Dicas JUnit e Lab1 - CES-28

(Last Edit 2017/07/25 15:27)

Cada aluno pode procurar manuais e dicas sobre o JUnit. Mas neste documento:

- coletei informações que precisei (ou curiosidades que quis responder) durante a minha implementação do Lab1.
- Coletei respostas às perguntas dos alunos e correções aos erros dos mesmos.

Portanto, já que agora vocês conhecem todas essas coisas, posso corrigir com mais rigor, certo?

Como implementar bons testes: conceitos

Failure x Error

Failure é uma falha do teste, ou seja, o teste foi executado e uma assertion deu false.
Errors indica erros na execuçao do teste, ou seja, o teste em si não pode ser executado.
http://stackoverflow.com/questions/3425995/whats-the-difference-between-failure-and-error-in-junit

Varios testes, melhor que um super teste

Havendo um bug, o JUnit te indica qual teste quebrou. Se o teste eh pequeno e faz relativamente poucas coisas, já sabemos o que quebrou. Mas se falha um super teste com zilhoes de asserts verificando zilhoes de coisas diferentes, temos que verificar a linha da falha, ler o codigo, verificar o que exatamente estava sendo testado naquele momento,etc..

O importante não é tanto contar os asserts (embora muitos asserts sejam um mal sinal), mas, conceitualmente, cada teste deve testar um “caso”, indicado pelo título do metodo

Funcionalidades que envolvem Colecoes devem ser testadas

- caso vazio, inclusive, se ha construtor sem parametros, se a colecao nasce com zero elementos. Por exemplo, se o moneybag esta vazio ao ser criado.
- caso unitario - funciona com um elemento?
- caso geral

Nao precisa apagar testes só porque eles são simples. Quando precisar refatorar, refatore, e mantenha o teste

No comeco, somamos 20 USD com 10 USD e temos 30 USD. ok. É o primeiro teste.
Depois, precisamos criar Currency, MoneyBag, somar moedas diferentes, etc.
Então o teste anterior para de funcionar, porque o construtor mudou.
Entao refatore (corrija) o teste e mantenha ele funcionando! Mantenha um teste apenas com uma soma simples.

Toda a ideia da ferramenta eh ter inumeros testes simples funcionando sempre.
Depois, somaremos moedas diferentes, ok. Mas ainda queremos saber se 20 USD + 10 USD continuam resultando em 30 USD!

Tive ja a experiencia de ver testes simples quebrarem ao longo do tempo conforme o codigo era modificado. As vezes um bug simples, que poderia passar despercebido, eh detectado por um teste simples, e isso ja indica que eh é um bug basico e nao algo mais complexo que so afeta os testes mais complicados.

Um teste pode se tornar redundante, mas ele não é redundante apenas por ser um caso mais simples ou por estar implícito na execução de outro teste.

Lei de Demeter: veremos com mais detalhes o que eh

basicamente evite longas sequencias de “.”
compare:

```
// errado
moneybag.listamoneys.get(1).getCurrency().getString().equals( “CHF”) &&
moneybag.listamoneys.get(1).getCurrency().getAmount()== 10
//   isso nao cheira mal???  cade o encapsulamento ?

// correto, onde CHF10 eh variavel criada no text fixture
moneybag.hasMoney(CHF10);  // curtinho! Dificil de errar, facil de manter
```

A primeira alternativa:
(a) quebra o encapsulamento expondo estrutura de dados internas;
(b) faz o codigo depender de constantes (no caso, ‘1’), facil de errar, dificil de manter
(c) torna o teste dependente da interface de varios objetos internos a moneybag. Fica dificl de manter, pois se moneybag usar outra estrutura de dados, talvez tenhamos que mudar o teste.
(d) eh menos legivel, mais longo.
(e) pode testar aspectos irrelevantes. Precisamos mesmo saber se este money eh o primeiro da lista? Ou so queremos saber se esta presente na lista?

Testes sem Assert: fora do contexto de unit testing

O contexto das ferramentas xUnit, ou de testes de unidade em qualquer metodo agil, sao **centenas de testes que devem ser instantâneos e executados a cada compilação.**
Não faz sentido um teste que nao tem nenhum assert e imprime uma saida que o humano deve ler para saber que esta correto!

