Programming – Block C

http://www.win.tue.nl/~wstomv/2ip05/

Lecture 13

Tom Verhoeff

Technische Universiteit Eindhoven
Faculteit Wiskunde en Informatica
Software Engineering & Technology

Feedback to T.Verhoeff@TUE.NL

Today's Topics

- Implement an ADT, given its contract
- Workflow for ADT development
- Built-in checks using `Assert`
- Dynamic arrays in Object Pascal

Object Pascal `unit` structure

```
1  unit UnitName;
2    { purpose of this unit: ... }
3
4  interface
5    { externally visible definitions of constants, types,
6      procedure and function headers, variables }
7
8  implementation
9    { externally invisible implementation details }
10
11 initialization
12   { optional code called when application starts }
13
14 end.
```


Implementing an ADT in Object Pascal

- Representation of (internal) state, using **`private`** fields
- Private invariants for the state representation (when appropriate)
- Abstraction function relating representation to specification model (when appropriate)
- Bodies for methods

Rectangles ADT Implementation: Private Fields

```
1  unit Rectangles;
2    { provides ADT for axis-parallel grid rectangles }
3
4  interface
5
6  type
7    TRectangle = class(TObject)  // axis-parallel grid rectangles
8    private (* ignore implementation details *)
9      // fields
10     FXL: Integer; // lower X coordinate
11     FYL: Integer; // lower Y coordinate
12     FXH: Integer; // higher X coordinate
13     FYH: Integer; // higher Y coordinate
```


Rectangles ADT Specification: Constructor Contract

```
15   public
16     // construction
17     constructor Create(AXL, AYL, AXH, AYH: Integer);
18       // pre: AXL <= AXH  /\  AYL <= AYH
19       // post: XL=AXL /\ YL=AYL /\ XH=AXH /\ YH=AYH
```


Rectangles ADT Implementation: Constructor Bodies

```
63 implementation
64
65 uses
66   Math; { for Min and Max }
67
68 constructor TRectangle.Create(AXL, AYL, AXH, AYH: Integer);
69   begin
70     (* pre not checked *)
71     inherited Create
72   ; FXL := AXL
73   ; FYL := AYL
74   ; FXH := AXH
75   ; FYH := AYH
76   end; { Create }
```


Rectangles ADT Implementation: Basic Query Bodies

Interface (see **public** part in definition of **class** TRectangle)

```
21     // basic queries
22     function XL: Integer; // lower X coordinate
```

Implementation

```
78 function TRectangle.XL: Integer;
79   begin
80     Result := FXL
81   end; { XL }
```

Rectangles ADT Implementation: Derived Query Bodies

```
30    // derived queries
31    function Contains(AX, AY: Integer): Boolean;
32      // pre: true
33      // ret: XL <= AX <= XH  /\  YL <= AY <= YH
```

```
98  function TRectangle.Contains(AX, AY: Integer): Boolean;
99    begin
100     Result := (XL <= AX) and (AX <= XH) and
101               (YL <= AY) and (AY <= YH)
102   end; { Contains }
```


Rectangles ADT Implementation: Basic Command Bodies

```
38    //commands
39    procedure SetXL(AX: Integer);
40      // pre: AX <= XH
41      // effect: XL := AX
```

```
112 procedure TRectangle.SetXL(AX: Integer);
113   begin
114     (* pre not checked *)
115     FXL := AX
116   end; { SetXL }
```


Rectangles ADT Implementation: Derived Command Bodies

```
57    procedure Hull(const ARectangle: TRectangle);
58      // pre: true
59      // effect: Self := smallest rectangle enclosing
```

```
153 procedure TRectangle.Hull(const ARectangle: TRectangle);
154   begin
155     SetXL(Min(XL, ARectangle.XL))
156   ; SetYL(Min(YL, ARectangle.YL))
157   ; SetXH(Max(XH, ARectangle.XH))
158   ; SetYH(Max(YH, ARectangle.YH))
159   end; { Hull }
```

Why are the preconditions of all calls **`Set...`** satisfied?

