**Teste de aplicações Web**

**Resumo**

Empresas de desenvolvimento de software trabalham sob pressão para entregar sistemas de software, e atender exigências de clientes, temerosos de que vão perder oportunidades de negócios. Como consequência atividades técnicas que frequentemente ocorrem mais tarde no processo de engenharia do software, tais como atividades de teste de software, recebem algumas vezes pouco prazo. Atividades de testes devem ser planejadas antecipadamente com o objetivo de se maximizar a eficácia de detecção de defeitos. Uma abordagem importante, para controlar a qualidade do produto, é se utilizar de uma arquitetura que aumente a testabilidade do software. Muitas aplicações web se utilizam de uma arquitetura em camadas que consiste de uma camada de apresentação, camada de negócios e camada de dados. Em uma arquitetura que maximiza a testabilidade existe uma camada de teste dedicada para cada uma das camadas da arquitetura. Uma analise de priorização dos requisitos e o nível de cobertura de teste por camada pode levar a uma estratégia de teste de qualidade com baixo custo.

1. **Introdução**

A plataforma Java 2 Enterprise Edition (J2EE) define um padrão para o desenvolvimento de aplicações multicamadas. Nesta arquitetura, uma camada cliente fornece a interface de usuário, uma camada intermediaria fornece serviços e a lógica de negócio para as aplicações e uma camada de sistemas de informação, fornece o gerenciamento de dados e a integração com sistemas legados. A camada cliente (interface de usuário) não contém lógica do negócio, concentrando-se somente na apresentação dos dados. Na camada cliente normalmente são utilizadas as tecnologias de Servlets e JavaServer Pages(Johnson, 2002).

1. **Projeto arquitetural**

Um software construído para sistemas baseados em computador exibe um ou vários estilos arquiteturais. Pressman (2006) relaciona alguns estilos de organização de sistemas comumente usados para software:

Arquitetura centrada nos dados– Os subsistemas que constituem um sistema devem trocar informações de modo que possam trabalhar juntos eficientemente. Existem duas maneiras fundamentais pelas quais isso pode ser feito:

1.Todos os dados compartilhados são mantidos em um banco de dados - Um depósito de dados (por exemplo, um arquivo ou banco de dados) fica no centro dessa arquitetura e dá acesso frequentemente a outros componentes que atualizam, adicionam, retiram ou modificam de outra forma os dados contidos no deposito. A figura 1 ilustra um estilo típico centrado nos dados.

2.Cada subsistema mantem seu próprio banco de dados – os dados são trocados com outros subsistemas por passagens de mensagens entre eles (o componente quadro-negro serve para coordenar a transferência de informações entre clientes).

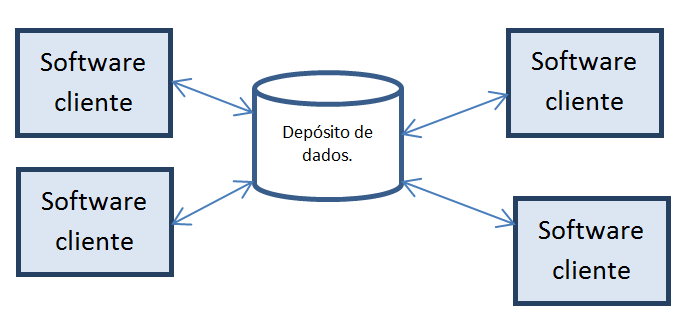
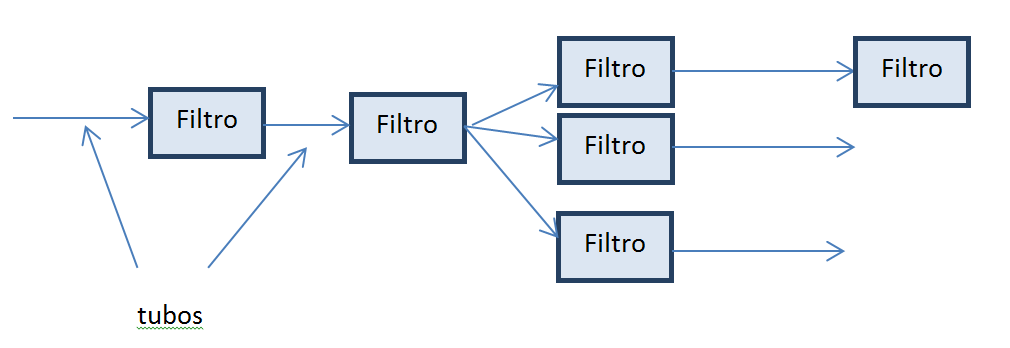


