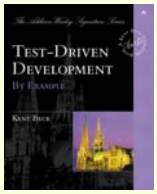
Modulo II
Técnicas para desenvolvimento de Software Ágil

Prof. Ismael H F Santos

Bibliografia

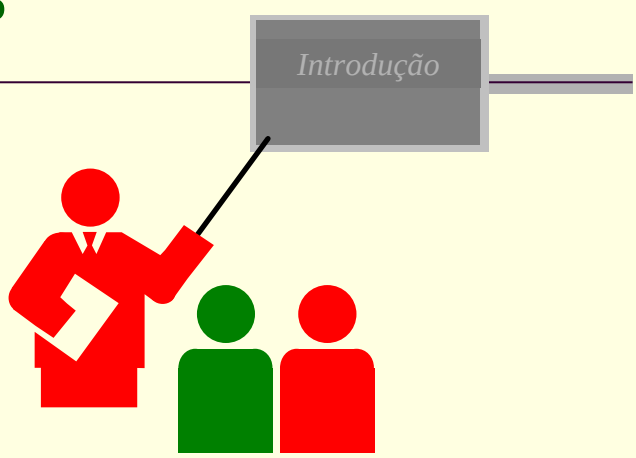
- *Software Estimation: Demystifying the Black Art (Best Practices (Microsoft))*
  by Steve McConnell
- **Test-Driven Development: By Example,** Kent Beck - Addison-Wesley, 2003, ISBN 0-321-14653-0
- **Refactoring: Improving the Design of Existing Code,** Martin Fowler - Addison-Wesley, 1999 SBN 0-201-48567-2



Agenda

- Introdução
- TDD
- Testes em XP
- Boas Práticas de Teste

MA-XP



O que é "Testar código"?

- *É a coisa mais importante do desenvolvimento*
  - *Se seu código não funciona, ele não presta!*
- *Todos testam*
  - *Você testa um objeto quando escreve uma classe e cria algumas instâncias no método main()*
  - *Seu cliente testa seu software quando ele o utiliza (ele espera que você o tenha testado antes)*
- *O que são testes automáticos?*
  - *São programas que avaliam se outro programa funciona como esperado e retornam resposta tipo "sim" ou "não"*
  - *Ex: um main() que cria um objeto de uma classe testada, chama seus métodos e avalia os resultados*
  - *Validam os requisitos de um sistema*

Por que testar?

- *Por que não?*
  - *Como saber se o recurso funciona sem testar?*
  - *Como saber se ainda funciona após refatoramento?*
- *Testes dão maior segurança: coragem para mudar*
  - *Que adianta a OO isolar a interface da implementação se programador tem medo de mudar a implementação?*
  - *Código testado é mais confiável*
  - *Código testado pode ser alterado sem medo*
- *Como saber quando o projeto está pronto*
  - *Testes == requisitos 'executáveis'*
  - *Testes de unidade devem ser executados o tempo todo*
  - *Escreva os testes antes. Quando todos rodarem 100%, o projeto está concluído!*

Testes: para quê ?

- Para verificar a validade de uma implementação com respeito aos requisitos.
- Para especificar formalmente o critério de aceitação de uma implementação pelo cliente.
- Para detalhar o design e a forma de uso da funcionalidade.
- Para garantir que não quebramos nada quando fazemos uma alteração.

Teste X Revisão X Prova X Métrica

- Teste:
  - Execução de um programa através de procedimentos manuais ou automáticos.
- Revisão/Inspeção:
  - Revisão de produtos de trabalho por especialistas: documentos, código-fonte de programas.
- Prova/Verificação:
  - Procedimento sistemático e formal para garantir que uma implementação está correta.
- Métrica:
  - Aferição de características mensuráveis dos produtos de trabalho ou da execução de programas.

