

ENG SW

Engenharia de Software

Prof. Marcelo Soares Pimenta
mpimenta@inf.ufrgs.br

Slides – arquivo 3

©Pimenta 2011

Engenharia de Software

Engenharia
de Software =

Desenvolver
Usar
Reusar
Integrar

n



Combinação de conhecimentos necessários
em **todo o ciclo de vida** do software
para a obtenção de **software de qualidade**

Linguagens
Ferramentas
Técnicas
Métodos
Modelos
Conceitos
Princípios
Equipes de
Pessoas

n

©Pimenta 2011

***Etapas do Ciclo de Vida do
Software: Objetivos, Principais
Abordagens e Artefatos***

©Pimenta 2011

***Etapa 1: Análise de Requisitos e
Engenharia de Requisitos***

©Pimenta 2011

Ciclo da Engenharia de Requisitos

• Os processos de engenharia de requisitos variam de uma organização para outra, mas a maioria dos processos de Engenharia de Requisitos é composta das seguintes atividades

- **Determinação:** identificação das fontes de informação; coleta, refinamento e integração de informações
- **Expressão:** representação das informações obtidas; representação das várias versões dos requisitos
- **Validação:** avaliação da informação recolhida e representada quanto à correção, completude, coerência para o(s) usuário(s)

©Pimenta 2011

Determinação

- Também denominada Elicitação, Levantamento ou Identificação de Requisitos
- Envolve stakeholders (os envolvidos e interessados no sistema) e não somente clientes e usuários
- Usa várias Técnicas para Coleta

Taxonomia de Técnicas

- **Técnicas Baseadas em Comunicação**
 - Comunicação Direta: Entrevistas
 - Comunicação Indireta: Questionários, Textos de requisitos
 - Prototipação
- **Técnicas Baseadas em Estudo**
 - Estudo de documentos e formulários
 - Estudo de Sistemas Existentes (Engenharia reversa)
 - Estudo Bibliográfico
- **Técnicas Baseadas em Observação**
 - Imersão, Etnografia, etc
 - Observação Direta (pessoal ou vídeo)
 - Observação Verbalizada
 - Observação seguida de diálogo

©Pimenta 2011

Expressão de requisitos

- Estabelecer um conjunto de requisitos consistentes e sem ambigüidades, que possa ser usado como base para o desenvolvimento do software
- Deve-se classificar os requisitos em: Funcionais e não funcionais
- Para esta atividade, alguns tipos de modelos podem ser construídos
- Um modelo é uma representação de alguma coisa do mundo real, uma abstração da realidade, e, portanto, representa uma seleção de características do mundo real relevantes para o propósito do sistema em questão

Expressão de requisitos

- Produção de um documento que possa ser revisado, avaliado e aprovado
 - Documento (Especificação) de Requisitos de software
 - Estabelece as bases para o acordo entre clientes
 - Estabelece o que o software deve fazer e o que ele não deve
 - Permite avaliação rigorosa dos requisitos e evita retrabalho
 - Fornece base para estimar custo, prazo e risco e para elaboração dos planos de verificação e validação
 - Normalmente escrito em linguagem natural, mas pode utilizar outras notações que auxiliem na compreensão do software, como: modelos conceituais para ilustrar o contexto do sistema, cenários (ou telas), principais entidades do domínio e fluxogramas

Validação de requisitos

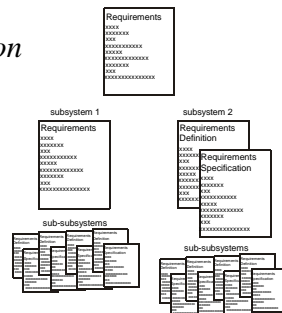
- O documento de requisitos deve ser validado pelos diferentes stakeholders (clientes e desenvolvedores)
- Processo de examinar os documentos para ter certeza que eles definem se o software faz o que o usuário espera dele
 - Revisões de requisitos
 - A maneira mais comum é a inspeção ou revisão do documento
 - O grupo de revisão (com pelo menos um representante do cliente) busca por erros, contradições, falta de clareza e desvios de padrões
 - As revisões podem acontecer após o término de um dos documentos de requisitos ou em qualquer ponto do processo

Types of Requirements Document

Two extremes:

- An informal outline of the requirements using a few paragraphs or simple diagrams
requirements *definition*
- A long list of specifications that contain thousands of pages of intricate detail
requirements *specification*

- Requirements documents for large systems are normally arranged in a hierarchy



Reviewing Requirements

– Each individual requirement should

- Have **benefits that outweigh the costs** of development
- Be **important** for the solution of the current problem
- Be expressed using a **clear and consistent notation**
- Be **unambiguous**
- Be **logically consistent**
- Lead to a system of **sufficient quality**
- Be **realistic** with available resources
- Be **verifiable**
- Be uniquely **identifiable**
- **Does not over-constrain the design** of the system

Managing Changing Requirements

Requirements change because:

- Business process changes
- Technology changes
- The problem becomes better understood

Requirements analysis never stops

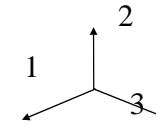
- Continue to interact with the clients and users
- The benefits of changes must outweigh the costs.
 - Certain small changes (e.g. look and feel of the UI) are usually quick and easy to make at relatively little cost.
 - Larger-scale changes have to be carefully assessed
 - Forcing unexpected changes into a partially built system will probably result in a poor design and late delivery
- Some changes are enhancements in disguise
 - Avoid making the system *bigger*, only make it *better*

© Lethbridge/Laganière

13
©Pimenta 2011

Dimensões da Engenharia de Requisitos

- Compreensão vaga, confusa e incompleta do sistema → Compreensão mais completa e clara do sistema
- Vários pontos de vista das diferentes pessoas envolvidas → Consenso negociado entre os envolvidos
- Representação informal → Representação mais rigorosa



©Pimenta 2011

Rastreabilidade (*traceability*)

- Objetivo: apresentar informações de conexão entre um requisito e sua origem, entre requisitos ou ainda entre o requisito e os artefatos (modelos) do sistema
- Rastreabilidade de origem (ou pré-rastreabilidade)
 - Ligação entre um requisito e o stakeholders que o propôs
- Rastreabilidade entre Requisitos
 - Ligação entre requisitos (de mesmo tipo ou não)
- Rastreabilidade entre requisitos e artefatos de sistema
- Rastreabilidade para FRENTE (*forward*) ou para TRÁS (*backward*)

©Pimenta 2011

Rastreabilidade

- Entre Requisitos e Origem
 - **principal**: o nome do papel (ou pessoa) que o validará
 - **autor**: o nome do papel (ou pessoa) que o criou/propôs
 - **fonte**: o nome do papel (ou da pessoa) que conhece mais detalhes sobre este requisito, ou o título do documento onde se encontra mais informações sobre ele

©Pimenta 2011

Rastreabilidade

- Entre Requisitos
 - Requisito de Stakeholder, Requisito de Sistema
- Relações entre Requisitos (R) e Artefatos (A)
 - **R Implementado por A (Impacto)**
 - **A Derivado de R**
 - **R/A Faz referência a R/A:** requisitos referenciados por ele
 - {R} é coberto por {A/R} (Cobertura)

©Pimenta 2011

Matriz de Rastreabilidade

Req. id	1.1	1.2	1.3	2.1	2.2	2.3	3.1	3.2
1.1		U	R					
1.2			U			R		U
1.3	R			R				
2.1			R		U			U
2.2								U
2.3		R		U				
3.1								R
3.2							R	

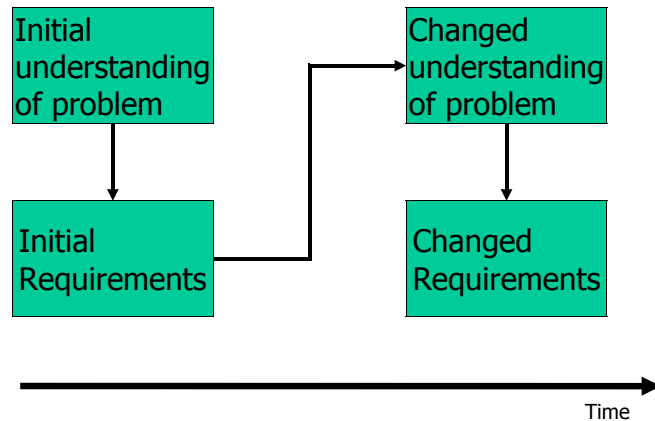
Gerência de Requisitos

- Gerência (ou Gestão) de Requisitos é o processo de lidar com as mudanças de requisitos que ocorrem durante a Engenharia de Requisitos e durante o desenvolvimento do sistema
- Requisitos são inevitavelmente incompletos e inconsistentes
 - Novos requisitos emergem durante o processo pois acontece um melhor entendimento do sistema e/ou as necessidades dos clientes mudam.
 - Diferentes pontos de vista sobre o sistema levam a diferentes requisitos, muitas vezes contraditórios e conflituosos

Mudanças nos Requisitos

- A prioridade dos requisitos muda durante o processo de desenvolvimento
- Clientes (business) e usuários (end user) podem estar na origem de requisitos conflituosos e há necessidade de negociação
- Contexto dos negócios e ambiente técnico mudam durante o desenvolvimento

Evolução dos Requisitos



Resumo

- Engenharia de Requisitos é o processo cíclico de determinar e manter um conjunto de requisitos, ou seja, elicitar, documentar, analisar, validar e gerenciar requisitos
- É fundamental adotar um processo e registrar cada estágio
- Registro geralmente através de documentos e/ou modelos
- Objetivo é obter especificações completas, precisas, claras e rigorosas de requisitos

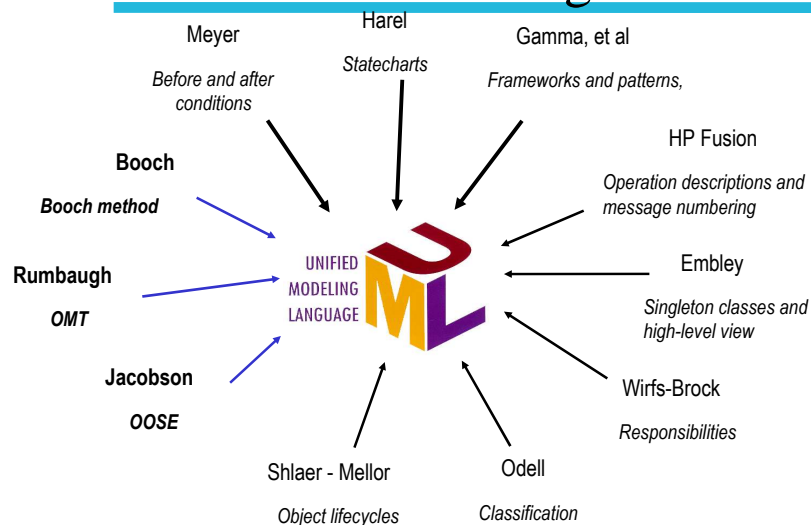
Introdução a AOO

- **OO**
 - Encapsulamento de atributos e métodos
 - Alta Coesão e Baixo acoplamento
 - Herança - especialização - polimorfismo
 - Agregação
 - Colaboração entre classes
- **Análise Orientada a Objetos (AOO)**
 - Entender e modelar aplicativos como conjuntos de classes colaborativas
- **UML** – diagramas e modelos

O que é a UML?

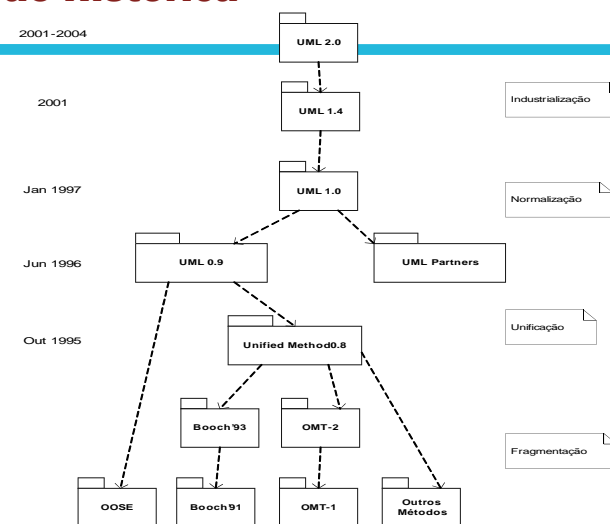
- UML = *Unified Modeling Language*
- UML é uma linguagem (notação com semântica associada) para
 - visualizar
 - especificar
 - construir
 - documentar
- UML não é uma metodologia
 - não diz quem deve fazer o quê, quando e como
 - UML pode ser usado segundo diferentes metodologias, tais como RUP (*Rational Unified Process*), FDD (*Feature Driven Development*), etc.
- UML não é uma linguagem de programação

Origens da UML



©Pimenta 2011

Visão Histórica



26

UML 2.0

- Há 2 documentos que descrevem UML 2.0
 - UML 2.0 Infrastructure – **metalinguagem** e elementos para definição e adaptação de UML
 - orientada a DESENVOLVEDORES DE FERRAMENTAS
 - UML 2.0 Superstructure, que descreve os elementos para modelagem de estrutura e comportamento dos sistemas
- Pacote UML2 inclui ainda:
 - Object Constraint **Language (OCL)**, linguagem para especificação de restrições entre modelos
 - Diagram Interchange (XMI) para intercâmbio de modelos UML entre ferramentas
- “ For the normal user, UML 2.0 does not turn the previous versions of UML upside down, but represents an improvement on existing concepts. It is probably wise to use UML 2.0 for future models. On the other hand, it should be possible to continue using existing constructs and models based on earlier UML versions.”

©Pimenta 2011

Tipos de Elementos Básicos (I)

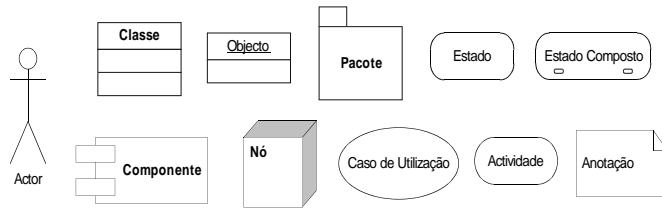
- Estrutura de Conceitos
 - Conjunto variado de notações, as quais podem ser aplicados em diferentes domínios de problemas e a diferentes níveis de abstracção.
 - (1) “**coisas**” ou elementos básicos através dos quais se definem os modelos.
 - (2) **relações**, que relacionam elementos
 - (3) **diagramas**, que definem regras de agrupamento de elementos

28

Tipos de Elementos Básicos (II)

■ Principais elementos da estrutura

- Classes, Objectos, Interfaces, Casos de Utilização, Actores, Colaborações, Componentes, Artefactos e Nós.
- Elementos de especificação de comportamento (estados e actividades)
- Elementos de agrupamento (pacotes e fragmentos)
- Anotações e restrições



29

Tipos de Diagramas Principais (I)

■ Diagramas

- Possibilidade de agrupar elementos básicos e as suas relações de uma forma lógica ou que estruturalmente faça sentido.
- “Nenhum modelo é suficiente por si só. Qualquer sistema não-trivial é melhor representado através de um pequeno número de modelos razoavelmente independentes”
- Princípio da REDUNDÂNCIA CONTROLADA
- Três categorias de diagramas : Comportamento, Interação e Estrutura.

30

Tipos de Diagramas Principais (II)

■ Diagramas de Comportamento

- Demonstram os aspectos comportamentais e de reação do sistema ou da lógica de negócio do projeto.
 - Actividade, Estados, Casos de Uso e todos os de interação.

■ Diagramas de Interação

- Sub-conjunto de diagramas de comportamento que enfatizam a interação entre os objectos.
 - Comunicação, Interação entre objectos, Sequência, Visão geral da interacção e Temporal.

■ Diagramas de Estrutura

- Especificação estrutural dos elementos independente do tempo.
 - Classes, Estrutura Composta, Componentes, implantação, Objetos e Pacotes.

31

Modelos UML: por quê?

- Por que precisamos de modelos e diagramas?
 - Estruturar processo de solução de um problema
 - Explorar múltiplas soluções (sem implementá-las)
 - Permitir abstrações para gerenciar complexidade e ocultar detalhes
 - Diminuir riscos de cometer erros
- Objetivos de modelos:
 - Registrar o andamento do projeto DURANTE o projeto
 - permite organização de idéias para reflexão
 - Documentar o projeto (registro para APÓS o projeto)
 - Apoiar a Comunicação entre membros da equipe e usuários
 - membro-membro, membro-usuário

UML e esta disciplina

- You can model 80% of most problems by using about 20% UML

- You learn those 20%

For the rest, see the official UML definition (www.uml.org)

- UML is a set of notations, not a single methodology
We will use Larman's viewpoint methodology

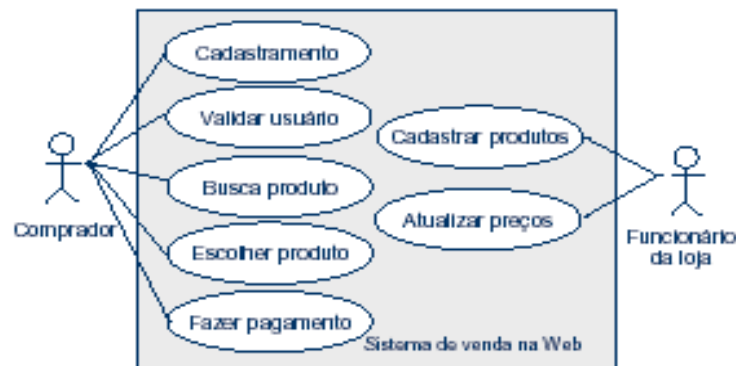
Larman, C. Usando UML e Padrões, Bookman, 2007 (3a ed)

UML First Pass

- Use case diagrams (na video conferência)
Functional behavior of the system as seen by the user.
- Class diagrams (na video conferência)
Static structure of the system: Objects, Attributes, and Associations.
- Sequence diagrams (nas atividades assíncronas)
Dynamic behavior between actors and system objects.
- Statechart diagrams (nas atividades assíncronas)
Dynamic behavior of an individual object as FSM.
- Activity diagrams (nas atividades assíncronas)
Dynamic behavior of a system, in particular the workflow, i.e. a flowchart.

These are not a sequence of steps!!!!

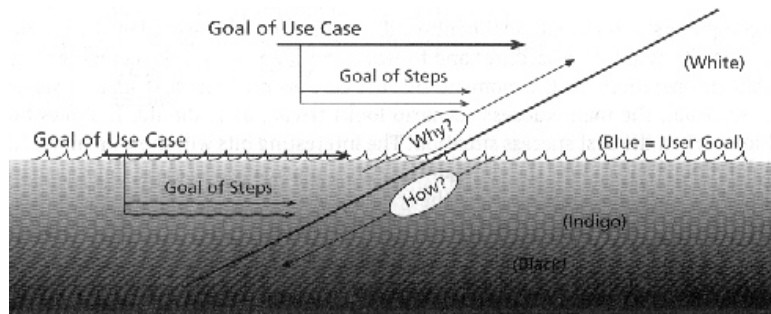
Caso de uso para expressar requisitos funcionais



Descrição Textual de Casos de Uso Preliminares

- Descrição breve (2 ou 3 sentenças)
- Útil durante planejamento e determinação do escopo inicial do projeto
- Define fronteiras típicas do sistema: responsabilidades internas e ambiente externo (atores)
 - Ex. de Escopo do Sistema: Caixa automático *standalone* Guichê *versus* sistema integrado com computador central
- Categorias de casos de uso
 - Prioritários : Mais críticos e mais frequentes
 - Secundários: Menos importantes, raros
 - Opcionais: Podem não ser considerados

Gráfico sobre Níveis de Casos de Uso



Fonte: Cockburn, A. *Escrevendo Casos de Uso Eficazes*, Bookman, 2004

©Pimenta 2011

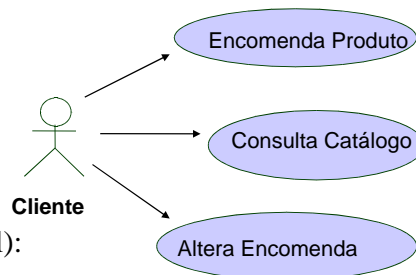
Tipos de casos de uso

- Casos de uso são *informais*
- Podem ser usados a vários níveis de abstração
- Conteúdo pode ser adequado às necessidades da aplicação
- **Tipos de casos de uso**
 - *Preliminar (ou alto-nível)*
 - Conceitual, abstrato e pouco detalhado (independente de implementação)
 - usados na especificação de requisitos e delimitação de escopo no início da análise
 - *Essencial (ou expandido)*
 - Conceitual (independente de implementação)
 - Detalhado em termos de funcionalidade
 - *Real (ou concreto)*
 - *Concreto* (dependente de tecnologia)
 - Detalhado em termos de funcionalidade, comportamento (operação) e referências a interface

©Pimenta 2011

Caso de Uso Preliminar

Diagrama de Caso de Uso Preliminar
(trecho mostrando apenas 1 ator)



- Formato Textual Básico (Alto Nível):

Caso de Uso: Encomenda Produto

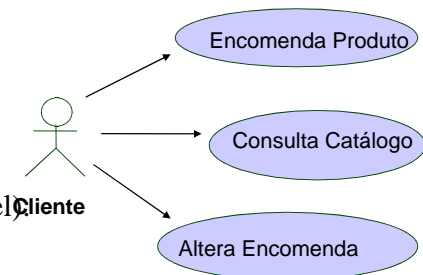
Atores: Cliente

Descrição: Cliente solicita produtos presentes no catálogo, fornecendo seu código e a quantidade desejada. Cliente escolhe forma de entrega e de pagamento. Lista de produtos encomendados é exibida ao Cliente que confirma (ou não) a encomenda.

