Teste de Software

Prof. Dr. Fabio Kon

Departamento de Ciência da Computação
IME / USP

Técnicas de Programação 1 - 2020

Testar ≠ Depurar

- Simplificando
  - Depurar - o que se faz quando se sabe que o programa não funciona;
  - Teste - tentativas sistemáticas de encontrar erros em programa que você "acha" que está funcionando.

- "Testes podem mostrar a presença de erros, não a sua ausência (Dijkstra)"

Teste enquanto você escreve código

- Se possível escreva os testes antes mesmo de escrever o código
  - uma das técnicas de XP
- quanto antes for encontrado o erro melhor !!

Técnicas básicas

- Teste o código em seus limites
- Teste de pré- e pós-condições
- Uso de premissas (*assert*)
- Programe defensivamente
- Use os códigos de erro

Teste o código em seus limites

- Para cada pequeno trecho de código (um laço, ou if por exemplo) verifique o seu bom funcionamento

- Tente ume entrada vazia, um único item, um vetor cheio, etc.

Exemplo:

```
int i;
char s[MAX];

for(i=0; s[i] = getchar() != '\n' &&
          i < MAX - 1; i++);
s[--i]='\0';
```

Primeiro erro fácil:

```
// o = tem precedência menor do que o !=
for(i=0; (s[i] = getchar()) != '\n' &&
          i < MAX - 1; i++);
```

Exemplo:

```
int i;
char s[MAX];

for(i=0; i < MAX - 1; i++)
    if ((s[i] = getchar()) == '\n')
        break;
s[i]='\0';
```

Testes:

linha vazia ok; 1 caractere ok; 2 caracteres ok;
MAX caracteres ok

e se o primeiro caractere já é o de fim de arquivo ?

Exemplo:

```
int i;
char s[MAX];

for(i=0; i < MAX - 1; i++)
    if ((s[i] = getchar()) == '\n' || s[i]==EOF)
        break;
s[i]='\0';
```

Testes:

ok.

Mas o que se deve fazer se a string s fica cheia antes do '\n'

Depende, estes caracteres são necessários, ou não ?

Teste de pré- e pós-condições

- Verificar certas propriedades antes e depois de trechos de código

```
double avg(double a[], int n){
    int i;
    double sum = 0.0;

    for(i = 0; i < n; i++)
        sum += a[i];
    return sum / n;
}
```

- x = y = 5;

Teste de pré e pós condições

- Solução possível

```
// mudar o return
return n <= 0 ? 0.0 : sum / n;
```

- Não existe uma única resposta certa
  - A única resposta claramente errada é ignorar o erro !!
  - Ex: USS Yorktown
    - Exceção de divisão por zero ignorada no código

Uso de premissas

- Em C e C++ use <assert.h>, jdk 1.4+
  - ex:

```
assert (n>0);
```

  - se a condição for violadada:

```
Assertion failed: n>0, file
 avgtest.c, line 7.
```

- Ajuda a identificar "culpados" pelos erros

Programação defensiva

- Tratar situações que não "podem" acontecer

Exemplo:

```
if (nota < 0 || nota > 10) // não pode acontecer
    letra = '?';
else if (nota > 9)
    letra = 'A';
else ...
```

- Isso é bom para evitar coisas do tipo:

mqz@nyquist:~/Área de Trabalho$ rm AIM/

rm: imposível remover `AIM/': É um diretório

mqz@nyquist:~/Área de Trabalho$ rmdir AIM/

rmdir: falha ao remover `AIM/': Não é um diretório

Utilizar códigos de erro

- Checar os códigos de erro de funções e métodos;
  - você sabia que o scanf retorna o número de parâmetros lidos, ou EOF ?
- Sempre verificar se ocorreram erros ao abrir, ler, escrever e principalmente fechar arquivos.
- Em Java sempre tratar as possíveis exceções

Pequeno exercício:

```
int fatorial(int n) {
    int fat = 1;

    while (n--) {
        fat *= n;
    }
    return fat;
}
```

como testar isso?

