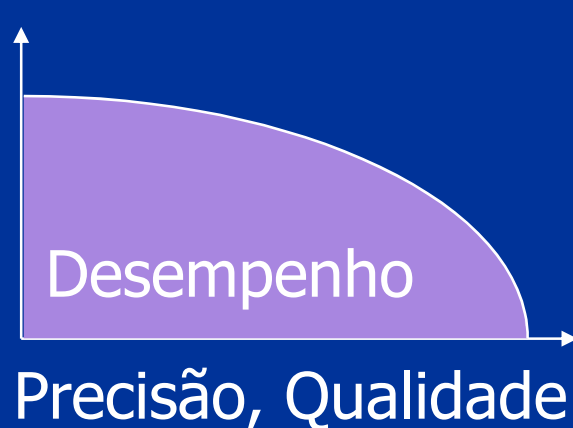


Interfaces

- Polimorfismo: uma mesma interface pode ser implementada por dois ou mais componentes de maneiras diferentes
- Se dois ou mais componentes implementam uma mesma interface:
 - Fornecem os mesmos serviços
 - São intercambiáveis
- Polimorfismo permite que o desenvolvedor escolha qual componente é mais adequado para a aplicação que está construindo

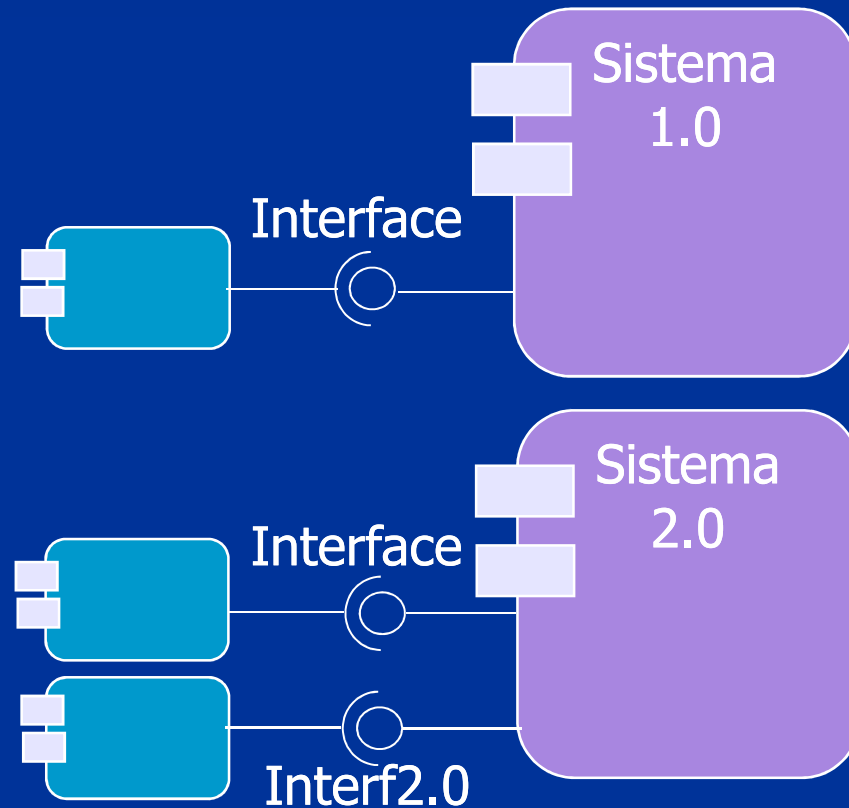
Interfaces

- Escolha de componentes polimórficos
 - O desenvolvedor deve considerar fatores como precisão, qualidade, desempenho e custo dos componentes polimórficos ao fazer a escolha entre os componentes existentes
 - De modo geral, a relação entre estas grandezas é a seguinte:



Interfaces

- Polimorfismo também permite manter o suporte a versões 'legadas' do componente
 - Nova versão do componente suporta a interface antiga e a nova
 - Nova versão pode incorporar novos serviços através de uma nova interface

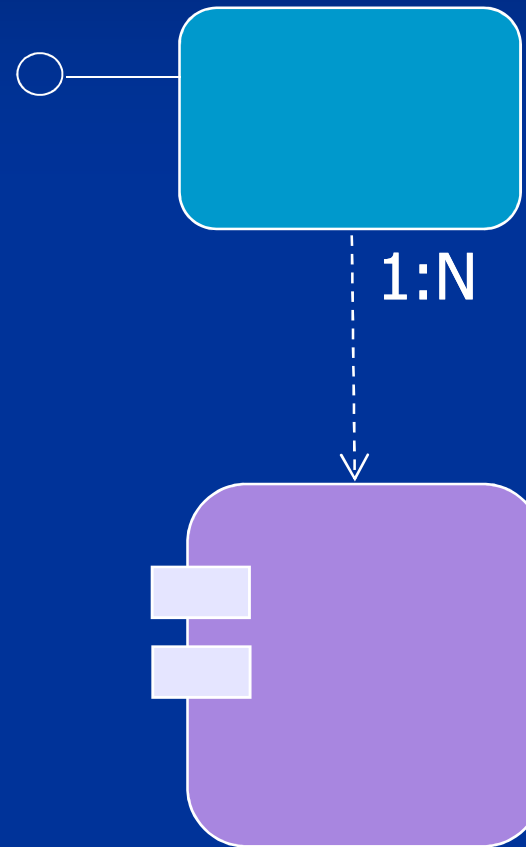


Interfaces

- Interfaces de gerenciamento chamadas Homes são utilizadas em alguns modelos de componentes
 - Usada para criar novas instâncias e para localizar instâncias já existentes de um determinado tipo de componente
 - Funciona como a parte estática de uma classe, ou seja, reúne as informações que independem de instância

Interfaces

- Uma *Home* administra várias instâncias de um mesmo tipo de componente
- Sua interface é padronizada pelo modelo de componentes, mas pode ser estendida pelo desenvolvedor



Contêiners

- Um Contêiner é um ambiente de execução para componentes
 - Faz a ligação entre os componentes e o mundo exterior
 - Recebe pedidos de execução de serviços e os repassa ao componente, que executa o serviço
 - Evita que o componente tenha que interagir com o sistema operacional, o suporte de comunicação e com os serviços de aplicação
 - Permite que componente seja independente do ambiente de execução, tornando-o mais portátil e mais fácil de reutilizar

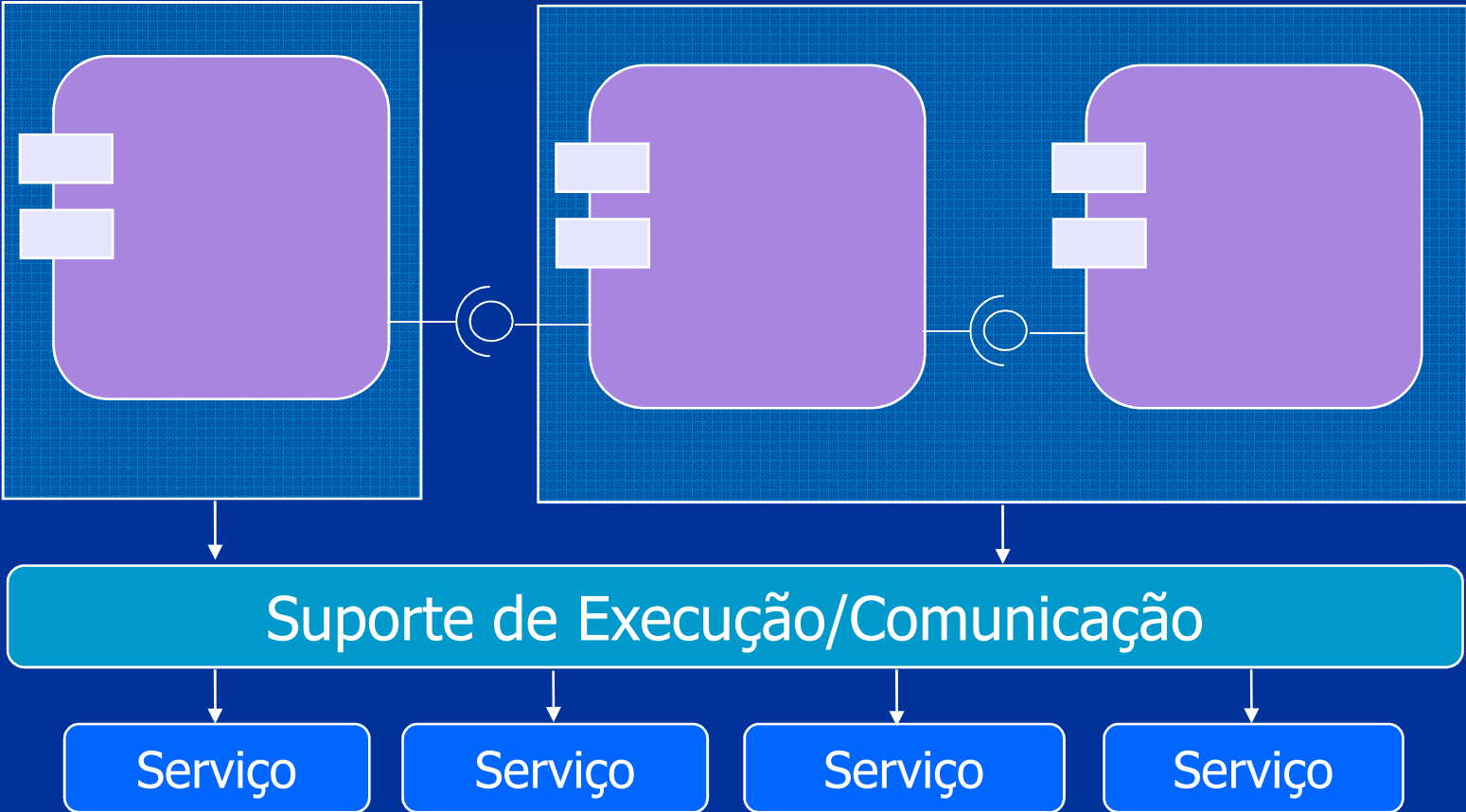
Contêiners

- As interfaces do Contêiners são definidas pelo modelo de componentes
 - Podem ser vistas como uma API completa para execução de componentes
 - Tornam o código do componente mais portátil
- Podem haver diferentes tipos de Contêiner
 - Exemplo: Contêiners para componentes sem estado, com estado transiente (volátil) ou persistente

Contêiners

- Contêiners utilizam recursos de software e hardware e serviços da plataforma de execução para executar o componente
 - Serviço de Nomes: permite localizar instâncias de componentes
 - Serviço de Comunicação: troca de informação
 - Serviço de Persistência: faz o armazenamento de estado dos componentes
 - Serviço de Transações: mantém a consistência
 - Serviço de Segurança: autentica componentes e verifica a autorização para executar serviços

