



MÓDULO DE:

TÓPICOS AVANÇADOS de ENGENHARIA de SOFTWARE

AUTORIA:

Ms. Carlos Valente

Copyright © 2008, ESAB – Escola Superior Aberta do Brasil

Módulo de: TÓPICOS AVANÇADOS de ENGENHARIA de SISTEMAS

Autoria: Ms. Carlos Valente

Primeira edição: 2008

Todos os direitos desta edição reservados à
ESAB – ESCOLA SUPERIOR ABERTA DO BRASIL LTDA
<http://www.esab.edu.br>
Av. Santa Leopoldina, nº 840/07
Bairro Itaparica – Vila Velha, ES
CEP: 29102-040
Copyright © 2008, ESAB – Escola Superior Aberta do Brasil

Apresentação

A Engenharia de Sistemas ainda continua em plena expansão e a cada dia apresenta novas técnicas e recursos. A Engenharia de Software como seu principal expoente apresenta tópicos especiais dos quais o profissional da área precisa para estar sempre atualizado.

O bjetivo

Apresentar tópicos mais avançados de Engenharia de Software, conforme o SWEBOK, destacando aqueles mais atuais ou com tendências para o futuro. Destacar a parte de segurança, qualidade, gerenciamento de pessoal técnico e sistemas para a Web

Ementa

SWEBOK, Estilos Arquiteturais, Engenharia de Proteção, Interface com Usuário, Gestão de Pessoal, ISO 9001, Sistemas Críticos, Arquiteturas tolerantes a falhas, Estratégias de Teste de Software, Softwares de Tempo Real, Engenharia de Serviços, Estimativa de custo de Software, Métricas para pequenas Organizações, COCOMO II, Engenharia de Software para a WEB, Ferramentas de Gestão de Projetos para a WEB

Sobre o Autor

- Professor e Consultor de Tecnologia de Informação
- Doutorando (ITA) e Mestre (IPT) em Engenharia de Computação, Pós-Graduado em Análise de Sistemas (Mackenzie), Administração (Luzwell-SP), e Reengenharia (FGV-SP). Graduado/Licenciado em Matemática.
- Professor e Pesquisador da Universidade Anhembi Morumbi, UNIBAN, e ESAB (Ensino a Distância). Autor de livros em Conectividade Empresarial. Prêmio em E-Learning no Ensino Superior (ABED/Blackboard).
- Consultor de T.I. em grandes empresas como Sebrae, Senac, Granero, Transvalor, etc. Viagens internacionais: EUA, França, Inglaterra, Itália, Portugal, Espanha, etc.

SUMÁRIO

UNIDADE 1	7
Swebok – Software Engineering Body Of Knowledge	7
UNIDADE 2	14
Taxonomia De Estilos Arquiteturais	14
UNIDADE 3	19
Engenharia De Proteção.....	19
UNIDADE 4	23
Diretrizes Para Um Projeto Seguro	23
UNIDADE 5	28
Projeto De Interface Com O Usuário.....	28
UNIDADE 6	34
Gerenciamento De Pessoal	34
UNIDADE 7	38
Modelos De Gerenciamento De Pessoal.....	38
UNIDADE 8	42
Gerenciamento De Qualidade	42
UNIDADE 9	47
Padrão De Qualidade - Iso 9001	47
UNIDADE 10	51
Desenvolvimento De Sistemas Críticos.....	51
UNIDADE 11	55
Desenvolvimento De Sistemas Críticos (Continuação).....	55
UNIDADE 12	58
Arquiteturas Tolerantes A Defeitos.....	58
UNIDADE 13	62
Estratégias De Teste De Software	62
UNIDADE 14	65
Espiral Dos Testes De Software.....	65
UNIDADE 15	68

Testes Caixa-Preta E Caixa-Branca.....	68
UNIDADE 16	72
Métodos De Teste Orientado A Objetos.....	72
UNIDADE 17	75
Projeto De Software De Tempo Real	75
UNIDADE 18	80
Engenharia De Software Orientada A Serviços.....	80
UNIDADE 19	84
Engenharia De Serviços.....	84
UNIDADE 20	89
Desenvolvimento De Software Orientado A Aspectos.....	89
UNIDADE 21	93
Engenharia De Software Com Aspectos	93
UNIDADE 22	96
Estimativa De Custo De Software	96
UNIDADE 23	100
Métricas Para Pequenas Organizações	100
UNIDADE 24	105
Estimativa Do Projeto De Software	105
UNIDADE 25	109
Modelo De Estimativa De Software – Cocomo Ii	109
UNIDADE 26	113
Engenharia De Software Para A Web	113
UNIDADE 27	119
Engenharia De Software Para A Web – Metodologia.....	119
UNIDADE 28	122
Engenharia De Software Para A Web – Oohdm.....	122
UNIDADE 29	127
Planejamento Da Webapp	127
UNIDADE 30	130
Ferramentas De Software Para Gestão De Projetos Web.....	130
GLOSSÁRIO	135
REFERÊNCIAS	136

UNIDADE 1

Swebok – Software Engineering Body Of Knowledge

Objetivo: Destacar a importância e as principais divisões que o SWEBOK apresenta quanto à Engenharia de Software.

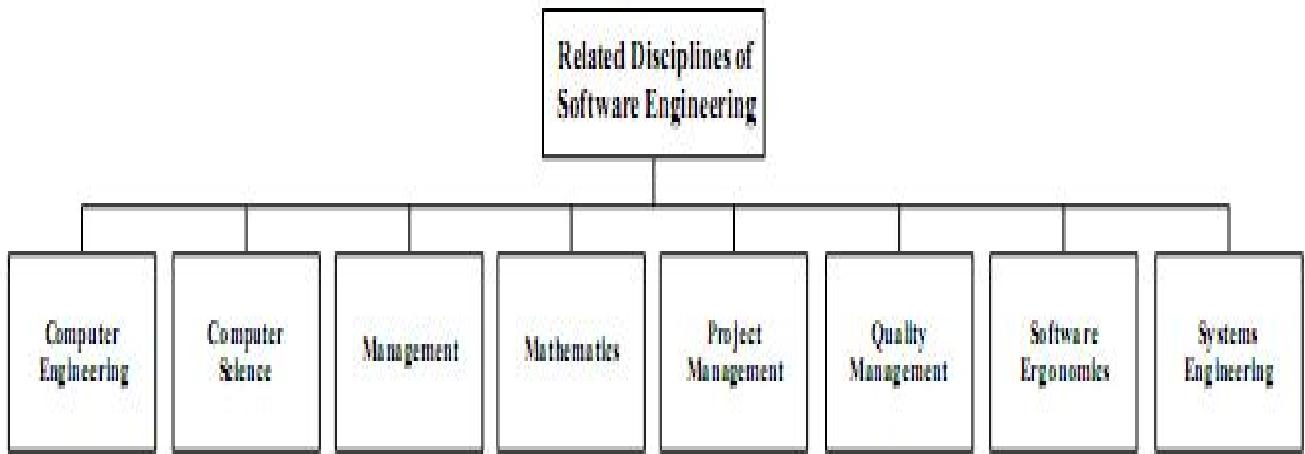
É provável que o projeto SWEBOK torne-se tão importante dentro da área de Tecnologia da Informação quanto o PMBOK é dentro da área de Gerência de Projetos. Seus padrões se transformarão em guias para o profissional de Tecnologia da informação.

Até última versão do SWEBOK, versão 2004, houve a participação de nove pesquisadores brasileiros na sua elaboração. E em seu painel de especialistas estavam relacionados três profissionais renomados, dentre eles Pressman e Sommerville.

Podemos definir Engenharia de Software como sendo: “Engenharia de Software é uma área de interesse (disciplina) preocupada com a criação e manutenção de aplicações de software pela aplicação de tecnologias e práticas da ciência da computação, gerência de projetos, engenharia, domínios de aplicação e outros campos”. Mas, a Engenharia de software é uma disciplina que está em desenvolvimento e existe uma grande tendência ao aumento no seu nível de maturidade, mas não é uma disciplina legítima de engenharia, nem uma profissão reconhecida (SWEBOK, 2004).

Os principais objetivos que o SWEBOK se propõe são:

- Promover uma visão consistente da engenharia de software no âmbito mundial.
- Clarear e marcar as fronteiras entre a engenharia de software e as outras disciplinas relacionadas (ciência da computação, gerência de projetos, matemática, entre outros).



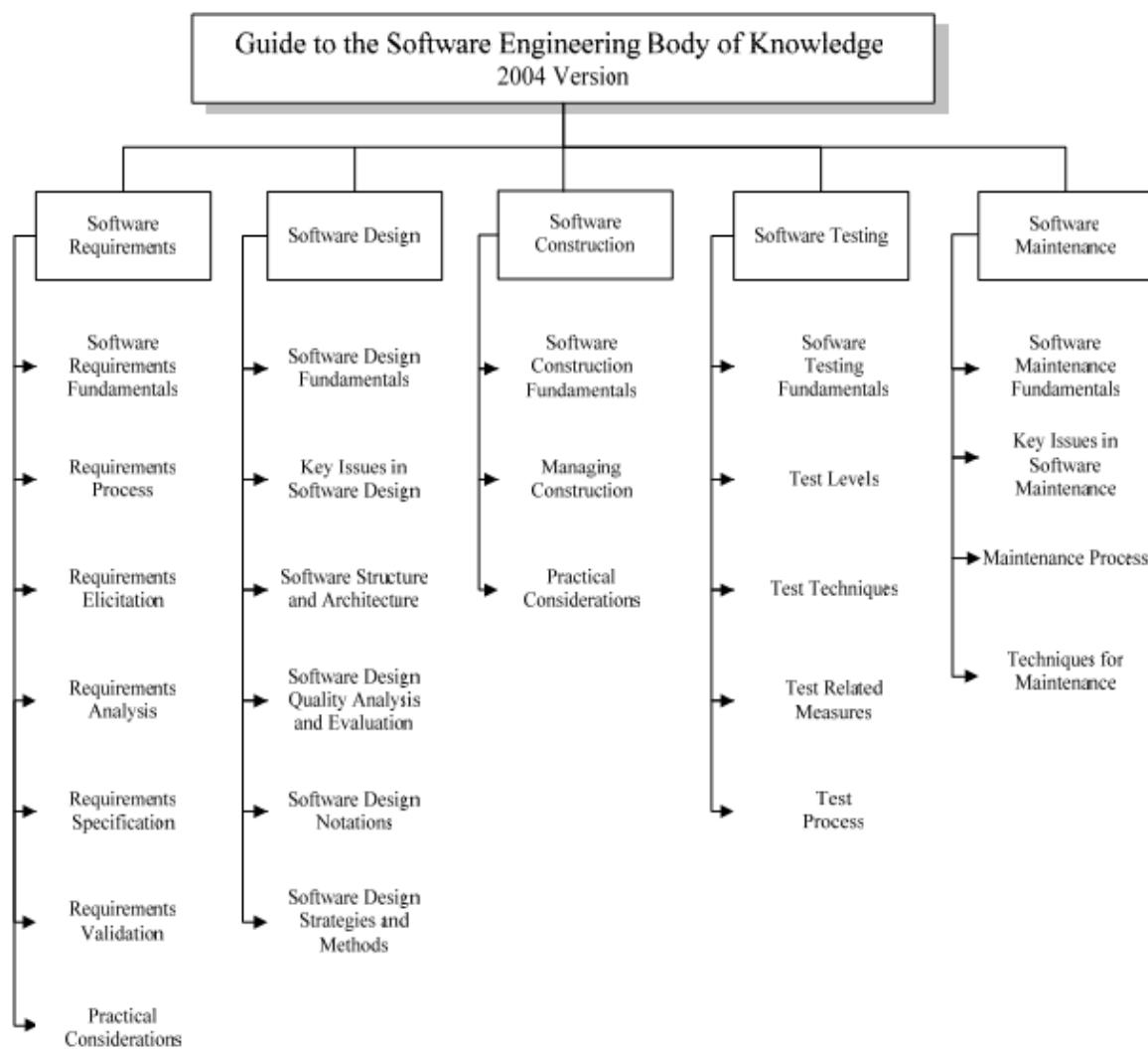
Origens do Corpo de Conhecimentos da Engenharia de Software:

- Caracterizar o conteúdo da disciplina de engenharia de software.
- Classificar em tópicos a área de conhecimento da engenharia de software
- Prover uma base para o desenvolvimento curricular, para certificação individual e para licenciamento de material.

O Swebok está dividido em partes para facilitar o entendimento e possibilitar a especialização. Cada parte é chamada de Área de Conhecimento. As dez áreas de conhecimento definidas são:

PRIMEIRO DESENHO	<ul style="list-style-type: none"> • Requisitos de Software • Design de Software • Construção de Software • Teste de Software • Manutenção de Software
-----------------------------	---

SEGUNDO DESENHO	<ul style="list-style-type: none"> • Gerência de Configuração de Software • Gerência de Engenharia de Software • Processo de Engenharia de Software • Métodos e Ferramentas de Engenharia de Software • Qualidade de Software
----------------------------	--



Requisitos De Software

Esta área está relacionada com o levantamento das propriedades que o software deve possuir e as restrições existentes. Ela explicita, analisa e valida os requisitos de maneira a permitir que no final do projeto o software tenha uma chance maior de atender as necessidades dos clientes. Esta área é de extrema importância, pois ao fazer a análise e a validação dos requisitos do software, podem ser percebidos carência de algumas propriedades que o software deve possuir e até mesmo conflito entre as necessidades existentes, possibilitando prever problemas e propor mudanças antes do software começar a ser implementado.

Design De Software

Transformação de requisitos de software, tipicamente estabelecidos em termos relevantes ao domínio do problema, em uma descrição explicando como solucionar os aspectos do problema relacionados com software.

Construção De Software

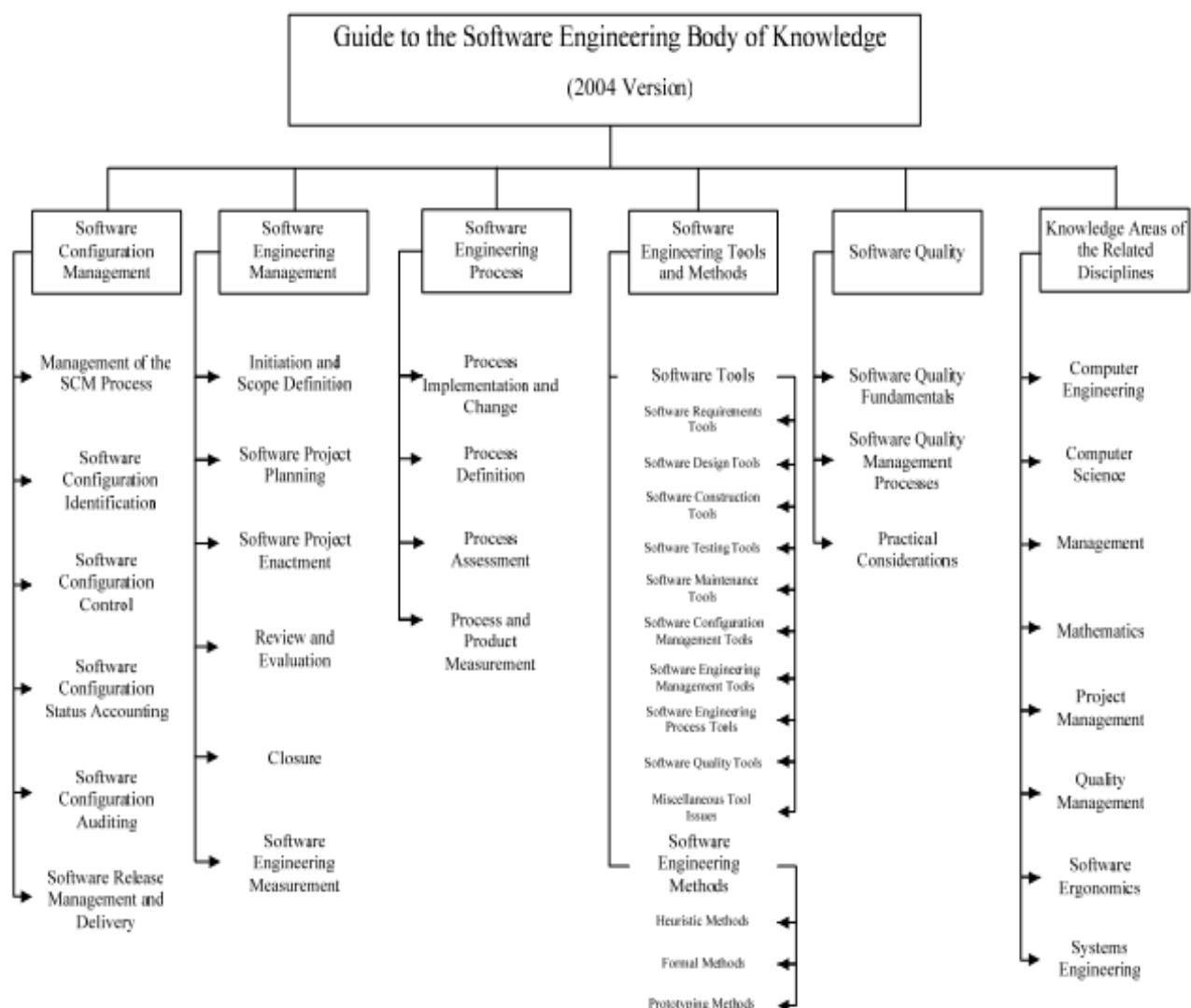
Construção de programas funcionais e coerentes através da codificação, autovalidação, e teste unitário.

Teste De Software

Verificação dinâmica do comportamento do programa através do uso de um conjunto finito de casos de teste – adequadamente selecionados de um domínio de execuções usualmente infinito - contra o comportamento esperado deste.

Manutenção De Software

Atividades de suporte custo-efetivo a um sistema de software, que pode ocorrer antes e após a entrega do software. Após a entrega do software são feitas modificações com o objetivo de corrigir falhas, melhorar seu desempenho ou adaptá-lo a um ambiente modificado. Antes da entrega do software são feitas atividades de planejamento.



Gerência De Configuração De Software

Identifica a configuração do sistema (características documentadas do hardware e software que o compõem) em pontos discretos no tempo, de modo a controlar sistematicamente suas mudanças e manter sua integridade e rastreabilidade durante o ciclo de vida do sistema.

Gerência De Engenharia De Software

Gerencia projetos de desenvolvimento de software.

Processo De Engenharia De Software

Define, implementa, mede, gerencia, modifica e aperfeiçoa o processo de desenvolvimento de software.

Métodos E Ferramentas De Engenharia De Software

Ferramentas de software que automatizam o processo de engenharia de software.

Métodos impõem estrutura sobre a atividade de desenvolvimento e manutenção de software com o objetivo de torná-la sistemática e mais propensa ao sucesso.

Qualidade De Software

A área de qualidade software está relacionada com considerações de qualidade de software que transcendem os processos de ciclo de vida do software. Ela está ligada diretamente com a qualidade do produto do software. Porém é impossível separar a qualidade final do produto da qualidade do processo onde o produto está sendo desenvolvido, pois o processo de

desenvolvimento influencia fortemente o produto final gerado e se este processo não possuir um determinado grau de qualidade o resultado final não será satisfatório.



Estudo Complementar

<http://www.swebok.org/>

http://pt.wikipedia.org/wiki/Software_Engineering_Body_of_Knowledge

<http://www.cic.unb.br/~jhcf/MyBooks/iess/Intro/10AreasDaEngenhariaDeSoftware.pdf>

<http://www.cin.ufpe.br/~dar/SWEBOK-%20cinco%20areas%20da%20eng%20software%20-%20dar.pdf>



Atividades

1. Atividade 1: O BLOG do Marco Mendes realmente é muito interessante !!
Aproveite para visitá-lo e receber mais informações de outros BOK's existentes:
 - <http://blog.marcomendes.com/2007/08/29/swebok-pmbok-babok-outros-boks/>
2. Atividade 2: Baixe o SWEBOK através do link abaixo (existe a necessidade de entrar com nome e email para baixar do site oficial):
 - <http://www.swebok.org/pdfformat.html>



UNIDADE 2

Taxonomia De Estilos Arquiteturais

Objetivo: Apresentar os estilos arquiteturais mais importantes para a Engenharia de Software.

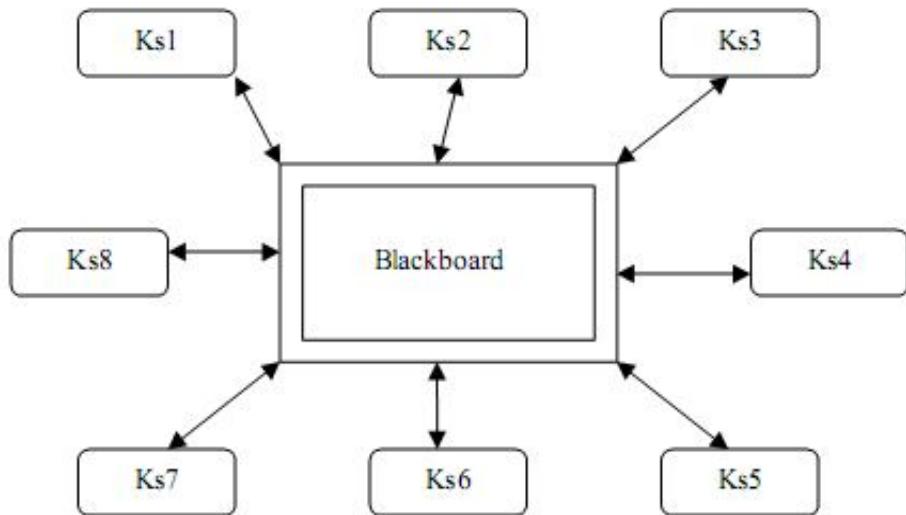
Embora tenhamos milhões de sistemas desenvolvidos ao longo do tempo podemos categorizá-los em poucos estilos arquiteturais. Conforme os estudos dos pesquisadores Swaw, M. e Garlan, D., Buschmann, F., Bass L., Clements P. e Kazman, R. chega-se basicamente em cinco estilos arquiteturais.

A taxonomia é definida como sendo a ciência ou técnica de classificação. Para o nosso caso ela estrutura os vários tipos de arquitetura existentes. Vejamos, portanto, os estilos arquiteturais mais importantes:

Arquitetura Centrada Nos Dados

Nesse tipo de arquitetura, também chamada de Repositório, temos o conceito de Depósito de Dados e que alguns autores intitulam tanto de Repositório, como Blackboard (Quadro-Negro). Conforme a figura a seguir, temos na área central a Base Central de Dados (Blackboard), e os componentes que executam operações na referida base, inclusive suas intercomunicações.