©Pimenta 2011

Caso de Uso Preliminar

Descrição textual:



- Formato Textual Básico (Alto Nível)

Caso de Uso: Altera Encomenda

Atores: Cliente

Descrição: Cliente altera dados da encomenda efetuada previamente por ele. Os dados que podem ser alterados são: endereço de entrega do produto, produto(s) encomendados, quantidade(s) de produto(s), DESDE QUE a encomenda não esteja sendo entregue. Nova encomenda é exibida ao Cliente e este confirma (ou não) a (nova) encomenda. Se não confirmar, continua valendo a antiga encomenda.

©Pimenta 2011

Descrição textual de caso de uso essencial

Cliente Retira Cédulas e Recibo

Encerra Saque

©Pimenta 2011

Casos de uso alternativos

- Percorrer o caso de uso básico em busca de
 - ações alternativas ou optativas (casos de uso alternativos)
 - possibilidades de erro, obstáculos e singularidades (casos de uso de exceção: **E se ...?**)
 - ações que se repetem em diversos cenários (casos de uso extensão)

©Pimenta 2011

Exemplo de Identificação de Caso de Uso Alternativo

ID da singularidade	ID do episódio onde a singularidade acontece	ID da ação do episódio onde a singularidade acontece	Problema	Singularidade (Causa do problema)	Ações (Defensivas ou Corretivas)
S09	EP3	3	Valor de saque incorreto	Cliente entrou com valores incorretos	(C) Notificar o cliente e permitir escolher o valor, novamente
S10	EP3	3	Valor de saque não escolhido	Cliente não entra com o valor de saque a tempo (<i>timeout</i>)	(C) Notificar o cliente e retornar ao estado inicial, à pré-condição do episódio
S11	EP3	4	Dinheiro insuficiente	O valor do saque escolhido excede a quantidade disponível no caixa eletrônico	(C) Informar o cliente da quantidade disponível e permitir que escolha esta quantidade
S12	EP3	5	Saldo Insuficiente	O valor do saque escolhido excede o saldo da conta do cliente	(D) Exibir o saldo da conta do cliente (C) Notificar o cliente e permitir escolher o valor, novamente
S13	EP3	8	Transação não confirmada	Cliente não confirma a transação a tempo (<i>timeout</i>)	(C) Notificar o cliente e retornar ao estado inicial, à pré-condição do episódio

©Pimenta 2011

Caso de uso real : exemplo

Ação do Ator

1. Fornecedor seleciona 1 ou mais produtos para executar o algoritmo (Figura 1). Pode ser selecionado todos os produtos ao mesmo tempo clicando-se no botão acima dos *check box*.

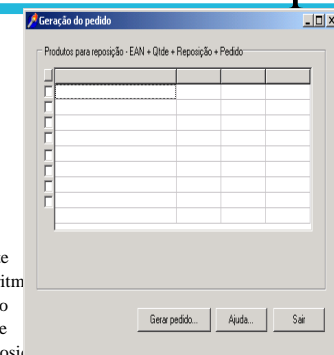
2. Fornecedor clica no botão "Executar...". Se nenhum produto estiver selecionado, é exibida uma mensagem de erro ao usuário.

.....(continua)

Reação do Sistema

3. Para cada produto anteriormente selecionado, é executado um algoritmo de reposição, que verifica, de modo geral, se a quantidade em estoque atual é menor que o Ponto de Reposição. Para todos os produtos é inserido na base de dados um histórico da execução do algoritmo

.....(continua)



©Pimenta 2011

Ex. de Análise – passo 1

- Casos de Uso Essenciais (descrição textual)

Use Case: Select a Product	
Summary: Our customer visits the store and selects a product category, after which she is presented with a list of products. Next, she chooses a product and quantity, which is added to her shopping cart. After this, she can either keep shopping, or proceed to the checkout.	
User Action	System Response
1. Select a product category.	2. Display a descriptive list of products and their price within the selected category.
3. Choose a product and quantity.	4. Put the product and quantity in the shopping cart. 5. Display the contents of the shopping cart with prices and total cost. Give the customer the opportunity to shop some more or check out.

©Pimenta 2011

Ex. de Análise – passo 2

- Identificar conceitos

User Action	System Response
1. Select a <u>product category</u> .	2. Display a <u>descriptive list of products</u> and their <u>price</u> within the <u>selected category</u> .
3. Choose a <u>product</u> and <u>quantity</u> .	4. Put the <u>product</u> and <u>quantity</u> in the <u>shopping cart</u> . 5. Display the <u>contents of the shopping cart</u> with <u>prices</u> and <u>total cost</u> . Give the <u>customer</u> the opportunity to shop some more or check out.

©Pimenta 2011

Ex. de Análise – passo 3

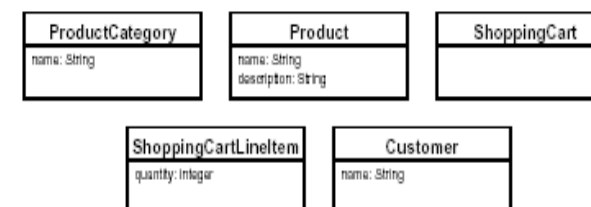
- Definir o que é conceito e o que é atributo

Noun Phrase	Concept Name
product category, selected category	ProductCategory
descriptive list of products	ProductList
product	Product
quantity	Quantity
price, prices, total cost	Price
shopping cart, contents of the shopping cart	ShoppingCart
customer	Customer

©Pimenta 2011

Ex. de Análise – passo 4

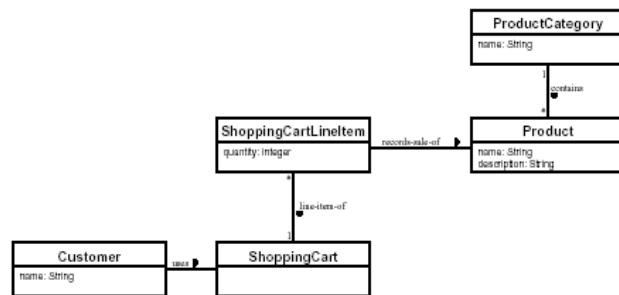
- Desenhar Diagrama Conceitual Preliminar (diagrama de classes simplificado)



©Pimenta 2011

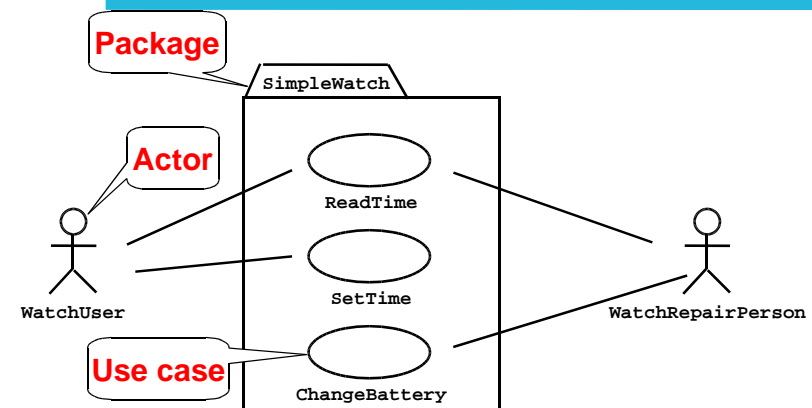
Ex. de Análise – passo 5

- Identificar associações



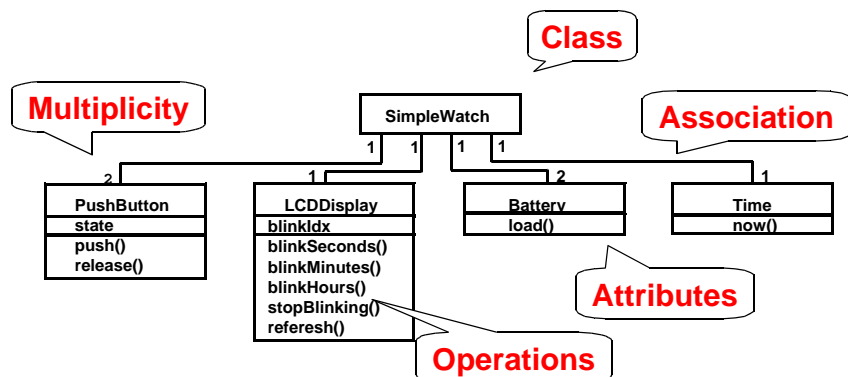
©Pimenta 2011

UML First Pass: Use Case Diagrams



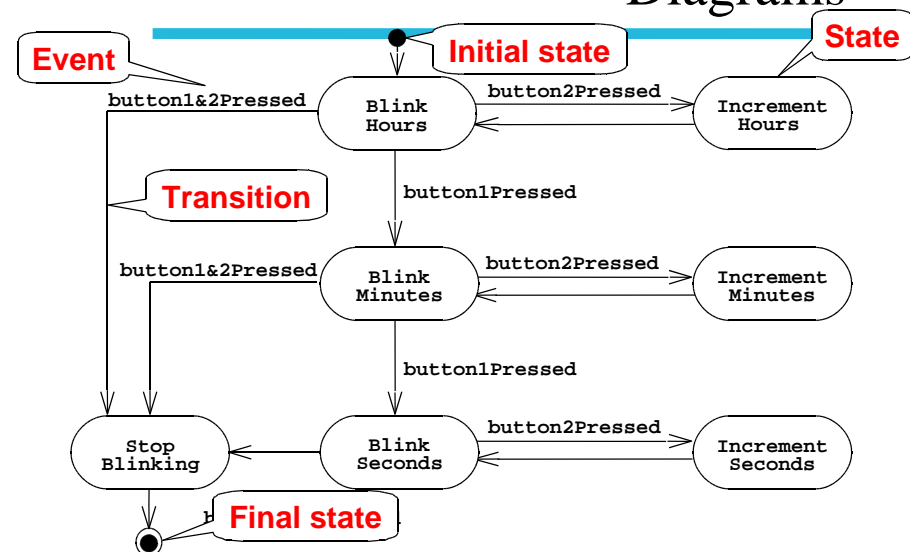
Use case diagrams represent the functionality of the system from user's point of view

UML First Pass: Class Diagrams



Class diagrams represent the structure of the domain or system

UML First Pass: Statechart Diagrams



Other UML Notations

UML provides other notations.

- Implementation diagrams
 - Component diagrams
 - Deployment diagrams
 - Oriented to System Design
- Object Constraint Language (OCL)
 - Part of UML package, a standard way to state logical rules formally

Summary

- UML provides a wide variety of notations for representing many aspects of software development
 - Powerful, but complex language
 - Can be misused to generate unreadable models
 - Can be misunderstood when using too many exotic features
 - We concentrate on a few notations:
 - Functional model: use case diagram
 - Object model: class diagram
- More about Dynamic model: sequence diagrams, statechart and activity diagrams

Etapa: Projeto

Projeto OO

- Projeto OO
 - Definir Casos de Uso Reais
 - Definir Diagramas de Interação (Seqüência ou Colaboração)
 - Refinar Arquitetura do Sistema
 - Definir Diagramas de Classe de Projeto
 - Classes de Persistência, de Interface, de Administração
 - Definir Interface com Usuário: unidades de apresentação (telas) e relatórios (Design de Interação)
 - Definir Esquema de BD (Projeto de BD)

Resultado da análise

Questões respondidas	Modelos usados
Quais são os <i>processos</i> do domínio de problema?	Casos de uso(UML)
Quais são os <i>conceitos</i> , qual é o vocabulário, com se relaciona?	Modelo Conceitual (Diagrama de classes UML simplificado)
Quais são os <i>eventos externos</i> e as operações do sistema em computador?	Descrição Textual dos Casos de Uso
<i>O que</i> as <i>operações</i> do sistema em computador devem executar?	Descrição Textual dos Casos de Uso

©Pimenta 2011

Projeto OO

- Objetivo
 - Definir a *arquitetura geral* da aplicação
 - identificar as *classes de software* que serão implementadas
 - inclui definir *atributos* e *métodos* e posicioná-los nas classes
- Envolve
 - definir os *métodos* das classes de domínio de problema
 - usar *diagramas de colaboração*
 - introduzir classes de implementação (*interface*, *banco de dados*, *computacionais*)
 - padrões de projeto (*design patterns*)
 - casos de uso *reais* (não abstratos, como na análise)

©Pimenta 2011

Passos do projeto

1. Construir casos de uso reais e esboçar a interface do sistema
2. Definir a arquitetura geral da aplicação (incluindo camadas como interface homem/computador, persistência)
3. Definir os métodos das classes domínio de problema
 - Construir diagramas de colaboração
4. Construir o diagrama de classes a implementar
 - Incluir classes de gerência de dados e outras classes computacionais
 - Limitar associações
 - Revisar especializações

©Pimenta 2011

Conceitos de Orientação a Objetos

- Decomposição algorítmica versus Decomposição OO
- Encapsulamento
- Associações entre objetos
- Agregação

©Pimenta 2011

Objeto - SW e domínio de problema

- Objeto \Rightarrow **artefato de software**
não: coisa, objeto de uma organização
- Mas
base do paradigma OO (orientação a objetos) é
um objeto (de **software**)
representa
uma coisa, um **objeto** do **domínio de problema** da aplicação
- Baixo “gap” semântico
Idéia conhecida de modelagem de dados:
Conceito de *entidade*

©Pimenta 2011

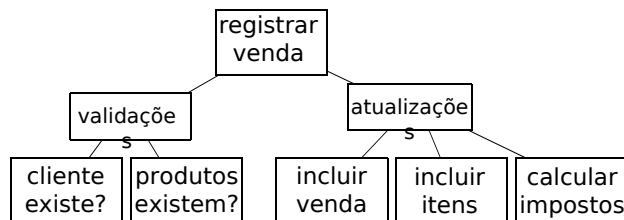
Decomposição de software

- Problema
Como tratar a complexidade inerente a sistemas de software?
- Solução clássica da Engenharia de Software:
“*divide et impera*” (dividir para conquistar)
- Decomposição em sistemas complexos de software:
 - Dividir o sistema em partes menores
 - Cada parte pode ser refinada independentemente das demais
 - Cada parte pode ser compreendida independentemente das demais
- Há vários tipos de decomposição:
 - decomposição algorítmica (“estruturada”)
 - decomposição OO

©Pimenta 2011

Decomposição algorítmica

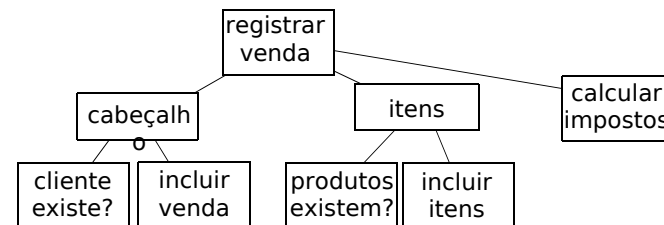
- Projeto e programação estruturada
 - O *algoritmo* é decomposto de forma “top-down”



©Pimenta 2011

Decomposição algorítmica

- Outra decomposição
- Qual é melhor?



©Pimenta 2011

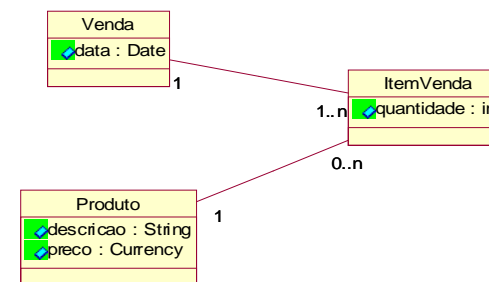
Decomposição algorítmica

- Qual é melhor?
- Não há uma única forma de decomposição estruturada
- Software resultante depende do projetista
 - Facilidade de manutenção
 - Facilidade de compreensão

©Pimenta 2011

Decomposição OO

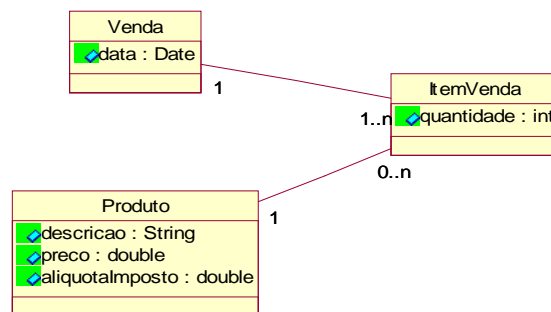
- Decompor o sistema de acordo com conceitos abstratos encontrados no problema
- Sistema é visto como uma série de agentes autônomos (os objetos) que colaboram entre si para atingir um objetivo.



©Pimenta 2011

Decomposição OO

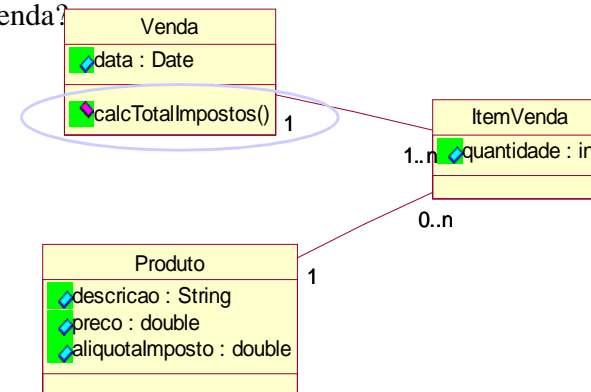
- Onde colocar o cálculo do total de vendas?



©Pimenta 2011

Decomposição OO

- Onde colocar o cálculo do total de impostos de uma venda?



©Pimenta 2011

CalcImpostos – implementação estruturada

```
public double calcTotalImpostos()
{
    double total = 0;
    ListIterator cursorLista =
osItensVenda.listIterator();

    while (cursorLista.hasNext())
    {
        total = total
            + (cursorLista.Next().aVenda.preco *
                cursorLista.Next().aVenda.aliquota
*
                cursorLista.Next().quantidade);
    }
}
```

©Pimenta 2011

Cálculo de impostos – implementação OO

```
public class Venda
{
    public Date data;
    public List osItensVenda;

    public double calcTotalImpostos()
    {
        double total = 0;
        ListIterator cursorLista =
osItensVenda.listIterator();
        while (cursorLista.hasNext())
        { total = total +
cursorLista.Next().calcImposto();
        }
    }
}
```

©Pimenta 2011

Cálculo de impostos – implementação OO

```
public class ItemVenda
{
    public int quantidade;
    public Venda aVenda;
    public Produto oProduto;

    public double calcImposto() {
        return quantidade *
oProduto.calcImposto();
    }
}
```

©Pimenta 2011

Cálculo de impostos – implementação OO

```
public class Produto
{
    public String descricao;
    public double preco;
    public float aliquotaImposto;
    public ItemVenda theItemVenda[];

    public double calImposto() {
        return preco * aliquota;
    }
}
```

©Pimenta 2011

decomposição algorítmica VS decomposição OO

- Decomposição OO
 - Código está particionado de acordo com os objetos manipulados
- Exemplo
 - Procedimento de cálculo de impostos
 - Um procedimento na decomposição algorítmica
 - Vários procedimentos na decomposição OO
- Vantagem paradigma OO
 - Manutenção e reuso
 - Sistemas menores, através de reuso de componentes
 - Maior estabilidade por estar mais próximos da realidade modelada
 - O que ocorre se o imposto não está informado no produto, mas em uma tabela separada de alíquotas de imposto?