Responda em www.menti.com código 25 71 26 1

Testes sistemáticos (1/4)

- Teste incrementalmente
  - durante a construção do sistema
    - após testar dois pacotes independentemente teste se eles funcionam juntos
- Teste primeiro partes simples
  - tenha certeza que partes básicas funcionam antes de prosseguir
  - testes simples encontram erros simples
  - teste as funções/métodos individualmente

Testes Sistemáticos (2/4)

- Conheça as saídas esperadas
  - conheça a resposta certa
  - para programas mais complexos valide a saída com exemplos conhecidos
    - compiladores - arquivos de teste;
    - numéricos - exemplos conhecidos, características;
    - gráficos - exemplos, não confie apenas nos seus olhos.

Testes Sistemáticos (3/4)

- Verifique as propriedades invariantes
  - alguns programas mantém propriedades da entrada
    - número de linhas
    - tamanho da entrada
    - frequência de caracteres
      - Ex: a qualquer instante o número de elementos em uma estrutura de dados deve ser igual ao número de inserções menos o número de remoções.

Testes Sistemáticos (4/4)

- Compare implementações independentes
  - os resultados devem ser os mesmos
    - se forem diferentes, pelo menos uma das implementações está incorreta
- Cobertura dos testes
  - cada comando do programa deve ser executado por algum teste
    - existem *profilers* que indicam a cobertura de testes

Automação de testes

- Testes manuais
  - tedioso, não confiável
- Testes automatizados
  - devem ser facilmente executáveis
    - junte em um *script* todos os testes

Automação de testes

- Teste de regressão automáticos
  - Comparar a nova versão com a antiga
  - verificar se os erros da versão antiga foram corrigidos
  - verificar que novos erros não foram criados
- Testes devem rodar de maneira silenciosa
  - se tudo estiver OK

Automação de testes

Exemplo de shell script:

```
for i in Ka_data.*      # laço sobre os testes
do
  old_ka $i > out1      # versao antiga
  new_ka $i > out2      # nova versao
  if !cmp -s out1 out2 # compara
  then
    echo $i: Erro       # imprime mensagem
  fi
done
```

Automação de testes

- Crie testes autocontidos
  - testes que contém suas próprias entradas e respectivas saídas esperadas

- O que fazer quando um erro é encontrado?
  - se não foi encontrado por um teste
    - faça um teste que o provoque

Ambiente de testes

- As vezes, para se testar um componente isoladamente é necessários criar um ambiente com características de onde esse componente será executado
  - ex: testar funções `mem*` do C (como `memset`)

Ambiente de testes

```
/* memset: set the first n bytes of s to the byte c */
void *memset(void *s, int c, size_t n) {
    size_t i;
    char *p;

    p = (char *) s;
    for (i=0; i<n; i++)
       p[i] = c;
    return s;
}

// memset(s0 + offset, c, n);
// memset2(s1 + offset, c, n);
// compare s0 e s1 byte a byte
```

Como testar funções do math.h ?

Testes de estresse

- Testar com grandes quantidades de dados
  - gerados automaticamente
  - erros comuns:
    - *overflow* nos buffers de entrada, vetores e contadores
  - Exemplo: ataques de segurança
    - `gets` do C não limita o tamanho da entrada
    - o `scanf(``%s'', str)` também não...
    - Erro conhecido por "buffer overflow error" NYT98

Testes de estresse

Exemplos de erros que podem ser encontrados:

```
char *p;

p = (char *) malloc (valorBemGrande);
```

Conversão entre tipos diferentes:

Foguete Ariane 5
conversão de double de 64 bits em int de 16 bits => BOOM
http://www-users.math.umn.edu/~arnold/disasters/ariane.html

Dicas para fazer testes

- Cheque os limites dos vetores
  - caso a linguagem não faça isto por você
  - faça com que o tamanho dos vetores seja pequeno; ao invés de criar testes muito grandes
- Faça funções de hashing constantes
- Crie versões de malloc que ocasionalmente falham
- Desligue todos os testes antes de lançar a versão final

Dicas para fazer testes

- Inicialize os vetores e variáveis com um valor não nulo
  - ex: 0xDEADBEEF pode ser facilmente encontrado
- Não continue a implementação de novas características se já foram encontrados erros
- Teste em várias máquinas, compiladores e SOs

Tipos de teste

- “white box”
  - testes feitos por quem conhece (escreveu) o código
- “black box”
  - testes sem conhecer o código
- “usuários”
  - encontram novos erros pois usam o programa de formas que não foram previstas

Teste de Software Orientado a Objetos

- Testes em geral (não apenas a la XP);
- Diferenças em relação a teste de software tradicional?
  - Podemos não conhecer a implementação de objetos que o nosso código usa;
  - a modularização e o encapsulamento ajudam a organização dos testes.

