A Importância dos Testes Automatizados

Controle ágil, rápido e confiável de qualidade

Paulo Cheque Bernardo (paulocheque@agilcoop.org.br) e Fabio Kon (kon@ime.usp.br)

Artigo publicado na Engenharia de Software Magazine, 1(3), pp. 54-57. 2008.
www.devmedia.com.br/esmag

Introdução

Controlar a qualidade de sistemas de software é um grande desafio devido à alta complexidade dos produtos e às inúmeras dificuldades relacionadas ao processo de desenvolvimento, que envolve questões humanas, técnicas, burocráticas, de negócio e políticas. Idealmente, os sistemas de software devem não só fazer corretamente o que o cliente precisa, mas também fazê-lo de forma segura, eficiente e escalável e serem flexíveis, de fácil manutenção e evolução.

Salvo honrosas exceções, na indústria de software brasileira, estas características são muitas vezes asseguradas através de testes manuais do sistema após o término de módulos específicos ou até mesmo do sistema inteiro. Como veremos mais adiante neste artigo, esta abordagem manual e ad hoc leva à ocorrência de muitos problemas e deve ser evitada. Este artigo se inspira na filosofia dos Métodos Ágeis de Desenvolvimento de Software e em práticas recomendadas pela Programação eXtrema (XP), em especial nos Testes Automatizados, que é uma técnica voltada principalmente para a melhoria da qualidade dos sistemas de software. Este artigo apresenta também alguns exemplos de testes automatizados baseados em excelentes ferramentas gratuitas disponíveis como software livre.

Cenário comum de desenvolvimento

O modo convencional de desenvolvimento de uma funcionalidade é estudar o problema, pensar em uma solução e, em seguida, implementá-la. Após esses três passos, o desenvolvedor faz testes manuais para verificar se está tudo funcionando como o esperado. É normal que erros sejam detectados ao longo do processo de desenvolvimento, então os desenvolvedores precisam encontrar o erro, corrigi-lo e refazer o conjunto de testes manuais.

Além disso, para verificar se algum erro deixou de ser identificado durante a fase de desenvolvimento, antes de colocar o sistema em produção, é muito comum submeter o software a um processo de avaliação de qualidade. Esse controle de qualidade geralmente é realizado com o auxílio de testes manuais executados por desenvolvedores, usuários ou mesmo por equipes especializadas em testes.

Este cenário é comum principalmente em empresas que utilizam metodologias rígidas que possuem fases bem definidas, geralmente derivadas do modelo de cascata. Este tipo de metodologia freqüentemente leva à aparição de diversos problemas recorrentes nas indústria de software, tais como atrasos nas entregas, criação de produtos com grande quantidade de erros e dificuldade de manutenção e evolução. Isto deve-se às dificuldades da realização de testes manuais.

A execução manual de um caso de teste é rápida e efetiva, mas a execução e repetição de um vasto conjunto de testes manualmente é uma tarefa muito dispendiosa e cansativa. É normal e compreensivo que os testadores não verifiquem novamente todos os casos a cada mudança significativa do código; é deste cenário que surgem os erros de software, trazendo prejuízo para as

equipes de desenvolvimento que perdem muito tempo para identificar e corrigir os erros e também prejuízo para o cliente que, entre outros problemas, sofre com constantes atrasos nos prazos combinados e com a entrega de software de qualidade duvidosa.

Mas o aspecto mais crítico deste cenário é o efeito "bola de neve". Como é necessário muito esforço para executar todo o conjunto de testes manuais, dificilmente a cada correção de um erro, a bateria de testes será executada novamente como seria desejável. Muitas vezes, isso leva a erros de regressão (erros em módulos do sistema que estavam funcionando corretamente e deixam de funcionar). A tendência é este ciclo se repetir até que a manutenção do sistema se torne uma tarefa tão custosa que passa a valer a pena reconstruí-lo completamente.

