Testes de Software

Teste de Software é um processo que faz parte do #desenvolvimento de software, e tem como principal objetivo revelar falhas/bugs para que sejam corrigidas até que o produto final atinja a qualidade desejada / acordada.

Profissionais que trabalham com testes (denominados analistas de testes, técnicos de testes, homologador, ou simplesmente testes) estão habituados a realizar uma bateria de testes de diferentes naturezas e propósitos, envolvendo não apenas os testes funcionais da aplicação, mas diversas outras atividades como:

- Avaliação da especificação de requisitos,
- Avaliação de projeto técnico,
- Verificações em outros documentos,
- Testes de performance e capacidade,
- Avaliação de interface,
- Dentre outros.

Para ter uma ideia a respeito da diversidade e abrangência de atividades que fazem parte do processo de testes, é apresentada a seguir uma tabela com alguns tipos de testes comuns.

Tipos de Testes

Lista dos tipos de testes que podem ser feitos:

Tipo de Teste	**Descrição**
Teste de Unidade	Teste em um nível de componente ou classe. É o teste cujo objetivo é um “pedaço do código”.
Teste de Integração	Garante que um ou mais componentes combinados (ou unidades) funcionam. Podemos dizer que um teste de integração é composto por diversos testes de unidade.
Teste Operacional	Garante que a aplicação pode rodar muito tempo sem falhar.
Teste Positivo-negativo	Garante que a aplicação vai funcionar no “caminho feliz” de sua execução e vai funcionar no seu fluxo de exceção.
Teste de Regressão	Toda vez que algo for mudado, deve ser testada toda a aplicação novamente.
Teste de Caixa-preta	Testar todas as entradas e saídas desejadas. Não se está preocupado com o código, cada saída indesejada é visto como um erro.
Teste Caixa-branca	O objetivo é testar o código. Às vezes, existem partes do código que nunca foram testadas.
Teste Funcional	Testar as funcionalidades, requerimentos, regras de negócio presentes na documentação. Validar as funcionalidades descritas na documentação (pode acontecer de a documentação estar inválida)

Teste de Interface	Verifica se a navegabilidade e os objetivos da tela funcionam como especificados e se atendem da melhor forma ao usuário.
Teste de Performance	Verifica se o tempo de resposta é o desejado para o momento de utilização da aplicação.
Teste de Carga	Verifica o funcionamento da aplicação com a utilização de uma quantidade grande de usuários simultâneos.
Teste de Aceitação do usuário	Testa se a solução será bem vista pelo usuário. Ex: caso exista um botão pequeno demais para executar uma função, isso deve ser criticado em fase de testes. (aqui, cabem quesitos fora da interface, também).
Teste de Volume	Testar a quantidade de dados envolvidos (pode ser pouca, normal, grande, ou além de grande).
Testes de Stress	Testar a aplicação sem situações inesperadas. Testar caminhos, às vezes, antes não previstos no desenvolvimento/documentação.
Testes de Configuração	Testar se a aplicação funciona corretamente em diferentes ambientes de hardware ou de software.
Testes de Instalação	Testar se a instalação da aplicação foi OK.
Testes de Segurança	Testar a segurança da aplicação das mais diversas formas. Utilizar os diversos papéis, perfis, permissões, para navegar no sistema.

TDD

O custo de correção de um bug aumenta até mais de 100x quando corrigido nas fases finais de desenvolvimento, quando comparado ao custo de corrigir a mesma falha em fases iniciais. Por reconhecer este fato e por entender a relevância dos testes no processo de desenvolvimento, muitas empresas, profissionais e equipes optam por um método de desenvolvimento denominado TDD (Test Driven Development) – Desenvolvimento Orientado a Testes.

A ideia é que funcionalidades de testes sejam escritas antes mesmo do desenvolvimento das funcionalidades do sistema. Desta forma, as funcionalidades desenvolvidas precisarão “passar no teste” para serem validadas. Para saber mais, recomendamos: Programação Orientada a Testes.

JUnit

O JUnit é um framework open-source, que se assemelha ao raio de testes software java, criado por Erich Gamma e Kent Beck, com suporte à criação de testes automatizados na linguagem de programação Java.

Esse framework facilita a criação de código para a automação de testes com apresentação dos resultados. Com ele, pode ser verificado se cada método de uma classe funciona da forma esperada, exibindo possíveis erros ou falhas podendo ser utilizado tanto para a execução de baterias de testes como para extensão.

Com JUnit, o programador tem a possibilidade de usar esta ferramenta para criar um modelo padrão de testes, muitas vezes de forma automatizada.

O teste de unidade testa o menor dos componentes de um sistema de maneira isolada. Cada uma dessas unidades define um conjunto de estímulos (chamada de métodos), e de dados de entrada e

saída associados a cada estímulo. As entradas são parâmetros e as saídas são o valor de retorno, exceções ou o estado do objeto. Tipicamente um teste unitário executa um método individualmente e compara uma saída conhecida após o processamento da mesma. Por exemplo:

Assert.assertEquals(2, algumMetodo(1));

A expressão acima verifica se a saída de algumMetodo() é 2 quando esse método recebe o parâmetro 1. Normalmente o desenvolvedor já realiza testes semelhantes a esse pequeno exemplo, o que é chamado de testes unitários em linha. Assim sendo, o conceito chave de um teste de unidade é exercitar um código e qual o resultado esperado.