Em Java eh facil comparar Strings, e com o metodo equals() é facil comparar qualquer objeto. Nao tem desculpa pra nao fazer o teste testar alguma coisa!

Todo teste deve passar com verde no JUnit, mesmo que precise soltar exceção!

Testes que poluem console: fora do contexto de unit testing

Um corolário do anterior: não podemos usar mensagens e esperar que o usuário leia, e também não faz sentido imprimir mensagens ou poluir o console com mensagens de debug. Ninguem lerá, nem entenderá nada depois do 300o teste executado!

Testes com entrada do usuário: fora do contexto de unit testing

Um corolário do anterior: são centenas de testes executados a cada compilação. Ninguem está lá para digitar entradas! Entradas devem ser simuladas, hardcoded, no próprio teste ou no text fixture.

Como implementar bons testes: JUnit

Ordem de execução: @BeforeClass, @Before, @Test

Conceito importante: uma nova instância a cada @Test

Antes de executar cada método @Test, uma nova instância da classe de teste (aquela que contém o método @Test é instanciada. Após a execução do método @Test, esta instancia é destruída.

Então a ordem considerando as anottations básicas é:

1. Executa-se o método @BeforeClass antes de todos os métodos @Test e antes de construir a classe teste, logo não pode preencher valores de atributos da classe teste aqui.
2. Para cada método @Test
    a. O construtor da classe teste é chamado e uma nova instância é criada.
    b. O metodo @Before é chamado. Aqui podemos inicializar atributos da classe teste.
    c. O método @Test é chamado e o teste realizado.
    d. O método @After é chamado e a instância da classe é destruída
3. Depois de executar (2) para todos os @Test, é chamado o metodo @AfterClass

Os métodos @BeforeClass e @AfterClass sao usados quando algum setup/cleanup demorado é necessário para todos os testes, e não se deseja repetir isso para todos os @Test. Exemplo clássico é estabelecer uma conexão com Banco de Dados.

antes de cada teste, JUnit instancia a classe Teste. Cada @Test entao recebe um objeto novo.

https://garygregory.wordpress.com/2011/09/25/understaning-junit-method-order-execution/

Oh, então porque nao usar o construtor ao invés do @Before?
http://stackoverflow.com/questions/6094081/junit-using-constructor-instead-of-before

Exemplo simples de Uso do @Before

```
class MoneyTest {
        private Money mCH20;

        @Before
        void public setup() {
                mCH20 = new Money(20,new Currency(“CHF”));
        }

        @Test
        // funcao de teste, usa mCH20
```

```
	@Test
	// outra funcao de teste, usa mCH20

}
```

Neste exemplo, a variavel mCH20 foi criada e inicializada antes de cada teste. Evitamos copy-paste, simplificamos e encurtamos o codigo, facilitamos manutencao, se usamos a mesma variavel em varios testes.

Lista de annotations

http://stackoverflow.com/questions/15760881/list-of-annotations-in-junit

Testes que dependem de Testes.

O problema: vários testes precisam de uma MoneyBag pré-inicializada com algumas moedas, e isso poderia estar em um método @Before

```
@Before
void setup() {
	// variaveis são atributos privados da classe de teste
	mb = new MoneyBag();
	m1 = new Money(10,new Currency("USD");
	m2 = … // outro money
	mb.add(m1);
	mb.add(m2);
}
```

Portanto o método @Before depende do método add()! Assim todos os testes dependem do add()! Nem o @Before funciona se houver um bug no add()! ***Como testamos o metodo add()??? Em geral, como testamos metodos simples se o proprio text fixture depende dos mesmos metodos simples?***
Resposta: podem haver varias classes de teste… Veja uma solucao elegante:

```
class TesteMoneyBagSimple {
	 // private etc….

	@Before
	// nao usa add(). Não usa nenhum metodo de MoneyBag.
	void setup() {
		// variaveis são atributos privados da classe de teste
		mb = new MoneyBag();
	}


	@Test
	// testa o add() em um caso simples
```

```
	// (e.g. adiciona mb e verifica se amount eh correto)

	@Test
	// verifica se o money bag nasceu vazio

	@Test
	// testa diretamente outros metodos e casos simples
} // classe TesteMoneyBagSimple

// Testa casos mais complexos, que dependem dos casos basicos.
// os metodos basicos como add ja foram testados em casos simples.
class TesteMoneyBagVariousMoney {
	 // private etc….

	@Before
	// supoe que metodos de MoneyBag funcionam nos casos simples.
	void setup() {
		mb = new MoneyBag();
		m1 = new Money(10,new Currency(“USD”);
		mb.add(m1);
		// …
		// inicializa varias variaveis;
		// preenche a MoneyBag com varios Money
	}

	// sequencia de testes mais complexos
	@Test
	// e.g.: testa conversao de tudo pra BRL

	@Test
	// adiciona varios Money com Currencies que ja estao no bag
	// verifica se nao ha currencies repetidas
	// um jeito facil eh testar o resultado de toString()

	@Test
	// outro teste que comeca com uma MoneyBag preenchida

} // classe TesteMoneyBagVariousMoney
```

