Software Baseado em Componentes

O que são componentes?

- Uma afirmação feita no final dos anos 90:
  - "The advent of component software may be the most important new development in the software industry since the introduction of high-level programming languages."
- Uau! Talvez seja exagero, mas, no mínimo, temos que investigar isso!

Qual é o problema?

- Duas classes (sem trocadilho!) de problemas
  - Pressões do negócio
  - Pressões tecnológicas

Pressões do negócio

- Steve Jobs fala: "Few companies have the luxury of reinventing themselves when they compete on Internet time"
  - "Internet time" é um novo conceito (1997-1998) e significa que as empresas devem implementar novos sistemas *muito* rapidamente
- Muitos sistemas de empresas devem migrar para Internet/Intranet/Extranet
- Novos tipos de sistemas devem ser desenvolvidos para usar a tecnologia como *business advantage*, dando um diferencial nos negócios
- Muitos sistemas devem mudar devido a mudanças nos negócios
  - Fusões e aquisições
- Resultado: tem *muito* mais software a fazer, *muito mais rapidamente*

- Mas há pressões para manter custos de IT (*Information Technology*) baixos
- Hoje (2001), de 4 a 6% da receita é gasta com TI! É altíssimo!

Pressões tecnológicas

- Evolução tecnológica força mudanças de sistemas
  - Muitas empresas acabam de fazer a transição para cliente/servidor e agora a palavra de ordem é "client/server is dead!"
    - Isto é, client/server em 2 camadas
  - Sistemas devem migrar para arquiteturas distribuídas N-tier
    - Para ter escala
    - Para ter acesso Internet mas com acesso a sistemas legados
      - Não podemos jogar o legado fora
    - Para juntar sistemas operando em plataformas diferentes
  - Poxa! Que saco! Mudar de novo! Com que tipo de tecnologia de desenvolvimento?
- O desenvolvimento de software ficou muito mais complexo nos últimos anos
  - Pelos motivos acima
  - Porque usuários querem funcionalidade mais sofisticada
- Problemas de complexidade
  - O desenvolvimento de muitos grandes sistemas tem fracassado recentemente
  - A figura mais citada: 80% dos projetos são fracassos!
- Resumindo: fazer sistemas de produção customizados do zero in-house:
  - É muito caro
  - Demora muito tempo

- Não produz boa qualidade

O que queremos?