Desenvolvimento em cascata

→ Levantamento de requisitos
→ Análise
→ Design de alto nível
→ Design detalhado
→ Codificação
+ teste unitário
→ Integração
+ teste de integração
→ Implantação
+ teste de sistema
→ Manutenção
+ teste de regressão

Cascata

s. f. (do italiano *cascata*)

- 1. Queda de água em cachoeira, natural ou artificial;
- 2. Brasil, gíria: Conversa fiada; mentira; bazófia

Fonte:
Dicionário Universal da Língua Portuguesa
http://www.priberam.pt/DLPO/

Algumas cascatas ...

- Inspeções são sempre mais eficientes que testes na contenção de defeitos.
- Usar uma matriz para mapear casos de teste X requisitos.
- Só se pode rodar os testes depois de implementar.
- Fazer certo da primeira vez, para evitar retrabalho.
- Procedimentos formais garantem uma implementação correta.

..., mais cascatas ...

- Definir todos os casos de teste antes de começar a programar.
- Se funciona mais ou menos, é melhor não mexer.
- Torcer para o teste dar certo na primeira rodada.
- Automatizar os testes é muito caro: não compensa.
- É impossível que quem vai rodar os testes não entenda este roteiro: está super claro.
- Vai pegar mal documentar que esse teste falhou: vai ferrar as nossas métricas.

... e mais essas.

- O teste passou: está 90% certo.
- Testes (ou inspeções, ou provas, ou métricas) *garantem* qualidade.
- Quem testa não pode ser quem programou.
- Revisar o código muito cuidadosamente antes de compilar: não depender do compilador para detectar erros triviais.
- Rodar exaustivamente testes de mesa antes de digitar o código.

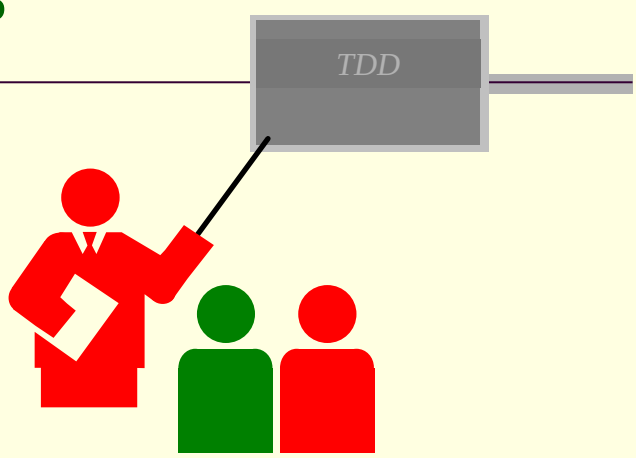
Como escrever bons testes

- *JUnit facilita bastante a criação e execução de testes, mas elaborar bons testes exige mais*
  - *O que testar? Como saber se testes estão completos?*
- *"Teste tudo o que pode falhar" [2]*
  - *Métodos triviais (get/set) não precisam ser testados.*
  - *E se houver uma rotina de validação no método set?*
- *É melhor ter testes a mais que testes a menos*
  - *Escreva testes curtos (quebre testes maiores)*
  - *Use assertNotNull() (reduz drasticamente erros de NullPointerException difíceis de encontrar)*
  - *Reescreva seu código para que fique mais fácil de testar*

Como descobrir testes?

- *Escreva listas de tarefas (to-do list)*
  - *Comece pelas mais simples e deixe os testes "realistas" para o final*
  - *Requerimentos, use-cases, diagramas UML: rescreva os requerimentos em termos de testes*
- *Dados*
  - *Use apenas dados suficientes (não teste 10 condições se três forem suficientes)*
- *Bugs revelam testes*
  - *Achou um bug? Não conserte sem antes escrever um teste que o pegue (se você não o fizer, ele volta)!*
- *Teste sempre! Não escreva uma linha de código sem antes escrever um teste!*

MA-XP



Motivation

- TDD sits nicely in the XP “way of doing things”
  - TDD can be used without practicing XP
- Writing articles and giving presentations is one such way of achieving that goal
- To reduce the amount of re-testing that is required
  - Especially with legacy applications
  - To avoid introducing new bugs after refactoring existing code