Uma nova abordagem

Muitos métodos ágeis, como *Lean*, *Scrum* e *XP* (vide referências) recomendam que todas as pessoas de um projeto (programadores, gerentes, equipes de homologação e até mesmo os clientes) trabalhem controlando a qualidade do produto todos os dias e a todo momento, pois acreditam que prevenir defeitos é mais fácil e barato que identificá-los e corrigi-los. A Programação eXtrema, em particular, recomenda explicitamente testes automatizados para ajudar a garantir a qualidade dos sistemas de software. Vale ressaltar ainda, que os métodos ágeis não se opõem a quaisquer revisões adicionais que sejam feitas para aumentar a qualidade.

Testes automatizados são programas ou *scripts* simples que exercitam funcionalidades do sistema sendo testado e fazem verificações automáticas nos efeitos colaterais obtidos. A grande vantagem desta abordagem, é que **todos** os casos de teste podem ser facilmente e rapidamente repetidos a qualquer momento e com pouco esforço.

A reprodutibilidade dos testes permite simular identicamente e inúmeras vezes situações específicas, garantindo que passos importantes não serão ignorados por falha humana e facilitando a identificação de um possível comportamento não desejado.

Além disso, como os casos para verificação são descritos através de um código interpretado por um computador, é possível criar situações de testes bem mais elaboradas e complexas do que as realizadas manualmente, possibilitando qualquer combinação de comandos e operações. A magnitude dos testes pode também facilmente ser alterada. Por exemplo, é trivial simular centenas de usuários acessando um sistema ou inserir milhares de registros em uma base de dados, o que não é factível com testes manuais.

Todas estas características ajudam a solucionar os problemas encontrados nos testes manuais, diminuindo a quantidade de erros e aumentando a qualidade do software. Como é relativamente fácil executar todos os testes a qualquer momento, mudanças no sistema podem ser feitas com segurança, o que aumenta a vida útil do produto.

Na grande maioria das vezes, os testes automatizados são escritos programaticamente, por isso é necessário conhecimento de programação. Nas seções seguintes, veremos alguns exemplos de código de testes automatizados que utilizam algumas ferramentas e arcabouços livres bem populares. No entanto, existem alguns tipos de testes que possuem ferramentas gráficas que escondem os detalhes de implementação permitindo que outros profissionais também consigam escrever seus próprios testes, por exemplo JMeter[1] para testes de estresse e desempenho e Selenium-IDE[2] para criação de testes de interface gráfica para aplicações Web.

Exemplo de Teste Automatizado de Unidade

Se o leitor quer começar a colocar em prática testes automatizados, o primeiro tipo de teste que se deve fazer são os de unidade. Uma unidade pode ser entendida como o menor trecho de código de um sistema que pode ser testado, podendo ser uma função ou módulo em programas

1 JMeter: http://jakarta.apache.org/jmeter
2 Selenium-IDE: http://selenium-ide.openqa.org

procedimentais ou métodos ou classes em programas orientados a objetos. O teste de uma unidade é o tipo mais importante de teste para a grande maioria das situações, já que é ele que deve testar se um algoritmo faz o que deveria ser feito e garantir que o código encapsulado por uma unidade deve produzir o efeito colateral esperado. Outra vantagem importante é que o teste de unidade é focalizado em um trecho específico do código, desta forma os erros encontrados são facilmente localizados, diminuindo o tempo gasto com depuração.

Uma prática comum no passado (até a década de 90) era criar uma função de teste em cada módulo ou classe do sistema que continha algumas simulações de uso da unidade. O problema desta prática está na falta de organização, já que o código adicional era misturado ao próprio sistema afetando a legibilidade do código. Por isso, surgiram os arcabouços (*frameworks*) que auxiliam e padronizam a escrita de testes automatizados, facilitando o isolamento do código de teste do código da aplicação. Estes arcabouços de testes de unidade são conhecidos como parte da família de arcabouços xUnit. Por exemplo, existe o arcabouço pioneiro SUnit[3] para SmallTalk criado por Kent Beck, que foi seguido por implementações para outras linguagens, tais como JUnit[4] e TestNG[5] para Java, JSUnit[6] para Javascript, CppTest[7] para C++, csUnit[8] para .NET, entre outras.

