# Aula 17: Estudo de Caso: JUnit

O framework JUnit o qual você tem usado para testar seu próprio código no curso 6170 merece, ele próprio, ser estudado. Ele foi desenvolvido por Kent Beck e Erich Gamma. Beck é um notório conhecido por seu trabalho com padrões e com a Programação XP (*Extreme Programming*); Gamma é um dos autores de um dos mais famosos livros de padrões de projeto. JUnit tem código livre, portanto, você pode estudar seu código fonte por si próprio. Há também um ótimo artigo explicativo na distribuição do JUnit, intitulado '*A Cook's Tour*', que explica o projeto do JUnit em termos de padrões de projeto, e sobre o qual muito do material desta aula se baseia.

O JUnit tem sido um grande sucesso. Martin Fowler, um astuto e pragmático defensor dos padrões e da programação XP (e também autor do maravilhoso livro sobre modelos de objetos denominado *Analysis Patterns*) menciona a respeito do JUnit:

> *Nunca no campo do desenvolvimento de software se deveu tanto e por tantos para tão poucas linhas de código.*

Em grande parte, o JUnit é fácil de ser utilizado, sem dúvida, por razão de sua popularidade. Pode-se pensar que, já que não faz muita coisa - ele só roda um grupo de testes e reporta seus resultados - JUnit deve ser bem simples. De fato, seu código é um tanto complicado. A razão principal para a sua complexidade é que ele foi projetado como um framework, para ser estendido de diversas formas não previstas, portanto, ele é cheio de padrões complexos e generalizações projetadas para permitir que um desenvolvedor sobreponha algumas de suas partes enquanto preserva outras.

Uma outra influência que adicionou complicação ao projeto foi o desejo de tornar a escrita de testes uma atividade simples. Foi utilizada uma técnica bem engenhosa (um 'hack') envolvendo a técnica de reflexão, que transforma métodos de uma classe em instâncias individuais do tipo ***Test***.

Também foi utilizada uma outra técnica que, à primeira vista, parece inaceitável. A classe abstrata ***TestCase*** herda da classe ***Assert***, que contém um punhado de métodos estáticos de certificação, simplesmente, para que a invocação do método *assert* fosse apenas o comando *assert(...)*, ao invés de *Assert.assert(...)*. De nenhuma forma ***TestCase*** é um subtipo de ***Assert***, é claro, portanto esta estruturação realmente não faz sentido. Mas ela permite que código pertencente a ***TestCase*** seja escrito de maneira mais sucinta. E como todos os casos de teste que o usuário escreve são métodos da classe ***TestCase***, este fato é bastante significativo.

A utilização de padrões é uma atividade de muita perícia e muito justificável. Os padrões chaves que iremos analisar são: *Template Method*, o padrão chave da programação de frameworks;

*Command*, *Composite*, e *Observer*. Todos estes padrões são amplamente explicados em Gama et al, e, com exceção de *Command*, já foram abordados neste curso.

Minha opinião pessoal é que o próprio JUnit, a jóia da programação XP, fere a mensagem fundamental do movimento XP - a de que código sozinho é suficiente para sua compreensão. O JUnit é um exemplo perfeito de programa que é quase incompreensível sem que algumas representações globais do projeto sejam explicadas de maneira que se possa entender como elas se encaixam. O fato de que o código possui poucos comentários não ajuda - e onde existem comentários, eles são extremamente obscuros. O artigo '*Cook's Tour*' é essencial; sem ele, seriam necessárias várias horas para se compreender as sutilezas do que está acontecendo no código. Além disso, seria de grande ajuda se tivéssemos mais representações do projeto. O artigo apresenta uma visão simplificada e, eu mesmo, tive que construir um modelo de objeto explicando, por exemplo, como o esquema de *listeners* (os métodos que respondem a eventos de interface) funciona.

