

Aula Testes 2

Baseado em Arndt von Staa

Especificação

- Objetivo dessa aula
 - apresentar técnicas para teste caixa fechada
- Referência básica:
 - Capítulo 15

Sumário

- Critérios caixa fechada
- Exemplo: teste de data
- Critério: partição em classes de equivalência
- Exemplo: tabela de símbolos
- Exemplo: nomes C/C++
- Tabelas de decisão
- Exemplo: espelhamento de arquivos
- Considerações finais sobre testes

Critérios caixa fechada

- Como você testaria?
 - Verificador de data:
 - { ano , mês , dia } vale ou não?
 - Verificador de triângulos
 - { LadoA , LadoB , LadoC } é triângulo ou não?
 - de qual tipo – equilátero, isóceles ou escaleno?

Data: escolha não sistemática

1900 <= Ano	2000	2000	2100	2000	2000	2000
1 <= Mês <= 12	1	2	2	1	4	4
1 <= Dia <= 31	1	29	29	0	30	31
É legal	x	x			x	
É bissexto		x				
É ilegal			x	x		x

Data: escolha não sistemática

- O teste é completo?
- Como você mostraria isto?

Data: escolha sistemática

- Especificar e gerar os dados

Sempre deveria existir uma especificação, afinal a finalidade do teste é verificar se o programa corresponde à sua especificação

- *Ano* ≥ 1900
- primeiro *Dia* de cada *Mês* é 1
- *Jan, Mar, Mai, Jul, Ago, Out, Dez* têm 31 *Dias*
- *Abr, Jun, Set, Nov* têm 30 *Dias*
- *Fev* em ano não bissexto tem 28 *Dias*
- *Fev* em ano bissexto tem 29 *Dias*
- São anos *bissextos*:
 - No calendário Gregoriano: os anos divisíveis por 4, sendo que não são bissextos os anos divisíveis por 100 e não divisíveis por 400
 - $365 + 97 / 400 \text{ dias} = 365,2425$
 - **Ano Tropical** medido é aproximadamente 365,24219 dias
 - dá um erro de um dia a cada 3300 anos com relação ao calendário Gregoriano

Data: escolha sistemática

- == Ano errado < 1900 → 1899
- == Mês errado < 1 → 0
- == Mês errado > 12 → 13
- == Dia errado < 1 → 0
- == Dia correto >= 1 → 1 , 2 , 15
- == Dia janeiro correto <= 31 → 30 , 31
- == Dia janeiro errado > 31 → 32
- == Dia fevereiro
 - ano bissexto divisível por 4 e por 100 e por 400 correto <= 29 → 2000 , 28 , 29
 - ano bissexto divisível por 4 e por 100 e por 400 errado > 29 → 2000 , 30
 - ano não bissexto divisível por 4 e por 100 e não por 400 correto <= 28 → 1900 , 27 , 28
 - ano não bissexto divisível por 4 e por 100 e não por 400 errado > 28 → 1900 , 29
 - ano bissexto divisível por 4 e não por 100 e não por 400 correto <= 29 → 2004 , 28 , 29
 - ano bissexto divisível por 4 e não por 100 e não por 400 errado > 29 → 2004 , 30
 - ano não bissexto não divisível por 4 correto <= 28 → 2003 , 27 , 28
 - ano não bissexto não divisível por 4 errado > 28 → 2003 , 29
- == Dia março correto <= 31 → 30 , 31
- == Dia março errado > 31 → 32
- ...

Partição em classes de equivalência

- Identifique todas as condições dos dados de entrada descritas na especificação
- Identifique todas as condições dos resultados descritas na especificação
- Para cada uma das condições determine as suas partições
 - valores válidos
 - valores não válidos
 - enumerações válidas
 - resultado existe ou não existe na saída

Partição em classes de equivalência

- Verifique se existem condições compostas ligadas por operadores lógicos, ex. *and* e *or*
 - decomponha a condição composta em um conjunto de condições elementares
 - $1 \leq t \ \&\& \ t \leq 32$, resulta nas condições elementares
 - $1 == t$
 - $1 < t$
 - $t < 32$
 - $t == 32$
- Verifique se existem casos de teste ambíguos
 - é ambíguo quando o resultado **não permite discernir** entre a ocorrência ou não de uma ou mais condições
 - decomponha cada casos de teste ambíguo em diversos outros casos de teste
 - ou reformule o conjunto