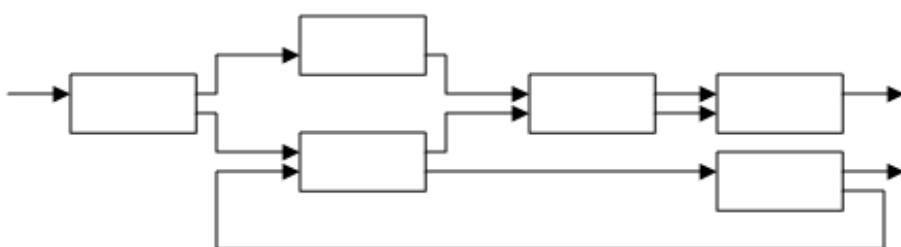
Alguns autores chamam esses componentes que acessam o repositório central de Software-Cliente, e outros de fontes de informação (Knowledge Source - Ks). Uma das características fortes dessa arquitetura é que embora os componentes estejam interligados através da base, eles não podem se comunicar um com outro, exceto via repositório central.



Arquitetura De Fluxo De Dados

Essa estrutura também chamada de tubos e filtros, pois os dados de entrada devem ser transformados, por meio de uma série de componentes computacionais, em dados de saída (Pressman, 2006).

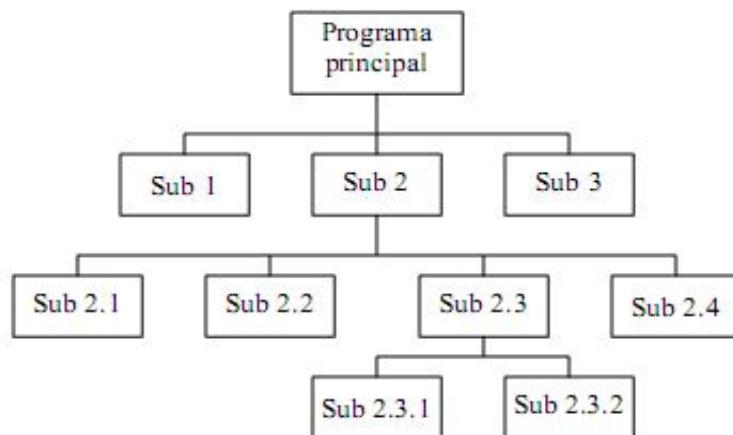
Na figura abaixo temos as linhas representando os “tubos”, e os retângulos como sendo os “filtros”. Cada filtro trabalha independentemente dos componentes afluentes e efluentes, e é projetado para esperar entrada de dados de certa forma e produzir dados de saída (para o filtro seguinte) de um modo específico.



Arquitetura De Chamada E Retorno

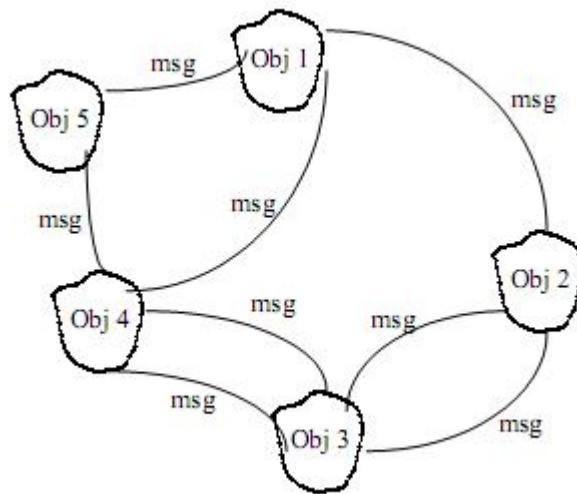
Arquitetura clássica muito utilizada na análise estruturada, e quando linguagens de programação não possuem um suporte efetivo para o encapsulamento esta é a arquitetura preferencial.

Esta arquitetura também chamada de arquitetura de programa principal/sub-rotinas caracteriza-se pela existência de um programa inicial (programa principal) que chama diversos outros programas (sub-rotinas) em uma sequência preestabelecida. Cada um dos programas chamados ao terminar, retorna seu controle ao programa chamador.



Arquitetura Orientada A Objetos

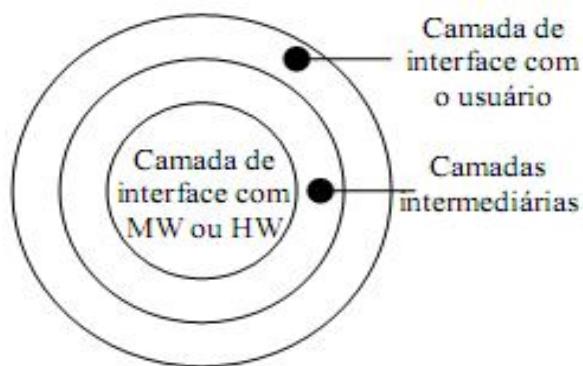
A característica principal dessa arquitetura são que os elementos de um sistema, os dados e as operações, são encapsulados em um componente chamado objeto. A comunicação e a coordenação entre esses componentes, intitulados objetos, são obtidos por meio de passagem de mensagens.



Arquitetura Em Camadas

Esta arquitetura é representada através de uma figura que faz lembrar uma “cebola”. Na camada mais interna temos operações mais próximas do conjunto de instruções de máquina fazendo interface com o Sistema Operacional. As camadas intermediárias fornecem serviços utilitários e funções do software de aplicação.

A arquitetura em camadas possui uma característica forte para reuso, e como exemplo, podemos citar o conceito de máquinas virtuais em linguagens de programação, como o Java.





Estudo Complementar

http://pt.wikipedia.org/wiki/Arquitetura_de_software

Um site dedicado a discussão das arquiteturas de sistema:

<http://www.softwarearchitectures.com>

http://www.sei.cmu.edu/architecture/published_definitions.html



Atividades

Veja a interessante Dissertação de Mestrado de Marcelo Cabral da Silva com o título “Identificação de estilos de Arquitetura” em:

<http://www-di.inf.puc-rio.br/~julio/DISSE01.pdf>

OBS.: *imagens que ilustraram esta unidade foram retiradas deste material*



UNIDADE 3

Engenharia De Proteção

Objetivo: apresentar questões que precisam ser consideradas na especificação e no projeto de software seguro.

A difusão do uso da Internet na década de 1990 introduziu um novo desafio para os engenheiros de sistemas: projetar e implementar sistemas seguros. Como mais e mais sistemas foram conectados à Internet, uma variedade de diferentes ataques externos foi inventada, e eles ameaçam esses sistemas.

Os problemas de produção de sistemas confiáveis aumentaram drasticamente. Os engenheiros de sistemas tiveram que considerar ameaças de agressores maliciosos e tecnicamente experientes, bem como problemas resultantes de erros acidentais em processos de desenvolvimento (Sommerville, 2007).

As ameaças de proteção enquadram-se basicamente em três categorias principais:

1. Ameaças à confidencialidade do sistema e aos seus dados

- Essas ameaças podem revelar informações a pessoas ou programas que não estão autorizados a ter acesso a elas.

2. Ameaças à integridade do sistema e aos seus dados

- Essas ameaças podem danificar ou corromper o software ou seus dados.

3. Ameaças à disponibilidade do sistema e aos seus dados

- Esse tipo de ameaça pode restringir acessos ao software ou a seus dados para usuários autorizados.

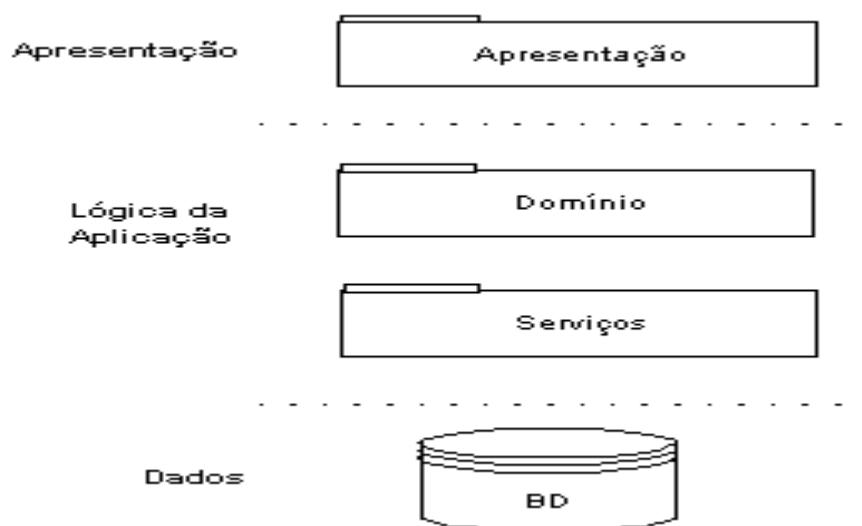
Gerenciamento de proteção não é uma tarefa simples, mas inclui uma série de atividades tais como gerenciamento de usuários e permissões, implantação e manutenção de software de sistema e monitoração, detecção e recuperação de ataques.

O gerenciamento de usuários e permissões inclui adicionar e remover usuários do sistema, assegurar que os mecanismos adequados de autenticação estejam funcionando e estabelecer as permissões no sistema de modo que os usuários tenham acesso somente aos recursos de que necessitam.

A implantação e manutenção de sistema de software incluem a instalação de sistema de software e de *middleware* e sua configuração adequada para que as vulnerabilidades de proteção sejam evitadas. Isso também envolve a atualização regular desse software com as novas versões ou *patches* que reparam os problemas de proteção descobertos.

A monitoração, detecção e recuperação de ataques incluem atividades que monitoram o sistema em relação a acessos não autorizados, detectam e colocam em funcionamento as estratégias para resistir aos ataques e atividades de *back-up* de modo que as operações normais possam se reassumidas depois de um ataque externo.

Para prover proteção em um sistema, deve-se utilizar uma arquitetura de camadas com os ativos críticos protegidos nos níveis mais baixos do sistema, e com várias camadas de proteção em torno deles.



O número de camadas de proteção que uma aplicação necessita ter depende da importância dos seus dados. A fim de acessar e modificar os registros de pacientes, um agressor precisa penetrar basicamente em três camadas de sistema:

1. Proteção No Nível De Plataforma

- O nível mais alto controla o acesso à plataforma na qual opera o sistema de registro. Isso normalmente envolve a identificação de usuário em determinado computador. A plataforma incluirá também normalmente apoio para a manutenção da integridade de arquivos do sistema.

2. Proteção No Nível De Aplicação

- O próximo nível de proteção é construído na própria aplicação. Ele envolve um usuário que acessa a aplicação, sendo autenticado e autorizado para realizar ações tais como visualizar ou modificar dados. O apoio de gerenciamento de integridade específico da aplicação pode estar disponível.

3. Proteção No Nível De Registro

- Esse nível é invocado quando o acesso a registros específicos for necessário e envolve a verificação de um usuário se está autorizado a realizar as operações solicitadas nesses registros. A proteção nesse nível poderia também envolver criptografia para assegurar que os registros não possam ser pesquisados com o uso de um navegador de arquivos. A verificação da integridade que utiliza, por exemplo, *checksums* criptográficos pode detectar mudanças feitas fora dos mecanismos normais de atualização de registros.



Estudo Complementar

<http://www.cl.cam.ac.uk/%7Erja14/book.html>

http://download.cissp.com.br/Ed.001_20.09.06.pdf

http://camargoneves.com/Apresentacoes/unicap_proespe.pdf



Atividades

Visite o interessante BLOG do profissional de Segurança da Informação Camargo Neves, destaque os principais artigos sobre o tema:

<http://camargoneves.com/>



UNIDADE 4

Diretrizes Para Um Projeto Seguro

Objetivo: Apresentar diretrizes para um projeto seguro de software.

Diferentes tipos de sistemas exigem diferentes medidas técnicas para atingir o nível de proteção aceitável para o usuário do sistema. Por exemplo, um sistema bancário exige muito mais em termos de segurança do que uma universidade, em que existe a necessidade dos sistemas terem uma segurança relativamente flexível.

No entanto, existem diretrizes gerais que têm aplicabilidade ampla quando se projetam soluções de proteção de sistema que encapsulam boas práticas de projeto para sistemas seguros. As dez diretrizes de projeto que Sommerville (2007) orienta têm como objetivo tanto de conscientizar os pontos críticos de segurança, como também ser um *checklist* de revisão que pode ser usada no processo de validação de sistema. Vamos a elas:

Diretriz 1: Basear as decisões de proteção em uma política de proteção explícita

Uma política de proteção deve definir o “que” da proteção, no lugar de “como”. A política não deve definir mecanismos usados para prover e fazer cumprir a proteção. Os projetistas devem, portanto, consultar a política de proteção quando ela provê um *framework* para tomar e avaliar decisões de projeto.

Diretriz 2: Evitar um ponto único de falha

Num sistema crítico é uma boa prática evitar um ponto único de falha. Pois, uma falha única, numa parte do sistema, pode resultar na falha geral de sistemas. Em termos de proteção, isso significa que você não deve contar com um único mecanismo para assegurar a

proteção, mas deve empregar várias técnicas distintas. Pode-se chamar isso de “defesa em profundidade”.

Para proteger a integridade de dados o sistema, você poderia manter um *log* de todas as mudanças feitas nos dados, de modo que, se ocorrer uma falha, você poderá examinar o *log* para recriar o conjunto de dados.

Diretriz 3: Falhar de maneira protegida

Nenhuma falha de sistema deve possibilitar a um agressor acesso aos dados que não seria normalmente permitido. Portanto, embora as falhas de sistema sejam inevitáveis, os sistemas críticos de segurança devem sempre falhar de maneira segura (*fail-safe*) e sistemas críticos de proteção devem sempre falhar de maneira protegida (*fail-secure*).

Um uso dessa filosofia seria, por exemplo, a técnica de numa aplicação cliente/servidor deixar um pequeno arquivo no computador cliente para facilitar o acesso aos dados. Uma vez utilizado do serviço, esse arquivo é eliminado. No entanto, em caso de falha esse arquivo poderá ficar inseguro. Uma estratégia seria de manter essa mesma técnica, mas com o arquivo criptografado, impossibilitando a leitura desses dados por pessoas não autorizadas.

Diretriz 4: Equilibrar proteção e facilidade de uso

À medida que um sistema aumenta suas características de proteção, é inevitável que ele se torne menos fácil de usar. Um bom exemplo seria os atuais sistemas bancários na Internet. Exigem tantas chaves para acessarmos os dados, que muitas vezes provocam irritação, e perda de tempo.

Portanto, chega-se a um ponto em que é contraproducente manter a adição de novas características de proteção à custa da facilidade de uso. Por exemplo, se você exigir que os usuários insiram várias senhas ou alterem suas senhas com uma constância muito pequena,

eles simplesmente escreverão suas senhas em algum lugar. Um agressor poderá, então, encontrar as senhas e obter acesso ao sistema.

Diretriz 5: Estar ciente da possibilidade de Engenharia Social

Em Segurança da informação, chamam-se Engenharia Social as práticas utilizadas para obter acesso às informações importantes ou sigilosas em organizações ou sistemas por meio da enganação ou exploração da confiança das pessoas. Para isso, o golpista pode se passar por outra pessoa, assumir outra personalidade, fingir que é um profissional de determinada área, etc. É uma forma de entrar em organizações que não necessita da força bruta ou de erros em máquinas. Explora as falhas de segurança das próprias pessoas que, quando não forem devidamente treinadas para esses ataques, podem ser facilmente manipuladas.

Do ponto de vista de projeto, enfrentar a Engenharia Social é muito difícil. Mecanismos de registro que acompanham a localização e a identificação dos usuários e os programas de análise de registros podem também ser úteis, na medida em que permitem detectar brechas de proteção.

Diretriz 6: Usar redundância e diversidade para reduzir riscos

Redundância significa que você mantém mais de uma versão de software ou de dados no sistema. Diversidade, quando aplicada ao software, significa que versões diferentes não devem ser baseadas na mesma plataforma ou usar as mesmas tecnologias. Portanto, uma vulnerabilidade de plataforma ou de tecnologia não afetará todas as versões e, dessa maneira, conduzirá a falhas de modo comum.

Uma aplicação dessa diretriz seria utilizar sistemas operacionais diferentes no cliente e no servidor (por exemplo: Linux no servidor e Windows no cliente). De modo a assegurar que um ataque baseado na vulnerabilidade de algum desses sistemas operacionais não afete o servidor e o cliente simultaneamente.

Diretriz 7: Validar todas as entradas

Um ataque comum ao sistema envolve o fornecimento de entradas inesperadas ao sistema que causam um comportamento imprevisto. Pode-se evitar muito desses problemas ao se projetar uma validação de entradas no sistema. Essencialmente, nunca se deve aceitar nenhuma entrada sem aplicar alguma verificação sobre ela.

Por exemplo, ninguém possui um sobrenome com mais de 70 caracteres e nenhum endereço é maior do que 100 caracteres. Se usar menus para apresentar as entradas permitidas, podem-se evitar alguns dos problemas de validação de entradas.

Diretriz 8: Compartilhar seus ativos

Deve-se organizar a informação no sistema de modo que os usuários somente tenham acesso à informação de que necessitam, no lugar de toda a informação do sistema. Pode-se também precisar de mecanismos no sistema para conceder acessos inesperados. Nessas circunstâncias, pode-se usar um mecanismo de proteção alternativo para ignorar a compartimentalização do sistema.

Diretriz 9: Projetar para implantação

Muitos dos problemas de proteção surgem porque o sistema não está configurado corretamente quando é implantado no seu ambiente operacional. Deve-se, portanto, projetar sempre os sistemas de modo que os recursos sejam incluídos para simplificar a implantação e para verificar erros potenciais de configuração e omissões do sistema implantado.

Diretriz 10: Projetar para capacidade de recuperação

Independentemente de quanto esforço você dispense na manutenção de proteção do sistema, você deve sempre projetar o seu sistema com a hipótese de que uma falha de

proteção possa ocorrer. Portanto, você deve pensar em como se recuperar de possíveis falhas e restaurar o sistema para um estado operacional seguro.

Por exemplo, suponhamos um ataque externo aonde uma pessoa não autorizada tenha obtido uma combinação válida de *login* e senha. Você precisa, portanto, projetar seu sistema para negar acesso a qualquer pessoa até que todos os usuários tenham alterado suas senhas e autenticar usuários reais. Uma maneira de fazer isso seria usar um mecanismo do tipo desafio/resposta, no qual os usuários precisam responder a questões para as quais possuam respostas pré-registradas. Isso seria invocado somente quando as senhas fossem alteradas.



Estudo Complementar

http://pt.wikipedia.org/wiki/Seguran%C3%A7a_da_informa%C3%A7%C3%A3o

<http://www.sobresites.com/segurancadainformacao/>



Atividades

Realize uma pesquisa na Internet sobre Engenharia Social e veja filmes comerciais que já trataram desse tema.



Dica

Pesquise o excelente filme “Prenda-me se for capaz”.



UNIDADE 5

Projeto De Interface Com O Usuário

Objetivo: Apresentar Aspectos Do Projeto De Interface Com O Usuário Que São Importantes Para O Engenheiro De Sistemas.

O projeto de sistema de computador abrange um espectro de atividades desde o projeto de hardware até o projeto de interface com o usuário. Enquanto os especialistas são, com frequência, contratados para o projeto de hardware e para projetos gráficos de páginas Web, somente grandes organizações normalmente contratam projetistas especializados em interface para seu software de aplicação. Portanto, os engenheiros de software devem, muitas, vezes, assumir a responsabilidade pelo projeto de interface com o usuário, bem como pelo projeto do software para implementar essa interface.

Existem basicamente três tipos de projeto de interface tais como:

1. Projeto de Interfaces entre componentes do software.
2. Projeto de Interfaces entre o software e outros produtores e consumidores de informação não humanos (outras entidades externas).
3. Projeto de Interface entre um ser humano (o usuário) e o computador.

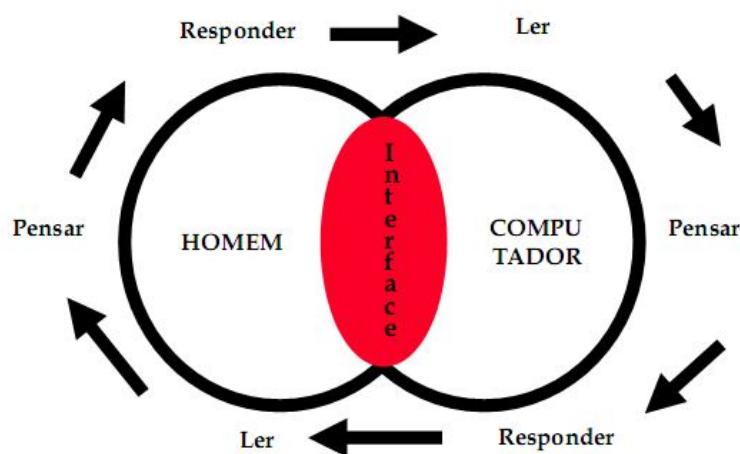
Iremos aqui nos concentrar no último tipo de Projeto de Interface onde é envolvido o ser humano, comumente chamado de interface homem-computador (IHC). Existem várias siglas nessa área e uma delas é também a IHM representando interface homem-máquina.

Atualmente as interfaces mais comuns são as seguintes:

- Interface gráfica do usuário – GUI (Graphical User Interface): aceita a entrada através de sistemas como o teclado ou mouse e fornece saída gráfica articulada no monitor.

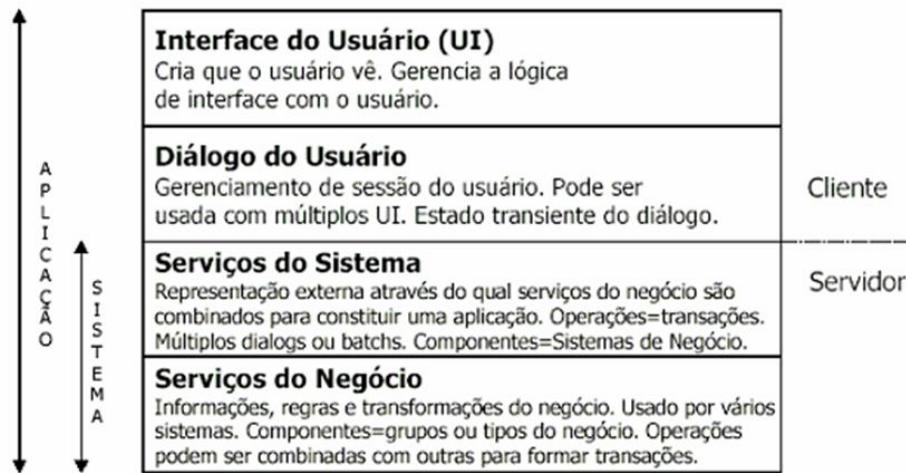
- Interface web do usuário - aceita a entrada e fornece saída ao gerar páginas web, que são transportadas pela Internet e visualizadas através de um navegador.
- Interface de linha de comando - aceita a entrada através de comandos de texto utilizando o teclado e fornece como saída caracteres no monitor.
- Interface tátil - interface gráfica do usuário que usa telas sensíveis ao toque (touch screen) como forma de entrada, tornando o monitor um dispositivo tanto de entrada como de saída do sistema.