©Pimenta 2011

decomposição algorítmica VS decomposição OO

- Qual é a forma “certa” de fazer a decomposição?
- São ortogonais: não é possível usar ambas em paralelo
- Com qual começar?
 - Abordagens OO baseiam-se em começar com decomposição OO

©Pimenta 2011

Encapsulamento

TRegistradora
<i>public</i> estadoOper: Estado <i>public</i> posicao: Posicao <i>public</i> saldo: valor
<i>public</i> abrir() <i>public</i> fechar() <i>private</i> acionarGaveta()

Encapsular – esconder
implementação de
comportamento

método privado
usado na implementação de
abrir() e fechar()

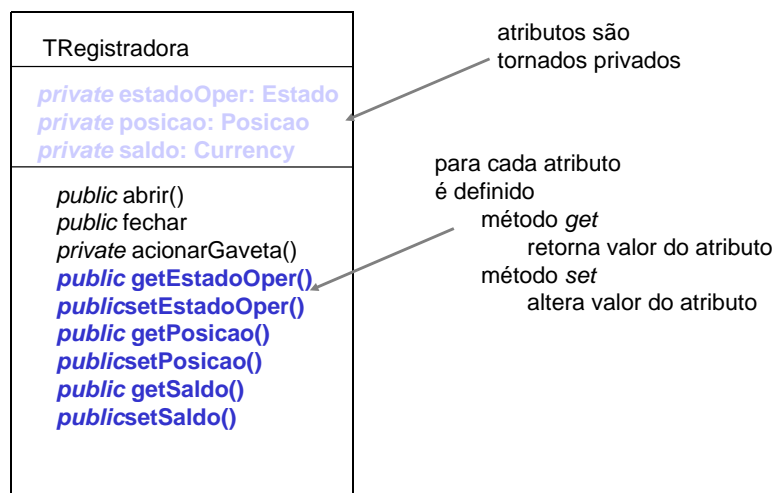
©Pimenta 2011

Encapsulamento

- Encapsulamento também é usado para
 - esconder a implementação interna do estado
- É considerada boa prática em programação OO
 - tornar cada atributo *privado*, impedindo acesso a ele
 - para cada atributo, construir
 - método *acessador* (método “get”)
 - tem por função devolver o valor do atributo
 - método *alterador* (método “set”)
 - tem por função alterar o valor do atributo
- Métodos “get” e “set” não aparecem na análise e projeto
- Gerados automaticamente por muitas ferramentas CASE

©Pimenta 2011

Encapsulamento



©Pimenta 2011

Passos do projeto

1. Construir casos de uso reais e esboçar a interface do sistema
2. Definir a arquitetura geral da aplicação (incluindo camadas como interface homem/computador, persistência)
3. Definir os métodos das classes domínio de problema
 - Construir diagramas de colaboração
4. Construir o diagrama de classes a implementar
 - Incluir classes de gerência de dados e outras classes computacionais
 - Limitar associações
 - Revisar especializações

©Pimenta 2011

Arquitetura de aplicações

- Apresentação:
 - Fornecimento de serviços, exibição de info (p.ex windows ou HTML), tratamento de solicitações do usuário (clicks, teclas), requisições http, chamadas em linhas de comando
- Domínio:
 - Lógica de negócio que é o real propósito do sistema
- Fonte de Dados:
 - Comunicação com o banco de dados, sistemas de mensagens, gerenciadores de transações, outros pacotes

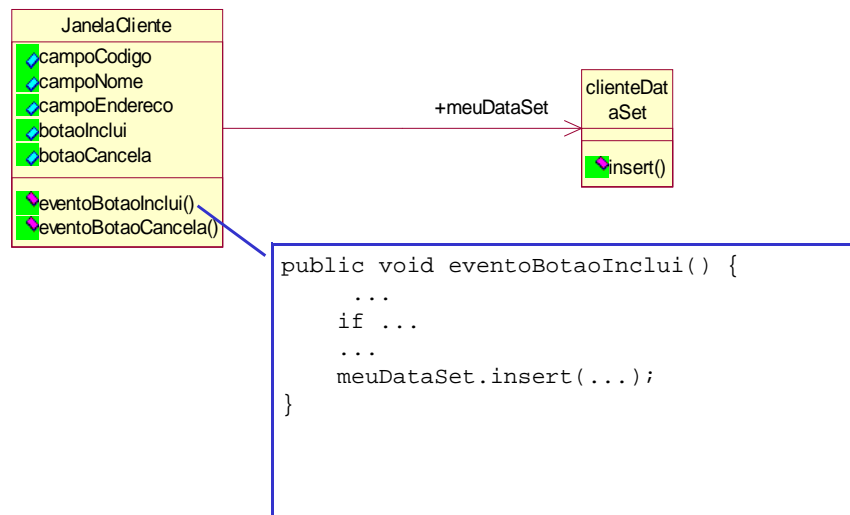
©Pimenta 2011

Arquitetura em dois pacotes

- Aplicação é dividida em dois grandes pacotes:
 - Classes de **interface**
 - Classes de **persistência** que encapsulam as tabelas do SGBD relacional:
 - Para cada tabela e visão é definida uma classe na linguagem OO
 - Classe oferece os serviços do SGBD (incluir, excluir linhas, alterar atributos, buscar linhas)

©Pimenta 2011

Arquitetura em dois pacotes



©Pimenta 2011

Arquitetura em dois pacotes (avaliação)

- Vantagem sobre a solução anterior
 - Aplicação fica melhor modularizada
 - Aplicação fica menos dependente da API do SGBD em questão
- Problema:
 - Objetos referentes ao domínio do problema não aparecem
- Ponto positivo:
 - Aplicação pode ser, em grande parte, gerada automaticamente por ferramenta visual

©Pimenta 2011

Arquitetura em três pacotes

- Aplicação é organizada em (pelo menos) três pacotes:
 - Classes de *interface homem/máquina* (classes HM)
 - Classes de *gerência de dados* (classes GD)
 - Classes do *domínio do problema* da aplicação (classes DP)

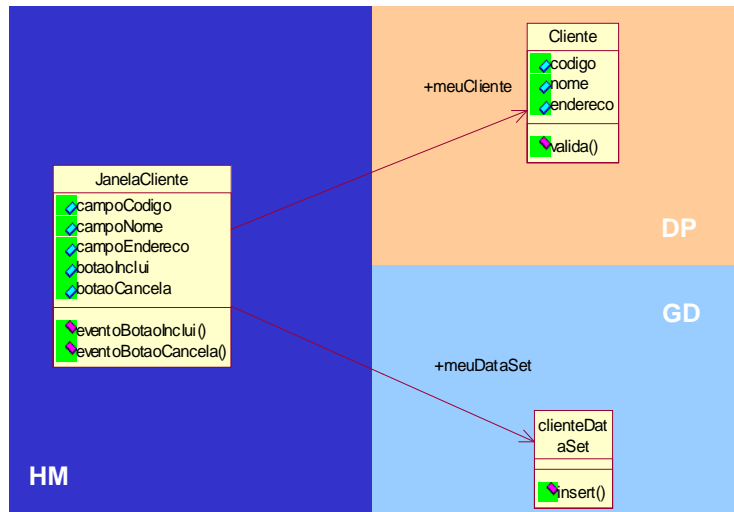
©Pimenta 2011

Pacotes da arquitetura

- Classes HM são construídas a partir das classes de interface (janelas, botões, campos, listas, ...) da biblioteca oferecida pela ferramenta em uso
 - Métodos desta classe tem comportamento muito simples, restringindo-se a chamar classes do domínio do problema
- Classes DP representam os objetos do domínio do problema dentro da aplicação
 - Seu comportamento reflete o comportamento dos objetos modelados
 - Recebem e enviam mensagens das classes HM
 - Usam objetos GD para armazenamento de dados
- Classes GD encapsulam detalhes do SGBD em uso

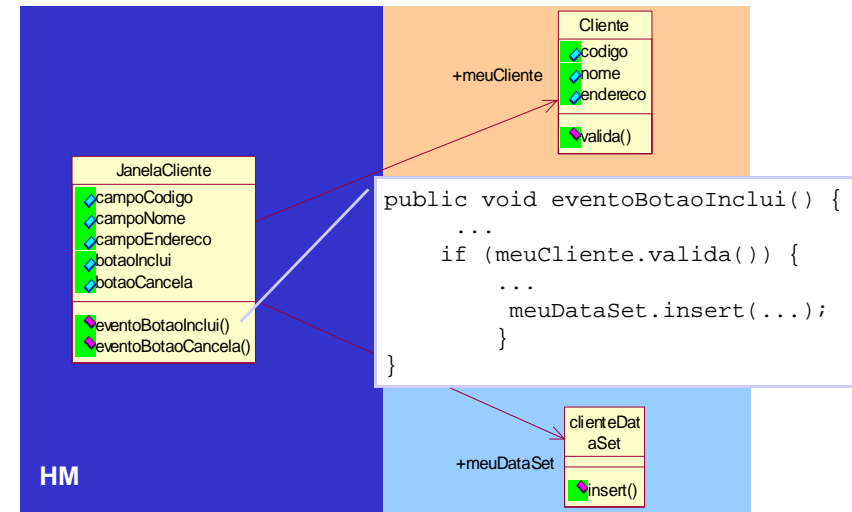
©Pimenta 2011

Componentes da arquitetura de três pacotes



©Pimenta 2011

Componentes da arquitetura de três pacotes



©Pimenta 2011

Arquitetura em três pacotes (avaliação)

- Aplicação é organizada de forma OO
 - Objetos dentro da aplicação correspondem a objetos do domínio de problema:
 - Manutenção mais fácil
 - Possibilidade de reuso e extensão
- Mudanças no ambiente de implementação (tipo de interface gráfica, tipo de SGBD) tem repercussão menor
 - É possível usar uma outra biblioteca de classes de interface ou um outro SGBD relacional, sem ter que rescrever toda aplicação

©Pimenta 2011

Arquitetura em três pacotes (avaliação)

- Aplicação pode ser distribuída:
 - Arquitetura “3-tier”:
 - cliente
 - servidor de aplicações
 - servidor de banco de dados
- Problema
 - Geração automática de aplicações dificultada
 - Arquitetura mais complexa: exige mais tempo de desenvolvimento
 - Analistas e programadores têm que “pensar OO”

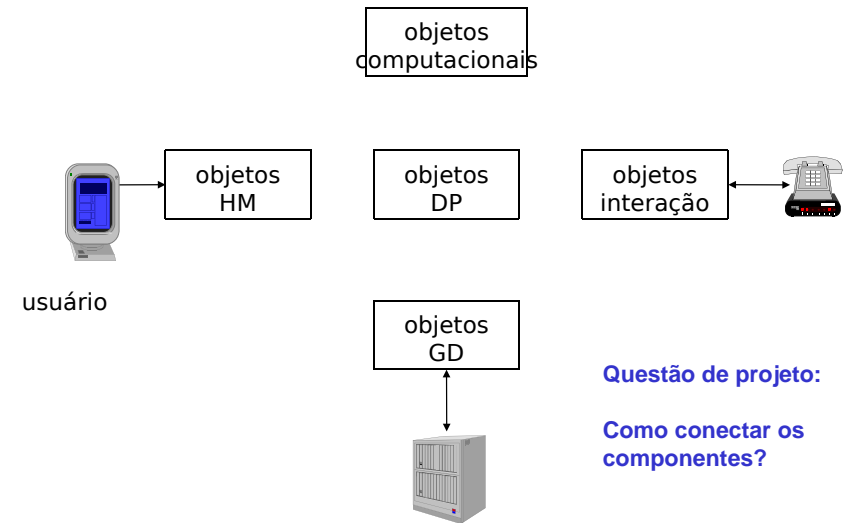
©Pimenta 2011

Arquitetura genérica de aplicações

- Aplicação é organizada nos seguintes pacotes:
 - Classes de interface homem/máquina (classes HM)
 - Classes de gerência de dados (classes GD)
 - Classes do domínio do problema da aplicação (classes DP)
 - Classes computacionais
 - Classes de interação
- Classes **computacionais** são classes que implementam objetos freqüentemente usados dentro de programas (estruturas de dados, códigos, etc.)
- Classes de **interação** são classes que implementam a comunicação com outros sistemas (envio ou consulta de dados a um outro equipamento) ou classes que implementam a comunicação com equipamentos (coletores de dados, sensores/atuadores)

©Pimenta 2011

Comunicação entre componentes da arquitetura



©Pimenta 2011

Almoxarifado – casos de uso reais(1)

Caso de Uso Registrar OC

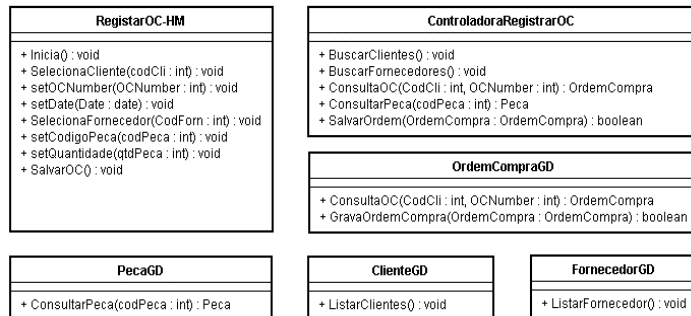
©Pimenta 2011

Registrar OC- Caso de uso real (2)

Ação do Ator	Resposta do Sistema
1. Este caso de uso inicia quando o ator recebe a cópia da Ordem de Compra do cliente e acessa o site de registro de ordem de compra.	2. Carrega no combo A os clientes cadastrados..
3. Seleciona a identificação do cliente no campo A.	
4. Informa o número da Ordem de Compra no campo B.	5. Valida que Ordem de Compra não existe. Carrega no combo D os fornecedores cadastrados.
6. Informa data da Ordem de Compra campo C.	
7. Seleciona o fornecedor no campo D.	8. Para cada item de Ordem de Compra, acionar caso de uso <i>Incluir Item Ordem de Compra</i> .
	9. Verifica se há item incluído, habilitando o botão N (Salvar Ordem de Compra).
10. Aciona o botão N para salvar a Ordem de Compra.	11. Sistema persiste os dados.
	12. Pronto para registrar nova Ordem de Compra.

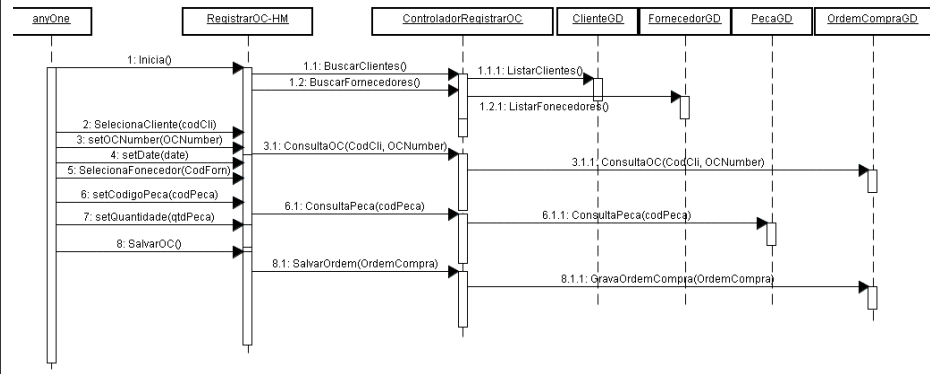
©Pimenta 2011

Almoxarifado – diagrama de classe (3)



©Pimenta 2011

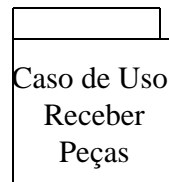
Almoxarifado – D. Seqüência (4)



©Pimenta 2011

Trecho de Caso de Uso real

Ação do Ator	Resposta do Sistema
1. A tela é montada exibindo somente o quadro "Dados da Ordem de Compra"	
2. Seleciona o cliente na lista A e informa o nro. da OC na caixa de texto B.	
3. Clica no botão C, submetendo a página	
4. O sistema valida a OC digitada e busca todas as peças daquela OC. Para cada peça, uma linha na tabela do quadro "Peças na Ordem de Compra" é gerada, com as seguintes colunas:	
5. Informa a quantidade para cada peça recebida nas caixas de texto da coluna F	
6. Clica no botão H	
.....



©Pimenta 2011

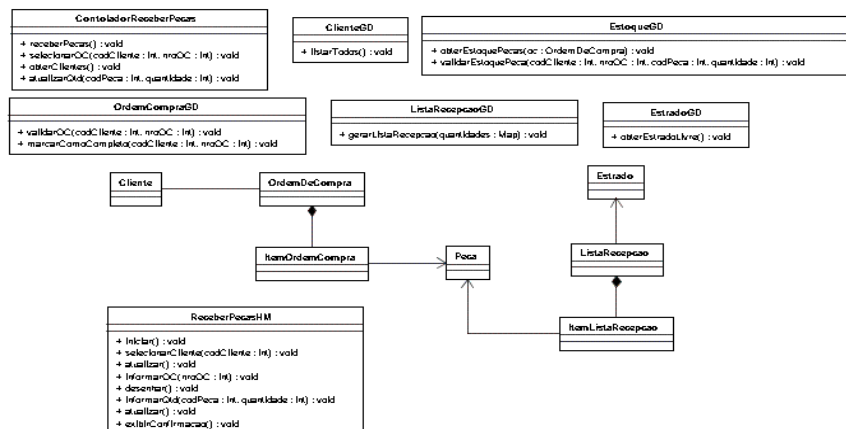
Caso de uso real – Receber Peças

Exceções:

- Passo 4 -** O sistema valida a OC digitada e busca todas as peças daquela OC.
Exceção: O usuário não digitou nro. de OC, ou digitou um nro. não registrado para o cliente selecionado.
Tratamento: Exibir uma mensagem para o ator e retornar à tela.
- Passo 4 -** O sistema valida a OC digitada e busca todas as peças daquela OC.
Exceção: A OC informada já foi concluída.
Tratamento: Exibir uma mensagem para o ator e retornar à tela.
- Passo 7 -** Valida se cada quantidade digitada é superior a zero e menor que a quantidade da peça na OC – quantidade atual em estoque.
Exceção: O usuário digitou uma quantidade superior a que falta para a peça ser concluída
Tratamento: Exibir uma mensagem para o ator e retornar à tela.

©Pimenta 2011

Diag. De Classes - Receber Peças

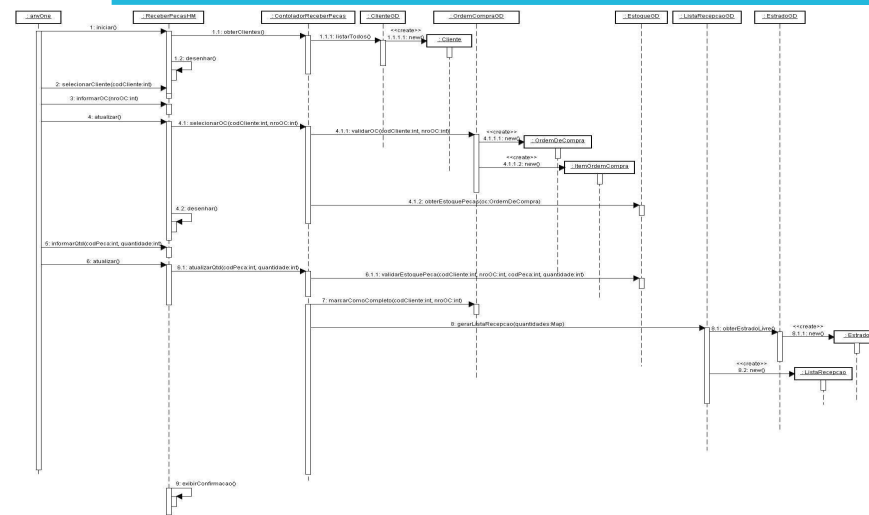


O modelo de comunicação é: HM <-> Controlador <-> GD.

As classes de DP não contém métodos de negócio, são usadas somente para conter os dados.