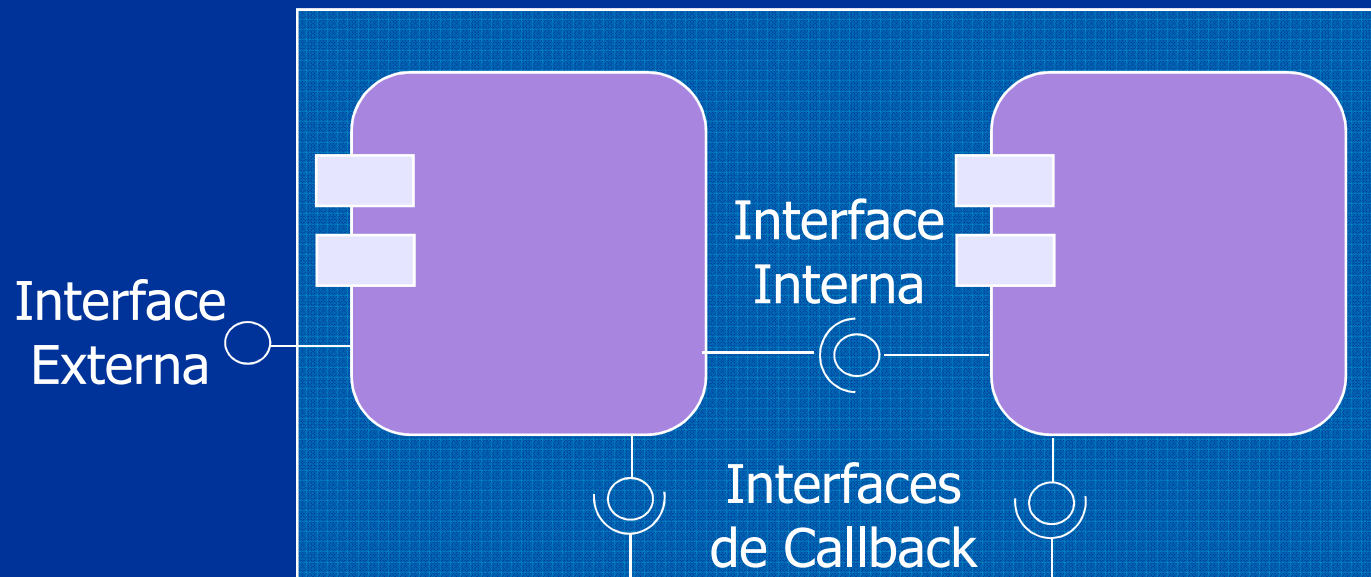
Contêineres



Contêiners

■ Tipos de Interfaces

- Externa: atravessa o contêiner
- Interna: acessível somente dentro do contêiner
- De Callback: interface usada pelo contêiner para se comunicar com o componente



Contêiners

- O contêiner efetua *Callbacks* para:
 - Indicar falhas na obtenção ou na utilização de recursos do suporte de execução
 - Entregar mensagens do serviço de comunicação
 - Salvar o estado do componente e restaurá-lo em caso de reinicialização
 - Relatar a violação de regras de funcionamento ou de segurança

Comunicação



- Componentes interagem para executar tarefas cooperativamente
- Componentes podem estar localizados em:
 - Processos diferentes
 - Máquinas diferentes
 - Redes diferentes
- Deve ser usado um protocolo padrão para comunicação entre componentes

Comunicação

- Mecanismos de comunicação de alto nível são necessários para simplificar o desenvolvimento
 - Chamada Remota de Procedimento (RPC)
 - Chamada Remota de Método (RMI)
 - Notificação de Eventos

Desenvolvimento

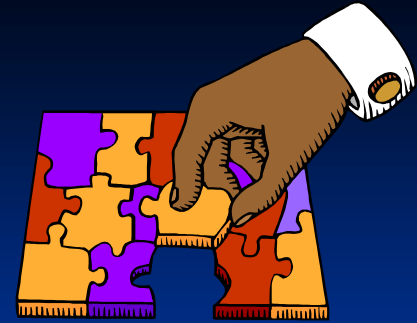


- O processo de desenvolvimento de componentes segue regras bem definidas, podendo utilizar-se de:
 - Metodologias de desenvolvimento
 - Compiladores e geradores de código
- Fases do desenvolvimento
 - Métodos tradicionais de engenharia de software podem ser usados na análise e projeto
 - São usadas linguagens de programação tradicionais na implementação
 - Devem ser aplicados testes do tipo *blackbox*

Desenvolvimento

- O desenvolvimento de componentes deve levar em conta que:
 - Um componente deve fornecer serviços a terceiros
 - Um componente deve procurar ser independente de outros componentes
 - Um componente é sujeito a composição

Desenvolvimento



■ Composição

- Permite que aplicações ou novos componentes sejam criados a partir de outros componentes
- Composição \neq Herança
 - Herança: usada para estender funcionalidade
 - X é um Y
 - Composição: usada para incorporar a funcionalidade
 - X tem um Y

Desenvolvimento

- Composição é mais adequada que herança para reutilizar código
 - Geralmente não queremos estender o componente, e sim incorporar a sua funcionalidade a uma aplicação



Desenvolvimento

- Composição x Herança

- Exemplos:

- Interface gráfica é um Botão + Rótulo + ...
?? ou ??

- Interface gráfica tem um Botão, Rótulo, ...

- Processador de texto é um corretor ortográfico

- ?? ou ??

- Processador de texto tem um corretor ortográfico

Desenvolvimento

- Composição de componentes pode ser feita através de:
 - Código de programas
 - Linguagens de composição
 - Ferramentas gráficas de composição
- Linguagens e ferramentas de composição devem:
 - Observar as dependências dos componentes
 - Verificar se as interfaces dos componentes são compatíveis

Desenvolvimento

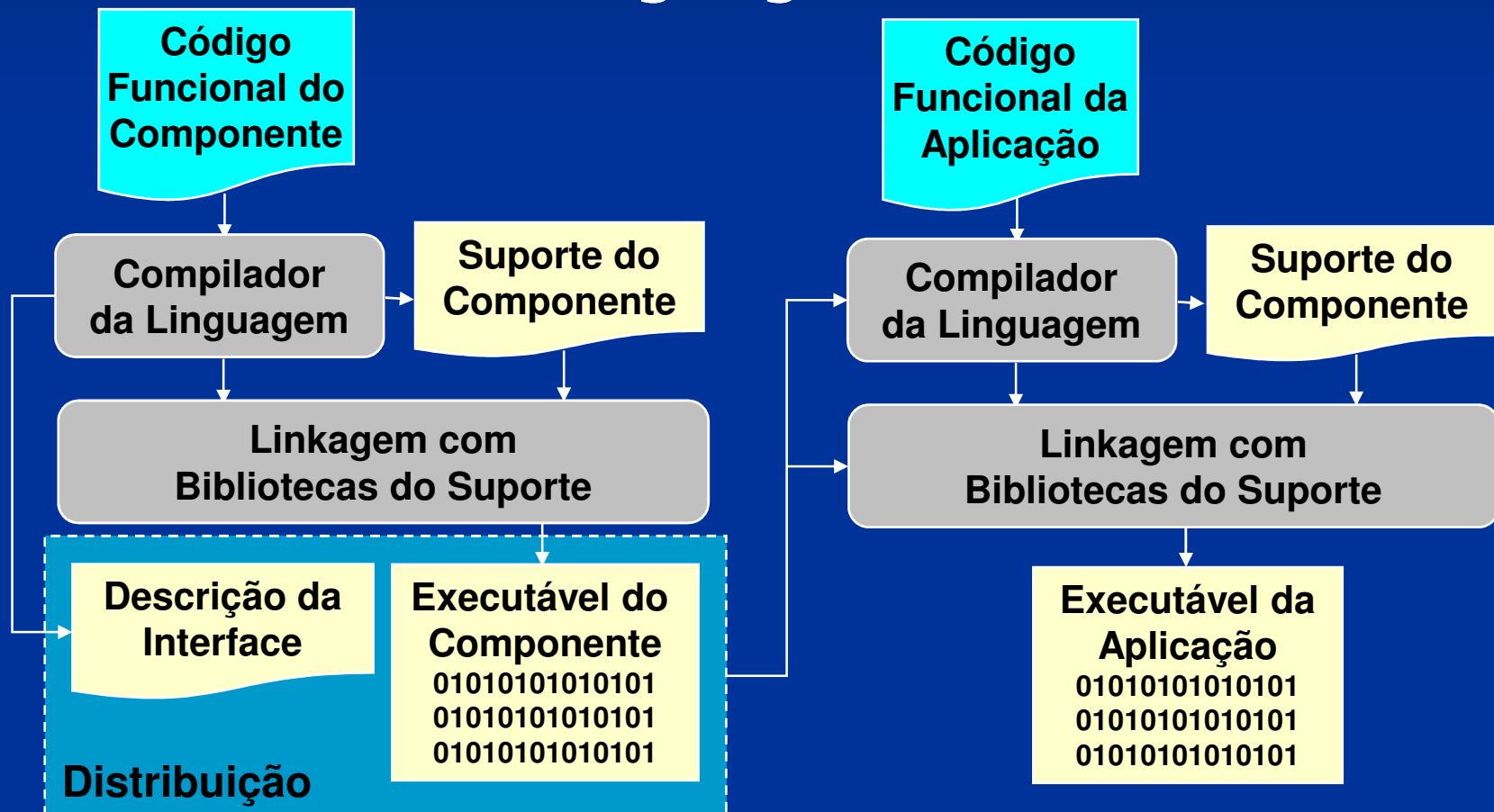
- Em geral, ambientes de desenvolvimento fornecem ferramentas para geração de código
 - Código não-funcional é gerado automaticamente
 - Código funcional deve ser escrito pelo programador

Desenvolvimento

- O processo de desenvolvimento varia conforme o modelo de componentes
 - Modelos multi-linguagem
 - Permitem que componentes escritos em linguagens diferentes interajam
 - Desenvolvimento é mais complexo
 - Modelos mono-linguagem
 - Não permitem interoperabilidade entre linguagens
 - Desenvolvimento é mais simples