Conforme vemos na figura a seguir, a interface com o usuário (IU) intermedeia o homem com a máquina. Existem processamentos de ambos os lados (pensar), e interpretações das mensagens encaminhadas para cada um deles (ler), assim como respostas a tudo isso (responder). Quanto mais simples for todo esse processo maior o sucesso da interface.



REGRAS DE OURO NO PROJETO DE INTERFACE (Theo Mandel, 1997)	Coloque o usuário No controle	Defina os modos de interação
		Proporcione interação flexível
		Permita a interrupção e o desfazer
		Simplifique a interação e permita a personalização
		Esconda detalhes técnicos
		Faça a interação direta com os objetos na tela
	Reduza a carga de memória do usuário	Reduza a necessidade de memória de curto prazo
		Estabeleça <i>defaults</i> significativos
		Defina atalhos intuitivos
		Interface análoga ao mundo real
	Faça a interface consistente	Revele informações progressivamente
		Situe o usuário dentro do contexto maior
		Mantenha consistência
		Se modelos anteriores foram significativos p/ os usuários não modificar

Vejamos na figura a seguir que podemos projetar numa arquitetura Cliente-Servidor (lado direito), divisões estratégicas para o nosso usuário. Numa aplicação como um todo, podemos dividir a com a parte mais sistêmica do lado servidor, e as interações com o usuário do lado cliente (lado esquerdo).



Quando o desenvolvimento interativo é usado, o projeto de interface com o usuário prossegue incrementando-se, à medida que o software é desenvolvido. Algumas vezes, o protótipo da interface é desenvolvido separadamente, em paralelo com outras atividades de engenharia de software. Em ambos os casos, contudo, antes de iniciar a programação, você deve ter desenvolvido e, de preferência, testado alguns projetos baseados em papel. Existem três atividades principais nesse processo:

Análise de usuário:

No processo de análise de usuário, você desenvolve uma compreensão das tarefas que os usuários realizam, seus ambientes de trabalho, os outros sistemas que eles usam, como eles interagem com outras pessoas em seu trabalho, etc. Para produtos com vários usuários, você deve tentar desenvolver essa compreensão por meio do foco em grupos, ensaios com usuários potenciais e exercícios similares.

Prototipação de Sistema:

O projeto e desenvolvimento da interface com o usuário é um processo interativo. Embora os usuários possam informar sobre os recursos de interface de que necessitam, é muito difícil para eles serem específicos até que vejam algo tangível. Portanto, você deve desenvolver sistemas de protótipo e expô-los aos usuários, que poderão, então, guiar a evolução da interface.

Avaliação da Interface:

Embora haja, obviamente, discussões com os usuários durante o processo de prototipação, você deve também ter uma atividade de avaliação mais formalizada, na qual sejam coletadas informações sobre a experiência do usuário com a interface.

O cronograma de projeto de UI dentro do processo de software depende, em alguma extensão, de outras atividades. A prototipação pode ser usada como parte do processo de engenharia de requisitos e, nesse caso, faz sentido iniciar o processo de projeto de UI por esse estágio. Nos processos interativos, o projeto de UI é integrado ao desenvolvimento de software. Como o software propriamente dito, a UI deve passar por refactoring e reprojeto durante o desenvolvimento.



Estudo Complementar

http://pt.wikipedia.org/wiki/Interface_do_utilizador

<http://www.pucrs.campus2.br/~jiani/trabalhos/levacov.htm>

http://www.dcce.ibilce.unesp.br/~ines/cursos/eng_soft/aula11.PDF

http://www.dimap.ufrn.br/~jair/piu/JAI_Apostila.pdf





Atividades

Verifique em sua empresa, ou na de colegas, seus principais sistemas e faça uma análise crítica sobre as suas interfaces de usuário.



UNIDADE 6

Gerenciamento De Pessoal

Objetivo: Explicar a importância das pessoas no processo de Engenharia de Sistemas.

A experiência mostra que, à medida que o número de pessoas em uma equipe de projeto de software aumenta, a produtividade global do grupo pode sofrer. Um modo de contornar esse problema é criar várias equipes de engenharia de software, compartimentalizando consequentemente as pessoas em grupos de trabalhos individuais.

No entanto, enquanto o número de equipes de engenharia de software cresce, a comunicação entre elas se torna tão difícil e demorada quanto a comunicação entre indivíduos. Pior, a comunicação (entre indivíduos ou equipes) tende a ser ineficiente – isto é, gasta-se muito tempo transferindo muito pouco conteúdo de informação e, muito frequentemente, informação importante “se perde nas frestas” (Pressman, 2006).

Se a comunidade de engenharia de sistemas tiver que lidar efetivamente com o dilema de comunicação, a estrada adiante, para os engenheiros de software é incluir modificações radicais no modo pelo qual indivíduos e equipes comunicam-se uns com os outros.

E-mail, sites Web e vídeo-conferência centralizada são agora comuns como mecanismos para conectar um grande número de pessoas a uma rede de informação. A importância dessas ferramentas no contexto do trabalho de engenharia de software não pode ser por demais enfatizada. Com um sistema efetivo de correio eletrônico ou sistema de mensagem instantânea, o problema encontrado por um engenheiro em Nova York pode ser resolvido com ajuda de um colega em Tóquio.

Em um sentido muito real, sessões de conversa (chat sessions) bem focadas e grupos de interesse especializados tornam-se repositórios de conhecimento que permitem que a

sabedoria coletiva de um grande número de tecnólogos seja concentrada em um problema técnico ou em um assunto gerencial.

A boa comunicação entre os membros de um grupo de desenvolvimento de software é essencial. Os membros do grupo devem trocar informações sobre o status do seu trabalho, as decisões de projeto tomadas e as mudanças necessárias em decisões anteriores. A boa comunicação também fortalece a coesão do grupo uma vez que os membros passam a compreender as motivações, os pontos fortes e fracos de outras pessoas no grupo. Os fatores principais que influenciam a eficiência da comunicação são:

Tamanho do grupo:

À medida que um grupo cresce em tamanho, torna-se mais difícil assegurar que todos os membros se comuniquem eficientemente uns com os outros. O número de elos de comunicação em uma direção é $n(n-1)$, sendo n o tamanho do grupo; portanto, por exemplo, num grupo de sete a oito membros, é muito provável que algumas pessoas raramente se comuniquem.

Estrutura do grupo:

As pessoas em grupos estruturados informalmente se comunicam com mais eficiência do que as pessoas em grupos com uma estrutura formal e hierárquica. As pessoas de nível podem não conversar umas com as outras. Esse é um problema específico em um projeto de grande porte com vários grupos de desenvolvimento. Quando as pessoas que trabalham em subsistemas diferentes se comunicam somente por meio de seus gerentes, o projeto pode sofrer atrasos e haver desentendimentos.

Composição do grupo:

Pessoas com o mesmo tipo de personalidade podem entrar em conflito e a comunicação pode ficar comprometida. A comunicação geralmente é melhor em grupos compostos de homens e mulheres do que em grupos com pessoas de mesmo sexo, conforme estudos de Marshall e Heslin (1975). As mulheres tendem a ser mais orientadas a interações do que os homens, e podem atuar como controladoras e facilitadoras das interações do grupo.

Ambiente físico de trabalho:

A organização do local de trabalho é um fator importante para facilitar ou inibir as comunicações. Um estudo realizado por McCue (1978) mostrava que a arquitetura plana e aberta fornecida por muitas organizações não era popular nem produtiva. Os fatores mais importantes identificados nesse estudo foram:

Privacidade:

Programadores rendem melhor em lugares mais fechados.

Consciência Externa:

Pessoas preferem luz natural e com visão do exterior.

Personalização:

Os indivíduos adotam diferentes práticas de trabalho e têm opiniões diferentes sobre a decoração. A possibilidade de reorganizar o espaço para se adequar às práticas de trabalho e poder personalizar esse ambiente é importante.

Um dos papéis do gerente de projeto é motivar as pessoas que trabalham com ele. Motivação significa organização do trabalho e ambiente de trabalho de forma que as pessoas se sintam estimuladas a trabalhar tão eficientemente quanto possível. Se as pessoas não estão motivadas, elas não terão interesse no trabalho que estão realizando. Elas trabalharão lentamente, serão mais propensas a cometer erros e não contribuirão para as metas mais amplas da equipe ou da organização.

Das necessidades registradas por Maslow (1954) na sua famosa pirâmide das necessidades humanas, as que mais afetam as pessoas que trabalham em organizações de desenvolvimento de software são as necessidades de satisfação social, autoestima e auto-realização.



Estudo Complementar

http://pt.wikipedia.org/wiki/Gest%C3%A3o_de_recursos_humanos



Atividades

Faça uma pesquisa sobre o impacto da Gestão de Recursos Humanos na equipe de tecnologia



UNIDADE 7

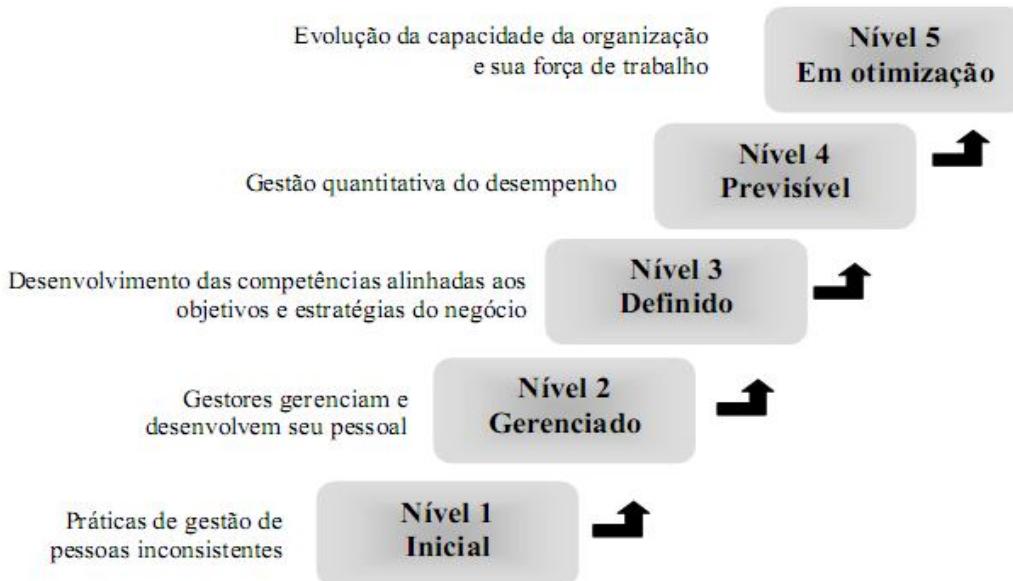
Modelos De Gerenciamento De Pessoal

Objetivo: Apresentar modelos de gerenciamento de pessoal relacionados com a tecnologia de sistemas.

Modelo De Maturidade De Capacitação De Pessoal (P-Cmm)

O Software Engineering Institute (SEI) nos Estados Unidos está engajado em um programa de longo prazo de aprimoramento do processo de software. Parte desse programa, como já vimos, é o Modelo de Maturidade de Capacitação CMM (Capability Maturity Model). Ele está relacionado com as melhores práticas em Engenharia de Software.

Para apoiar esse modelo, o instituto também propôs o Modelo de Maturidade de Capacitação de Pessoal P-CMM (People Capability Maturity Model). O P-CMM pode ser usado como *framework* de aprimoramento da maneira pela qual uma organização gerencia seu patrimônio humano. Assim como o CMM, o P-CMM é um modelo de cinco níveis, conforme ilustrado na figura a seguir.



O P-CMM reforça a necessidade de reconhecer a importância das pessoas como indivíduos e de desenvolver suas capacidades. É uma ferramenta prática para aprimoramento do gerenciamento de pessoal em uma organização, pois fornece um *framework* para motivação, reconhecimento, padronização e aprimoramento de boas práticas. Entretanto, como todos os modelos de capacitação criados pelo SEI, ele é projetado para grandes empresas, não levando em conta as pequenas.

Naturalmente, a aplicação completa desse modelo é muito dispendiosa e, provavelmente, desnecessária para a maioria das organizações. Contudo, ele é um guia útil que pode conduzir a melhorias significativas na capacidade da organização em produzir software de qualidade.

PSP - Personal Software Process

PSP é um processo de software focado para engenheiros de software individuais. O PSP foi desenvolvido por Watts Humphrey e está descrito no seu livro "A Discipline for Software Engineering" (Uma disciplina para Engenheiros de Software) de 1995. O PSP foi desenvolvido para guiar como planejar e desenvolver módulos de software ou pequenos programas, mas pode ser adaptado para outras tarefas pessoais.

O PSP, assim como o CMM, é baseado no princípio da melhoria do processo. Enquanto o CMM é focado da melhoria da capacidade organizacional, o foco do PSP é no profissional de software.

Os objetivos principais do PSP são:

- Melhorar as estimativas;
- Melhorar o planejamento e o acompanhamento de cronogramas;
- Proteger contra o excesso de compromissos;
- Criar um comprometimento pessoal para a qualidade;
- Envolvimento contínuo do engenheiro na melhoria continua do processo;

- Além disso, favorece a melhoria da capacidade organizacional como um todo ao melhorar o trabalho individual.

Em 1997 Will Hayes e James W. Over fizeram um estudo com 298 engenheiros de software e o resultado deste estudo está disponível no relatório técnico "An Empirical Study of Impact of PSP on Individual Engineers" disponível no site da SEI. No estudo eles verificaram o impacto da implantação do PSP. Como resultados obtidos podemos citar:

- Uma melhoria de 75% (em média) na estimativa de esforço;
- Uma melhoria de 150% (em média) na estimativa de tamanho;
- A tendência de subestimar o tamanho e o esforço foram reduzidos. E o número de sobrestimativas e de subestimativas foi balanceado;
- *Qualidade do Produto*: os defeitos encontrados em uma unidade de produto testada melhoraram em uma relação de 2.5 vezes (em média);
- *Qualidade do Processo*: os defeitos encontrados antes da compilação aumentaram em 50% (em média);
- *Produtividade Pessoal*: o número de linhas de código por hora não alterou substancialmente. Mas a melhoria da qualidade do produto resulta na melhora da produtividade do ciclo de desenvolvimento. Testes de produto e testes de integração são executados mais rapidamente por causa da melhoria da qualidade do produto, diminuindo assim o ciclo de desenvolvimento.

O PSP é uma ferramenta que pode ser aplicada em qualquer estrutura organizacional. Pode ser introduzida em qualquer modelo de desenvolvimento. Melhora a qualidade de desenvolvimento pessoal tornando-se uma ferramenta indispensável em pequenos projetos e serve de grande auxílio em projetos maiores. Aumenta a qualidade do produto, pois tem-se um esforço de detecção de erros antes mesmo da fase de testes (simplificando e barateando esta fase).



Estudo Complementar

<http://blog.marcomendes.com/category/gestao-de-pessoas/>

<http://www.sei.cmu.edu/tsp/psp.html>

http://pt.wikipedia.org/wiki/Personal_software_process

www.ufpel.tche.br/prg/sisbi/bibct/acervo/info/2000/Mono-JoseWilson.pdf

www.simpros.com.br/simpros2005/upload/A04_2_artigo14181.pdf

<http://www.dsc.ufcg.edu.br/~patricia/esii2001.1/materialaulas/aula02.pdf>



Atividades

Veja a aplicabilidade do PSP testando o software de apoio a implantação dessa técnica em:

<http://processdash.sourceforge.net/>



UNIDADE 8

Gerenciamento De Qualidade

Objetivo: Explicar o gerenciamento de qualidade, como o SWEBOk trata esse tema e os padrões existentes.

A qualidade de software tem se aprimorado significativamente nos últimos 15 anos. Uma razão para isso é o fato de as empresas terem adotado novas técnicas e tecnologia, como o uso de desenvolvimento orientado a objetos e de ferramenta de apoio CASE associada. Além disso, contudo, tem havido uma conscientização maior da importância do gerenciamento de qualidade de software e da adoção de técnicas de gerenciamento de qualidade provenientes da manufatura de software.

Entretanto, qualidade de software é um conceito complexo que não é diretamente comparável com a qualidade na manufatura. Na manufatura, a noção de qualidade tem sido aquela em que o produto desenvolvido deve atender às suas especificações conforme Crosby (1979). Em um mundo ideal essa definição deveria ser aplicada a todos os produtos, mas, para sistemas de software, há problemas com isso, pois:

- A especificação deve se orientada para as características do produto que o cliente deseja. Entretanto, a organização que desenvolve pode também ter requisitos (como requisitos de facilidade de manutenção) que não estão incluídos na especificação;
- Não sabemos como especificar certas características de qualidade (por exemplo, facilidade de manutenção) de maneira não ambígua;
- Conforme a engenharia de requisitos, é muito difícil escrever especificações de software completas. Portanto, embora um produto de software possa estar de acordo com a sua especificação, os usuários podem mesmo assim não considerá-lo como um

produto de alta qualidade, pelo fato do software entregue não corresponder às suas expectativas.

Para um melhor entendimento e estudo, o SWEBOK divide a Qualidade de Software em três tópicos, cada tópico é subdividido em atividades, da seguinte forma:

1. Fundamentos de Qualidade de Software

- Cultura e Ética de Engenharia de Software
- Valores e Custos de Qualidade
- Modelos e Características de Qualidade
- Melhoria da Qualidade

2. Gerência do Processo de Qualidade de Software

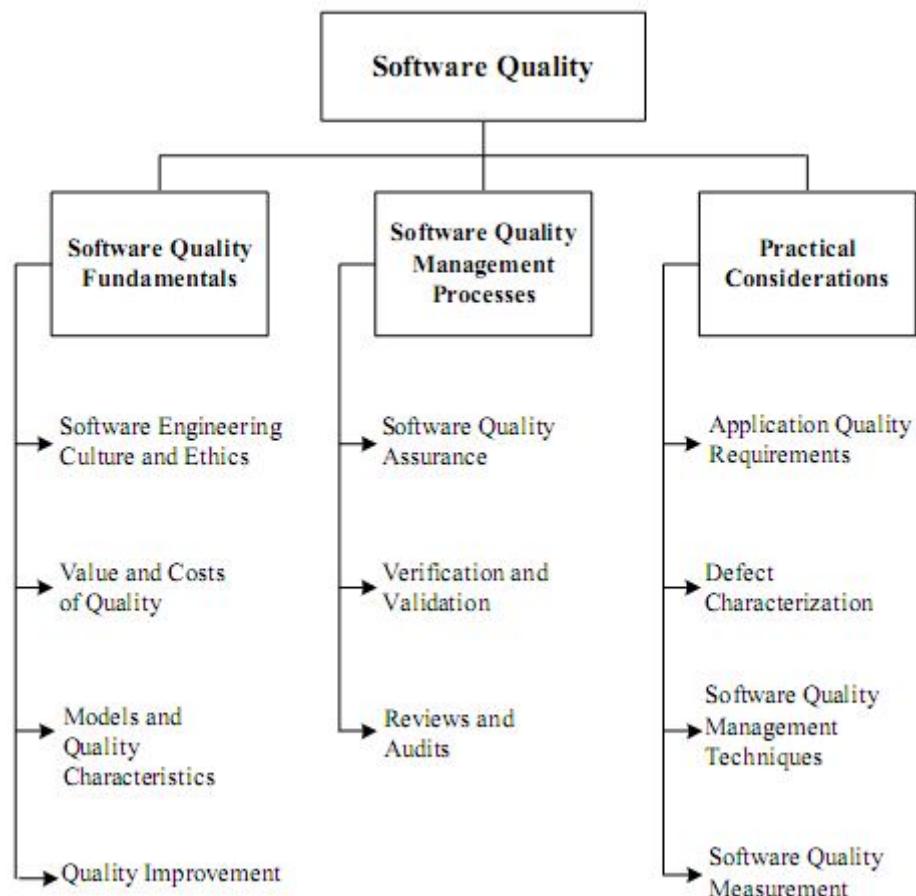
- Garantia de Qualidade de Software
- Verificação e Validação
- Revisões e Auditorias

3. Considerações Práticas

- Requisitos de Qualidade para Aplicações
- Caracterização de Defeitos
- Técnicas de Gerência de Qualidade de Software
- Medidas de Qualidade de Software

Ainda segundo o SWEBOK, a Qualidade de Software é um tema tão importante que é encontrado, de forma ubíqua, em todas as outras áreas de conhecimento envolvidas em um projeto. Além disso, ele deixa claro que essa área, como nele definido, trata dos aspectos estáticos, ou seja, daqueles que não exigem a execução do software para avaliá-lo, em

contraposição á área de conhecimento Teste de software. Porém, é normal que se encontrem autores e empresas que afirmam serem os Testes de Software uma etapa da Qualidade de Software.



Para sistemas menores, o gerenciamento da qualidade é ainda importante, mas uma abordagem mais informal pode ser adotada. Nem tanta papelada é necessária porque uma equipe pequena de desenvolvimento pode se comunicar informalmente. A questão principal de qualidade para desenvolvimento de sistemas pequenos é estabelecer uma cultura de qualidade e assegurar que todos os membros da equipe tenham uma abordagem positiva para a qualidade de software.

O gerenciamento de qualidade de software para sistemas de grande porte pode ser estruturado em três atividades principais:

Garantia de Qualidade:

Estabelecimento de um *framework* de procedimentos organizacionais e padrões que conduzem a um software de alta qualidade.

Planejamento de Qualidade:

Seleção de procedimentos e padrões apropriados deste *framework*, adaptados para um projeto de software específico.

Controle de Qualidade:

Definição e aprovação de processos que assegurem que a equipe de desenvolvimento de software tenha seguido os procedimentos e os padrões de qualidade do projeto.

O gerenciamento de qualidade de processo envolve:

- Definição de padrões de processo, como “como” e “quando” as revisões devem ser conduzidas.
- Monitoração do processo de desenvolvimento para assegurar que os padrões estão sendo seguidos.
- Relato do processo de software para a gerência de projeto e para o comprador do software.

Os dois tipos de padrões que podem ser estabelecidos como parte do processo de garantia de qualidade são:

PADRÕES de PRODUTO:

Esses padrões se aplicam especificamente ao produto de software em desenvolvimento. Eles incluem padrões de documentos, como a estrutura de documentos de requisitos, padrões de documentação, como um cabeçalho de comentário padronizado para uma definição de classe de objeto; e padrões de codificação, que definem como uma linguagem de programação deve ser utilizada.

PADRÕES de PROCESSO:

Esses padrões definem os processos que devem ser seguidos durante o desenvolvimento de software. Podem incluir definições de processos de especificação, projeto e de validação, e uma descrição dos documentos que devem ser escritos ao longo destes processos.



Estudo Complementar

http://pt.wikipedia.org/wiki/Qualidade_de_Software



Atividades

Um colega, que é programador muito bom, produz software com baixo número de defeitos, mas ele simplesmente ignora os padrões de qualidade da organização. Se você fosse o gerente dessa equipe como reagiria em relação a esse comportamento?



UNIDADE 9

Padrão De Qualidade - Iso 9001

Objetivo: Mostrar o histórico da ISO 9000 e o impacto na área de Tecnologia.