©Pimenta 2011

Diag. De Seqüência – Receber Peças



©Pimenta 2011

Classe de Projeto

- **Análise**
 - Diagrama de classes serve como modelo conceitual
 - Descreve conceitos, não classes de software a implementar
- **Projeto**
 - A partir dos diagramas de colaboração, construir diagramas com as classes de software que efetivamente serão implementadas
 - Em realidade, diagramas de classe são construídos concomitantemente com os diagramas de interação
 - Exemplificando: alguns CASEs (p.ex. Rational*Rose) permitem definir os métodos das classes ao construir o diagrama de colaboração (ou seqüência)

©Pimenta 2011

Objetos persistentes

- Linguagens de programação OO não implementam o conceito de persistência
- Persistência pode ser provida de várias formas
 - Uso de um SGBDOO ou biblioteca de classes de persistência
 - Basta indicar de alguma forma (normalmente, especialização de classe de objetos persistentes) que a classe é persistente
 - Software implementa a persistência com vários graus de automatismo (da carga e descarga de objetos)
 - Implementação de persistência sobre SGBD relacional ou sistema de arquivos

©Pimenta 2011

CMP102

Engenharia de Software

Prof. Marcelo Soares Pimenta
mpimenta@inf.ufrgs.br

Slides – Arq 2

©Pimenta 2011

Etapas do Ciclo de Vida do Software: Objetivos, Principais Abordagens e Artefatos

©Pimenta 2011

Etapas : Teste

©Pimenta 2011

Bibliografia

- Pezzè, M.; Young, M. Software Testing and Analysis – Process, principles and Techniques, Wiley, 2008. (tem tradução pela Bookman)
- Myers, G. The Art of Software Testing John Wiley and Sons , New York, 1979.
- Beck, K. Test-Driven Development by Example, Addison Wesley, 2003
- *Sommerville, I. Software Engineering, ed. Pearson/Prentice Hall, 2003.*

©Pimenta 2011

Bibliografia adicional

- Braude, E. Software Engineering - An Object-Oriented Perspective, John Wiley, 2001.
- Bach, J. Heuristic Based-Risk Test. Software Testing and Quality Engineering Magazine, November 1999 (versão eletrônica em PDF disponível com o professor)
- revista Software Testing and Quality Engineering,
- Glass, R.L. "Persistent Software Errors," IEEE Transactions on Software Engineering, March 1981.
- V. Basili and R.W. Selby. 'Comparing the Effectiveness of Software Testing Strategies.' IEEE Transactions on Software Engineering, vol. SE-13, No. 12, December, 1987.

©Pimenta 2011

Programa

- Fundamentos: Verificação, Validação e Testes
- Objetivos dos testes
- Princípios, métodos, níveis, técnicas e tipos de teste
- Abordagens de Testes de Integração e de Sistema
- Abordagens de Testes de Unidade
- Projeto de Casos de Teste
- Rumo a Automação de Testes

©Pimenta 2011

Primeira Parte

©Pimenta 2011

Teste nos Processos de Software

- **Aspectos comuns a todos modelos:**
 - Todos começam com (ou visam) uma **compreensão** melhor dos **requisitos do sistema** a ser desenvolvido
 - Todos visam **aumento da qualidade** dos resultados parciais e finais
 - Todos visam **melhoria da gerência** do processo de desenvolvimento (alguns, o controle do processo; outros, o acompanhamento do processo)
- Todos têm alguma forma de **validação/teste !!**

©Pimenta 2011

Sessão Depoimentos

- Discussão:

- Qual o modelo de ciclo de vida adotado no(s) seu(s) ambiente(s) de trabalho?
- A seu ver, quais as razões para a adoção deste modelo de ciclo?

©Pimenta 2011

Erros (falhas) e Software

- Primeira premissa: ERRO é fato **cotidiano** e não excepcional
- 3 formas de tratar erros (falhas) de software:
 1. **Evitar falhas** (*fault-avoidance*): especificação, projeto, implementação e manutenção usando métodos sistemáticos e confiáveis, tipicamente baseadas em métodos formais e reuso de componentes de software altamente confiáveis; -> praticável somente para sistemas altamente críticos
 2. **Eliminar falhas** (*fault-elimination*): análise, detecção e correção de erros cometidos durante a desenvolvimento. **Aqui incluem-se as atividades verificação, validação e teste.**
 3. **Tolerar falhas** (*fault-tolerance*): compensação em tempo real de problemas residuais como mudanças fora da especificação no ambiente operacional, erros de usuário, etc. Geralmente lida com recursos de hardware e/ou software adicionais ou redundantes;
- Devido ao fato de *fault-avoidance* ser economicamente impraticável para a maioria das empresas, e de *fault-tolerance* exigir muitos recursos a tempo de execução, a técnica de eliminação de falhas (*fault-elimination*) geralmente é a adotada pelos desenvolvedores de software.

©Pimenta 2011

Objetivo dos Testes

- Objetivo:
 - Detectar a presença (existência) de erros
- Suposição incorreta 1: 'Mostrar **ausência** de erros'
- Suposição incorreta 2: '**Assegurar** que o programa funciona corretamente'
- Verificação ('We Build it right ?'):
 - O que foi especificado e projetado foi construído corretamente?
 - Checado em relação à especificação e aos requisitos
- Validação ('We Build the right thing ?'):
 - Construímos o que era certo ?
 - Checado pelo cliente/usuário

©Pimenta 2011

Teste - Exercício

- Como você testaria um programa com a seguinte definição?
 - Seja um programa que leia três inteiros que corresponderiam aos lados de um triângulo (A,B,C), verifica se é efetivamente um triângulo, e em caso positivo classifica-o em triângulo escaleno, isósceles e equilátero

©Pimenta 2011

Testes no Desenvolvimento de software

- Testes iterativos e contínuos possibilitam uma medida objetiva do status do projeto
- Inconsistências nos requisitos, projeto e implementação são detectadas mais cedo
- A responsabilidade pela qualidade de trabalho da equipe é melhor distribuída ao longo do ciclo de vida
- Os envolvidos (*stakeholders*) no projeto podem ter evidências concretas do andamento do projeto
- Testes – PAs Verification & Validation para nível 3 CMMI

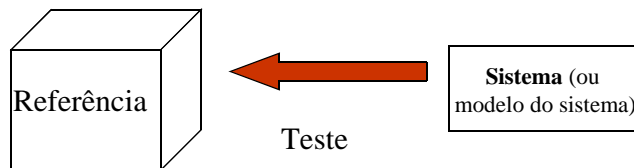
©Pimenta 2011

Testes no Desenvolvimento de SW

- Processo efetivo de teste é solução para alguns problemas do desenvolvimento:
 - Medida do status do projeto é objetiva porque os testes reportados são avaliados
 - Esta medida objetiva expõe inconsistências nos requisitos, projeto e implementação
 - Testes e verificações estão focadas nas áreas de maior risco, aumentando a qualidade destas áreas.
 - Defeitos são descobertos antes, reduzindo o custo de seu conserto
 - Ferramentas automatizadas de teste provêm testes para funcionalidade e performance.
 - Teste é a **única** técnica de validação para requisitos não-funcionais

©Pimenta 2011

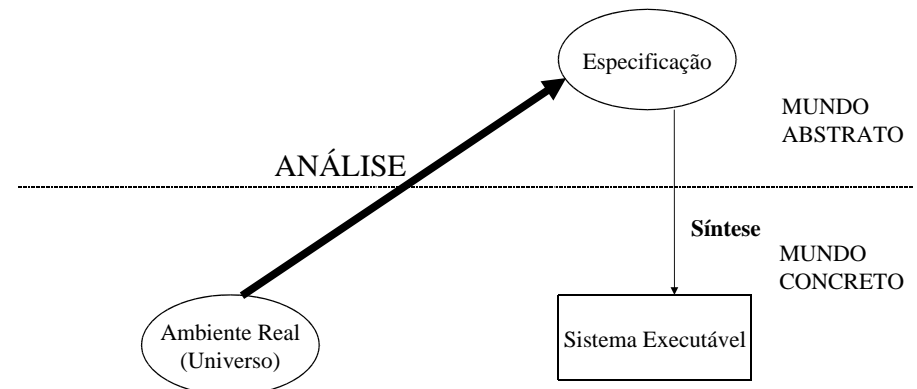
Teste x Análise



- Modelo explícito
- Idéias, expectativas (modelo implícito)
- Procedimentos manuais realizados
- Sistema atual existente
- Especificação de Requisitos

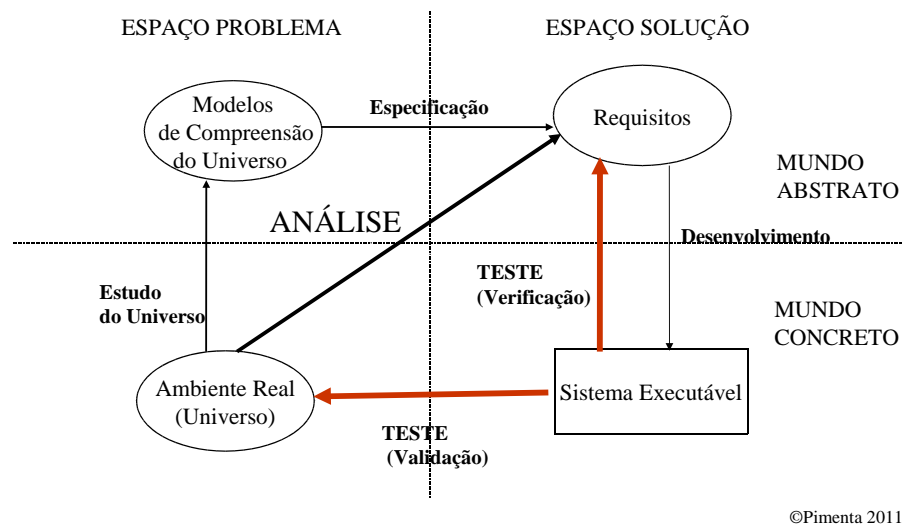
©Pimenta 2011

Teste x Análise

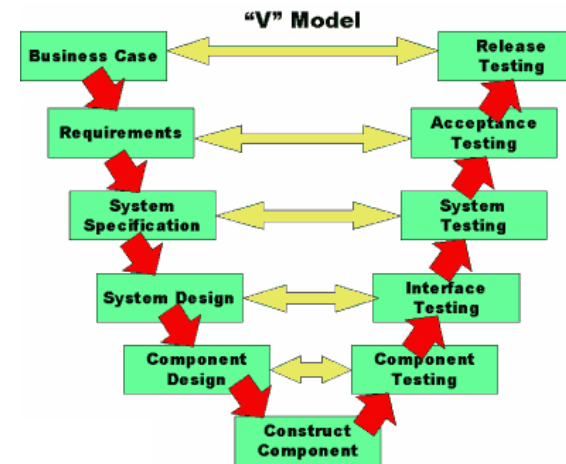


©Pimenta 2011

Teste x Análise



Priorizando Teste: o modelo 'V'



Tópicos abordados

- Fundamentos de teste:
 - Objetivos dos testes
 - Princípios de teste
- Métodos de Teste
- Níveis de Teste
- Técnicas de Teste
- Tipos de Teste
- Abordagens de Testes de Integração e de Sistema
- Abordagens de Testes de Unidade
- Projeto de Casos de Teste
- Rumo a Automação de Testes

©Pimenta 2011

Por que testar?

- Achar erros mais rapidamente
 - Antes do usuário achar
 - Quando ainda são mais fáceis de corrigir
- Poupar tempo para dedicar ao trabalho e não a re-trabalho
 - Tempo realizando teste deve ser menor que tempo de retrabalho
 - Por isto TESTES devem ser BEM PLANEJADOS
- Ter testes definidos aumenta confiança em fazer mudanças sem receios de injetar efeitos colaterais indesejados

Se desejar isto, teste !!!

Se - ao testar - não está conseguindo isto, mude o modo de testar !!

©Pimenta 2011

Um exemplo (1)

```
class BindingSet {...  
Boolean eval(Expr expr) {...}  
...}
```

- Suponham $\text{expr} = a \ \&\& (b \ || \ !c)$
Com { $a = \text{true}, b = \text{false}, c = \text{true}$ }
- O resultado seria Boolean.FALSE.

©Pimenta 2011

Um exemplo (2)

- Os valores para a, b e c formam as entradas de um caso de teste ;
 - “Entrada” = qualquer dado que pode afetar a computação do código sendo testado
 - No exemplo, as entradas são argumentos para `eval()` e o estado da instância de `BindingSet`
- Um caso de teste também descreve o resultado esperado (saída):
 - Se a saída real = saída esperada , o código passou no teste; senão falhou
- **CASO DE TESTE** = combinação de entrada e saída esperada para uma determinada funcionalidade a se testar em diferentes condições
 - Os melhores testes são aqueles que apontam o maior número de erros diferentes com o menor número de casos de teste;

©Pimenta 2011

Pontos chave

- Planejar testes e critérios de aceitação; depois, codificar (*Test, then code*)
- Execute testes continuamente durante período de codificação para aumentar confiança no que está sendo feito
 - Automação de teste viabiliza teste contínuo
- Teste é parte de sua vida como programador e analista
- Projeto de teste visa sistematicamente aprender dos erros passados
 - Criar memória de teste
- Implementação do teste visa reduzir o custo dos testes
 - Automação da implementação via uso de ferramentas
- Cada programador e analista deve ficar estreitamente “focado” nos testes de seus sistemas
 - “Advogados do diabo” de seu próprio sistema

©Pimenta 2011

Objetivo dos Testes

- Objetivo:
 - Detectar a presença (existência) de erros
- Suposição incorreta 1: ‘Mostrar **ausência** de erros’
- Suposição incorreta 2: ‘**Assegurar** que o programa funciona corretamente’
- Verificação (‘We Build it right ?’):
 - O que foi especificado e projetado foi construído corretamente?
 - Checado em relação à especificação e aos requisitos pela equipe SEM PARTICIPAÇÃO DO USUÁRIO
- Validação (‘We Build the right thing ?’):
 - Construímos o que era certo ?
 - Checado pelo cliente/usuário

©Pimenta 2011

Testes - Finalidades

- Finalidade:
 - Localizar e documentar defeitos na qualidade do software.
 - Verificar se as funções do software agem conforme projetadas.
 - Verificar se os requisitos foram implementados de forma adequada.
 - Validar as especificações de requisitos através de demonstração concreta.
 - Averiguar de forma geral a qualidade observada no software.

©Pimenta 2011

Visão Geral de Testes

- Princípios de teste:
 - Teste é uma fase planejada DURANTE o desenvolvimento do sistema e NÃO após;
 - Teste é **responsabilidade de todos na equipe**:
 - Teste deve ser planejado pelo analista e projetista ;
 - Teste funcional deve ser realizado pelo desenvolvedor;
 - Todos outros testes podem ser realizados por outra pessoa (evitar indução);
 - Os melhores testes são aqueles que apontam o maior número de erros diferentes com o menor número de experimentos;
 - Fim dos testes é ditado geralmente por argumentos econômicos;
 - É melhor ter em mente que os erros são inevitáveis e você é que tem dificuldade de encontrá-los

©Pimenta 2011

Após o Teste: Depuração

- Depuração
 - Objetivo: Localizar e Corrigir o erro
- Processo de Investigação que geralmente envolve novos testes para determinar A FONTE do erro
- 2 etapas:
 - Compreensão (superficial) das idéias do programa
 - Análise (aprofundada) dos trechos suspeitos
- Técnicas: instrumentação do código (inclusão de comandos) ('var=', 'passei por aqui')
- Ferramentas: depuradores (traces, dumps, breakpoints, execuções condicionais, etc)

©Pimenta 2011

TDD (Test-Driven Development)

- Prática de XP que ganhou independência
- Estilo de programação baseado na premissa de que os testes são escritos **antes** do código das classes de produção:
 - Testes de aceitação definidos ANTES da codificação
 - Usuário participa ativamente da definição dos testes
 - Desenvolvedor sabe perfeitamente o que deve ser feito (o que deve passar no teste de aceitação definido)
 - Os testes auxiliam a diminuir o acoplamento e aumentar a coesão entre classes
 - Mais fácil medir o real progresso (% do sistema já está funcionando ?)
 - Testes OK aumentam a confiança nas funcionalidades implementadas

©Pimenta 2011

Escrever testes antes de codificar?

- **Por que é tão importante escrever testes ANTES de codificar ?**
- **Razão 1:** Especificar os erros antes de codificar promove um fluxo de experiência muito eficiente durante codificação. Sabe-se precisamente onde se quer chegar e como se deve chegar
 - Escrever testes primeiro previne erros. E ajuda a pensar em situações que tradicionalmente não são previstas !!!
- **Razão 2:** Surpreendentemente, quando define-se testes antes, codifica-se defensivamente mas objetivamente; Ao definir testes depois de codificar quem fica na defensiva é o programador;

©Pimenta 2011

Dimensões de Testes

- **Dimensão de qualidade**
 - Os atributos de qualidade que estão sendo o foco do teste
 - P.ex. : Tipos de teste - Teste de funcionalidade, Teste de segurança, Teste de Carga (ou Teste de Stress), Teste de Desempenho (ou Teste de performance), Teste de Usabilidade
- **Categoria do teste**
 - Técnica de teste usada
 - P.ex. Teste estático vs Teste dinâmico
- **Estágio do teste**
 - O ponto dentro do ciclo de desenvolvimento em que o teste está sendo executado
 - P.ex. Níveis de teste: Teste de Unidade, Teste de Integração, Teste de Sistema, Teste de Aceitação (Homologação)
- **Método de teste**
 - O objetivo específico do teste individual
 - P.ex. “caixa preta” vs “caixa branca”

©Pimenta 2011

Testes Estáticos vs Testes Dinâmicos

- **Categorias de Teste** (em relação à execução)
 - Estático: checagem do código SEM execução
 - Inspeção (leitura cruzada em equipe com argumentação)
 - Walkthrough (‘teste de mesa’)
 - Análise estática de propriedades do código com ferramentas :
 - verificação FORMAL de tipos, estruturas de controle, métricas de complexidade
 - Dinâmico: checagem do código a partir de sua execução sobre conjuntos de dados (casos de teste)
 - Como escolher casos de teste?
 - Como decidir se um resultado é correto ou não?
 - Quando decidir parar os testes?
 - Geralmente o MAIS ADOTADO é o Dinâmico

©Pimenta 2011

Inspeção de Software

- Envolve pessoas examinando (inspecionando) o modelos do sistema ou seu código fonte visando descobrir anomalias e falhas
- Não requer execução do sistema, logo pode ser realizada antes da implementação terminar (ou começar)
- Pode ser aplicada a qualquer representação do sistema (requisitos , modelos de especificação, dados de teste, etc)
- Técnica MUITO efetiva para descobrir erros conceituais

©Pimenta 2011

Fault class	Inspection check
Data faults	Are all program variables initialised before their values are used? Have all constants been named? Should the lower bound of arrays be 0, 1, or something else? Should the upper bound of arrays be equal to the size of the array or Size -1? If character strings are used, is a delimiter explicitly assigned?
Control faults	For each conditional statement, is the condition correct? Is each loop certain to terminate? Are compound statements correctly bracketed? In case statements, are all possible cases accounted for?
Input/output faults	Are all input variables used? Are all output variables assigned a value before they are output?
Interface faults	Do all function and procedure calls have the correct number of parameters? Do formal and actual parameter types match? Are the parameters in the right order? If components access shared memory, do they have the same model of the shared memory structure?
Storage management faults	If a linked structure is modified, have all links been correctly reassigned? If dynamic storage is used, has space been allocated correctly? Is space explicitly de-allocated after it is no longer required?
Exception management faults	Have all possible error conditions been taken into account?

Ex. de Inspeção (checklist)

©Pimenta 2011

Análise Estática Automatizada

- Analisadores estáticos são ferramentas (sw) para processamento (análise automática) do código fonte
- Basicamente, fazem um “parse” do programa e tentam descobrir condições potencialmente suspeitas (na nomenclatura OO, “bad smells”) e alertá-las para a equipe de teste
- MUITO efetiva como auxílio a inspeção , mas apenas a complementa , não a substitue

©Pimenta 2011

O que pode ser analisado automaticamente?

Fault class	Static analysis check
Data faults	Variables used before initialisation Variables declared but never used Variables assigned twice but never used Possible array bound violations Undeclared variables
Control faults	Unreachable code Unconditional branches into loops
Input/output faults	Variables output twice with no intervening assignment
Interface faults	Parameter type mismatches Parameter number mismatches Non-usage of the results of functions Uncalled functions and procedures
Storage management faults	Unassigned pointers Pointer arithmetic

©Pimenta 2011

Ex. de resultado do analisador LINT

138% more lint_ex.c

```
#include <stdio.h>
printarray (Anarray)
int Anarray;
{
    printf("%d",Anarray);
}
main ()
{
    int Anarray[5]; int i; char c;
    printarray (Anarray, i, c);
    printarray (Anarray) ;
}
```

139% cc lint_ex.c

140% lint lint_ex.c

```
lint_ex.c(10): warning: c may be used before set
lint_ex.c(10): warning: i may be used before set
printarray: variable # of args. lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) ::
lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) ::
lint_ex.c(11)
printf returns value which is always ignored
```

©Pimenta 2011

Uso da Análise Estática

- Particularmente valiosa quando uma linguagem como C (tipagem fraca, uso intensivo de ponteiros sem controle de uso de memória) é usada e muitos erros ou alertas não são detectados pelo compilador
- Menos efetiva para linguagens como Java (fortemente tipadas) cujo compilador adianta-se e detecta muitos erros e alertas

©Pimenta 2011

Teste Estático: Pontos-Chave

- Técnicas de Verificação Estática envolvem exame e análise do código do programa para detecção de erros
- **Inspecções** de programa são muito efetivas para descobrir erros em alto nível de abstração
- Em inspecções, o código do programa é checado por um pequeno time para localizar falhas do sistema
- Ferramentas de **análise estática** podem descobrir anomalias do programa, que podem ser uma indicação de falhas no código

©Pimenta 2011

Objetivos do processo de testes

- Testes de validação
 - Auxiliar a verificação para o desenvolvedor e clientes de que o software atende seus requisitos
 - Um teste bem sucedido mostra uma situação onde o sistema age como pretendido
- Testes de verificação de defeito (ou teste de detecção de defeito)
 - Descobrir falhas no software onde o comportamento é incorreto ou não está de acordo com a especificação
 - Um teste bem sucedido é aquele que mostra estes problemas

©Pimenta 2011

Testes para a detecção de defeitos

- O objetivo é descobrir defeitos em programas
- Um teste bem sucedido é aquele que faz com que o programa se comporte de maneira anômala
 - ‘Encontrar o máximo de defeitos com o mínimo de testes’
- Lembrete: Testes mostram a presença de defeitos
 - Mas não a sua ausência

©Pimenta 2011

Requisitos x Teste

- Uma clara compreensão dos requisitos é fundamental para qualquer atividade de teste, verificação ou validação
- Como checar algo se não se sabe precisamente o que deveria ser feito?