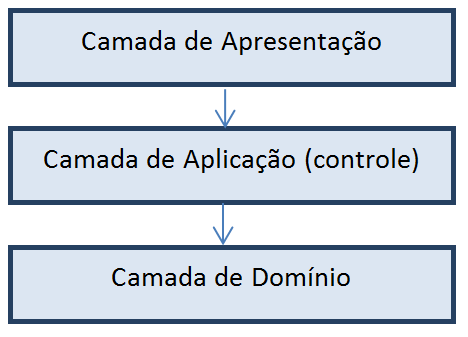
Figura - arquitetura centrada nos dados

Arquitetura de fluxo de dados (pipelining orientado a funções) - Essa arquitetura é aplicada quando dados de entrada devem ser transformados, por meio de uma série de componentes computacionais em dados de saída. Uma estrutura tubo e filtro (figura 2) tem um conjunto de componentes chamados filtros, conectados por tubos que transmitem dados de um conjunto de componentes para o próximo filtro. Cada filtro trabalha independentemente dos componentes de entrada/saída, ou seja, o filtro não exige conhecimento do trabalho dos filtros vizinhos.



Arquitetura de chamada e retorno – permite ao projetista de software uma estrutura de programa relativamente fácil de modificar e ampliar

Arquitetura em camadas – O modelo em camadas de uma arquitetura organiza um sistema em camadas, cada uma das quais fornece um conjunto de serviços (figura 2). A abordagem em camadas apoia o desenvolvimento incremental de sistemas. À medida que uma camada é desenvolvida alguns serviços fornecidos por essa camada podem ser disponibilizados para os usuários. Essa arquitetura é modificável e portável. Desde que sua interface permaneça inalterada, uma camada poderá ser substituída por outra equivalente. Mesmo quando a as interfaces da camada são alteradas, ou novos recursos são adicionados a uma camada, somente a camada adjacente é afetada. Uma desvantagem da abordagem em camadas esta relacionada a necessidade de um serviço de nível superior precisar passar por varias camadas para solicitar um serviço de uma camada inferior.



1. **Testabilidade da arquitetura**

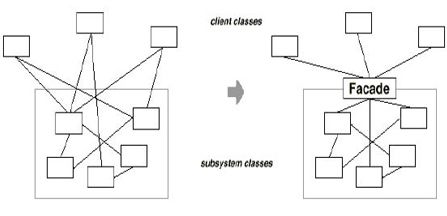
Eriksson (1997) propõe uma arquitetura de software onde o sistema é dividido em camada de apresentação, camada de aplicação e camada de domínio. Em uma arquitetura que maximiza a testabilidade existe uma camada de teste dedicada para cada uma das camadas da arquitetura. Uma analise de priorização dos requisitos e o nível de cobertura de teste por camada pode levar a uma estratégia de teste de qualidade com baixo custo.



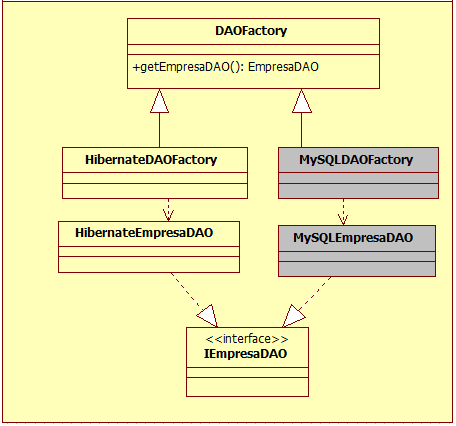
Na arquitetura proposta, a comunicação entre as camadas atende o seguinte protocolo: a camada superior chama a camada inferior evitando dependências cíclicas. Uma classe do tipo <<entidade>> não precisa preocupar-se com as tarefas de persistência de seu conteúdo, delegando-as para as classes de menor nível (repositório).

A camada de aplicação se utiliza do padrão Command (Gamma, 1994) que tem como objetivo transformar um método de uma classe em um objeto o qual pode executar uma ação deste método. O padrão Command encapsula uma solicitação como objeto o que permite enfileirar ou registrar solicitações e implementar recursos de cancelamento de operações (undo). A principal motivação esta relacionada a necessidade de se emitir solicitações para objetos sem nada saber sobra operação que está sendo solicitada ou sobre seu receptor.

O padrão “Facade” também facilita a estratégia de teste, pois tem como objetivo fornecer uma interface unificada para um conjunto de interfaces em um subsistema. O padrão “Facade” define uma interface de nível mais alto que torna o subsistema mais fácil de ser testado.