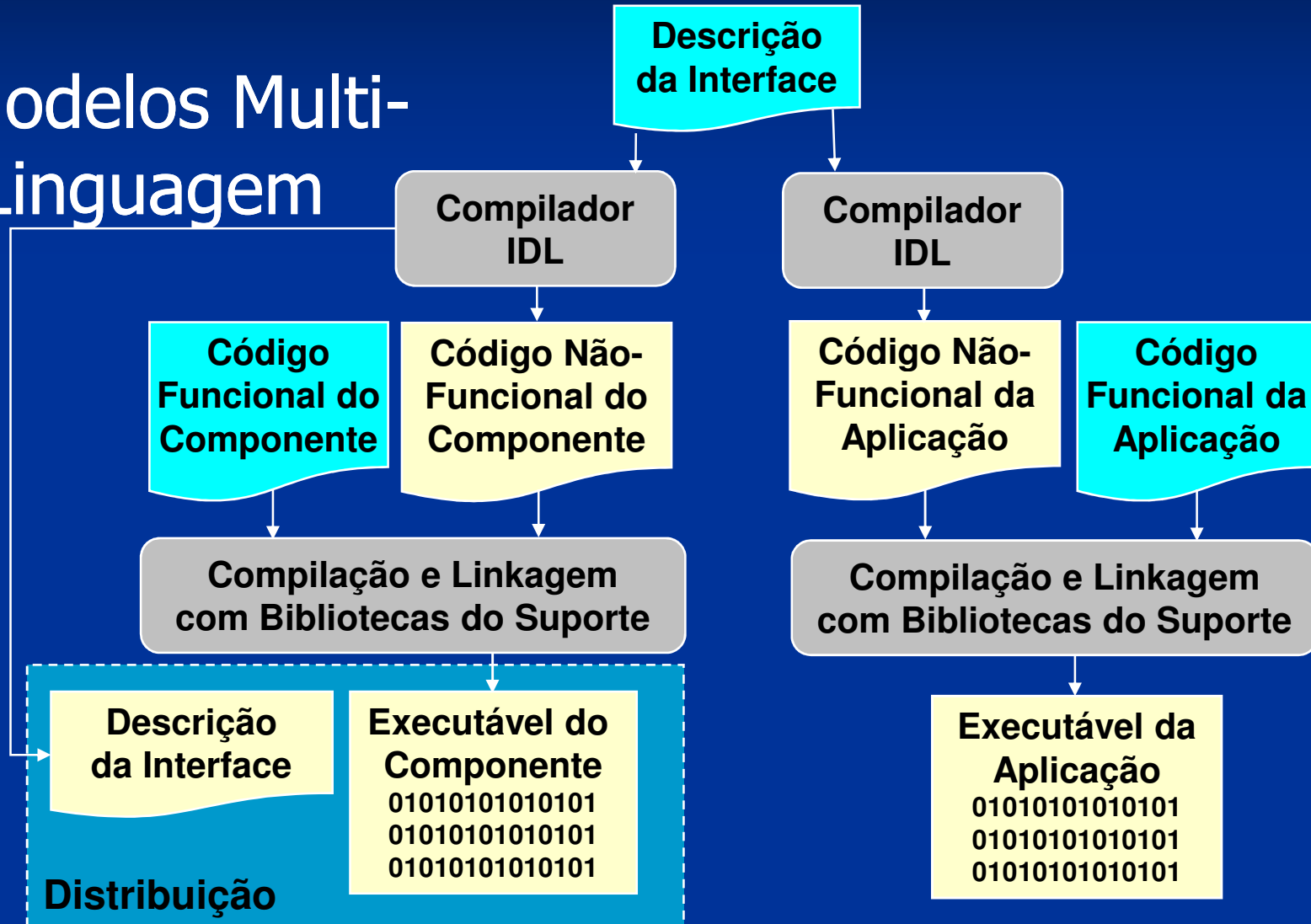
Desenvolvimento

■ Modelos Mono-Linguagem



Desenvolvimento

■ Modelos Multi-Linguagem



Distribuição e Implantação

- Componentes são distribuídos na forma de pacotes de distribuição (*packages*)
 - Geralmente são compactados (.ZIP, .JAR)
 - Pacotes de distribuição podem conter:
 - O código binário do(s) componente(s)
 - Apenas um componente
 - Um *framework* de componentes
 - Uma aplicação completa
 - Descritores de componentes



Distribuição e Implantação

- Em geral, o código binário é dependente da plataforma de execução
 - Neste caso, pode-se gerar executáveis para cada plataforma suportada e colocá-los no mesmo pacote
 - Na implantação, deve ser escolhida a versão do componente compatível com a plataforma
- Algumas linguagens são multi-plataforma
 - Máquina virtual converte suas instruções para instruções da máquina local
 - Exemplos: JVM (*Java*), CLR (.NET)

Distribuição e Implantação

- Descritores de componentes
 - Usados para descobrir os serviços fornecidos pelo componente e a sua forma de obtenção
 - Informam versão, nº de série e dependências
 - Especificam como instanciar, configurar e conectar os componentes
 - Tendência do mercado é migrar os descritores para uma linguagem padrão, como XML OSD (*Open Software Description*)

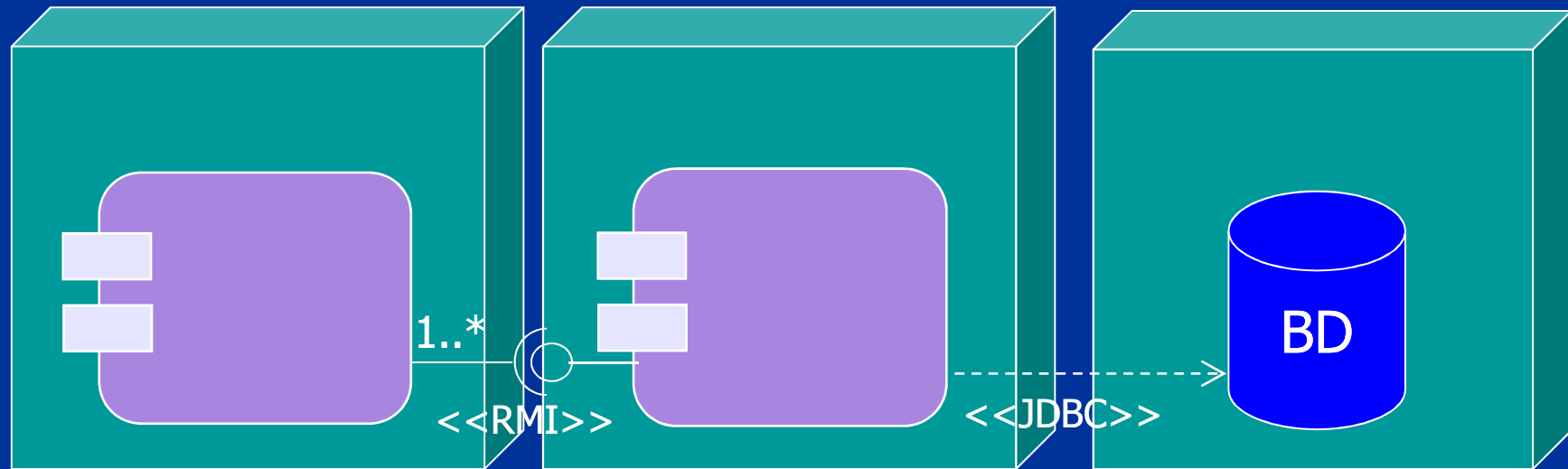
Distribuição e Implantação

- Componentes são implantados em servidores de aplicação
 - Em geral são servidores Web modificados para executar componentes
 - Pacotes são instalados no servidor
 - Servidor lê o descritor do componente e seleciona a implementação adequada
 - A partir da implantação, o componente está pronto para ser utilizado



Distribuição e Implantação

- Para implantar uma aplicação construída a partir de componentes é importante definir onde cada componente será implantado
- Diagrama de implantação da UML



Distribuição e Implantação

- Servidores de aplicação devem prover:
 - Interface para localização dos componentes
 - Suporte para comunicação
 - Serviços adicionais de propósito geral
 - Mecanismos para gerenciamento de aplicações
- Há vários serv. de aplicação no mercado:
 - Borland AppServer
 - Oracle Application Server
 - WebSphere – IBM
 - JBoss (gratuito)
 - etc.

Padrões de Mercado

- Padronização é necessária para que os componentes sejam compatíveis entre si
- Vários padrões podem coexistir, desde que:
 - Não haja um número grande de 'padrões'
 - Mercado de cada um seja grande suficiente
 - Haja meios de interconexão de tecnologias
- A convivência entre padrões é normal
 - Linguagens: C, C++, Pascal, Java, etc.
 - Redes: Ethernet, ATM, Wireless, etc.
 - Telefonia Celular: TDMA, CDMA, GSM, etc.

Padrões de Mercado

- Padrões
 - JavaBeans e Enterprise JavaBeans
 - Padrões proprietários da Sun Microsystems
 - Restritos à plataforma Java
 - COM, DCOM e ActiveX
 - Padrões proprietários da Microsoft
 - Restritos ao Windows
 - CORBA Component Model (CCM)
 - Padrão aberto proposto pela OMG
 - Independente de linguagem e plataforma

Java



- Plataforma Java

- Disponível em três edições

- Java ME (*Micro Edition*)

- Java SE (*Standard Edition*)

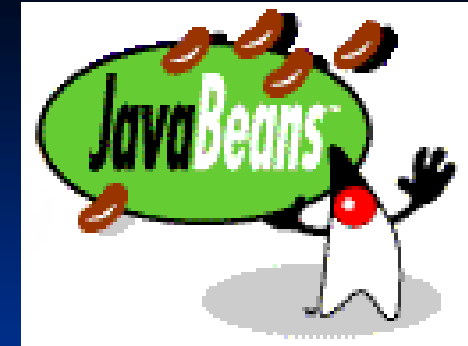
- Java EE (*Enterprise Edition*)

- Suporte para componentes

- *JavaBeans*: disponível no Java SE, EE e nas versões mais completas do Java ME

- *Enterprise JavaBeans* (EJB): presente no Java EE

JavaBeans



- JavaBeans
 - São componentes escritos em Java
 - Situados na camada de aplicação
 - Podem ser usados em aplicações, *applets*, servlets, páginas JSP, ...
 - API JavaBeans: `java.beans.*`
- JavaBeans possuem
 - Métodos e dados internos, como qualquer classe Java
 - Propriedades: modificadas em tempo de projeto

JavaBeans

- Comunicação entre Beans
 - Chamadas de métodos locais
 - Canais de eventos locais
 - Produtor: envia objetos
`java.util.EventObject`
 - Consumidor: implementa
`java.util.EventListener`
 - Não possui suporte nativo para comunicação remota

JavaBeans

- JavaBeans seguem os seguintes padrões:
 - São classes públicas
 - Possuem um construtor sem parâmetros
 - Nomes de métodos para acesso a propriedades e eventos:
 - Propriedade *X* acessada por métodos:
 - *setX()* e *isX()* se *X* for do tipo *boolean*
 - *setX()* e *getX()* para qualquer outro tipo
 - Consumidor do evento *Y* registrado com o método *addYListener()* e removido com *removeYListener()*

JavaBeans

- Detalhes sobre a interface de JavaBeans são obtidos:
 - Usando a API `java.lang.reflect` e buscando pelos nomes de métodos padronizados para JavaBeans
 - `get<Atributo>`, `set<Atributo>`
 - `add<Evento>Listener`,
`remove<Evento>Listener`
 - Através da interface `java.beans.BeanInfo`, que deve ser implementada por uma classe chamada `<NomeDoBean>BeanInfo`

JavaBeans

- Características adicionais dos JavaBeans
 - Salvam estado: interface `java.io.Serializable`
 - Controlam a concorrência e a segurança
 - Devem usar contêiners para acessar a plataforma e seus serviços
 - *Package* `java.beans.beancontext.*`

JavaBeans

- Vantagens e Limitações dos JavaBeans
 - Beans são reutilizáveis e configuráveis
 - São fáceis de usar e de compor com outros Beans
 - São mais fáceis de manter e distribuir que classes
 - Seu desenvolvimento é um pouco mais complexo que o desenvolvimento de classes e *packages* Java