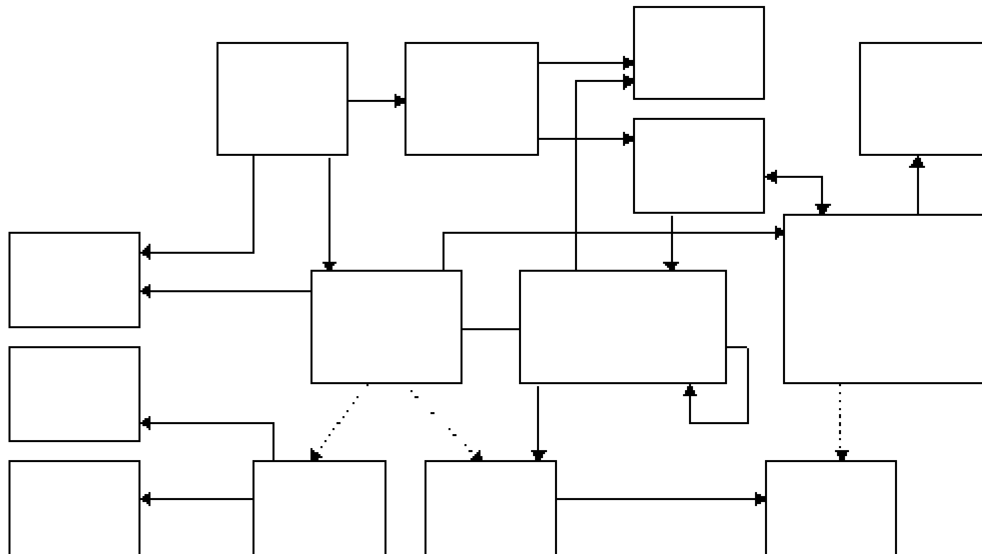
- O que grandes empresas, com grandes problemas de desenvolvimento de sistemas, querem?
- Na visão do usuário, software de qualidade:
  - Tem que satisfazer as necessidades do usuário
  - Tem que ser feito no prazo combinado
  - Tem que ser feito dentro do custo combinado
  - Tem que ter nível aceitável de defeitos
- Mas queremos ver a visão do desenvolvedor ...
- Melhor flexibilidade
  - Possibilitando satisfazer novos requisitos do negócio (=funcionalidade) rapidamente
- Melhor adaptabilidade
  - Possibilitando customizar uma aplicação para vários usuários, usando várias alternativas de *delivery* dos serviços da aplicação com impacto mínimo ao núcleo da aplicação
- Melhor manutenabilidade
  - Possibilitando atualizar uma aplicação, mas minimizando o impacto da maioria das mudanças
- Melhor reusabilidade
  - Possibilitando rapidamente montar aplicações únicas e dinâmicas
- Melhor aproveitamento do legado
  - Possibilitando reusar a funcionalidade de sistemas legados em novas aplicações
- Melhor interoperabilidade
  - Possibilitando integrar 2 aplicações executando em plataformas diferentes
- Melhor escalabilidade
  - Possibilitando distribuir e configurar a execução da aplicação para satisfazer vários volumes de

transação

- Menor tempo de desenvolvimento
  - Possibilitando viver em "Internet time" e com baixo orçamento
- Melhor robustez
  - Possibilitando ter soluções com menos defeitos
- Menor risco
  - Possibilitando tudo que falamos acima e ainda não se arriscar a ter projetos fracassados
- Resumindo: tudo que ISO 9126 caracteriza como "qualidade de software"
  - Funcionalidade, Manutenibilidade, Usabilidade, Eficiência, Confiabilidade, Portabilidade
  - E mais algumas coisinhas ...

Orientação a Objetos não está resolvendo?

- Muitos grandes projetos baseados em OO estão fracassando recentemente
- Aparentemente, OO não está à altura
- OO é apenas uma *Enabling Technology*
  - Benefícios não são automáticos mas dependem do uso correto da tecnologia
- Precisamos descobrir o que acrescentar a OO para melhorar as coisas
- Onde OO peca?
  - OO (sozinha) não produz software reutilizável automaticamente
  - OO (sozinha) não tem boa escalabilidade
  - OO (sozinha) não provê boa encapsulação (esconder informação)
- Isto pode ser visto com o resultado do uso de OO sem disciplina
  - Hyperspaghetti Objects (ver figura abaixo)



- Ocorre mais quando software cresce em tamanho
  - Um objeto pode referenciar qualquer outro
  - Referências demais não permitem arrancar um objeto e reutilizá-lo sozinho
  - Manutenção difícil, causando instabilidade
  - Tirar bugs introduz novos bugs
- Resultado: o programa está muito complexo e não gerenciável
- Onde OO peca mais?
  - OO (sozinha) não permite a construção de sistemas verdadeiramente extensíveis
    - Para ser verdadeiramente extensível, um sistema deve permitir a adição *tardia* de uma extensão sem necessitar de uma verificação global de integridade
    - Em outras palavras, para adicionar algo, somos obrigados a passar por compilação e/ou link edição, testes de integração, etc.
    - Queremos fazer mais ou menos como num sistema operacional
      - Aplicações estendem o SO mas não precisamos de um teste total de sistema
      - Temos problemas de instalação e configuração, apenas
  - O uso de herança em OO cria um acoplamento muito forte entre os pedaços

      - Por este motivo, a extensão de uma classe usando herança (de implementação) requer freqüentemente que se tenha o código fonte da classe sendo estendida
      - Muitos fornecedores de bibliotecas OO fornecem código fonte por este motivo

Então como fazer?