Um conjunto internacional de padrões que pode ser usado no desenvolvimento de um sistema de gerenciamento de qualidade em todas as indústrias é chamado de ISO 9000. Os padrões ISO 9000 podem ser aplicados para uma variedade de organizações, desde a indústria de manufatura até a indústria de serviço.

O padrão ISO 9001 é o mais geral desses padrões e se aplica às organizações que se dedicam a processos de qualidade nas empresas que projetam, desenvolvem e mantêm produtos. Um documento de apoio (ISO 9000-3) interpreta a ISO 9001 para o desenvolvimento de software. Vários livros descrevem o padrão ISO 9001.

O padrão ISO 9001 não é especificamente voltado para o desenvolvimento de software, mas estabelece princípios gerais que podem ser aplicados ao software. O padrão ISO 9001 descreve vários aspectos do processo de qualidade e exibe os padrões e os procedimentos organizacionais que a empresa deve definir.

Eles devem ser documentados em um manual de qualidade da organização. A definição de processo deve incluir descrições da documentação necessária para demonstrar que os processos definidos foram seguidos durante o desenvolvimento do produto.



O padrão ISO 9001 não define os processos de qualidade que devem ser usados. De fato, ele não restringe de modo nenhum os processos usados em uma organização. Isso permite flexibilidade por meio de setores industriais e significa que pequenas empresas podem ter processos não burocráticos e, ainda assim, estar em conformidade com a norma. Contudo, essa flexibilidade significa que você não pode fazer quaisquer suposições sobre a similaridade ou a diferença entre os processos nas empresas em conformidade a ISO 9001.

Os procedimentos de garantia de qualidade de uma organização são documentados em um manual que define o processo de qualidade. Em alguns países, as autoridades certificadoras asseguram que o processo de qualidade, conforme expresso no manual de qualidade, está em conformidade ao padrão ISO 9001. Cada vez mais, os clientes procuram pela certificação ISO 9000 de um fornecedor como um indicador de com que seriedade esse fornecedor trata a qualidade.

Algumas pessoas pensam que a certificação ISO 9000 significa que a qualidade do software produzido pelas empresas certificadas será melhor do que a das empresas sem certificação. Esse certamente não é o caso. O ISO 9000 está simplesmente relacionado com a definição de processos que serão usados em uma empresa e a documentação associada, como

processos de controle que podem explicitamente mostrar que esses processos foram seguidos. Isso não está relacionado com a garantia de que os processos refletem as melhores práticas ou com a qualidade de produto.

Portanto, digamos que uma empresa possa definir procedimentos de teste de produto que conduzam aos testes de softwares incompletos. Assim, à medida que esses processos forem seguidos e documentados, a empresa estaria seguindo o padrão ISO 9001. Enquanto essa situação for indesejável, não há dúvida de que alguns padrões de empresas serão muito inadequados e trarão pouca contribuição para a real qualidade de software.

Padrões de documentação:

Em um projeto de software são importantes porque os documentos são o único modo tangível de representação do software e do processo desse. Existem três tipos de padrões de documentação:

Padrões de Processo de Documentação:

Esses padrões definem o processo que deve ser seguido para a produção de documentos.

Padrões de Documentos:

Esses padrões regem a estrutura e a apresentação física dos documentos.

Padrões de Intercâmbio de Documentos:

Esses padrões asseguram que todas as cópias eletrônicas de documentos sejam compatíveis.



Estudo Complementar

http://pt.wikipedia.org/wiki/ISO_9000

<http://www.inmetro.gov.br/qualidade/docOrientativo.asp>



Atividades

Leia o importante documento sobre a aplicabilidade do padrão de qualidade ISO 9001 em:

www.bsibrasil.com.br/documentos/What_is_9kBR.pdf



UNIDADE 10

Desenvolvimento De Sistemas Críticos

Objetivo: apresentar técnicas de implementação usadas no desenvolvimento de sistemas críticos.

Técnicas de Engenharia de Sistemas avançadas, linguagens de programação aprimoradas e melhor gerenciamento de qualidade conduziram a aprimoramentos significativos na confiabilidade da maioria dos softwares. Entretanto, sistemas críticos, como os que controlam máquinas não assistidas, sistemas médicos, comutadores de telecomunicações ou aeronaves necessitam de níveis mais altos de confiabilidade.

Nesses casos, técnicas especiais de desenvolvimento podem ser usadas para assegurar que o sistema seja seguro, protegido e confiável. Existem basicamente três abordagens complementares para desenvolver um software confiável:

Prevenção de Defeitos:

O processo de projeto e de implementação do sistema deve usar abordagens de desenvolvimento de software que ajudem a evitar erros de programação e, assim, minimizar o número de defeitos de um programa.

Detecção de Defeitos:

Os processos de verificação e validação são projetados para descobrir e remover defeitos de um programa antes que este seja implantado para uso operacional.

Tolerância a defeitos:

O sistema é projetado de forma que os defeitos ou o comportamento inesperado do sistema, durante a execução, sejam detectados e gerenciados de modo que a falha do sistema não ocorra.

Fundamentais para a confiabilidade de qualquer sistema são as noções básicas de redundância e diversidade. Estas são estratégias cotidianas de proteção para evitarem falhas.

- Se você está investindo na bolsa, não deve alocar todos os seus investimentos numa única empresa, pois poderá perder tudo se a empresa vier a falir (diversidade). As pessoas guardam pilhas e lâmpadas reservas em seus lares para que possam se recuperar rapidamente de falhas (redundância).
- Todos nós devemos fazer *back-up* de nossos computadores regularmente em casos de falha no disco (redundância) e, para proteger nossos lares em termos de segurança, geralmente temos mais de um tipo de fechadura na porta principal de entrada (diversidade).
- Sistemas críticos podem incluir componentes que replicam a funcionalidade de outros componentes (redundância) ou código adicional de verificação que não é estritamente necessário para que os sistemas funcionem (redundância).

Portanto, os defeitos podem ser detectados antes que causem falhas, e o sistema poderá ser capaz de dar continuidade operando caso os componentes individuais falhem. Se os componentes redundantes não forem os mesmos que outros componentes (diversidade), uma falha em comum no mesmo componente replicado não resultará na falha completa do sistema.

Em sistemas em que a disponibilidade é um requisito crítico, servidores redundantes são normalmente disponibilizados. Eles entram em operação automaticamente caso um servidor designado falhe.

Algumas vezes, para assegurar que ataques ao sistema não possam explorar uma vulnerabilidade comum, os servidores podem ser de tipos diferentes e executar diferentes Sistemas Operacionais. O uso de Sistemas Operacionais diferentes é um exemplo concreto de diversidade e de redundância de software.

Infelizmente, incluir diversidade e redundância nos sistemas torna-os mais complexos e, dessa maneira, mais difíceis de compreender. Portanto, é mais provável que os programadores venham a cometer erros e menos provável que pessoas, ao verificar os programas, encontrem erros.

Consequentemente, algumas pessoas consideram melhor evitar a redundância e a diversidade em um software para que o desenvolvimento do sistema seja tão simples e seguro quanto possível, e ter procedimentos de verificação e validação extremamente rigorosos.

Ambas as abordagens são utilizadas em sistemas de segurança comerciais críticos. O sistema de controle de vôo do Airbus 340 é redundante e tem diversidade, enquanto o sistema de controle de vôo do Boeing 777 baseia-se em uma única versão de software.

Um dos objetivos da pesquisa em Engenharia de Sistemas tem sido desenvolver ferramentas, técnicas e métodos que conduzam a uma produção de software livre de defeitos. Um software livre de defeitos é o que atende exatamente a sua especificação.

Entretanto, isso não significa que o software nunca irá falhar. Podem ocorrer erros na especificação que serão refletidos no software ou os usuários podem não compreender ou usar inapropriadamente o sistema de software. No entanto, a eliminação de defeitos do software tem certamente impacto enorme no número de falhas do sistema.



Estudo Complementar

<http://www.dstan.mod.uk/>

<http://www.poli.usp.br/pro/procsoft/tpcsepusp03.pdf>

<http://www.lbd.dcc.ufmg.br:8080/colecoes/sbes/2001/002.pdf>



Atividades

Realize uma pesquisa na Internet sobre Sistemas Críticos que não tiveram sucesso.

Existem vários casos que ficaram famosos na história ...



Atividades

Antes de dar continuidades aos seus estudos é fundamental que você acesse sua SALA DE AULA e faça a Atividade 1 no “link” ATIVIDADES.



UNIDADE 11

Desenvolvimento De Sistemas Críticos (Continuação)

Objetivo: Dar continuidade aos aspectos do desenvolvimento de Sistemas Críticos.

Para sistemas de pequeno e médio porte, as técnicas de Engenharia de Software provavelmente tornam possível desenvolver software livre de defeitos. Para atingir esse objetivo, você precisa usar uma gama de técnicas de Engenharia de Software:

Processos de Software Confiáveis:

O uso de um processo de software confiável com atividades de validação e verificação apropriadas é essencial caso o número de defeitos em um programa deva ser minimizado e aqueles ignorados devam ser detectados.

Gerenciamento de Qualidade:

A organização que desenvolve o sistema deve ter uma cultura na qual a qualidade conduz o processo de software. A cultura organizacional deve estimular os programadores a criar programas livres de defeitos. Padrões de desenvolvimento e projeto devem ser estabelecidos e procedimentos devem estar disponíveis para verificar se os padrões foram seguidos.

Especificação Formal:

Deve haver uma especificação precisa (de preferência formal) que defina o sistema a ser implantado. Muitos defeitos de projeto e programação resultam de uma interpretação errada de uma especificação ambígua ou mal especificada.

Verificação Estática:

Técnicas de verificação estática, como o uso de analisadores estáticos, podem encontrar características anômalas de programação que podem ser defeitos. A verificação formal, baseada na especificação do sistema, também pode ser usada.

Tipagem Forte:

Uma linguagem de programação com tipos fortes de dados, como Java ou ADA, deve ser usada para o desenvolvimento. Se a linguagem tiver tipagem forte, o compilador poderá detectar muitos defeitos de programação antes que sejam incluídos no programa a ser entregue.

Programação Segura:

Algumas construções da linguagem de programação são mais complexas e propensas a erros do que outras, e será mais provável cometer erros caso sejam usadas. A programação segura significa evitar, ou ao menos minimizar, o uso dessas construções.

Informações Protegidas:

Deve ser usada uma abordagem para projeto e implementação de software baseada em ocultamento e encapsulamento de informações. Linguagens orientadas a objetos, como Java, obviamente satisfazem essa condição. Deve ser encorajado o desenvolvimento de programas projetados para facilidade de leitura e compreensão.

As empresas de desenvolvimento de software admitem que seu software sempre conterá alguns defeitos residuais. O nível de defeitos depende do tipo de sistema. Produtos comerciais têm nível relativamente alto de defeitos, embora estejam muito melhores que há dez anos, enquanto sistemas críticos têm, normalmente, uma densidade de defeitos menor.

A lógica para a aceitação de defeitos é que, se e quando o sistema falhar, é mais vantajoso pagar pelas consequências da falha do que descobrir e remover os defeitos antes da entrega final do sistema. No entanto, a decisão de liberar o software com defeitos não é só

econômica. A aceitação política e social, ou seja, a imagem da empresa no mercado, da falha do sistema também deve ser considerada.



Atividades

Como você enxerga o conteúdo apresentado nesta unidade com o restante da nossa apostila? Existem itens relacionados?!?



UNIDADE 12

Arquiteturas Tolerantes A Defeitos

Objetivo: Visualisar as arquiteturas clássicas de hardware e as de software tolerantes a falhas.

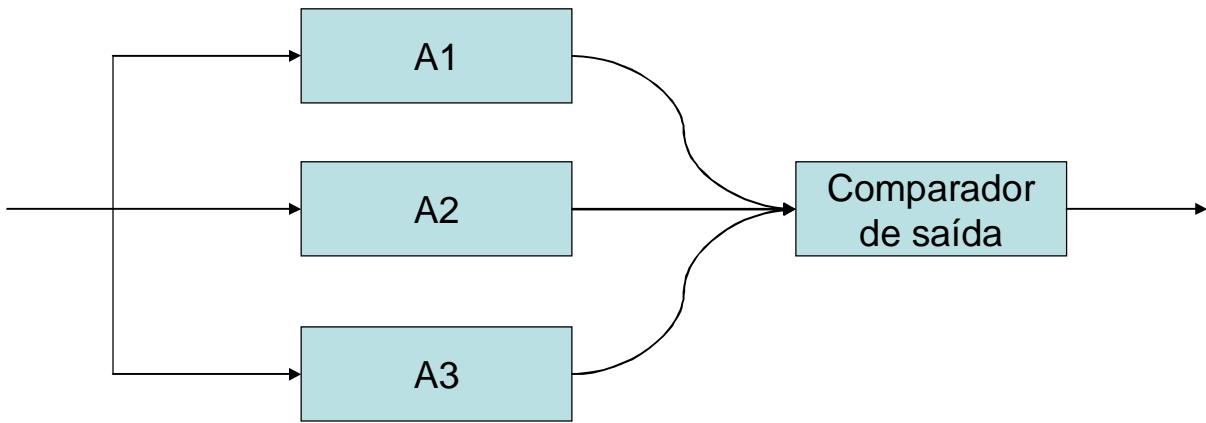
Em muitos sistemas, é possível implementar a tolerância aos defeitos de software por meio da inclusão explícita de ações de verificação e recuperação no software. Isso é denominado programação defensiva. Contudo, essa abordagem não trabalha eficientemente com defeitos do sistema que surgem com base em interações entre o hardware e o software. Além disso, a má compreensão dos requisitos pode significar que tanto o código do sistema quanto a defesa associada estão incorretos.

Para a maioria dos sistemas críticos, particularmente aqueles com requisitos rigorosos de disponibilidade, pode ser necessária uma arquitetura específica projetada para apoiar a tolerância a defeitos.

Exemplos de sistemas que usam essa abordagem de tolerância a defeitos são os sistemas de aviação, que devem estar em operação durante o vôo, os sistemas de telecomunicações, e os sistemas críticos de comando e controle.

Há muitos anos existe a necessidade de se construir hardware tolerante a defeitos. A técnica de hardware tolerante a defeitos mais comum é baseada na noção de redundância modular tripla – TMR (Triple-Modular Redundancy).

A unidade de hardware é replicada três (ou algumas mais) vezes. A saída de cada unidade passa para um comparador de saída normalmente implementado como um sistema de votação. Se uma das unidades falha e não produz a mesma saída que as outras unidades, sua saída é ignorada. Veja a figura a seguir:



Um gerenciador de defeitos pode tentar reparar a unidade defeituosa automaticamente, mas se isso for impossível, o sistema será automaticamente reconfigurado para tirar a unidade de serviço. Em seguida, o sistema continuará a funcionar com duas unidades operando.

Através dessa técnica é que foi possível a NASA colocar o homem na lua. Embora com computadores precários e extremamente simples, foi somente com esse tipo de arquitetura tolerante a falhas que foi possível garantir o pleno sucesso dessa missão.

Essa abordagem de tolerância a defeitos baseia-se no fato de que a maioria das falhas de hardware é resultante das falhas de componente em vez de ser ocasionada por defeitos de projeto. Os componentes são, portanto, propensos a falhar independentemente.

Supõe-se que, quando completamente operacionais, todas as unidades de hardware operem de acordo com a especificação. Há, portanto, baixa probabilidade de falha simultânea de componentes em todas as unidades de hardware.

Naturalmente, todos os componentes poderiam ter um defeito de projeto em comum e, assim, todos produziriam a mesma resposta errada. Usando unidades de hardware com uma especificação em comum, mas projetadas e construídas por fabricantes diferentes, reduzem-se as chances de uma falha de modo comum.

Presume-se que a probabilidade de equipes diferentes cometerem o mesmo erro de projeto ou de fabricação seja bem pequena.

Se os requisitos de disponibilidade e confiabilidade de um sistema forem tais que você necessite usar um hardware tolerante a falhas, você pode também precisar de um software tolerante a defeitos. Existem duas abordagens para fornecer software tolerante a defeitos.

Ambas as técnicas foram derivadas do modelo de hardware em que componentes, ou sistemas, redundantes são incluídos e componentes defeituosos podem ser desativados. As duas abordagens para tolerância a defeitos de software são:

Programação Em N-Versões:

Usando uma especificação comum, o sistema de software é implementado em uma série de versões por diferentes equipes. Essas versões são executadas paralelamente em computadores separados. Suas saídas são comparadas com a utilização de um sistema de votação, e saídas inconsistentes ou saídas não produzidas em tempo, são rejeitadas. No mínimo três versões de sistema devem ser disponibilizadas de modo que duas versões sejam consideradas consistentes no evento de uma única falha. Essa é a abordagem mais comum usada para tolerância a defeitos de software. Ela foi usada em sistemas de sinalização de ferrovias, em sistemas de aviação e em sistemas de proteção de reatores.

Blocos De Recuperação:

Nessa abordagem, cada componente do programa inclui um teste para verificar se o componente foi executado com sucesso. Também inclui um código alternativo que permite ao sistema fazer uma cópia e repetir o processamento se o teste detectar uma falha. As implementações são, deliberadamente, interpretações diferentes da mesma especificação. Elas são mais executadas em sequência do que em paralelo. Dessa maneira, o hardware replicado não é necessário. Na programação em n-versões, as implementações podem ser diferentes, mas não é incomum que duas ou mais equipes escolham o mesmo algoritmo para implementar a especificação.



Estudo Complementar

www.ic.unicamp.br/~beatriz/projetos/bpm/CIAWI07_A%20Fault%20Tolerant%20Architecture%20for.pdf



Atividades

Forneça motivos pelos quais as versões de sistema na abordagem de programação em n-versões possam falhar de maneira similar.



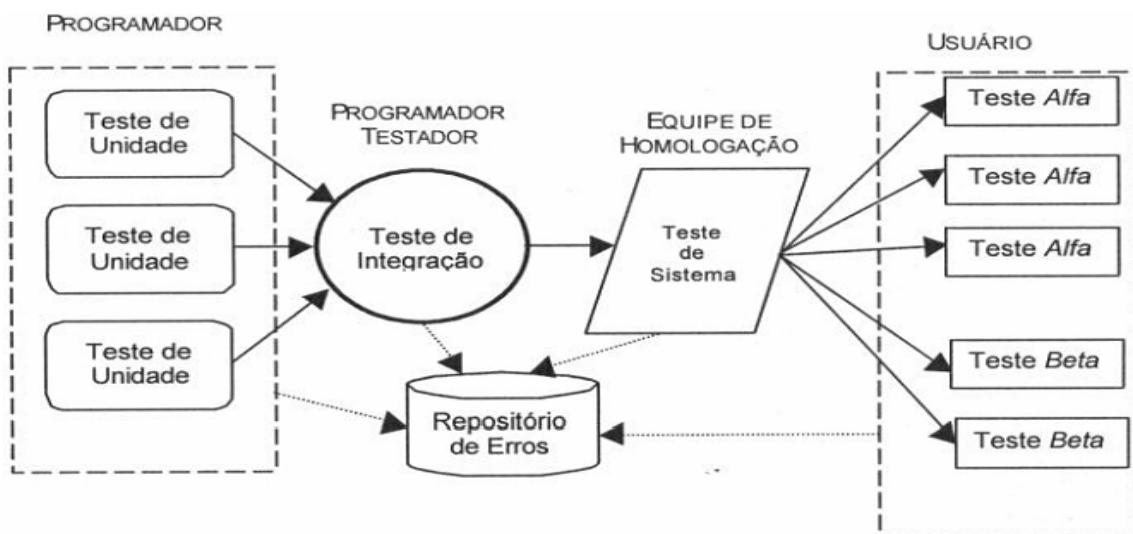
UNIDADE 13

Estratégias De Teste De Software

Objetivo: Abordar a problemática da realização de testes com a equipe de desenvolvimento e como organizá-la para essa atividade.

Uma estratégia de testes de software deve ser suficientemente flexível para promover uma abordagem de teste sob medida. Ao mesmo tempo, deve ser suficientemente rígida para promover planejamento razoável e acompanhamento gerencial, à medida que o projeto progide. Shooman discute esses pontos da seguinte forma:

"Sob vários aspectos, o teste é um processo independente e o número de tipos diferentes de teste varia tanto quanto as diferentes abordagens de desenvolvimento. Durante muitos anos, nossa única defesa contra erros de programação era o projeto cuidadoso e a inteligência própria do programador. Estamos agora em uma era em que técnicas modernas de projeto estão nos ajudando a reduzir o número de erros iniciais que são inerentes ao código. Analogamente, diferentes métodos de teste estão começando a se agregar em várias abordagens e filosofias distintas".



Em cada projeto de software há um conflito de interesses que ocorre quando o teste começa. O pessoal que construiu o software é agora solicitado a testá-lo. Isso parece inócuo em si mesmo, afinal de contas, quem conhece o programa melhor de que seus desenvolvedores?

Infelizmente, esses mesmos desenvolvedores têm interesses ocultos em demonstrar que o programa está livre de erros, que funciona de acordo com os requisitos do cliente e que será completado de acordo com o cronograma e dentro do orçamento. Cada um desses interesses se contrapõe a um teste rigoroso.

Conforme a figura anterior o desenvolvedor de software (programador) é sempre responsável por testar as unidades individuais do programa (teste de unidade), tanto testes funcionais quanto estruturais, garantindo que cada uma realiza a função ou exibe o comportamento para o qual foi projetada.

Em muitos casos, o desenvolvedor também conduz os testes de integração – uma etapa de teste que leva à construção da arquitetura completa do software. Um programador designado realiza o teste de integração de cada módulo com os demais módulos do sistema (esta pessoa preferencialmente não deve ter participado do desenvolvimento).

Apenas depois da arquitetura do software ser completada, um grupo independente de teste (equipe de homologação) começa a ser envolvido. O papel do grupo independente de teste (Independent Test Group – ITG) é remover os problemas inerentes associados de deixar o construtor testar o software que ele construiu. O teste independente remove o conflito de interesses que pode, de outra forma, estar presente. Afinal de contas, o pessoal de ITG é pago para encontrar erros.

As características da equipe ITG devem ser especiais e a escolha de seus profissionais com critério. É interessante que sejam pessoas que não se envolveram no processo de desenvolvimento, e profissionais que conheçam o domínio do negócio. Exemplos: Analistas de negócio, profissionais mais antigos (no caso de softwares de uso interno), e profissionais que atuam no *help-desk* da empresa.

O ITG é parte da equipe do projeto de desenvolvimento de software no sentido de que esteve envolvido durante a análise e o projeto e continua envolvido ao longo de um projeto de grande porte. No entanto, em muitos casos, o ITG pertence à equipe de garantia da qualidade de software, alcançando assim um grau de independência que poderia não ser possível se ele fizesse parte da organização de Engenharia de Software.