Tipos de testes em software OO

- testes das classes
- testes de interações
- testes de regressão
- teste do sistema e sub-sistemas
  - Está conforme aos requisitos?
- teste de aceitação
  - Posso usar a componente X?
- testes de implantação

Abordagem de McGregor/Sykes

- Lema:
  - *Teste cedo. Teste com frequência. Teste o necessário*
- Processo iterativo:
  - projete um pouco
  - escreva um pouco de código
  - teste o que puder
  - analise um pouco

Dimensões do Processo de Testes 1/2

- Quem cria os testes?
  - Os desenvolvedores? uma equipe especializada em testes? ambos?
- Quais partes são testadas?
  - Todas? Nenhuma? Ou só as de alto risco?
- Quando os testes serão realizados?
  - Sempre? Rotineiramente? No final do projeto?

Dimensões do Processo de Testes 2/2

- Como será feito?
  - Baseado no que o software faz ou em como o software faz?
  - Os testadores conhecem a implementação ou só a interface?
- Quanto de testes é o adequado?

Papéis no Processo de Testes

- Testador de unidades (classes)
- Testador da Integração
  - testa as interações entre objetos
- Testador do sistema
  - conhece o domínio e é capaz de verificar a aplicação como um todo
  - ponto de vista do usuário do sistema
- Gerente do Processo de Testes
  - coordena e escalona os testes e as pessoas

Planejamento de Testes 1/2

- Muitas vezes é esquecido ou não é considerado pelos gerentes de projeto
- Atividades de planejamento:
  - Escalonamento das Atividades de Testes
  - Estimativas de custo, tempo e pessoal necessário para realizar os testes
  - Equipamento necessário

Planejamento de Testes 2/2

- Atividades de planejamento:
  - Definição do nível de cobertura: quanto maior, mais código será exigido.
    - Beizer: 2% a 80% do tamanho da aplicação.
  - métricas para avaliar eficácia de um conjunto de testes
    - cobertura do código
    - cobertura das pós-condições
    - cobertura dos elementos do modelo

Testes das Classes (unidades)

- Uma maneira é o *peer-review*
  - Errar é humano
- Testes automatizados são melhores
  - Difíceis de construir
- Testes automatizados devem cobrir
  - alguns casos normais
  - o maior número possível de casos limítrofes

Testes das Interações

- Objetos podem interagir de 4 formas diferentes:
  - um objeto é passado como parâmetro para outro objeto numa chamada de método
  - um objeto devolve uma referência para outro objeto numa chamada de método
  - um método cria uma instância de outro objeto
  - um método usa uma instância global de outra classe (normalmente evitado)

Casos: Teste das interações 1/2

- Chamadas de métodos
- 2 abordagens:
  - Programação defensiva
    - O receptor verifica os parâmetros
  - Programação por contrato
    - A mensagem é verificada antes do envio

Casos: Teste das interações 2/2

- Subclasses/superclasses
  - Use o diagrama de classes para identificar quais testes de regressão devem ser realizados quando uma classe é alterada ou uma nova classe é criada.
  - Execute os testes escritos para a superclasse mas agora usando a nova subclasse
  - Para testar classes abstratas, somos obrigados a criar classes concretas só para testá-las

Lembre-se

- Por que não escrever testes ?
  - estou com pressa
- Quanto maior a pressão
  - menos testes
- Com menos testes
  - menos produtividade e menor estabilidade
- Logo, a pressão aumenta....