JavaBeans

- Distribuição e Implantação
 - Beans são distribuídos em arquivos JAR
 - Arquivos JAR devem conter uma descrição do Bean
 - Para implantar o Bean, basta ter o arquivo JAR
 - Depois de implantados, os Beans podem ser configurados e compostos com outros componentes usando ferramentas de desenvolvimento

JavaBeans

- Componentes gráficos são JavaBeans
 - .. mas nem todos os JavaBeans são gráficos!
 - APIs AWT e Swing fornecem Beans gráficos
 - Propriedades alteram a aparência ou o comportamento do componente
 - Eventos são 'contidos': se propagam somente em uma janela da interface gráfica (*Frame* ou *Dialog*)

Java EE



- Plataforma Java Enterprise Edition
 - Adiciona ao Java suporte para:
 - Desenvolvimento de aplicações Web: JSP, *Servlets* e JSF
 - Desenvolvimento de Web Services: JAX-WS, JAXP, JAXB, JAX-RPC, SAAJ, JAXR
 - Componentes de Negócio: EJB
 - Suporte a Transações: JTA
 - Interconexão com Sistemas Legados: *Connectors*
 - Mecanismos de Comunicação Remota: JMS e JavaMail

Java EE

■ *Java Server Pages (JSP)*

- Permite a criação de páginas Web com conteúdo dinâmico, gerado por código Java ou por *JavaBeans*
- Usa APIs voltadas para a criação de HTML e XML

■ *Servlets*

- Aplicações que rodam em servidores Web
- São persistentes, ao contrário de scripts CGI
- Recebem requisições pela rede via HTTP ou HTTPS

Java EE

- *Java Server Faces (JSF)*
 - Framework para criação de interfaces Web gráficas
 - Acessível através da API Java e de arquivos de configuração em XML
- Suporte a Web Services
 - JAX-WS (*Java API for XML Web Services*): provê suporte para criação de serviços web e seus clientes
 - JAXP (*Java API for XML Processing*): realiza o processamento de dados em XML

Java EE

- Suporte a Web Services (cont.)
 - JAXB (*Java API for XML Binding*): faz o mapeamento entre dados em XML e objetos Java
 - JAX-RPC (*Java API for XML RPC*): permite o envio de chamadas remotas de procedimento a serviços web
 - SAAJ (*SOAP with Attachments API for Java*): permite o envio de mensagens SOAP, inclusive com anexos
 - JAXR (*Java API for XML Registries*): permite o acesso a registros de serviços em repositórios UDDI e ebXML

Java EE

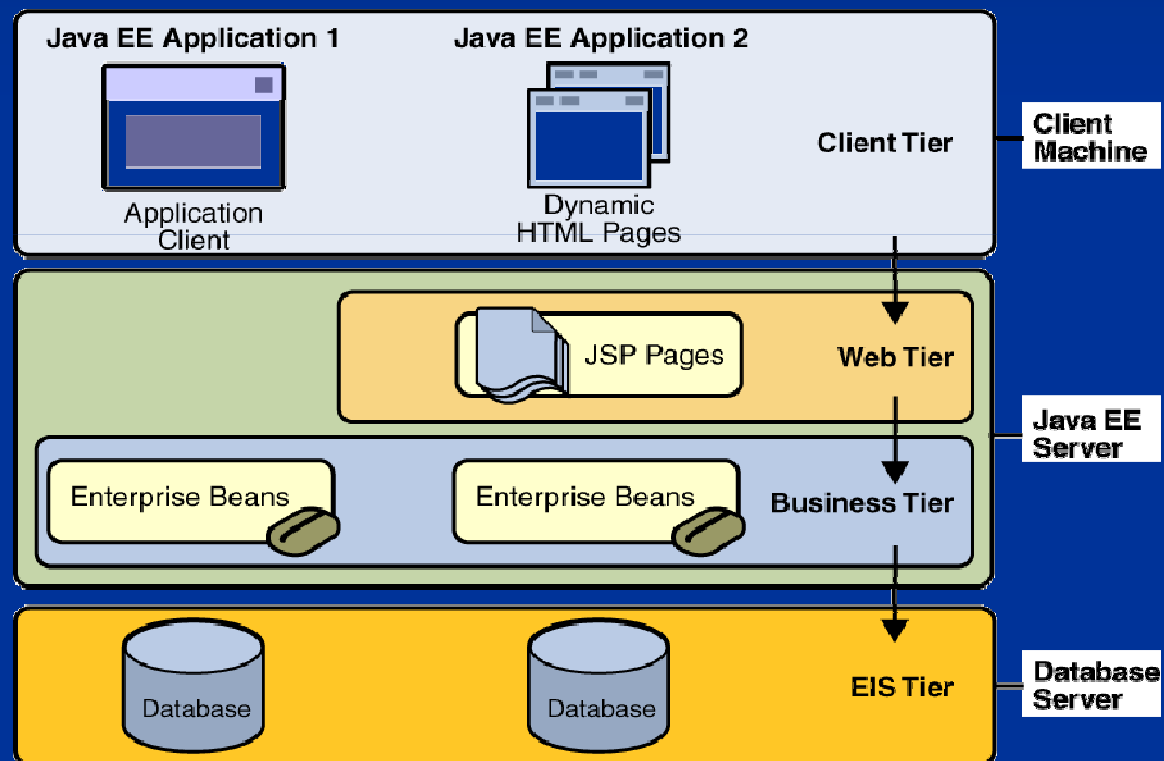
- *Enterprise JavaBeans* (EJB)
 - Componentes que rodam no servidor
 - Acessam os sistemas legados da empresa para implementar regras de negócio
- *Java Transaction API* (JTA)
 - API de suporte a transações
 - Transações são modeladas como objetos Java

Java EE

- *Java Messaging Service (JMS)*
 - Serviço para comunicação através de mensagens assíncronas (eventos)
- *JavaMail*
 - API para envio e recepção de e-mails
- *Java Connectors*
 - Permite integrar os sistemas legados usados nas empresas à arquitetura Java EE

Java EE

- Arquitetura multi-camadas do Java EE



Java EE

- Camadas do Java EE
 - Camada Cliente
 - Clientes Web: navegador, aplicações web, ...
 - Aplicações Java
 - Camada Web
 - Páginas JSP, JSF, *Servlets* e *JavaBeans*
 - Camada de Negócios
 - Componentes EJB
 - Camada de Sist. de Informações Empresariais
 - Integração com BDs e outros sist. legados

Java EE

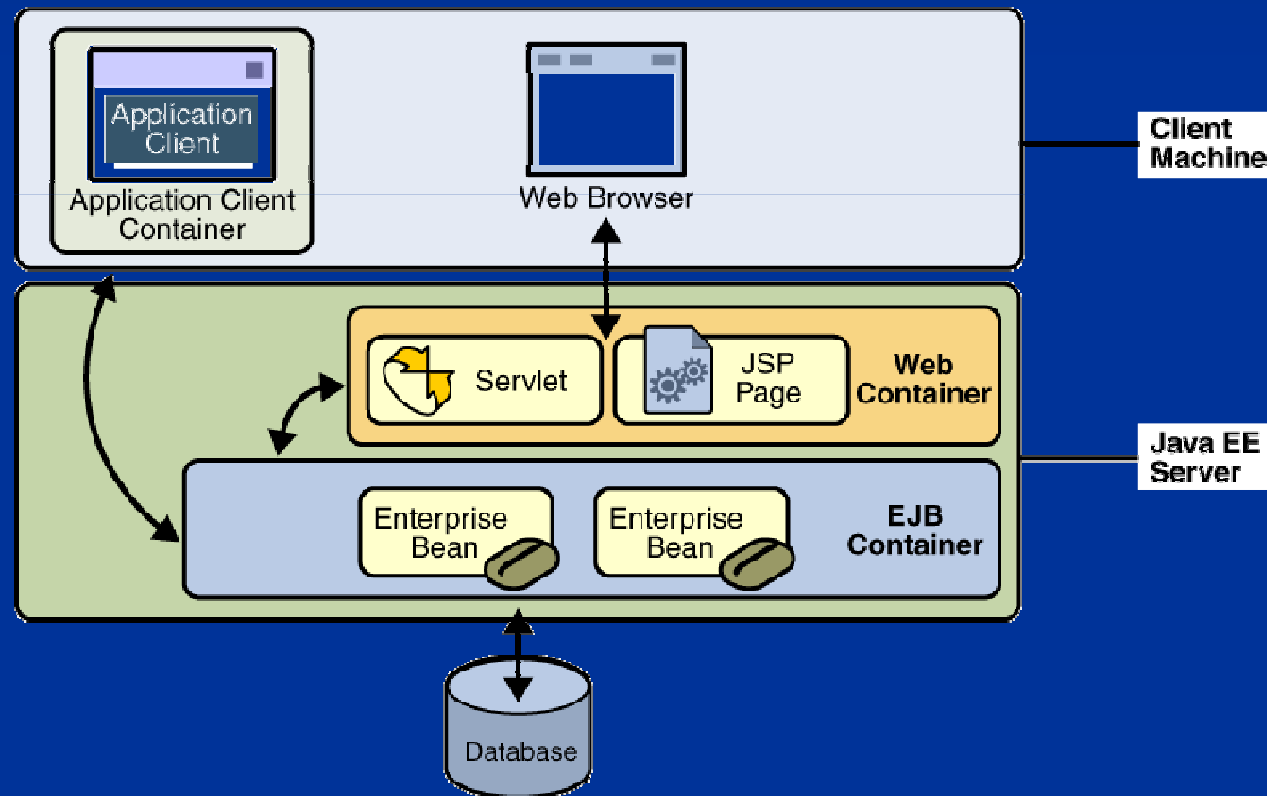
- Servidor Java EE
 - Possui duas camadas:
 - Camada Web
 - Composta por páginas JSP, JSF, *Servlets* e *JavaBeans*
 - Acessada pelos clientes Web
 - Camada de Negócios
 - Composta por componentes EJB
 - Usada pela camada Web e por aplicações clientes escritas em Java

Java EE

- Servidor Java EE
 - Fornece ambientes controlados de execução – os contêiners – para componentes Java EE
 - Contêiner Web: para *Servlets* e Páginas JSP/JSF
 - Contêiner EJB: para *Enterprise Beans*
 - Provê acesso transparente a serviços:
 - Transação: JTA
 - Segurança: JAAS
 - Localização: JNDI
 - etc.