TDD

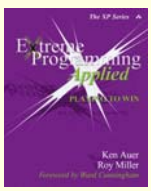
- *Desenvolvimento guiado pelos testes*
  - *Só escreva código novo se um teste falhar*
  - *Refatore até que o teste funcione*
  - *Alternância: "red/green/refactor" - nunca passe mais de 10 minutos sem que a barra do JUnit fique verde.*
- *Técnicas*
  - *"Fake It Til You Make It": faça um teste rodar simplesmente fazendo método retornar constante*
  - *Triangulação: abstraia o código apenas quando houver dois ou mais testes que esperam respostas diferentes*
  - *Implementação óbvia: se operações são simples, implemente-as e faça que os testes rodem*

What is TDD?

- TDD is a technique whereby you write your test cases before you write any implementation code
- Tests drive or dictate the code that is developed
- An indication of “intent”
  - Tests provide a specification of “what” a piece of code actually does
  - Some might argue that “tests are part of the documentation”

What is TDD?

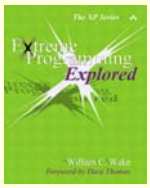
- “Before you write code, think about what it will do.
  Write a test that will use the methods you haven’t even written yet.”
  - Extreme Programming Applied: Playing To Win
  - Ken Auer, Roy Miller
  - “The Purple Book”
- A test is not something you “do”, it is something you “write” and run once, twice, three times, etc.
  - It is a piece of code
  - Testing is therefore “automated”
  - Repeatedly executed, even after small changes



TDD Stages

- In Extreme Programming Explored (The Green Book), Bill Wake describes the test / code cycle:

1. Write a single test
2. Compile it. It shouldn't compile because you've not written the implementation code
3. Implement just enough code to get the test to compile
4. Run the test and see it **fail**
5. Implement just enough code to get the test to pass
6. Run the test and see it **pass**
7. **Refactor** for clarity and "once and only once"
8. Repeat



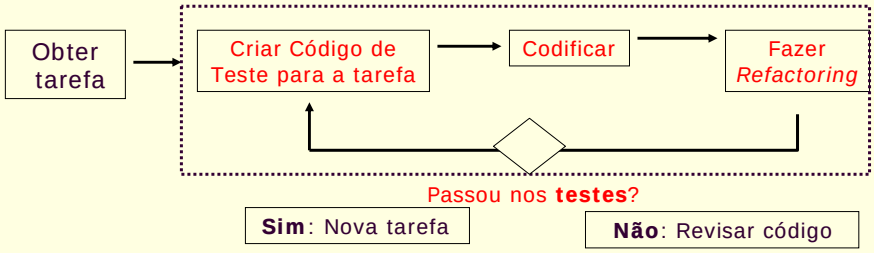
JUnit para guiar o desenvolvimento

Cenário de Test-Driven Development (TDD)

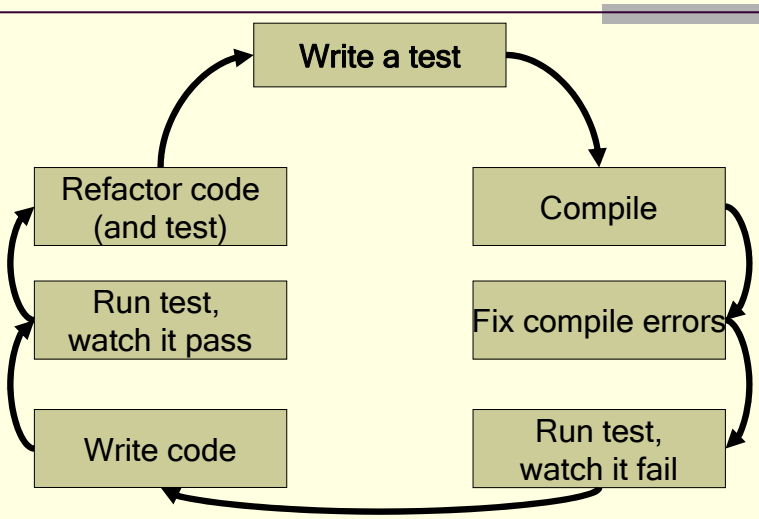
1. Defina uma lista de tarefas a implementar
2. Escreva uma classe (test case) e implemente um método de teste para uma tarefa da lista.
3. Rode o JUnit e certifique-se que o teste falha
4. Implemente o código mais simples que rode o teste
   - *Crie classes, métodos, etc. para que código compile*
   - *Código pode ser código feio, óbvio, mas deve rodar!*
5. Refatore o código para remover a duplicação de dados
6. Escreva mais um teste ou refine o teste existente
7. Repita os passos 2 a 6 até implementar toda a lista