Para ilustrar, vamos ver um exemplo de teste automatizado para aplicações Java com o auxílio da versão 4 do popular JUnit. Os testes são organizados em casos de teste definidos através de anotações Java `@Test` antes da definição de cada método de teste. O arcabouço também possui uma gama de métodos auxiliares para comparar os efeitos colaterais esperados com os obtidos, tais como `assertEquals` que compara a igualdade do conteúdo de objetos, `assertSame` que compara se duas referências se referem ao mesmo objeto, `assertNull` que verifica se uma dada referência é nula, entre outros (vide Figura 1).

```
public class ExemploJUnitTest {
    @Test // Este método é um caso de teste
    public void testaAdicaoDeDiasEmUmaData() throws Exception {
        SimpleDateFormat formatador = new SimpleDateFormat("dd/MM/yyyy");
        Date dataDeReferencia = formatador.parse("05/06/2008");
        Date dataDaqui5Dias = formatador.parse("10/06/2008");
        Date dataObtida = DateUtil.adicionaDiasEmUmaData(dataDeReferencia, 5);
        assertEquals(dataDaqui5Dias, dataObtida);
    }
}
```

Figura 1: Exemplo de teste automatizado com o JUnit 4

Uma característica comum a muitos arcabouços de testes que facilita ainda mais o desenvolvimento dos testes automatizados é o conjunto de ferramentas auxiliares que as integram com ferramentas e ambientes de programação (IDE), facilitando a execução da bateria de testes e a leitura dos resultados obtidos. Na Figura 2, vemos o relatório gerado com os resultados dos testes executados dentro da IDE Eclipse[9]. O relatório contém uma barra que fica verde quando todos os testes passam com sucesso, ou vermelha quando pelo menos um caso de teste falha. Para facilitar a localização de erros, é impressa a pilha de execução apenas dos testes que falharam.

3 SUnit: http://sunit.sourceforge.net
4 JUnit: http://www.junit.org
5 TestNG: http://testng.org
6 JSUnit: http://www.jsunit.net
7 CppTest: http://cpptest.sourceforge.net
8 csUnit: http://www.csunit.org
9 Eclipse: http://www.eclipse.org