Se você é um daqueles estudantes que não acredita em representações de projeto, e que ainda pensa que código é tudo o que importa, você deveria parar de ler por aqui, e se enclausurar em algum quarto para dedicar uma manhã inteira ao código do JUnit. Com isto, talvez você mude de idéia...

Você pode fazer o download do código fonte e da documentação do JUnit no endereço:

***http://www.junit.org/.***

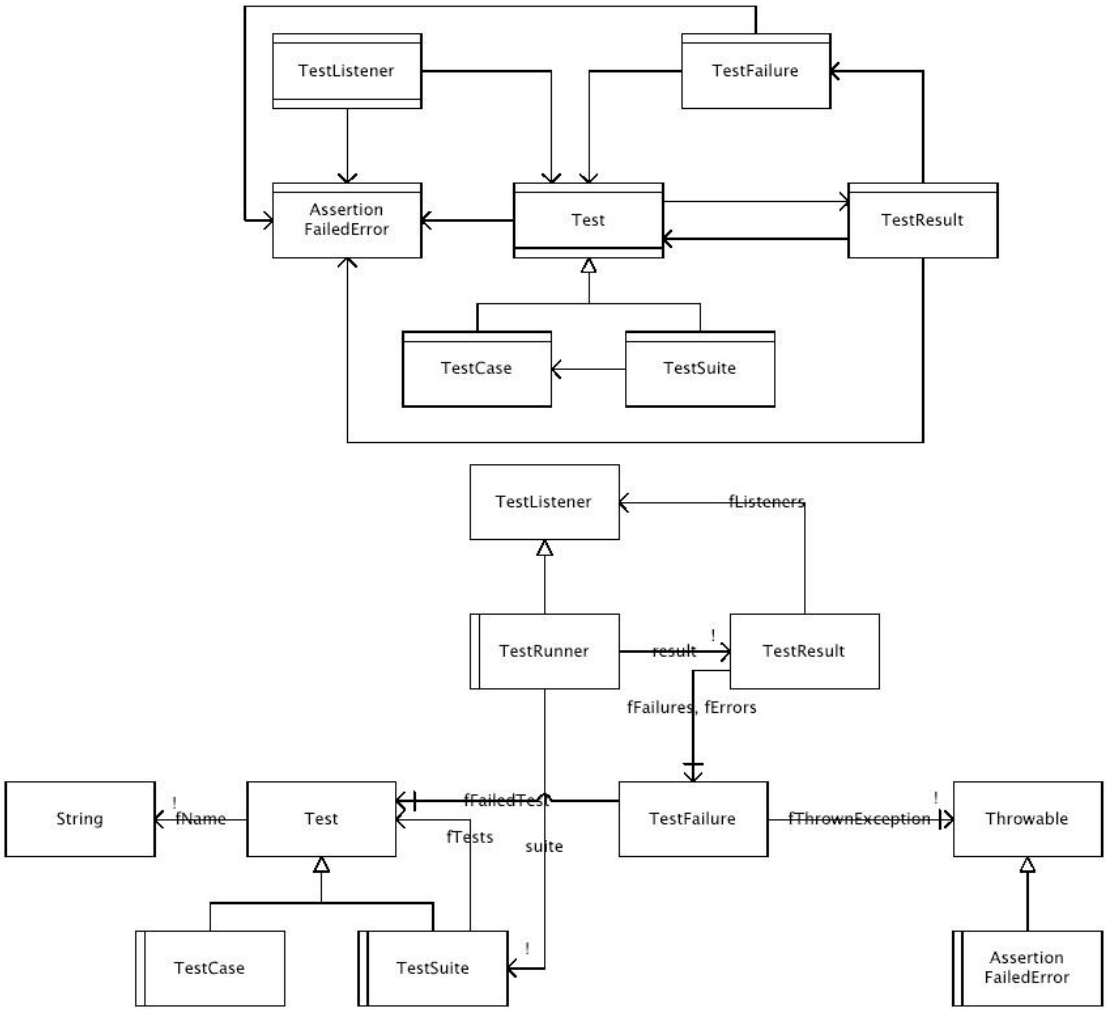
Há um repositório de código livre no endereço:

***http://sourceforge.net/projects/junit/***

onde pode-se ver (e contribuir) com descobertas de bugs.