©Pimenta 2011

Requisitos

- Requisitos são o PONTO DE PARTIDA para qualquer processo de software !!!!
 - Tradicional, prototipação, incrementais, orientado a modelos (MDA), etc
- Tipos de Requisitos
 - Requisitos Funcionais : associados a funcionalidade
 - Requisitos Não Funcionais: associados à propriedades (critérios de qualidade) e restrições

©Pimenta 2011

Requisitos x Teste

- Uma clara compreensão dos requisitos é fundamental para qualquer atividade de teste, verificação ou validação
- Como checar algo se não se sabe precisamente o que deveria ser feito?

©Pimenta 2011

Estrutura de um Plano de Teste

- OBS: Planos de teste são construídos para guiar o processo de teste
- O processo de teste
- Rastreabilidade dos requisitos
- Itens (a serem) testados
- Cronograma de teste
- Procedimentos de registro do teste
- Requisitos de HW e SW para teste
- Restrições

©Pimenta 2011

Políticas de Teste

- Somente Teste Exaustivo pode mostrar que um programa é livre de erros. Teste exaustivo é testar para **todas** as possibilidades de execução. Entretanto, para **empresas**, na **prática** teste exaustivo é impossível ...
- Políticas de teste definem o enfoque a ser usado na seleção dos testes dos sistemas de uma organização:
 - O quê testar ?
 - Como testar? Que nível ou estratégia ou método de teste usar?
 - Dimensões de qualidade, estágio de teste e/ou método de teste
 - Quando testar?

©Pimenta 2011

Métodos de Teste

- **Teste Funcional (caixa preta)**
 - Visa verificar funcionalidade baseado apenas nos dados de entrada e saída.
- **Teste Estrutural (caixa branca)**
 - Visa explorar certos caminhos do programa (ou sistema).

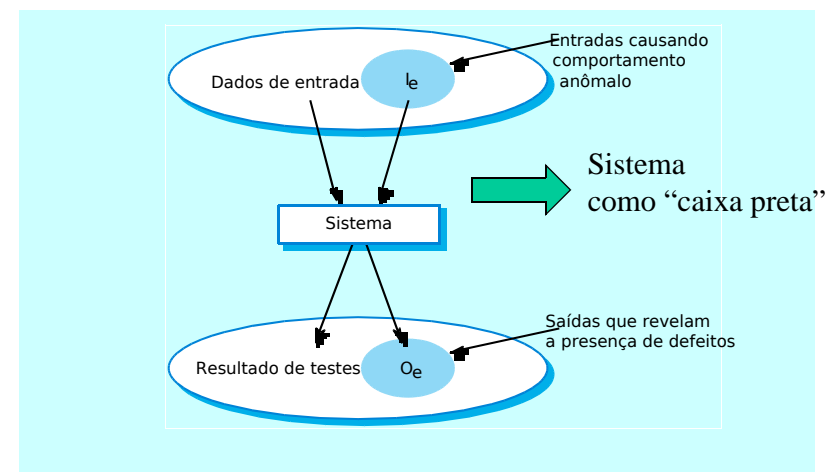
©Pimenta 2011

Métodos de Teste

- **Teste Funcional (caixa preta)**
 - Visa verificar funcionalidade baseado apenas nos dados de entrada e saída.
 - Verificar resultados finais , **não importando a estrutura, estados e comportamento internos**.
 - O termo ‘caixa preta’ traduz a idéia de uma caixa que **não** permite que seu conteúdo seja visível do lado de fora.

©Pimenta 2011

Testes Caixa-Preta



©Pimenta 2011

Técnicas de Teste

- Teste Funcional usa técnicas de:
 - Testes por Particionamento de Equivalência
 - Teste de Limites

©Pimenta 2011

Teste Funcional

- **Particionamento de Equivalência**
 - Visa selecionar os casos de teste certos para cobrir todos os cenários possíveis.
 - Partição de equivalência é derivado da especificação do comportamento dos componentes:
 - Uma entrada de um componente tem um certo domínio de valores que são válidos e outros domínios de valores que são inválidos.
- A teoria do teste de software diz que somente um caso de teste de cada partição é necessária para validar o comportamento do programa .

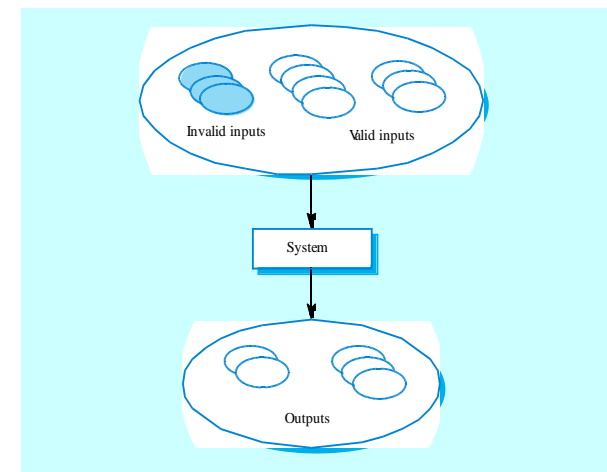
©Pimenta 2011

Testes de partição

- Dados de entrada e resultados de saída normalmente enquadram-se em diferentes classes, nos quais os membros destas classes são relacionados
- Cada uma destas classes é uma partição de equivalência onde o programa comporta-se de maneira equivalente para cada um dos membros desta classe
- Casos de testes podem ser escolhidos para cada partição

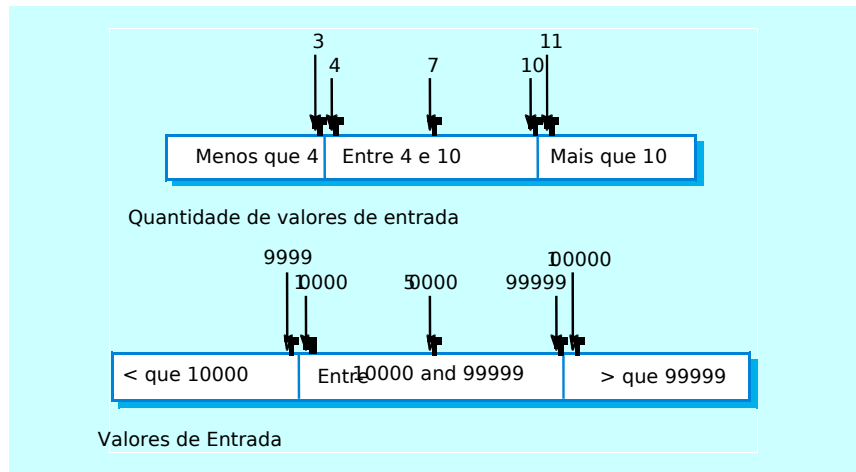
©Pimenta 2011

Partições de equivalência



©Pimenta 2011

Partições de equivalência



©Pimenta 2011

Especificação de uma rotina de busca

procedure Search (Key : ELEM ; T: SEQ of ELEM;
Found : **in out** BOOLEAN; L: **in out** ELEM_INDEX) ;

Pre-condition

-- O vetor tem ao menos um elemento
T'FIRST <= T'LAST

Post-condition

-- O elemento é encontrado e referenciado por L
(Found and T (L) = Key)

or

-- O elemento não está no vetor
(**not** Found and
not (exists i, T'FIRST >= i <= T'LAST, T (i) = Key))

©Pimenta 2011

Rotina de Busca - Recomendações de testes

- Testar o software com arrays com apenas um elemento
- Usar arrays de diferentes tamanhos em diferentes testes
- Derivar testes de forma que o primeiro elemento, o do meio e o último do array sejam acessados
- Testar com sequências de tamanho zero

©Pimenta 2011

Busca Binária – Partições de Equivalência

- Pré-condições satisfeitas, elemento no vetor
- Pré-condições satisfeitas, elemento não está no vetor
- Pré-condições não satisfeitas, elemento no vetor
- Pré-condições não satisfeitas, elemento não está no vetor
- O vetor de entrada tem um único valor
- O vetor de entrada tem um número par de valores
- O vetor de entrada tem um número ímpar de valores

©Pimenta 2011

Rotina de Busca – Partições de Entrada

Vetor	Elemento
Valor Único	No vetor
Valor Único	Não existe no vetor
Mais de um valor	Primeiro elemento
Mais de um valor	Último elemento
Mais de um valor	Elemento do meio
Mais de um valor	Não no vetor

Input sequence (T)	Key (Key)	Output (Found, L)
17	17	true, 1
17	0	false, 0
17, 29, 21, 23	17	true, 1
41, 18, 9, 31, 30, 16, 45	45	true, 7
17, 18, 21, 23, 29, 41, 38	23	true, 4
21, 23, 29, 33, 38	25	false, ??

©Pimenta 2011

Busca Binária – Casos de teste

Sequencia de entrada (T)	Chave (Key)	Saída (Found, L)
17	17	true, 1
17	0	false, ??
17, 21, 23, 29	17	true, 1
9, 16, 18, 30, 31, 41, 45	45	true, 7
17, 18, 21, 23, 29, 38, 41	23	true, 4
17, 18, 21, 23, 29, 33, 38	21	true, 3
12, 18, 21, 23, 32	23	true, 4
21, 23, 29, 33, 38	25	false, ??

©Pimenta 2011

Teste Funcional

- **Teste de Limites**
- É necessário testar os limites de entrada de dados do sistema para que defeitos relacionados a esses limites apareçam facilmente.
- Por exemplo, verificar o tamanho e tipo de um determinado campo listado em um formulário e então gerar uma entrada com o tamanho máximo, salvar os dados e verificar se o registro foi salvo no banco de dados e se não ocorreu nenhum erro a esse respeito.
- Outra forma é testar os valores máximos e mínimos para campos numéricos e para condições de loop.

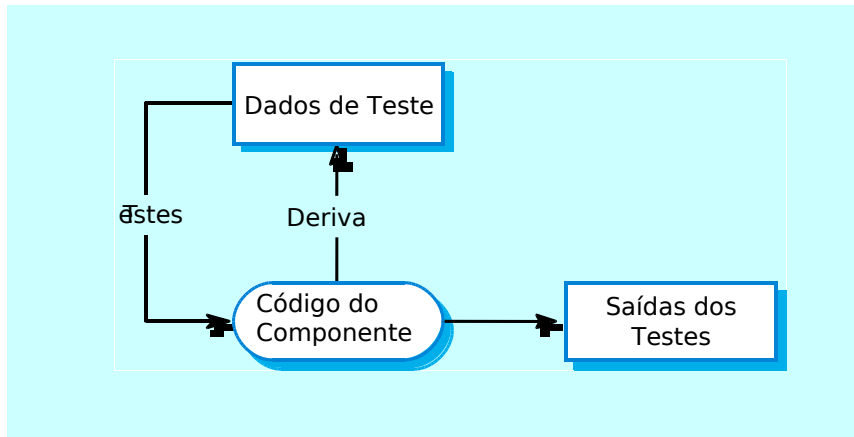
©Pimenta 2011

Métodos de Teste

- **Teste Estrutural (caixa branca)**
 - Visa exploração de certos caminhos do programa (ou sistema).
 - (Um caminho é uma das sequências possíveis de serem executadas no fluxo de controle do programa.)
 - Um teste estrutural é realizado prevendo que cada caminho de um programa seja executado ao menos uma vez, e por isto necessita ter acesso e conhecimento da estrutura ou do código fonte do programa.
 - Na verdade, os casos de teste são derivados da estrutura dos programas. O conhecimento do código programa é usado para propor casos de teste adicionais – para “exercitar” um caminho.
 - O termo ‘caixa branca’ traduz a idéia de transparência de uma caixa que permite que seu conteúdo seja visível do lado de fora.

©Pimenta 2011

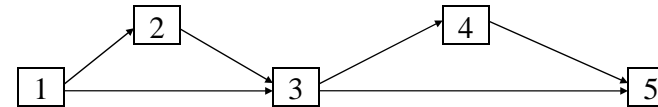
Testes Estruturais



©Pimenta 2011

Teste Estrutural

- Exemplo de Casos de Teste para Teste Caixa-Branca
 - Grafo de Controle entre blocos de instruções elementares



Cobertura de instrução: (1,2,3,4,5)

Cobertura de Ramos: (1,2,3,4,5) (1,3,5)

Cobertura de Caminhos Lógicos: (1,2,3,4,5) (1,3,5) (1,2,3,5)
(1,3,4,5)

©Pimenta 2011

Técnicas de Teste

- Teste Estrutural usa técnicas de:
 - Teste de Caminho
 - Teste de Condição

©Pimenta 2011

Testes de Caminho

- O objetivo destes testes é garantir que o conjunto de casos de testes é tal que cada caminho de um programa seja executado ao menos uma vez
- O início de um teste de caminho é um grafo de fluxo que mostra os nós representando decisões em um programa e arcs representando o fluxo de controle
- Sentenças com condições são nós no grafo de fluxo

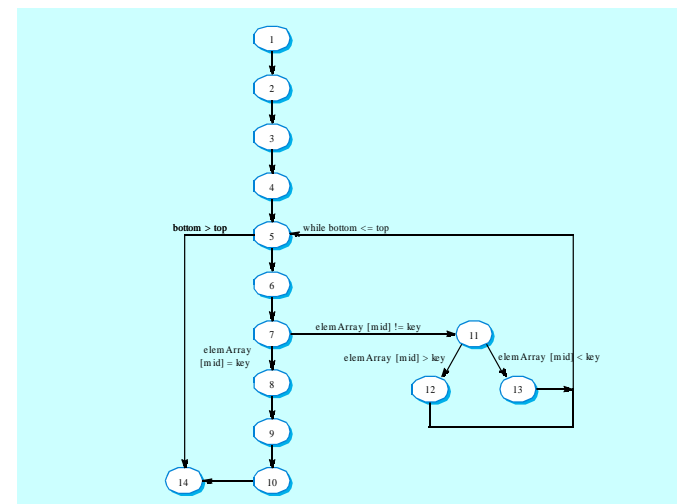
©Pimenta 2011

Testes de Condição

- O objetivo destes testes é garantir que o conjunto de casos de testes é tal que cada condição de um programa seja avaliada ao menos uma vez
- O início de um teste de condição pode ser o mesmo grafo de fluxo que mostra os nós representando decisões em um programa e arcos representando o fluxo de controle
- Sentenças com condições são nós no grafo de fluxo

©Pimenta 2011

Grafo – Busca binária



©Pimenta 2011

Caminhos independentes

- 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 14
- 1, 2, 3, 4, 5, 14
- 1, 2, 3, 4, 5, 6, 7, 11, 12, 5, ...
- 1, 2, 3, 4, 6, 7, 2, 11, 13, 5, ...
- Casos de teste podem ser derivados de forma que estes caminhos sejam executados
- Um analisador dinâmico ou depurador pode ser usado para verificar se os caminhos foram executados

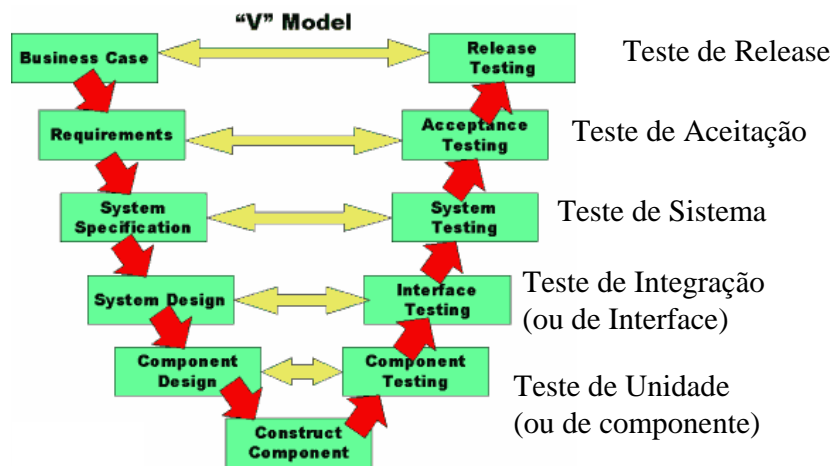
©Pimenta 2011

Níveis de Teste

- As atividades de testes ocorrem ao longo do processo de desenvolvimento.
 - Teste de Aceitação
 - Teste de Integração
 - Teste de Sistema
 - Teste de Unidade (ou Teste Unitário)
 - Teste de Regressão

©Pimenta 2011

Níveis de Teste: o modelo 'V'



©Pimenta 2011

Teste de Unidade (ou Teste Unitário)

- **Teste de Unidade (ou Teste Unitário ou Teste de Componente)**
- Seu objetivo é encontrar erros em unidades individuais do sistema, sendo que essas unidades são testadas isoladamente.
- É feito pelo **desenvolvedor da unidade (programador)**
- Verificação de uma unidade de software
 - Pode ser via **teste funcional**, desenvolvido a partir da especificação das funções previstas para a unidade, ou via **teste estrutural**, desenvolvido a partir da descrição da estrutura do código da unidade.
- Uma unidade pode ser um módulo, uma subrotina, uma *procedure*, uma classe ou até mesmo um programa simples. No contexto da UDS, unidade é uma porção de código com identificação única.