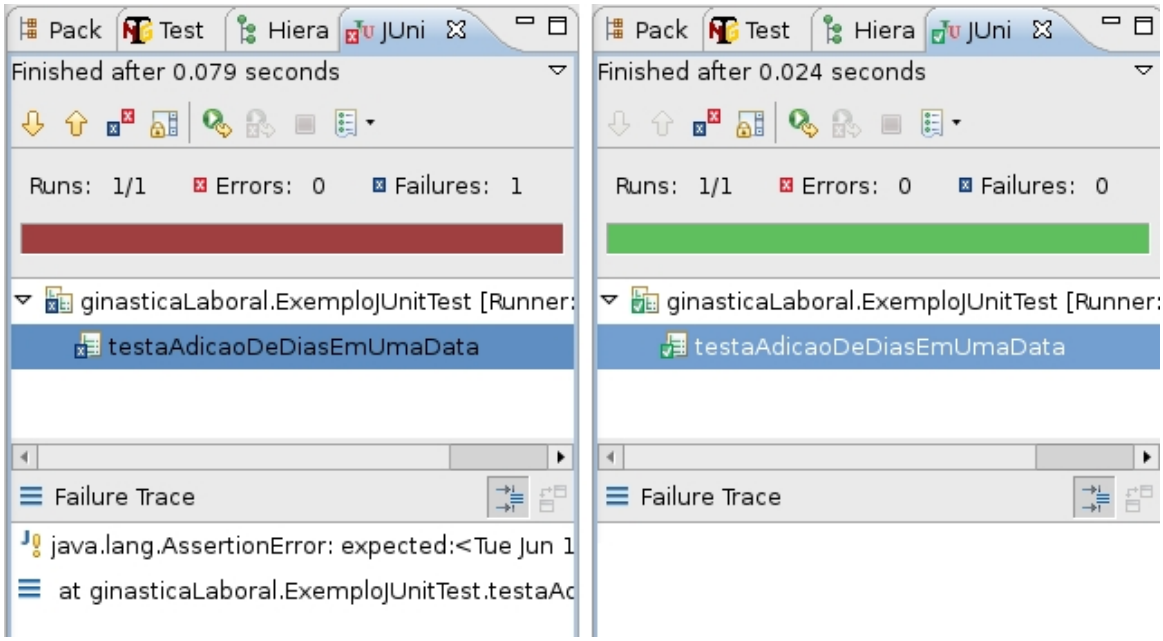

Figura 2: Relatório dos testes do plug-in do JUnit para a IDE Eclipse

O arcabouço é bem simples, assim como os códigos dos testes devem ser. Com esta noção básica da ferramenta já é possível ao leitor criar seus próprios testes automatizados para seus projetos. No entanto, é recomendado um estudo um pouco mais profundo para que seus testes tenham mais qualidade. Duas características fundamentais para garantir a qualidade dos testes são: simplicidade e legibilidade.

O código do teste, assim como o código do sistema, pode conter erros; por isso, é imprescindível que o código do teste seja o mais simples possível. Evite testes longos, código repleto de condições (`if`s), testes responsáveis por muitas verificações e também nomes obscuros.

É também importante que a legibilidade dos testes seja boa para facilitar a sua manutenção e para se obter pistas explícitas de qual funcionalidade está quebrada quando um teste falhar. Além disso, testes claros e legíveis podem ser utilizados como documentação do sistema ou como base para gerar a documentação através dos relatórios dos resultados. A Figura 3 contém um exemplo de um teste para um algoritmo que recebe um inteiro representando uma quantidade de segundos (enviado ao construtor da classe `Horario`) e devolve uma `String` representando um *display* digital (método `formatoDeRelogioDigital`).

```java
@Test
public void displayDeSegundosDeveExibirDe00Ate59() {
	assertEquals("00:00:00", new Horario(0).formatoDeRelogioDigital());
	assertEquals("00:00:01", new Horario(1).formatoDeRelogioDigital());
	assertEquals("00:00:05", new Horario(5).formatoDeRelogioDigital());
	assertEquals("00:00:59", new Horario(59).formatoDeRelogioDigital());
	assertTrue(new Horario(60).formatoDeRelogioDigital().endsWith(":00"));
	assertTrue(new Horario(61).formatoDeRelogioDigital().endsWith(":01"));
}

@Test
public void displayDeMinutosDeveExibirDe00Ate59() {
	assertEquals("00:01:00", new Horario(60).formatoDeRelogioDigital());
	assertEquals("00:01:01", new Horario(61).formatoDeRelogioDigital());
	assertEquals("00:02:00", new Horario(120).formatoDeRelogioDigital());
	assertEquals("00:02:59", new Horario(179).formatoDeRelogioDigital());
	assertEquals("00:03:00", new Horario(180).formatoDeRelogioDigital());
	assertEquals("00:59:59", new Horario(59 * 60 + 59).formatoDeRelogioDigital());
	assertTrue(new Horario(60*60).formatoDeRelogioDigital().endsWith("00:00"));
	assertTrue(new Horario(60*60 + 1).formatoDeRelogioDigital().endsWith("00:01"));
	assertTrue(new Horario(60*60 + 120).formatoDeRelogioDigital().endsWith("02:00"));
}

@Test
public void displayDeHorasDeveExibirDe00Ate23() {
	assertEquals("00:59:59", new Horario(3599).formatoDeRelogioDigital());
	assertEquals("01:00:00", new Horario(3600).formatoDeRelogioDigital());
	assertEquals("01:00:01", new Horario(3601).formatoDeRelogioDigital());
	assertEquals("01:10:01", new Horario(4201).formatoDeRelogioDigital());
	assertEquals("02:00:01", new Horario(7201).formatoDeRelogioDigital());
	assertEquals("02:26:01", new Horario(7201 + 26 * 60).formatoDeRelogioDigital());
	assertEquals("23:59:59", new Horario(24 * 3600 - 1).formatoDeRelogioDigital());
	assertEquals("00:00:00", new Horario(24 * 3600).formatoDeRelogioDigital());
}

@Test
public void displayDeveReiniciarSeAQuantidadeDeSegundosForMaiorQueDeUmDiaCompleto() {
	assertEquals("00:00:01", new Horario(24 * 3600 + 1).formatoDeRelogioDigital());
	assertEquals("00:01:00", new Horario(24 * 3600 + 60).formatoDeRelogioDigital());
}

@Test
public void displayQuantidadeDeSegundosNegativaDeveExibirDisplayZerado() {
	assertEquals("00:00:00", new Horario(-1).formatoDeRelogioDigital());
	assertEquals("00:00:00", new Horario(-100).formatoDeRelogioDigital());
}
```