Java EE

- Servidor Java EE: Contêineres Web e EJB



Java EE

- Camada de Sistemas de Informações Empresariais (EIS)
 - Usada pelos componentes EJB da camada de negócio p/ acesso a *software* de infraestrutura
 - Banco de Dados
 - Monitores de Transações
 - *Enterprise Resource Planning* (ERP)
 - *Customer Relationship Management* (CRM)
 - ... e outros sistemas legados
 - Estes sistemas geralmente rodam em *mainframes* ou servidores de médio porte
 - Conectores permitem o acesso a sist. legados

Java EE

■ Conectores

- Integram diversos sistemas à plataforma Java EE
- Fornecido pelo fabricante do sistema legado ou por terceiros
- Para desenvolver um conector geralmente é necessário escrever código nativo para a plataforma do sistema legado e integrar ao Java usando JNI (*Java Native Interface*), CORBA ou Sockets

Java EE

- Distribuição de aplicações corporativas
 - Arquivos que compõem uma aplicação Web são empacotados num arquivo WAR
 - Arquivos necessários para implantar EJBs devem ser empacotados em arquivos JAR
 - Uma aplicação corporativa completa é empacotada em um arquivo EAR
 - Contém uma ou mais aplicações Web em arquivos WAR
 - Contém um ou mais arquivos JAR com os componentes EJB da aplicação

Java EE

- Implantação de aplicações corporativas
 - Arquivos EAR são carregados no servidor Java EE, que abre o pacote e coloca a aplicação em execução
 - Conteúdo de arquivos é WAR implantado no contêiner Web
 - Componentes EJB contidos nos arquivos JAR são implantados no contêiner EJB
 - A implantação é efetuada com base em informações obtidas de descritores em XML e de anotações feitas nas próprias classes Java

Java EE

■ Interoperabilidade

- Java EE é compatível com o CORBA, padrão da OMG para comunicação remota entre objetos
 - Interfaces são compatíveis com CORBA IDL
 - Comunicação via RMI sobre CORBA IIOP
 - Transações são controladas pelo JTS, que é uma implementação do serviço de transações do CORBA
- *Beans* podem interagir com objetos COM e ActiveX através de *bridges*
- Interoperação com .NET é feita utilizando Web Services

EJB

- *Enterprise JavaBeans*

- Integra um modelo de componentes de negócio à arquitetura Java EE
- Cria uma camada composta de *beans* especializados, não-gráficos
- *Beans* rodam em servidores Java EE

- Componentes EJB

- São objetos Java escaláveis e reutilizáveis
- Utilizam anotações/arquivos XML para informar ao contêiner como devem ser gerenciados

EJB

■ Comunicação

- EJBs interagem com clientes remotos através de interfaces/Beans anotados com @Remote
- *Beans* podem ser acessados remotamente por:
 - Aplicações Java usando RMI/IIOP
 - Aplicações CORBA usando IIOP
 - Clientes Web via páginas JSP ou *Servlets*
- Clientes locais podem interagir com os EJBs utilizando injeção de dependência ou interfaces/*beans* anotados com @Local

EJB

- Tipos de *Enterprise Beans*
 - *Session Beans*
 - Executam uma tarefa durante uma sessão de interação entre o *Bean* o cliente
 - *Entity Beans*
 - Representam dados armazenados em BDs
 - Persistência transparente
 - *Message-Driven Beans*
 - São consumidores de mensagens JMS
 - Mensagens tratadas ao serem recebidas

EJB

■ *Session Bean*

- Representam um cliente em particular no servidor – ou seja, o *bean* não é compartilhado entre os clientes
- O cliente invoca métodos do *bean* para acessar o servidor – o *bean* age como uma extensão do cliente
- Pode ser acessado remotamente quando possui a anotação `@Remote` na classe do *bean* ou em uma interface que ela implementa

EJB

- Estado dos *Session Beans*
 - *Stateless Session Bean*
 - Não possui estado que o ligue a um cliente
 - Instâncias diferentes são equivalentes se inativas
 - *Stateful Session Bean*
 - Armazena estado durante a sessão de um cliente (entre invocações sucessivas)
 - O estado não é persistido (é transiente)

EJB

- Exemplo de *Stateless Session Bean*

```
@Stateless
public class HelloWorldSessionBean{
    public String hello(){
        return "Hello World";
    }
}
```

- A anotação @Stateless indica que a classe anotada é um *Stateless Session Bean*

EJB

- *Stateless Session Bean* com suporte a clientes remotos
 - Anotando a interface com @Remote

```
@Remote
```

```
public interface Hello{  
    public String hello();  
}
```

```
@Stateless
```

```
public class HelloWorldSessionBean implements Hello{  
    public String hello(){  
        return "Hello World";  
    }  
}
```

EJB

- *Stateless Session Bean* com suporte a clientes remotos
 - Anotando classe com @Remote

```
public interface Hello{  
    public String hello();  
}
```

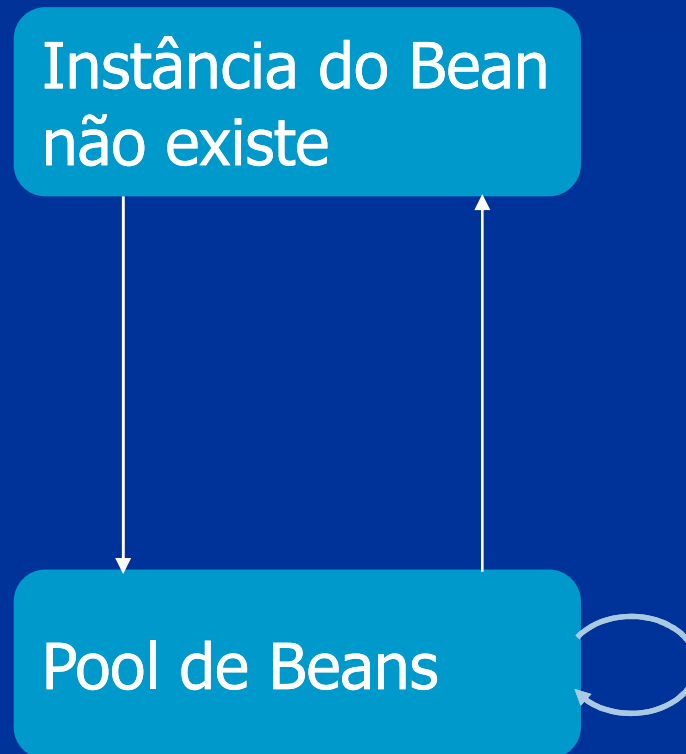
```
@Stateless
```

```
@Remote(Hello.class)
```

```
public class HelloWorldSessionBean implements Hello{  
    public String hello(){  
        return "Hello World";  
    }  
}
```

EJB

- Ciclo de vida dos *Stateless Session Beans*



EJB

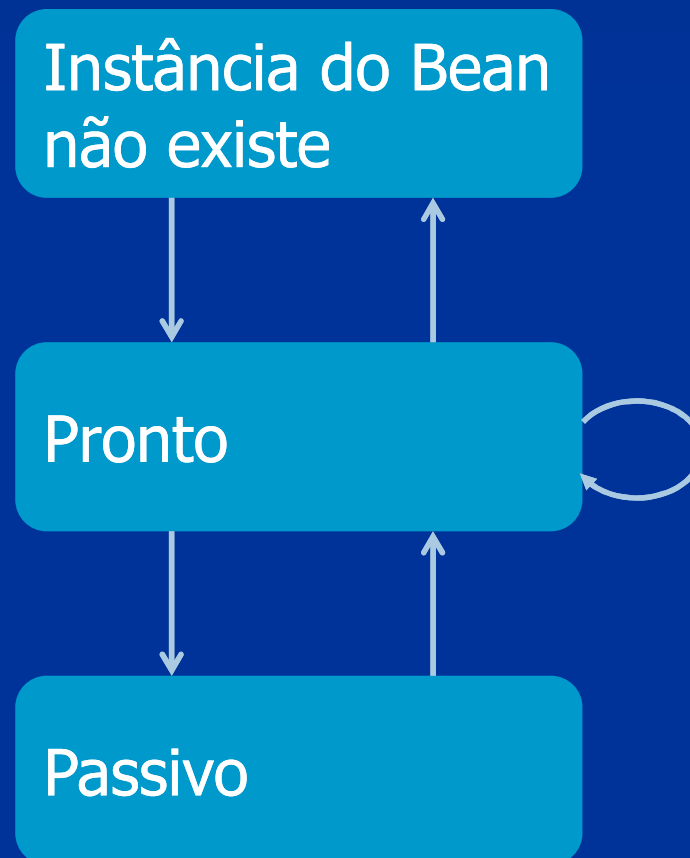
- Exemplo de *Stateful Session Bean*

```
@Stateful
@Remote(Count.class)
public class CountBean implements Count {
    private int count;
    public int getCount() { return this.count; }
    public void setCount(int count) { this.count = count; }
    @Remove
    public void remove() { }
}
```

- A anotação `@Stateful` indica que a classe é um *Stateful Session Bean*
- A anotação `@Remove` indica que o *bean* deve ser removido após a execução do método anotado

EJB

- Ciclo de vida dos *Stateful Session Beans*



EJB

- *Message-Driven Beans*
 - Consomem mensagens de uma fila ou associadas a um determinado tópico
 - Podem receber mensagens de uma ou de múltiplas fontes
 - Não possuem estado nem interfaces remotas
 - Beans são anotados com @MessageDriven
 - Sua interface depende do serviço de mensagens utilizado
 - Geralmente é usado JMS (*Java Message Service*)

EJB

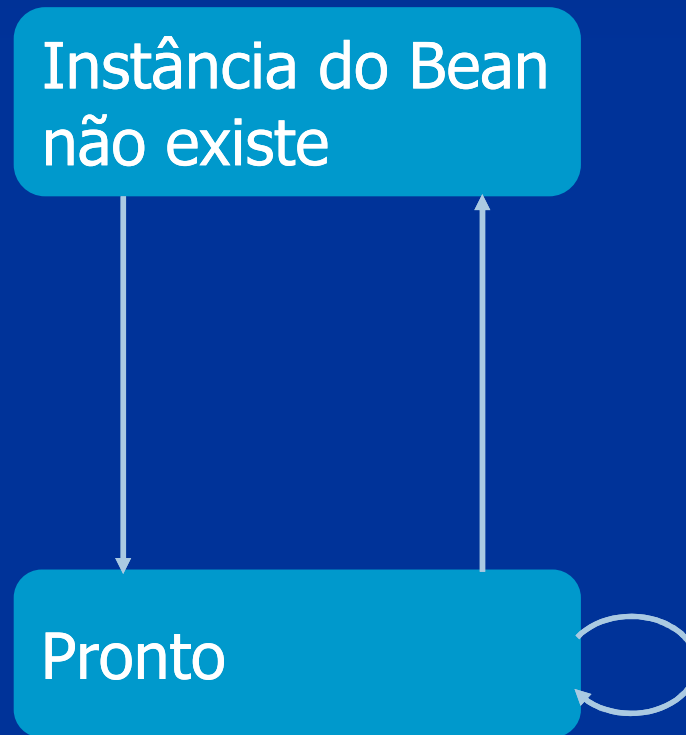
- Exemplo de *Message-Driven Bean*

```
@MessageDriven(mappedName="jms/Queue")
public class SimpleMessageBean implements MessageListener {
    public void onMessage (Message msg) {
        // utiliza a mensagem
    }
}
```

- O *bean* utiliza o JMS, que requer a implementação da interface *javax.jms.MessageListener*
- A fila de mensagens especificada na anotação `@MessageDriven` é mantida num provedor JMS