©Pimenta 2011

Teste de Unidade (ou Componentes)

- **Teste Funcional (Caixa-Preta)**
 - Casos de Teste selecionados a partir das especificações
 - Regras para Casos de Teste:
 - No mínimo, um caso de teste para exercitar cada característica importante do módulo
 - Descobrir casos especiais não cobertos pelos casos de teste da regra anterior (p.ex. valores limites entre classes de soluções, valores extremos de solução)
 - Examinar casos de teste para dados de entrada ausentes, inválidos, errados ou pouco usuais; Exemplos incluem combinações com zero, dados negativos ou não pertencentes ao domínio de entrada

©Pimenta 2011

Testes de Unidade (ou Componentes)

- Testes de componentes ou testes unitários visam testar componentes isolados do sistema, um a um
- É um processo de descoberta de defeitos
- Componentes podem ser:
 - Métodos de uma classe
 - Classes com diversos atributos e métodos
 - Conjuntos de classes que proporcionam uma fachada de acesso às suas funcionalidades

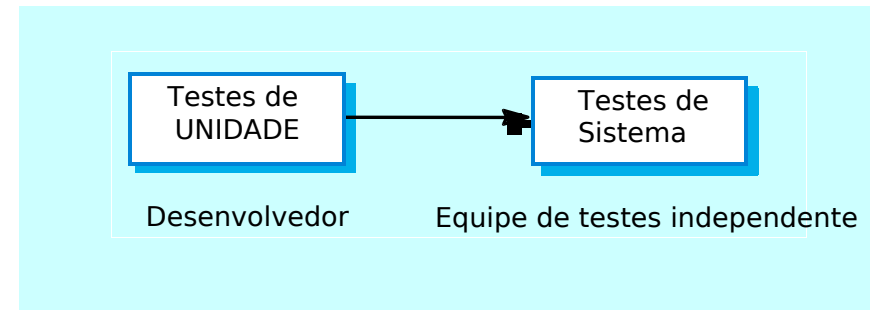
©Pimenta 2011

Teste de Integração

- Teste da interconexão entre os módulos que visa encontrar defeitos relacionados às interfaces entre unidades.
- Deve ser feito com as unidades já devidamente testadas; Portanto, sucede ao Teste de Unidade;
- Geralmente começa verificando poucas unidades interagindo entre si e termina verificando a cooperação de todas unidades do sistema, incrementalmente.
- Facilitado pela modularização do sistema e pela programação defensiva (módulos que testam seus parâmetros de E/S - pré e pós condições)

©Pimenta 2011

Fases de Testes



Teste de UNIDADE = Teste de Unidade (isoladamente)
Teste de Sistema = Teste após integrar unidades

©Pimenta 2011

O processo de testes

- Testes Unitários de Componentes
 - Testar os componentes individuais de um programa
 - Normalmente são de responsabilidade do desenvolvedor do componente
 - Os testes são derivados da experiência do desenvolvedor
- Testes de integração e de sistema
 - Testes de grupos de componentes integrados, que formam um sistema ou sub-sistema
 - Responsabilidade de um time de testes independente
 - Os testes são baseados em especificações do sistema

©Pimenta 2011

Abordagens de testes de integração

- Validação de arquitetura
 - Testes de integração top-down são a melhor maneira de descobrir erros em uma arquitetura
- Demonstração do sistema
 - Testes de integração top-down permitem uma demonstração limitada do sistema em estágios iniciais do processo de desenvolvimento
- Implementação de testes
 - Normalmente é mais fácil com testes de integração bottom-up

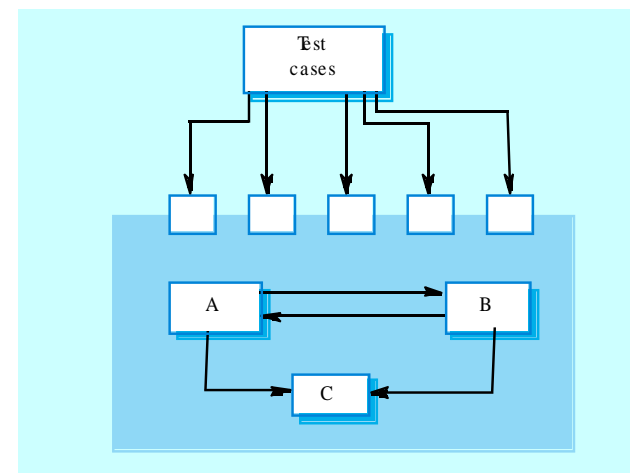
©Pimenta 2011

Testes de integração

- Envolve construir um sistema através de seus componentes e testá-lo em busca de problemas que podem ocorrer na interação entre os componentes
 - Integração Ascendente (*bottom up*)
 - Uso de *Drivers* (chamadores artificiais)
 - Desenvolve-se o esqueleto de um sistema que é populado por componentes
 - Integração Descendente (*top down*)
 - Uso de *Stubs* (subordinados artificiais)
 - Componentes de infra-estrutura são integrados e depois são adicionados os componentes funcionais
 - Integração híbrida (combinação das anteriores)
 - Para simplificar a localização de erros, os sistemas devem ser integrados de maneira incremental

©Pimenta 2011

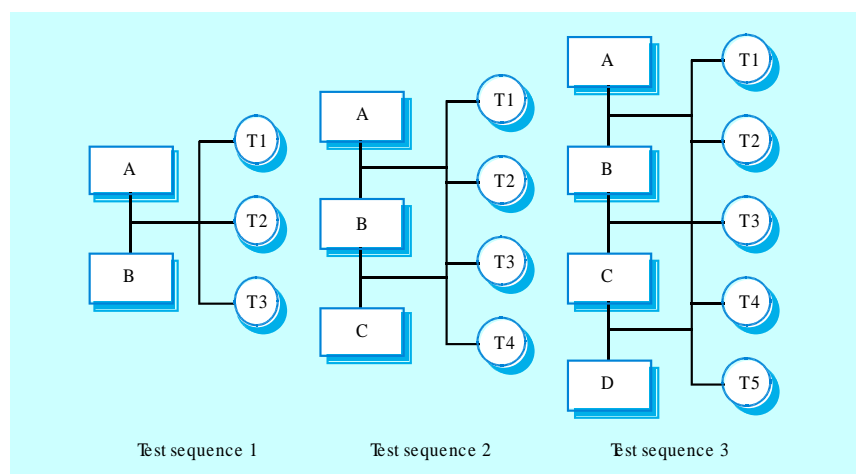
Teste de Integração



Teste de Integração = Teste de Interfaces

©Pimenta 2011

Testes Incrementais



©Pimenta 2011

Tipos de Interface

- Interfaces via Parâmetros
 - Dados passados de um procedimento a outro
- Interfaces via memória compartilhada
 - Bloco de memória é compartilhado entre diferentes funções
- Interfaces procedurais
 - Sub-sistema encapsula um conjunto de funções para ser chamado por outros subsistemas
- Interfaces via troca de mensagens
 - Sub-sistemas solicitam serviços de outros sub-sistemas

©Pimenta 2011

Erros típicos de Interface

- Interfaces mal utilizadas
 - Um componente chama outro componente e faz um erro no uso de sua interface , p.ex., parâmetros na ordem errada
- Interfaces mal compreendidas
 - Um componente embute suposições INCORRETAS sobre o comportamento de um componente chamado
- Erros de tempo
 - Os componentes chamador e chamado operam em diferentes velocidades e informações desatualizadas são acessadas

©Pimenta 2011

Diretrizes para teste de interface

- Projete testes de tal forma que parâmetros a um procedimento chamado estão nos extremos de seus intervalos
- SEMPRE teste parâmetros tipo ponteiro com valores nulos
- Projete testes que façam um componente falhar
- Use teste de stress em sistemas baseados em trocas de mensagens (p.ex. Web-based)
- Em sistemas que usam memória compartilhada, varie a ordem em que os componentes são ativados

©Pimenta 2011

Níveis de Teste

- As atividades de testes ocorrem ao longo do processo de desenvolvimento.
 - Teste de Aceitação
 - Teste de Sistema
 - Teste de Integração
 - Teste de Unidade (ou Teste Unitário)
- Teste de Regressão

©Pimenta 2011

Teste de Sistema

- Verificação global do sistema após a integração com outros sistemas que deverão operar juntos
- Processo de testar um sistema **feito pelo analista** para verificar se satisfaz seus requisitos especificados, ou seja, todos os requisitos funcionais e não funcionais.
- Teste do sistema deve ser realizado com todas as partes do sistema já integradas, operando conjuntamente. Portanto, sucede ao Teste de Integração.

©Pimenta 2011

Testes de Sistema

- Envolve integrar componentes de forma a criar um sistema ou sub-sistema
- Pode envolver testar um incremento a ser entregue a um cliente
- Duas fases:
 - Testes de integração: A equipe de testes deve ter acesso ao código fonte. O sistema é testado na medida em que os componentes são integrados
 - Testes de release: A equipe de testes testa o sistema completo a ser entregue como uma 'caixa-preta'

©Pimenta 2011

Ex. de Testes de sistema

1. Testar o mecanismo de login usando logins corretos e incorretos, de forma a verificar que usuários válidos são aceitos e usuários inválidos são rejeitados
2. Testar os mecanismos de busca usando diferentes consultas para fontes conhecidas, de forma a verificar se o mecanismo está encontrando os documentos
3. Testar os mecanismos de apresentação, de forma a verificar se as informações sobre documentos são mostradas corretamente
4. Testar o mecanismo de solicitação de permissão para download
5. Testar a resposta por email, indicando que o documento baixado está disponível

©Pimenta 2011

Teste de Aceitação

- Teste que envolve checagem do sistema em relação a seus requisitos iniciais feito pelo usuário, ou às necessidades do gestor que solicitou o sistema.
- É bastante similar ao teste do sistema; a diferença é que é realizado pelo usuário do sistema.
- Teste conduzido para determinar se um sistema satisfaz ou não seus critérios de aceitação e para permitir ao usuário ou gestor determinar se aceita ou não o sistema solicitado.
- **Requer participação do usuário** pois ele é o único que pode validar e aceitar!!

©Pimenta 2011

Testes de Release

- É o processo de testar um release de um sistema a ser implantado/distribuído aos clientes
- O principal objetivo é aumentar a confiança de que o sistema atende as necessidades do usuário
- Testes de release são normalmente testes funcionais, caixa-preta
 - Baseados apenas na especificação do sistema
 - Testadores não precisam ter conhecimento dos detalhes de implementação do sistema

©Pimenta 2011

E o que são Testes de Regressão?

- Estratégia de testes que tenta garantir que:
 - Os defeitos identificados em execuções anteriores dos testes foram corrigidos
 - As mudanças feitas no código não introduziram novos defeitos (ou re-ativaram defeitos antigos)
- Testes de regressão podem envolver a re-execução de quaisquer tipos de testes
- São feitos periodicamente, tipicamente relacionados a interações e testes anteriores

©Pimenta 2011

Tipos de Teste

- Atributos de qualidade em foco:
 - o Teste de funcionalidade
 - o Teste de interface
 - o Teste de segurança
 - o Teste de Carga (ou Teste de Stress)
 - o Teste de Desempenho (ou Teste de performance)
 - o Teste de Usabilidade
 - o Teste de Tolerância a Falha
 - o Teste de Instalação

©Pimenta 2011

Teste de Funcionalidade

- Verifica se as funcionalidades definidas na especificação de requisitos funcionais são executadas adequadamente, de acordo com o comportamento esperado.

©Pimenta 2011

Teste de Interface

- Visa verificação do software a partir da integração de suas unidades previamente testadas. Portanto é a verificação da correção da interconexão entre os módulos após o teste de unidade de cada um.
- Se as unidades estão previamente (e isoladamente) testadas, este teste visa descobrir erros associados a comunicação e interoperação entre unidades.
- É utilizado no teste de integração.
- Não confundir com teste de interface com usuário realizado no teste de usabilidade (ver adiante).

©Pimenta 2011

Teste de Segurança

- Verificação dos objetivos de segurança e privacidade específicos de um sistema.
- Visam definir casos de teste que subvertam deliberadamente as validações de segurança do sistema

©Pimenta 2011

Teste de Carga (ou Teste de Stress)

- Verificação do sistema quando submetido a volume pesado de processamento ou grandes taxas de utilização tipicamente utilizando-se ou simulando-se um pico de uso num pequeno período de tempo.

©Pimenta 2011

Teste de Stress

- Exercita o sistema além de sua carga máxima projetada. “Estressar” o sistema muitas vezes força os defeitos a virem à tona ;
- “Estressar” o sistema testa o comportamento em relação a falhas (robustez). Sistemas não devem falhar catastróficamente. Teste de stress verifica perdas inaceitáveis de serviço ou dados .
- Teste de Stress é particularmente relevante para sistemas distribuídos que podem exibir degradação de comportamento quando uma rede torna-se sobrecarregada

©Pimenta 2011

Teste de Performance (ou Teste de Desempenho)

- Verificação dos objetivos de eficiência e desempenho de um sistema, estabelecendo propriedades como tempo de resposta e taxas de carga sob certas condições de trabalho e configuração.

©Pimenta 2011

Testes de desempenho

- Parte dos testes de release envolvem testar propriedades de um sistema, como desempenho e tolerância a falhas
- Testes de desempenho envolvem planejar uma série de testes onde a carga é incrementalmente aumentada até que o desempenho da aplicação torne-se inaceitável

©Pimenta 2011

Teste de Usabilidade

- Avaliação da qualidade de uso (usabilidade) de um sistema.
- Diz respeito basicamente a avaliação da interface com usuário e envolve a checagem das facilidades/dificuldades de reconhecimento das funcionalidades disponíveis ao usuário, a facilidade de uso e aprendizagem de uso e à intuitividade.
- Teste realizado tendo em mente a lógica de uso do sistema por seus usuários, de preferência em situações reais de utilização.

©Pimenta 2011

Qualidade de Interfaces

- Qualidade historicamente ligada a 'amigabilidade' (*user friendliness*)
- Diferentes pontos de vista de qualidade de interfaces:
 - a) Performance humana satisfatória(ISO)
 - Eficácia (%) - coeficientes de erro
 - Eficiência (*t*, \$) - velocidade de uso
 - b) Tempo de aprendizado e de retenção

©Pimenta 2011

Usabilidade

- Adequação entre características (físicas/cognitivas) dos usuários e características da interação (com o sistema) para realização de tarefas
- Não é propriedade intrínseca do sistema mas do trio (usuário, sistema, tarefa)
- Expressa por alguns fatores:
 - **facilidade de aprendizado**, intuitiva e "natural"
 - **flexibilidade de interação**, multiplicidade de formas de uso
 - **robustez de interação**, acompanhamento e recuperação em situações de incidentes

©Pimenta 2011

Usabilidade

- Resumo: USABILIDADE É QUALIDADE DE USO
 - facilidade de aprendizado
 - facilidade de uso (operação)
 - taxa de erros minimizada
 - Satisfação dos usuários
 - adequação à tarefa
- Usabilidade é obtida por **construção**
 - Clara compreensão dos requisitos de usabilidade **durante** as etapas iniciais da concepção e não somente ao final
 - BUSCAR: Usabilidade como requisito do sistema ('built-in approach')
 - EVITAR: Usabilidade somente como critério de avaliação ('day-after approach')

©Pimenta 2011

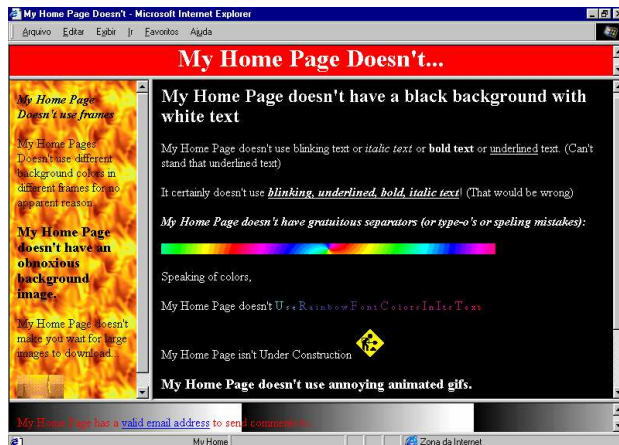
Problema de Usabilidade

- se há dificuldades (reais ou potenciais) para determinado usuário ou (grupo de usuários) realizar uma tarefa com a interface !!!
- Graus de severidade

Tipo	Descrição (necessidade de reparo)
0 Sem importância	Não afeta a operação da interface
1 Cosmético	Não há necessidade imediata de solução
2 Simples reparado)	Problema de baixa prioridade (<u>pode</u> ser reparado)
3 Grave reparado)	Problema de alta prioridade (<u>deve</u> ser reparado)
4 Catastrófico	PRIORIDADE MÁXIMA no reparo

©Pimenta 2011

Exemplos de problemas



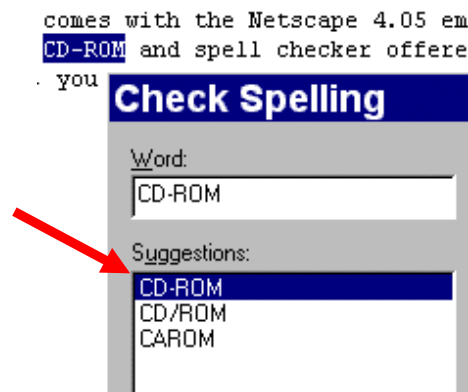
©Pimenta 2011

Exemplos de problemas



©Pimenta 2011

Exemplos de problemas



©Pimenta 2011

Exemplos de problemas

A screenshot of a form with two questions. The first question is "8) Age:" followed by an empty text input field. The second question is "9)" followed by two radio buttons. The first radio button is checked and labeled "Female". The second radio button is checked and labeled "Male".

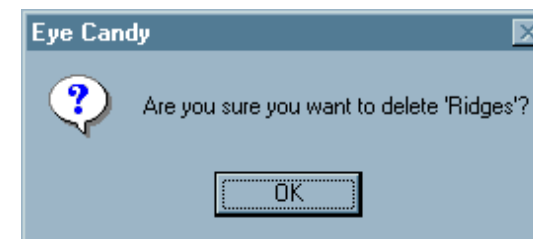
©Pimenta 2011

Exemplos de problemas



©Pimenta 2011

Exemplos de problemas



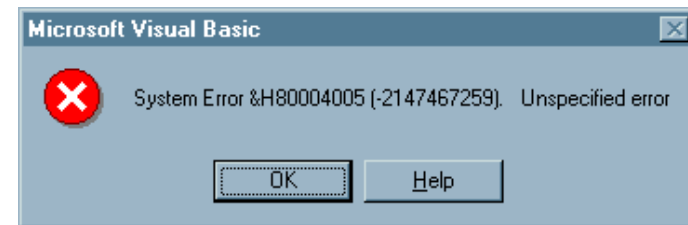
©Pimenta 2011

Exemplos de problemas



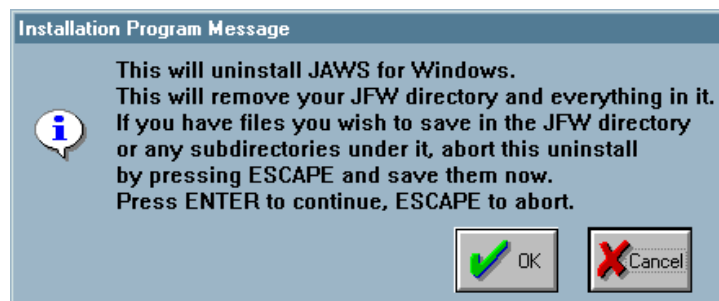
©Pimenta 2011

Exemplos de problemas



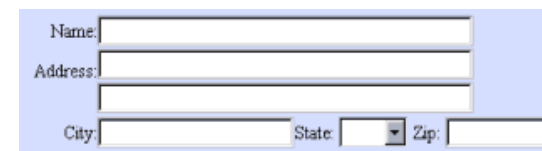
©Pimenta 2011

Exemplos de problemas



©Pimenta 2011

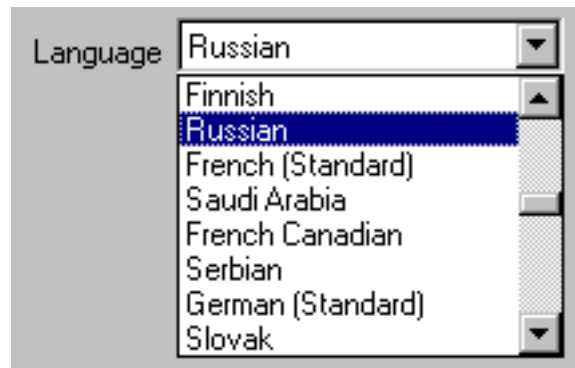
Exemplos de problemas

A screenshot of a form with five input fields. The first field is labeled "Name:", the second "Address:", the third "City:", the fourth "State:" (with a dropdown arrow), and the fifth "Zip:". The form has a light blue background.A screenshot of a form with five input fields. The first field is labeled "Name:", the second "Email:", the third "Address1:", the fourth "Address2:", and the fifth "Address3:". The form has a light blue background.

Internacional

©Pimenta 2011

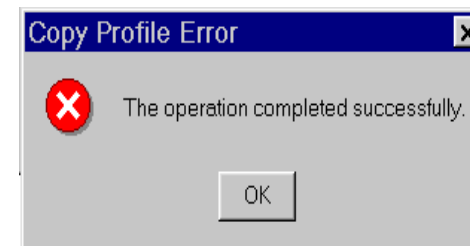
Exemplos de problemas



Onde achar “Portuguese”? Acima ou abaixo?

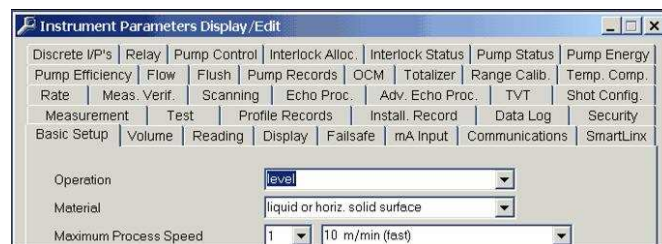
©Pimenta 2011

Exemplos de problemas



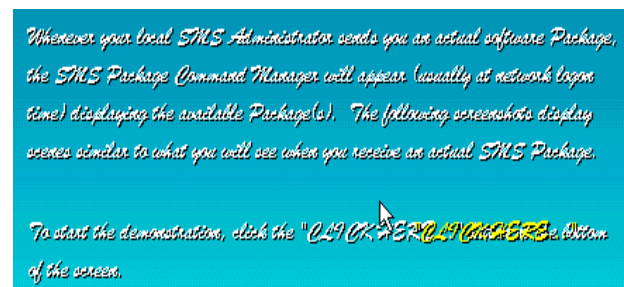
©Pimenta 2011

Exemplos de problemas



©Pimenta 2011

Exemplos de problemas



©Pimenta 2011

Exemplos de problemas



Problemas em Interfaces em geral: NÃO somente em Software !!!

Don Norman, The Design of Everyday Things,

©Pimenta 2011

Exercício provocativo

- Forme um grupo (2 a 3 membros)
- Escolha um aplicativo qualquer (interativo)
- Investigue-o procurando problemas (reais ou potenciais) de usabilidade
- Faça uma lista
- Iniciar trabalho em aula, terminar em casa, entregar na próxima aula (via email ou em papel)
 - Não esquecer os nomes dos componentes do grupo e o nome (e se possível URL com informacoes) do aplicativo
- **Atenção:** pare após 20 problemas ;-)

©Pimenta 2011

Métricas para medir usabilidade?

