Modulo II
Técnicas para desenvolvimento de Software Ágil

Prof. Ismael H F Santos

Bibliografia

- *Vinicius Manhaes Teles, Extreme Programming, Novatec Editora*
- *Agile Software Development*
- *Scrum and XP from the Trenches*
- *Martin Fowler, Analysis Patterns - Reusable Object Models, Addison-Wesley,1997*
- *Martin Fowler, Refatoração - Aperfeiçoando o projeto de código existente, Ed Bookman*

Ementa

- Modulo I – Xtreme Programming
  - Valores e Princípios do XP
  - Desenvolvimento centrado em Testes
    - Continuous Integration
      - JUnit, Maven, Code-cruiser

MA-XP

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.

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.

Desenvolvimento conduzido pelos testes: O quê é ? (1)

- Não é técnica de teste: é disciplina de desenvolvimento.
- Todo teste é automatizado.
- Código do teste é escrito antes do código que deve ser testado.
- É bom que o teste falhe na primeira vez.
- Nenhum código funcional é escrito sem testes pré-existentes.

Desenvolvimento conduzido pelos testes: O quê é ? (2)

- A maioria dos testes roda muito rápido (em segundos).
- Toda implementação tem um jeito razoável de ser testada.
- Ciclo de desenvolvimento na ordem de *minutos.*
- Toda correção começa com um teste que falha para o defeito a ser corrigido.
- Escreve-se mais código de teste que código funcional.

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.

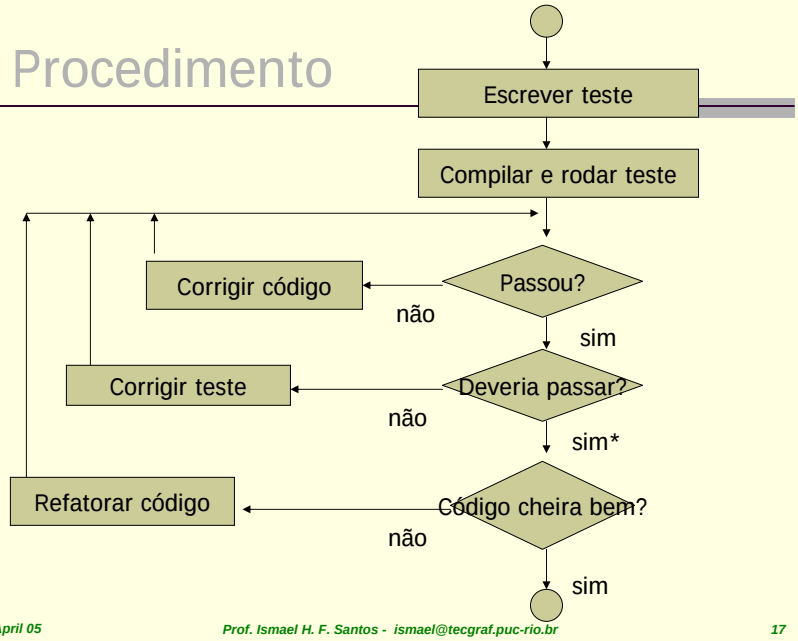
Testes em XP

- 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.

Testes em XP

- Integração Contínua
  - Toda vez que é integrado código, rodam-se os testes
  - Todo teste é teste de regressão

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 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 caracter ok; 2 caracteres ok;
MAX caracteres ok

e se o primeiro caracter 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 quê 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;
}
```

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

Exemplos / exercícios:

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

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


Exemplos / exercícios:

Imprimir caracteres, um por linha

```
i = 0;
do {
    putchar(s[i++]);
    putchar('\n');
} while (s[i] != 0);
```

Exemplos / exercícios:

Copia uma string de src a dest

```
void strcpy( char *dest, char *src) {
    int i;

    for( i = 0; src[i] != '\0'; i++) {
        dest[i] = src[i];
    }
}
```


Exemplos / exercícios:

Copia até n caracteres de uma string de s a d

```
void strcpy( char *d, char *s, int n) {
    while (n > 0 && *s != '\0'){
        *t = *s;
        t++;
        s++;
        n--;
    }
}
```

Exemplos / exercícios:

Uma comparação

```
if (i > j)
     printf(``%d e maior que %d.\n'', i, j);
else
     printf(``%d e menor que %d.\n'', i, j);
```


Exemplos / exercícios:

Um programa simples com caracteres

```
if (c >= `A' && c <= `Z') {
    if (c <= `L')
        cout << ``first half of alphabet'';
    else
        cout << ``second half of alphabet'';
```

Exemplos / exercícios:

1) com que datas você checaria se um programa passaria pelo bug do milênio ?
2) se os testes fossem caros o que você testaria após 1/1/2000 ?
3) que outros erros relacionados você consegue imaginar ?

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`)

Framework 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 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

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

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

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