Aula 24
Testes I

Eiji Adachi

LES/DI/PUC-Rio

Novembro 2011

Aulas passadas

- Aula 15: Tratamento de Exceções
  - O que são exceções
  - Por que tratar exceções
  - Dialeto em C
  - Dificuldade em testar código de tratamento de exceções

Aulas passadas

- Aula 22 e 23: Instrumentação
  - Objetivo: controlar a corretude durante a execução
  - Tipos de instrumentos:
    - Externos
      - Armaduras de testes
      - Depuradores
    - Internos
      - Assertivas executáveis
      - Controladores de espaços de dados (CESPDIN.h)
      - Contadores de passagens (CONTA.h)
      - Verificadores de estruturas de dados
      - Deturpadores de estruturas de dados

Aula de Hoje – Sumário

- Testes de Exceções
  - Falta de Memória
  - Falhas em operações de Entrada / Saída
  - Parâmetros errados / inválidos
    - Deturpadores
  - Cenários compostos
  - Detecção automática de vazamento de recursos
- Testes de Regressão
- Testes Caixa Preta
  - Aleatórios
  - Classes de Equivalência
  - Análise de Limites

Testes de Exceções

- É (relativamente) fácil construir um sistema que se comporta corretamente quando este recebe apenas entradas corretas, recursos estão sempre disponíveis, o hardware não apresenta falhas, etc...
- É (bem mais) difícil construir um sistema que se comporte adequadamente à entradas incorretas, à falta de recursos, à falhas de hardware, etc...
  - Código de tratamento de exceção visa detectar e lidar com estas situações

Testes de Exceções

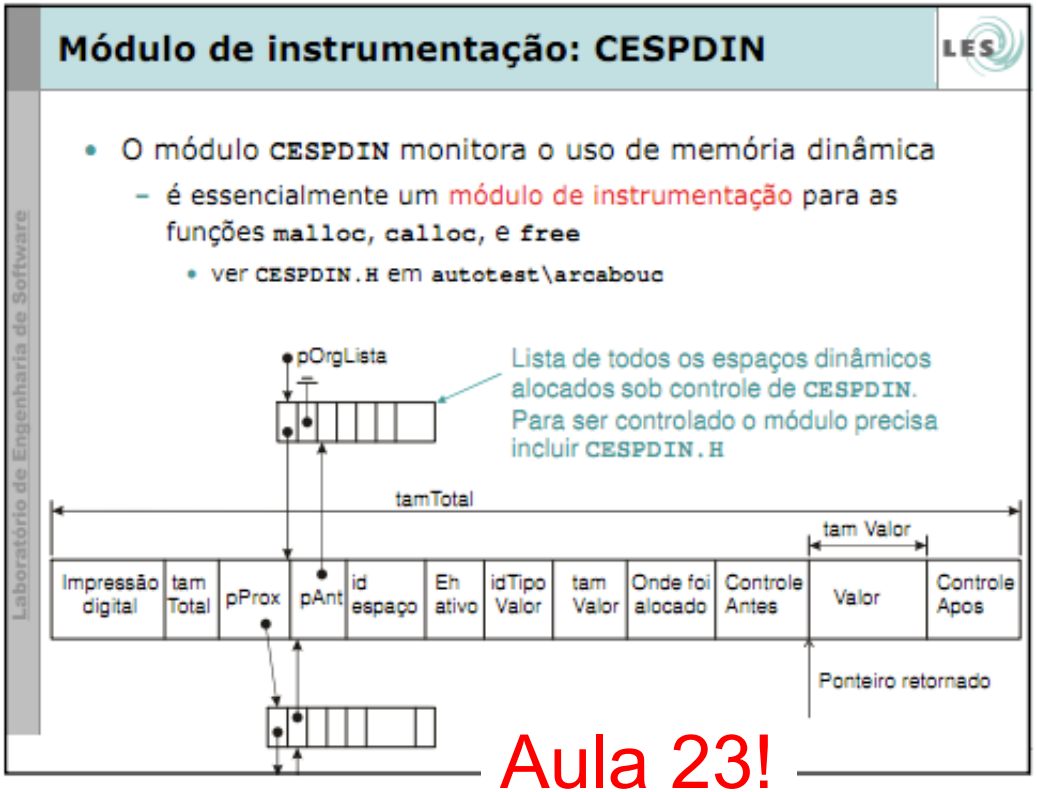
- Testes de exceções visam verificar o comportamento correto de um sistema quando algo "inesperado" / "excepcional" ocorre
  - Falta de Memória
  - Faltas em operações de Entrada / Saída
  - Acidentes ("*crashes*")
  - Parâmetros errados / inválidos
- Nem sempre é fácil testar o código do tratamento de exceções
  - Descobrir quais assertivas devem ser quebradas
  - Qual o comportamento esperado em situações excepcionais?
  - Geralmente é necessário um mecanismo de injeção de erros