TDD
Test Driven Development

- **Desenvolvimento Guiado por Testes**, define que antes de criarmos um código novo, devemos escrever um teste para ele.
- E testes serão usados como métrica em todo o tempo de vida do projeto.



TDD Stages

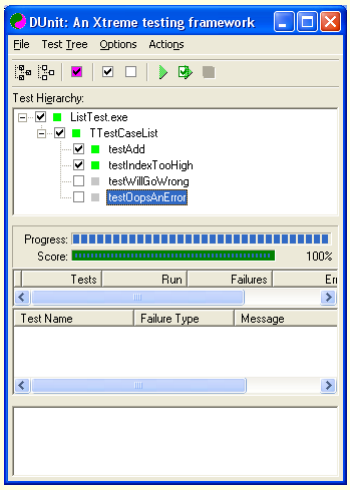


Why TDD?

- Programmers dislike testing
  - They will test reasonably thoroughly the first time
  - The second time however, testing is usually less thorough
  - The third time, well..
- Testing is considered a “boring” task
- Testing might be the job of another department / person
- TDD encourages programmers to maintain an exhaustive set of repeatable tests
  - Tests live alongside the Class/Code Under Test (CUT)
  - With tool support, tests can be run selectively
  - The tests can be run after every single change

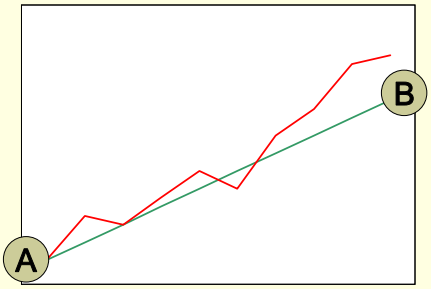
Why TDD?

- Bob Martin:
  - “The act of writing a unit test is more an act of design than of verification”
- Confidence boost
  - By practicing TDD, developers will strive to improve their code – without the fear that is normally associated with code changes
  - Isn’t the green bar a feel good factor?
- Remove / Reduce reliance on the debugger
  - No more “debug-later” attitudes

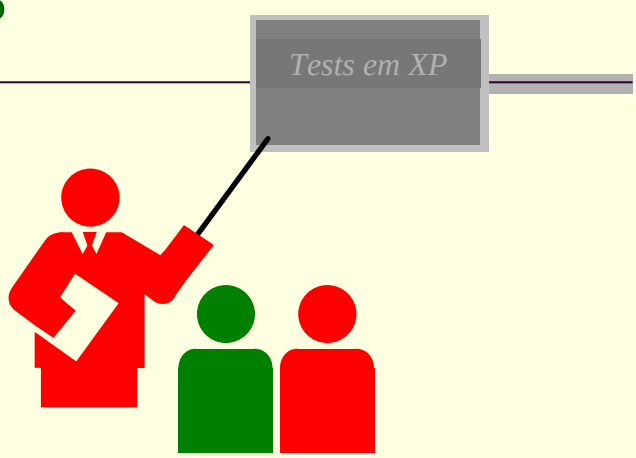


Who should write the tests?