## 17.1 Visão Geral

O JUnit possui diversos packages: *framework* como pacote básico do framework, *runner* para algumas classes abstratas e para a execução de testes, *textui* e *swingui* para interfaces de usuário, e *extensions* para algumas contribuições úteis ao framework. Iremos focar no pacote framework. Os diagramas abaixo apresentam o modelo de objeto e o diagrama de dependência modular. Pode ser necessário acompanhar estes diagramas ao passo que você lê nossa discussão. Ambos os diagramas incluem apenas os módulos do framework, no entanto, eu inclui *TestRunner* no modelo de objeto para demonstrar como os *listeners* são conectados; suas relações *suite* e *result* são variáveis locais de seu método *doRun*.




Perceba que o diagrama de dependência modular está quase todo conectado. Isto não é surpreendente para um framework; não se supõe que módulos sejam utilizados independentemente.

## 17.2 O Padrão *Command*

O padrão ***Command*** encapsula uma função como um objeto. É dessa forma que se implementa um fechamento (closure, ou clausura) - lembra-se do curso 6001? - em uma linguagem orientada a objetos. A classe de comando, tipicamente, possui um único método com um nome do tipo *do*, *run* ou *perform*. Uma instância de uma subclasse é criada sobrepondo este método, encapsulando também, normalmente, algum estado da classe (na linguagem do curso 6001, o ambiente do fechamento). O comando pode então ser passado como um objeto, e executado invocando-se o método.

No JUnit, casos de teste são representados através de objetos comando que implementam a interface ***Test***:

```
public interface Test {
        public void run();
}
```

Casos de teste verdadeiros são instâncias de uma subclasse de uma classe concreta ***TestCase***:

```
public abstract class TestCase implements Test {
        private String fName;
        public TestCase(String name) {
                fName= name;
        }
        public void run() {
                …
        }
}
```

De fato, o código verdadeiro não é muito parecido com isto, mas partir desta versão simplificada irá nos permitir explicar os padrões básicos mais facilmente. Perceba que o construtor associa um nome ao caso de teste, o qual será útil quando os resultados forem reportados. De fato, todas as classes que implementam ***Test*** possuem esta propriedade, portanto, pode ter sido uma boa idéia ter-se adicionado o método

```
public String getName ()
```

à interface ***Test***. Perceba também que os autores do JUnit utilizam a convenção de que identificadores iniciados por um *f* minúsculo são campos de uma classe (isto é, variáveis de instância). Veremos um exemplo mais elaborado do padrão ***Command*** quando estudarmos o programa Tagger mais adiante no curso.

## 17.3 O Método *Template*

Pode-se determinar que *run* (da interface ***Test***) seja um método abstrato, que exige, portanto, que todas as subclasses o sobreponham. Mas a maioria dos casos de teste possuem três fases: determinação do contexto, execução do teste e, então, desmontagem do contexto. Podemos automatizar a utilização desta estrutura fazendo com que *run* seja um método ***template***.

```
public void run() {
        setUp();
        runTest();
        tearDown();
}
```

As implementações padrão dos métodos hook (gancho, como visto na última aula), invocados por *run*, não fazem nenhum processamento:

```
protected void setUp() { }
protected void runTest() { }
protected void tearDown() { }
```

Eles são declarados como ***protected*** de forma que sejam acessíveis a partir de subclasses (podendo, portanto, serem sobrepostos), mas que não sejam acessíveis de fora do package. Seria ótimo restringir o acesso apenas às subclasses, mas o Java não oferece este modo. Uma subclasse pode arbitrariamente sobrepor estes métodos; se sobrepuser apenas *runTest*, por exemplo, não haverá nenhum comportamento especial dos métodos *setUp* ou *TearDown*.