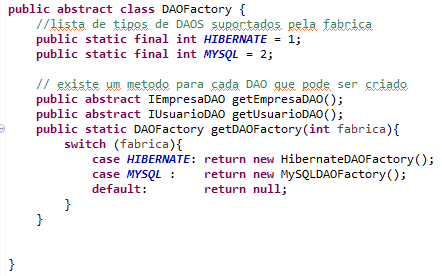
Quando sistema de persistência, na camada de negócios, esta sujeito a sofrer modificações de uma implementação para outra pode ser necessário se projetar o sistema de persistência de maneira a facilitar as modificações quando necessário. Uma possível abordagem seria utilizar do padrão Abstract Factory com objetivo de implementar uma interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas. Nesta estratégia um objeto abstrato fabrica objetos DAO (Abstract Factory) que podem construir vários tipos de fabricas concretas de DAOs. Cada fabrica da suporte a um diferente tipo de implementação de persistência. A figura x apresenta um exemplo do uso desta estratégia.



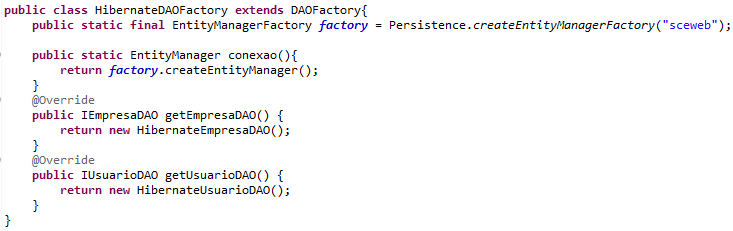
Benefícios:

1. Habilita a transparência – objetos de negocio podem se utilizar de uma fonte de dados sem conhecer detalhes específicos da implementação.
2. Facilita a migração – a camada DAO facilita uma aplicação a migrar para uma implementação diferente de banco de dados
3. Reduz a complexidade de objetos de negócios – como o objetos DAO gerenciam o acesso aos dados isto simplifica a implementação dos objetos de negócios.
4. Centraliza o acesso aos dados a uma camada separada isto facilita a estratégia de teste da camada de dados.

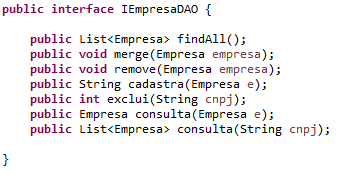
Considere o exemplo da implementação abaixo supondo que a aplicação foi projetada para ser executada utilizando MySQL ou Hibernate. O padrão Abstract Factory foi utilizado para diminuir o acoplamento entre as classes:



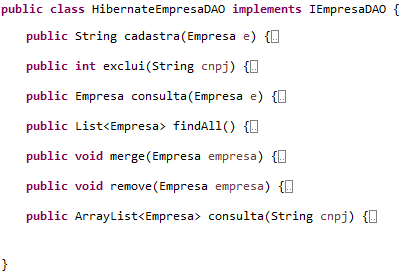
O código de exemplo para HibernateDAOFactory é mostrado na figura 2. A implementação para MySQLDAOFactory é similar exceto para as especificidades de cada implementação, tais como o driver JDBC, URL do banco de dados e diferenças na sintaxe SQL.



A interface IEmpresaDAO mostrado no exemplo 3 define os métodos DAO para persistir o objeto Empresa que são implementados por todas as implementações DAO concretas tais como HibernateEmpresaDAO.



A classe HibernateEmpresaDAO implementa a interface IEmpresaDAO como mostrado na figura 4.



**Total de linhas de código em um método** – um método deve caber em uma tela, um método com mais de 50 linhas de código deve ser refatorado.

**Numero de métodos por classe** – máximo de 10 métodos funcionais excluindo os que começam por “set”, “get”, “is”.

**Complexidade ciclomatica** < 10

Analise de dependência

**Acoplamento aferente** (CA) – numero de classes (ou pacotes) externos que dependem de classes dentro de um determinado pacote

**Acoplamento aferente** (AC) – numero de classes (ou pacotes) externos dos quais as classes dentro de um pacote dependem.

O valor do acoplamento aferente é um bom indicador da dependência que o sistema tem de determinado pacote. Se for um pacote de classes de domínio (figura 2 – visão logica da arquitetura) pode ser aceitável que exista um grande numero de acoplamentos aferentes. Dependendo do resultado da analise de dependências deve ser verificada a necessidade de revisar as responsabilidades do pacote.

O valor do acoplamento eferente indica a fragilidade de um determinado pacote, que depende muito da estabilidade de outros pacotes. Neste caso, pode ser necessário rever as classes do pacote, movendo-as para pacotes mais apropriados.

