INE 5612

Desenvolvimento de Sistemas Orientados a Objetos II

Prof. Frank A. Siqueira
E-Mail: frank@inf.ufsc.br
URL: http://www.inf.ufsc.br/~frank/INE5612

INE 5612 Conteúdo Programático

- Unidade I – Componentes de Software
  Motivação; O Paradigma de Componentes; Vantagens e Limitações; Interfaces; Contêineres; Interação; Projeto; Desenvolvimento; Testes; Distribuição; Implantação; Padrões e Produtos
- Unidade II - Componentes no Java SE
  A Plataforma Java SE; JavaBeans; Desenvolvimento de Aplicações usando JavaBeans; Construção de JavaBeans
- Unidade III - Componentes no Java EE
  A Plataforma Java EE; Java Server Faces; Enterprise JavaBeans; Java Persistence API
- Unidade IV – Novas Tecnologias de Desenvolvimento de Sistemas (Seminários)

Unidade I

Componentes de Software

- Motivação
- O Paradigma de Componentes
- Vantagens e Limitações
- Interfaces
- Contêineres
- Interação
- Projeto
- Desenvolvimento
- Testes
- Distribuição
- Implantação
- Padrões e Produtos

Motivação

- Crise do Software
  - Constatada ao final dos anos 60
  - Software evoluía mais lentamente e era visto como menos confiável do que o hardware
  - Poucos fornecedores de software no mercado
  - Usuários especializados em computação
  - Técnicas de Eng. Software pouco difundidas
  - Ferramentas de desenvolvimento limitadas
  - Linguagens de mais baixo nível
  - Pouca proteção contra falhas nos SOs

Motivação

- Outras indústrias passaram por obstáculos similares aos enfrentados pela indústria de software naquela época
- Na maioria delas, a adoção de componentes resultou em ganhos significativos
  - Indústria eletrônica
  - Indústria automobilística
  - Construção civil
  - etc.
- Por que não utilizar componentes também para construir software?

Motivação

- Já ao final dos anos 60 reconhecia-se os benefícios advindos do uso de componentes de software reutilizáveis
- A tecnologia da época não permitiu que a ideia de desenvolver software a partir de componentes fosse usada na prática
- Nos anos 80, apesar da popularização das linguagens orientadas a objetos, não houve a melhoria esperada em relação à qualidade do software

Motivação

- O que mudou nos últimos anos?
  - Softwares estão cada vez mais complexos e com suporte à interação via rede
  - Maioria dos usuários sem formação na área
  - Evolução de linguagens, SOs, ferramentas e técnicas de desenvolvimento de software
  - O mercado de software tornou-se altamente competitivo, exigindo que o desenvolvedor produza mais em menos tempo
  - Qualidade do software ainda é insatisfatória!

Motivação

- A tecnologia de componentes tornou-se viável com as novas ferramentas de desenvolvimento e ambientes de execução
  - Ambientes de desenvolvimento permitem que aplicações sejam "montadas" graficamente
  - Ambientes de execução podem, via reflexão computacional, inspecionar um componente de modo a prover o suporte adequado
  - Servidores de aplicação fornecem o suporte necessário para execução de componentes, permitindo, por exemplo, a interação via rede

O Paradigma de Componentes

- Componente
  - "*Aquilo que entra na composição de alguma coisa. Parte elementar de um sistema.*" (Fonte: Aurélio)
- Composição
  - "*Formar ou construir de diferentes partes.*" (Fonte: Aurélio)

O Paradigma de Componentes

- 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)

O Paradigma de Componentes

- Componentes de Software
  - Unidades funcionais de software
  - Executam uma atividade bem definida no sistema
  - Produzidos e implantados independentemente
  - Distribuídos em código binário
  - Combinados para compor aplicações
  - Podem ser configurados de acordo com as necessidades dos usuários
  - Podem ser facilmente atualizados e reutilizados
  - Podem ser criados usando métodos, técnicas e ferramentas de desenvolvimento padronizados

O Paradigma de Componentes

- A estrutura de um componente é definida por um modelo seguido na implementação
- 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