- Desempenho durante a realização de tarefas:
 - Conclusão de tarefas (c/ sucesso, parcialmente concluída, não-concluída);
 - Tempo de realização da tarefa;
 - Ocorrência de erros;
- Satisfação subjetiva do usuário e correspondência com os objetivos do usuário;
- Adequação a padrões (normas, recomendações, regras ergonômicas, etc.)
 - Internacionais (ISO)
 - Continentais (Comunidade Européia, MercoSul, etc)
 - Nacionais
 - Institucionais (Style guide)

©Pimenta 2011

Interessado em IHC?

- Disciplina INFO1043 – IHC, na **UFRGS** desde **1993**
- Livro:
 - Preece, J. et alli. *Design da Interação*, Bookman, 2005.
- Links interessantes:
 - Usabilidade: www.useit.com
 - Sun's guide to Web style
 - * <http://www.sun.com/980713/webwriting/index.html>
 - HCI Bibliography : <http://www.hcibib.org/>
- **Entre em contato com Prof. Marcelo Soares Pimenta**
Ou simplesmente procure por “Marcelo Pimenta” no Google...

©Pimenta 2011

Usabilidade tem futuro !!



©Pimenta 2011



©Pimenta 2011

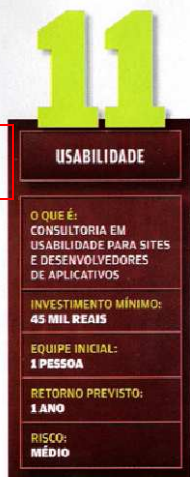
USABILIDADE DÁ DINHEIRO

CRESCE A PREOCUPAÇÃO DAS EMPRESAS COM SITES AMIGÁVEIS

→ Quantas vezes você já entrou num site e teve dificuldade para encontrar alguma informação absolutamente básica, como o telefone de contato da empresa? Falhas de usabilidade como essa são comuns, tanto na web como nos aplicativos que rodam nos computadores. Num estudo clássico sobre o tema, a pesquisadora americana Claire Karat demonstrou que cada dólar investido em usabilidade pode trazer um retorno entre 3 e 100 dólares. Essa regra tem sido comprovada em numerosos casos práticos e é o principal argumento de vendas de consultores especializados no tema. Em TI, as oportunidades incluem prestar serviços a sites e a desenvolvedores de aplicativos. Existem várias empresas atuando dessa forma, mas sobra espaço para mais gente. "O momento é ótimo. As pessoas estão deixando de ver a usabilidade como algo pitoresco", diz Amyris Fernandez, coordenadora dos cursos de usabilidade do IGroup.

Um pré-requisito para atuar como consultor nessa área é ter boa familiaridade com as tecnologias de desenvolvimento de aplicativos e sites. Assim, será mais fácil sugerir alterações

no produto para melhorá-lo. Também será preciso obter treinamento específico. A PUC-RJ (www.puc-rio.br/artes/esperadesign.htm) e a UFRGS (www.inf.ufgrs.br/esp/cursos/corso2.php) são duas das instituições que oferecem cursos de pós-graduação em usabilidade no Brasil. Além disso, há empresas, como o IGroup, que realizam seminários e cursos rápidos sobre esse tema, alguns via web. Um profissional capacitado precisa apenas de um notebook para começar a prestar consultoria. Com essa estrutura, é possível fazer a chamada avaliação heurística. Nela, o especialista verifica uma série de itens e determina o nível de usabilidade do aplicativo ou site. Depois, monta um relatório apontando os problemas e as soluções recomendadas. Um serviço mais completo exige um ambiente para testes, do tipo conhecido como sala de focus group. Nela, usuários vão utilizar o aplicativo ou site que está em estudo, seguindo um roteiro. Seus movimentos serão observados e filmados para análise. Como essa sala só é usada durante períodos curtos, é mais barato alugá-la de empresas que realizam pesquisas de mercado.



WWW.INFO.ABRIL.COM.BR | SETEMBRO 2007 | INFO 51

©Pimenta 2011

Teste de Tolerância a Falha

- Avaliação da capacidade do software de continuar fornecendo o serviço especificado mesmo na presença de falhas e da capacidade de tratamento ou recuperação destas falhas, explicando-as através de mensagens claras e explicativas.
- Está muito relacionada a requisitos de confiabilidade e robustez do sistema

©Pimenta 2011

Teste de Instalação

- Verifica se o sistema está com todas suas funções operacionais e roda corretamente no ambiente onde será efetivamente utilizado. Usado em testes de release.

©Pimenta 2011

Teste Dinâmico: Projeto de Casos de Teste

- Casos de Teste
- Dados de Teste
- Plano de Teste

©Pimenta 2011

Recomendações para Testes

- Recomendações auxiliam a escolher testes que revelam a presença de defeitos no software:
 - Escolha entradas que forcem o sistema a gerar todas as condições de erros
 - Escolha entradas que causem estouro de memória
 - Forçe que saídas inválidas sejam geradas
 - Faça com que os resultados da computação assumam seus valores máximos e mínimos
- Não jogue fora seus casos de teste a não ser que esteja também jogando fora seu programa!!

©Pimenta 2011

Projeto de casos de teste

- Envolve projetar os casos de teste (entradas e saídas) usadas para testar o sistema
- O objetivo do projeto de casos de teste é criar um conjunto de testes que sejam efetivos para Teste de Validação e Teste de Defeitos
- Enfoques de Projeto de casos de teste :
 - Teste baseado em requisitos (para teste funcional)
 - Casos de teste para exercitar teste estrutural

©Pimenta 2011

Testes baseados em requisitos

- Um princípio geral da engenharia de requisitos é que os requisitos devem ser testáveis
- Testes baseados em requisitos são uma técnica de validação na qual considera-se cada requisito e deriva-se um conjunto de casos de testes para estes requisitos

©Pimenta 2011

Diretrizes e dicas de teste

- Diretrizes de teste são dicas ou recomendações para a equipe de teste para auxiliá-las a escolher testes que de fato revelem defeitos no sistema
 - Escolha entradas que forcem o sistema a gerar todas mensagens de erro;
 - Projete entradas que causam overflow nos buffers e estruturas de dados compostas ;
 - Repita a mesma entrada ou séries de entrada várias vezes
 - Force que saídas inválidas sejam geradas
 - Force que resultados sejam muito pequenos ou muito grandes

©Pimenta 2011

3 dicas de Técnicas de design de testes

- Projeto de teste:
 - O que faz um bom teste ? Quanto teste se deve fazer?
- Dica 1:
 - Preste atenção a seu catálogo de erros
 - Erros se repetem - frequentemente

©Pimenta 2011

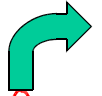
Dica 1 – faça um catálogo de erros

- Pessoas não são muito imaginativas quando cometem erros :
 - Tendem a cometer SEMPRE OS MESMOS erros de antes
- Projeto de testes é quase sempre o ato de verificar os erros historicamente plausíveis e deve verificar se o código faz tudo que era intencionado – catálogo é um bom início
- Dica 1: Crie um catálogo – sua memória de erros – e preste atenção a ele na hora de projetar testes
- Ex de catalogo <http://www.testing.com/writings/catalog.pdf>.
- Ex. de conteúdo de catálogo:
 - DESVIOS de fluxo – IFs, SWITCHs, etc
 - Estruturas recursivas
 - Ponteiros, etc
 - Todos tipos de entradas devem ser manipuladas diferentemente
 - Booleanos, inteiros , strings, ponteiros ou referências , etc
- CATÁLOGO tem idéias de teste , não obrigatoriamente TUDO vira teste em outra ocasião

©Pimenta 2011

Dica 2

- Erros prováveis: Expressões Booleanas



&&

if (publicClear || technicianClear) {

bomb.detonate();

}

Acharam o erro??

©Pimenta 2011

Dica 2 (cont)

- Teste1: Condição “public clear” (P) E “technician clear”(T)**

 $P = V, T = V, P \parallel T = V, P \&\& T = V$
- Teste2: Condição “técnico junto à bomba” e “tem público”**

 $P = F, T = F, P \parallel T = F, P \&\& T = F$
- Teste3: Condição “public clear” E “técnico junto à bomba”**

 $P = V, T = F, P \parallel T = V, P \&\& T = F$

©Pimenta 2011

Dica 2 (cont)

- Expressões booleanas mal formuladas

A||B&&C ?

- Poderia ser:

- (A||B)&&C
- A&&B&&C
- A||B&& !C
- A|| !B&&C
- A||B ||C
- A|| !(B&&C)
- !A||B&&C
- Ou mesmo A||D&&C

A dica consiste em tentar achar causas de erros prováveis na formulação destas expressões...Pode não achar erros mas aumenta a atenção sobre o uso deste tipo de expressão.

Test ideas for (a || (b && c)):

a	b	c
True	true	FALSE
FALSE	true	true
FALSE	FALSE	true
FALSE	true	FALSE

Ver <http://www.testing.com/tools.html>

©Pimenta 2011

Dica 2 (cont)

- ATENÇÃO !!!:**
- Expressões somente com &&**

 $A1 \&\& A2 \&\& \dots \&\& A_n$

- têm N+1 casos:
 - all conditions true
 - A1 FALSE, all others true
 - A2 FALSE, all others true
 - ...
 - An FALSE, all others true

A && B

A	B
V	V
F	T
T	F

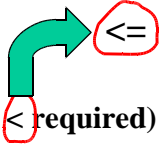
©Pimenta 2011

Dica 3

- Erros prováveis:
Expressões Relacionais
(>, <, etc)

Achando o erro:

finished	required	F < R	F <= R
0	100	V	V
5	5	F	T



- `...if (finished < required)`
`{`
- `siren.sound();`
- `}...`

©Pimenta 2011

Dica 3(cont)

- Sugestão de testes:

Se `expr A < B` ou `A >= B`:

- `A = B - e`
- `A = B`

No caso da expressão

`finished < required`

Se `expr A > B` ou `A <= B`

- `A = B`
- `A = B + e`

Deve have ao menos 2 testes:

- Se A e B inteiros, $e=1$
- Se A e B reais, $e < 1$

- 7) FINISHED um pouco menor que REQUIRED: V
- 8) FINISHED exatamente igual a REQUIRED: F

©Pimenta 2011

Resumo das dicas

- Dicas servem para auxiliar criatividade e não substituí-la
- Catálogo como gerador de idéias
- Converse com colegas e troque catálogos e idéias
- Tenha foco no usuário – erros aparecem a eles

©Pimenta 2011

Interpretando resultados dos testes (1)

- ...
- `if (a || b) {`
- `bomb.detonate(2 * time);`
- `} else {`
- `bomb.detonate(time * time);`
- ...
- E SE `time = 2` ???

©Pimenta 2011

Interpretando resultados dos testes (2)

- Mesmo se o programa produz resultado errado, às vezes NÃO notamos
- Pessoas interpretam os resultados de um programa incorreto e não percebem erros.
- A melhor forma de evitar isto é DECIDIR O QUE É CORRETO ANTES :
 - ESCREVER TESTES ANTES DE ESCREVER CODIGO
 - ESCREVER CRITERIOS DE ACEITACAO ANTES DE DESENVOLVER,

©Pimenta 2011

Tipos de erros comuns

- Erros de omissão
 - Não testar retornos de funções:
 - **File file = new File(stringName);**
 - **if (file.delete() == false) {...}**
 - O programador assume que sempre funciona ok. Bem, nem sempre

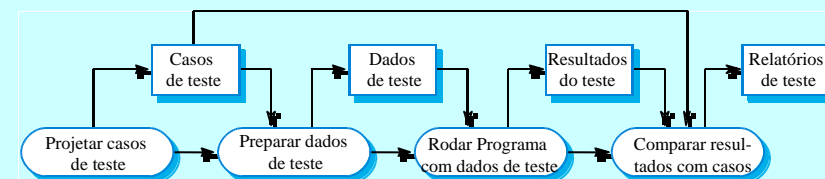
©Pimenta 2011

Erros de omissão

- Erros de omissão são comuns
 - A) Não testar retornos de funções:
 - **File file = new File(stringName);**
 - **if (file.delete() == false) {...}**
 - B) Não testar resultados de funções, códigos de erros e exceções
- O programador assume que sempre funciona ok. Bem, nem sempre
- 30% dos erros classificados da Microsoft são erros de omissão (dados de 2000)

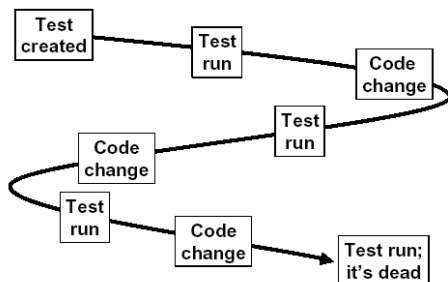
©Pimenta 2011

O processo de testes de software



©Pimenta 2011

Ciclo de vida dos testes



©Pimenta 2011

Exemplo de caso de teste (CT)

(Test Set #, CT #, Nome do CT, Condições, Status, Regras relacionadas, Obs)

TS01	TC04	Remove origin without shipments	Condition 1: Remove an origin for a TBN voyage Condition 2: Remove an origin for a FOB voyage Condition 3: Remove the last origin and add another origin Condition 4: Remove a TBN voyage that is already assigned to a berth				
	TC05	Remove an origin with all shipments belonging to user's trading office	Condition 1: User selects to move the removed the shipment to unassigned Condition 2: User selects to remove shipment permanently				
	TC06	Remove an origin with shipments for a trading office not assigned to the user	Condition 1: Some shipments belongs to user trading office and some does not belongs to it and selects to unassign shipments Condition 2: Some shipments belongs to user trading office and some does not belongs to it and selects to remove shipments Condition 3: All the shipments does not belong to users trading office				

©Pimenta 2011

Teste de Unidade Exercício

- Escolha Casos de Teste para teste funcional do seguinte programa
 - Seja um programa que leia três inteiros que corresponderiam aos lados de um triângulo, verifica se é efetivamente um triângulo, e em caso positivo classifica-o em triângulo escaleno, isósceles e equilátero

©Pimenta 2011

Teste de Unidade Respostas do Exercício

- Casos de teste das características principais:
 - Equilátero 10,10,10
 - Isósceles 10,10,17 10,17,10 17,10,10
 - Escaleno 8,10,12 8,12,10 10,12,8
 - Não é Triângulo 10,10,22 10,22,10 22,10,10

©Pimenta 2011

Casos de Teste usando Template

(Test Set #, CT #, Nome do CT, Condições, [Status, Regras relacionadas, Obs])

(1, 1, Caract.Principal 1 – Equilátero, cond1 – ok normal, [-, -, -]
cond2 – lados zerados, [-, -, -])

(1, 2, Caract. Principal 2 – Isósceles, cond1 – ok normal a,a,b []
cond2 – ok normal a,b,a []
cond3 – ok normal a,b,b [])

(1,3, Caract Principal 3 – Escaleno, cond1 – ok normal [])

(1,4, Caract Principal 4 – não triângulo, cond 1 – ok normal [])

©Pimenta 2011

Teste de Unidade

Respostas do Exercício

- Casos de teste do domínio de dados
 - Casos 5 a 8: Executar os testes 1 a 4 acima incluindo casos que contenham o menor (MININT) e o maior (MAXINT) valores inteiros aceitáveis pelo programa

Dados Anormais

9. Zero 0,0,0
0, 0, 25 25,0,0 0, 25,0

10.Negativos -10,-10,-10
-10,-10,15 -10,15,-10 15, -10,-10
-8,10,17 -8,17,10 17,-8,10

©Pimenta 2011

Teste de Unidade

Respostas do Exercício

Dados Anormais

11. Omitindo dados _,-, _ 10,-, _ _,-,15, _
_,-, 25 10,5, _ _,-,10,5

10.Incorretos A,B,C =,+,-
8,10,A A,8,10 8,A,10
7E3, 10.5, A 10.5, 7E3, A
A, 10.5, 7E3

Limites Máximo e mínimo:

- incluir nos casos de teste inteiros com valores próximos acima e abaixo dos valores máximo e mínimo aceitos pelo programa

©Pimenta 2011

Rumo a Automação de Testes

- Ferramentas de teste de software
- JUNit e similares – idéias e conceitos !!!

©Pimenta 2011

Princípios

- Os testes devem ser:
 - Automatizados (tanto quanto possível)
 - Repetíveis
 - Auto-verificáveis

©Pimenta 2011

Automação de testes

- Testes são atividades caras. Workbenches de testes provêm ferramentas para reduzir o tempo necessário e os custos de testes
- Sistemas como Junit suportam a execução automática de testes
- A maioria dos workbenches de teste são sistemas abertos, já que as necessidades de testes variam conforme a organização
- Normalmente podem ocorrer problemas ao integrar com ambientes fechados para análise e projeto

©Pimenta 2011

Introdução

- Junit é um framework de testes de regressão desenvolvido por Erich Gamma e Kent Beck
- É usado por desenvolvedores que utilizam testes unitários em Java
- É um software open source, disponível como um projeto sourceforge:
 - <http://www.junit.org>

©Pimenta 2011

Porque usar Junit?

- Testes automatizados provam que funcionalidades estão corretamente implementadas
- Testes mantêm seu valor com o passar do tempo
- Permite que outras pessoas verifiquem se o software ainda está funcionando após mudanças
- Melhora a confiança e a qualidade da aplicação
- Efetivo, open source, integrado

©Pimenta 2011

O que é JUnit?

- JUnit é uma ferramenta que suporta a criação e execução de testes de unidade
- JUnit estrutura os testes e provê mecanismos para executá-los automaticamente
- Provê ferramentas para:
 - Asserções
 - Rodar testes
 - Agregar testes (suites)
 - Mostrar resultados

©Pimenta 2011

Asserções disponíveis no JUnit

Table 1.2 The JUnit class `Assert` provides several methods for making assertions.

Method	What it does
<code>assertTrue(boolean condition)</code>	Fails if condition is false; passes otherwise.
<code>assertEquals(Object expected, Object actual)</code>	Fails if expected and actual are not equal, according to the <code>equals()</code> method; passes otherwise.
<code>assertEquals(int expected, int actual)</code>	Fails if expected and actual are not equal according to the <code>==</code> operator; passes otherwise. There is an overloaded version of this method for each primitive type: <code>int</code> , <code>float</code> , <code>double</code> , <code>char</code> , <code>byte</code> , <code>long</code> , <code>short</code> , and <code>boolean</code> . (See Note about <code>assertEquals()</code> .)
<code>assertSame(Object expected, Object actual)</code>	Fails if expected and actual refer to different objects in memory; passes if they refer to the same object in memory. Objects that are not the same might still be equal according to the <code>equals()</code> method.
<code>assertNull(Object object)</code>	Passes if object is null; fails otherwise.

©Pimenta 2011

Idéias do Teste Automatizado

- A filosofia é:
 - Deixar que os desenvolvedores escrevam os testes
 - Tornar fácil o desenvolvimento de testes
 - Testar cedo e testar sempre
- Permite experimentar diferentes idéias de projeto (sem quebrar o q estava funcionando):
 - Inicie com “o mais simples que possa funcionar”
 - Refine o projeto através do uso de padrões e aplicação de refatorações
- Tentar quebrar o ciclo:
 - Mais pressão, Menos testes
- Menos tempo com atividades de depuração

©Pimenta 2011

Como Usar o JUnit

- Escreva uma classe que estende *TestCase*
 - Para ter acesso aos métodos privados, coloque a classe de teste no mesmo pacote da classe que está sendo testada
- Cada teste é um método
- O conjunto de testes a serem executados é definido por `public static Test suite()`

©Pimenta 2011

Síntese - Pontos Chave

- Testes podem mostrar a presença de falhas em um sistema
 - Não podem provar que não existem falhas
- Desenvolvedores de componentes são responsáveis pelos testes de componentes.
 - Testes de sistema normalmente são realizados por uma equipe separada de testes
- Testes de integração são testes realizados para incrementos de um sistema

©Pimenta 2011

Síntese - Pontos Chave

- Testes de release envolve testar um sistema a ser entregue/implantado em um cliente
- Utilize a sua experiência e recomendações existentes para projetar casos de testes
- Testes de interface são projetados para descobrir defeitos nas interfaces de componentes compostos

©Pimenta 2011

Síntese - Pontos Chave

- Partições de equivalência são uma maneira de descobrir casos de testes
 - Todos os casos de teste em uma mesma partição devem se comportar da mesma maneira
- Análise estrutural enfoca em analisar um programa e derivar casos de testes
- A automação das atividades de testes reduzem os custos relacionados a testes

©Pimenta 2011

Exercício 1

- Definir um conjunto de possíveis testes para a seguinte situação:
 - O sistema de avaliação de uma disciplina obedece aos seguintes critérios:
 - Durante o semestre são dadas três notas;
 - A nota final é obtida pela média aritmética das notas dadas durante o curso;
 - É considerado aprovado o aluno que obtiver a nota final superior ou igual a 60 e que tiver comparecido a um mínimo de 40% das aulas.
 - Deve ser possível para este sistema:
 - Calcular a nota final de cada aluno, a maior e a menor nota da turma, bem como a média da turma.
 - Calcular o total de alunos reprovados
 - Calcular a porcentagem de alunos reprovados por frequência

©Pimenta 2011

Exercício 2

Para a situação a seguir, leia a descrição do problema, a sequência de eventos obtida e as pré e pós-condições. Elabore um conjunto de casos de testes para este caso em especial

©Pimenta 2011

Exercício 2

- Para que um aluno possa matricular-se em uma disciplina, ele dirige-se ao setor responsável e informa seu número de matrícula. Após confirmar seus dados, a atendente solicita que o aluno informe quais disciplinas gostaria de cursar. Como cada uma das disciplinas tem pré-requisitos distintos, estes são verificados um a um pela atendente. Caso tudo corra com sucesso, o aluno sai do setor com um comprovante de matrícula e um carnê de pagamento. Caso contrário, o aluno deve deixar a área de atendimento e verificar os pré-requisitos para as disciplinas selecionadas com mais cuidado. Após a verificação, este pode retornar e efetuar a matrícula.