Durante a condução do Teste de Sistema, o desenvolvedor deve estar disponível para corrigir os erros eventualmente descobertos. Em seguida, deverá ser realizado o Teste de Aceitação aonde usuários reais do sistema são selecionados e convidados a realizarem testes alfa e beta do sistema. Estes usuários devem ter:

- Boa capacidade crítica
- Certa cumplicidade com a empresa desenvolvedora
- Já ser usuário de outros sistemas desenvolvidos pela mesma empresa (preferencialmente)



Estudo Complementar

<http://dinf.unicruz.edu.br/~plentz/Teste.html>

www.pucrs.br/inf/eventos2003/workshop/arquivo/MiniCurso_Teste.pdf



Atividades

Se você fosse o Gerente de Sistemas de uma empresa como organizaria a sua equipe de desenvolvimento para a tarefa de realização de testes de software?



UNIDADE 14

Espiral Dos Testes De Software

Objetivo: Apresentar através da espiral dos teste de software os principais testes e a sua relação com as etapas de desenvolvimento.



Uma estratégia para teste de software pode também ser vista no contexto da espiral da figura anterior. O **teste de unidade** começa no centro da espiral e se concentra em cada unidade (por exemplo, componente) do software como implementado em código-fonte.

O teste progride movendo-se para fora ao longo da espiral para o **teste de integração**, em que o foco fica no projeto e na construção da arquitetura do software. Dando outra volta pela espiral para fora, encontramos o **teste de validação**, em que os requisitos estabelecidos

como parte da análise dos requisitos do software são validados em contraste com o software que acabou de ser construído.

Finalmente, chegamos ao **teste de sistema**, em que o software e os outros elementos do sistema são testados como um todo. Para testar o software de computador, nos movemos ao longo da espiral para fora aonde se amplia o escopo do teste a cada volta.

A estratégia global pata teste de software orientado a objetos é idêntica em filosofia à que é aplicada a arquiteturas convencionais, mas difere na abordagem. Começamos com o “teste no varejo” e trabalhamos para fora em direção ao “teste por atacado”. No entanto, nosso foco quando “testando no varejo” desloca-se de um módulo individual (na visão convencional) para uma classe que abrange atributos e operações e implica comunicação e colaboração.

À medida que as classes são integradas em uma arquitetura orientada a objetos, uma série de testes de regressão é feita para descobrir erros devidos a comunicação e colaboração entre classes (componentes) e efeitos colaterais causados pela adição de novas classes. Finalmente, o sistema como um todo é testado para garantir que erros nos requisitos sejam descobertos.



Estudo Complementar

<http://dinf.unicruz.edu.br/~plentz/Teste.html>

www.pucrs.br/inf/eventos2003/workshop/arquivo/MiniCurso_Testes.pdf





Atividades

Tente refazer a espiral dos testes de software sem olhar o conteúdo desta unidade.

Você consegue?



UNIDADE 15

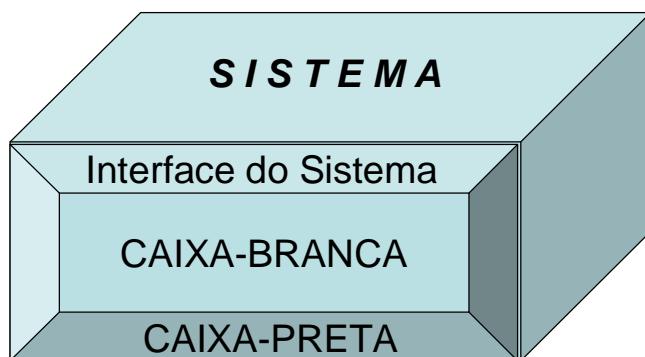
Testes Caixa-Preta E Caixa-Branca

Objetivo: Apresentar técnicas de realização de testes de software comumente utilizados no mercado.

Qualquer produto que passe por engenharia pode ser testado por uma das duas maneiras:

- Conhecendo-se a função especificada que o produto foi projetado para realizar, podem ser realizados testes que demonstrem que cada função está plenamente operacional e, ao mesmo tempo, procurem erros em cada função.
- Sabendo-se como é o trabalho interno de um produto, podem ser realizados testes para garantir que “todas as engrenagens combinem”, isto é, que as operações internas sejam realizadas de acordo com as especificações e que todos os componentes internos foram adequadamente exercitados.

A primeira abordagem de teste é chamada de teste de caixa-preta e a segunda, de teste caixa-branca.



- Teste caixa-preta, ou Teste Funcional, refere-se aos testes que são conduzidos na interface do software. Um teste caixa-preta examina algum aspecto fundamental do sistema, pouco se preocupando com a estrutura lógica interna do software.
- Teste caixa-branca de software, também chamado de Teste Estrutural, é baseado em um exame rigoroso do detalhe procedural. Caminhos lógicos internos ao software e colaborações entre componentes são testados, definindo-se caso de testes que exercitam conjuntos específicos de condições e/ou ciclos.

Assim como nos aviões, a caixa-preta é um teste importante a ser realizado com a parte externa do sistema, ou seja, com a parte que aparece para o usuário: a sua interface. Pelo contrário, a caixa-branca é um teste realizado mais profundamente na parte interna do sistema, aonde um usuário leigo não terá acesso.

À primeira vista pode parecer que um teste caixa-branca bastante rigoroso levaria a programas 100% corretos. Tudo que precisaríamos seria definir todos os caminhos lógicos, desenvolver casos de teste para exercitá-los e avaliar os resultados, isto é, gerar casos de teste para exercitar a lógica do programa de forma exaustiva.

Infelizmente, um teste completo apresenta certos problemas logísticos. Um simples programa com 100 linhas poderá facilmente exigir, para a realização de testes exaustivos, anos de processamento para verificar todas as possibilidades existentes.

Um teste caixa-branca não deve, no entanto, ser descartado como não prático. Um número limitado de caminhos lógicos importantes pode ser selecionado e exercitado. Estruturas de dados importantes podem ser submetidas à prova quanto à sua validade.

Uma regra básica defendida por Boris Beizer é que “os defeitos juntam-se nos cantos e se congregam nas fronteiras”, ou seja, testar os limites do sistema e dentro dos intervalos operacionais. Um exemplo bem prático desta técnica de teste é o uso da ferramenta livre JUnit para desenvolvimento de classes de teste (test cases) para testar classes ou métodos desenvolvidos em Java.

A técnica de teste de Caixa-Branca é recomendada para as fases de Teste da Unidade e Teste da Integração, cuja responsabilidade principal fica ao cargo dos desenvolvedores do software, que por sua vez conhecem bem o código-fonte produzido.

A técnica de teste de Caixa-Preta é aplicável a todas as fases de teste - fase de teste de unidade (ou teste unitário), fase de teste de integração, fase de teste de sistema e fase de teste de aceitação.

Alguns autores chegam a definir uma técnica de Teste Caixa Cinza, que seria um mesclado do uso das técnicas de Caixa Preta e Caixa Branca. Mas, como toda execução de trabalho relacionado à atividade de teste utilizar simultaneamente mais de uma técnica de teste, é recomendável para que se fixem os conceitos primários de cada técnica.

Porém, na prática, o jargão “caixa preta” ou “caixa branca” está sendo substituído por basicamente 3 tipos de técnicas para realizar testes de software, mais próximas da realidade das equipes envolvidas. São elas:

Técnicas Baseadas em Especificação:

Modelos formais ou informais são utilizados para especificação de um problema a ser resolvido, o software ou seu componente. Os casos de testes podem ser derivados sistematicamente destes modelos.

Técnicas baseadas em estrutura:

Informações sobre como o software é construído são utilizadas para derivar os casos de testes. Por exemplo, código e modelagem. A extensão da cobertura de código pode ser medida pelos casos de testes. Além disso, os casos de testes podem ser derivados sistematicamente para aumentar a cobertura.

Técnicas baseadas em experiência:

Conhecimento e experiência de pessoas são utilizados para derivar os casos de testes. Conhecimento sobre defeitos prováveis e sua distribuição. Conhecimento de testadores,

desenvolvedores, usuários, outros interessados (*stakeholders*) responsáveis pelo software, seu uso e ambiente.

A escolha de qual técnica utilizar dependerá de uma série de fatores, incluindo o tipo de sistema, padrões, clientes, requisitos contratuais, nível do risco, tipos de riscos, objetivos do teste, documentação disponível, conhecimento dos testadores, tempo, dinheiro, ciclo de desenvolvimento, modelo de caso de uso e uma experiência prévia do tipo de defeitos encontrados.

O importante é que estas técnicas sejam mescladas ao longo do Ciclo de Desenvolvimento do Sistema, estejam alinhadas a um Processo de Teste bem definido e sejam utilizadas de forma pontual e contínua para garantir a qualidade final do seu sistema.



Estudo Complementar

http://pt.wikipedia.org/wiki/Teste_de_software

http://www.timaster.com.br/revista/artigos/main_artigo.asp?codigo=1362



Atividades

Resuma de forma bem didática a diferença entre os testes de caixa-preta e os de caixa-branca.



UNIDADE 16

Métodos De Teste Orientado A Objetos

Objetivo: Diferenciar os testes de software orientado a objetos em relação ao sistemas convencionais.

A arquitetura de software orientado a objetos resulta em uma série de subsistemas em camadas que encapsulam classes de colaboração. Cada um desses elementos do sistema (subsistemas e classes) executa funções que ajudam a satisfazer aos requisitos do sistema.

É necessário testar um Sistema OO (Orientado a Objetos) em uma variedade de níveis diferentes para descobrir erros que podem ocorrer à medida que classes colaboram umas com as outras e subsistemas se comunicam entre as camadas arquiteturais.

Teste orientado a objetos é estrategicamente similar ao teste de sistemas convencionais, mas é taticamente diferente. Como modelos de análise e projeto OO são similares na estrutura e no conteúdo para o programa OO resultante, o “teste” pode começar com a revisão desses modelos.

Uma vez gerado o código, o teste OO real começa “no varejo” com uma série de testes projetados para exercitar operações de classes e examinar se existem erros quando uma classe colabora com outras classes.

À medida que as classes são integradas para formar um subsistema, o teste baseado no uso, em conjunto com abordagens tolerantes a falhas, é aplicado para exercitar completamente as classes que colaboram entre si. Finalmente, Casos de Uso são usados para descobrir erros no nível de validação do software.

Os métodos de teste caixa-branca, descritos na unidade anterior, podem ser aplicados a operações definidas para uma classe. Testes mais complexos tais como os de Caminho-

base, ou Teste de Ciclos de Fluxos de Dados podem ajudar a garantir que cada declaração em uma operação tenha sido testada.

No entanto, a estrutura concisa de muitas operações de classe faz alguns argumentarem que o esforço aplicado ao teste caixa-branca poderia ser mais bem redirecionado para testes no nível de classe.

Os métodos de teste caixa-preta são tão adequados para Sistemas quanto são para sistemas desenvolvidos usando os métodos convencionais da Engenharia de Software. Casos de Uso podem fornecer entrada útil no projeto de testes caixa-preta e baseados em estado.

Teste Baseado Em Erro

O objetivo do teste baseado em erros em um sistema OO é projetar testes que tenham uma probabilidade de descobrir erros plausíveis. Como o produto ou sistema deve satisfazer aos requisitos do cliente, o planejamento preliminar necessário para realizar o teste baseado em erros começa com o modelo de análise.

O testador procura erros plausíveis, isto é, aspectos da implementação do sistema que podem resultar em defeitos, para determinar se esses erros existem, casos de teste são projetados para exercitar o projeto ou o código.

Sem dúvida, a efetividade dessas técnicas depende de como os testadores consideram um erro plausível. Se erros reais em um sistema OO são considerados não plausíveis, então essa abordagem não é realmente melhor do que qualquer técnica de teste aleatório.

No entanto, se os modelos de análise e projeto puderem fornecer conhecimento aprofundado sobre o que é provável dar errado, então, o teste baseado em erros pode encontrar um número significativo de erros com dispêndio de esforço relativamente baixo.

O teste de integração procura erros plausíveis na chamada de operações ou nas conexões de mensagens. Três tipos de erros são encontrados nesse contexto:

1. Resultado inesperado
2. Uso da operação/mensagem errada
3. Invocação incorreta

Para determinar os erros plausíveis, quando as funções (operações) são invocadas, o comportamento da operação deve ser examinado.

O teste de integração se aplica tanto a atributos quanto a operações. Os “comportamentos” de um objeto são definidos pelos valores atribuídos aos seus atributos. O teste deve exercitar os atributos para determinar se ocorrem valores adequados para os tipos distintos de comportamento do objeto.

É importante notar que o teste de integração tenta encontrar erros no objeto cliente, não no servidor. Dito em termos convencionais, o foco do teste de integração está em determinar se existem erros no código que chama, não no código chamado. A chamada da operação é usada como um indício, um modo de encontrar requisitos de teste que exercitem o código que chama.



Atividades

Quais são as principais diferenças que devemos tomar ao realizar testes de software baseados em orientação de objetos?



UNIDADE 17

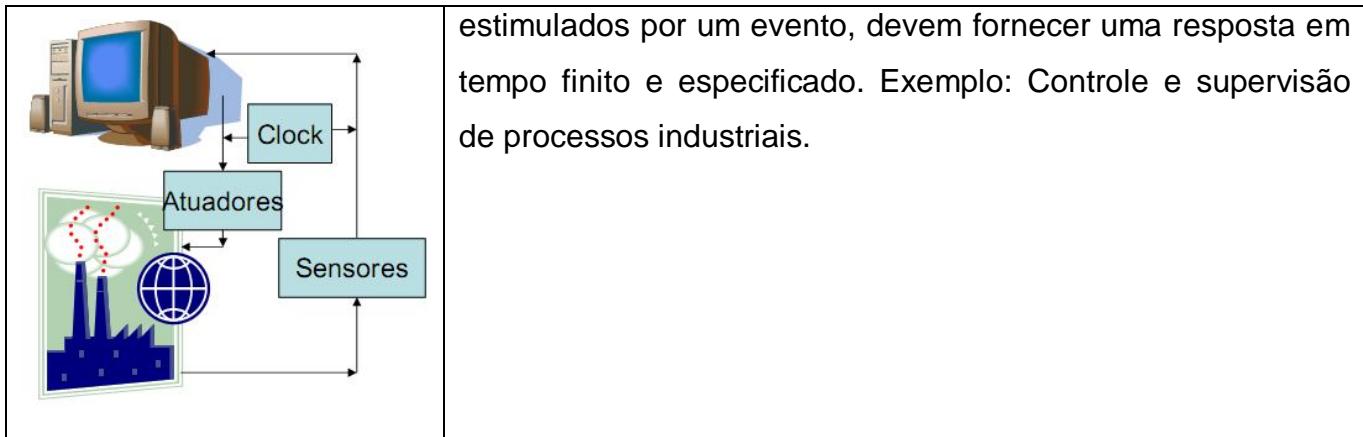
Projeto De Software De Tempo Real

Objetivo: Apresentar técnicas usadas no projeto de sistemas de tempo real e descrever algumas arquiteturas genéricas nesses sistemas.

Vamos inicialmente pegar a definição que Sommerville (2007) utiliza para explicar o que são Sistemas de Tempo Real (STR): “Um Sistema de Tempo Real é um sistema de software cujo funcionamento correto depende dos resultados produzidos pelo sistema e do tempo em que estes resultados são produzidos”.

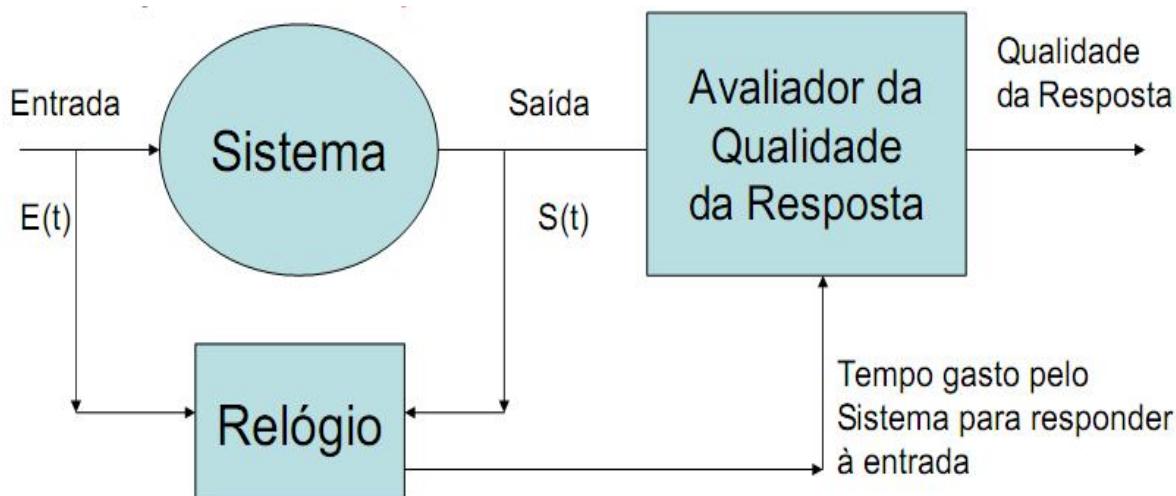
A seguir o mesmo Sommerville divide esses sistemas em duas subcategorias: leve e rígido.

Sistema de Tempo Real	Definições
LEVE (classe A)	 <p>Um Sistema de Tempo Real leve é aquele cuja operação será degradada caso os resultados não sejam produzidos de acordo com os requisitos de timing específicos. Ou ainda, quando o tempo de resposta variável é admissível. Exemplo: Sistemas on-line como Automação Bancária, reserva de passagem, etc.</p>
RÍGIDO (classe B)	<p>Um Sistema de Tempo Real rígido é aquele cuja operação será incorreta se os resultados não forem produzidos de acordo com a especificação de timing. Ou seja, quando</p>

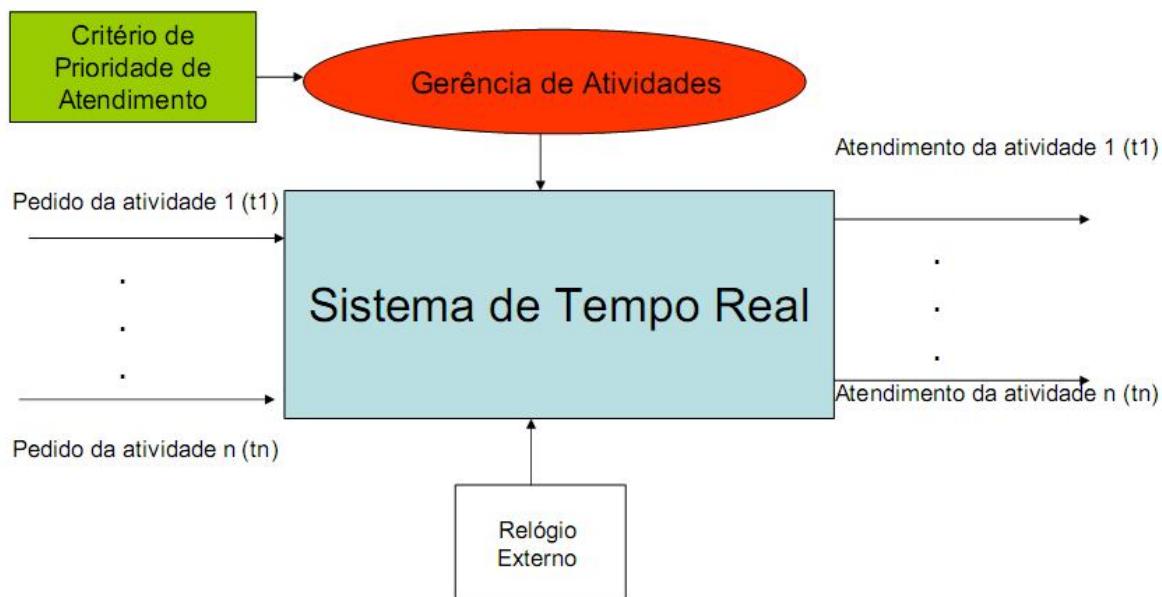


Podemos também utilizar os interessantes pontos que o Prof. Guedes, da UFRN, explica em suas aulas:

- O nome de Tempo Real é devido ao fato de que a resposta do sistema a uma dada entrada é função do tempo físico, que é externo ao sistema.
- Nos Sistemas de Tempo Real (STR), a qualidade da resposta do sistema a um dado estímulo é função do tempo externo demandado para esta resposta.
- A corretude num STR não é somente função da exatidão da resposta, mas também do tempo necessário para produzi-la.



Para atender a todas as tarefas solicitadas, dentro das suas respectivas restrições de tempo, os STR necessitam invariavelmente de alguns mecanismos de gerência de atividades, classificando-as (ordenando) segundo algum critério de prioridade de atendimento.



Sistemas Operacionais de Tempo Real (RTOS)

Quase todos os sistemas embutidos mais simples funcionam, atualmente, em conjunto com um Sistema Operacional de Tempo Real - RTOS (Real Time Operating System).

Um Sistema embutido é um computador de propósito especial, normalmente composto por um processador programado para executar tarefas específicas. Eles são encapsulados nos dispositivos por eles controlados.

No projeto APOLLO da NASA foi utilizado pela primeira vez um sistema embutido moderno. Como exemplos atuais, temos os seguintes dispositivos que utilizam de sistemas embutidos: celulares, alarme de segurança, câmera de vídeo, calculadoras, impressoras, etc.

Diferenças entre Sistemas Operacionais (OS) e os RTOS

OS toma o controle do sistema na inicialização e dispara as diferentes aplicações.	Em sistemas embutidos, a aplicação é geralmente “linkeditada” junto com o OS e é ela que é disparada na inicialização.
OS tradicionais garantem certa proteção contra erros dos aplicativos (ponteiros perdidos) de forma a garantir que os outros aplicativos não sejam afetados.	RTOS geralmente não verificam erros da aplicação.



O Robô motorizado de pesquisa a Marte tem embebido Sistemas Operacionais de Tempo Real

Microprocessador E Microcontrolador

É interessante apresentar neste momento a diferença básica entre esse dois dispositivos. O Microprocessador que basicamente é utilizado como o processador principal em nossos computadores é de propósito geral. Já os Microcontroladores são de propósitos específicos, e são utilizados em sistemas embutidos.

No entanto, o próprio Microprocessador precisa de Microcontroladores para ajudá-lo a processar nossos dados. Por exemplo, a unidade leitora de CD/DVD é constituída basicamente de Microcontroladores com objetivos específicos para aliviar o processamento da CPU principal.



Estudo Complementar

Veja o primeiro capítulo do livro “Sistemas de Tempo Real” de Jean-Marie Farines, Joni da Silva Fraga e Rômulo Silva de Oliveira:

<http://www.das.ufsc.br/~romulo/livro-tr/cap1.pdf>

<http://www.cs.bu.edu/pub/ieee-rts/Publications.html>

<http://www.dimap.ufrn.br/~jair/dstr/index.html>

<http://pet.inf.ufsc.br/projetos/real-time/index.php?q=home>



Atividades

Visite o interessante material do Prof. Luiz Affonso Guedes, da UFRN, para explorar mais o tema desta unidade:

http://www.dca.ufrn.br/~affonso/DCA0409/pdf/str_cap1.pdf

OBS.: alguns conceitos e imagens desse material foram aproveitados nesta unidade.



