mundo net

número 09 • junho / julho 2008 • ano II • R$11,90

Colunas

Tools

Integração Contínua (CI)
Analisando o código-fonte sem precisar da IDE

Porta25

Open Source, Open Standard e Interoperabilidade

Visual Studio 2008

Um Guia Prático

Tire o máximo de proveito, conheça sua história, versões e dezenas de dicas.

Cache Multinível em .NET

Aumente o desempenho de aplicações Web com cache em cascata

Global Assembly Cache

Saiba como tirar proveito do GAC para o compartilhamento de assemblies

Desenvolvimento de Jogos com XNA — parte 2

A segunda parte do artigo de games mostra como criar um jogo completo

Gerando código com MyGeneration

Como ser mais produtivo com essa poderosa ferramenta de geração de código

Arquitetura baseada em Office Business Applications

Desenvolva soluções que combinem camadas de front-office por meio de soluções Office System

SharePoint Designer 2007

Ferramenta para aplicar customizações, trabalhar identidade visual e modificar interfaces gráficas

Testes automatizados com DB4O

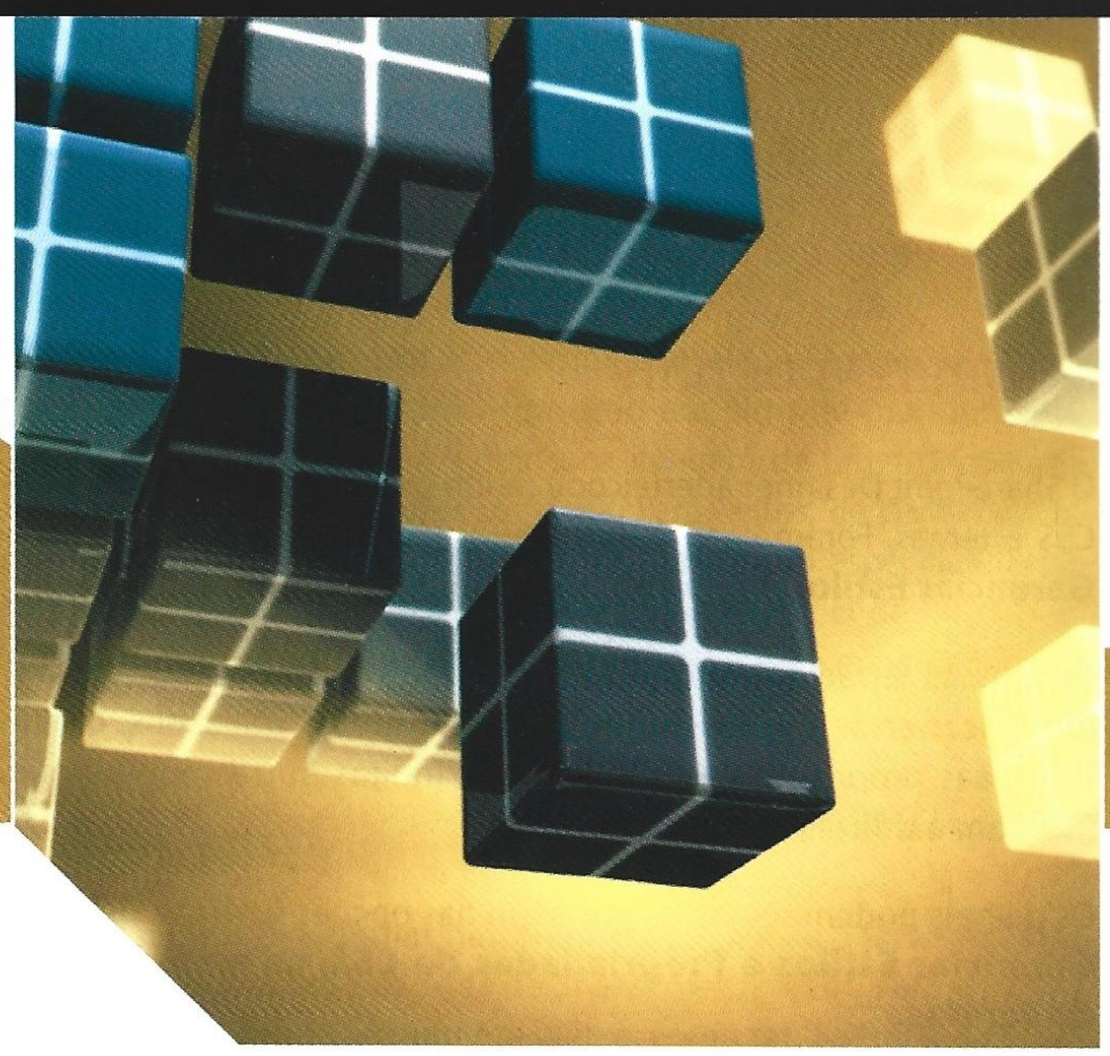
Como ele pode ser uma interessante alternativa para administração de testes automatizados

editora mundo

ISSN 1980-7996

[artigo]

Cenários para Testes Automatizados com o db4o



Alisson Vale

(alisson.vale@phidelis.com.br)

Tem mais de 15 anos de experiência com desenvolvimento de software. Lidera projetos de software desde 2001 com atenção especial à tecnologia.Net desde 2003. Hoje é líder de Projeto e diretor da Phidelis Tecnologia, onde lidera a equipe de desenvolvimento da solução acadêmica oferecida pela empresa. Mantém textos e idéias sobre desenvolvimento ágil de software em seu blog: http://www.phidelis.com.br/blogs/alissonvale.