O Paradigma de Componentes

- Estrutura interna de um componente
  - Código funcional
    - Responsável pela implementação dos serviços fornecidos pelo componente
    - Escrito pelo desenvolvedor
  - Código não-funcional
    - Responsável por interagir com o suporte de comunicação e de execução padronizado pelo modelo de componentes
    - Pode ser gerado automaticamente pelo suporte de desenvolvimento

O Paradigma 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

O Paradigma de Componentes

- Componentes podem ser classificados em função da forma como lidam com seus dados de estado
  - *Stateless*: componentes sem estado
  - *Stateful*: componentes com estado
    - Estado interno: dados privados do componente
    - Estado externo: dados acessíveis através da interface

O Paradigma de Componentes

- Estado do componente pode ser persistente ou volátil
  - Estado persistente: mantido em memória persistente (HD, BD, etc.) e restaurado em execuções subsequentes
  - Estado volátil (não-persistente): dados são perdidos entre uma execução e outra

O Paradigma de Componentes

- Componentes rodam sobre um suporte de execução definido pelo modelo
- O suporte de execução pode ser fornecido por:
  - Linguagem de programação
  - Máquina virtual
  - Contêiner
  - Servidor de aplicação

O Paradigma de Componentes

- Mecanismos de comunicação são fornecidos pelo suporte de execução para interação entre componentes
  - O formato dos dados e o protocolo de comunicação utilizado devem ser padronizados para que componentes possam interagir
- Comunicação entre componentes pode ser:
  - Local: componentes em uma mesma máquina
  - Remota: componentes em máquinas diferentes

O Paradigma de Componentes

- O processo de desenvolvimento, distribuição e implantação de componentes segue regras bem definidas, podendo utilizar-se de:
  - Metodologias de desenvolvimento
  - Compiladores e geradores de código
  - Servidores de aplicação
  - Mecanismos de distribuição
  - etc.

Vantagens e Limitações

- O paradigma de componentes simplifica o desenvolvimento de sistemas
  - A complexidade necessária para executar uma tarefa é encapsulada por um componente, o que esconde detalhes internos do mundo externo
  - Uma aplicação pode ser totalmente construída a partir de componentes reutilizados ou fornecidos por terceiros
  - Um componente pode ser substituído/atualizado independentemente do restante da aplicação, facilitando a manutenção e evolução do sistema
  - Componentes são testados individualmente, melhorando a confiabilidade da aplicação

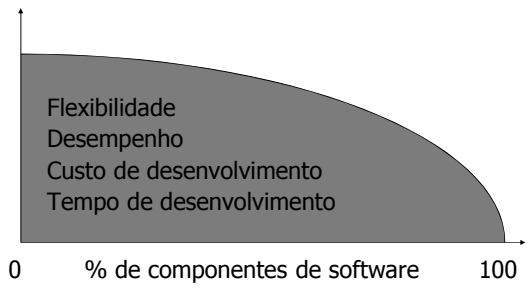
Vantagens e Limitações

- Reutilização é favorecida pelo uso de componentes de software
- Outras unidades de desenvolvimento de software, como classes e bibliotecas, não trazem os mesmos benefícios:
  - São dependentes de linguagem/ambiente
  - Possuem granularidade inadequada
  - São difíceis de combinar
  - Não são tão flexíveis quanto componentes

Vantagens e Limitações

- Nem todos os problemas são resolvidos pelos componentes de software
  - Generalização pode levar a ineficiência
  - Interfaces e/ou modelos de componentes podem ser incompatíveis, impedindo sua composição
  - Mecanismos para interação entre componentes precisam ser padronizados
  - É raro encontrar componentes de fabricantes diferentes que sejam intercambiáveis
  - Nem sempre é possível configurar um componente para atender todas as necessidades do usuário

Vantagens e Limitações



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

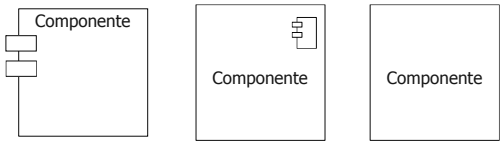
```
public interface CadastroUsuarios {
   public void cadastrar(String nome, String email,
      long senha) throws UsuarioJaCadastrado;
   public boolean autenticar(String email, long senha);
   public void alterarSenha(String email, long senha,
      long novaSenha) throws UsuarioNaoEncontrado;
}
```