EJB

- Ciclo de vida dos *Message-Driven Beans*



EJB

■ *Entity Beans*

- São POJOS (*Plain Old Java Objects*)
- Permitem o acesso compartilhado a dados armazenados em um BD
- Dados são materializados na forma de objetos (mapeamento objeto-relacional)
 - *Bean* = tabela de um BD relacional
 - Instância do *Bean* = linha da tabela
 - Identificador do *Bean* = chave primária
- Utilizam a JPA (*Java Persistence API*) para controlar a persistência dos dados

JPA

- *Java Persistence API*
 - Modelo simplificado e leve de persistência
 - Pode ser utilizado tanto em contêineres JavaEE quanto em aplicações JavaSE
 - Permite utilização de herança e polimorfismo
 - Permite criação de testes independentes do contêiner quando utilizado com JavaEE
 - Possui anotações para definição de mapeamento objeto-relacional
 - Principais implementações da JPA
 - *Hibernate*
 - *Oracle TopLink*

JPA

■ Entidade

- No JPA uma entidade é um objeto comum Java (um POJO) que pode ser gravado pelo mecanismo de persistência
- Uma classe que representa uma entidade é anotada com @Entity
- Toda entidade deve possuir um construtor sem argumentos
- Toda entidade deve possuir uma chave primária, simples ou composta, identificada pela anotação @Id
- Chaves compostas devem ser representadas por uma classe Java em separado

JPA

■ Exemplo de Entidade JPA

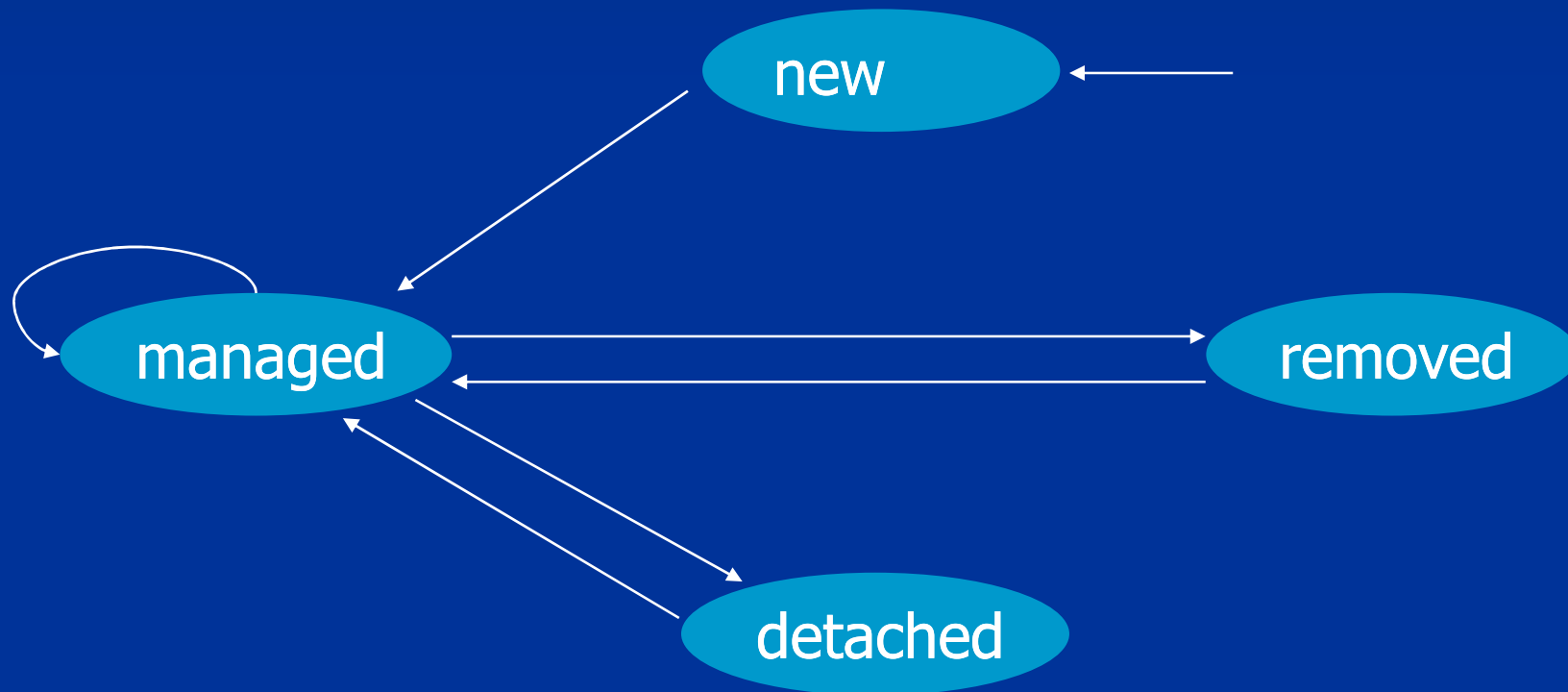
```
@Entity
public class Conta {
    @Id
    private Long numConta;
    private String nomeTitular;
    private long saldo;
    public Conta() { } //Construtor sem argumentos é obrigatório
    public void setNumConta(Long numConta) {
        this.numConta = numConta;
    }
    public Long getNumConta() { return this.numConta; }
    //Métodos getters e setters para os outros atributos
}
```

JPA

- Anotações de Relacionamento
 - Especificam relacionamento entre entidades e a cardinalidade da relação
 - @OneToOne
 - @OneToMany
 - @ManyToOne
 - @ManyToMany

JPA

- Ciclo de vida das entidades



JPA

■ Estados

- `NEW` : Estado da entidade após ser criada.
- `PERSISTENT` : Entidade persistida, com id associado a um contexto de persistência.
- `REMOVED` : marcada para ser removida do BD.
- `DETACHED` : Entidade possui um id persistente mas não possui um contexto de persistência.

■ Operações

- `persist` : Cria nova entidade
- `merge` : Persiste uma entidade
- `refresh` : Atualiza o estado de uma entidade
- `detach` : Sincroniza entidade desacoplada

JPA

- *Entity Manager*
 - Controla o ciclo de vida das entidades
 - Possui métodos para buscar, salvar, remover e atualizar estado das entidades
 - Referência para o *Entity Manager* é obtida com injeção de dependências, utilizando a anotação `@PersistenceContext`

Componentes CORBA

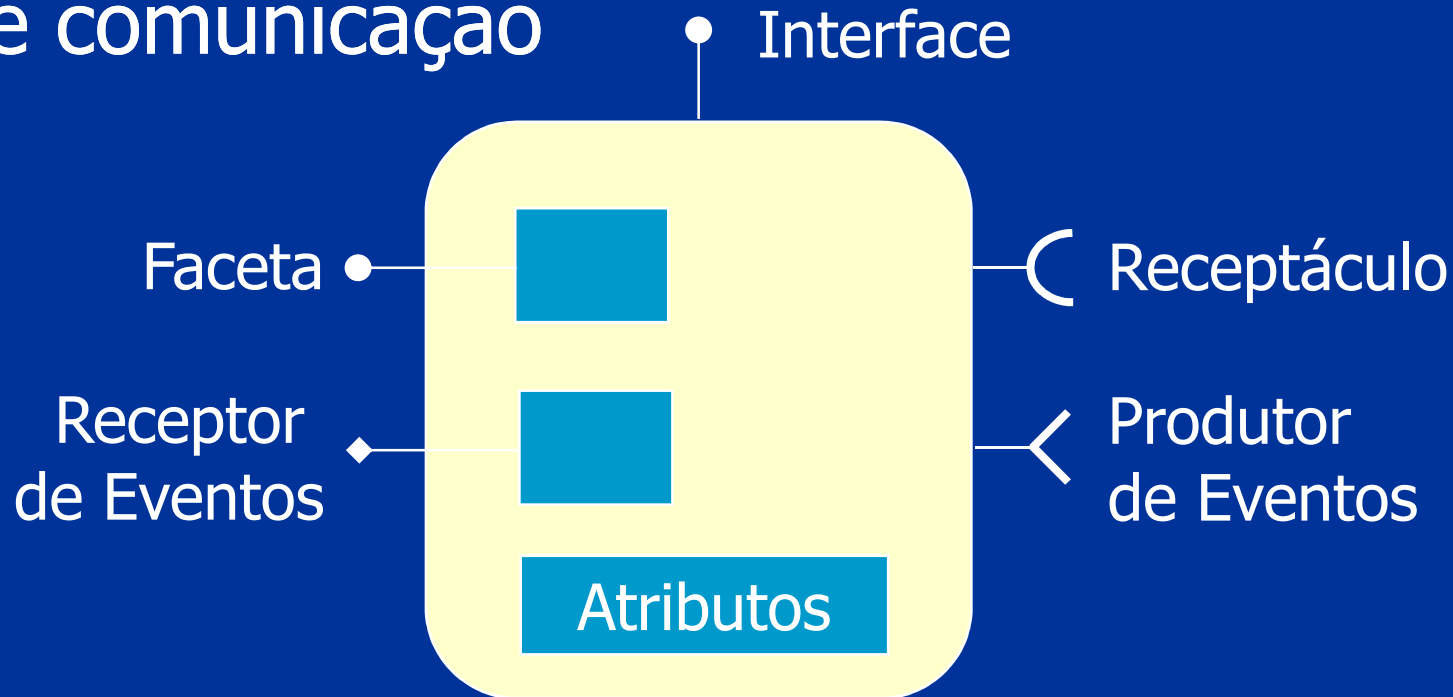
- CORBA *Component Model* (CCM)
 - Tem como objetivo suprir as deficiências do modelo de objetos do CORBA
 - Usa o CORBA como suporte de execução
- Histórico
 - Sua 1ª versão foi aprovada pela OMG em 1999
 - Tornou-se parte do CORBA 3.0, lançado em 2002
 - Alguns produtos já estão disponíveis

Componentes CORBA

- Componentes CORBA são elementos básicos usados para compor aplicações
 - Seus ciclo de desenvolvimento é padronizado
 - Podem ser estendidos, configurados e/ou combinados
 - Métodos padronizados de desenvolvimento, armazenamento, instanciação e interconexão
 - Ficam disponíveis em servidores

Componentes CORBA

- Componentes CORBA suportam uma interface e possuem vários tipos de portas de comunicação



Componentes CORBA

■ Facetas

- Interfaces IDL fornecidas pelo componente para acesso aos seus serviços
- Invocação síncrona nas facetas usa chamadas de operações CORBA
- Chamadas assíncronas usam AMI (*Asynchronous Method Invocation*)