Dependências cíclicas – Existem se duas classes dependem uma da outra, o caso mais grave é quando existe uma dependência cíclica entre pacotes diferentes. Dependências cíclicas na estrutura de dependências entre pacotes devem ser evitadas.

A figura 6 apresenta uma analise de dependência utilizando o “CodePro AnalytiX” para eclipse. Neste exemplo existe uma dependência cíclica entre os pacotes domínio e serviço. A análise de dependência deve gerar uma “não conformidade” e a equipe deve discutir os mecanismos para se evitar a violação nos critérios de aceitação, estabelecer um prazo para correção ou justificar a necessidade de se manter a dependência entre os pacotes.

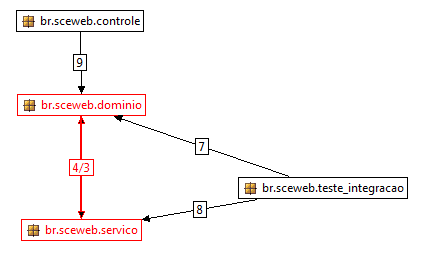


Figura 6 - exemplo de dependência cíclica

A figura 7 apresenta o acoplamento aferente do pacote domínio (13), neste caso pose ser aceitável que exista um numero alto de classes, fora do pacote, que dependam do pacote domínio em outros casos pode ser necessária uma revisão do alto acoplamento.

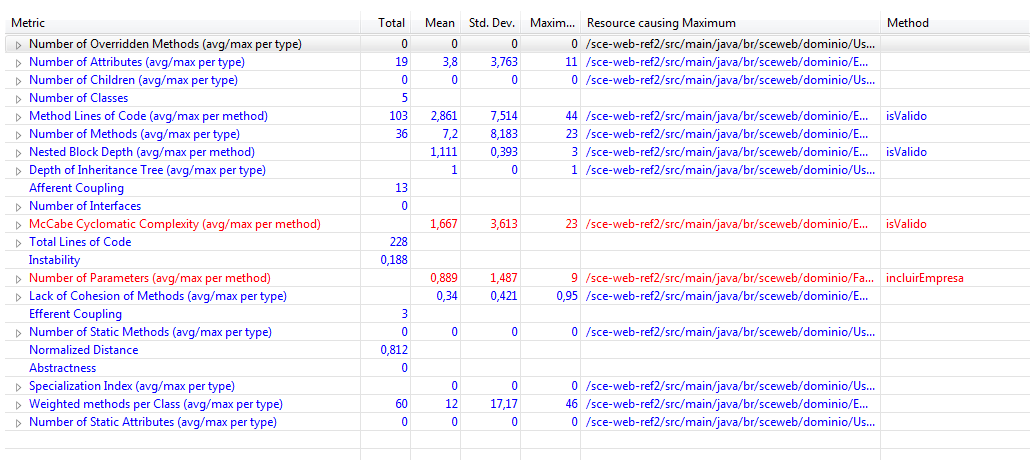


Figura 7 – Analise de dependências – acoplamento aferente = 13 para o pacote domínio usando o plugin Metrics para o Eclipse

Cobertura de teste unitário – A analise de cobertura do teste unitário (figura 8) deve atender aos critérios de priorização dos requisitos estabelecidos para o projeto. Requisitos com prioridade alta devem obter 100% de cobertura de testes unitários. Violações desta prática devem gerar não conformidade e devem ser justificadas pela equipe.

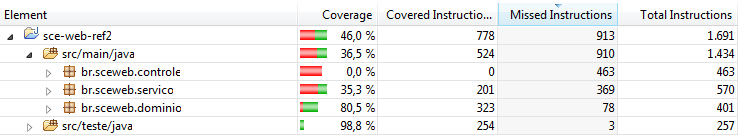


Figura 8 - analise de cobertura utilizando o plugin Coverage para o Eclipse

Referencias

ALUR, Deepak et al. Core J2EE Patterns (Core Design Series): Best Practices and Design Strategies. Sun Microsystems, Inc., 2003.

ERIKSSON, Hans-Erik; PENKER, Magnus. UML toolkit. John Wiley & Sons, Inc., 1997

GAMMA, Erich et al. Design patterns: elements of reusable object-oriented software. Pearson Education, 1994

JOHNSON, Mark. Designing enterprise applications with the J2EE platform. Addison-Wesley Professional, 2002.

KRUCHTEN, Philippe B. The 4+ 1 view model of architecture. **Software, IEEE**, v. 12, n. 6, p. 42-50, 1995.