- Exemplo de Interface em CORBA IDL

```
interface CadastroUsuarios {
   void cadastrar(in string nome, in string email,
      in long senha) raises(UsuarioJaCadastrado);
   boolean autenticar(in string email, in long senha);
   void alterarSenha(in string email, in long senha,
      in long novaSenha) raises (UsuarioNaoEncontrado);
}
```

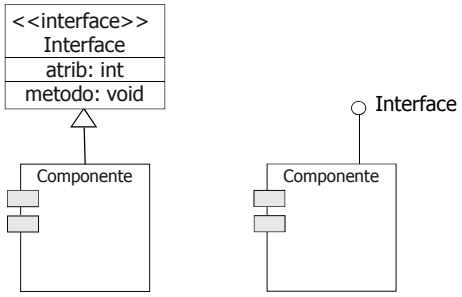
Interfaces

- Representação em UML de componentes



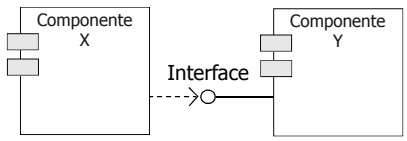
Interfaces

- Representação em UML de interfaces



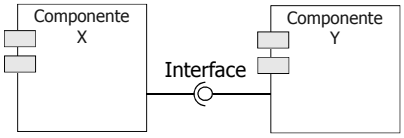
Interfaces

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



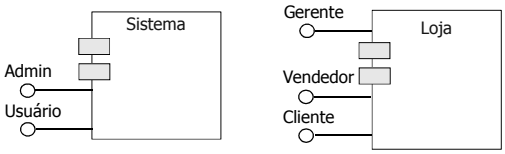
Interfaces

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



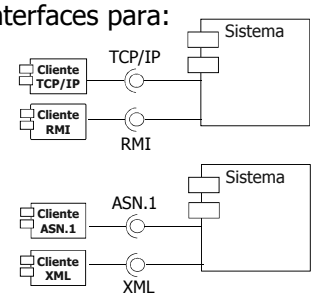
Interfaces

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



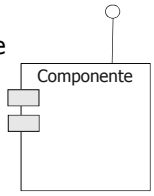
Interfaces

- 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

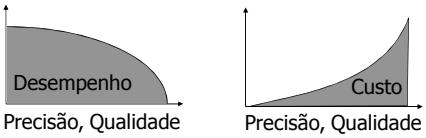


Interfaces

- 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

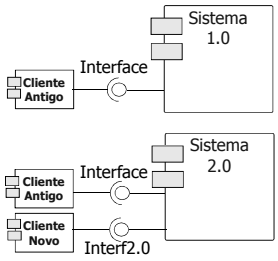
Interfaces

- 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



Contêineres

- 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 portável e mais fácil de reutilizar

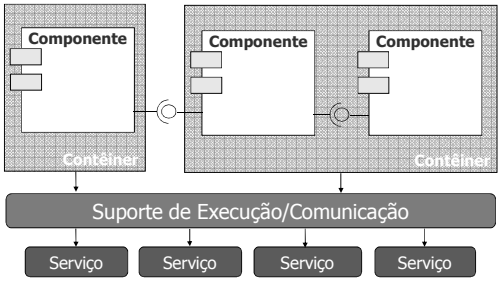
Contêineres

- 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 portável
- Podem haver diferentes tipos de Contêiner
  - Exemplo: Contêiners para componentes sem estado, com estado transiente (volátil) ou persistente

Contêineres

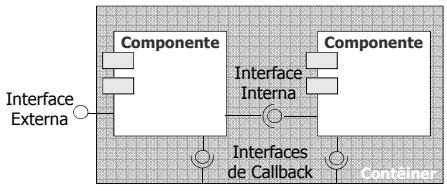
- Contêiners utilizam recursos de software e hardware e serviços da plataforma de execução para executar o componente
  - Serviço de Nomes: permite localizar instâncias de componentes
  - Serviço de Comunicação: troca de informação
  - Serviço de Persistência: faz o armazenamento de estado dos componentes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