Paulo César Fernandes

(paulo.cesar@phidelis.com.br)

É team leader na Phidelis Tecnologia, trabalhando há mais de 5 anos com a tecnologia .Net. É graduado em Ciência da Computação e está se especializando em MBA de Gerência de Projetos.

Técnicas e ferramentas para automação de testes são assuntos corriqueiros em qualquer comunidade de desenvolvedores. Há certa unanimidade quanto à idéia de se criar testes automatizados para aumentar a qualidade do software que desenvolvemos. Apesar do prestígio da técnica, poucos realmente conseguem vencer os desafios que surgem no caminho para adotá-la. Neste artigo, convidamos o leitor a entrar na discussão não só dos elementos conceituais que cercam o assunto, mas também conhecer uma interessante alternativa para administração de cenários de testes automatizados.

Uma nova visão para as atividades de teste de software

Quando se fala em testes, a primeira idéia que nos vem à cabeça é qualidade. Fazemos testes para tentar garantir a qualidade do produto que desenvolvemos. Como há várias formas de se avaliar qualidade, também há várias formas de teste. Um produto de software pode ser testado quanto ao seu propósito, funcionamento, robustez, performance, usabilidade e qualquer outro critério considerado relevante por aqueles que irão utilizá-lo.

Por outro lado, não se pode falar em qualidade sem citar o premiado estatístico americano W. Edward Deming. As idéias de Deming influenciaram e transformaram o modo como administradores do mundo inteiro trabalham com o conceito de qualidade. Obviamente quando se fala em qualidade, há um grande número de elementos a serem considerados. Deming sintetizou tais elementos nos seus famosos 14 princípios de administração. Dentre esses princípios – que envolvem, entre outras coisas, aperfeiçoamento contínuo, liderança participativa e valorização das pessoas –, o princípio de número #3 é especialmente interessante para o contexto deste artigo. Ele fala sobre "não depender de inspeções para garantir qualidade".

Segundo Deming, deve-se "eliminar a necessidade de inspeção em larga escala, o que poderá ser alcançado se a qualidade for embutida no próprio produto" [1]. Em outras palavras, quando criamos instrumentos que nos permitem embutir qualidade no produto, de fato reduzimos a necessidade de grandes processos de inspeção após a sua construção. Inspeções são tardias e dispendiosas. Elas não evitam os defeitos e, além disso, o sucesso de uma inspeção (encontrar um defeito) sugere fracasso no processo de produzir software com qualidade. Afinal, qual a vantagem de se encontrar um defeito no seu produto após ele ter sido construído, se não o de admitir que o seu processo de produção é falho e ineficiente?

Mas o que Deming tem a ver com software? Na verdade, Deming influenciou e diretamente conduziu a revolução da indústria japonesa do pós-guerra. Suas idéias ajudaram a formar um novo estilo de gerenciamento baseado em uma filosofia que redefine qualidade como um elemento muito mais amplo em qualquer processo produtivo. Tais idéias também contribuíram para a formação de um novo estilo de administração e gerenciamento. Em 1986, Nonaka e Takeushi publicaram um paper [2] que serviu de influência para a formação de um dos principais e mais recentes movimentos da indústria de software: o Movimento Ágil [3]. Hoje, muito desse modelo de trabalho oriental foi adaptado às questões de engenharia de software, o que nos levou a uma nova visão sobre as atividades de teste de software.

Métodos e metodologias ágeis surgiram durante a década de 90 defenden-

do uma nova perspectiva para a engenharia de software. Atividades de gerenciamento, análise, codificação, design, testes foram significativamente reinterpretadas e reorganizadas de forma a dar origem a um novo modelo de trabalho cujo interesse cresce ano a ano no mundo todo. Dentre elas, as atividades de testes são uma das que geram maiores dúvidas e controvérsias. Como testar? O que testar? Quando testar?

Criando oportunidades para aumentar a qualidade

Antes de começar a pensar no o quê, no como e no quando, é preciso estabelecer algum senso de propósito. Testes devem se converter em oportunidades para inserir qualidade no produto. Assim, sob o ponto de vista do Modelo Ágil, quando falamos em testes, procuramos oportunidades para moldar o nosso produto de tal forma que ele possa absorver elementos de qualidade de forma permanente. Essa palavra "permanente" é o elemento conceitual que nos leva à prática de automatizar testes. Quando automatizamos um teste, esse passa a ser parte do produto. Não é mais um elemento externo desassociado dele. Um teste automatizado injeta qualidade dentro do nosso software.

Essa nova visão trouxe novas práticas de teste. Tais práticas elevaram o propósito da atividade de testes para um patamar diferenciado. Não se trata mais da simples checagem de operações. Trata-se de obter ganhos não só para o produto, mas também para o processo de desenvolvimento.