■ Receptáculos

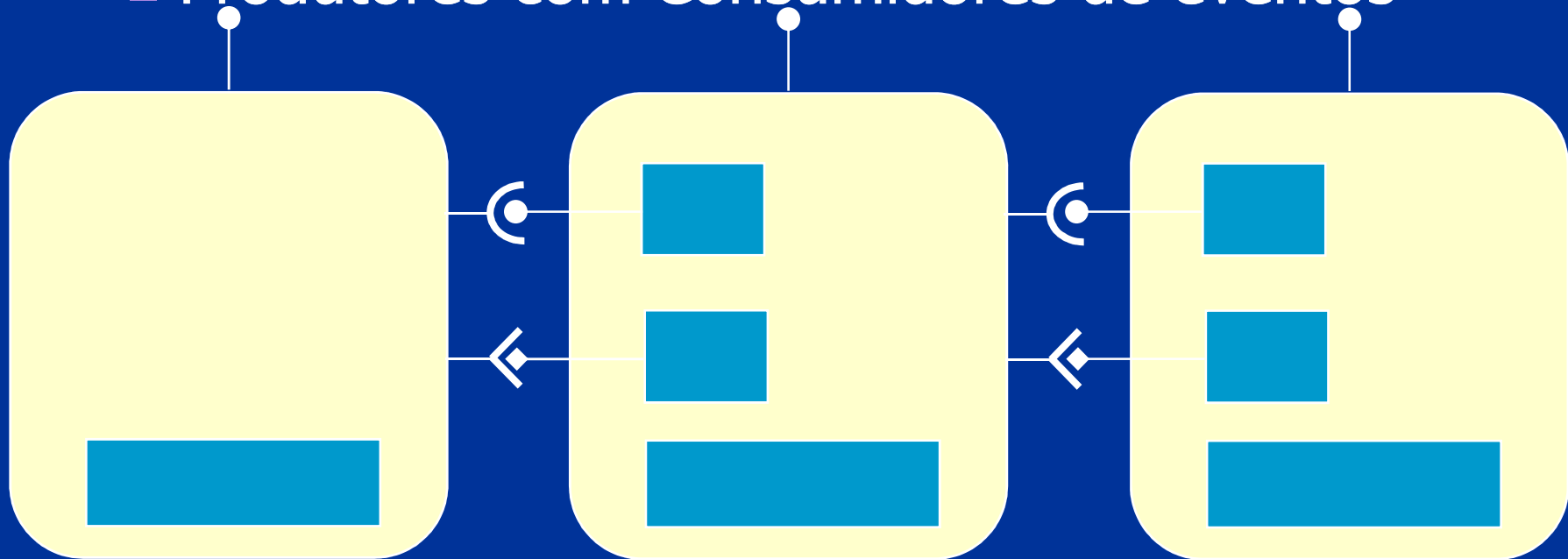
- Referências para outros objetos com os quais o componente interage
- Através deles o componente pode invocar operações de outros componentes

Componentes CORBA

- Produtores de Eventos
 - Emitem eventos assíncronos (para um consumidor) ou os publicam (número arbitrário de consumidores) usando o serviço de notificação do CORBA
- Consumidores de Eventos
 - Recebem eventos assíncronos usando o serviço de notificação do CORBA
- Atributos
 - Parâmetros de configuração do componente
 - CCM permite que eventos gerem exceções

Componentes CORBA

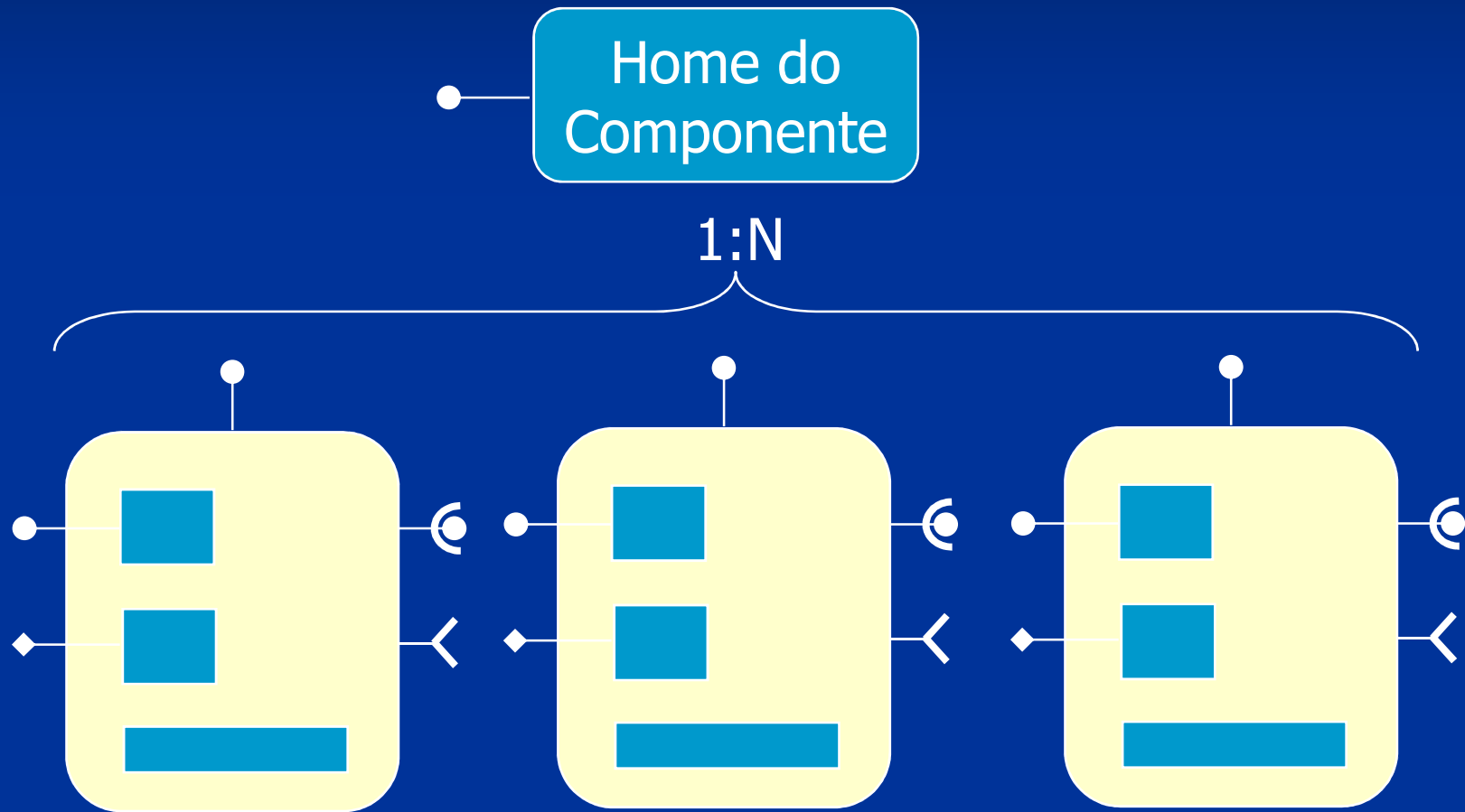
- Portas compatíveis podem ser conectadas
 - Facetas com Receptáculos
 - Produtores com Consumidores de eventos



Componentes CORBA

- Componentes CORBA possuem *Homes*
 - *Homes* são objetos CORBA especializados
- *Homes* gerenciam um tipo de componente
 - Criam e destroem instâncias
 - Localizam instâncias do componente
 - Efetua outras operações de gerenciamento (operações estáticas)
 - Armazenam atributos relacionados a um tipo de componente e suas instâncias (dados estáticos)