Contêineres



Contêineres

- 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



Contêineres

- 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

Interação

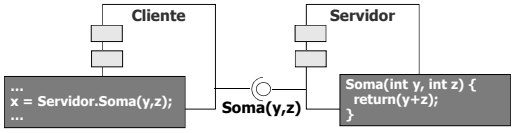
- Componentes interagem para trocar dados e solicitar a execução de serviços
- Componentes podem estar localizados em:
  - Um mesmo servidor → Interação local
  - Servidores diferentes → Comunicação remota
- O ideal é que o desenvolvedor abstraia a localização dos componentes, podendo agir da mesma forma independentemente de os componentes estarem na mesma máquina ou em máquinas diferentes

Interação

- Para abstrair a localização de componentes o ideal é que os mecanismos usados para interação local sejam usados também para comunicação remota
- Mecanismos de interação local
  - Chamada de procedimento/método
  - Notificação de eventos/mensagens
- Mecanismos de comunicação remota
  - Chamada remota de proced./método (RPC)
  - Notificação de eventos/mensagens remotos

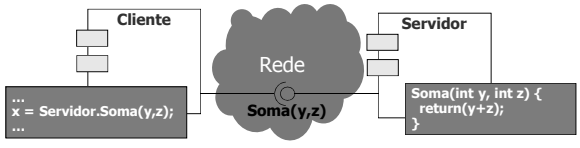
Interação

- Chamada de procedimento/método
  - Segue o modelo Cliente/Servidor
  - Um componente fornece uma interface com procedimentos/métodos para solicitar a execução de seus serviços – é um servidor
  - Componentes/aplicações utilizam os serviços de outros componentes – são seus clientes



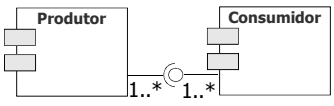
Interação

- Chamada remota de procedimento/método
  - Idêntica a uma chamada local
  - Os componentes estão em máquinas diferentes
  - A chamada tem que ser enviada como uma mensagem pela rede
  - O suporte de execução se encarrega disso



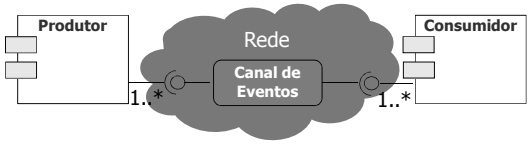
Interação

- Notificação de eventos
  - Notificações de eventos ocorridos são difundidas por produtores e entregues a consumidores de eventos
  - Eventos podem ser oriundos da interface gráfica. Ex.: click de um botão do mouse, tecla digitada, seleção de um elemento, etc.



Interação

- Notificação de eventos remotos
  - Produtores e consumidores de eventos podem estar em máquinas diferentes
- Canal de eventos permite desacoplamento
  - Não é necessário que as partes se conheçam ou se sincronizem para enviar o evento

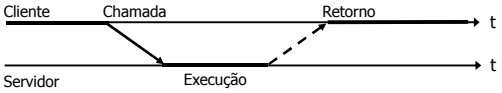


Interação

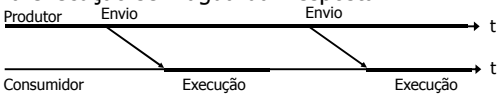
- Formato dos dados
  - Na chamada de método, a sua assinatura define os parâmetros e seus tipos de dados
  - O formato dos eventos é em geral mais flexível
- Cardinalidade
  - Chamada de método tem sempre apenas um cliente e um servidor
  - Notificação de eventos pode ser de M para N

Interação

- Sincronismo
  - Na chamada de método o emissor fica bloqueado aguardando o término da execução



  - Na notificação de eventos o produtor continua a execução sem aguardar resposta



Interação

- Protocolos de comunicação de alto nível são necessários para interação entre componentes em máquinas diferentes
- Escolha natural: usar TCP/IP como base
  - Cria conexões entre processos para troca de mensagens
  - Amplamente disponível, confiável e robusto
  - Relativamente simples e eficiente