Quando os testes automatizados podem aumentar a qualidade do processo

A primeira oportunidade de aumentar a qualidade com testes ocorre logo no início do processo de desenvolvimento, enquanto clientes e desenvolvedores ainda tentam se entender a respeito dos comportamentos que o sistema deve apresentar para uma determinada funcionalidade. A criação de um Teste de Aceitação (também conhecido como Customer Acceptance Test) é o instrumento utilizado para otimizar a comunicação necessária nesse processo e garantir a confiabilidade dos resultados esperados. Ao criarem um teste de aceitação, clientes e desenvolvedores definem as regras do sistema por meio de uma especificação formal e executável. É formal no sentido de que está sujeita a um conjunto de regras para redação e formatação, é executável porque poderá ser automaticamente executada por um computador. Esse tipo de teste é também uma boa documentação, porque revela exatamente aquilo que o produto deve fazer. Como ele pode ser executado automaticamente, ele estará sempre sincronizado com as regras implementadas no produto. Qualquer alteração que leve este teste a falhar será facilmente detectável, tornando fácil o trabalho de atualizá-lo ou de mantê-lo funcionando à medida que o sistema cresce ou que ele precise ser alterado.

O propósito dos testes de aceitação é, assim, aumentar a qualidade do processo de comunicação necessário para as atividade de análise e levantamento de requisitos, por meio da criação de especificações executáveis. Note que o teste faz o seu papel, ele verifica se o sistema faz aquilo que deveria. Mas não é esse seu propósito principal. É um benefício secundário se comparado ao ganho de qualidade obtido quando o produto é desenvolvido com tal abordagem.

Benefícios dos Testes de Aceitação:

- processo de comunicação mais eficiente entre os clientes e os desenvolvedores;
- geração de uma especificação que pode validar as regras de negócio do sistema de forma automatizada;
- uma documentação atualizada, fácil de ler, consultar e organizar;
- uma garantia de que futuras intervenções não afetarão as regras definidas agora.

Saiba Mais

O FitNesse [4] é uma das ferramentas de colaboração mais utilizadas para a criação de testes de aceitação. É baseada no Fit Framework [5] criado por Ward Cunningham e utilizado por uma grande quantidade de equipes de software em todo o mundo. Os links para as ferramentas podem ser obtidos na seção Referências ao final deste artigo.

Quando os testes automatizados podem aumentar a qualidade do produto

Na medida em que o desenvolvedor começa a escrever código-fonte, um novo elemento de qualidade surge: o design do software. Design é muito importante porque revela o nível de organização em que está seu código-fonte de modo que ele possa ser facilmente alterado ou extendido. O desenvolvimento de qualquer software exigirá uma longa seqüência de alterações no código até a sua entrega final. Um código com um design ruim pode rapidamente se tornar um problema na medida em que cria obstáculos para que o desenvolvedor possa evoluir o produto com rapidez. A deterioração do design afeta diretamente a qualidade do produto final, pois ele afeta a confiança e a velocidade da equipe de desenvolvimento. Equipes improdutivas sofrem mais pressão da gerência, o que as leva a produzir com pressa e sem esmero. Essa pressa causa menos atenção para os problemas de design, o que o degrada ainda mais, gerando um efeito bola-de-neve que elevará a quantidade de defeitos a serem posteriormente descobertos nas inspeções.

Assim, é importante tentar utilizar técnicas que atuem na direção de proteger o design da aplicação enquanto novas linhas de código são produzidas. Uma dessas técnicas chama-se Test Driven Development (TDD) e é utilizada em desenvolvimento ágil para aumentar a qualidade do design do software enquanto o desenvolvedor o codifica. Isso é feito por meio da criação de testes conhecidos como Testes Unitários (Unit Tests). Estes tipos de teste atuam no nível de classes e métodos criados durante o processo de codificação. Em termos gerais, antes de o desenvolvedor escrever novas linhas de código para implementar um determinado comportamento, ele escreve um teste para validar essa implementação.

Ao executar este teste, ele falha porque em um primeiro momento ainda não há implementação, ou ela ainda não está completa. O desenvolvedor começa então um ciclo de implementação-execução até que o teste gere um resultado positivo. Quando um teste passa, o desenvolvedor está

livre para revisar o código e reorganizá-lo. O desenvolvedor pode agora avançar para uma nova implementação adotando a mesma abordagem. Obviamente, esses testes poderão ser executados de forma automatizada no futuro o que garantirá não só o funcionamento das classes e objetos, mas também a evolução sustentável do design do software.

Como no caso do Teste de Aceitação, o propósito do Teste Unitário é influenciar positivamente o design da aplicação, permitindo que ele evolua sem sofrer os danos causados pelo processo de degradação. O produto tem sua qualidade aumentada na medida em que tem uma estrutura de código mais bem organizada e mais propensa para novas implementações. Além disso, o desenvolvedor tem mais condições de melhorar continuamente o design já que o conjunto de teste lhe oferece a segurança necessária para alterar o código sem afetar o funcionamento do produto.

Benefícios dos Testes Unitários:

- ajudam a melhorar o design do software;
- tornam-se uma boa documentação para o uso do modelo de objetos;
- podem ser executados de forma automatizada;
- futuras intervenções que afetem o comportamento esperado dos objetos serão detectadas de forma imediata.