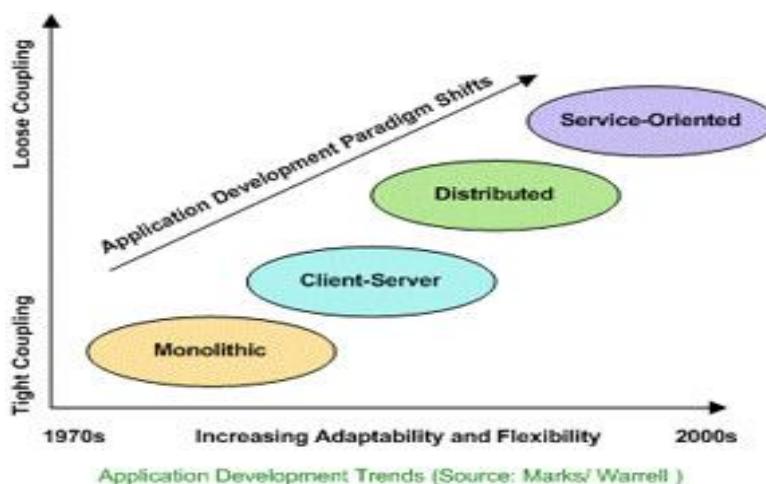
UNIDADE 18

Engenharia De Software Orientada A Serviços

Objetivo: Apresentar os principais conceitos da Engenharia de Software orientada a Serviços.

As arquiteturas orientadas a serviços – SOA (Service-Oriented Architectures) são um caminho para o desenvolvimento de sistemas distribuídos nos quais os componentes desses sistemas são serviços dedicados. Os serviços podem ser executados em computadores distribuídos geograficamente. Protocolos padronizados foram projetados para apoiar troca de serviços de comunicação e de informações.

Um serviço pode ser considerado simplesmente como uma abstração reusável. Podemos defini-lo, conforme Sommerville, como sendo: “Um componente de software reusável, não firmemente acoplado, que engloba a funcionalidade discreta que pode ser distribuída e acessada por meio de programas. Um Web Service é um serviço acessado que usa protocolos padrões da Internet e baseados em XML”.

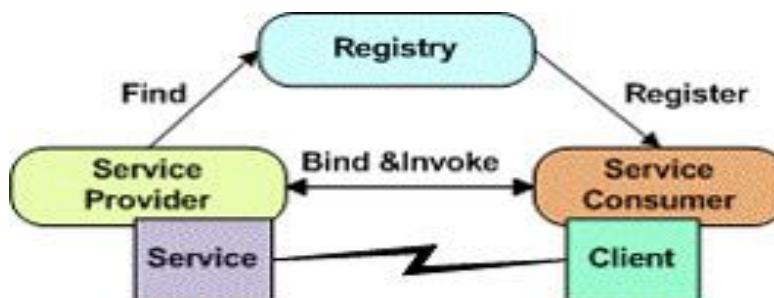


Consequentemente, os serviços são plataformas e linguagens de implementação independentes. Sistemas de software podem ser construídos usando serviços de provedores diferentes sem nenhuma ligação de interação entre esses serviços (Sommerville, 2007).

Na figura a seguir vemos como os *Web services* funcionam. Os provedores de serviços projetam e implementam serviços e os especificam em uma linguagem chamada WSDL. Eles também publicam informações sobre esses serviços em um registro de acesso geral usando um padrão de publicação chamado UDDI. Os solicitantes de serviços (algumas vezes chamados de clientes de serviços), que desejam fazer uso de um serviço, buscam o registro UDDI para descobrir a especificação desse serviço e para localizar um provedor de serviços. Eles podem então unir as suas aplicações para um serviço específico e se comunicar com ele, usando geralmente um protocolo chamado de SOAP.

Em outras palavras podemos dizer que a arquitetura Orientada a Serviços pode ser bem representada a partir do seguinte processo, chamado de "*find-bind-execute paradigm*" o que significa aproximadamente paradigma de "procura-consolida-executa". Tal conceito é análogo a "*Ciclo de Deming*" aplicado aos serviços, que define o ciclo que envolve o planejamento, a execução, o monitoramento e a tomada de ação proativa para a melhoria da qualidade.

Tecnicamente falando, o processo preconiza que os provedores de serviços registrem informações em um registro central, com suas características, indicadores, e aspectos relevantes às tomadas de decisões. O registro é utilizado pelo cliente para determinar as características dos serviços necessários, e se o mesmo estiver disponível no registro central, como por exemplo, por um catálogo de serviços, o cliente poderá utilizá-lo, sendo este oficializado através de um contrato que enderece este serviço.



Resumidamente, os padrões principais das arquiteturas orientadas a serviços são:

- SOAP (Simple Object Access Protocol): é um padrão de troca de mensagens que dá suporte à comunicação entre serviços. Ele define os componentes essenciais e opcionais das mensagens passadas entre serviços.
- WSDL (Web Service Definition Language): o padrão de linguagem de definição de serviço Web estabelece o meio pelo qual os provedores de serviços devem definir a interface para esses serviços. Essencialmente, ele permite que a interface de um serviço (operações de serviços, parâmetros e seus tipos) e suas ligações sejam definidas de maneira padronizada.
- UDDI (Universal Description, Discovery and Integration): O padrão de descrição, descoberta e integração universal define os componentes de uma especificação de serviços que pode ser usada para descobrir a existência de um serviço. Esses componentes incluem informações sobre o provedor de serviços, os serviços fornecidos, a localização da descrição de serviços (usualmente expressa em WSDL) e informações sobre relacionamentos de negócio. O UDDI registra os usuários potenciais de um serviço capazes de descobrir quais serviços estão disponíveis.



Estudo Complementar

http://pt.wikipedia.org/wiki/Service-oriented_architecture

http://pt.wikipedia.org/wiki/Simple_Object_Access_Protocol

http://www.abepro.org.br/biblioteca/ENEGEP2007_TR670485_9968.pdf

<http://cio.uol.com.br/tecnologia/2006/07/17/idgnoticia.2006-07-17.3732358054/>



Atividades

Explique em poucas palavras os princípios básicos das arquiteturas orientadas a serviços.



UNIDADE 19

Engenharia De Serviços

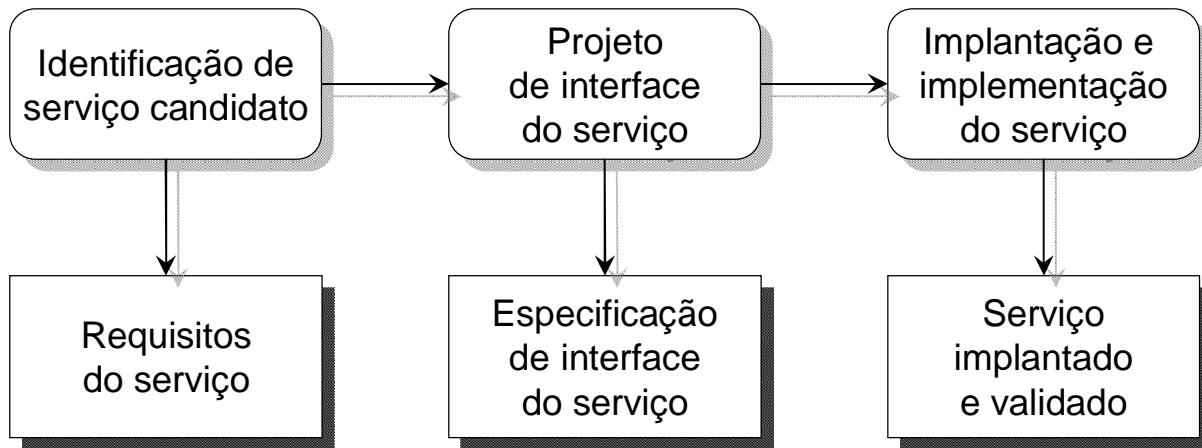
Objetivo: Apresentar os estágios lógicos no processo de engenharia de serviços

A Engenharia de Serviços é o processo de serviços de desenvolvimento para reuso nas aplicações orientadas a serviços. Ela tem muito em comum com a engenharia de componentes. Os engenheiros de serviços têm que assegurar que o serviço representa uma abstração reusável que poderia ser útil em diferentes sistemas.

Os engenheiros de serviços devem projetar e desenvolver geralmente funcionalidades úteis associadas com essas abstrações e devem assegurar que o serviço seja robusto e confiável de modo a operar confiavelmente em diferentes aplicações. Têm de documentar o serviço de modo que possa ser descoberto e compreendido por usuários potenciais.

Há três estágios lógicos no processo de engenharia de serviços. São eles:

1. Identificação do serviço candidato em que você identifica possíveis serviços que poderiam se implementados e define os requisitos do serviço.
2. Projeto do serviço no qual você projeta a lógica e as interfaces de serviços WSDL.
3. Implementação e implantação do serviço em que você implementa e testa o serviço, e o faz disponível para o uso.



Identificação De Serviço Candidato

A noção básica da computação orientada a serviços é que os serviços devem apoiar os processos de negócios. Como toda organização tem grande quantidade de processos há, portanto, muitos serviços possíveis que podem ser implementados. A identificação do serviço candidato envolve compreensão e análises dos processos de negócios da organização, para decidir quais serviços reusáveis são necessários para apoiar os processos.

Erl identifica três tipos fundamentais de serviço que podem ser identificados:

SERVIÇO de UTILIDADES:

São serviços que implementam algumas funcionalidades gerais que podem ser usadas por diferentes processos de negócios. Por exemplo, conversão de moedas.

SERVIÇO de NEGÓCIOS:

São serviços associados com uma função específica de negócio. Por exemplo, numa universidade o registro de estudantes.

SERVIÇO de COORDENAÇÃO ou de PROCESSO:

São serviços que apoiam os processos de negócio mais gerais que envolvem diferentes atores e atividades. Um exemplo seria o serviço de pedidos a serem feitos para fornecedores, mercadorias aceitas e pagamentos feitos.

Projeto De Interface Do Serviço

Uma vez selecionados os serviços candidatos, o próximo estágio no processo da Engenharia de Serviço é projetar as interfaces dos serviços. Deve-se pensar também cuidadosamente sobre como as operações e as mensagens de serviço podem ser projetadas para minimizar o número de troca de mensagens que ocorrerem para completar a solicitação de serviço.

Há três estágios para o projeto de interface de serviço:

1. Projeto de interface lógica no qual se identificam as operações associadas com o serviço, as entradas e as saídas dessas operações e as exceções associadas com essas operações.
2. Projeto de mensagem no qual se projeta a estrutura das mensagens enviadas e recebidas pelo serviço.
3. Desenvolvimento WSDL no qual se traduz o projeto lógico e de mensagem para uma descrição abstrata de interface escrita em WSDL.

Implementação E Implantação De Serviço

Uma vez identificados os serviços candidatos e projetadas as interfaces, o estágio final do processo de Engenharia de Serviços é a implementação do serviço. Essa implementação pode envolver a programação dos serviços usando uma linguagem padronizada de

programação, como Java ou C#. Ambas as linguagens atualmente incluem bibliotecas com apoio extensivo para desenvolvimento de serviços.

A implantação do serviço, o estágio final do processo, envolve tornar o serviço disponível para o uso no servidor Web. Muitos softwares de servidores tornam isso muito simples. Tem-se somente que instalar o arquivo que contém o serviço executável num diretório específico. Então, torna-se automaticamente disponível para o uso. Se a intenção for disponibilizar o serviço publicamente, tem-se de escrever uma descrição UDDI para que os usuários potenciais possam achar o serviço.

Há atualmente um número de registros públicos para descrições UDDI e os negócios podem também manter o registro privado de UDDI. Uma descrição UDDI consiste em um número de diferentes tipos de informação:

Detalhes dos negócios que fornecem o serviço.

Isso é importante por motivos de confiabilidade. Usuários de um serviço têm de ser confiáveis no sentido de que não terão comportamento malicioso. As informações sobre o provedor dos serviços permitem aos usuários verificarem as credenciais do provedor.

Descrição informal das funcionalidades fornecidas pelo serviço.

Auxilia os usuários potenciais a decidirem se o serviço é o que eles desejam. No entanto, a descrição funcional é uma linguagem natural, portanto ela não é uma descrição semântica e sem ambiguidade do que o serviço faz.

Informações sobre onde encontrar a especificação WSDL associada ao serviço

Informações de assinatura

Permitem aos usuários se registrarem para obter informações sobre atualizações de serviços.



Estudo Complementar

http://wnews.uol.com.br/site/noticias/materia_especial.php?id_secao=17&id_conteudo=425

<http://apconcursos.blogspot.com/2007/05/desenvolvimento-soa.html>



Atividades

Realize uma síntese desta unidade com o objetivo de gerar os passos principais na Engenharia de Serviços.



UNIDADE 20

Desenvolvimento De Software Orientado A Aspectos

Objetivo: apresentar o desenvolvimento de software orientado a aspectos, que é baseado na ideia de separação de assuntos em módulos de sistema separados

O desenvolvimento de software orientado a aspectos - AOSD (Aspect-Oriented Software Development) é uma abordagem emergente para desenvolvimento de software com a intenção de resolver o problema do reuso de componentes, pois, devido a eles não implementarem uma única abstração de sistema, mas também incluírem fragmentos de código que implementam outros requisitos.

O AOSD é baseado em torno de um novo tipo de abstração chamado de aspecto. Aspectos são usados junto com outras abstrações como objetos e métodos. Eles englobam funcionalidades que atravessam e coexistem com outras funcionalidades incluídas no sistema.

O benefício principal de uma abordagem orientada a aspectos é que ela apoia a separação de assuntos. A separação de assuntos em elementos independentes no lugar de incluir diferentes assuntos numa mesma abstração lógica é uma boa prática de engenharia de software.

A separação de assuntos é o principal princípio do projeto e da implementação de software. Essencialmente, ele significa que você deve organizar o seu software de modo que cada elemento no programa (classe, método, procedimento, etc.) faça uma coisa e somente uma coisa. Pode-se compreender cada parte do programa pelo conhecimento do seu assunto, sem necessitar entender outros elementos. Quando são necessárias mudanças, elas estão localizadas em um pequeno número de elementos.

Embora se concorde geralmente com que a separação de assuntos é uma boa prática de engenharia de software, é difícil definir qual é o significado real de um assunto. Algumas

vezes, ele é definido como uma noção funcional na qual um assunto é algum elemento funcional do sistema; alternativamente, ele pode ser definido de maneira mais abrangente como “alguma parte de interesse ou foco de um programa’. Um assunto é, portanto, algo de interesse ou significativo para um *stakeholder* ou para um grupo de *stakeholders*.

Com essa visão de um assunto, vemos então por que uma abordagem para implementação que separa assuntos em diferentes elementos de programa é uma boa prática. É mais fácil rastrear assuntos, definidos como requisito ou conjunto de requisitos relacionados e os componentes de programa que implementam esses assuntos. Se os requisitos mudam (e mudam constantemente !), a parte do programa que precisa ser mudada é obvia.

Existem muitos tipos diferentes de assuntos de *stakeholders*:

Assuntos funcionais

Relacionados a uma funcionalidade específica a ser incluída em um sistema. Por exemplo, num sistema de controle de trens, um assunto funcional específico é o freio do trem.

Assuntos de qualidade de serviço

Que estão relacionados ao ambiente não funcional de um sistema. Esses assuntos incluem características como desempenho, confiabilidade e disponibilidade.

Assuntos de política

Relacionados às políticas gerais que governam o uso do sistema. Assuntos de política incluem assuntos de proteção e segurança e assuntos relacionados às regras de negócio.

Assuntos de sistema

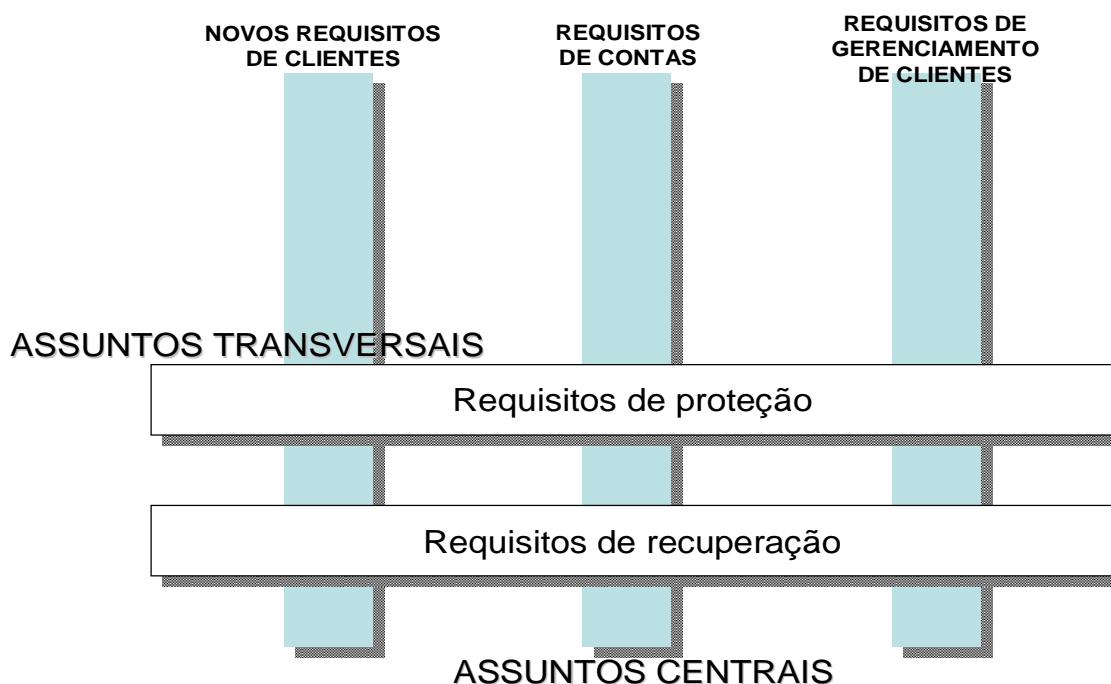
Relacionados aos tributos do sistema como um todo, como sua facilidade de manutenção e de configuração.

Assuntos organizacionais

Relacionados aos objetivos e às prioridades organizacionais como a produção de um sistema dentro do orçamento, o uso de ativos de software existentes ou a manutenção de reputação da organização.

Assuntos de sistema

Aplicam-se ao sistema como um todo em vez de requisitos individuais ou à realização dos requisitos em um programa. A esses assuntos secundários chamamos de transversais para distingui-los de assuntos centrais. Veja na figura abaixo os conceitos anteriores com um exemplo de um sistema bancário de Internet.





Estudo Complementar

<http://www.webartigos.com/articles/1954/1/Projeto-de-Software-Orientado-a-Aspectos/Pagina1.html>

http://twiki.cin.ufpe.br/twiki/pub/SPG/GenteAreaPublications/SBES04_soares.pdf

<http://aspect-oriented.awardspace.com/desvpoa.html>



Atividades

Realize uma pesquisa na Web sobre os locais em que está sendo aplicada a tecnologia AOSD.



Atividades

Antes de dar continuidades aos seus estudos é fundamental que você acesse sua SALA DE AULA e faça a Atividade 2 no “link” ATIVIDADES.



UNIDADE 21

Engenharia De Software Com Aspectos

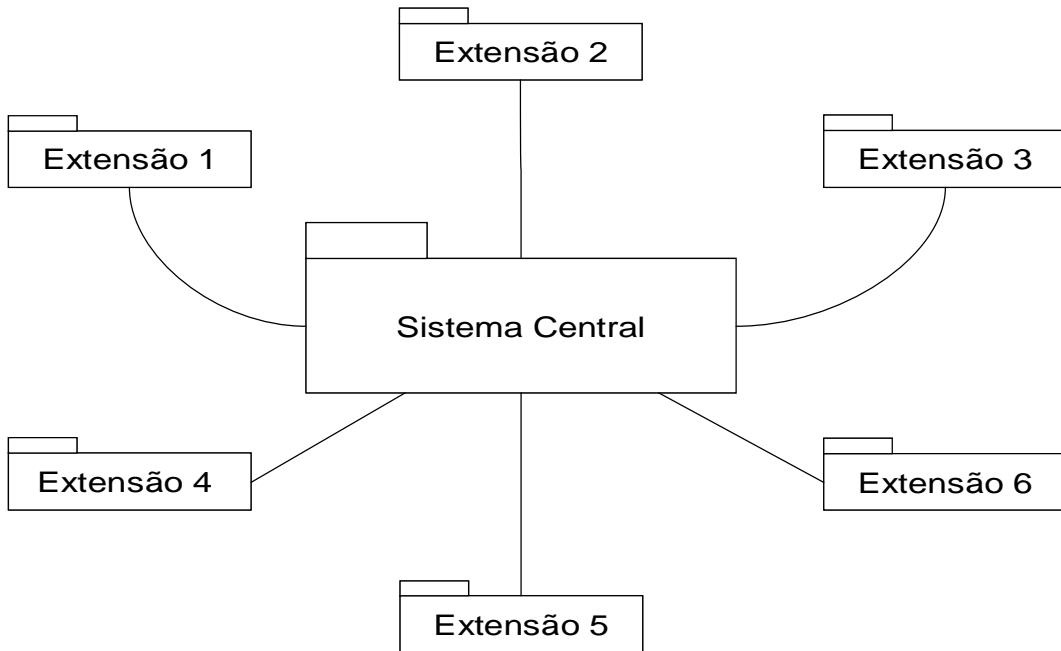
Objetivo: Conceituar os princípios da Engenharia de Software com Aspectos.

Os aspectos foram inicialmente introduzidos como uma construção de linguagem de programação, o conceito de assuntos vem de requisitos de sistema. Portanto, faz sentido adotar uma abordagem orientada a aspectos em todos os estágios do processo de desenvolvimento de sistema.

Nos estágios iniciais de Engenharia de Software, a adoção de uma abordagem orientada a aspectos significa usar o conceito de separação de assuntos como base para pensar em requisitos e o projeto de sistemas.

A identificação e a modelagem de assuntos devem ser parte dos processos de engenharia de requisitos e projeto. As linguagens de programação orientada a aspectos fornecem, portanto, o apoio tecnológico para manter a separação de assuntos na sua implementação do sistema.

Jacobsen e Ng (2004) em seu livro “Aspect-oriented software development with use cases” sugerem que se deva pensar num sistema que apoie diferentes assuntos envolvidos como um sistema central mais extensões. Na figura a seguir através de *packages* UML são representadas essas ideias representando tanto o centro quanto as extensões. O sistema central é o conjunto de características de sistema que fornecem apoio para o propósito essencial do sistema.