- Precisamos de novos *approaches*
- Exemplos
  - Frameworks
  - Componentes
- Cuidado! Houve promessas de Nirvana antes!
  - Mas hoje (2001), as pessoas ainda estão entusiasmadas
- Aqui, examinaremos Desenvolvimento de Software Baseado em Componentes
- Idéias básicas
  - Vamos examinar mercados bem sucedidos para ver o que eles fazem diferente do que é feito no mercado de software
  - Como mercados maduros dão boa relação qualidade/preço?
  - Mercado industrial ensina que a manufatura (carros, p. ex.) não tem muitos dos problemas da construção de software
    - Porque a manufatura usa Componentes Pré-Fabricados
  - Pense em carros, bicicletas, hardware (com circuitos integrados)
  - Por que usar componentes é bom?
    - A empresa não tem que fazer tudo
    - Ela se concentra no que ela é boa
    - Ela *compra* componentes de outros especialistas
    - É a boa e velha "terceirização"
    - Desta forma, pode-se focar melhor apenas no que falta

- Resultado:
  - Componentes reutilizáveis
  - Compre, não construa porque deve ser mais barato
  - Pare de escrever aplicações do zero cada vez que inicia um projeto
  - Muda a ênfase da programação (construção) para a composição (montagem)
    - É como usar bloquinhos Lego
- Veremos definições do que é um Componente adiante
  - Pense numa aplicação que lê o preço de ações de um *newsfeed* e que pode ser "ligado" a uma planilha
    - Este componente faz alguns cálculos e passa os resultados para um banco de dados
  - Outro exemplo conhecido: pequenos componentes como "Controls VBX" do Visual Basic (ou OCX, ou ActiveX)
- Precisamos de uma forma simples de dinamicamente conectar os componentes entre si mas *não* em tempo de compilação ou link-edição
  - Quero usar uma ferramenta visual para
    - Configurar os componentes
    - Interconectar os componentes (estabelecer associações)
  - Estou "programando" sem codificar
  - Também chamado de
    - Attribute programming
    - Visual programming
    - Interactive programming
    - Rapid Application Development (RAD)
- Isso leva ao conceito de "design time" (ou Assembly Time)
  - E, na realidade, a um novo processo de desenvolvimento
- Parte disso já está sendo feito há um certo tempo com Controls em VB e Delphi

- Mas componentes vão muito mais longe!
- Como Component-Based Development (CBD) ajuda a obter as qualidades de software da ISO 9126?
  - Cuidado! A tabela abaixo tem muitos argumentos de "marketing"
  - Na tabela, o que você acha está certo? Exagerado? Errado?

Funcionalidade	Uso de componentes pré-existentes permite entregar mais funcionalidade em menos tempo
Manutenabilidade	A estrutura modular de uma solução baseada em componentes permite a substituição de componentes individuais
Usabilidade	Substituição de componentes em tempo de execução permite boa customização. Uso de componentes padronizados uniformiza a interface GUI
Eficiência	Componentes podem mudar de plataforma para ganhar mais desempenho. A escalabilidade é maior com o uso de componentes atuando em paralelo para tratar a maior carga de informação
Confiabilidade	Componentes reutilizáveis já foram testado em outros contextos e são portanto mais robustos
Portabilidade	A especificação de um componente independe da plataforma. Reimplementar um componente para outra plataforma não deve afetar a arquitetura ou solução final

- Espera-se que a migração das aplicações para o uso de componentes ocorrerá na seguinte seqüência:
  - Em computadores clientes executando código "off-the-shelf", principalmente na interfaces gráficas
    - Como VB/Delphi, hoje
  - Em software customizado que executa em clientes
    - Vimos isso acontecer com applets, ActiveX em aplicações que executam em rede
  - Em software de servidor, que é mais estável do que software cliente
    - É o que está acontecendo agora (desde 1999)

O que são Componentes?