Outros testes para garantia de qualidade

Em alguns casos, os testes apresentados até o momento (Aceitação e Unitário) podem não ser suficientes para garantir a qualidade do produto. As Metodologias Ágeis em geral lidam bem com a possibilidade de pessoas com a função de quality assurance (QA) serem integradas ao time de projeto de forma a olhar para o software sob uma nova perspectiva. Diferente de uma abordagem tradicional, essas pessoas têm o papel de se integrar ao time de desenvolvimento de forma a ajudá-lo a melhorar suas habilidades para inserir mais qualidade no produto. A meta é alcançar a margem de zero bugs. Para isso, eles podem trabalhar na direção de automatizar ou executar outros tipos de testes. Veja alguns exemplos de testes que podem ser criados por pessoas que desempenham esse papel:

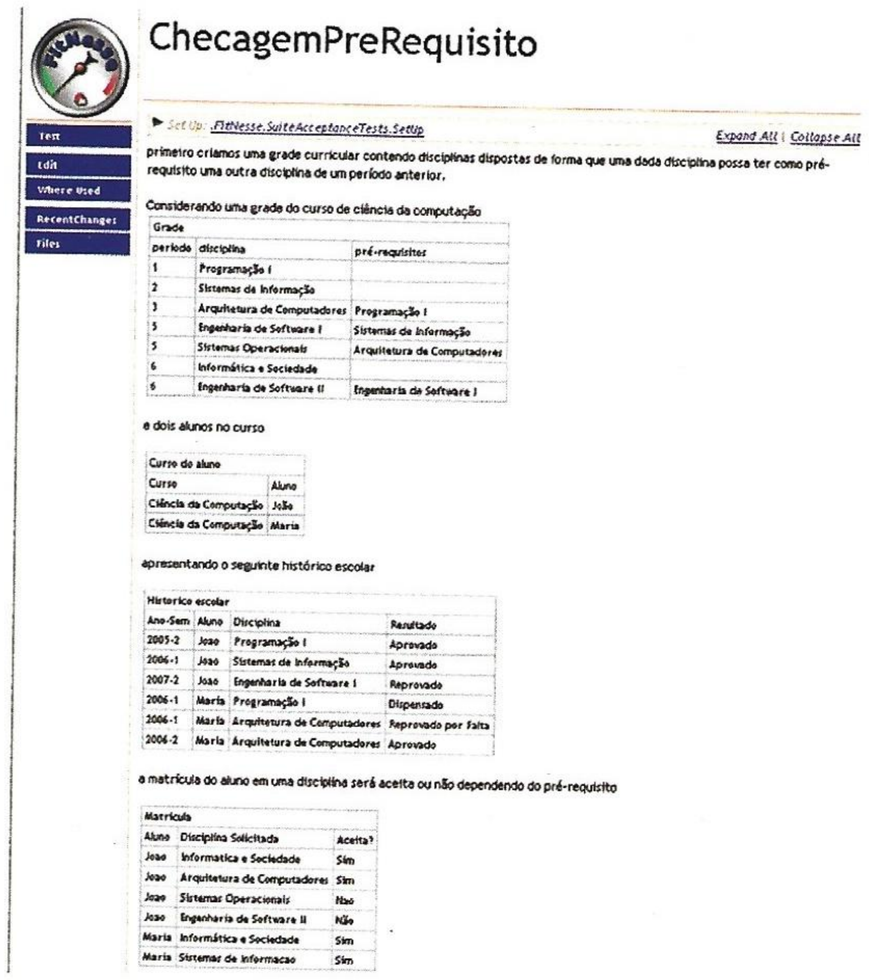
Testes End-to-end [6]: exercitam a aplicação da forma que ela será efetivamente utilizada pelos usuários. Ferramentas como o Watir [7] ou o Selenium [8] podem, por exemplo, serem utilizadas para automatizar o uso completo da funcionalidade de um produto web-based. Elas permitem especificar o preenchimento de caixas de texto, seleção de listas, cliques em botões e navegação por entre páginas Web.

Testes Exploratórios [9]: fazer análises sistêmicas sobre o funcionamento do software para descobrir pontos fracos ou formas de se gerar inconsistências. Pode ser automatizado ou não. É uma atividade focada na prevençao de bugs.

Testes Especializados: testes de carga, de performance, de integração, de aderência a padrões e outros testes com propósito específico.

Cenários

No formato de teste de aceitação mais comum, você descreve os conceitos utilizando tabelas e depois cruza os dados dessas tabelas para indicar o que é válido ou o que não é. Essa técnica pode ser utilizada para descrever cenários que variam desde um simples cadastro de clientes até uma fórmula de cálculo em uma rotina mais complexa. Blocos de textos simples podem ser utilizados para conduzir o leitor durante a interpretação das tabelas. O documento é construído em um formato wiki, o que abre a possibilidade para enriquecê-lo de inúmeras maneiras.



ChecagemPreRequisito

Test | Edit | Where Used | RecentChanges | Files

SetUp: .FitNesse.SuiteAcceptanceTests.SetUp — Expand All | Collapse All

primeiro criamos uma grade curricular contendo disciplinas dispostas de forma que uma dada disciplina possa ter como pré-requisito uma outra disciplina de um período anterior.