Interação

- Mecanismo de comunicação entre componentes deve tratar questões não resolvidas pelo TCP/IP
  - Formato comum dos dados
  - Localização de componentes
  - Segurança
- Os protocolos usados pelos suportes de execução de componentes já incorporam tais mecanismos

Projeto

- Projeto de sistemas basedos em componentes
  - Técnicas tradicionais de Eng. de Software podem ser usadas na análise e projeto de sistemas baseados em componentes
  - Com base nas funcionalidades requeridas do sistema, define-se quais componentes são necessários para sua construção
  - Deve-se buscar reutilizar componentes e/ou desenvolver novos componentes que possam ser reaproveitados

Projeto

- Projeto deve levar em conta que:
  - O componente deve fornecer serviços a outros componentes, que os utilizarão através das interfaces do componente
  - Deve ser possível ajustar a forma como os serviços são executados, alterando valores de propriedades
  - O componente deve ser sujeito a composição
  - O componente deve procurar ser independente de outros componentes

Projeto

- Abordagem de Cheesman & Daniels
  - Modelagem do Domínio: realizada para entender o contexto do problema a ser solucionado, sem assumir nenhuma solução de software
  - Especificação do Software: consiste na concepção de um sistema de software para resolver o problema em questão

Projeto

- Modelagem do Domínio
  - Definir casos de uso
  - Definir modelo conceitual
  - Definir modelo comportamental
- Especificação do Software
  - Identificar componentes
  - Identificar interações entre componentes
  - Especificar componentes

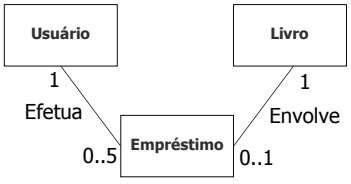
Projeto

- Casos de uso
  - Especificam as funcionalidades existentes no sistema
  - Exemplo: Sistema de Bibliotecas

Nome:	**Efetuar empréstimo**
Objetivo:	Emprestar um livro a um usuário
Pré-condição:	Livro deve estar disponível para empréstimo; usuário deve ter <5 livros emprestados.
Ação:	emprestar (livro, usuário)
Nome:	**Fazer devolução**
Objetivo:	Devolver um livro emprestado a um usuário
Pré-condição:	Livro estar emprestado; não estar danificado.
Ação:	devolver (livro)

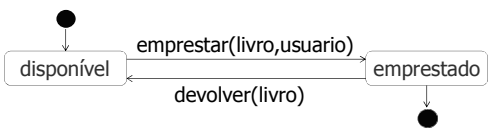
Projeto

- Modelo conceitual
  - Especificar as entidades do mundo real manipuladas pelo sistema e suas associações
  - Exemplo:



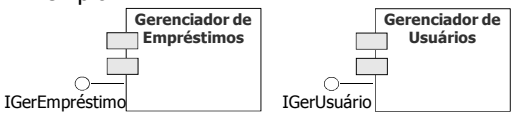
Projeto

- Modelo comportamental
  - Identificar estados, eventos e transições de estado
  - Exemplo: Diagrama de estados de livro



Projeto

- Identificação de componentes
  - Produz uma especificação de arquitetura inicial de um sistema
  - Identifica as interfaces suportadas por cada componente
  - Deve-se sempre buscar reutilizar componentes
  - Exemplo:



Projeto

- Interação entre componentes
  - Identifica as operações necessárias
  - Atribui responsabilidades
  - Exemplo:
    - Se uma pré-condição exige que o usuário respeite um limite máximo de empréstimos, deve haver uma operação para recuperar o número de empréstimos de um usuário
    - O componente Gerenciador de Usuários e a entidade Usuário devem ter essa operação

Projeto

- Especificação dos componentes
  - Cria uma especificação detalhada das interfaces dos componentes, definindo as assinaturas de suas operações e suas propriedades
  - Exemplo:

<<interface>> IGerEmprestimo
numLimiteEmprestimos: integer
emprestar(livro: Livro, usuario: Usuario) devolver(livro: Livro)