Partição em classes de equivalência

- Verifique se existe algum caso de avaliando condições mascaradas
 - uma condição *A* mascara outra condição *B* quando a ocorrência da condição *A* torna impossível determinar se a condição *B* ocorreu ou não
 - decomponha este caso de teste em diversos outros casos de teste
 - ou reformule o conjunto
- Crie um conjunto de casos de teste valorado
 - ajuste os casos de teste ao critério de valoração, criando mais casos de teste se for necessário
 - no conjunto de casos de teste cada caso exercita pelo menos uma condição não exercitada nos demais casos de teste do conjunto

Classes de equivalência: exemplo 1

- Exemplo: pesquisa de símbolos em uma tabela
 - partições de dados de entrada:
 - tabela vazia
 - tabela contém exatamente um elemento
 - tabela contém n elementos
 - O resultado poderá ser:
 - símbolo existe
 - símbolo não existe na tabela
- Como é caixa fechada não temos como saber a organização interna da tabela

Classes de equivalência: exemplo 1

Caso	Conjunto	Valor	Existe	Não existe
1	vazio	A		x
2	A	A	x	
3	A	B		x
4	A B C	A	x	
5	A B C	B	x	
6	A B C	C	x	
7	A B C	D		x

Classes de equivalência: exemplo 2

- Nomes de elementos em C/C++

Condição	Válida	Não válida
Caracteres	(1) letra simples (2) dígito (3) sublinhado	(4) todos os outros
Tamanho	(5) ≥ 1 (6) ≤ 32	(7) $= 0$ (8) > 32
Dígito	(9) após primeiro	(10) primeiro

Classes de equivalência

- 4 ➔ Todos os outros caracteres: teste de intervalo

Caso	Condições	Símbolo
1	1 , 5=	A
2	1 , 3 , 5>	A_
3	7 (impossível testar)	nulo
4	1 , 2 , 5> , 6< , 9	B123
5	1 , 2 , 6< , 9	A1234567890BCDEFGHIJKLMNOPQRSTU
6	1 , 2 , 6== , 9	a1234567890bcdefghijklmnopqrstuv
7	1 , 2 , 3 , 8 , 9	_1234567890bcdefghijklmnWXYZwxyz_
8	10	1

Tabelas de decisão

- Contexto típico
 - grande volume de decisões compostas
 - cada combinação de decisões dispara um conjunto de zero ou mais ações
- Código utilizando um número grande de `if then else` aninhados pode se tornar complexo e difícil de verificar
- Este tipo de problema é facilmente resolvido com **tabelas de decisão**
 - tabelas de decisão têm por objetivo simplificar o entendimento de decisões compostas

Tabelas de decisão

	Col 1	Col 2	...	Col L
Cond 1				
Cond 2				

Cond n				
Ação 1				
Ação 2				

Ação k				

Tabelas de decisão

- Na figura anterior são identificadas n condições e k ações
- Cada uma das L colunas define uma combinação de condições que, se satisfeitas, ativam uma ou mais ações
- Usualmente as tabelas utilizam decisões binárias
 - V verdadeiro
 - F falso
 - — indiferente (tanto faz se verdadeiro ou falso)
- Decisões mais complexas podem ser convertidas em decisões binárias.

Tabelas de decisão: casos especiais

- Condições **mutuamente exclusivas** no máximo uma das condições poderá ser V
 - exemplo: a decisão ternária $x < N$, $x == N$ e $x > N$, pode ser implementada com as duas decisões binárias mutuamente exclusivas: $x < N$ e $x == N$
 - $x > N :: !(x < N) \ \&\& \ !(x == N)$
 - corresponde a 2^n colunas, n é o número de condições
- **Conjunto obrigatório** é um conjunto de 2 ou mais condições no qual pelo menos uma deve resultar em V
 - exemplo: a decisão ternária $x < N$, $x == N$ e $x > N$, forma um conjunto obrigatório em que cada condição é mutuamente exclusiva com as demais
 - corresponde a 2^n colunas