Built-In Checks Using `Assert`

Design principle: Fail early, do not postpone

With pseudo procedure `Assert(P: Boolean; const Msg: string)`

Meaning: **`if not`** `P` **`then raise`** `EAssertionFailed.Create(Msg)` (an exception; runtime error 227), dus na `Assert(P, Msg)` geldt `P`

Assertion checking can be enabled/disabled:

- Lazarus: *Project* > *Compiler Options* > *Parsing* > *Include Assertion Code*
- FreePascal: `fpc -Sa ...`
- *{$C+}* or *{$ASSERTIONS ON}* *{$C-}* or *{$ASSERTIONS OFF}*

Example using `Assert`

```
1  program AssertExample;
2    { To illustrate the usage of Assert }
3
4  var
5    r: Real; { input }
6
7  begin
8    Write ( 'Give me a nonnegative number: ' )
9  ; ReadLn ( r )
10
11 ; Assert ( 0 <= r, 'This is negative!' )
12
13 ; WriteLn ( 'Its square root is ', Sqrt(r) : 1 : 2 )
14 end.
```


Example using `Assert` with `Format`

```
1  program AssertFormatExample; { with Format }
2
3  uses
4    SysUtils; { for Format }
5
6  var
7    r: Real; { input }
8
9  begin
10   Write ( 'Give me a nonnegative number: ' )
11 ; ReadLn ( r )
12 ; Assert ( 0 <= r, Format ( '%1.2F is negative!', [ r ] ) )
13 ; WriteLn ( 'Its square root is ', Sqrt(r) : 1 : 2 )
14 end.
```


Rectangles ADT Implementation: Built-in Checks (aux. func.)

```
31     // derived queries
32     function ToStr: String;
33       // pre: true
34       // ret: string representation of Self
```

```
68 uses
69   SysUtils, { for Format }
```

```
105 function TRectangle.ToStr: String;
106   begin
107     Result := Format('(%D, %D; %D, %D)', [XL, YL, XH, YH])
108   end; { ToStr }
```

`Format(`*format string*`, [`*value list*`])`

Rectangles ADT Implementation: Built-in Check for `Create`

```
72 constructor TRectangle.Create(AXL, AYL, AXH, AYH: Integer);
73   begin
74     // check pre
75     Assert((AXL <= AXH) and (AYL <= AYH),
76       Format('TRectangle.Create(%D, %D, %D, %D).pre failed',
77              [AXL, AYL, AXH, AYH]))
78   ; inherited Create
79   ; FXL := AXL
80   ; FYL := AYL
81   ; FXH := AXH
82   ; FYH := AYH
83   end; { Create }
```

N.B. Cannot use `ToStr`, because there is no object yet!

Rectangles ADT Implementation: Built-in Check for `SetXL`

```
124 procedure TRectangle.SetXL(AX: Integer);
125   begin
126     // check pre
127     Assert(AX <= XH,
128       Format('TRectangle.SetXL(%D).pre failed: XH = %D',
129              [AX, XH]))
130   ; FXL := AX
131   end; { SetXL }
```


Rectangles ADT Implementation: Built-in Check for `Intersect`

```
168 procedure TRectangle.Intersect(const ARectangle: TRectangle);
169   begin
170     // check pre
171     Assert(Intersects(ARectangle),
172       Format('TRectangle.Intersect(%S).pre failed: Self = %S',
173              [ARectangle.ToStr, ToStr]))
174   ; SetXL(Max(XL, ARectangle.XL))
175   ; SetYL(Max(YL, ARectangle.YL))
176   ; SetXH(Min(XH, ARectangle.XH))
177   ; SetYH(Min(YH, ARectangle.YH))
178   end; { Intersect }
```

Why are the preconditions of all calls **`Set...`** satisfied?

Alternative Rectangles ADT Implementation

Revised design decision: maintain lower-left corner and dimensions

(N.B. In Lazarus, the positive Y-axis points downward!)

What must be changed to adapt the implementation?

- Only the code *implementing* the ADT needs to be changed.
- The *contract* and all code *using* the ADT remains unchanged.