De novo, sao executados testes mais simples e mais complexos. Se aparecer um bug, quais testes falham é uma indicação para achar o bug.
Mas isso sera raro, porque normalmente vamos implementar e passar os testes simples antes dos complexos, certo?

O JUnit tem opcoes mais complexas, mas pra esse lab nao precisa mais do que isso.

Cuidado com testes tautologicos!

Tambem deixei passar mas comparem as duas situacoes:

```
MoneyBag m1 = new MoneyBag();
m1.add(CHF10);
m1.add(CHF20);
m1.add(BRL10);

////////// situação 1
assertEquals(m1, m_expected); // testa so o equals global
// onde m_expected foi setada em @Before com os mesmos “dinheiros”

////////// situação 2
assertEquals(m1.convertBRL(), 70); // testa valor total
assertTrue(m1.hasMoney( CHF30 ) ); // testa se existe 30 CHFl
assertTrue(m1.hasMoney( BRL10 ) ); // testa se existe 10 BRL.
assertEquals(m1.getSize(), 2); // testa se temos 2 dinheiros.
// agora sabemos que nao existe um 3o dinheiro na moneybag
assertEquals(m1, m_expected); // testa o equals global
```

Em ambas as situacoes, os testes checam quantidades de cada moeda, mas o problema eh que na 1a situacao o teste eh tautologico, ou seja, eh sempre verdade. Se houver um bug, há uma boa chance do bug se repetir em m1 e m_expected e entao o resultado da comparacao m1 == m_expected será ainda true!
Na segunda os valores e quantidade de elementos da lista sao checados explicitamente!

toPrint(), equals() não são enfeites, são convenções da linguagem: facilitam a implementação e melhoram a legibilidade!

Vale a pena implementar toPrint() (ao inves de um metodo qualquer que percorre o conteudo e imprime) e equals() para fazer comparação.
Primeiro, é isso que o leitor do codigo (eu, nesse caso) espera encontrar, alem disso ele ja sabe o que significam metodos com esses nomes.
Segundo, tudo fica mais facil, curto, e legivel.
na aplicacao:

```
System.out.println(mymoneybag);
/// a chamada do metodo toPrint() eh implicita!
```

nos testes, se precisar comparar moneybags ou moneys, fica facil com equals.

```
MoneyBag mb1 = …. //inicializa mb1
// faz varias adicoes e remocoes de moneys e moneybags em mb1
// e/ou faz outras operacoes chatas que nao implementamos!
```

```
MoneyBag mb2 = …
// inicializa mb2 exatamente com os valores finais esperados.
```

Agora, se ja existe antes um teste basico com valores explicitos (vide acima), ou seja, se o metodo equals() em si mesmo ja foi testado e confiamos nele, podemos comparar moneybags diretamente com assertEqual(mb1,mb2); para o caso mais complexo.

e se money e currency tambem tem metodos equal, ao percorrer a lista de moneys a implementacao da comparacao tambem eh facil de implementar, com codugo curto e legivel.

Ou seja, fazer com que cada classe saiba se comparar e se imprimir facilita tudo! Faça do jeito facil ao invés de bater cabeça e reclamar!

http://www.wideskills.com/java-tutorial/java-tostring-method

Específico sobre o lab: FAQ

Mapeie com comentarios onde estao os testes que verificam os exercicios:

exemplo: readme.txt com

MyMoneyBagSimple.java testa o ex 1,2,3,4
MyMoneyTest.java testa o ex 6,7
MyMoneyBagTest.java testa o ex 10

E dentro do MyMoneyBagTest.java comentarios como o seguinte:

```
/************************
testes do ex 10.
*************************/

testEmpty() {
….
}

testSum_4Moneys_3WithSameCurrency() {
….
}
```

Vou procurar os testes que verificam cada exercício. Pode mudar o formato, mas quem não me ajudar a encontrar as respostas de cada exercício perde nota. Tambem é uma forma de verificar se vocês sabem o que estão fazendo.