Testes de Exceções – Falta de Memória

- Falta de Memória
  - Mais comum em sistemas que fazem uso extensivo de alocação dinâmica de memória (ex. Bancos de Dados)
    - Em servidores / estações de trabalho é incomum a *Falta de Memória*
    - Mas em sistemas embarcados faltas por *Falta de Memória* são bem mais comuns
      - Nestes casos, é importante que o sistema lide "elegantemente" com estas faltas

Testes de Exceções – Falta de Memória

- Como testar situações em que ocorre Falta de Memória?
  - R: Simulando a falta de memória!
- Como simular Falta de Memória?
  - R: Instrumentação!



Testes de Exceções – Entrada / Saída

- Faltas em operações de Entrada / Saída
  - Podem ser decorrentes de:
    - Disco cheio
    - Hardware defeituoso
    - Interrupções em transmissões via rede
    - Problemas relacionados a permissões
    - Problemas relacionados ao sistema operacional
    - ...
  - Novamente: é importante que o sistema lide "elegantemente" com estas faltas

Testes de Exceções – Entrada / Saída

- Faltas em operações de Entrada / Saída
  - Como testar situações em que ocorrem faltas em Operações de Entrada / Saída?
    - R: Simulando a faltas em Operações de Entrada / Saída
  - Como simular faltas em Operações de Entrada / Saída?
    - R: Instrumentação!

```
#ifdef _TESTE
        #include "stdio_instr.h"
#else
        #include <stdio.h>
#endif
```

```
int scanf( const char * format, ...)
{
    if( count > MAX_THRESHOLD )
    {
        return EOF;
    }
    else
    {
        //chama normalmente scanf
    }
}
```

Testes de Exceções – Parâmetros Inválidos

- Parâmetros errados / inválidos
  - Em muitos casos, os parâmetros de entrada são tipos primitivos ou estruturas de dados simples
    - Nestes casos, é fácil testar uma função passando parâmetros inválidos
      - Ex.: Ponteiro nulo, valor inteiro negativo, valor maior que um determinado limite, string vazia, etc...
  - Em outros casos, os parâmetros de entrada podem ser
    - Estruturas de dados complexas, ou
    - Arquivos
      - Nestes casos, é difícil e custoso criar manualmente estes parâmetros com as condições desejadas
        - » Ex.: uma árvore cujo nó mais a direita possui o ponteiro esquerdo apontando para "lixo"
        - » Ex.: arquivo .doc corrompido

Testes de Exceções – Parâmetros Inválidos

- Parâmetros errados / inválidos
  - Como testar casos em que dados errados / inválidos são passados como parâmetros?
    - R: Passando dados errados / inválidos
  - Mas e nos casos complexos?
    - Usando deturpadores
      - Deturpadores de estrutura de dados
      - Deturpadores de arquivos