Tabelas de decisão: casos especiais

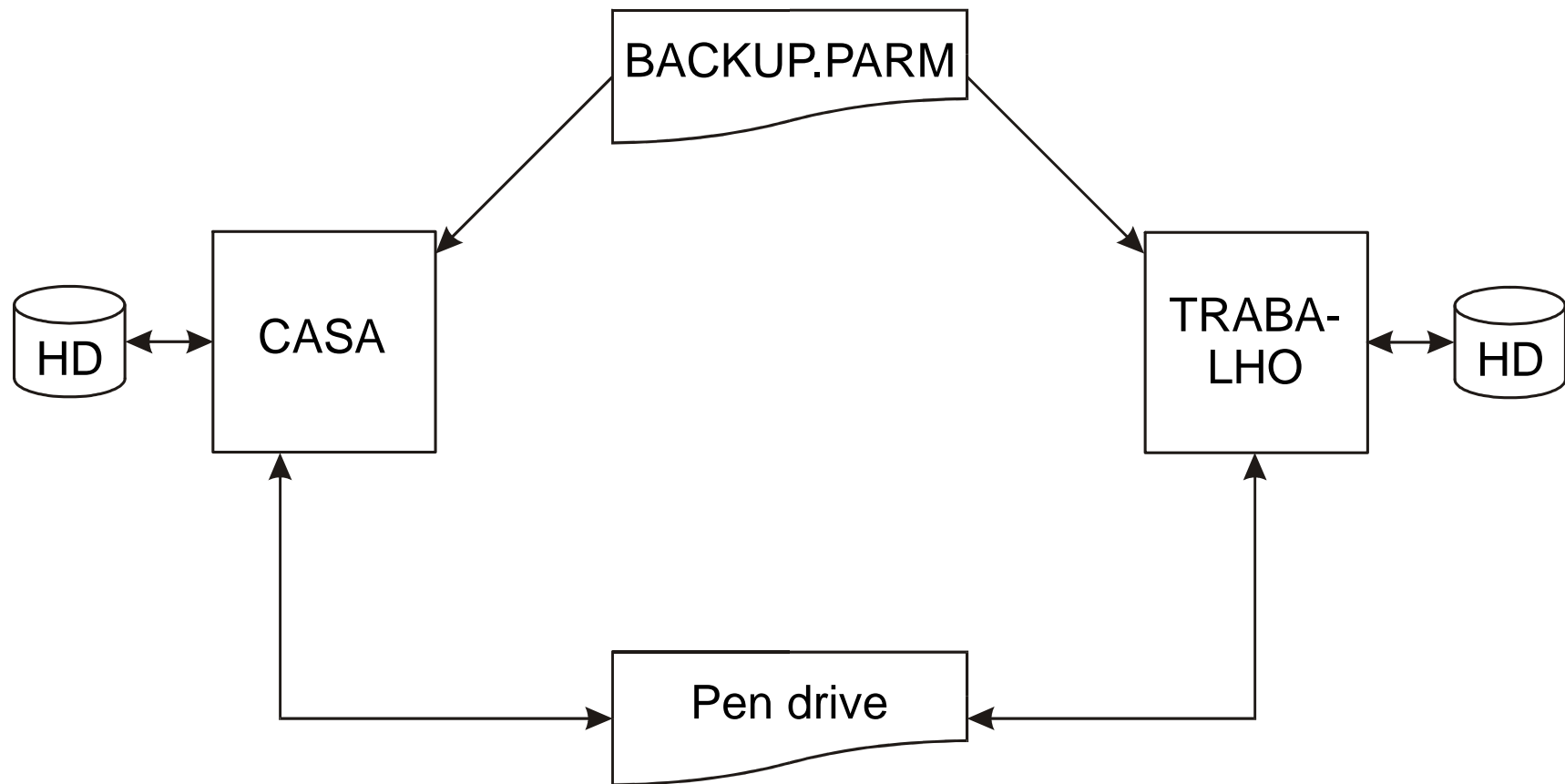
- **maskamento** ou **consequência de outra**, em um conjunto de decisões se uma resultar em verdadeiro outras também necessariamente resultarão em verdadeiro.
 - exemplo: se particionarmos os valores de uma variável em 4 faixas, teremos ao todo as seguintes 6 condições binárias:
 - $x < V1$, $V1 \leq x$, $x < V2$, $V2 \leq x$, $x < V3$, $V3 \leq x$
em que $V1 < V2 < V3$
 - se $x < V1$ então $x < V2$, ou seja mascara a condição
 - se $x < V1$ é mutuamente exclusivo com $V1 \leq x$