A ideia eh voces pensarem e completarem o codigo.

Precisa comparar, logo precisa de equals()
Precisa imprimir, logo precisa de toString()
Money() pode ter um construtor mais pratico do que criar vazio e chamar os setters.
precisa adicionar Money na MoneyBag. logo precisa de um metodo pra isso, certo?
Uma MoneyBag não pode ter varios Money com a mesma moeda, ele deve somar BRL com BRL. Tem que verificar quando um novo money eh adicionado. Isso implica que precisamos testar se a quantidade de itens da lista é correta depois de várias inserções com moedas repetidas, e testar o valor de cada moeda. Quem garante que não ficou um money extra ou repetido dentro da lista?
Assim por diante, o que voces precisarem para completar os testes precisa ser feito.

Ja vi essa ferramenta encontrar bugs que eu nao vi ou veria na hora da implementacao.

Não seja minimalista! Pense nos casos normais de uso! Teste todas as funcionalidades.

Eclipse e zips para entrega

alguns se confundem com projetos, packages, diretorios, etc. Eu tento arrumar o que posso, mas aparecem alguns projetos com tudo bagunçado.

O eclipse permite exportar o projeto inteiro (inclusive escolher subdiretorios) como um zip, que ja coloca os packages em diretorios com nomes apropriados dentro do zip, etc.

Se pode incluir as imagens dos diagramas, por exemplo, dentro do projeto do eclipse (tudo fica visivel na arvore do project manager), e tudo vai junto no mesmo zip.

Facil pra voces, tudo zipado sem ter que fazer manualmente, e facil pra mim corrigir depois pois o eclise sabe importar tudo de uma vez!!!!!

Eclipse?

Estou usando Eclipse. Para usar outro, precisa me mostrar como resolver a submissao e correção. Talvez seja facil, porque o eclipse aparentemente nao cria complicacoes para abrir um diretorio com codigo. Talvez haja um jeito facil de exportar/salvar algo que consiga abrir facil. Mas precisa de um procedimento escrito e disponivel no tidia, e conversado comigo.
E depois quem nao seguir o procedimento combinado e me der trabalho extra pra corrigir perderá nota.

Basta alguns voluntarios fazerem testes, virem me mostrar, e escrever um passo-a-passo para todos.

Sobre o que fazer

>Quando adicionarmos a uma MoneyBag, mais de um Money com a mesma Currency, eles devem:
>1 - ser somados e permanecer como um unico Money daquela Currency na MoneyBag,
>2 - permancerem como Money's separados, havendo assim mais de um Money com a mesma Currency?
a alternativa 1. faz mais sentido 10 reais + 10 reais darem 20 reais, certo?

>Caso seja para fazer da segunda forma, como devemos proceder as retiradas? Ou nao eh necessario implementar essa funcionalidade?
nao precisa implementar se nao esta pedido.
Mas a ideia do moneybag seria guardar um cesto de moedas. se so existem 10 reais e 20 dolares, nao daria pra retirar 20 reais.

>duvida sobre o passo 10. tem que ser com template?
nao sei se entendi. mas eh suficiente usar polimorphismo de alguma forma, pra escolher entre qual classe retornar. nao precisa de classe com <template>, se eh isso que vc quer dizer com template.

>O senhor poderia passar uma dica sobre o item 10 do laboratório?
>(sobre mudar o método add() de Money de modo que ele retorne Money ou MoneyBag)
se Money.add receber um Money da mesma moeda, basta somar os valores.
se receber um money de outra moeda, nao da pra somar, tem que colocar as duas moedas em um moneybag. essa eh a ideia basica
mas se eh pra poder retornar as duas coisas, deve usar polimorfismo, ou seja, money e moneybag devem ter um pai comum, que pode ser uma interface.

Faça em dupla, não sozinho

Perde ponto.