Observamos este mesmo padrão na última aula, nas implementações organizadas em hierarquias de esqueleto da API de coleções do Java. Este padrão, às vezes, é referenciado de uma forma um tanto quanto brincalhona como o '*Princípio de Hollywood*': uma API tradicional fornece métodos que podem ser chamados pelo cliente; um framework, em contraste, faz chamadas aos métodos de seu cliente: 'não nos chame, iremos chamar você'.

A utilização cada vez mais abrangente dos ***templates*** é a essência da programação de frameworks. Seu uso é bastante poderoso, mas os frameworks tornam possível, também, códigos escritos de maneira completamente incompreensível, pois as implementações realizam chamadas em todos os níveis da hierarquia de herança.

Pode ser difícil saber o que esperar de uma subclasse em um framework. Uma analogia das cláusulas *pre-conditions* e *pos-conditions* não foi desenvolvida, e o estado da arte ainda é um tanto imaturo. Normalmente, você deve ler o código fonte do framework para utilizá-lo eficientemente. A API de coleções do Java é melhor do que a maioria dos frameworks, pois inclui, nas especificações dos métodos ***template***, descrições cuidadosas de como eles são implementados. Isto pode ser considerado uma afronta à idéia de especificação abstrata, mas é inevitável no contexto de um framework.

## 17.4 O Padrão *Composite*

Como discutimos na aula 11, casos de teste são agrupados em suítes de teste. Mas aquilo que se faz com um suíte de testes é essencialmente a mesma coisa que se faz com um teste: você o executa, e reporta o resultado. Isto nos sugere o utilizar o padrão ***Composite***, no qual um objeto composto compartilha sua interface com seus componentes elementares.

Aqui, a interface é a interface ***Test***, o objeto composto (ou objeto ***composite***) é ***TestSuite***, e os componentes elementares são membros de ***TestCase***. ***TestSuite*** é uma classe concreta que implementa ***Test***, mas cujo método *run*, diferente do método *run* de ***TestCase***, invoca o método *run* de cada um dos testes que o suíte de testes contém. Instâncias de ***TestCase*** são adicionadas à instância de ***TestSuite*** através do método *addTest*; há também um construtor que cria um ***TestSuite*** como um grupo de casos de teste, como veremos mais tarde.

No exemplo de ***Composite*** do livro de Gamma a interface inclui todas as operações do objeto composto. Seguindo esta abordagem, ***Test*** deveria incluir métodos como *addTest*, que se aplicam somente a objetos ***TestSuite***. A sessão de implementação da descrição do padrão explica que há uma troca entre transparência - fazer com que o objeto composto e seus objetos folha (objetos que estão ligados) mostrem-se da mesma forma - e segurança - prevenção de invocações a operações não apropriadas. Nos termos da discussão de nossa aula sobre subtipagem, questiona-se se a interface deveria ser um verdadeiro supertipo. Na minha opinião, deveria ser, pois os benefícios da segurança são maiores do que os benefícios da transparência, e, além disso, a inclusão das operações compostas na interface gera confusão. O projeto JUnit segue esta abordagem, e não incluiu *addTest* em sua interface ***Test***.

## 17.5 O Padrão Parâmetro de Coleta

O método *run* de ***Test***, na verdade, possui esta assinatura:

***public void run(TestResult result);***

Ele recebe um único argumento que é alterado para registrar o resultado do código executado. Beck chama esta técnica de 'Parâmetro de coleta', e a vê como um padrão de projeto da sua própria maneira.