Desenvolvimento

- Desenvolvimento de sistemas é baseado em reutilização de componentes
- Quando não houver um componente disponível que forneça as operações desejadas, um novo componente deverá ser desenvolvido, tendo em mente a possibilidade de reutilização
- Desenvolvimento de componentes
  - São usadas linguagens de programação tradicionais
  - Deve seguir padrões definidos por um modelo de componentes

Desenvolvimento

- Composição: permite que aplicações ou novos componentes sejam criados a partir de outros componentes
- Diferenças entre herança e composição
  - Herança
    - X é um Y
    - Usada para estender a funcionalidade
  - Composição
    - X tem um Y
    - Usada para incorporar a funcionalidade

Desenvolvimento

- Composição é mais adequada que herança para reutilizar código
  - Na maioria dos casos não queremos estender, mas incorporar a funcionalidade do componente
- Compare:
  - Interface gráfica é um Botão + Rótulo + ...
    ?? ou ??
    Interface gráfica tem um Botão + Rótulo + ...
  - Processador de texto é um dicionário
    ?? ou ??
    Processador de texto tem um dicionário

Desenvolvimento

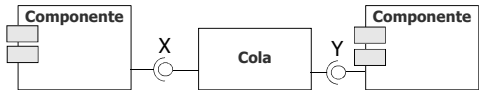
- Composição de componentes pode ser feita através de:
  - Código de programas
  - Linguagens de composição
  - Ferramentas gráficas de composição
- Linguagens e ferramentas de composição devem verificar se:
  - As dependências de componentes são satisfeitas
  - As interfaces conectadas são compatíveis

Desenvolvimento

- Grande parte dos ambientes de desenvolvimento fornece ferramentas para geração de código
  - Código não-funcional (para interação com o suporte) é gerado automaticamente
  - Código funcional (os métodos que implementam o comportamento do componente) deve ser escrito pelo programador

Desenvolvimento

- Código "cola" (*glue*)
  - Código adicional usado para adaptação de interfaces ligeiramente diferentes
  - Permite conectar um componente que requer a interface X a outro que provê a interface Y, caso X e Y implementem as mesmas funcionalidades, mas as assinaturas de métodos sejam diferentes

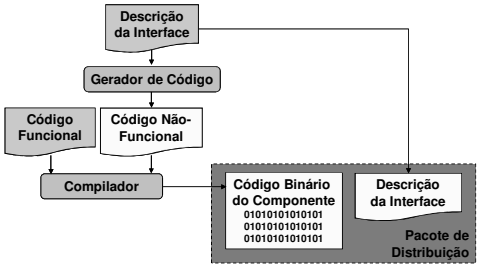


Desenvolvimento

- 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

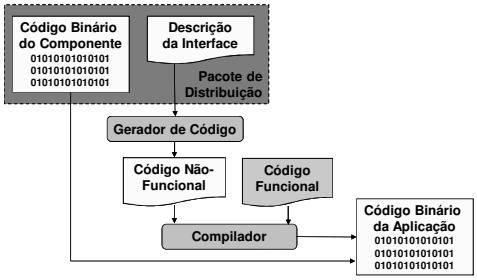
Desenvolvimento

- Desenvolvimento do Componente



Desenvolvimento

- Uso do Componente em uma Aplicação



Testes

- Softwares baseados em componentes precisam ser testados para garantir que são estáveis e corretos
- Fases de teste
  - Testes de unidade
  - Testes de integração
  - Testes de sistema

Testes

- Testes de unidade
  - Componentes são testados individualmente, antes de efetuar a composição
  - O desenvolvedor do componente pode executar testes estruturais, também chamados de caixa branca (*white box*), que exigem acesso ao cód. fonte para testar cada operação
  - Já quem reutiliza o componente pode apenas realizar testes funcionais, também chamados de caixa preta (*black box*), que consistem em aplicar entradas e observar se as saídas produzidas pelo componente são corretas