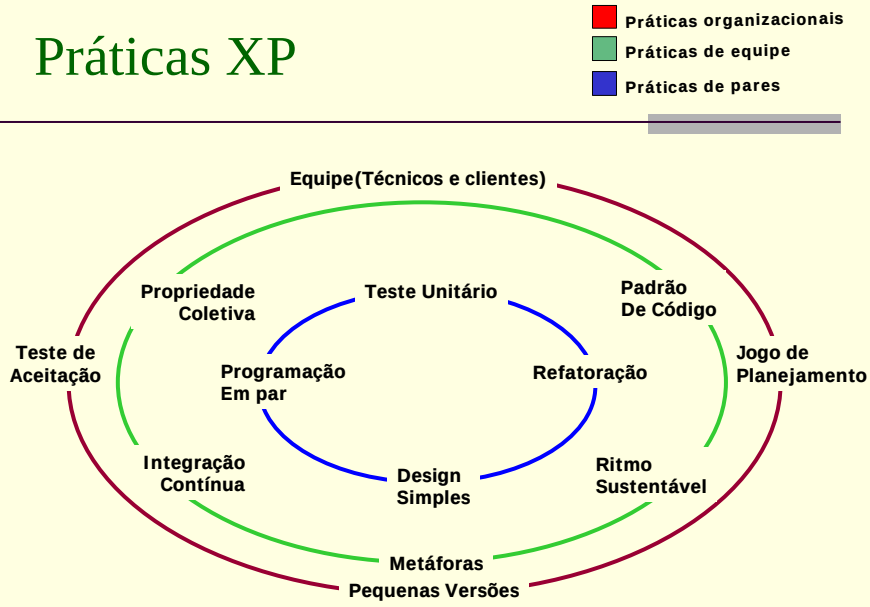
- The programmers should write the tests
  - The programmers can't wait for somebody else to write tests
- TDD promotes "small steps", and lots of them
  - Small steps: the shortest distance between two points
  - Your destination is closer…



MA-XP



Práticas XP



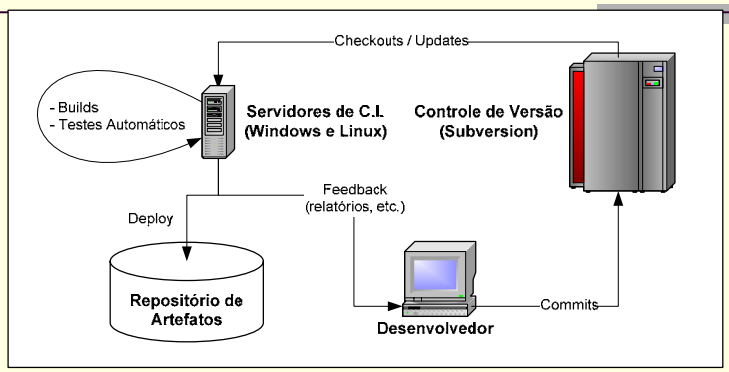
Testes em XP

- Testes do programador
  - Testes unitários (caixa branca)
  - Codificados pelo desenvolvedor.
  - Detalham o design da implementação.
  - Rodam muito rápido.
  - Todo código integrado roda 100% desses testes.

Testes em XP

- Testes do cliente
  - Testes de sistema (caixa preta)
  - Escritos pelo cliente.
  - Detalham os critérios de aceitação de uma implementação.
  - Podem demorar mais.
  - Mais difíceis
  - Quando estão rodando 100%, o produto pode ser entregue.