O único conceito mais importante de testes é

DO IT

Baseado em

- Baseado em:
  - The Practice of Programming: Kernighan & Pie
  - *A Practical Guide to Testing Object-Oriented Software*. John McGregor & David Sykes
  - http://www.testing.com/

www.menti.com
código 25 71 26 1

Testes em Métodos Ágeis

- Cenário:
  - Defeitos são caros
  - Quanto mais tarde são encontrados, mais caros
  - Conclusão: É melhor encontrar defeitos o mais cedo possível
- Portanto:
  - Teste cedo e frequentemente

Testes em Métodos Ágeis

- Quando testar?
  - sempre!
    - antes, durante e depois da implementação
- Como testar?
  - usando um arcabouço apropriado
    - JUnit, CPPUnit, Sunit, C#Unit, Cunit, PyTest
    - HTTPUnit, JWebUnit, Selenium, Robot
    - mais obscuros: PDFUnit, XMLUnit, SQLUnit

Introdução - Junit

- Arcabouço livre para testes automatizados escrito em Java
- Escrito originalmente por Kent Beck e Erich Gamma
- Parte de uma família de arquitetura para testes conhecida como xUnit
- Utilizado principalmente no desenvolvimento de testes de unidade
- `http://www.junit.org`

Por que usar JUnit?

- Facilita a escrita de testes automatizados
- Funcionalidades inclusas:
  - Asserções para testar resultados esperados
  - *Fixtures* para reutilização de dados para teste
  - *Test Suites* para organizar e executar conjuntos de testes
  - Interface gráfica e textual para execução de testes
- Integração com as principais IDEs
- Grande comunidade de usuários

Quando escrever um teste?

"Sempre que estiver tentado a escrever um `print()` ou uma expressão de depuração, escreva um teste"

-- Martin Fowler

Momentos em que é bom investir em testes:

- Durante o desenvolvimento
  - Crie testes para as classes que está desenvolvendo
- Durante a correção de defeitos
  - Crie um teste que reproduza o erro antes de corrigí-lo

Como escrever um teste?

- Mais simples:
  - Crie uma subclasse de TestCase

```
public class TesteSimples extends TestCase {
    (…)
}
```

  - Crie um método de teste (que comece com `test`) que verifica os resultados esperados

```
public void testColecaoVazia() {
    Collection colecao = new ArrayList();
    assertTrue(colecao.isEmpty());
}
```

Como escrever um teste?

- *Fixture*: Conjunto de dados de teste e objetos utilizados na execução de um ou mais testes
- Para reaproveitar uma *Fixture* em mais de um teste:
  - Sobrescreva o método `setUp()` (inicialização)

```
protected void setUp() {
    colecao = new ArrayList();
}
```

  - Sobrescreva o método `tearDown()` (limpeza)

```
protected void tearDown() {
    colecao.clear();
}
```

Como escrever um teste?

```
public class TesteSimples extends TestCase {
    private Collection colecao;
    protected void setUp() {
        colecao = new ArrayList();
    }
    protected void tearDown() {
        colecao.clear();
    }
    public void testColecaoVazia() {
        assertTrue(colecao.isEmpty());
    }
    public void testColecaoComUmItem() {
        colecao.add("itemA");
        assertEquals(1, colecao.size());
    }
}
```

Como escrever um teste?

- Possível ordem de execução:
  - `setUp()`
  - `testColecaoComUmItem()`
  - `tearDown()`
  - `setUp()`
  - `testColecaoVazia()`
  - `tearDown()`
- Como os testes são chamados por reflexão, a ordem de execução dos testes pode não seguir o mesmo fluxo do código
- Garantia: `setUp()` será executado antes e `tearDown()` será executado depois

Como escrever um teste?

- Testando uma exceção esperada (cenário de erro)
  - Capture a exceção num bloco try/catch e falhe o teste caso ela não seja lançada

```
public void testIndexOutOfBoundsException() {
    ArrayList listaVazia = new ArrayList();
    try {
        Object o = listaVazia.get(0);
        fail("Não lançou exceção esperada.");
    } catch (IndexOutOfBoundsException e) {
        assertTrue(true);
    }
}
```

Como escrever um teste?