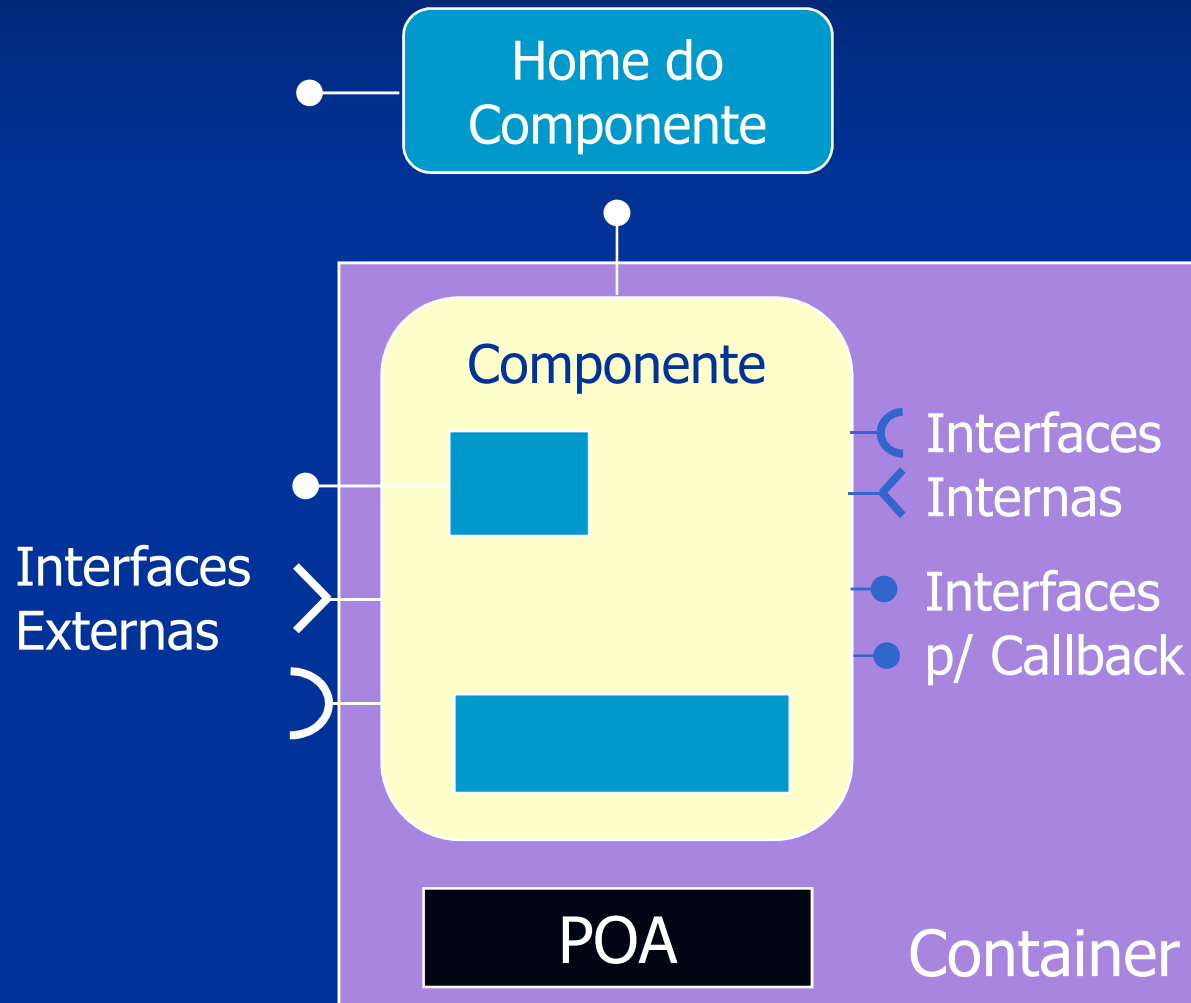
Componentes CORBA



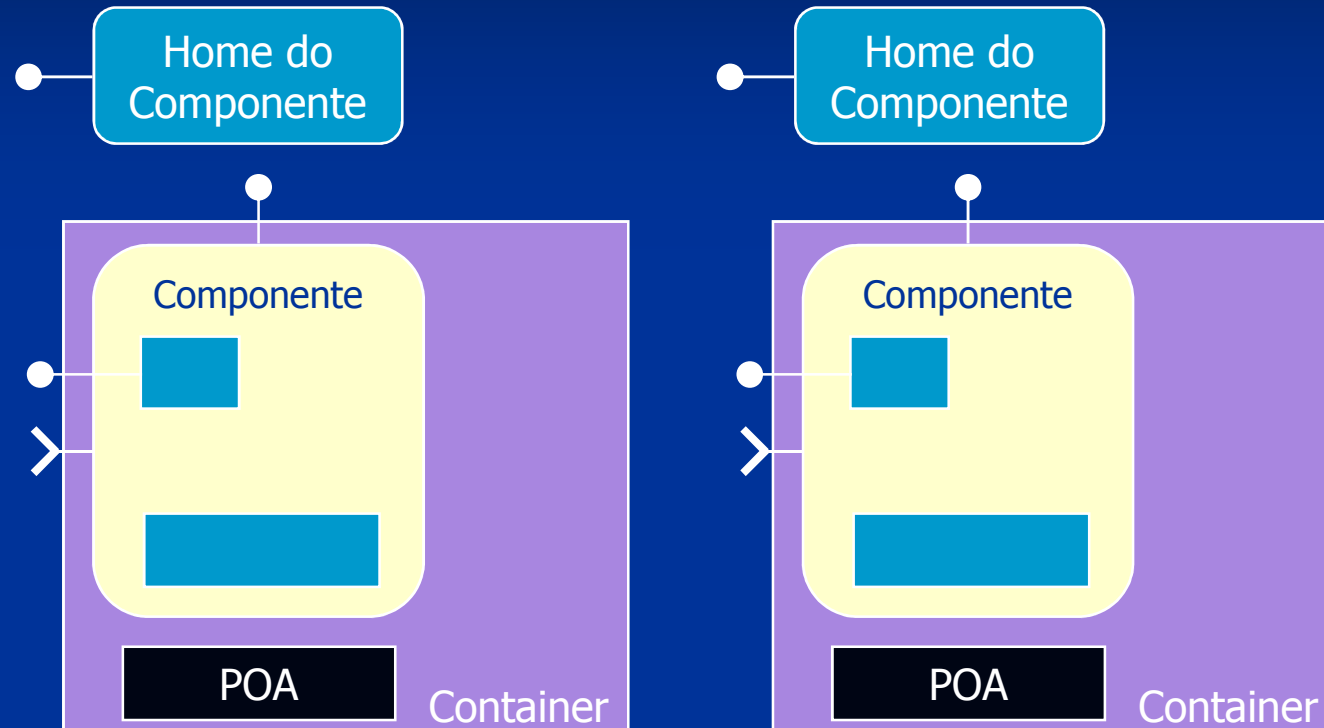
Componentes CORBA

- Componentes CCM usam *Containers*
 - *Containers* simplificam o desenvolvimento, a instanciação e a execução de componentes
- Funções do *Container*
 - Ativar/desativar componentes
 - Gerenciar o POA do componente
 - Interagir com serviços do ORB
 - Serviços de Notificação (Eventos), Segurança, Transação e Persistência
 - Interagir com o componente através de *callbacks*

Componentes CORBA



Componentes CORBA



Serviço de
Transação

Serviço de
Segurança

Serviço de
Persistência

Serviço de
Notificação

Componentes CORBA

- Tipos de Componentes CORBA
 - Serviço: sem estado
 - Sessão: estado transiente
 - Processo: com estado persistente
 - Entidade: estado persistente e chave primária
- Tipos de Contêiner
 - Contêiner sem suporte a persistência
 - Usado por componentes de serviço e sessão
 - Contêiner com suporte a persistência
 - Usado por componentes de processo e entidade

Componentes CORBA

- Especificação de Componentes
 - A OMG modificou a linguagem IDL para permitir a especificação de componentes
 - Na IDL do componente são descritas:
 - As interfaces suportadas pelo componente
 - As portas de comunicação do componente
 - Componentes suportam uma interface padrão com operações para gerenciamento, navegação, conexão de portas, etc.

Componentes CORBA

- Exemplo de especificação IDL de componente

```
component MyComponent supports MyInterface {  
  provides      MyFacet      faceta;  
  uses         MyReceptacle  receptaculo;  
  uses multiple MyReceptacle  receptac2;  
  emits       MyEventSource  evento1;  
  publishes   MyEventSource2 evento2;  
  consumes    MyEventSink    evento3;  
  attribute   MyAttribute    atributo;  
};
```

Componentes CORBA

- *Homes* também são descritas em IDL
 - *Homes* derivam de interfaces padrão
 - Contém operações de configuração, alocação e localização de componentes
 - *PrimaryKeys* podem ser usadas para identificar unicamente instâncias do componente
- São declarados na IDL de uma *Home*:
 - o tipo de componente gerenciado
 - operações de criação e localização
 - operações e atributos normais do CORBA

Componentes CORBA

- Declaração de uma *Home* em IDL:

```
home MyHome manages MyComponent
  primaryKey MyKeyType {
  factory     factory_operation(parameters)
              raises (exceptions);
  finder      finder_operation(parameters)
              raises (exceptions);
  MyRetType  mgmt_operation(parameters)
              raises (exceptions);
  attribute  MyAttributeType attr;
};
```

Componentes CORBA

- O CCM define um *Framework* que automatiza parte do processo de implementação
- *Component Implementation Description Language* (CIDL)
 - É uma linguagem declarativa, assim como a IDL
 - Faz a composição de componentes e suas *homes*

```
composition session MyComposition {  
    home executor MyHomeImpl {  
        implements MyHome;  
        manages MyComponentImpl;  
    };  
};
```

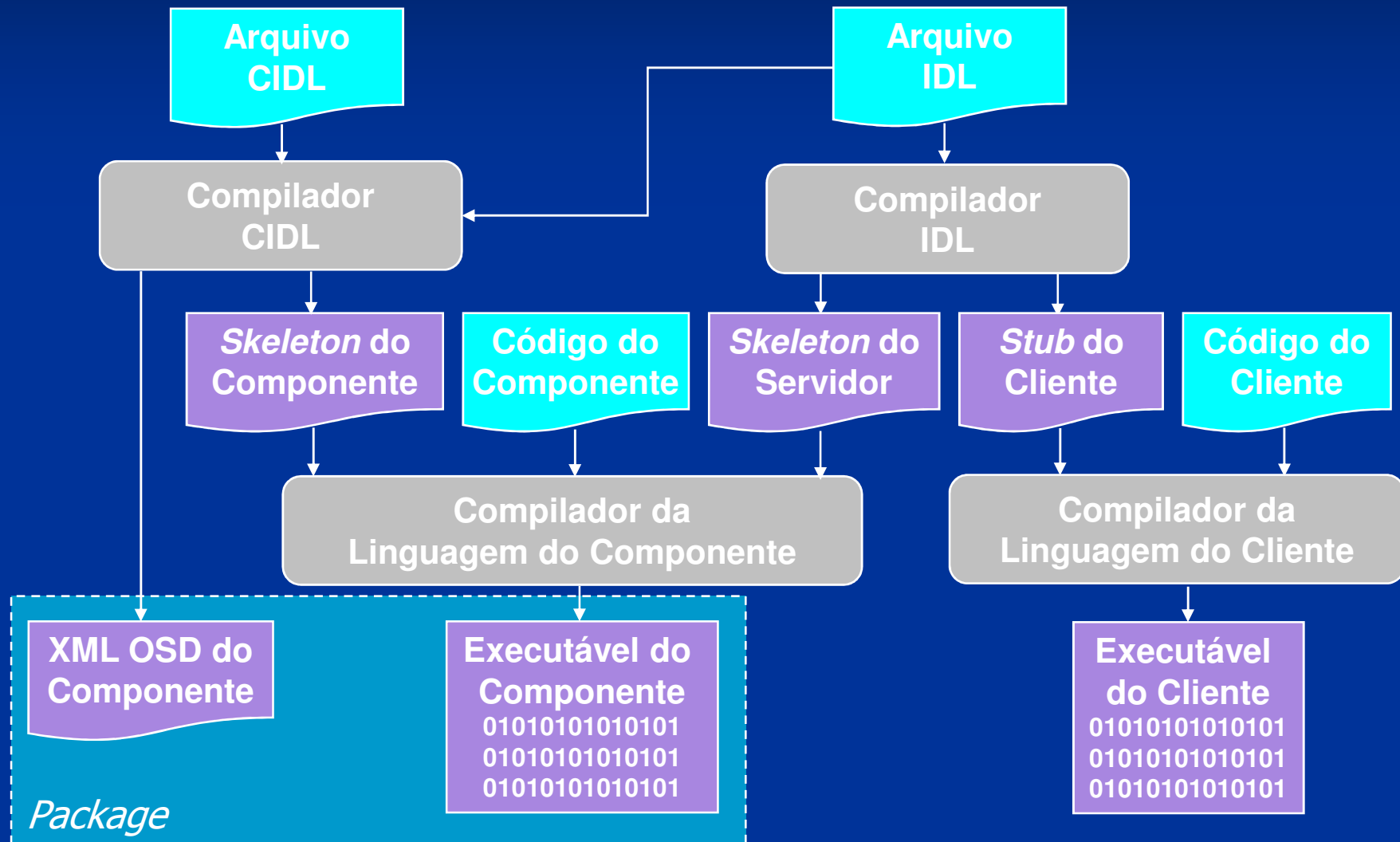
Componentes CORBA

- Compilador CIDL gera:
 - *Skeleton* do componente: código para navegação, identificação, ativação e gerenciamento de estado
 - Descritor do componente em XML OSD (*Open Software Description*)

Componentes CORBA

- Distribuição de Componentes:
 - O código executável de um componente e seu descritor são empacotados em um *package*
 - Um *package* pode ter uma ou mais implementações de um componente
 - *Packages* podem ser disponibilizados em servidores de componentes

Componentes CORBA



Componentes CORBA

- CCM define procedimentos padrão para distribuir e instanciar componentes
 - O componente é instanciado carregando seu *package* em um servidor de componentes
 - Instanciado pelo servidor de componentes, o componente torna-se apto a receber requisições

Componentes CORBA

- Componentes podem ser reunidos em um *assembly*
 - Um *assembly* é um agrupamento de componentes
 - Os componentes de um *assembly* e as conexões entre eles são descritos em um arquivo no formato CSD (*CORBA Software Descriptor*)

Componentes CORBA

- CCM pode ser suportado em dois níveis
 - Nível Básico
 - Componentes possuem somente atributos
 - O suporte deve ser capaz de transformar objetos CORBA em componentes
 - Outras limitações existem em relação a herança e ao uso de homes
 - Nível Estendido: CCM completo

Nível básico do CCM
implementado em Java
usando RMI sobre IIOP



*Enterprise
JavaBeans
(EJB)*

Componentes CORBA

- Vantagens do CCM em relação ao EJB
 - Permite usar outras linguagens além de Java para implementar componentes
 - Possui um modelo de comunicação mais completo
 - É um padrão da OMG
- Limitações do CCM
 - Poucas implementações no mercado
 - Maior complexidade