Figura 3: bateria de testes para a classe `Horario`

Exemplo de Teste de Inteface Web

Outro tipo de teste importante é o de aceitação, que visa a verificar se o que foi implementado atende corretamente ao que o cliente esperava, ou seja, validar o sistema do ponto de vista do cliente. Normalmente, estes testes são realizados através da interface de usuário, que pode ser, por exemplo, um console textual, uma interface de uma aplicação local ou uma interface Web. A escrita deste tipo de teste exige mais do que chamadas de métodos e procedimentos. Para se testar uma funcionalidade é necessário a simulação das ações de um usuário interagindo com o programa, isto é, um clique do mouse, uma tecla pressionada, uma opção selecionada, entre outras ações. Por isso é fundamental a utilização de arcabouços que consigam abstrair as ações de usuário encapsulando o funcionamento interno das interfaces para facilitar a escrita e manutenção dos testes de aceitação.

Este é um tipo de teste bem abrangente, pois envolve todas as camadas do sistema, dependendo da modelagem correta da base de dados, do funcionamento correto dos módulos internos do sistema, da integração entre eles e da interação do usuário com a interface. Portanto,

escrever bons testes de aceitação que verifiquem adequadamente todos estes componentes é uma tarefa não-trivial que exige um bom conhecimento e experiência.

A Figura 4 ilustra um exemplo de teste automatizado de uma interface Web com o auxílio das ferramentas Selenium[10], Selenium Remote Control (Selenium RC[11]) e JUnit. O Selenium, diferentemente de outras ferramentas, permite executar os testes diretamente nos navegadores mais populares (Firefox, Internet Explorer, Opera, Safari), o que torna possível testar facilmente a compatibilidade de um projeto Web às diferentes plataformas. Já o Selenium RC permite que o código dos testes seja escrito em várias linguagens de alto nível, por exemplo, Java, Ruby, Python ou C#. O JUnit recebe as informações obtidas do navegador e compara com os valores esperados.

```
@Test
public void testaPaginaDoProjetoQualipso() throws Exception {
        String url = "http://www.qualipso.org";
        String servidor = "localhost";
        String porta = 4444;
        String navegador = "*firefox";
        Selenium selenium = new DefaultSelenium(servidor, porta, navegador, url);

        selenium.start(); // Abre o navegador

        // Entra no site
        selenium.open("/");
        // Verifica se um texto apareceu
        assertTrue(selenium.isTextPresent("It is all about trust"));
        // Clica em um link
        selenium.click("link=Work Areas");
        // Espera carregar a nova página
        selenium.waitForPageToLoad("30000");
        // Verifica se apareceu um outro texto
        assertTrue(selenium.isTextPresent("TRUSTWORTHY RESULTS"));

        selenium.stop(); // Fecha o navegador
}
```

Figura 4: Exemplo de teste com Selenium-RC e JUnit

A classe `DefaultSelenium` possui métodos prontos que abstraem o funcionamento interno de eventos dos navegadores, tais como `click` ou `select` que simulam um clique do mouse em um objeto e a seleção de uma opção em caixas de seleção, respectivamente. Todos os métodos que fazem uma interação com um objeto em particular exigem um argumento que permite a identificação do objeto. Este argumento pode ser expressões DOM[12] ou XPath[13], ou simplesmente um identificador definido no código HTML. Desta forma, o código do teste passa a ser um exemplo de uso do sistema.