- Testando uma exceção não esperada

  - Declare a exceção na assinatura do método e não capture-a no código do teste

```
public void testFalhaIndexOutOfBoundsException()
    throws IndexOutOfBoundsException {

    ArrayList listaVazia = new ArrayList();
    Object o = listaVazia.get(0);
}
```

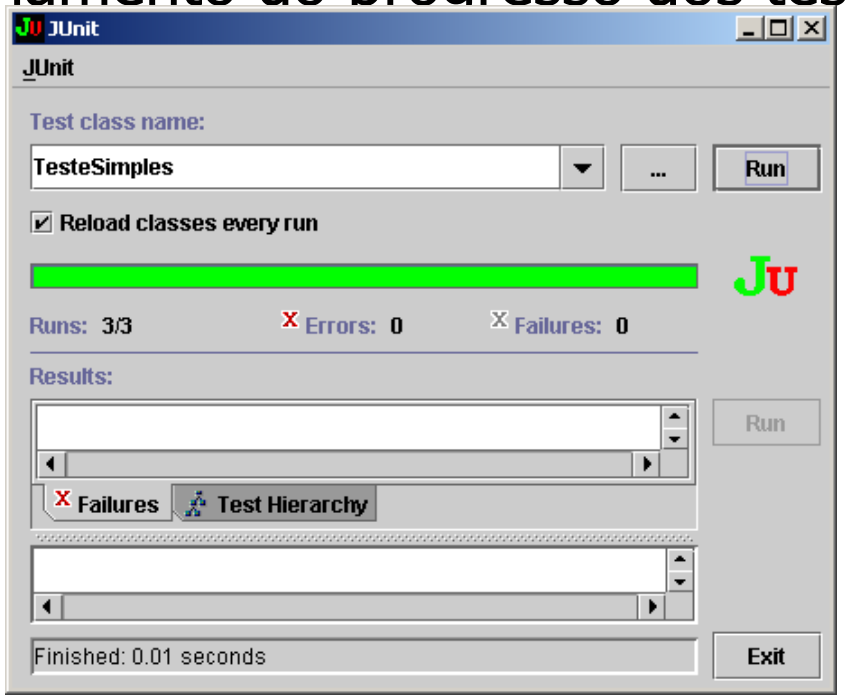
Obs: Esse teste irá falhar

Como escrever um teste?

- Algumas considerações:
  - Testes de unidade devem exercitar o comportamento isolado de uma classe
  - Geralmente, o comportamento de um objeto depende da interação com outros objetos
  - Nesse caso, é comum utilizar objetos "dublês" para isolar o comportamento
  - Alguns tipos de objetos "dublês":
    - Dummy
    - Fake
    - Stubs
    - Mocks

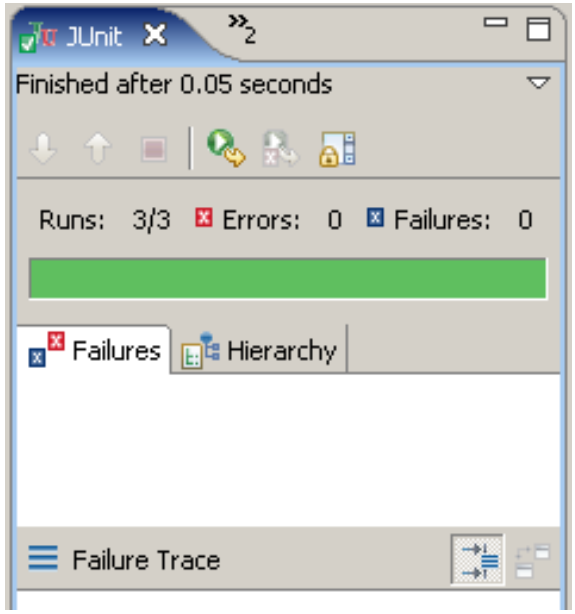
Como rodar um teste?

- JUnit vem com dois *TestRunners*:
  - Textual: utilizado na linha de comando
  - Gráfico: interface gráfica simples para execução e acompanhamento do progresso dos testes



Como rodar um teste?

- Eclipse
  - Clicar com o botão direito na classe de teste e escolher “Run As > JUnit Test”



Conclusão

- O mais importante é:
  - Testar cedo
  - Testar frequentemente
  - Testar de forma automatizada
- Arcabouços de teste ajudam com o item 3
- O resto é com você!

Desenvolvimento Dirigido por Testes

- TDD (*test-driven development*)
- Testes a Priori (*test-first programming*)

Ciclo em **Passos Pequenos**:

1. Vermelho: Escreva um teste que falha
2. Verde: Faça o teste passar rapidamente
3. Refatore

Referências

- http://www.junit.org
- K. Beck and E. Gamma. *Test Infected: Programmers Love Writing Tests*. Java Report, July 1998, Volume 3, Number 7