Considerando uma grade do curso de ciência da computação

Grade		
periodo	disciplina	pré-requisitos
1	Programação I	
2	Sistemas de Informação	
3	Arquitetura de Computadores	Programação I
5	Engenharia de Software I	Sistemas de Informação
5	Sistemas Operacionais	Arquitetura de Computadores
6	Informática e Sociedade	
6	Engenharia de Software II	Engenharia de Software I

e dois alunos no curso

Curso de aluno	
Curso	Aluno
Ciência da Computação	João
Ciência da Computação	Maria

apresentando o seguinte histórico escolar

Historico escolar			
Ano-Sem	Aluno	Disciplina	Resultado
2005-2	Joao	Programação I	Aprovado
2006-1	Joao	Sistemas de Informação	Aprovado
2007-2	Joao	Engenharia de Software I	Reprovado
2006-1	Maria	Programação I	Dispensado
2006-1	Maria	Arquitetura de Computadores	Reprovado por Falta
2006-2	Maria	Arquitetura de Computadores	Aprovado

a matrícula do aluno em uma disciplina será aceita ou não dependendo do pré-requisito

Matricula		
Aluno	Disciplina Solicitada	Aceita?
Joao	Informatica e Sociedade	Sim
Joao	Arquitetura de Computadores	Sim
Joao	Sistemas Operacionais	Nao
Joao	Engenharia de Software II	Não
Maria	Informática e Sociedade	Sim
Maria	Sistemas de Informacao	Sim

Figura 1. Um teste de aceitação redigido no Fitnesse.

Vamos supor que em um projeto para a construção de um sistema acadêmico para uma universidade, sua equipe de desenvolvimento se depare com a necessidade de criação de uma nova funcionalidade para checagem de pré-requisitos na matrícula de um aluno. Ao detalhar o funcionamento desse novo recurso com o seu cliente, você chega ao teste de aceitação apresentado na figura 1.

Esse tipo de teste é criado no momento em que cliente e desenvolvedor tentam chegar a um consenso sobre como o sistema deverá se comportar. Quando ele é criado ainda não existe implementação. Na medida em que essa implementação vai sendo produzida, o desenvolvedor poderá executar o documento de forma a verificar se o que ele produziu está consistente com o que foi acordado inicialmente (veja figura 2). O cliente também poderá alterar o documento criando novas possibilidades. Embora o formato de código dessas tabelas seja muito simples, o cliente pode utilizar uma planilha eletrônica, como o Excel, para criá-las. O Fitnesse fará o processo de conversão sem maiores problemas.

Note que as três primeiras tabelas apresentadas no teste da figura 2 nada mais são do que cenários preexistentes, ou seja, "considerando algumas pré-condições precisamos testar a seguinte regra". Assim, para fazer a checagem de pré-requisito, é preciso ter uma grade curricular com algumas disciplinas, é preciso ter alguns alunos matriculados em um dado curso e também é preciso saber o histórico escolar desses alunos. Tudo isso apenas para testar se ele pode ou não se matricular em uma dada disciplina em virtude dos seus pré-requisitos.

Um dos grandes desafios de quem trabalha com testes automatizados é a necessidade de compor cenários diversos para sua execução. Esse desafio cresce proporcionalmente ao aumento do tamanho do software. Por

Figura 2. Um teste de aceitação depois de sua execução automatizada.

exemplo, em um software de automação comercial, para testar o fechamento de uma venda, você pode precisar de produtos pré-cadastrados, de um estoque alimentado, de usuários registrados com perfis específicos, de um caixa aberto etc.

Sem essa rede de conceitos pré-configurados, você não terá condições de exercitar os cenários para o fechamento de uma venda. Você pode levar muito tempo compondo os objetos necessários para simplesmente começar a trabalhar na funcionalidade. Além disso, o cliente pode vislumbrar vários cenários a serem considerados. Uma venda de um produto sem estoque, por exemplo, ou uma venda com comissionamento ou sem comissionamento, ou ainda uma venda com caixa fechado.

Em uma situação como essa é bem provável que o esforço para criar os cenários adjacentes seja maior do que a própria implementação da regra. No entanto, mesmo considerando todo esse esforço ainda é melhor usar essa abordagem do que esperar o software ficar pronto para inspecioná-lo. A tendência é que a equipe consiga criar seus próprios recursos para reuso de cenários ao longo do tempo, minimizando esse incoveniente.

Há diversas formas de reuso de cenários de forma que eles possam ser aproveitados em diferentes ocasiões. Quando há reuso, o tempo necessário para criar um teste automatizado obviamente é diminuído. Há quem crie assemblies inteiros com objetos pré-configurados. Há também quem gere os cenários em bancos de dados e depois os carregue para a aplicação e há também quem crie novos cenários toda vez que precisa de um teste. Cada forma tem seus prós e contras. Veja na Listagem 2 um exemplo do código necessário para criar os cenários descritos no teste do nosso exemplo.

A Listagem 1 apresenta a estrutura de objetos que seria necessária para configurar as condições de execução do teste. Simplificamos bastante o modelo para ajudar o leitor no entendimento, mas em um sistema real poderíamos ter um modelo com uma quantidade bem maior de detalhes e classes envolvidas. Para validar as várias alternativas de matrícula precisamos pré-configurar um cenário com a criação de uma grade curricular completa e com a criação de um histórico para o aluno cujas regras de matrícula serão validadas. O código da Listagem 1 ilustra o tipo de problema que podemos ter com cenários quando queremos manter nossos testes automatizados.