Deturpador de estruturas de dados

- Deturpadores são funções especificamente projetadas para danificar estruturas de dados
  - podem ser funções que alteram um sub-espaço do espaço dado, baseando-se no deslocamento relativo à origem desse espaço, na dimensão do espaço a deturpar e no valor a inserir
    - neste caso o deturpador pode ser externo ao módulo em teste
    - porém torna o deturpador sensível à plataforma e a parâmetros de compilação
  - podem requerer conhecimento da organização dos dados
    - neste caso as funções serão internas ao módulo que contém os dados a serem deturpados
      - evitar perda de encapsulamento
    - facilita a escolha do que deve ser danificado bastando fornecer o nome, ou id, do atributo e o seu novo valor

Deturpador de estruturas de dados

- Deturpadores devem sempre ser desativados quando o módulo for compilado para produção
  - deturpadores intrinsecamente abrem uma brecha de segurança
  - deturpadores podem fazer com que o construto em teste "voe". Por exemplo ao substituir um ponteiro por algum outro ilegal.
    - sugestão: criar espaços de dados lixo (caixas de areia) e somente deturpar com ponteiros ilegais que estejam apontando para estes espaços
- o módulo **`ARVORE`** em **`autotest\instrum\fontes`** do arcabouço de apoio aos testes ilustra verificadores e deturpadores

Deturpador de estruturas de dados


```
void ARV_Deturpar( void * pArvoreParm , int ModoDeturpar )
{
...
   switch ( ModoDeturpar ) {
      /* Modifica o tipo da cabeça */
         case DeturpaTipoCabeca :
            CED_DefinirTipoEspaco( pArvore , CED_ID_TIPO_VALOR_NULO ) ;
            break ;
      /* Anula ponteiro raiz */
         case DeturpaRaizNula :
            pArvore->pNoRaiz = NULL ;
            break ;
      /* Faz raiz apontar para lixo */
         case DeturpaRaizLixo :
            pArvore->pNoRaiz = ( tpNoArvore * )( EspacoLixo ) ;
            break ;
...
```

Deturpador de arquivos

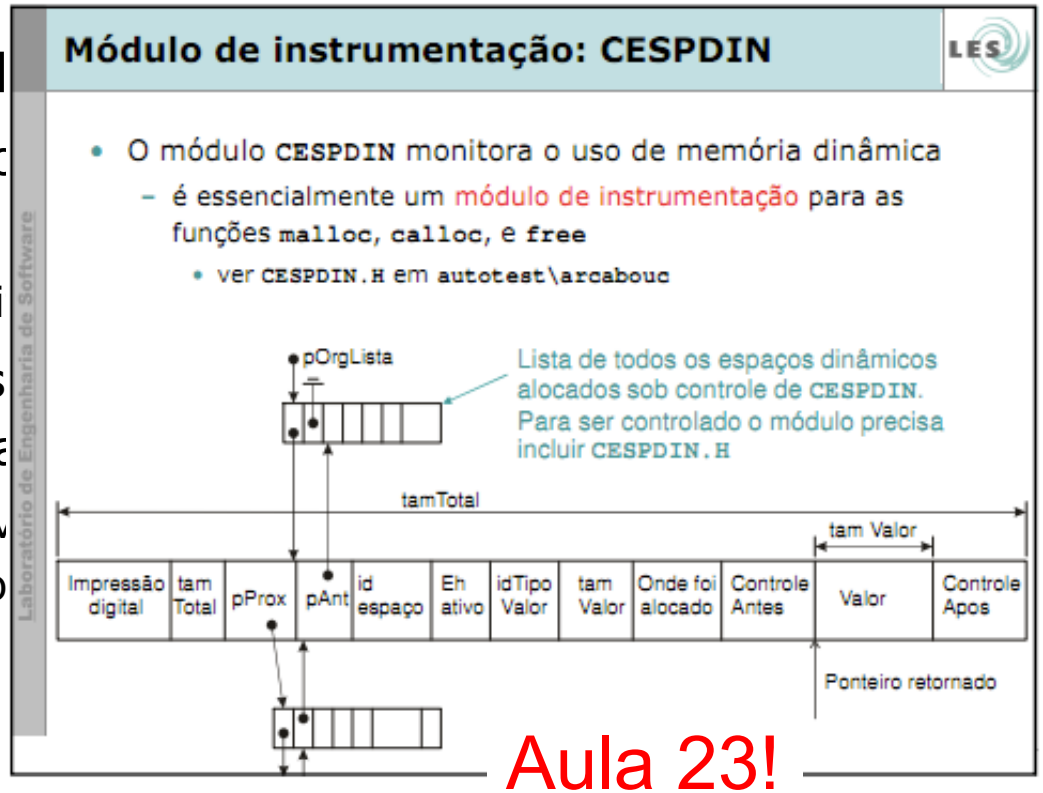
- São funções ou módulos externos a aplicação que criam arquivos inválidos / corrompidos
  - Recebe como parâmetro de entrada um arquivo válido
  - Corrompe o arquivo alterando um ou mais bytes
    - Deve-se alterar os bytes que correspondem a estrutura do arquivo
- Como nem sempre sabe-se quais bytes correspondem a estrutura do arquivo, pode-se alterar bytes aleatoriamente
  - Alterando aleatoriamente, podem ocorrer casos que não são interessantes aos testes de exceções
    - Os bytes alterados fazem parte dos dados do arquivo. Nestes casos, o arquivo permanece válido.
    - Os bytes alterados fazem parte de bytes não utilizados pelo arquivo. Nestes casos, o arquivo também permanece válido.