Alternative Rectangles ADT Implementation: Fields

```
7  type
8    TRectangle = class(TObject)  // axis-parallel grid rectangles
9    private
10     // fields
11     FXL: Integer; // lower X coordinate
12     FYL: Integer; // lower Y coordinate
13     FWidth: Integer; // width                          (*CHANGED*)
14     FHeight: Integer; // height                        (*CHANGED*)
15
16     // invariants                                      (*ADDED*)
17     // Positive: 1 <= FWidth  /\  1 <= FHeight         (*ADDED*)
18     // HighCoord: XH = FXL + FWidth - 1  /\            (*ADDED*)
19     //            YH = FYL + FHeight - 1               (*ADDED*)
```

Alternative Rectangles ADT Implementation: Constructor

```
77 constructor TRectangle.Create(AXL, AYL, AXH, AYH: Integer);
78   begin
79     // check pre
80     Assert((AXL <= AXH) and (AYL <= AYH),
81       Format('TRectangle.Create(%D, %D, %D, %D).pre failed',
82              [AXL, AYL, AXH, AYH]))
83   ; inherited Create
84   ; FXL := AXL
85   ; FYL := AYL
86   ; FWidth := AXH - AXL + 1                          (*CHANGED*)
87   ; FHeight := AYH - AYL + 1                         (*CHANGED*)
88   end; { Create }
```


Alternative Rectangles ADT Implementation: Basic Query

```
100 function TRectangle.XH: Integer;
101   begin
102     Result := FXL + FWidth - 1                       (*CHANGED*)
103   end; { XH }
104
105 function TRectangle.YH: Integer;
106   begin
107     Result := FYL + FHeight - 1                      (*CHANGED*)
108   end; { YH }
```


Alternative Rectangles ADT Implementation: Command

```
129 procedure TRectangle.SetXL(AX: Integer);
130   begin
131     // check pre
132     Assert(AX <= XH,
133       Format('TRectangle.SetXL(%D).pre failed: XH = %D',
134              [AX, XH]))
135   ; FWidth := XH - AX + 1  { must be done first }    (*ADDED*)
136   ; FXL := AX
137   end; { SetXL }
```


Alternative Rectangles ADT Implementation: Command

```
167 procedure TRectangle.Shift(AX, AY: Integer);
168   begin
169     FXL := FXL + AX
170   ; FYL := FYL + AY
171                                                      (*DELETED*)
172                                                      (*DELETED*)
173   end; { Shift }
```

Summary of changes in data representation for `TRectangles`

- Contract and use of `TRectangles` has not changed
- Implementation has changed:
  - Some private fields changed: same memory usage
  - Some method bodies changed: simplified or more involved

Workflow for Development of ADT as Product

1. Describe and analyse requirements (informal)
2. Design class interface (public methods, parameters, incl. types)
3. Design contract (method pre/post, public invariants)
4. Design internal representation (private fields, private invariants, abstraction function)
5. Implement method bodies (incl. assertion checking)
6. Verify the whole ADT: code inspection and unit testing

Unit Test for ADTs: What to Test

- Test for correct functionality; input satisfies precondition
  - No exception → check postcondition
  - Exception → FAILURE
- Test for correct checking; input does not satisfy precondition
  - Exception → check kind of exception
  - No exception → FAILURE

Unit Test for ADTs: How to Test

- Manually (ad hoc, improvised) versus Automated
- Design test inputs, how to check output (test driver)
- Execute tests
- Inspect test results, take action if necessary

Bounded and Unbounded Stacks and Queues

- Contracts
- Manual GUI test driver
- Implementation of Stacks given
- Implementation of Queues missing

Dynamic Arrays in Object Pascal: Overview

- **`array of`** *type* index type is `Integer` from 0
- `SetLength` (*dyn. array var*, *number of items*)
  Elements are added/deleted at high end.
  Newly added elements are undefined.
- `Length` (*dyn. array var*)
  `Low` (*dyn. array var*) (actually always 0)
  `High` (*dyn. array var*)
- `Copy` (*dyn. array var*, *first index*, *number of items*)
  Result is newly allocated dynamic array.

Dynamic Arrays in Object Pascal: Example

```
1  program DynamicArrayExample;
2
3  var
4    a, b: array of Integer;   (* dynamic array declaration *)
5    i: Integer;   { to traverse a }
6
7  begin
8    SetLength(a, 10)          (* allocate memory for 10 items *)
9  ; for i := Low(a) to High(a) do a[i] := Sqr(i)
10 ; b := Copy(a, 3, 5)        (* copy 5 items from index 3 on *)
11 ; SetLength(a, 5)           (* adjust length of a downward *)
12 ; for i := Low(a) to High(a) do writeln(i:2, a[i]:3, b[i]:3)
13 ; SetLength(a, 0)           (* de-allocate all memory for a *)
14 ; SetLength(b, 0)           (* de-allocate all memory for b *)
15 end.
```


What Lies Ahead

- Pointers, dynamic variables
- Recursion: lists, trees