Quando falamos de Testes Unitários, o fato de você precisar de um grande esforço para configurar os objetos que trabalharão em um dado cenário de teste pode revelar um problema de design. Mas dificilmente você conseguirá evitar alguns momentos em que será necessário um grande esforço para configurar o teste. Muitas vezes tentamos automatizar um teste com um nível de abstração elevado (o que é normal em testes de aceitação). Nesses casos, esse problema será uma constante, pois é o todo da regra que interessa para o cliente. E quando você precisa verificar o todo de uma regra, você certamente precisará trabalhar com uma rede muito grande de pré-condições.

Cenários também são necessários para a criação de testes que exponham bugs. Muitas equipes ágeis trabalham com a idéia de que quando um bug é encontrado no sistema (seja ele encontrado na inspeção ou já em produção), a primeira coisa a se fazer é criar um teste automatizado que "exponha" o bug, só então ele é corrigido. Isso evita que o problema volte a ocorrer. A dificuldade nessa abordagem reside no fato de que há uma grande pressão para que o bug seja corrigido, então, se a criação do teste não for um processo rápido, a chance desse fluxo não ser obedecido pelo desenvolvedor será grande.

Precisamos então encontrar alguma forma de otimizar a confecção desses cenários de forma a dar sustentação para as boas práticas de teste trazidas pelos Métodos Ágeis. Ser capaz de administrar cenários grandes e complexos é uma realidade quando se trabalha com testes automatizados. Como fazer então para tentar minimizar esse problema e conseguir um pouco mais de produtividade no processo de preparação de um teste?

Uma possível solução: db4o

Uma maneira de otimizar esse processo é criando recursos de interceptação na sua aplicação de forma a inserir no seu produto a capacidade de gerar os cenários automaticamente utilizando-se de seus próprios recursos. Nesse modelo, métodos da sua aplicação poderiam ser configurados de forma a serem interceptados quando forem executados. Quando em execução, os métodos contêm todo um conjunto de objetos prontos, que se interceptados no momento apropriado, disponibilizarão um cenário pronto para ser armazenado de forma a ser reutilizado. Vamos dar um exemplo mais concreto para faciltar o entendimento.

Imagine no exemplo da figura 1 que eu já tenha no meu produto as funcionalidades para configuração de grades, matrículas de alunos e geração de histórico escolar. Não seria interessante se eu pudesse utilizar o próprio produto para gerar esses cenários de forma a reusá-los nos meus testes? Assim, eu poderia navegar pela interface do produto ou simplesmente rodar um dos seus testes. O próprio sistema geraria os objetos relacionados com aquele cenário e a partir daí eles poderiam ser utilizados para a implementação de novos testes.

O db4o [10] é um banco de dados orientado a objetos com uma ótima reputação na comunidade de desenvolvimento. Ele é capaz de armazenar e recuperar rapidamente grandes estruturas de objetos. É simples de usar, muito rápido, pode estar embutido no seu produto sem a necessidade de uma instalação separada e obviamente não precisa de mapeamento objeto-relacional para a persistência dos objetos. É a solução ideal para a técnica aqui proposta. Com ele, poderemos criar interceptações no código (mesmo que seu produto já trabalhe com outras soluções de armazenamento) de forma a conseguir recuperar os objetos prontos, persistindo-os em sua forma pura no disco.

Listagem 1. Criando os cenários com objetos puros.

```
[TestFixture]
public class TesteChecagemPreRequisito
{
    [Test]
    public void PreRequisitoDeveSerObrigatorioParaMatricula()
    {
        Curso computacao = new Curso("Ciência da Computação");

        Disciplina prog1 = new Disciplina("PROGI", "Programação I");
        Disciplina sis = new Disciplina("SIS", "Sistemas de Informação");
        Disciplina arqcom = new Disciplina("ARQCOM",
            "Arquitetura de Computadores");
        Disciplina engsoft1 = new Disciplina("ENGSOFTI",
            "Engenharia de Software I");
        Disciplina sisope = new Disciplina("SISOPE",
            "Sistemas Operacionais");
        Disciplina infosoc = new Disciplina("INFOSOC",
            "Informática e Sociedade");
        Disciplina engsoft2 = new Disciplina("ENGSOFTII",
            "Engenharia de Software II");

        Grade grade = new Grade(computacao);
        grade.Disciplinas.Add(new DisciplinaGrade(prog1, null, 1));
        grade.Disciplinas.Add(new DisciplinaGrade(sis, null, 2));
        grade.Disciplinas.Add(new DisciplinaGrade(arqcom, prog1, 3));
        grade.Disciplinas.Add(new DisciplinaGrade(engsoft1, sis, 5));
        grade.Disciplinas.Add(new DisciplinaGrade(sisope, arqcom, 5));
        grade.Disciplinas.Add(new DisciplinaGrade(infosoc, null, 6));
        grade.Disciplinas.Add(new DisciplinaGrade(engsoft2,
            engsoft1, 6));

        Aluno joao = new Aluno("João Batista", grade);

        HistoricoEscolar historicoJoao = new HistoricoEscolar(joao,
            computacao);

        historicoJoao.AddDisciplina(new HistoricoEscolarDisciplina(
            2005, 2, prog1, EnumResultado.Aprovado));
        historicoJoao.AddDisciplina(new HistoricoEscolarDisciplina(
            2006, 1, sis, EnumResultado.Aprovado));
        historicoJoao.AddDisciplina(new HistoricoEscolarDisciplina(
            2007, 2, engsoft1, EnumResultado.Reprovado));

        Matricula matricula = new Matricula();

        Assert.AreEqual(true, matricula.ChecarPreRequisito(joao,
            infosoc));
        Assert.AreEqual(true, matricula.ChecarPreRequisito(joao,
            arqcom));
        Assert.AreEqual(false, matricula.ChecarPreRequisito(joao,
            sisope));
        Assert.AreEqual(false, matricula.ChecarPreRequisito(joao,
            engsoft1));
    }
}
```