Testes de Exceções

- Cenários Compostos
  - O testador deve também criar casos de testes que simulem situações em que múltiplas falhas ocorrem seguidamente
    - Ex.: Falta de Memória após uma falha em Operação de Entrada / Saída
- Detecção automática de vazamento de recursos:
  - Vazamento de recursos ocorrem quando recursos são alocados e nunca são liberados
    - Ex.: Memória, descritores de arquivos, threads, mutexes, etc.
  - Todos os casos de teste devem monitorar alocação de recursos e reportar qualquer vazamento
    - Não podem ocorrer vazamentos em nenhuma circunstância: terminação normal ou terminação excepcional

Testes de Exceções

- Cenários Compostos
  - O testador deve também criar casos de testes que simulem situações em que múltiplas falhas ocorrem seguidamente
    - Ex.: Falta de Memória após uma falha em Operação de Entrada / Saída
- Detecção automática d
  - Vazamento de recurso
    e nunca são liberados
    - Ex.: Memória, descri
  - Todos os casos de tes
    e reportar qualquer va
    - Não podem ocorrer v
      terminação normal o



Testes de Regressão

- Testes de Regressão é o processo de re-executar casos de teste para garantir que mudanças no software não causaram "efeitos colaterais" indesejados (defeitos)
  - Se defeitos aparecerem em componentes testados anteriormente, é dito que o sistema regrediu
- Mudanças no software podem ser
  - como adição de novas funcionalidades
  - integração de novos módulos
  - a correção de um defeito, etc
- Modificações em código de Tratamento de Exceções costumam afetar partes de um sistema que não são obviamente detectáveis

Testes de Regressão

- Em um cenário ideal:
  - Os testes são automatizados
  - Existe um conjunto de testes que cobre 100% do código
  - O conjunto de testes é executada após toda modificação no código
- Mas para muitos projetos esse cenário é impraticável
  - Executar os testes demanda muito tempo
  - Alterações são muito freqüentes
  - Recursos escassos para realizar os testes

Testes de Regressão

- Sistemática durante o desenvolvimento:
  - Para cada novo defeito encontrado:
    - Crie novos casos de teste que detectem esse defeito
    - Corrija o defeito
    - Certifique-se que os testes detectam esse defeito não falham mais
- Sistemática durante a execução dos testes de regressão
  - Para cada teste que falhar (esperado diferente do obtido):
    - Examine o código executado pelo teste (e o seu próprio código) e verifique se realmente foi detectado um defeito
    - Se o teste detectou um defeito, siga os passos acima
    - Se o teste falhou e não era um defeito, então houve uma mudança no comportamento do sistema. Neste caso, atualize o caso de teste