O JUnit permite a realização de testes de unidades, conhecidos como "caixa branca", facilitando assim a correção de métodos e objetos.

Algumas vantagens de se utilizar JUnit:

Permite a criação rápida de código de teste enquanto possibilita um aumento na qualidade do sistema sendo desenvolvido e testado;

Não é necessário escrever o próprio framework;

Amplamente utilizado pelos desenvolvedores da comunidade código-aberto, possuindo um grande número de exemplos;

Uma vez escritos, os testes são executados rapidamente sem que, para isso, seja interrompido o processo de desenvolvimento;

JUnit checa os resultados dos testes e fornece uma resposta imediata;

Pode-se criar uma hierarquia de testes que permitirá testar apenas uma parte do sistema ou todo ele;

Escrever testes com JUnit permite que o programador perca menos tempo depurando seu código;

JUnit é LIVRE.

A experiência adquirida com o JUnit tem sido importante na consolidação do Test Driven Development (desenvolvimento direcionado a testes). Além disso, ele foi adaptado a outras linguagens, tais como C♯ (NUnit), Python, Fortran, e C++.

Padrão De Projeto De Software

Em Engenharia de Software, um padrão de desenho (português europeu) ou padrão de projeto (português brasileiro) (do inglês design pattern) é uma solução geral para um problema que ocorre com frequência dentro de um determinado contexto no projeto de software. Um padrão de projeto não é um projeto finalizado que pode ser diretamente transformado em código fonte ou de máquina, ele é uma descrição ou modelo (template) de como resolver um problema que pode ser usado em muitas situações diferentes.

Padrões são melhores práticas formalizadas que o programador pode usar para resolver problemas comuns quando projetar uma aplicação ou sistema. Padrões de projeto orientados a objeto normalmente mostram relacionamentos e interações entre classes ou objetos, sem especificar as classes ou objetos da aplicação final que estão envolvidas. Padrões que implicam orientação a objetos ou estado mutável mais geral, não são tão aplicáveis em linguagens de programação funcional.

Padrões de projeto residem no domínio de módulos e interconexões. Em um nível mais alto há padrões arquiteturais que são maiores em escopo, usualmente descrevendo um padrão global seguido por um sistema inteiro.

As características obrigatórias que devem ser atendidas por um padrão de projeto são composto basicamente por 4 (quatro) elementos que são:

Nome do padrão;

Problema a ser resolvido;

Solução dada pelo padrão; e

Consequências.

Os padrões de projeto:

visam facilitar a reutilização de soluções de desenho - isto é, soluções na fase de projeto do software - e

estabelecem um vocabulário comum de desenho, facilitando comunicação, documentação e aprendizado dos sistemas de software.

História

O arquiteto Christopher Alexander, em seus livros (1977/1979) Notes on the Synthesis of Form, The Timeless Way of Building e A Pattern Language, estabelece que um padrão deve ter, idealmente, as seguintes características:

Encapsulamento: um padrão encapsula um problema ou solução bem definida. Ele deve ser independente, específico e formulado de maneira a ficar claro onde ele se aplica.

Generalidade: todo padrão deve permitir a construção de outras realizações a partir deste padrão.

Equilíbrio: quando um padrão é utilizado em uma aplicação, o equilíbrio dá a razão, relacionada com cada uma das restrições envolvidas, para cada passo do projeto. Uma análise racional que envolva uma abstração de dados empíricos, uma observação da aplicação de padrões em artefatos tradicionais, uma série convincente de exemplos e uma análise de soluções ruins ou fracassadas pode ser a forma de encontrar este equilíbrio.

Abstração: os padrões representam abstrações da experiência empírica ou do conhecimento cotidiano.

Abertura: um padrão deve permitir a sua extensão para níveis mais baixos de detalhe.

Combinatoriedade: os padrões são relacionados hierarquicamente. Padrões de alto nível podem ser compostos ou relacionados com padrões que endereçam problemas de nível mais baixo.

Além da definição das características de um padrão, Alexander definiu o formato que a descrição de um padrão deve ter. Ele estabeleceu que um padrão deve ser descrito em cinco partes:

Nome: uma descrição da solução, mais do que do problema ou do contexto;

Exemplo: uma ou mais figuras, diagramas ou descrições que ilustrem um protótipo de aplicação;

Contexto: a descrição das situações sob as quais o padrão se aplica;

Problema: uma descrição das forças e restrições envolvidos e como elas interagem;

Solução: relacionamentos estáticos e regras dinâmicas descrevendo como construir artefatos de acordo com o padrão, frequentemente citando variações e formas de ajustar a solução segundo as circunstâncias. Inclui referências a outras soluções e o relacionamento com outros padrões de nível mais baixo ou mais alto.

Os patterns de Alexander procuravam prover uma fonte de ideias provadas para indivíduos e comunidades para serem usadas em construções, mostrando assim o quanto belo, confortável e flexível os ambientes podem ser construídos.