Portanto, se o propósito do sistema é manter a informação de pacientes em um hospital, o sistema fornece um meio para criar, editar, gerenciar e acessar um banco de dados de registros de pacientes. As extensões para o Sistema Central refletem assuntos adicionais dos *stakeholders* do sistema que devem ser integrados com o sistema central. Por exemplo, é importante que o sistema de informação de hospital mantenha a confidencialidade de informações de pacientes e, assim, uma extensão possa ser dedicada ao controle de acesso, outra à criptografia, etc.

Há uma quantidade de tipos de extensões derivadas de diferentes tipos de assuntos:

EXTENSÕES FUNCIONAIS SECUNDÁRIAS:

Essas extensões adicionam capacidades funcionais para funcionalidades fornecidas no Sistema Central. Por exemplo, a produção de relatórios de medicamentos prescritos no mês anterior seria uma extensão funcional secundária para um sistema de informação de pacientes.

EXTENSÕES DE POLÍTICAS:

Essas extensões adicionam capacidades funcionais para apoiar algumas políticas organizacionais. Extensões que adicionam características de proteção são exemplos de extensões de políticas.

EXTENSÕES de QoS (Quality of Service):

Essas extensões adicionam capacidades funcionais para auxiliar a atender os requisitos da qualidade de serviço – QoS que forem especificados para o sistema. Por exemplo, uma extensão poderia fornecer apoio a um cachê para reduzir o número de acessos ao Banco de Dados ou a *back-ups* automáticos para se recuperar no evento de uma falha de sistema.

EXTENSÕES DE INFRA-ESTRUTURA:

Essas extensões adicionam capacidades funcionais para dar suporte à implementação de um sistema em alguma plataforma específica. Por exemplo, em um sistema de informação de pacientes, as extensões de infraestrutura poderiam ser usadas para implementar a interface para o sistema de gerenciamento de Banco de Dados. Se isso mudar, as mudanças poderão ser feitas por meio da alteração de extensões de infraestrutura associadas.

As extensões adicionam sempre algum tipo de funcionalidade ou características para o Sistema Central. Aspectos são um meio para implementar as extensões e podem ser compostos com a funcionalidade do Sistema Central usando os recursos de composição no ambiente de programação orientado a aspectos.



Atividades

Quais são os tipos de extensões derivadas de diferentes tipos de assuntos apresentadas nesta unidade?



UNIDADE 22

Estimativa De Custo De Software

Objetivo: apresentar técnicas de estimativa de custo e esforço necessários para a produção de software.

A estimativa de recursos, de custo e de cronograma para um esforço de engenharia de software exige experiência, acesso a boas informações históricas (métricas) e coragem de se empenhar em previsões quantitativas, quando informação qualitativa é tudo que existe. Estimativa tem risco inerente e esse risco leva à incerteza.

Sempre que são feitas estimativas, olhamos para o futuro e aceitamos algum grau de incerteza como inevitável. Apesar de estimar ser tanto arte como ciência, essa importante atividade não precisa ser conduzida de modo aleatório. Já existem técnicas úteis para estimativa de tempo e esforço. Métricas de processo e projeto podem fornecer perspectiva histórica e insumo poderoso para a geração de estimativas quantitativas. Experiência anterior, de todo o pessoal envolvido, pode ajudar imensamente, à medida que estimativas são desenvolvidas e revistas.



Como a estimativa estabelece uma base para todas as outras atividades de planejamento de projeto e como este fornece um guia para a engenharia de software bem-sucedida, seria temerário começar sem ela.

A disponibilidade de informação histórica tem forte influência no risco da estimativa. Olhando para trás, podemos imitar coisa que funcionaram e aperfeiçoar áreas em que surgiram problemas. Quando métricas de software abrangentes, oriundas de projetos anteriores, estão

disponíveis, as estimativas podem ser feitas com maior segurança, os cronogramas podem ser estabelecidos para evitar dificuldades enfrentadas antes e o risco global é reduzido.

O risco da estimativa é medido pelo grau de incerteza das estimativas quantitativas estabelecidas para recursos, custo e tempo. Se o escopo do projeto é mal entendido ou se os requisitos estão sujeitos a mudanças, a incerteza e risco tornam-se perigosamente altos.

O planejador e, principalmente, o cliente devem reconhecer que variabilidade nos requisitos de software significa instabilidade no custo e no cronograma.

O objetivo do planejamento de projeto é fornecer um arcabouço que permita ao gerente fazer estimativas razoáveis de recursos, custo e cronograma. Além disso, as estimativas devem tentar definir cenários (positivos e negativos) correspondentes tanto ao melhor quanto ao pior caso, de modo que o comportamento do projeto possa ser delimitado. Assim, o plano deve ser adaptado e atualizado à medida que o projeto avançar.

Conforme Pressman (2006), sugere-se um conjunto de tarefas para o devido planejamento do projeto quanto às estimativas de custo do software:

- **Estabeleça o escopo do projeto**
- **Determine a viabilidade**
- **Analise riscos**
- **Defina recursos necessários**
- Determine recursos humanos necessários
- Defina recursos reusáveis de software
- Identifique os recursos que o ambiente oferece
- **Estime custo e esforço**

- Decomponha o problema
- Desenvolva duas ou mais estimativas usando tamanho, ponto por função, tarefas de processo, ou casos de uso
- Harmonize as estimativas
- **Desenvolva um Cronograma do Projeto**
- Estabeleça um conjunto significativo de tarefas
- Defina uma rede de tarefas a serem realizadas
- Use ferramentas para desenvolver um diagrama de tempo
- Defina mecanismos de rastreamento do cronograma



Estudo Complementar

http://www.consiste.dimap.ufrn.br/~claudia/Mestrado/Disciplinas/Engenharia_Software/Artigo%5B6%5D_Estimativas.pdf

<http://sunset.usc.edu/research/cocomosuite/index.html>

<http://www.linhadecodigo.com.br/Artigo.aspx?id=102>

<http://www.inf.ufrgs.br/pos/SemanaAcademica/Semana99/mariaisabel/mariaisabel.html>





Atividades

Faça uma análise crítica do conjunto de tarefas sugerido por Pressman para o devido planejamento do projeto quanto às estimativas de custo do software.



UNIDADE 23

Métricas Para Pequenas Organizações

Objetivo: Apresentar possíveis formas de trabalhar com métricas num pequena organização

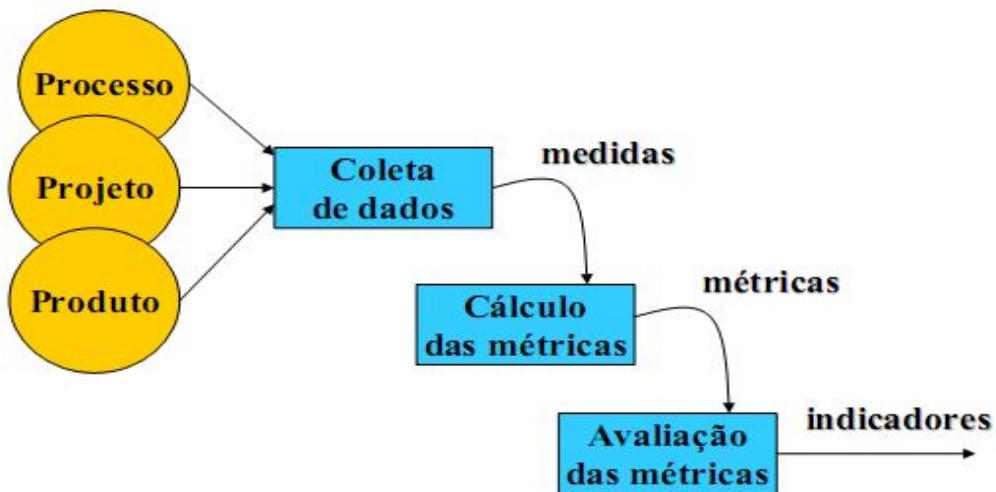
Medição

É uma ferramenta de gestão. Se conduzida adequadamente, fornece conhecimentos a um gerente de projeto. E, como resultado, apóia o gerente de projeto e a equipe de software na tomada de decisões que irão conduzir a um projeto de sucesso. Medição pode ser aplicada ao processo de software com o objetivo de melhorá-lo de forma contínua.

No livro de diretrizes sobre medição de software, Park, Goethert e Florac (1996) discutem as razões pelas quais medimos:

- Para caracterizar um esforço a fim de obter entendimento de processos, produtos, recursos e ambientes, e para estabelecer referências para comparação com futuras avaliações.
- Para avaliar a fim de determinar o estado em relação aos planos.
- Para prever pela obtenção de entendimento de relacionamentos entre processos, produtos e construção de modelos desses relacionamentos.
- Para aperfeiçoar pela identificação de bloqueios, causas fundamentais, ineficiências e outras oportunidades para melhorar a qualidade do produto e o desempenho do processo.

A grande maioria das organizações de desenvolvimento possui menos de 20 profissionais de software. É irracional, e, na maior parte dos casos, inviável esperar que essas organizações desenvolvam programas abrangentes de métricas de software.



Todavia, é razoável sugerir que organizações de software de todos os tamanhos meçam e depois usem as métricas resultantes para ajudar a aperfeiçoar seu processo local de software, e a qualidade e pontualidade dos produtos que produzem.

A atividade de medir ajuda aos gerentes e componentes da equipe a melhorar o processo de desenvolvimento de software. Para calcular as métricas são utilizadas medições de atributos específicos do processo, projeto e produto. As métricas permitem a organização a ter uma visão estratégica, oferecendo informações sobre o processo de software. Medição resulta em uma mudança na cultura da empresa.

Uma abordagem de bom senso para a implementação de qualquer atividade relacionada a processo de software seria:

- Que seja simples;
- Ajuste às necessidades locais; e

- Certificar-se de que tenha valor.

A simplicidade é uma diretriz que funciona razoavelmente bem em muitas atividades. Contudo, como originarmos um conjunto de métricas de software “simples”, que ainda mesmo assim tenha valor, e como podemos estar certos de que essas métricas “simples” vão satisfazer às necessidades da minha organização?!?

A chave é começarmos focalizando não a medição, mas os resultados!!

O grupo de software é interrogado para definir um único objetivo que requer aperfeiçoamento. Por exemplo, “reduza o prazo para avaliar e implementar pedidos de modificação”. Para este caso específico, uma pequena empresa poderia selecionar o seguinte conjunto de medidas facilmente coletáveis:

- t_{fila} → TEMPO (horas ou dias) transcorrido entre o momento em que o pedido é feito até que a avaliação esteja completa.
- W_{aval} → ESFORÇO (homem/hora - h/h) para realizar a avaliação.
- t_{aval} → TEMPO (horas ou dias) transcorrido desde o término da avaliação até a atribuição da ordem de modificação ao pessoal.
- W_{mod} → ESFORÇO (homem/hora - h/h) necessário para fazer a modificação.
- t_{mod} → TEMPO (horas ou dias) necessário para fazer a modificação.
- E_{mod} → ERROS descobertos durante o trabalho para fazer a modificação.
- D_{mod} → DEFEITOS descobertos depois que a modificação é entregue ao cliente.

Com base nesses dados coletados, para certo número de pedidos de modificação, é possível calcular:

- O tempo total transcorrido desde o pedido até a implementação da modificação;
- Porcentagem do tempo transcorrido, absorvida pelo escalonamento inicial, avaliação, atribuição e implementação da modificação;

- Porcentagem do esforço necessário para avaliação e implementação.

Ou seja, os percentuais de tempo gastos em cada fase são determinados e, então, podem ser melhorados. Essas métricas podem ser avaliadas no contexto dos dados de qualidade E_{mod} e D_{mod} .

As porcentagens permitem discernir onde o processo do pedido de modificação sofre demora, e podem levar a passos de melhoria do processo para reduzir t_{fila} , W_{aval} , t_{aval} , W_{mod} , e/ou E_{mod} . Além disso, a eficiência na remoção de defeitos pode ser calculada como:

$$DRE = E_{mod} / (E_{mod} + D_{mod})$$

DRE pode ser comparada com o tempo transcorrido e o esforço total para determinar o impacto das atividades de garantia pelo tempo e o esforço necessários para fazer uma modificação.

As vantagens são muito grandes, além de que o custo, no início, de coletar medidas é da ordem de 3 a 8% do orçamento do projeto, e cai para 1% depois que a equipe começa a se familiarizar e ver os benefícios dessas medidas.

Estabelecendo Um Programa De Métricas

O Software Engineering Institute - SEI desenvolveu um manual de diretrizes abrangentes para estabelecer um programa de métricas de software “orientado a metas”, que sugere os seguintes passos:

- Identifique as metas do seu negócio
- Identifique o que você necessita conhecer ou aprender
- Identifique suas submetas
- Identifique as entidades e atributos relacionados às suas submetas
- Formalize suas metas de medição

- Identificar as perguntas e os indicadores desejados
- Identificar os elementos de dados requeridos
- Definir a medidas a serem feitas e torná-las operacionais
- Identifique as ações a serem executadas para implementar as medidas
- Prepare um plano para implementar as medições



Estudo Complementar

http://pt.wikipedia.org/wiki/M%C3%A9tricas_de_software

www.des.ime.eb.br/~cgmello/gep/4-GEP-Metricas.pdf



Atividades

Realize uma pesquisa na sua empresa, ou na de colegas, e verifique quais são as métricas adotadas pela área de sistemas.



UNIDADE 24

Estimativa Do Projeto De Software

Objetivo: apresentar técnicas de estimativa de custo e esforço necessários para a produção de software.

A estimativa de custo e de esforço de software nunca será uma ciência exata. Um demasiado número de variáveis – humanas, técnicas, ambientais, políticas – podem afetar o custo final do software e o esforço aplicado para desenvolvê-lo. Todavia, a estimativa de projetos de software pode ser transformada de algo sobrenatural em uma série de passos sistemáticos que fornecem estimativas com risco aceitável. Algumas das opções para conseguir estimativas de custo e esforço confiáveis são:

- Adie a estimativa até que o projeto esteja o mais adiantado.
- Baseie as estimativas em projetos semelhantes, que já foram completados.
- Use técnicas de decomposição relativamente simples para gerar estimativas de custo e esforço do projeto.
- Use um ou mais modelos empíricos para estimativa de custo e esforço do software.

Técnicas de decomposição usam a abordagem “dividir para conquistar” para a estimativa de projetos de software. Pela decomposição de um projeto em suas funções principais e atividades relacionadas de engenharia de software, a estimativa de custo e esforço pode ser realizada passo a passo.

Modelos empíricos de estimativa podem ser usados para complementar as técnicas de decomposição e oferecem por si mesmos uma valiosa abordagem de estimativa.

Técnicas De Decomposição

A estimativa de projeto é boa na mesma medida em que o é a estimativa do tamanho do trabalho a ser realizado, o dimensionamento representa o primeiro grande desafio do planejador do projeto. O “tamanho” do software a ser construído pode ser estimado usando uma medida direta (LOC – Linhas de Código), ou uma medida indireta (FP – Pontos de Função).

As estimativas LOC e FP são técnicas distintas. Todavia, ambas têm algumas características em comum. O planejador do projeto começa com uma declaração delimitada do escopo do software e a partir dela tenta decompor o software em funções do problema que podem ser estimadas individualmente.

LOC ou FP, a variável de estimativa, é então estimada para cada função. Como alternativa, o planejador pode escolher outro componente para dimensionar, como classes ou objetos, modificações ou processos do negócio diretamente afetados.

Uma vantagem dos PF sobre as LOC é que os Pontos de Função podem ser obtidos logo no início do ciclo de vida, diretamente dos requisitos ou especificações. Os PF são úteis para estimativas independentes de linguagem, realizadas no início do ciclo de vida. Por outro lado, a utilização das LOC na previsão do esforço total de um projeto continua tendo sucesso para uma ampla quantidade de projetos, envolvendo diversas linguagens.

Embora Casos de Uso forneçam a uma equipe de software discernimento do escopo e requisitos do software, desenvolver uma abordagem de estimativa com Casos de Uso é problemático pelas seguintes razões:

- Casos de Uso são descritos usando muitos formatos e estilos diferentes (não há um formato padrão).
- Casos de Uso representam uma visão externa (a visão do usuário) do software e são frequentemente escritos em diferentes níveis de abstração.

- Casos de Uso não tratam da complexidade e das características das funções que são descritas.
- Casos de Uso não descrevem comportamentos complexos (por exemplo, interações) que envolvem muitas funções e características.

Diferentemente de uma LOC ou FP, um “caso de uso” de uma pessoa pode precisar meses de esforço, enquanto um “caso de uso” de outra pessoa pode ser implementado em um ou dois dias.

Modelos De Estimativa Empíricos

Os dados empíricos que apoiam a maioria dos modelos de estimativa são derivados de uma amostra limitada de projetos. Por esse motivo, nenhum modelo de estimativa é adequado a todas as classes de software e a todos os ambientes de desenvolvimento. Assim, os resultados obtidos de tais modelos devem ser usados cuidadosamente.

Um modelo de estimativa típico é derivado usando análise de regressão de dados coletados em projetos de software anteriores. Um modelo de estimativa deve ser calibrado para refletir as condições locais. O modelo deve ser testado aplicando os dados coletados de projetos já finalizados, ligando os dados ao modelo, e depois comparando os resultados reais com os previstos. Se a concordância for baixa, o modelo deverá ser sintonizado e retestado antes que possa ser usado.



Estudo Complementar

www.cefetrn.br/~placido/disciplina/pgp/aulas/Metricas.pdf





Atividades

Quais são as principais diferenças existentes entre os conceitos de LOC e FP ?



UNIDADE 25

Modelo De Estimativa De Software – Cocomo II

Objetivo: Conceituar o modelo clássico de estimativa de software.

Em seu livro clássico sobre a “Economia da Engenharia de Software”, Barry Boehm (1981), introduziu uma hierarquia de modelos de estimativa de software denominada COCOMO – **COnstructive COst MOdel**, que significa modelo construtivo de custo.

O modelo COCOMO original tornou-se um dos modelos de estimativa de custo de software mais amplamente usado e discutido na indústria. É um método que busca medir esforço, prazo, tamanho de equipe e custo necessário para o desenvolvimento do software, desde que se tenha a dimensão do mesmo, através de um modelo de estimativa de tamanho de software, como Análise de Pontos de Função. Evoluiu para um modelo de estimativa mais abrangente, chamado de COCOMO II, versão publicada em 2000.



O Modelo COCOMO II foi originalmente calibrado com dados de 161 projetos. Os

mesmos foram selecionados dentre mais de 2000 projetos candidatos. Para cada um dos 161 projetos escolhidos foram realizadas entrevistas e visitas, a fim de garantir a consistência das definições e suposições do modelo. O modelo nominal vem calibrado para esses projetos, cuja natureza pode diferir daquele que se deseja estimar.

Como seu predecessor, o COCOMO II é na verdade uma hierarquia de modelos de estimativa que tratam das seguintes áreas:

Modelo de composição da aplicação:

Usado durante os primeiros estágios da Engenharia de Software, quando a prototipagem das interfaces com o usuário, a consideração da interação do software com o sistema, a avaliação do desempenho e o julgamento da maturidade tecnológica são de extrema importância.

Modelo do primeiro estágio de projeto:

Usado após os requisitos terem sido estabilizados e a arquitetura básica do software ter sido estabelecida.

Modelo para o estágio após a arquitetura:

Usado durante a construção do software.

Os objetivos do COCOMO II foram definidos para suportar as necessidades primárias da estimativa de custo, devendo fornecer suporte para a etapa de planejamento de projetos, equipe de desenvolvimento necessária, preparação inicial do projeto, replanejamento, negociação de contratos, avaliações de propostas, necessidades e níveis de recursos (tecnológicos e humanos).

Como todos os modelos para estimativa de software, o modelo COCOMO II requer informação de tamanho. Três diferentes opções de dimensionamento estão disponíveis como partes da hierarquia de modelos:

1. Pontos por objeto
2. Pontos por função
3. Linhas de código-fonte

Para o cálculo do custo deve-se conhecer o prazo e equipe de trabalho, para então chegar ao valor, sendo que para definir o tamanho do programa, torna-se necessário que se caracterize que medida será adotada: linhas de código, pontos por função ou pontos por caso de uso (o menos adequado!).

O CoCoMo II estima o custo e tempo baseado em pessoas/mês e meses, para a determinação do *baseline* de exigências de um produto para a conclusão de uma atividade. Define-se *baseline* como o conjunto de produtos aceitos e controlados, e que serão utilizados em atividades posteriores à sua aceitação. O modelo ainda prevê um adicional de 20% ao tempo computado, como margem de erro (análise de risco).

Embora o CoCoMo II possa ser executado com os parâmetros nominais, pressupõe-se a calibração para o ambiente-alvo. Com a calibração em projetos do mesmo ramo é possível modificar o valor das constantes das fórmulas padrões. Na ausência de dados históricos disponíveis para o ambiente-alvo em questão, devem ser selecionados projetos equivalentes para efetuar a calibração.

Devido ao seu impacto exponencial, não é recomendável calibrar o modelo quando houver menos de 10 projetos disponíveis para a calibração, conforme Boehm (2000).



Estudo Complementar

http://pt.wikipedia.org/wiki/M%C3%A9todo_COCOMO

<http://en.wikipedia.org/wiki/COCOMO>

www.ulbra-to.br/ensino/43020/artigos/anais2005/anais/COCOMO.pdf

www.bfpug.com.br/islig-rio/Downloads/Estimando_Projetos_COCOMO_II.pdf



Atividades

Visite o site http://sunset.usc.edu/research/COCOMOII/cocomo_main.html e relate as principais características do software de apoio ao modelo.



UNIDADE 26

Engenharia De Software Para A Web

Objetivo: Apresentar as principais características da Engenharia de Sofware para a Web.

Sistemas e aplicações baseados na Web, também chamados de WebApps, englobam desde uma simples página Web com recursos de interação com o usuário, até um abrangente site fornecedor de serviços completos para toda uma comunidade específica de internautas.

Atualmente as WebApps evoluíram tanto com ferramentas computacionais sofisticas que não fornecem somente funções isoladas para o usuário final, mas também são integradas com banco de dados corporativos e aplicações de negócio (Pressman, 2006).

As aplicações na Internet, em sua grande maioria, possuem os seguintes atributos:

CONCENTRAÇÃO em REDES:

Necessariamente uma WebApp reside numa rede para atender uma comunidade diversificada de usuários. Existe a possibilidade de essa aplicação rodar tanto na própria Internet, que é o mais comum, como numa Intranet e nas Extranets. No caso da Intranet essas aplicações facilitam o processo de integração e comunicação dentro das organizações, e no caso da Extranet para aprimorar o relacionamento com os seus parceiros (fornecedores, clientes, distribuidores, etc.).

CONCORRÊNCIA:

Qualquer WebApp que tenha muitos usuários terá o problema de ser acessada simultaneamente ao mesmo tempo. Inclusive com a possibilidade de ataques do tipo denial-of-service (DoS) por hackers, com o intuito de impedir que um usuário legítimo utilize um determinado serviço.