Testes

- Testes de integração
  - Visam verificar se subconjuntos dos componentes do sistema funcionam corretamente após o processo de composição
  - Em geral são testes funcionais (caixa preta)
- Testes de sistema
  - Analisam se o sistema completo funciona conforme especificado na fase de projeto
  - Simula a operação em ambiente real
  - Costumam ser realizados testes funcionais

Testes

- Plano de teste
  - Definir entradas possíveis e saídas esperadas
  - Verificar a conformidade das saídas e se o comportamento (tempo exec., etc) é aceitável
  - Registrar o ambiente de execução utilizado
- Documentação de teste do componente
  - Vital para que, ao fazer a composição, o desenvolvedor saiba se o componente foi testado em condições análogas e em ambiente compatível com aquele no qual será utilizado

Distribuição

- Componentes são distribuídos na forma de pacotes de distribuição (*packages*)
- Pacotes de distribuição podem conter:
  - Apenas um componente
  - Um *framework* (coleção) de componentes
  - Uma aplicação completa
- Um *package* contém:
  - O código executável do(s) componente(s)
  - Pode ter também descritores de componentes
  - Geralmente são compactados (.Zip ou .Jar)

Distribuição

- Em geral, o código executável é dependente da plataforma de execução
  - Neste caso, pode-se gerar executáveis para cada plataforma suportada e colocar no mesmo *package*
  - Na implantação, deve ser escolhida a versão do componente compatível com a plataforma local

Distribuição

- Algumas linguagens são multi-plataforma
  - Máquina virtual converte o código para instruções da máquina onde o componente é executado
  - Exemplos:
    - Java usa JVM (*Java Virtual Machine*)
    - Linguagens do .NET (C#, VB.NET, etc.) usam CLR (*Common Language Runtime*)

Distribuição

- O descritor do componente é usado por:
  - Clientes
    - Descobrir os serviços fornecidos pelo componente
    - Descobrir a forma de obtenção dos serviços
  - Ferramentas de composição e servidores de aplicação
    - Conhecer a versão, nº de série, dependências, etc.
    - Saber como instanciar, configurar e conectar os componentes

Distribuição

- Os descritores devem estar em uma linguagem para descrição de software
  - A maior parte dos modelos de componentes utiliza linguagens proprietárias para descrever o conteúdo dos pacotes de distribuição
  - Tendência do mercado é migrar para um padrão como o XML OSD (*Open Software Description*)

Distribuição

- Exemplo de descrição de *package* em XML OSD

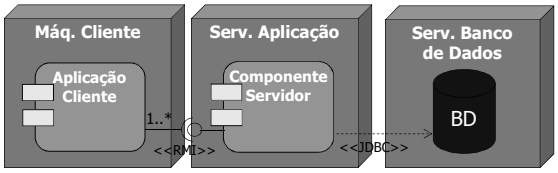
```
<SOFTPKG NAME="Crypt" VERSION="1,2,0,0">
  <TITLE>Crypt</TITLE>
  <AUTHOR>
    <COMPANY>MySoft Corporation</COMPANY>
    <WEBPAGE HREF="http://www.mysoft.com/crypt/" />
  </AUTHOR>
  <IMPLEMENTATION>
    <OS VALUE="WinXP"/>
    <OS VALUE="Win7"/>
    <CODE TYPE="Zip">
      <FILEINARCHIVE NAME="crypt.zip"/>
    </CODE>
  </IMPLEMENTATION>
  <IMPLEMENTATION>
    <IMPLTYPE VALUE="Java" />
    <CODE TYPE="Jar">
      <FILEINARCHIVE NAME="crypt.jar"/>
    </CODE>
  </IMPLEMENTATION>
</SOFTPKG>
```

Distribuição

- Componentes podem ser armazenados em repositórios, que são subdivididos em:
  - Repositórios locais centralizados: armazenam componentes de propósito geral; são geralmente inchados, dificultando a busca
  - Repositórios específicos: contêm componentes focados em um domínio de aplicação, inclusive componentes com as mesmas funcionalidades
  - Repositórios de referência: apontam para componentes contidos em outros repositórios, facilitando a busca ("páginas amarelas")