Em 1987, a partir dos conceitos criados por Alexander, os programadores Kent Beck e Ward Cunningham propuseram os primeiros padrões de projeto para a área da ciência da computação. Em um trabalho para a conferência OOPSLA, eles apresentaram alguns padrões para a construção de aplicações comerciais em linguagem Smalltalk. Nos anos seguintes Beck, Cunningham e outros seguiram com o desenvolvimento destas ideias.

Porém, o movimento ao redor de padrões de projeto só ganhou popularidade em 1995 quando foi publicado o livro Design Patterns: Elements of Reusable Object-Oriented Software. Os autores desse livro, Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, são conhecidos como a "Gangue dos Quatro" (Gang of Four) ou simplesmente "GoF".

Posteriormente, vários outros livros do estilo foram publicados, merecendo destaque Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, que introduziu um conjunto de padrões conhecidos como GRASP (General Responsibility Assignment Software Patterns).

Características De Um Padrão De Projeto

Embora um padrão seja a descrição de um problema, de uma solução genérica e sua justificativa, isso não significa que qualquer solução conhecida para um problema possa constituir um padrão, pois existem características obrigatórias que devem ser atendidas pelos padrões:

1. Devem possuir um NOME, que descreva o problema, as soluções e conseqüências. Um nome permite definir o vocabulário a ser utilizado pelos projetistas e desenvolvedores em um nível mais alto de abstração.

2. Todo padrão deve relatar de maneira clara a qual (is) PROBLEMA(s) ele deve ser aplicado, ou seja, quais são os problemas que quando inserido em um determinado contexto o padrão conseguirá resolve-lo. Alguns podendo exigir pré-condições.

3. Solução descreve os elementos que compõem o projeto, seus relacionamentos, responsabilidades e colaborações. Um padrão deve ser uma SOLUÇÃO concreta, ele deve ser exprimido em forma de gabarito (algoritmo) que, no entanto pode ser aplicado de maneiras diferentes.

4. Todo padrão deve relatar quais são as suas CONSEQUÊNCIAS para que possa ser analisada a solução alternativa de projetos e para a compreensão dos benefícios da aplicação do projeto.

Não pode ser considerado um padrão de projeto trecho de códigos específicos, mesmo que para o seu criador ele reflita um padrão, que soluciona um determinado problema, porque os padrões devem estar a um nível maior de abstração e não limitado a recursos de programação. Um padrão de projeto nomeia, abstrai e identifica os aspectos chaves de uma estrutura de projeto comum para torna-la útil para a criação de um projeto orientado a objetos reutilizável.

Padrões GoF ('Gang of Four')

De acordo com o livro: "Padrões de Projeto: soluções reutilizáveis de software orientado a objetos", os padrões "GoF" são divididos em 23 tipos. Em função dessa grande quantidade de padrões, foi necessário classificá-los de acordo com as suas finalidades.

São 3 As Classificações/Famílias:

Padrões de criação: Os padrões de criação são aqueles que abstraem e ou adiam o processo criação dos objetos. Eles ajudam a tornar um sistema independente de como seus objetos são criados, compostos e representados. Um padrão de criação de classe usa a herança para variar a classe que é instanciada, enquanto que um padrão de criação de objeto delegará a instanciação para outro objeto. Os padrões de criação tornam-se importantes à medida que os sistemas evoluem no sentido de dependerem mais da composição de objetos do que a herança de classes.

O desenvolvimento baseado na composição de objetos possibilita que os objetos sejam compostos sem a necessidade de expor o seu interior como acontece na herança de classe, o que possibilita a definição do comportamento dinamicamente e a ênfase desloca-se da codificação de maneira rígida de um conjunto fixo de comportamentos, para a definição de um conjunto menor de comportamentos que podem ser compostos em qualquer número para definir comportamentos mais complexos. Há dois temas recorrentes nesses padrões. Primeiro todos encapsulam conhecimento sobre quais classes concretas são usadas pelo sistema. Segundo ocultam o modo como essas classes são criadas e montadas. Tudo que o sistema sabe no geral sobre os objetos é que suas classes são definidas por classes abstratas. Conseqüentemente, os padrões de criação dão muita flexibilidade no que é criado, quem cria,

como e quando é criado. Eles permitem configurar um sistema com objetos “produto” que variam amplamente em estrutura e funcionalidade. A configuração pode ser estática (isto é, especificada em tempo de compilação) ou dinâmica (em tempo de execução).

Padrões estruturais: Os padrões estruturais se preocupam com a forma como classes e objetos são compostos para formar estruturas maiores. Os de classes utilizam a herança para compor interfaces ou implementações, e os de objeto ao invés de compor interfaces ou implementações, eles descrevem maneiras de compor objetos para obter novas funcionalidades. A flexibilidade obtida pela composição de objetos provém da capacidade de mudar a composição em tempo de execução o que não é possível com a composição estática (herança de classes).