Em outras palavras, você pode decorar seu código com instruções que permitem ao db4o interceptá-lo em pontos-chave para coletar toda a rede de objetos que está na memória em determinado momento, armazenando-os em uma estrutura persistente. Vamos a um exemplo.

Imagine que eu possa alterar o método ChecarPreRequisito apresentado anteriormente de forma a configurá-lo para ser interceptado pelo db4o (Listagem 2).

A Listagem 2 usa um pouco de Programação Orientada a Aspectos [11] de forma a acrescentar recursos de interceptação no método sem afetar

Listagem 2. Interceptação configurada no método ChecarPreRequisito.

```
[Intercept]
public class Matricula : ContextBoundObject
{
    [PreProcessamento(typeof(ParametroParaDB4O))]
    public bool ChecarPreRequisito(Aluno aluno,
        Disciplina disciplina)
    {
        // Código para checagem do pré-requisito
        return true;
    }
}
```

substancialmente seu design ou sua implementação. Quando o método ChecarPreRequisito for acionado pelo uso do sistema ou pela própria execução de um teste, o método Process de ParametroParaDB4O será automaticamente executado. Nesse momento, todos os objetos passados como parâmetro para o método ChecarPreRequisito estão disponíveis na lista "msg.Args". A seguir, poderemos persisti-los facilmente usando os recursos de armazenagem do db4o (veja Listagem 3).

Após a criação do banco de dados orientado a objetos via interceptação, podemos utilizar o ObjectManager para exibição dos dados. O Object Manager [13] é um aplicativo que permite consultas aos objetos armazenados em um banco db4o. Os dados podem ser visualizados como em um banco de dados relacional tradicional. Na figura 3 consultamos todos os objetos do tipo Disciplina armazenados nesse banco de dados.

Saiba Mais

Não é o propósito deste artigo entrar nos detalhes de implementação da interceptação. Essa implementação foi inspirada em um artigo intitulado "Intercepting method calls in C#, an approach to AOSD" [12] publicado no site CodeProject. Link disponível na seção Referências.

Recarregando o cenário para reuso

Depois de criado o cenário, a equipe de desenvolvimento pode então criar uma estrutura de pastas e arquivos para armazená-los. No nosso exemplo, utilizamos um número aleatório para gerar o banco de dados, mas isso pode ser facilmente convertido para uma estrutura de nomes padronizados. Depois de criado o cenário, será muito simples recarregá-lo para utilização nos testes. O próprio db4o oferece uma larga variedade de formas para obter os objetos do banco de dados. Veja um exemplo na Listagem 4.

Pode ser também uma opção interessante manter o código necessário para obtenção dos objetos armazenados no db4o encapsulados em uma área própria para a administração de cenários. Assim, você manterá o código do teste mais fácil de ler, de manter e de reusar. Veja um exemplo na Listagem 5.

Vantagens e desvantagens

Como é de se esperar, administrar cenários da forma exposta até aqui tem suas vantagens e desvantagens. A principal vantagem diz respeito

Listagem 3. Persistindo objetos passados como argumento para o método interceptado.

```
using Db4objects.Db4o;

public class ParametroParaDB4O : IPreProcessamento
{
   public void Process(ref IMethodCallMessage msg)
   {
      string guid = Guid.NewGuid().ToString();

      IObjectContainer db = Db4oFactory.OpenFile(guid);

      foreach (object item in msg.Args)
      {
         db.Set(item);
      }
      db.Close();
   }
}
```

