

Chapter 3 Programmer defined classes and objects

Problem solutions

Problem 3.1 Study Chapter 3

Identify the appropriate example(s) or section(s) of the chapter to illustrate each comment made in the summary above.

Problem 3.2 Double conveyor belts

Extend example 3.3 to work for a double conveyor belt system. One conveyor moves single items and boxes of six items, while the other moves single items only. Clicking down the Display button shows the totals for each conveyor, while releasing the button hides both totals (figure 19).

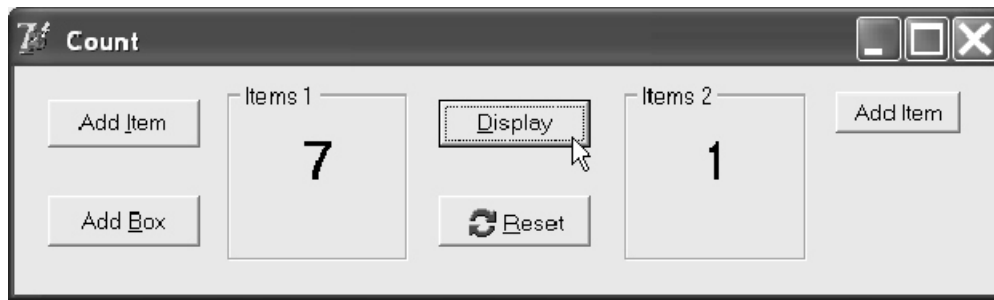


Figure 19 Extending the conveyor belt example

In writing this program place the definitions of TItem and TItemBox in separate units.

This problem gives an opportunity to work with more than one object of the same class and with packaging programmer-defined classes in separate units. We have made a few self-evident changes to the user interface component names from example 3.3 to avoid confusion with the additional components.

```

unit BoxCountU;

interface

{ Standard Delphi RAD interface declarations }

implementation

uses
  BoxU;                                     // Changed from ItemU

var
  ItemCount1, ItemCount2: TItemBox;

{$R *.dfm}

procedure TfrmCount.btnAddItem1Click(Sender: TObject);
begin
  ItemCount1.AddItem;
end; // end procedure TfrmCount.btnItemsClick

procedure TfrmCount.btnAddBox1Click(Sender: TObject);
begin
  ItemCount1.AddBox;
end; // end procedure TfrmCount.btnBoxClick

procedure TfrmCount.btnDisplayMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  lblTotal1.Caption := IntToStr(ItemCount1.GetCount);
  lblTotal2.Caption := IntToStr(ItemCount2.GetCount);
end; // end procedure TfrmCount.btnDisplayMouseDown

```

```

procedure TfrmCount.btnDisplayMouseUp(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  lblTotal1.Caption := '';
  lblTotal2.Caption := '';
end; // end procedure TfrmCount.btnDisplayMouseUp

procedure TfrmCount.bmbResetClick(Sender: TObject);
begin
  ItemCount1.ZeroCount;
  ItemCount2.ZeroCount;
end; // end procedure TfrmCount.bmResetClick

procedure TfrmCount.btnAddItem2Click(Sender: TObject);
begin
  ItemCount2.AddItem;
end; // end procedure TfrmCount.btnAddItem2Click

initialization
  ItemCount1 := TItemBox.Create;
  ItemCount2 := TItemBox.Create;

end. // end unit BoxCountU

unit ItemU;

interface

type
  TItem = class(TObject)
    protected // changed from private
      FCount: integer;
    public
      procedure AddItem;
      function GetCount: integer;
      procedure ZeroCount;
    end; // end TItem = class(TObject)

implementation

{ TItem }

procedure TItem.AddItem;
begin
  Inc(FCount);
end; // end procedure TItem.AddItem

function TItem.GetCount: integer;
begin
  Result := FCount;
end; // end function TItem.GetCount

procedure TItem.ZeroCount;
begin
  FCount := 0;
end; // end procedure TItem.ZeroCount

```

```

end. // end unit ItemU

unit BoxU;

interface

uses ItemU;

type
  TItemBox = class(TItem)
  public
    procedure AddBox;
  end; // end TItemBox = class(TItem)

implementation

{ TItemBox }

procedure TItemBox.AddBox;
const
  NoInBox = 6;
begin
  Inc (FCount, NoInBox);
end; // end procedure TItemBox.AddBox

end. // end unit BoxU;

```

Problem 3.3 Different box sizes

Consider another variation on the conveyor belt problem we have been looking at. In a warehouse there is a temporary storage area with three reversible conveyor belts. One carries individual items, the second carries small boxes of four items, and the third carries large boxes with twelve items. The screen looks like this (figure 20):