Padrões comportamentais: Os padrões de comportamento se concentram nos algoritmos e atribuições de responsabilidades entre os objetos. Eles não descrevem apenas padrões de objetos ou de classes, mas também os padrões de comunicação entre os objetos. Os padrões comportamentais de classes utilizam a herança para distribuir o comportamento entre classes, e os padrões de comportamento de objeto utilizam a composição de objetos em contrapartida a herança. Alguns descrevem como grupos de objetos cooperam para a execução de uma tarefa que não poderia ser executada por um objeto sozinho.

Padrões

Os padrões GRASP, sigla para General Responsibility Assignment Software Patterns (or Principles), consistem de um conjunto de práticas para atribuição de responsabilidades a classes e objetos em projetos orientados a objeto.

Os padrões GRASP (General Responsibility Assignment Software Patterns), são responsáveis pela descrição de princípios de fundamental importância para a atribuição de responsabilidades em projetos orientados a objetos, oferecendo um melhor desempenho do código, e trabalhando acerca de solucionar problemas, garantindo melhor interface do projeto.

Sendo assim, é importante sabermos que a qualidade do projeto orientado a objetos está diretamente relacionada com a distribuição dessas obrigações, que promovem a não sobrecarga de objetos já que ocorre nesse processo a delegação de atividades, ou seja, cada objeto terá uma função específica, de modo que, o que ele não souber fazer será repassado para o objeto que está mais preparado para fazer.

Todos esses padrões servem para a resolução de problemas comuns e bastante típicos de desenvolvimento de software orientado a objeto. Portanto, tais técnicas apenas documentam e normatizam as práticas já consolidadas, testadas e conhecidas no mercado.

Os padrões GRASP estão mais como uma ferramenta mental ou uma filosofia de design, mas que ainda assim são úteis para o aprendizado e desenvolvimento de um bom design de software. Note que alguns padrões GoF implementam soluções correspondentes com padrões GRASP.

Tipos de Padrões Grasp

Dentro do GRASP podemos encontrar vários padrões relacionados aqueles de caráter básicos e avançados.

Padrões Básicos

Information Expert;

Creator;

High Cohesion;

Low Coupling;

Controller.

Padrões Avançados:

Polymorphism;

Pure Fabrication;

Indirection;

Protected Variations.

Desenvolvimento Baseado Em Componentes

A importância do desenvolvimento com alta produtividade e redução de custos tem pressionado as empresas de tecnologia a buscar metodologias que se adéquem melhor às suas necessidades. O desenvolvimento baseado em componentes pode ajudar a conquistar esses requisitos, pois combina a forte reutilização com a robustez de sistemas já testados.

Desenvolvimento baseado em componentes:

Este artigo apresentará os conceitos do desenvolvimento baseado em componentes. Para isso, analisaremos os tipos de componentes de mercado e suas características, e por fim, apresentaremos o desenvolvimento de um componente de validação de documentos que adota os padrões Factory, Singleton e Strategy.

Atualmente as empresas de software procuram meios de desenvolvimento para alcançar uma alta produtividade com alto retorno sobre o investimento. Neste cenário, a margem para erros em prazos e custos deve ser mínima e o produto entregue deve contemplar praticamente todo o escopo proposto pelo solicitante (o cliente).

Esta busca pela alta produtividade passa por muitos itens. Dentre eles a contratação de profissionais experientes, treinamentos, utilização de ferramentas de última geração, adequação de processos, entre outros.

Com base nisso, esse artigo aborda uma estratégia de melhoria de produtividade focada na reutilização de artefatos já construídos, ou seja, uma abordagem arquitetural. Esta reutilização se dá pela construção de sistemas através de componentes.

O que é componentização?

O desenvolvimento baseado em componentes permite que o sistema final seja tratado como vários “minissistemas”, diminuindo sua complexidade e permitindo que cada componente empregado seja focado em apenas uma funcionalidade ou um conjunto de funcionalidades semelhantes. Deste modo, estas funcionalidades podem ser reutilizadas em diversas aplicações através do acesso ao componente.

Dito isso, verificamos que componentização de software é uma abordagem arquitetural baseada na divisão de sistemas de software em unidades menores, denominadas componentes.

Neste cenário, para que possamos garantir o baixo acoplamento entre o cliente e o componente, a implementação deste deve respeitar alguns princípios de desenvolvimento. E muitos desses princípios são herdados do desenvolvimento orientado a objetos (OO), tais como:

• Componentes são orientados a interfaces, ou seja, expõem seus serviços para que o cliente realize o acesso, mas sem expor a implementação;

• Componentes podem ter dependências com outros componentes. Neste caso um componente realiza a chamada para a interface de outro componente.

Benefícios Da Componentização

O desenvolvimento de aplicações baseado em componentes nos traz uma série de benefícios, dentre os quais podemos destacar:

• Produtividade: Pode-se economizar tempo de desenvolvimento, dependendo do portfólio de componentes já prontos;

• Robustez: Maior qualidade no produto final que utiliza componentes, pois os mesmos já foram largamente testados em um projeto dedicado à construção dos mesmos;

• Padrão de desenvolvimento: Equipe orientada a desenvolvimento nos moldes da componentização.