Implantação

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



Implantação

- Os componentes que disponibilizarão serviços remotamente devem ser implantados em servidores de aplicação
  - Servidores de aplicação servem como ambiente de execução dos componentes e servem como Contêiners
  - Servidores Web com extensões podem ser usados com este objetivo

Implantação

- *Packages* podem ser instalados no servidor
  - A partir da implantação no servidor, o componente está disponível para interagir com outros componentes no mesmo servidor ou em outros servidores da rede
  - Servidor verifica o descritor do pacote e seleciona a implementação do componente compatível com a sua plataforma

Implantação

- Servidores de aplicação devem fornecer:
  - Interface para localização dos componentes
  - Suporte para comunicação
  - Serviços adicionais de propósito geral
    - Segurança
    - Transações
    - Persistência
    - ...
  - Mecanismos para gerenciamento de aplicações

Implantação

- Servidores de aplicação gratuitos:
  - Glassfish (Sun/Oracle)
  - JBoss Application Server (JBoss/Red Hat)
  - Geronimo (Apache)
  - etc.
- Servidores de aplicação pagos:
  - WebSphere Application Server (IBM)
  - WebLogic Application Server (Oracle/BEA)
  - ColdFusion (Adobe)
  - etc.

Padrões e Produtos

- Padronização é necessária para que o mercado de componentes de software se desenvolva
  - Definição de interfaces, para permitir a composição entre componentes
  - Definição de modelos e arquiteturas, para permitir a interoperação entre componentes
- Ainda não existe um padrão definitivo para componentes de software

Padrões e Produtos

- Padrões podem coexistir, desde que:
  - Haja meios de interconexão de tecnologias
  - O mercado de cada tecnologia seja grande o suficiente para se manter ativo
  - Não haja um número grande de padrões
- A convivência entre padrões é normal em diversas áreas
  - Linguagens de programação
  - Redes
  - Telefonia
  - etc.

Padrões e Produtos

- Padrões podem ser:
  - De jure
    - Definido por um bureau de padronização reconhecido por lei (ex.: ISO, ANSI, ABNT)
  - De fato
    - Aceito como padrão pela indústria
    - Geralmente definido por um comitê sem poder legal (ex.: IETF, OMG, W3C, etc.)
  - De mercado
    - Proposto por alguma empresa que domina uma grande fatia do mercado

Padrões e Produtos

- Ainda não existe um padrão definitivo para componentes de software, mas apenas alguns candidatos a padrão
- São três os principais candidatos a padrão:
  - JavaBeans e Enterprise JavaBeans
    - Propostos pela Sun, baseados no Java
  - COM, DCOM, ActiveX, COM+ e .NET
    - Produtos da Microsoft
  - CORBA e CCM
    - Padrão de fato proposto pela OMG

Padrões e Produtos

- Modelo de Componentes da Oracle/Sun
  - JavaBeans
  - Enterprise JavaBeans (EJB)
- Estratégia da Oracle/Sun
  - Usar Java para integração com a Internet
  - Padrões de componentes baseados em Java
  - Permite discussão do padrão, mas mantém o controle sobre o copyright e sobre as decisões

Padrões e Produtos

- Modelo de Componentes da OMG
  - CORBA
  - CORBAServices
  - Modelo de Componentes CORBA (CCM)
- Estratégia da OMG
  - CORBA: permite independência de linguagem, de sistema operacional e de plataforma
  - Padrão de componentes baseado no CORBA
  - Busca a integração com outras tecnologias

Padrões e Produtos

- Produtos da Microsoft
  - COM/ActiveX e .NET
- Estratégia da Microsoft
  - Controla o padrão
  - Tem sempre Windows como base
  - Vem adotando padrões abertos (XML, SOAP, UDDI, WSDL) a partir do .NET

Padrões e Produtos

- Componentes são utilizados na construção das mais diversas aplicações
  - Sistemas operacionais
  - Processadores de texto e planilhas
  - Reprodutores de mídia
  - Navegadores e servidores Web
  - Sist. de informações geográfica
  - Sist. de informações gerenciais
  - Sist. de gerenciamento de bancos de dados
  - Sist. de comércio eletrônico
  - etc.