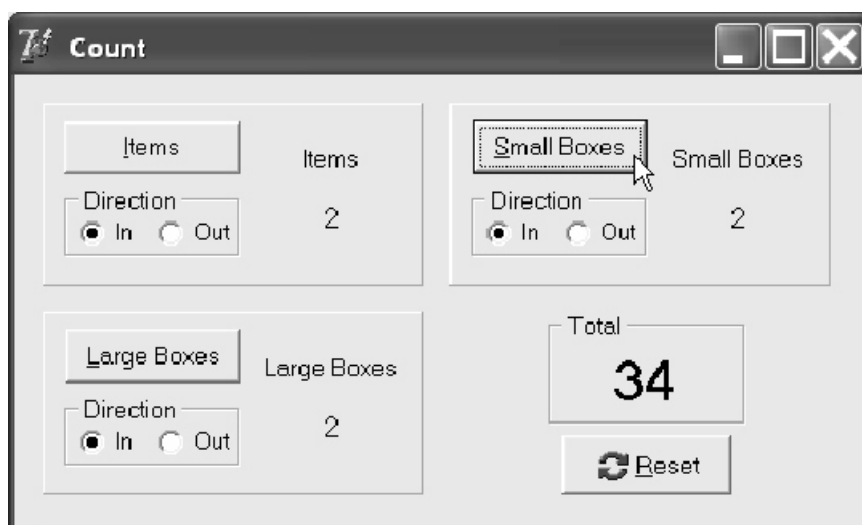


Figure 20 Different box sizes

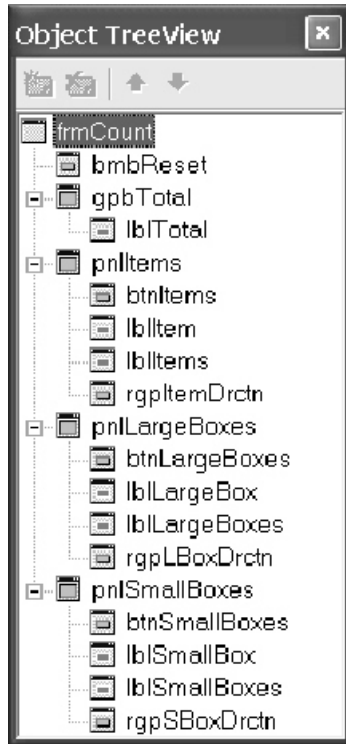


Figure 21 User interface components for different box sizes

There are several ways of structuring the application classes. One could, for example, use either a strongly hierarchical structure (figure 22) or a relatively flat structure (figure 23).

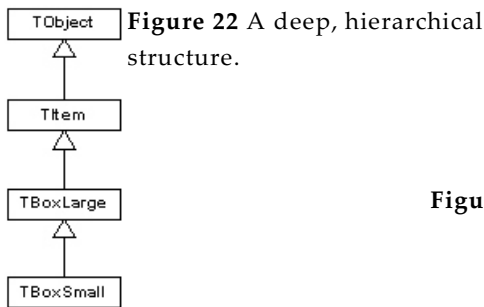


Figure 22 A deep, hierarchical structure.

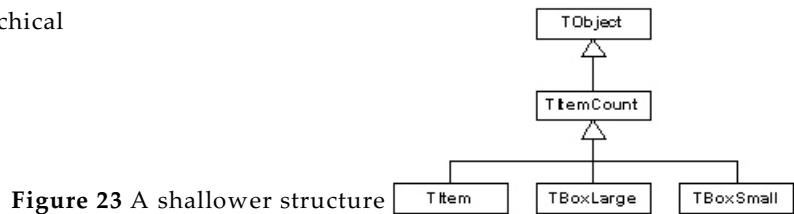


Figure 23 A shallower structure

1. Contrast these two approaches, describing the advantages and disadvantages of each.

Discuss these approaches along the following lines:

- Common advantage to both: Minimise repetitive coding through implementation reuse; code the similarities once and then code for differences in the subclasses.
- Deep advantage: Three classes versus the four needed for the shallow hierarchy, but

this is a relatively trivial consideration.

- Deep disadvantage: Going down the hierarchy adds unneeded (inherited) attributes & operations at each level. A small box is not a large box or an item, but inherits all the public attributes and behaviour of these. Another way of putting this is that items higher up the hierarchy are exposed to the items beneath them, and so TBoxSmall breaks the encapsulation of both TBoxLarge and TItem. (This is illustrated by the naming confusion TItemBox in example 3.3.)
- Shallow advantage: Concise packaging; only what is needed in each class, encapsulation maintained (so a small box is quite distinct from a large box, and both are quite distinct from an item).