Existem duas maneiras através das quais um teste pode falhar. Ou o teste produz o resultado errado (o que pode incluir não jogar a exceção esperada), ou joga uma exceção não esperada (tal como *IndexOutOfBoundsException*). JUnit denomina o primeiro caso de '*failure*', ou falha, e o segundo de '*error*', ou erro. Uma instância de ***TestResult*** contém uma seqüência de falhas e uma seqüência de

erros, sendo que cada falha ou erro é representado como uma instância da classe ***TestFailure***, que contém uma referência para um ***Test*** e uma referência para o objeto de exceção gerado pela falha ou erro. (falhas sempre produzem exceções, pois mesmo quando um resultado inesperado é produzido sem uma exceção, o método *assert* utilizado no teste converte a falha em uma exceção).

O método *run* de ***TestSuite*** é, essencialmente, inalterado; ele apenas passa um objeto ***TestResult*** quando invocar o método *run* de cada um de seus testes. O método *run* de ***TestCase*** se parece com alguma coisa do tipo:

```
public void run (TestResult result) {
        setUp ();
        try {
                runTest ();
        }
        catch
                (AssertionFailedError e) {
                        result.addFailure (test, e);
                }
                (Throwable e) {
                        result.addError (test, e);
                }
        tearDown ();
}
```

Na realidade, o controle do fluxo do método template *run* é mais complicado do que sugerimos no código acima. No código abaixo são apresentados alguns fragmentos de pseudocódigo demonstrando o que ocorre. No esquema apresentado o método *run* ignora as atividades *setUp* e *tearDown*, e contém uma utilização do ***TestSuite*** dentro de uma interface de usuário de texto:

```
junit.textui.TestRunner.doRun (TestSuite suite) {
        result = new TestResult ();
        result.addListener (this);
        suite.run (result);
        print (result);
}
junit.framework.TestSuite.run (TestResult result) {
        forall test: suite.tests
                test.run (result);
```

```
}
junit.framework.TestCase.run (TestResult result) {
        result.run (this);
}

junit.framework.TestResult.run (Test test) {
        try {
                test.runBare ();
        }
        catch (AssertionFailedError e) {
                addFailure (test, e);
        }
        catch (Throwable e) {
                addError (test, e);
        }
}
junit.framework.TestCase.runBare (TestResult result) {
        setUp();
        try {
                runTest();
        }
        finally {
                tearDown();
        }
}
```

***TestRunner*** é uma classe de interface de usuário (*user interface - ui*) que invoca o framework e exibi os resultados. Há uma versão com interface gráfica de usuário ***junit.swingui*** e uma versão simples com terminal de texto ***junit.textui***, da qual apresentamos um trecho. Iremos apresentar o sistema de *listener* mais tarde; ignore isso por enquanto. Veja como funciona o código acima: o objeto ***TestRunner*** cria um novo ***TestResult*** para armazenar os resultados do teste; o ***TestRunner*** executa a suíte de testes, e exibe os resultados. O método *run* de ***TestSuite*** invoca o método *run* de cada um de seus testes constituintes; que podem, eles próprios, serem suítes de testes (objetos do tipo ***TestSuite***), portanto, o método pode ser chamado recursivamente. Este é um ótimo exemplo da simplicidade que a técnica de ***Composite*** fornece. Eventualmente, como há uma invariante que determina que um ***TestSuite*** não pode conter a si mesmo - que, na verdade, não está especificada e que também não está definida pelo código de ***TestSuite*** - o método terminará por invocar os métodos *run* de objetos do tipo ***TestCase***.

Agora, no método *run* de ***TestCase***, o objeto receptor ***TestCase*** troca de lugar com o objeto ***TestResult***, e invoca o método *run* de ***TestResult*** com o ***TestCase*** como um argumento. (Por que?). O método *run* de ***TestResult***, então, invoca o método *runBare* de ***TestCase*** que, na verdade, é o método ***template*** que executa o teste. Se o teste falha, ele joga uma exceção, que é interceptada pelo método *run* de ***TestResult***, que, então, empacota o teste e a exceção como uma falha ou um erro do ***TestResult***.