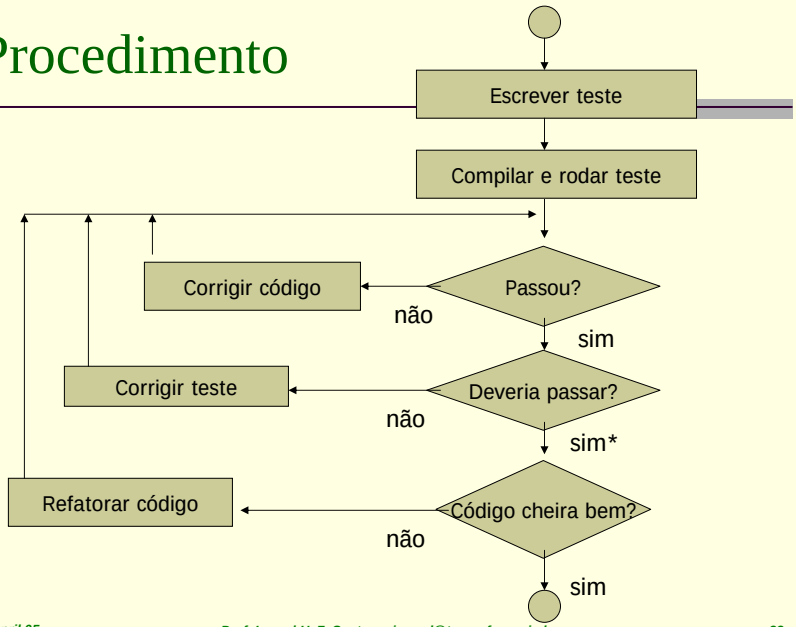
Integração Contínua



- Toda vez que é integrado código, rodam-se os testes. Todo teste é teste de regressão.
- Versionamento de Código, Builds e Testes Automáticos

  => Controle de Qualidade Superior + ,Histórico de Artefatos

Procedimento



Procedimento

- Entrada: estórias de usuário (requisitos), arquitetura

1. Escreva um teste para uma porção ridiculamente pequena da funcionalidade.
2. Compile e rode o teste.
3. Escreva o mínimo código funcional para passar o teste (com possível enganação). Compile.
4. Compile e rode o teste
5. Melhore o teste para desvendar a enganação, se houver Vá para (2)
6. Melhore (refatore) o código funcional. Vá para (2)

Testas ≠ 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 uma entrada vazia, um único item, um vetor cheio, etc.

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.

Uso de premissas

- Em C e C++ use <assert.h>
  - 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 ...
```


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

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
    - Ex: teste de função que faz a busca binária em inteiros

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 linha
    - tamanho da entrada
    - freqüência 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 (3/4)

```
#include <stdio.h>
#include <ctype.h>
#include <limits.h>
unsigned long count[UCHAR_MAX+1];

int main(void) {
    int c;
    while ((c = getchar()) != EOF){
        count[c]++;
    }
    for(c=0; c <= UCHAR_MAX; c++){
        printf(``%.2x %c %lu\n'', c, isprint(c) ? c: `-',
                count[c]);
    return 0;
}
```

1) Como melhorar e testar o programa acima ?
2) Como proceder no caso de outros tipos de dados de 32bits. Faça uma versão do programa que trate estes dados de maneira elegante.

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 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
  - programas tipo awk podem ajudar
- O quê fazer quando um erro é encontrado
  - se não foi encontrado por um teste
    - faça um teste que o provoque
- Como fazer um testador automático para o programa de freqüência ?

Framework de testes

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

```
/* 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 stress

- 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 stress

Exemplos de erros que podem ser encontrados:

```
char *p;

p = (char *) malloc (x * y * z);
```

Conversão entre tipos diferentes:

Ariane 5

conversão de double de 64 bits em int de 16 bits => BOOM

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
- 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

Testes difíceis

- Testes unitários não podem depender de estado
- Então, como testar:
  - Bancos de Dados
  - Servidor Web
  - Interface Usuário
  - ..., etc.

Teste em camadas

- Coloca-se uma camada que funciona como se o serviço sempre fizesse a coisa certa, para testar o cliente (*mock objects* = objetos fajutos)
- O serviço tem testes específicos que verificam seu funcionamento interno.

Extensões e Vantagens

- Desempenho (JUnitPerf)
- Métricas de qualidade
- Padrões de projeto (PatternTest)
- Cobertura de testes (Jester)
- ... Todas escritas seguindo o padrão Xunit.
- Vantagens
  - Integração com ambientes de desenvolvimento (IDEs)
  - Feedback rápido
  - Robustez

Referências

- Grupo testdrivendevelopment do Yahoo
- http://junit.org
- Test-driven development (Kent Beck, 2003)
- http://www.xispe.com.br