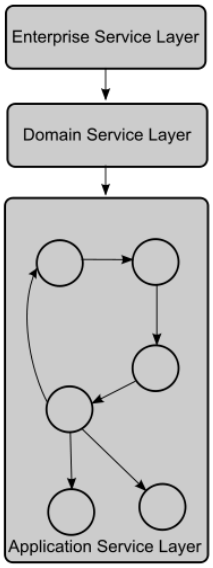
Desafios para a adoção da componentização

Mudanças nem sempre são bem-vindas, ainda mais quando tratamos de mudanças relacionadas a desenvolvimento de software. Neste contexto, muitas pessoas entendem que: “se está funcionando, para quê mexer?”.

SOA (Arquitetura Orientada A Serviços)

Mas o que venha ser este termo? O que é Arquitetura de serviços? É muito simples arquitetura orientada, corresponde a uma metodologia para desenvolvimento de software, serviços, representa todos ativos de softwares da empresa. Também podemos descrever neste caso serviços. Como sendo um componente, uma parte de desenvolvimento de um software onde ao fazer a junção de todos os ”módulos”, teremos um software completo para aquela determinada função para que foi desenhado, produto final do escopo do projeto onde foi determinado a criação de um serviço.

Podendo passar por vários departamentos, abrangendo uniformemente o tratamento de regras de negócio para cada necessidade.



Uma diferença entre Web Services e SOA, pode-se resumir que o SOA é voltado para dentro da empresa, fornecendo informações ou recursos para toda organização, um pouco mais abrangente podemos falar dos Web Services, que são sites abertos para o mundo todo fazer cosumo do serviço disponível.

Uma grande vantagem do SOA é sem dúvida a reutilização do software, que conseqüentemente gera aumentos de produtividade, melhor alinhamento de negócio, trazendo melhorias para toda corporação e facilidade para a gerencia da tecnologia da informação, onde esta pode despender mais tempo em melhorias continuas e automatizar processos, tornando assim a qualidade a disponibilidade um fator de diferencial para a informação que trafega por toda a corporação.

Um conceito usado pelo SOA, é a centralização, pois onde existe várias pessoas tomando decisões sem se interagir, fatalmente haverá colisão de informações. O SOA trabalha de forma centralizada, sendo sempre um líder tomando partido e conhecimento da situação, levando soluções e propostas para o conselho, onde escolhe-se a melhor e mais propicia decisão para a organização.

Além da perspectiva estritamente técnica, a arquitetura orientada a serviços também se relaciona com determinadas políticas e conjuntos de "boas práticas" que pretendem criar um processo para facilitar a

tarefa de encontrar, definir e gerenciar os serviços disponibilizados. A arquitetura orientada a serviços também se insere em um processo de reorganização dos departamentos de tecnologia da informação das organizações, permitindo um melhor relacionamento entre as áreas que dão suporte tecnológico à empresa e as áreas responsáveis pelo negócio propriamente dito, graças a maior agilidade na implementação de novos serviços e reutilização dos ativos existentes.

Requisitos

A fim de utilizar eficientemente uma SOA, deve-se atender aos seguintes requisitos: A interoperabilidade entre diferentes sistemas e linguagens de programação fornece a base para a integração entre aplicações em diferentes plataformas, através de um protocolo de comunicação. Um exemplo dessa comunicação depende do conceito de mensagens. Usando mensagens, através de canais de mensagens definidos, diminui a complexidade da aplicação final, permitindo que o desenvolvedor do aplicativo se concentre na funcionalidade do aplicativo de verdade, em vez das necessidades intrincadas de um protocolo de comunicação. O desejo é o de criar um conjunto de recursos a ser compartilhado, bem como estabelecer e manter o fluxo de dados para um sistema de banco de dados compartilhado. Isto permite que novas funcionalidades desenvolvidas para um formato de negócio de referência comum para cada elemento de dados.

Serviço

Um serviço, do ponto de vista da arquitetura SOA, é uma função de um sistema computacional que é disponibilizado para outro sistema. Um serviço deve funcionar de forma independente do estado de outros serviços, exceto nos casos de serviços compostos (composite services), e deve possuir uma interface bem definida. Normalmente, a comunicação entre o sistema cliente e aquele que disponibiliza o serviço é realizada através de web services.

Desenvolvimento de Software Baseado em Componentes / Orientado a Serviços (SOA)

DBC:

Investigar o uso de desenvolvimento baseado em componentes (DBC) em diversos contextos: linha de produtos de software, aplicações em outras áreas de conhecimento, como engenharia elétrica, teste de componentes, entre outros. o DBC diferencia-se de outras abordagens de desenvolvimento pela separação envolvendo a especificação do componente e a sua implementação, e na divisão da especificação funcional dos componentes em interfaces.

SOA:

Investigar o uso de arquiteturas orientadas a serviços (SOA) no contexto de linha de produtos de software e sistemas embarcados críticos. Investigar o teste estrutural de serviços e o desenvolvimento de ferramentas como serviços. SOA é um estilo arquitetural para a construção de aplicações de software que utilizam serviços disponíveis em uma rede. Um serviço é a implementação de uma funcionalidade de negócios bem definida, utilizada por clientes em diferentes aplicações e processos de negócios

Princípios Do Design De Interfaces

1. Clareza É O Trabalho Nº 1

Clareza é o primeiro e mais importante objetivo de qualquer interface. Para uma interface ser eficaz e eficiente, o usuário deve ser capaz de reconhecer a sua utilidade e entender como ela pode ajudá-lo, prever o que vai acontecer durante o uso, e conseguir interagir com sucesso. Não há motivos para mistérios nas interfaces. Clareza inspira confiança e cativa o uso.