## 17.6 O Padrão *Observer* (observador)

Para uma interface de usuário interativa, gostaríamos de mostrar os resultados do teste incrementalmente ao passo que ele é executado. Para conseguir isto, JUnit utiliza o padrão ***Observer***. A classe ***TestRunner*** implementa uma interface ***TestListener*** que possui métodos *addFailure* e *addError* próprios. A interface faz o papel de ***Observer***, ou observador. A classe ***TestResult*** faz o papel de *Subject*, isto é, o sujeito observado. ***TestResult*** fornece o método

```
public void addListener(TestListener listener)
```

que adiciona um observador. Quando o método *addFailure* de ***TestResult*** é invocado, além de atualizar sua lista de falhas, o método invoca o método *addFailure* em cada um de seus observadores:

```
public synchronized void addFailure(Test test, AssertionFailedError e) {
        fFailures.addElement(new TestFailure(test, e));
        for (Enumeration e= cloneListeners().elements(); e.hasMoreElements(); ) {
                ((TestListener)e.nextElement()).addFailure(test, e);
        }
}
```

Na interface de usuário textual, o método *addFailure* de ***TestRunner*** simplesmente exibe um caractere F na tela. Na interface gráfica de usuário, este método adiciona a falha a uma lista de exibição e altera a cor da barra de progresso para vermelho.

## 17.7 O Hack Reflexão

Recorde-se que um caso de teste é uma instância da classe ***TestCase***. Para criar um suíte de testes em Java puro e simples, um usuário teria que criar uma nova subclasse de ***TestCase*** para cada caso de teste, e instanciá-la. Uma forma elegante de se fazer isso é através de classes internas anônimas, ou *inner classes*, criando-se o caso de teste como uma instância de uma subclasse que não possui

nome. O que, ainda, é muito trabalhoso. No projeto do JUnit, ao invés disso, é utilizada uma técnica bastante engenhosa (o hack, ou técnia engenhosa, denominado ‘Reflexão’).

O usuário fornece uma classe para cada suíte de testes - denominada, digamos, ***MySuite*** - que é uma subclasse de ***TestCase***, e que contém muitos métodos de teste, cada um dos quais possuindo um nome iniciado com a string 'test'. Estas classes são tratadas como casos de teste individuais.

```
public class MySuite extends TestCase {
        void testFoo () {
                int x = MyClass.add (1, 2);
                assertEquals (x, 3);
        }
        void testBar () {
                …
        }
}
```

A classe objeto ***MySuite*** é passada, ela própria, para o construtor de ***TestSuite***. Através de reflexão, o código de ***TestSuite*** intancia ***MySuite*** para cada um dos seus métodos que são iniciados pelos caracteres 'test', passando os nomes dos métodos como um argumento para o construtor. Como resultado, para cada método de teste, um novo objeto ***TestCase*** é criado, com seu nome atado ao nome do método de teste. O método *runTest* de ***TestCase*** invoca, de novo através de reflexão, o método cujo nome corresponde ao nome do próprio objeto ***TestCase***, mais ou menos assim:

```
void runTest () {
        Method m = getMethod (fName);
        m.invoke ();
}
```

Este esquema é obscuro, e perigoso, e não é o tipo de coisa que você deve imitar no seu código. No código do JUnit este esquema é justificado, pois é limitado a apenas uma pequena parte do código trazendo uma enorme vantagem para o usuário de JUnit.

## 17.8 Questões para Estudo

Estas questões surgiram quando eu construí o modelo de objeto para JUnit. Nem todas possuem respostas únicas.

· Por que os *listeners* fazem parte do ***TestResult***? O ***TestResult*** não é, ele próprio, uma espécie de *listener*?
· É possível que um ***TestSuite*** não contenha nenhum teste? Ele pode conter a si mesmo?
· Os nomes de testes são únicos?
· O campo *fFailedTest* de ***TestFailure*** sempre aponta para um ***TestCase***?