Considerações Finais

Desenvolvimento de software é uma tarefa complexa que exige conhecimento técnico, organização, atenção, criatividade e também muita comunicação. É previsível que, em alguns momentos do desenvolvimento, em algumas das milhares de linhas de código, alguns destes requisitos falhe, mas é imprevisível o momento no qual eles irão falhar. Por isso, é imprescindível que exista uma maneira fácil e ágil de executar todos os testes em qualquer instante, e isso só é

10 Selenium: http://selenium.openqa.org
11 Selenium RC: http://selenium-rc.openqa.org
12 DOM: Document Object Model é uma linguagem para permitir que programas acessem e modifiquem dinamicamente objetos de um documento HTML.
13 XPath: XML Path Language é uma linguagem para identificar trechos específicos de um arquivo XML: http://www.w3.org/TR/xpath

viável com o auxílio de testes automatizados.

A automação dos testes dá segurança à equipe para fazer alterações no código, seja por manutenção, refatoração ou até mesmo para adição de novas funcionalidades. Além disso, representar casos de teste através de programas possibilita a criação de testes mais elaborados e complexos, que poderão ser repetidos identicamente inúmeras vezes e a qualquer momento.

Conseqüentemente, a quantidade de tempo gasto com a verificação do sistema aumenta, enquanto diminui o tempo gasto com a identificação e correção de erros, já que eles tendem a ser encontrados mais cedo. À medida que o tempo passa, o sistema vai crescendo e os programadores vão esquecendo certos detalhes de implementação, por isso quanto mais tarde um erro for encontrado, mais trabalhosa será a depuração para a sua localização.

Com essas noções básicas de testes automatizados, o leitor pode se aprofundar no assunto pesquisando a automação de outros tipos de testes, estudando padrões de escrita de bons testes automatizados e o Desenvolvimento Dirigido por Testes, que é uma técnica na qual os testes guiam a implementação do sistema. Outro tópico importante é a execução automática dos testes que pode ser feita com o auxílio de ferramentas que ficam constantemente verificando se um código foi alterado ou com ferramentas que executam a bateria de testes periodicamente.

Referências

- **AgilCoop:** Possui artigos, palestras, *podcasts* e vídeos relacionados a Métodos Ágeis www.agilcoop.org.br
- **QualiPSo:** Possui material de pesquisa de software livre, incluindo artigos relacionados a testes automatizados. www.qualipso.org
- Beck, K.; Andres C. **Extreme Programming Explained: Embrace Change**. Segunda edição. Ed. Addison-Wesley, 2004.
- Poppendieck M.; Poppendieck T. **Lean Software Development: An Agile Toolkit**. Ed. Addison-Wesley, 2003
- Schwaber, K.; Beedle, M. **Agile Software Development with SCRUM.** Ed. Prentice Hall, 2001.
- Beck, K. **Test-Driven Development: By Example**. Ed. Addison-Wesley Professional, 2002
- Delamaro, M.; Maldonado, J.; Jino, M. **Introdução ao Teste de Software**. Ed. Campus, 2007

Paulo Cheque Bernardo é formado em Ciência da Computação pelo Instituto de Matemática e Estatística da Universidade de São Paulo, está atualmente cursando mestrado na área de métodos ágeis e testes automatizados com ferramentas de software livre com apoio do projeto QualiPSo (http://www.qualipso.org) que é uma aliança entre Brasil, China e Europa para estudos em software livre.

Fabio Kon é PhD em Ciência da Computação pela Universidade de Illinois em Urbana-Champaign e atua há 20 anos com desenvolvimento de sistemas software. É professor Livre-Docente do Departamento de Ciência da Computação do IME-USP e, atualmente, está engajado na criação do Centro de Competência em Software Livre da USP (http://ccsl.ime.usp.br). Fabio escreveu seu primeiro teste automatizado em 1996 e não se arrepende disso.