CARGA IMPREVISÍVEL:

Embora seja bastante difícil, em certas aplicações, de dimensionar a quantidade de usuários que acessarão os serviços, existe a necessidade de controlar constantemente a carga dos servidores. Se a aplicação não for bem dimensionada e proporcional à demanda dos usuários, pode ocorrer inclusive travamento do servidor. Novamente um ataque por hackers pode camuflar todas as estatísticas de acesso.

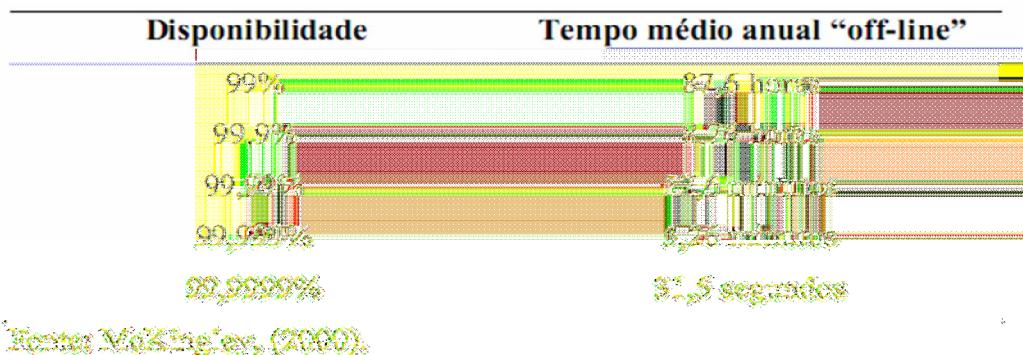
DESEMPENHO:

O tempo de resposta da aplicação Web tem que ser muito bem avaliada. Se os usuários são clientes de um serviço que facilmente encontrará em outro site, ocorrerá uma grande evasão se os tempos de resposta forem muito longos e decorrerão reclamações em quantidade.

DISPONIBILIDADE:

A grande maioria dos provedores trabalha com a metodologia 6 sigma, tentando atingir a disponibilidade de 4 dígitos depois da vírgula, pois dificilmente chegaremos a 100% de disponibilidade. Veja na tabela a seguir que cada vez que avançamos numa casa decimal, o impacto das horas fora do ar (“off-line”), em um ano, decresce significativamente.

O uso da Internet está tão disseminado que o sonho dos usuários é a base de “24 / 7 / 365” (24 horas por dia, 7 dias na semana, 365 dias no ano !!).



ORIENTADA a DADOS:

Embora a função principal de muitas WebApps sejam de usar os recursos de hipermídia (texto, gráficos, áudio, vídeo) ao usuário final, atualmente é muito comum elas acessarem grandes Bancos de Dados. Um exemplo típico é o e-commerce e as aplicações financeiras.

SENSÍVEL ao CONTEÚDO:

Um bom exemplo desse aspecto é o próprio Google. Se as várias ferramentas desse mecanismo de busca não fossem tão significativas em termo de conteúdo para os usuários, essa empresa não teria a projeção que tem hoje.

EVOLUÇÃO CONTINUADA:

Uma das premissas da atual Web 2.0 é o “beta teste eterno”, ou seja, a primeira versão do aplicativo é liberada aos usuários e mesmo assim continuará recebendo melhorias constantes ao longo do projeto. As aplicações Web evoluem continuamente.

IMEDIATISMO:

É freqüente que as aplicações da Internet tenham que estar disponibilizadas para os usuários em questões de semanas ou mesmo em dias. Portanto, os engenheiros Web precisam usar métodos de planejamento, análise, projeto, implementação e teste especiais conforme as exigências do desenvolvimento de WebApp.

SEGURANÇA:

A fim de proteger conteúdo reservado e fornecer modos seguros de transmissão de dados, fortes medidas de segurança precisam ser implementadas em toda a infraestrutura que apoia uma WebApp e na aplicação propriamente dita. A nossa unidade intitulada “Engenharia de Proteção” trata desse assunto com maiores detalhes.

ESTÉTICA:

Interessante como esse atributo é especial para a Web. Existem aplicações que com seu aspecto específico atraem grande número de usuários. Podemos citar novamente o mecanismo de busca do Google como exemplo. Através de sua interface minimalista essa aplicação ganhou praticamente todos os internautas. Por outro lado, a Amazon impressiona pela riqueza de detalhes que fornece aos seus usuários.



A engenharia de software para WebApps pode ser definida segundo Graef e Gaedke como: “Aplicação de um enfoque sistemático, disciplinado e quantificável para o desenvolvimento e evolução de aplicações Web, com alta qualidade e a um custo efetivo”.



Estudo Complementar

<http://www.sembrasil.com.br/artigos/engenharia-de-software-x-engenharia-web.html>

[http://www.ic.unicamp.br/~eliane/Cursos/MO409/Curso2004\(com%20MC746\)/Apresentacoes/OOHDM.ppt#256,1,Metodologia para Desenvolvimento Web](http://www.ic.unicamp.br/~eliane/Cursos/MO409/Curso2004(com%20MC746)/Apresentacoes/OOHDM.ppt#256,1,Metodologia%20para%20Desenvolvimento%20Web)

<http://www.inf.ufsc.br/~leandro/ensino/esp/monografiaMarcioHenriqueLocatelli.pdf>

<http://www.cin.ufpe.br/~processos/TAES3/slides-2003.1/RUPparaWeb.ppt#281,1,Slide 1>

<http://www.inf.ufsc.br/~leandro/ensino/esp/monografiaMarcioHenriqueLocatelli.pdf>



Atividades

Cite de memória pelo menos cinco características das aplicações feitas para a Internet.



UNIDADE 27

Engenharia De Software Para A Web – Metodologia

Objetivo: Destacar as metodologias mais adequadas para Engenharia de Software na Web.

É importante notar que muitos métodos da Engenharia Web foram adaptados diretamente de seus correlatos da Engenharia de Software clássica. Outros ainda estão em seus estágios de formação e em processo de maturidade. Alguns deles vão sobreviver, outros serão descartados à medida que melhores abordagens forem sugeridas (adaptado de Pressman, 2006).

O processo WebE (Engenharia Web) adota a filosofia de desenvolvimento ágil que enfatiza uma abordagem de engenharia “simples” que leva à entrega incremental do sistema a ser construído.

O modelo de processo da WebE está fixado em três pontos:

1. *ENTREGA INCREMENTAL*: as atividades vão ocorrer repetidamente à medida que cada incremento é submetido à engenharia e entregue;
2. *MODIFICAÇÕES CONTÍNUAS*: as modificações ocorrem devido aos resultados da avaliação de um incremento entregue, ou como consequência de condições de negócios mutáveis;
3. *CRONOGRAMAS CURTOS*: com prazos curtos exige a necessidade de criar estratégias especiais para a documentação de engenharia. No entanto, a análise, projeto e teste crítico devem ser registrados de algum modo.

Susan Weinshenk, em seu livro “Psychology and the Web: Designing for People”, sugere um conjunto de questões que devem ser consideradas à medida que à análise e o projeto progridem. Um pequeno subconjunto é aqui citado:

- Quão importante é uma página principal de um site Web? Ela deve conter informação útil ou uma simples lista de *links* que levam um usuário a ter mais detalhes em níveis mais baixos?
- Qual é o leiaute de página mais efetivo (por exemplo, menu no topo, à direita ou à esquerda?) e ele varia dependendo de tipo da WebApp que está sendo desenvolvido?
- Que opções de mídia têm maior impacto? Os gráficos são mais efetivos do que o texto? O vídeo (ou áudio) é uma opção eficaz? Quando as várias opções de mídia devem ser escolhidas?
- Quanto de trabalho pode-se esperar que um usuário faça quando ele ou ela está procurando a informação? Quantos cliques as pessoas estão dispostas a fazer?
- Quão importantes são os apoios navegacionais quando as WebApps são complexas ?
- Quão complexo pode ser o formulário de entrada antes que ele se torne irritante para o usuário? Como os formulários de entrada podem ser manipulados?
- Quão importantes são os recursos de busca? Que porcentagem de usuários navega, e que porcentagem usa buscas específicas? Quão importante é estruturar cada página de um modo que aceite um link de alguma fonte externa?
- A WebApp será projetada para que seja acessível àqueles que têm deficiências físicas ou outras ?

Veja a importância do planejamento em tudo isso através das palavras de Constantine e Lockwood (“User-Centered Engineering for Web Applications”, 2002):

“Apesar das declarações sem fôlego de que a Web representa um novo paradigma definido por novas regras, os desenvolvedores profissionais estão percebendo que as lições aprendidas nos dias do desenvolvimento de software antes da Internet ainda se aplicam. Páginas Web são interfaces com o usuário. Programação HTML é programação, e aplicações implantadas por navegador são sistemas de software que podem se beneficiar dos princípios básicos e engenharia de software.”



Atividades

Como você julga a qualidade de um site Web? Faça uma lista ordenada por ordem de importância dos atributos de qualidade que você acredita ser os mais importantes.



UNIDADE 28

Engenharia De Software Para A Web – Oohdm

Objetivo: Definir o método OOHDm e suas principais características.

As WebApps estão tornando-se cada vez mais complexas devido aos vários aspectos e tecnologias adicionais e específicas, além da complexidade natural do negócio. No entanto, os métodos e as técnicas tradicionais da Engenharia de Software não são adequados para o domínio Web, que tem requisitos próprios.

Segundo Rossi, Schwabe e Lyardet (1999), as principais diferenças entre WebApp e aplicações tradicionais (cliente-servidor) referem-se a questões de navegação, de interface e de implementação.

Tendo em vista que WebApp pode ser considerada um tipo de produto de software, vários métodos específicos para o seu desenvolvimento têm sido propostos e sistematizam as fases de análise, de projeto e de implementação, com foco na modelagem do domínio e da organização estrutural e navegacional.

Dentre os diversos métodos e modelos para a especificação de aplicações hipermídia encontrados atualmente, o método OOHDm tem se mostrado o método mais maduro, devido a sua ampla utilização pela comunidade hipermídia para projetos de diversas aplicações Web em diversos países.

OBS.: Na literatura técnica iremos encontrar o termo de desenvolvimento de uma aplicação hipermídia com o mesmo significado do desenvolvimento de software para a Web.

O desenvolvimento de uma aplicação hipermídia, ou seja, desenvolvimento de software para a Internet, compreende duas etapas principais:

1. A especificação da aplicação através de um método de projeto (por exemplo, o Object Oriented Hypermedia Design Method – OOHDMD) e;
2. A devida implementação em alguma linguagem (através de um ambiente de suporte).

OOHDM

O Object Oriented Hipermídia Design Method (OOHDM) permite o projeto de aplicações que seguem o paradigma de hipermídia, como as aplicações na Web.

Ele é uma evolução da experiência acumulada com o Hypermedia Design Model (HDM), trabalho realizado em 1990 por Daniel Schwabe, Franca Garzotto e Paolo Paolini, no Politecnico di Milano, que foi o primeiro modelo proposto para aplicações hipermídia.

No desenvolvimento de aplicações hipermídia, há a necessidade de se utilizar de um modelo de base, que possa nortear as etapas de construção do projeto e o uso da tecnologia relacionada. Schwabe propõe o uso do método OOHDM (Object-Oriented Hypermidia Design Method), que é composto por quatro atividades diferentes:

1. *Modelo conceitual*: modela a semântica do domínio da aplicação.
2. *Projeto da Navegação*: leva em consideração o perfil do usuário e a tarefa a ser executada, dando ênfase nos aspectos cognitivos.
3. *Projeto da interface abstrata*: modela objetos perceptíveis, implementa as metáforas escolhidas e descreve a interface para os objetos navegacionais.
4. *Implementação*: A implementação de uma aplicação hipermídia é complexa. Muitas questões técnicas e não-técnicas devem ser resolvidas. Uma vez que o ambiente de implementação tenha sido escolhido, o projeto deve ser mapeado para artefatos de implementação e todos os componentes hipermídia têm de ser instanciados.

Atividades	Produtos	Mecanismos	Interesses do Projeto
Modelagem Conceitual	Classes, subsistemas, relacionamentos, perspectivas de atributos.	Classificação, composição, generalização e especialização.	Modelagem da semântica e domínio de aplicação.
Projeto da Navegação	Nós, elos, objetos conceituais estruturas de acesso, contextos de navegação, transformações navegacionais.	Mapeamento entre e de navegação. Padrões de navegação para a descrição da estrutura geral da aplicação.	Leva em conta o perfil do usuário e a tarefa; ênfase em aspectos cognitivos e arquiteturais.
Projeto da Interface Abstrata	Objetos de interface abstrata, reações a eventos externos, transformações de interface.	Mapeamento entre objetos de navegação e objetos de interface.	Modelagem de objetos perceptíveis, implementa metáforas escolhidas.
Implementação	Aplicação execução.	Aqueles fornecidos pelo ambiente alvo.	Desempenho, completude.

Adaptado da dissertação de Gustavo Rossi sobre OOHDM

Benefícios da OOHDM

São utilizadas as mesmas primitivas de modelagem (objetos, classes), simplificando a transição de uma atividade para outra.

Ao longo do processo utilizamos agregação, classificação e generalização/especialização.

Pelo fato de os objetos serem artefatos reativos, podem ser construídas aplicações sofisticadas baseadas em hipermídia, definindo-se padrões de comportamento e de comunicação entre objetos.

Há poderosos formalismos, já existentes para especificar a estrutura, o comportamento e as relações dos objetos e podemos adaptá-los ao campo da hipermídia.

Aplicações projetadas e construídas em torno de objetos tendem a ser mais robustos e fáceis de modificar, tanto pelo uso de polimorfismo e herança como por composição.

Fornecemos primitivas de modelagem de alto nível na forma de padrões de projeto que podem ser utilizadas sem alterações, ou modificadas de acordo com as necessidades do projetista.

Construir novas aplicações reutilizando componentes existentes é altamente viável quando os componentes são descritos como objetos.



Estudo Complementar

Veja o WIKI construído sobre o OOHDM:

<http://www.tecweb.inf.puc-rio.br/oohdm/space/start>

<http://atlas.ucpel.tche.br/~lla/resumo.htm>





Atividades

1. Visite o site <http://www.porkworld.com.br/index.php?documento=109> e estude o case apresentado. O que você acha desse tipo de aplicação?
2. Assista a entrevista feita por Marcio Gomes, no programa Espaço Aberto da Globonews, através do site:

http://www.inf.puc-rio.br/~schwabe/DS_Espaco_Aberto_5-5-03.avi



UNIDADE 29

Planejamento Da Webapp

Objetivo: apresentar as técnicas de planejamento de projetos aplicados a Web.

A gestão de projetos WebE pequenos e de tamanho moderado (menos de 5 meses), requer uma abordagem ágil que não enfatize a gestão do projeto, mas que não elimine a necessidade de planejar. Princípios básicos de gestão de projeto ainda se aplicam, mas a abordagem global é mais simples e menos formal. Seguem os passos recomendados para esses tipos de projetos:

1. Entenda o escopo, as dimensões das modificações e as restrições de projeto: não se pode começar um projeto sem antes entender até onde ele irá, e o que ele aborda. Coleta de requisitos e comunicação com os clientes e usuários são antecedentes essenciais para o planejamento efetivo da WebApp.
2. Defina uma estratégia de projeto incremental: Como falamos anteriormente as aplicações Web evoluem com o tempo. Se essa evolução não for bem planejada em longo prazo, as chances de sucesso serão pequenas.
3. Realize análise de risco: os principais itens de preocupação as equipes de engenharia da Web são o risco de cronograma e risco de tecnologia. Preocupações quanto ao cumprimento dos prazos, e o domínio das tecnologias que serão adotadas pela equipe devem ser devidamente ponderadas.
4. Desenvolva uma rápida estimativa: o enfoque da estimativa do projeto deve ser basicamente em tópicos macroscópicos. Não deve existir a preocupação maior em levantar tópicos microscópicos, pois demandaria mais controles burocráticos.

5. Descreva o processo: selecione um conjunto de tarefas significativas que seja adequado para as características do problema, do produto, do projeto e do pessoal da equipe de engenharia.
6. Estabeleça um cronograma: deve-se adotar a estratégia de elaborar o cronograma com dois critérios bem distintos. Determinar as tarefas a serem realizadas em curto prazo com granularidade fina no cronograma. E com granularidade grossa para os incrementos adicionais que tiverem que ser entregues posteriormente.
7. Defina mecanismos para acompanhar o projeto: no desenvolvimento ágil, a entrega de um incremento operacional de software é a medida principal de progresso. Uma abordagem a ser adotada é determinar quantos casos de uso foram implementados e quantos dos casos de uso ainda restam ser implementados. Isso fornecerá uma indicação grosseira do relativo grau de “completeza” do incremento de projeto.
8. Estabeleça uma abordagem de modificação: como o prazo de desenvolvimento para um incremento é pequeno, é frequentemente possível adiar a introdução de uma modificação até o próximo incremento, reduzindo, assim, os efeitos de atraso associados a modificações que precisam ainda se implementadas “para ontem”.



Estudo Complementar

http://www.avellareduarte.com.br/projeto/planejamento/planejamento2/planejamento2_gestaoCriacao.htm





Atividades

Como você relacionaria o conteúdo apresentado nesta unidade com o PMBOK ?



UNIDADE 30

Ferramentas De Software Para Gestão De Projetos Web

Objetivo: Apresentar ferramentas de software para apoio a gestão de projetos na Web.

Com o objetivo de apoiar a equipe de engenharia da Web no planejamento, gestão, controle e rastreamento de projetos existem várias ferramentas na própria Internet que facilitam esse trabalho.

Ferramentas de gestão de projeto possibilitam a uma equipe de WebE estabelecer um conjunto de tarefas de trabalho, atribuir esforço e responsabilidade específica para cada tarefa, estabelecer as dependências de tarefas, definir um cronograma, e acompanhar e controlar as atividades no projeto.

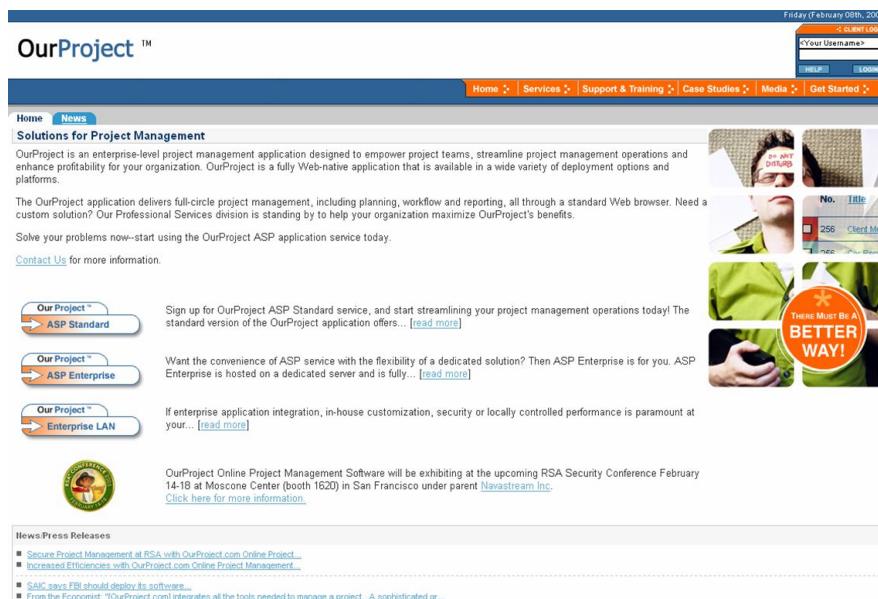
Ferramentas mais representativas

Business Engine, desenvolvida por Business Engine (www.businessengine.com), é um conjunto de ferramentas baseadas na Web que fornece plenas facilidades de gestão de projeto para projetos de WebE e de software convencional.



iTeamwork: desenvolvida por iTeamwork.com (www.iteamwork.com), é uma aplicação livre, on-line, baseada na Web, de gestão de equipe de projeto que você usa com o seu próprio navegador Web.

OurProject: desenvolvida por Our Project (www.ourproject.com) , é um conjunto de ferramentas de gestão de projetos que é aplicado a projetos da WebE e de software convencional.



The screenshot shows the homepage of OurProject. At the top, there's a navigation bar with links for Home, Services, Support & Training, Case Studies, Media, and Get Started. A login form is visible in the top right corner. Below the navigation, there's a section titled "Solutions for Project Management" featuring three service offerings: "Our Project™ ASP Standard", "Our Project™ ASP Enterprise", and "Our Project™ Enterprise LAN". Each service has a brief description and a "read more" link. To the right of these descriptions are several small images illustrating project management concepts like teamwork, client management, and better ways to work. At the bottom left, there's a news section titled "News/Press Releases" with a list of recent articles. On the far left, there's a sidebar with a logo for "RSA Security Conference" and a link to "Click here for more information".

Proj-Net, desenvolvida por Rational Concepts, Inc. (www.rationalconcepts.com), implementa um escritório virtual de projeto (virtual Project Office, VPO) para colaboração e comunicação.

A Productivity Company

Rational Concepts, Inc.
A Productivity Company

Helping Companies Better Manage Their Projects & Proposals ...through the power of the web













StartWright (www.startwright.com/project1.htm) desenvolveu um dos recursos mais abrangentes da Web tanto para WebE, quanto para software convencional de ferramentas e informação de gestão de projetos.



Estudo Complementar

<http://www.businessengine.com>

<http://www.iteamwork.com>

<http://www.ourproject.com>

<http://www.rationalconcepts.com>

<http://www.startwright.com/project1.htm>



Atividades

Nesta unidade exploramos o planejamento para aplicações Web, um dos itens desta unidade foi intitulado “Ferramentas mais representativas”. Visite os sites recomendados e explore a ferramentas sugeridas realizando um pequeno resumo do potencial de cada um desses softwares.



OBS.: pelo fato da Internet ser extremamente dinâmica, pode ocorrer que alguns sites recomendados estejam fora do ar, ou substituídos ao longo deste trabalho.



Atividades

Antes de dar início à sua Prova Online é fundamental que você acesse sua SALA DE AULA e faça a Atividade 3 no “link” ATIVIDADES.





Atividades

Atividade Dissertativa

Desenvolva uma pesquisa gerando um texto, de 2 a 3 folhas adicionando imagens, de uma das unidades da nossa apostila, de sua livre escolha, permitindo a expansão da temática selecionada.

Atenção: Qualquer bloco de texto igual ou existente na internet será devolvido para que o aluno o realize novamente.



GLOSSÁRIO

Caso haja dúvidas sobre algum termo ou sigla utilizada, consulte o link Glossário em sua sala de aula, no site da ESAB.

RREFERÊNCIAS

PRESSMAN, Roger. *Engenharia de Software*. São Paulo: McGraw-Hill Brasil, 2006

SOMMERVILLE, Ian. *Engenharia de Software*. São Paulo: Pearson Addison Wesley, 2007

REZENDE, Denis Alcides. *Engenharia de Software e Sistemas de Informação*. Rio de Janeiro: Brasport, 2005.