- Vamos agora detalhar as idéias
- Muitas definições foram oferecidas ao longo dos últimos anos (ver aqui)
- Apesar da confusão nas definições, podemos enxergar algo em comum
- Vamos iniciar com a definição mais geral de D'Souza
  - Componente geral: "Um pacote coerente de artefatos de software que pode ser desenvolvido independentemente e entregue como unidade e que pode ser composto, sem mudança, com outros componentes para construir algo maior."
- Baseando-se nesta definição, todas as coisas que seguem poderiam ser consideradas "componentes", pelo menos em espírito:
  - Arrastar um widget de GUI, tal como uma list box, e conectar as listas a uma fonte apropriada de dados do domínio do problema
  - Usar a mesma template de lista em C++ para implementar várias classes do domínio do problema através da especialização do parâmetro do template:

```
class Pedido {
    private: List<ItemDePedido> itens;
};
```

  - Usar os seguintes produtos *off-the-shelf* (produtos prontos): um pacote de calendário, um processador de textos e uma planilha, e montar esses componentes heterogêneos usando scripts que resolver um problema particular do domínio do problema
  - Usar um framework de classes, usando, por exemplo, componentes do pacote Swing da linguagem Java, para construir uma interface com o usuário para muitas aplicações e conectar os componentes gráficos a seus objetos de domínio de problema

- Usar um framework de alocação de recursos que modelar problemas variando da alocação de salas para seminários até o escalonamento de tempo de máquina para a produção de lotes de peças numa fábrica
- Usar construções pré-definidas de uma linguagem de programação numa infinidade de contextos:

```
for(...; ...; ...) {
    ...
}
```

- Usando essa definição, um componente pode incluir:
  - Código executável
  - Código fonte
  - Projetos (designs)
  - Especificações
  - Testes
  - Documentação
  - etc.
- Mas nós queremos falar apenas de componentes de implementação
- Definição de D'Souza, mais uma vez:
  - Componente de implementação: "Um pacote coerente de implementação de software que (a) pode ser desenvolvido independentemente e entregue como unidade; (b) tem interfaces explícitas e bem definidas para os serviços que oferece; (c) tem interfaces explícitas e bem definidas para os serviços que requer; e (d) pode ser composto com outros componentes, talvez após a customização de algumas propriedades mas sem modificar os componentes em si."
- Examinando esta definição (e as outras), podemos ver o que componentes têm de diferente comparados a bibliotecas ou outros artefatos de software
  - São 3 características básicas
- 1. Construção de aplicações por montagem
  - Essa é a diferença principal!

- Um componente deve permitir que todo o trabalho (ou quase) seja feito pela composição de pedaços existentes
- Isso significa que, no processo de desenvolvimento, novas etapas podem surgir
  - Design time (ou assembly time) para juntar componentes e montar aplicações
- A definição "oficial" da conferência sobre componentes em 1996 fala de "composição por terceiros". Por que isso é importante?
  - Um "terceiro" é alguém que não tem acesso a detalhes de construção e não possui o código fonte
  - Portanto, componentes podem ser modificados ao incluí-los na aplicação, mas sem ter código fonte
    - Componentes maximizam assim o "information hiding"
  - A modificação pode ser de atributos, por exemplo ("Attribute programming")
  - A modificação (configuração) do componente usa freqüentemente uma ferramenta visual ("Visual Programming")

- 2. Um componente explicita suas interfaces
  - Para ser plugável em vários contextos ("Plug-Replaceable"), temos que usar interfaces padronizadas
  - Uma interface é um contrato que especifica a funcionalidade do componente
  - Um componente tem duas interfaces:
    - Dos serviços que ele oferece (export interface)
    - Dos serviços que ele requer de outros componentes (import interface)
      - Esta interface explicita as "dependências de contexto"
      - O componente não pode depender do contexto onde ele vai atuar a não ser através dessa interface