Testes Caixa Preta

- Testes Caixa Preta (ou Caixa Fechada) consideram um módulo sob testes uma "caixa opaca", i.e., não há conhecimento a respeito da estrutura interna, apenas do que o módulo faz
- Testes Caixa Preta utilizam, exclusivamente, as especificações do artefato para formular os casos de teste
  - Consideram-se
    - Relações entre entradas e saídas
    - Assertivas
    - Requisitos
    - Conhecimento do domínio da aplicação

Testes Caixa Preta

- Mas como escolher um conjunto de casos de testes adequados dentre todo o conjunto de entradas válidas e inválidas?
  - É impraticável / impossível testar todas as possíveis entradas, mesmo para funções muito simples
    - Ex.: int soma(int x, int y)
- Critérios para seleção de entradas:
  - Aleatório
  - Classes de Equivalência
  - Análise de Limite

Testes Caixa Preta – Aleatório

- Cada módulo / sistema possui um conjunto domínio de entradas do qual as entradas de teste são selecionadas
- Se um testador escolhe aleatoriamente (ou ao acaso) entradas de teste deste domínio, isto é chamado de teste aleatório
  - Ex.: soma(2, 3), soma(0, 1)...
- Vantagens:
  - Fácil e de baixo custo
- Desvantagens:
  - Não há garantia da qualidade dos casos de teste

Testes Caixa Preta – Classes de Equivalência

- O critério de Classes de Equivalência divide o domínio das entradas em finitos conjuntos (cada um destes conjuntos é chamado de classe de equivalência)
- O objetivo deste critério é definir um conjunto minimal de casos de teste
  - Para cada classe de equivalência, um caso de teste
    - Se determinado caso de teste não conseguir encontrar uma falha, outros casos de testes equivalentes também não o conseguirão
- O testador deve considerar tanto classes de equivalência válidas como inválidas
- Também é possível definir classes de equivalência para o conjunto de saídas

Testes Caixa Preta – Classes de Equivalência

- Vantagens:
  - Reduz o conjunto de casos de testes
  - Diversifica o conjunto de casos de testes, aumentando a probabilidade de detectar defeitos
- Desvantagens:
  - Não é fácil identificar todas as classes de equivalência

Testes Caixa Preta – Classes de Equivalência

- Exemplo: Uma função que recebe uma string e verifica se esta corresponde a um identificador válido ou não. Um identificador é considerado válido se, e somente se, ele possui no mínimo 3 e no máximo 15 caracteres alfanuméricos, sendo que os dois primeiros caracteres são letras

- Do enunciado podemos derivar três condições básicas (ou assertivas de entrada):
  1. A string só possui caracteres alfanuméricos
  2. O número total de caracteres está entre [3, 15]
  3. Os dois primeiros caracteres são letras

Testes Caixa Preta – Classes de Equivalência

- Da primeira condição derivamos as seguintes classes de equivalência:

  "A string só possui caracteres alfanuméricos

  - Classe 1: A string só possui caracteres alfanuméricos
  - Classe 2: A string possui um ou mais caracteres não alfanuméricos

Testes Caixa Preta – Classes de Equivalência

- Da segunda condição derivamos as seguintes classes de equivalência:

  "O número total de caracteres está entre [3, 15]"

  - Classe 3: A string possui entre [3, 15] caracteres
  - Classe 4: A string possui menos do que 3 caracteres
  - Classe 5: A string possui mais do que 15 caracteres

Testes Caixa Preta – Classes de Equivalência

- Da terceira condição derivamos as seguintes classes de equivalência:

  "Os dois primeiros caracteres são letras"

  - Classe 6: Os dois primeiros caracteres são letras
  - Classe 7: Um dos dois primeiros caracteres não é letra

Testes Caixa Preta – Classes de Equivalência

- Ao fim, temos as seguintes classes de equivalência:

Condição	Classes Válidas	Classes Inválidas
1	Classe 1	Classe 2
2	Classe 3	Classes 4, 5
3	Classe 6	Classe 7