2. Interfaces Existem Para Permitir A Interação

Interfaces existem para permitir a interação entre as pessoas e nosso mundo. Elas podem ajudar a esclarecer, iluminar, mostrar as relações, nos unir, nos separar, gerenciar nossas expectativas, e nos dar acesso a diferentes serviços. O designer de interfaces não é um artista. Interfaces não são obras de arte. Uma interface existe para realizar um trabalho e sua eficácia pode ser medida. No entanto, as interfaces não devem possuir apenas função. As melhores interfaces são capazes de inspirar, mistificar e intensificar a nossa relação com o mundo.

3. Prenda A Atenção A Todo Custo

Vivemos em um mundo de constantes interrupções. É difícil até mesmo ler um livro sem que nada nos distraia e tire a nossa atenção por um minuto. “Atenção” é algo muito precioso. Não coloque em sua interface elementos desnecessários que podem distrair o seu usuário. Lembre-se em primeiro lugar do porquê dessa interface existir. Quando o uso é o principal objetivo, a atenção torna-se um pré-requisito e você deve conservá-la a todo custo.

4. Mantenha O Usuário No Controle

As pessoas se sentem mais confortáveis quando estão no controle de suas ações e de seu ambiente. Interfaces mal projetadas tiram esse conforto do usuário, forçando as pessoas a interações não planejadas e resultados inesperados. Mantenha o usuário no controle, fornecendo sempre o status do sistema, boas mensagens de erro, ajuda e documentação.

5. Manipulação Direta É Melhor

Sempre é melhor quando somos capazes de manipular diretamente os objetos, sem o intermédio de nenhuma ferramenta. Mas com objetos cada vez mais tecnológicos e informacionais, essa manipulação direta nem sempre é possível, e por isso criamos as interfaces: para ajudar as pessoas a interagir com os objetos. No entanto, muitas vezes há um certo exagero na criação de botões e outros elementos de interface desnecessários para mediar a interação do usuário com o objeto, o que acaba burocratizando a interação. O ideal é possibilitarmos a interação direta sempre que possível, com os elementos de interface servindo de suporte apenas quando necessário. O ideal é projetarmos interfaces mais compactas, que dão ao usuário uma sensação de manipulação (mais) direta com o objeto.

6. Um Objetivo Principal Por Tela

Cada tela que nós projetamos deve apoiar um único objetivo principal, uma única ação de real valor para a pessoa usar. Isto faz com que a curva de aprendizado seja menor, e a facilidade de uso maior. Telas que fornecem duas ou mais ações primárias tornam-se muito confusas. Como um artigo escrito deve ter um assunto único e forte, cada tela que você projetar deve apoiar uma ação única e forte, que será a sua razão de ser.

7. Mantenha Ações Secundárias Em Segundo Plano

Telas com uma única ação primária podem ter várias ações secundárias, e é importante que elas sejam realmente apresentadas como secundárias. A razão pela qual um artigo existe não é para que as pessoas possam compartilhá-la no Twitter, mas sim para que elas possam ler e compreender. Mantenha as ações secundárias em segundo plano, seja com um visual mais leve ou as mostrando somente quando a ação primária for concluída.

8. Fornecer Sempre Um “Próximo Passo”

Em nosso site ou aplicativo, dificilmente vamos querer que uma determinada ação do usuário seja a sua última. Por isso, é importante sempre projetar um “próximo passo” para cada interação que uma pessoa tem com a nossa interface. Devemos não só fornecer esse novo passo, como também mostrar o que acontecerá nessa próxima interação. Não abandone o usuário só porque ele já efetuou a interação que você queria, dê a ele um “próximo passo” natural e relevante, que o ajude ainda mais a alcançar seus objetivos.

9. O Aspecto Segue O Comportamento (Ou A Forma Segue A Função)

Nós, seres humanos, ficamos mais confortáveis quando as coisas se comportam da maneira que esperamos, sejam outras pessoas, animais, objetos ou softwares. Quando algo ou alguém se comporta de forma consistente com as nossas expectativas, sentimos que temos um bom relacionamento com ele. Por isso, os elementos projetados para uma interface devem aparentar o seu comportamento. Na prática, isso significa que o usuário deve ser capaz de prever como um elemento na interface vai se comportar, apenas olhando para ele. Se parece com um botão, deve funcionar como um botão.

10. Questões De Consistência

Seguindo ainda no princípio anterior, um elemento da interface não deve parecer com outro a não ser que tenham a mesma função ou comportamento. O oposto disso também é importante, ou seja, elementos que tenham a mesma função ou comportamento não devem ter aparências diferentes pois é para elementos como a aparecer consistente. Devemos nos ater a isso para manter a consistência dos elementos de interface.