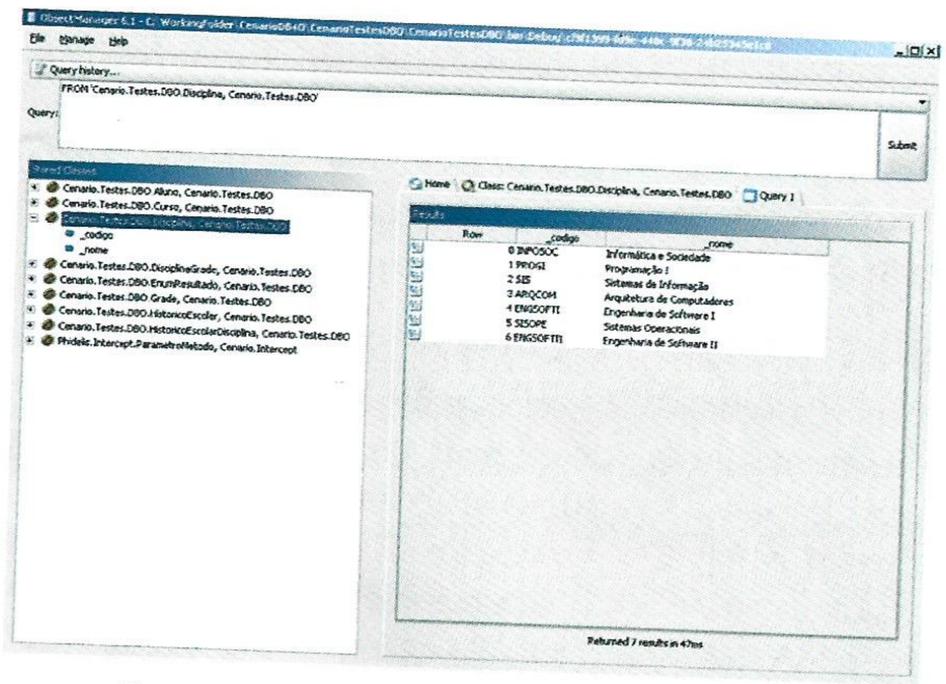


Figura 3. Visualização dos objetos que compõem um cenário armazenado.

ao tempo necessário para criar um cenário. Na medida em que a equipe se familiariza com a abordagem, os cenários serão produzidos de forma cada vez mais rápida. Isso afeta diretamente a capacidade da equipe de injetar qualidade no produto, sem que isso se torne um empecilho para a produtividade e para o ritmo na geração de valor que o projeto deve ter.

Quanto às desvantagens, o principal ponto de preocupação diz respeito à manutenibilidade dos cenários. Nessa abordagem, criamos mais um ponto de redundância no produto. Isso afetará a facilidade necessária para que ele possa ser modificado. Apesar disso ser uma preocupação, o próprio fato de os cenários estarem intimamente ligados aos testes, torna fácil a detecção imediata de todos os pontos da base de cenários onde serão necessárias intervenções, o que minimiza um pouco o problema.

De qualquer forma, o custo–benefício dessa técnica, assim como o de qualquer outra técnica ou prática a ser incorporada em um projeto, deve ser avaliada caso a caso pelos próprios membros da equipe. A adaptabilidade às suas condições técnicas (condições do ambiente, da equipe e do próprio produto) será crucial para determinar se esta será realmente efetiva no seu ambiente.

Considerações finais

Este artigo procurou cobrir os principais aspectos relacionados com garantia de qualidade sob a perspectiva ágil, dando especial ênfase aos testes automatizados e aos detalhes conceituais que cercam sua implementação. Quando se trata de testes automatizados, a criação e administração de cenários é especialmente desafiadora, o que tem nos levado a procurar soluções criativas para tentar otimizá-las. Nesse contexto, o db4o ou outras ferramentas similares podem ser utilizadas para criar formas de facilitar o trabalho com cenários de teste, o que, em caso de sucesso, representaria um grande ganho na capacidade de uma equipe de desenvolvimento incorporar mais qualidade aos seus produtos.

Listagem 4. O mesmo teste recarregando um cenário preexistente.

```
[TestFixture]
public class TesteChecagemPreRequisito
{
    [Test]
    public void PreRequisitoDeveSerObrigatorioParaMatricula()
    {
        IObjectContainer container = Db4oFactory
            .OpenFile("32dd8881-0309-4b55-b05c-038bb3aa059a");

        Aluno joao = (Aluno)container.Get(typeof(Aluno))[0];

        Disciplina infosoc = (Disciplina)container.Get(new
            Disciplina("INFOSOC", null))[0];

        container.Close();

        Matricula matricula = new Matricula();
        Assert.AreEqual(true, matricula.Executar(joao,
            infosoc));
    }
}
```

Listagem 5. Utilizando cenários encapsulados.

```
[TestFixture]
public class TesteChecagemPreRequisito
{
   [Test]
   public void PreRequisitoDeveSerObrigatorioParaMatricula()
   {
      String idCenario = "32dd8881-0309-4b55-b05c-038bb3aa059a";
      Cenario cenario = new CenariosMatricula(idCenario);

      Aluno joao = cenario.GetAlunoPeloNome("Joao");

      Disciplina infosoc = cenario
          .GetDisciplinaPeloCodigo("INFOSOC");

      Matricula matricula = new Matricula();
      Assert.AreEqual(true, matricula.Executar(joao,
          infosoc));
   }
}
```

Referências

[1] Deming, W. Edward. Saia da Crise, 2003. Ed. Futura. Pg. 39

[2] Nonaka, Ikujiro e Takeushi, Hirotaka. The New Product Development Game, Harvard Business Review Article, 1986.

[3] Movimento Ágil: Descrito pelo Manifesto Ágil em http://www.agilemanifesto.org

[4] FitNesse: http://fitnesse.org

[5] Fit: http://fit.c2.com

[6] Shore, James; Warden, Shane. The Art of Agile Development, O'reilly, 2007. Pg. 299

[7] Watir: http://wtr.rubyforge.org/

[8] Selenium: http://selenium.openqa.org/

[9] Bach, James. Exploratory Testing Explained. http://www.satisfice.com/articles/et-article.pdf

[10] db4o: http://www.db4o.com

[11] AOP: http://en.wikipedia.org/wiki/Aspect-oriented_programming

[12] J4amieC, Intercepting method calls in C#, an approach to AOSD. http://www.codeproject.com/KB/cs/aspectintercept.aspx

[13] db4o Object Browser: http://developer.db4o.com/files/folders/objectmanager_63/default.aspx.