©Pimenta 2011

Exercício 2 - Resposta

- Sequência Típica de Eventos:
 - 1. O aluno dirige-se ao setor de matrículas e informa seu número de matrícula.
 - 2. O sistema solicita ao aluno que informe quais disciplinas gostaria de cursar.
 - 3. Os pré-requisitos são verificados pelo sistema
 - 4. São emitidos e entregues ao aluno um comprovante de matrícula e um carnê de pagamento.

©Pimenta 2011

Exercício 2 - Resposta

- Pré-condições:
 - O número de matrícula deve ser informado
 - O número de matrícula deve ser válido
 - Deve ser informado um conjunto de disciplinas a serem cursadas
- Pós-condições:
 - Um comprovante de matrícula e um carnê de pagamento foram gerados
 - Foi criado um registro no sistema contendo os dados da matrícula

©Pimenta 2011

Etapa: Manutenção

©Pimenta 2011

Manutenção, por quê?

“Grande parte do software que dependemos atualmente tem em média de 10 a 15 anos. Mesmo quando esses programas foram criados, usando as melhores técnicas de projeto e codificação conhecidas na época [e muitos não o foram], o tamanho do programa e o espaço de armazenamento eram preocupações importantes. Depois migraram para novas plataformas, foram ajustados para modificações na tecnologia de máquina e de sistemas operacionais e foram melhorados para satisfazer as novas necessidades do usuário – tudo sem preocupação suficiente com a arquitetura global. O resultado são estruturas malprojetadas, malcodificadas, de lógica pobre e maldocumentadas em relação aos sistemas de software, para os quais somos chamados a fim de mantê-los rodando...”

Osborne e Chikofsky apud Presman (2003)

©Pimenta 2011

So what?

- Processo de Engenharia Orientada a Objetos nem prevê manutenção!
- Processos ágeis também não (explicitamente).
- A verdade é que ela existe...
 - Novos requisitos surgem
 - Problemas não previstos aparecem
 - Há programas mal feitos
 - ...
- **Todo o software lançado tem um arquivo “known issues” ou “readme” que descreve seus problemas (resolvidos ou por resolver)**

©Pimenta 2011

So What?

- Manutenção é cara e inevitável pois:
 - há muito SW legado (mais de 5 anos de uso)
 - implantação é incompleta (necessita ajustes)
 - sistemas mudam (refletem ambientes que mudam)
- Y2K

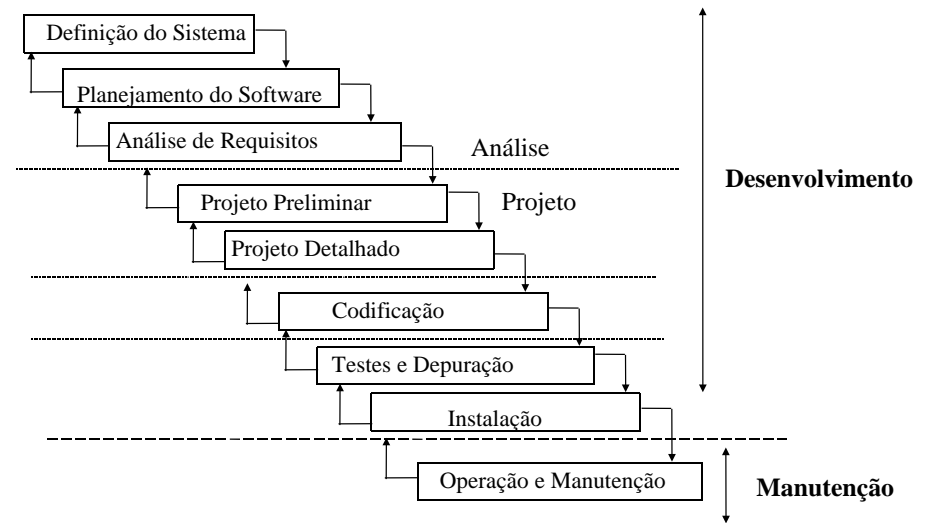
©Pimenta 2011

Por que existem problemas com manutenção?

- quase 80 % do software existente foi produzido sem o uso efetivo de técnicas de engenharia de software no seu desenvolvimento;
- é difícil determinar os "efeitos colaterais" da mudança, ou seja, se uma mudança em uma parte do código afetará outra parte e como.
- principal dificuldade em fazer manutenção é que não se pode fazer manutenção em um sistema que não foi projetado para manutenção, ou seja, que não tenha manutenibilidade.

©Pimenta 2011

Modelo Tradicional (“Waterfall”) tem fase específica!



©Pimenta 2011

Modelo Tradicional (“Waterfall”)

- ‘Desserviço’ à manutenção
- Manutenção após a entrega:
 - sem relação direta com desenvolvimento
 - modelos, técnicas e ferramentas da Engenharia de Software aplicam-se melhor ao desenvolvimento

©Pimenta 2011

Manutenção - Motivação

- Desenvolvimento: 30%
- Manutenção: 70%
- Manutenção é cara e inevitável pois:
 - há muito SW acumulado (mais de 5 anos de uso)
 - implantação é incompleta (necessita ajustes)
 - sistemas mudam (refletem ambientes que mudam)

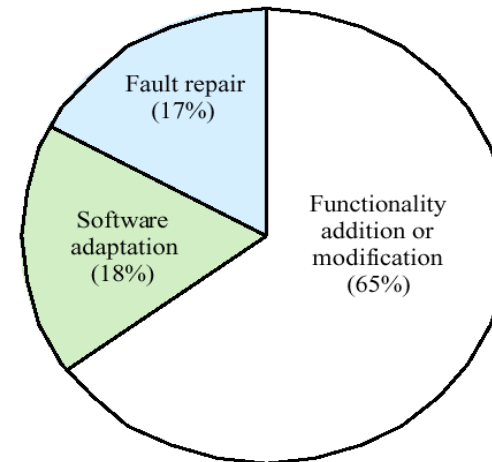
©Pimenta 2011

Manutenção - Definições

- **Manutenção:** Modificação de um software após a entrega
 - **Manutenção Corretiva** : para corrigir falhas ou deficiências detectadas.
 - **Manutenção Expansiva**: para expandir a funcionalidade , modificar a performance ou outros atributos em resposta a novos requisitos do usuário.
 - **Manutenção Adaptativa**: para adaptar o produto a uma mudança do ambiente (mudança de plataforma, hardware, sistemas operacional, SGBD, versão do compilador, linguagem, etc).
- **Manutenibilidade:** A facilidade que um sistema possui para que sua manutenção seja realizada.
 - **Manutibilidade:** Reparabilidade + Evolutibilidade
 - **Reparabilidade:** correção de erros com trabalho limitado
 - **Evolutibilidade:** adaptar e expandir com trabalho limitado

©Pimenta 2011

Distribuição do esforço



©Pimenta 2011

Como fazer?

- **Manutenção de software**
 - Resposta a requisitos modificados
 - Estrutura permanece
- **Transformação de arquitetura**
 - Mais radical, propõe alterações significativas
 - Arquitetura centralizada → cliente/servidor
- **Reengenharia de software**
 - Funcionalidades mantidas
 - Tornar mais fácil compreensão e alteração
 - Tipos: tradução de código, engenharia reversa, melhoria de arquitetura (*light*), modularização, reengenharia de dados

©Pimenta 2011

Problema com o pessoal que realiza a manutenção

- **Mito:** manutenção não é uma atividade criativa, não há desafios que envolvam criatividade do pessoal de manutenção.
- **Prática:** Manter código mal documentado, com uma estrutura incompreensível e com representações de dados espalhadas pelos sistema. Manter software torna-se um trabalho de detetive e cada tentativa de mudança pode acrescentar erros misteriosos ao já problemático programa.
- **Manutenção como 'punição' ou aprendizado** (aprender fazendo manutenção)

©Pimenta 2011

Por que a Manutenção é difícil e cara?

- i) Mudanças não são adequadamente documentadas;
- ii) Mudanças implicam em prováveis "efeitos colaterais";
- iii) Existe uma visão míope de manutenção estritamente como uma atividade com a qual deve-se preocupar apenas após a entrega do software;
- iv) Manutenção é cara: atividade intensivamente centrada nas pessoas com salários (teoricamente) altos e não automatizada
- v) Dificuldade de "rastrear" o produto e o processo que o criou.

©Pimenta 2011

Rastreamento

Habilidade de identificar e acompanhar o histórico de determinadas características técnicas do sistema (fontes de requisitos, origem de documentos, decisões de projeto, , erros, definições de estruturas, códigos, etc.) de modo a permitir verificar se foi definida na implementação, projeto, especificação ou até na definição de requisitos do sistema.

©Pimenta 2011

Manutenção - Modelos

- Modelo de Lehman
 - Evolução de Programa e **não mais** Desenvolvimento + Manutenção:
 - Mudanças são intrínsecas ao software e ao seu ciclo de vida;
 - Não há razão para distinguir manutenção do desenvolvimento;
 - Mudança é continuidade do desenvolvimento.

©Pimenta 2011

Leis de Lehman

Law	Description
Continuing change	A program that is used in a real-world environment necessarily must change or become progressively less useful in that environment.
Increasing complexity	As an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to <u>preserving and simplifying the structure</u> .
Large program evolution	Program evolution is a self-regulating process. System attributes such as size, time between releases and the number of reported errors are approximately invariant for each system release.
Organisational stability	Over a program's lifetime, its rate of development is approximately constant and independent of the resources devoted to system development.
Conservation of familiarity	Over the lifetime of a system, the incremental change in each release is approximately constant.

©Pimenta 2011

Como diminuir?

- Melhoria da qualidade do SW produzido
- **Melhoria do processo de produção de SW**
- **Ambas associadas à uma Mudança de Cultura do desenvolvimento do software**

©Pimenta 2011

Como diminuir?

- Devemos seguir metodologias, processos etc. que minimizem a manutenção/reengenharia após o *release*:
 - Processos ágeis
 - Refatoração
 - **Testes (*Test-Driven Development*)**

©Pimenta 2011

Métodos para Melhorar a Manutenção

- Enfoques de Projeto
 - Projetar SW com manutibilidade em mente;
 - Reavaliar o projeto em relação à complexidade: SW simples é mais fácil para manutenção
 - Gerenciamento de mudanças deve ser usado para:
 - limitar os efeitos na fase de manutenção de uma mudança feita na fase de projeto;
 - determinar os efeitos colaterais da mudança de um módulo:
 - variáveis globais ao programa
 - variáveis locais ao módulo
 - módulos chamados e chamadores

©Pimenta 2011

Métodos para Melhorar a Manutenção

- Práticas de Manutenção
 - Gerenciamento de mudanças:
 - Fazer as mudanças mais fáceis primeiro
 - Mudar um módulo de cada vez
 - Avaliar as mudanças propostas para cada tipo de efeito colateral
 - Efetuar testes após cada mudança
 - Produzir diretrizes para modificar e testar o SW:
 - Associar os trechos de código modificados às requisições de mudança;
 - Aprender a ler código desenvolvido por outrem
 - Manter um diário de erros e soluções encontradas
 - Documentar a manutenção pensando na próxima manutenção

©Pimenta 2011

Métodos para Melhorar a Manutenção

- Práticas de Manutenção (cont.)
 - Uso de ferramentas (p.ex. Engenharia Reversa, rastreamento, comparação de versões, depuração)
- Políticas de Gerência da Manutenção
 - Envolver o Pessoal de Manutenção (PM) em projeto e teste
 - Enfatizar o uso de padrões tanto no projeto quanto na manutenção
 - Fazer um ‘rodízio’ entre PM e o pessoal do desenvolvimento
 - Tornar a documentação de projeto disponível ao PM durante o desenvolvimento
 - Promover o uso de ferramentas CASE também na manutenção
 - Aumentar a ‘visibilidade’ da manutenção na empresa

©Pimenta 2011

Métodos para Melhorar a Manutenção

- Políticas de Gerência da Manutenção (cont.)
 - Usar procedimentos documentados de controle de configuração e de solicitação de mudanças de sistemas
 - Repartir as recompensas contratuais também com o PM, inclusive treinamento e prêmios de produtividade
 - Criar um orçamento prudente e prazos coerentes para a manutenção
 - Promover a participação do PM no desenvolvimento de padrões de desenvolvimento, documentação, avaliação do desenvolvimento e preparação dos padrões de testes de aceitação
 - Combater a imagem negativa da manutenção e do PM

INTEGRAR PROJETO e MANUTENÇÃO

©Pimenta 2011

Conduzindo a Manutenção

- Não há metodologia para manutenção:
 - Metodologias de desenvolvimento não são apropriadas à manutenção (considerada apenas nas consequências)
- Procedimento sugerido:
 - Desenvolver um plano de manutenção e de avaliação do aumento da manutabilidade
- Além de desenvolver SW com manutenção em mente, deve-se MANTER SW com manutenção em mente

©Pimenta 2011

Etapa: Gestão de Configuração

©Pimenta 2011

Gestão de Configuração de Software

- Introdução da atividade;
- Apresentação de ferramentas;
- Ótica para o modelo de componentes;
- Modelo de abstração para aspectos mensuráveis.

©Pimenta 2011

Introdução

- O que é GCS?
Trata-se de uma atividade que visa garantir a integridade dos artefatos de software.
- Qual é seu objetivo?
Disciplinar o desenvolvimento prático e efetivo de artefatos de software.

©Pimenta 2011

Introdução



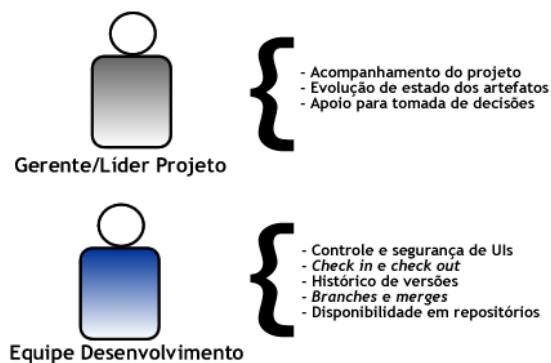
©Pimenta 2011

Introdução

- Quais os subsídios que ela provê?
Nível Gerencial = acompanhamento de métricas para o projeto;
Nível Técnico = controle e segurança sobre os artefatos do produto de software.
- Independência do modelo de desenvolvimento
Estruturado, Orientado a Objeto (RUP ou XP) e outros

©Pimenta 2011

Introdução



©Pimenta 2011

Introdução

- Aspectos da gerência
 - Item de Configuração
 - Baseline
 - Auditoria de Baseline
 - Papéis

©Pimenta 2011

Introdução

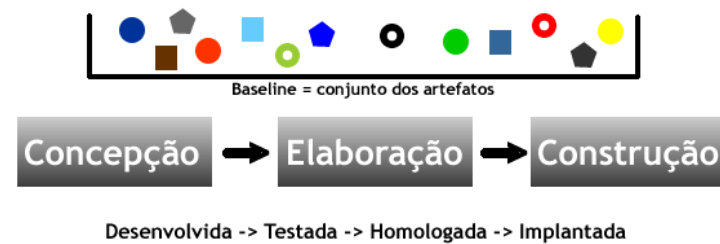
- Item de Configuração



©Pimenta 2011

Introdução

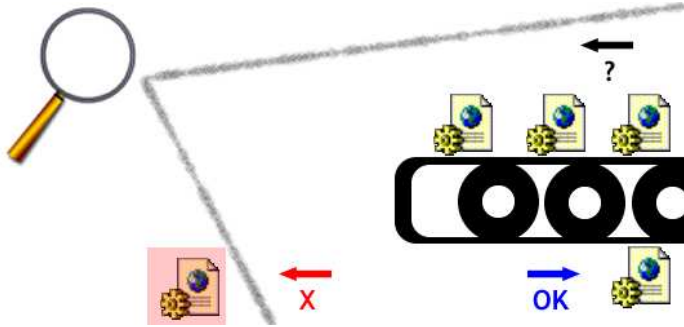
- Baseline



©Pimenta 2011

Introdução

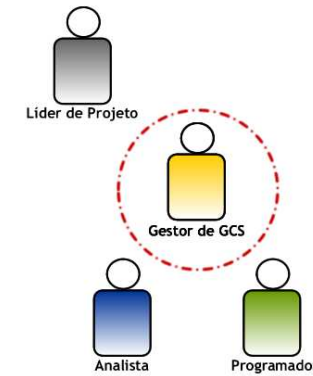
- Auditoria de Baseline



©Pimenta 2011

Introdução

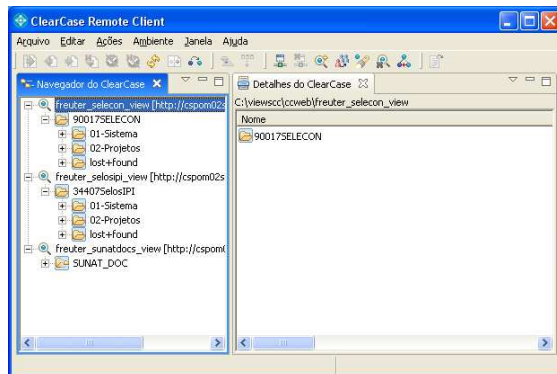
- Papéis



©Pimenta 2011

Ferramentas

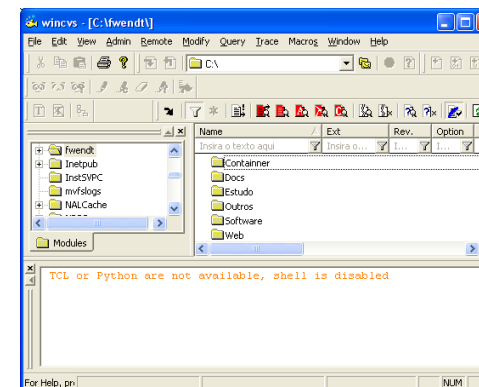
- IBM Rational ClearCase



©Pimenta 2011

Ferramentas

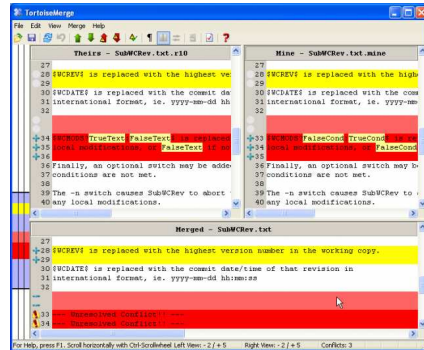
- CVS



©Pimenta 2011

Ferramentas

- Subversion (SVN)



©Pimenta 2011

GCS: A realidade em 2007

- É uma atividade de extrema relevância em processos de desenvolvimento de software;
- Apresenta ferramentas que facilitam a sua prática, embora muito mais a nível técnico do que gerencial;
- Sua mensuração de custo/benefício ainda não apresenta um modelo concreto e definido, além de apresentar algumas pendências frente a alguns modelos de desenvolvimento;

©Pimenta 2011

GCS: A realidade em 2007

- O investimento realizado em suas práticas é tratado como um aspecto de segurança e garantia frente a adversidades, apresentado mais um benefício implícito ao processo;
- O acompanhamento do projeto encontra apoio na atividade para a tomada de decisões corretivas e preventivas.

©Pimenta 2011