Programação Segura

Muitos programadores, pensam apenas em ataques que podem ser feitos apenas na máquina que está rodando o programa, e acabam não se preocupando com a segurança de seu próprio programa. Existem muitas vulnerabilidades conhecidas em C que podem ser evitadas introduzindo boas práticas para se fazer uma programação segura.

Inúmeras formas podem ser adotadas para se proteger o programa. A inicialização segura de um programa é muito importante, pois muitos ataques podem ser feitos simplesmente na hora da inicialização de seu programa. Algumas vulnerabilidades na inicialização do seu programa e soluções para as mesmas serão descritas neste artigo.

Outro ponto importante para segurança do seu programa é o controle de acesso em pastas e/ou arquivos utilizados pelo programa, que também será abordado no artigo. Um ataque pode ser feito simplesmente porque o programador se descuidou na hora de fazer validações em um campo de entrada de texto. Algumas boas práticas sobre a segurança do programa, serão demonstradas neste artigo.

Inicializando Seu Programa Com Segurança

Descritor De Arquivo

Uma potencial vulnerabilidade em seu código C, pode ser um deny of service, gerado por arquivos desnecessários preenchendo o descritor de arquivo até seu limite. Quando um processo é iniciado, ele herda todos os descritores de arquivo de seu processo pai, como no Unix o descritor de arquivo tem um tamanho fixo, esta herança de descritores abertos pode preencher todo o tamanho da tabela do descritor com arquivos lixo causando uma negação de serviço em seu programa.

Para se defender do problema de negação de serviço causado pelo descritor de arquivo cheio, devemos sempre que iniciar o programa fechar todos os descritores que não são os descritores default (stdin, stdout, stderr) e se certificar que os defaults estão abertos.

Matt Messier e John Viega (2008, sessão 1.5) dizem que no Windows, não tem como verificar quais arquivos estão abertos, mas o mesmo problema não está presente no Windows.

Mantendo Os Dados Do Programa Seguro Depois De Uma Falha De Sistema

Imagine se um programa que necessita armazenar dados de números de cartão de crédito no disco para utiliza-lo? Se por um acaso o programa falhe ou tranque, os dados não podem em hipótese alguma estar disponíveis depois da falha, pois pessoas com más intenções podem examinar os dados e usa-los.

Na maioria dos sistema Unix, quando o programa termina devido a uma falha de sistema, o sistema terá um despejo de memória. O problema deste despejo de memória, é que ele pode conter dados confidenciais e que podem ser usados por atackers.

Nos sistemas unix, podemos limitar o despejo de memória usando a função setrlimit( ) para setar o RLIMIT_CORE para zero. Sem esta limitação, um atacker, pode descobrir uma maneira de quebrar o sistema e causar um despejo de memória preenchendo todo o espaço em disco. Setando o valor de RLIMIT_CORE para 0, previne o sistema de preencher despejos de memória e ao invés disto ele apenas fecha o programa.

Revisão De Código

Apesar de não ser uma das práticas mais populares em times ágeis (ela não é nem citada no questionário de 2014 da VersionOne, por exemplo), a revisão de código vem ganhando seu espaço nas equipes de desenvolvimento de software. Afinal, as ditas vantagens da prática são várias: a diminuição na

quantidade de defeitos e aumento da qualidade do código produzido, facilitando sua manutenção e evolução.

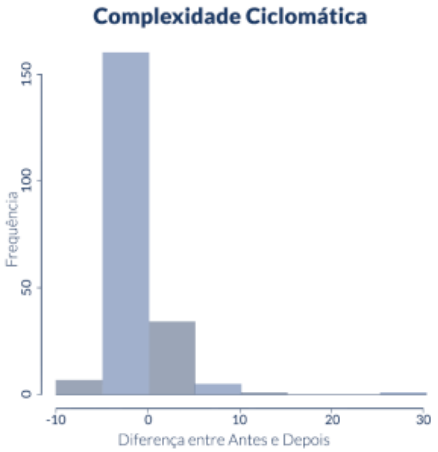
O próprio nome “revisão de código” deixa claro o objetivo da prática. A ideia é que o código escrito por um desenvolvedor, antes de ser promovido ao ambiente de produção, seja revisado por outro membro da equipe. Essas revisões podem ser feitas de diversas maneiras, como, por exemplo, programação pareada, ou mesmo navegando pelos artefatos modificados. Comumente, o revisor anota todos os problemas encontrados e devolve ao autor original daquele código. O autor então avalia os comentários recebidos e eventualmente os propaga para o código-fonte.

Na Caelum, em particular, as equipes fazem uso do próprio Github, o serviço de hospedagem de código, para fazer suas revisões. Após a finalização de uma história, o desenvolvedor anota a lista de commits e artefatos modificados. Um outro membro do time, em posse dessa lista, inspeciona todo o código modificado; todo e qualquer problema encontrado pelo revisor é salvo em comentários no próprio Github. O desenvolvedor original, ao receber os e-mails enviados automaticamente pela ferramenta, discute e tira dúvidas com seu revisor, e eventualmente altera o código para satisfazer a sugestão.