- As linguagens de programação têm se concentrado apenas em especificar as interfaces de serviços oferecidos e não de serviços usados pelos componentes
- Se juntarmos um pequeno grupo de objetos funcionando conjuntamente e isolá-los com uma interface de componente, a complexidade (o acoplamento) vai diminuir, reduzindo a chance de Objetos Hyperspaghetti
- Para permitir que o ambiente (freqüentemente visual) de composição funcione, as interfaces devem ser auto-descritivas
  - Deve ser possível descobri-las em tempo de execução, sem ter tido conhecimento algum do componente antes
  - Isso afeta as opções de binding (detalhes adiante)

- 3. Um componente é uma unidade de empacotamento (packaging), entrega (delivery), implantação (deployment), e carga (loading)
  - Unidade de empacotamento
    - Inclui tudo dentro dele: a especificação de suas interfaces, legível em tempo de execução, implementação, imagens, outros recursos, ...
  - Unidade de entrega
    - Um componente não é entregue parcialmente quando vendido. Ele é entregue num formato que é independente de outros componentes com os quais ele venha a ser composto
  - Unidade de implantação
    - Um componente não é implantado parcialmente
    - Durante a implantação, seus atributos podem ser alterados (configurados)
    - Também significa que é a unidade de troca na manutenção (a aplicação inteira não precisa ser trocada)
  - Unidade de carga
    - Na aplicação final um componente é

carregado por inteiro: "Quero um desses!"

        - Não posso dizer: "Me esse componente, mas só carne magra!"

- De que consiste um componente, tipicamente?
    - Várias classes (código binário)
    - Definições de interfaces, usando algum mecanismo utilizável em tempo de execução
    - Possivelmente objetos (factories para criar objetos através do componente)
    - "Recursos" (podem ser arquivos de dados contendo formulários, strings, parâmetros, imagens, etc.) que são usados para configurar o componente
        - Alguns desses recursos podem ser mudados em tempo de execução

Componentes versus Objetos

- As diferenças geram controvérsia
- Instanciação
    - A instanciação de um componente não gera outro componente mas um "objeto" criado a partir de um protótipo (o componente)
    - Um componente freqüentemente é visto como um factory de objetos e não um objeto em si
- Em termos de linguagem OO, um componente pode conter vários objetos
    - Componentes têm granularidade maior, freqüentemente
- Componentes freqüentemente tratam de sua persistência
    - Objetos não fazem isso
- Objetos não são empacotados como componentes
    - Componentes são sujeitos à composição em tempo de design
- Se o componente obedece às suas interfaces, ele pode ser implementado em qualquer linguagem (mesmo

sem ser OO!) e rodar em qualquer plataforma

- Objetos são manipulados com linguagens OO apenas

Categorias de Componentes

- Podemos classificar componentes sob várias dimensões
- Escopo
  - Componentes de Especificação (diagramas, documentos)
  - Componentes de Implementação (classes, ...)
- Objetivo
  - Domínio (voltado ao problema)
  - Tecnologia (para o suporte, infra-estrutura)
- Abstração
  - Geral (aplicação em muitos domínios: horizontal)
  - Específico (aplicação em um único ou poucos domínios: vertical)
- Granularidade
  - Fina (um widget GUI)
  - Grossa (shopping-cart, agência bancária, fatura, contas a receber)
    - As oportunidades de reuso podem ser menores mas tais componentes são cruciais para melhorar a produtividade no desenvolvimento
- Localização
  - Cliente
  - Servidor (Middle tier)
- Serviços providos
  - Gerência de configuração e de propriedades
  - Notificação de eventos
  - Acesso a metadados e reflexão
  - Persistência
  - Controle de transação e concorrência
  - Segurança

- Licenciamento
- Controle de versão
- Autotestes
- Auto-instalação

comp-1 programa próxima