Tabelas de decisão: casos especiais

- Verificação da **ambigüidade**
 - uma coluna X é ambígua com relação a outra Y , se puderem ser selecionadas simultaneamente
- Verificação da **completeza** em tabelas com n decisões binárias:
 - no quadro completo devem existir $L == 2^n$ colunas
 - cada condição indiferente conta por duas alternativas.
 - cada conjunto de e condições mutuamente exclusivas e/ou obrigatórias conta 2^e colunas
 - caso a condição X seja conseqüência de outra condição Y , se Y for registrada verdadeira na coluna, X deverá ser registrada como indiferente.

Tabelas de decisão: exemplo

- Sistema de espelhamento
 - mantém cópias backup (quase) idênticas em vários computadores



Tabelas de decisão: exemplo

- Arquivo Backup.parm especifica que arquivos devem ser espelhados
 - diferentes arquivos Backup.parm geram diferentes formas de espelhamento
 - possivelmente um subconjunto do que existe no Pen-drive

Tabelas de decisão: exemplo

- Condições
 - existe Backup.parm
 - solicitado Backup: HD para Pen-drive
 - se falso foi solicitado restauração Pen-drive para HD
 - Para todos arquivos $ArqX \in Backup.Parm$
 - $ArqX \in HD$
 - $ArqX \in Pen-drive$
 - $Data(Pen-drive , ArqX) < Data(HD , ArqX)$
 - $Data(Pen-drive , ArqX) == Data(HD , ArqX)$
 - $Data(Pen-drive , ArqX) > Data(HD , ArqX)$

Tabelas de decisão: exemplo

- Ações
 - Salvar copiar arquivo do HD para o Pen-drive
 - Restaurar copiar arquivo do Pen-drive para o HD
 - Excluir excluir o arquivo contido no Pen-Drive
 - Erro erros, idErro
 - Nada faz nada
 - Impossível combinação condições ilegal, idErro
deve cancelar a execução do programa
- idErro – uma mensagem condizente com o erro

Tabelas de decisão: exemplo

Tem Backup.parm	F	V	V	V	V	V	V	V	V	V	V	V	V
Faz backup		V	V	V	V	F	F	F	F	V	V	F	F
ArqX ∈ HD		V	V	V	V	V	V	V	V	F	F	F	F
ArqX ∈ Pen-drive		F	V	V	V	F	V	V	V	F	V	F	V
Data PenD < HD			V	F	F		V	F	F				
Data PenD == HD			F	V	F		F	V	F				
Data PenD > HD			F	F	V		F	F	V				
HD para Pen-drive		X	X										
Pen-drive para HD									X				X
Excluir													
Faz nada				X				X			X		
Erro					X	X	X			X		X	
Impossível	X												

64 8 8 8 8 16 16

Considerações finais sobre testes

- Procure utilizar
 - **desenvolvimento cuidadoso** procurando chegar perto da **corretude por construção**
 - modelagem
 - padrões de projeto e de programação
 - revisão ou inspeção
 - uma **combinação de critérios** de geração de casos de teste
- Realize testes
 - pelo programador durante o desenvolvimento
 - procure sempre utilizar **teste automatizado**
 - procure sempre utilizar **integração automatizada** → make
 - por outros para fins de **teste de aceitação**
- Registre a história dos testes realizados
 - extraia estatísticas que possam informar sobre a eficácia dos testes

Considerações finais sobre testes

- Caso encontre um módulo que “não convirja”
 - reveja cuidadosamente a sua especificação e projeto
 - mais radical: jogue fora e faça de novo
- Sempre que encontre um erro de projeto ou de implementação
 - corrija este erro (*refactoring* obrigatório)
 - mesmo que isto adicione custos
- Lembre-se sempre que
 - o **dano provocado** por falhas encontradas durante o uso produtivo **pode ser muito elevado**
 - o **custo da diagnose** e depuração de sistemas em produção são bastante elevados
 - usualmente estes custos são muito maiores do que os custos adicionados pelo desenvolvimento e teste cuidadosos

FIM