Interessantemente, imaginamos que um código que tenha passado revisão, tenha uma qualidade melhor, seja menos complexo, mais simples e fácil de ser entendido e mantido. Mas será que isso acontece sempre e de maneira natural? Resolvemos fazer um pequeno estudo dentro das equipes da Caelum para entender quais eram os benefícios que a revisão de código nos trazia.

O primeiro passo foi tentar medir o aumento da qualidade de código. Decidimos automatizar isso olhando para nossos repositórios no Github. Dado que todas as nossas revisões acontecem no Github, decidimos pegar todas as classes que foram revisadas, e a versão dela antes e depois da revisão. Por exemplo, a classe Aluno. Pegamos a quantidade de linhas de código antes e depois da revisão. A ideia é que esse número diminua. Fizemos isso com várias métricas de código diferentes, como complexidade ciclomática, linhas de código, falta de coesão dos métodos e acoplamento eferente.

E, para nossa surpresa, percebemos que os números não mudaram tanto assim após uma revisão. Ou seja, a qualidade do código, sob o ponto de vista dessas métricas, continuou a mesma na maioria absoluta das vezes. No desenho abaixo, você pode ver que a complexidade do código não mudou na maioria das vezes. Interessante, não?



Na sequência, optamos por mandar questionários para nossos desenvolvedores, perguntando a opinião deles sobre esses resultados, que eram bem contraditórios.

E a resposta deles foi bastante interessante! Os resultados mostraram que nossas equipes se beneficiam da inerente disseminação de conhecimento que acontece ao ler o código produzido por outro desenvolvedor. A percepção de redução de defeitos também acontece, já que o revisor comumente encontra problemas no código produzido pelo desenvolvedor original. Os desenvolvedores também têm a percepção de melhoria da qualidade interna, embora isso não tenha se refletido nas métricas de código coletadas.

Ou seja, a revisão de código não serve só para melhorar a qualidade do código, mas também para disseminar conhecimento entre os membros da equipe, e ajudar a encontrar bugs que o primeiro desenvolvedor deixou passar.

Verificando Vulnerabilidades Em Aplicações Web

Objetivos da análise de vulnerabilidade:

Essa documentação visa descrever os tipos de vulnerabilidades, como realizar testes em sua aplicação para protegê-la e também quais impactos uma invasão pode causar em seu site.

As vulnerabilidades em programação serão analisadas através da ferramente do Google Code no caso o nome é WEBSECURIFY, sendo assim a mesma realiza pentesting http://en.wikipedia.org/wiki/ Penetration_test um breve resumo sobre pentesting, no caso é um (teste de intrusão na aplicação) onde irá reportar um relatório com as vulnerabilidades.

Obs.: Ressaltamos que os passos de execução da ferramenta são apenas sugestões em como proceder, para maiores informações sobre a ferramenta e os logs 'informações' que ela gera para ser analisado em sua aplicação, por favor consulte o site http://code.google.com/p/websecurify/ ou desenvolvedor de sites com experiência em segurança da informação na web. A utilização dessa documentação requer experiência em programação em alguma linguagem e conhecimento específico em segurança da informação, neste caso a documentação é 'indicada a programadores' onde terá total flexibilidade para interpretar os resultados filtrados.

Severidade de vulnerabilidades em aplicações:

Critico.jpg Nível Crítico Quando sua aplicação reporta esse tipo de vulnerabilidade é viável que averigue a arquitetura de sua aplicação em caráter emergencial, pois a vulnerabilidade possivelmente pode comprometer riscos a operação de sua aplicação e os dados sigilosos, como exemplo do seus (banco de dados) da aplicação ou seus arquivos do site.

Vulnerabilidades:

SQL Injection Através dessa vulnerabilidade dependendo do nível do usuário que conecta no banco de dados, pode comprometer isoladamente os dados de seu banco e até mesmos edições dos registros.

Atenção: Para evitar esse tipo de vulnerabilidade é importante analisar o tratamento de consultas e inserção de dados no seu Banco de Dados na programação.

PHP Injection Nessa vulnerabilidade a exploração de injeção de códigos consegue processar arquivos através de intrusão LFI (Local File Inclusion) e executar arquivos remotos ou enviar arquivos através da intrusão RFI (Remote File Inclusion), assim comprometendo toda sua área de hospedagem e possibilitando as seguintes ameaças:

- Acesso a dados de sua aplicação, tais como acesso a String de acesso ao banco de dados, assim sendo possível explorar essa vulnerabilidade para alteração de seus dados.

- Inserção de vírus em seus arquivos, assim impactando na imagem de seu site e disseminando vírus aos visitantes.

- Envio de SPAM através de sua área de hospedagem, através de scripts de e-mail em massa, assim comprometendo seu ambiente a desativação por disseminar conteúdo impróprio via e-mail.

Obs.: Além da linguagem PHP existem outras que podem prover do ataque também, seja qual ambiente estiver operando.

Nível Alto Geralmente esse tipo de alerta é viável que verifique os pontos onde será reportado e o tipo de Vulnerabilidade.

Nível Baixo Geralmente não existe vulnerabilidade quando reportado, mas existem informações como exemplo: versão do PHP ou Apache que está sendo utilizada, assim possibilitando ataques a vulnerabilidades nas versões usadas em seu site.