- E o seguinte conjunto de casos de testes:
  - Classe 1: {"ab12345"}
  - Classe 2: {"sah12^*&^"}
  - Classe 3: {"abcdefgh"}
  - Classe 4: {"AA"}
  - Classe 5: {"ABCDE123456789012345678 90"}
  - Classe 6: {"ab12312312"}
  - Classe 7: { "1asd"}

Testes Caixa Preta – Análise de Limites

- Muitos erros comuns de programação ocorrem nas condições limites
  - Ex.: for(i = 0; i **<** x; i++) ou for(i = 0; i **<=** x; i++) ?
- O critério de Análise de Limites tem como objetivo usar entradas próximas aos limites para exercitar a checagem dessas condições
- Geralmente é usada para refinar as entradas criadas com o critério de Classes de Equivalência
- Também pode ser aplicado às saídas

Testes Caixa Preta – Análise de Limites

- Alguns *guidelines*:
  - Se uma condição sobre uma entrada /saída for um intervalo, defina:
    - Para cada limite (superior / inferior), três casos de teste:
      - Um logo abaixo do limite
      - Um logo acima do limite
      - Um em cima do limite
  - Se uma condição sobre uma entrada é do tipo "deve ser" (ou "é", "só pode", etc) defina casos de testes para os casos verdadeiro e falso
  - Se uma condição sobre uma entrada / saída for um conjunto ordenado (lista ou tabela) defina testes que foquem:
    - No primeiro e no último elemento
    - Conjunto vazio

Testes Caixa Preta – Análise de Limites

- Exemplo: Uma função que recebe uma string e verifica se esta corresponde a um identificador válido ou não. Um identificador é considerado válido se, e somente se, ele possui no mínimo 3 e no máximo 15 caracteres alfanuméricos, sendo que os dois primeiros caracteres são letras

- Aplicando o critério de Análise de Limites sobre o tamanho da string (condição 2):
  - Limite inferior: 3
    - Logo abaixo do limite inferior (AbLI): tamanho = 2 (inválido)
    - Logo acima do limite inferior (AcLI): tamanho = 4 (válido)
    - Em cima do (LI): tamanho = 3 (válido?)
  - Limite superior: 15
    - Logo abaixo do limite superior (AbLS): tamanho = 14 (válido)
    - Logo acima do limite superior (AcLS): tamanho = 16 (inválido)
    - Em cima do limite superior (LS): tamanho = 15 (válido?)

Testes Caixa Preta

- Combinando Classes de Equivalência e Análise de Limites:
  - É necessário ter um conjunto de casos de testes que cubra todas as classes de equivalência e todos os limites
  - Mas é importante que o conjunto de casos de testes seja minimal!
    - Se fizer um caso de teste para cada caso, o número aumentará rapidamente
    - É possível criar casos de testes que cubram mais de uma classe de equivalência ao mesmo tempo!

Testes Caixa Preta

- Exemplo: (a tabela está incompleta)

ID	Entrada	Classe válida e limite coberto	Classe inválida e limite coberto
1	"ABC1"	Classe 1, Classe 3(AcLI), Classe 6	
2	"ABC*"	Classe 3(AcLI), Classe 6	Classe 2
3	"A1"	Classe 1	Classe 4(AbLI), Classe 7
...			

Testes Caixa Preta

- Exercício:
  - Função char* substring(char *str, int begin)
  - Um função recebe uma string que contém apenas caracteres alfanuméricos e um inteiro, *begin,* e retorna uma string contendo os caracteres da string de entrada contidos (inclusive) entre begin e o último caractere
  - Ex.: substring(“hamburguer”, 4) == “urger”
  - Defina as classes de equivalência válidas e inválidas das entradas
  - Faça a análise de limite para a variável *begin*
  - Descreva um conjunto de casos de testes que exercite todas as classes de equivalência válidas e inválidas, bem como todos os limites das variáveis *begin* e *end*

FIM