2. Implement the deep hierarchy version.

```
unit BoxCountU;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, Buttons, ExtCtrls;

type
  TfrmCount = class(TForm)
    // standard RAD declarations
  private
    procedure DisplayValues;
  end; // end TfrmCount = class(TForm)

var
  frmCount: TfrmCount;

implementation

uses
  ItemU, BoxLargeU, BoxSmallU;

const
  Into = 0;
  OutOf = 1;

var
  ItemCount: TItem;
  LargeBoxCount: TBoxLarge;
  SmallBoxCount: TBoxSmall;

{$R *.dfm}
```

```

procedure TfrmCount.btnItemsClick(Sender: TObject);
begin
    if rgpItemDrctn.ItemIndex = Into then
        ItemCount.AddItem
    else if rgpItemDrctn.ItemIndex = OutOf then
        ItemCount.SubtractItem;
    DisplayValues;
end; // end procedure TfrmCount.btnItemsClick

procedure TfrmCount.btnLargeBoxesClick(Sender: TObject);
begin
    if rgpLBoxDrctn.ItemIndex = Into then
        LargeBoxCount.AddLargeBox
    else if rgpLBoxDrctn.ItemIndex = OutOf then
        LargeBoxCount.SubtractLargeBox;
    DisplayValues;
end; // end procedure TfrmCount.btnLargeBoxesClick

procedure TfrmCount.btnSmallBoxesClick(Sender: TObject);
begin
    if rgpSBoxDrctn.ItemIndex = Into then
        SmallBoxCount.AddSmallBox
    else if rgpSBoxDrctn.ItemIndex = OutOf then
        SmallBoxCount.SubtractSmallBox;
    DisplayValues;
end; // end procedure TfrmCount.btnSmallBoxesClick

procedure TfrmCount.DisplayValues;
var
    TotalCount: integer;
begin
    lblItems.Caption := IntToStr(ItemCount.GetCount);
    lblLargeBoxes.Caption := IntToStr(LargeBoxCount.GetCount);
    lblSmallBoxes.Caption := IntToStr(SmallBoxCount.GetCount);
    TotalCount := ItemCount.GetTotal +
        LargeBoxCount.GetTotal + SmallBoxCount.GetTotal;
    lblTotal.Caption := IntToStr(TotalCount);
end; // end procedure TfrmCount.DisplayValues

procedure TfrmCount.bmbResetClick(Sender: TObject);
begin
    ItemCount.ZeroCount;
    SmallBoxCount.ZeroCount;
    LargeBoxCount.ZeroCount;
    DisplayValues;
end; // end procedure TfrmCount.bmbResetClick

initialization
    ItemCount := TItem.Create;
    LargeBoxCount := TBoxLarge.Create;
    SmallBoxCount := TBoxSmall.Create;

end. // end unit BoxCountU

```

```

unit ItemU;

interface

type
  TItem = class(TObject) // derived from TObject
  private
    FCount: integer;
    FTotals: integer;
  protected
    procedure AddItem (NoOfItems: integer); overload;
    procedure SubtractItem (NoOfItems: integer); overload;
  public
    procedure AddItem; overload;
    procedure SubtractItem; overload;
    function GetCount: integer;
    function GetTotal: integer;
    procedure ZeroCount;
  end; // end TItem = class(TObject)

implementation

{ TItem }

procedure TItem.AddItem;
begin
  AddItem(1);
end; // end procedure TItem.AddItem

procedure TItem.AddItem(NoOfItems: integer);
begin
  Inc(FCount, 1);
  Inc(FTotals, NoOfItems);
end; // end procedure TItem.AddItem(NoOfItems: integer)

procedure TItem.SubtractItem;
begin
  SubtractItem(1);
end; // end procedure TItem.SubtractItem

procedure TItem.SubtractItem(NoOfItems: integer);
begin
  Inc(FCount, -1);
  Inc(FTotals, -NoOfItems);
end; // end procedure TItem.SubtractItem(NoOfItems: integer)

procedure TItem.ZeroCount;
begin
  FCount := 0;
  FTotals := 0;
end; // end procedure TItem.ZeroCount

function TItem.GetCount: integer;
begin
  Result := FCount;
end; // end function TItem.GetCount

```

```

function TItem.GetTotal: integer;
begin
    Result := FTotal;
end; // end function TItem.GetTotal

end. // end unit ItemU

unit BoxLargeU;

interface

uses ItemU;

type
    TBoxLarge = class(TItem) // derived from TItem
    public
        procedure AddLargeBox; // Public access method
        procedure SubtractLargeBox;
    end; // end TBoxLarge = class(TItem)

implementation

{ TBoxLarge }

const
    NoInLargeBox = 12;

procedure TBoxLarge.AddLargeBox;
begin
    AddItem (NoInLargeBox);
end; // end procedure TBoxLarge.AddLargeBox

procedure TBoxLarge.SubtractLargeBox;
begin
    SubtractItem (NoInLargeBox);
end; // end procedure TBoxLarge.SubtractLargeBox

end. // end BoxLargeU

unit BoxSmallU;

interface

uses BoxLargeU;

type
    TBoxSmall = class(TBoxLarge) // derived from TBoxLarge
    public
        procedure AddSmallBox; // Public access methods
        procedure SubtractSmallBox;
    end; // end TItemBox = class(TItem)

implementation

{ TBoxSmall }

```

```

const
  NoInSmallBox = 4;

procedure TBoxSmall.AddSmallBox;
begin
  AddItem (NoInSmallBox);
end; // end procedure TBoxSmall.AddSmallBox

procedure TBoxSmall.SubtractSmallBox;
begin
  SubtractItem (NoInSmallBox);
end; // end procedure TBoxSmall.SubtractSmallBox

end. // end BoxSmallU

```

In the code defining each class (ie under the type declaration), note how each successive class is derived from the previous one, leading to the deep hierarchy.

```

In unit ItemU:      TItem = class (TObject)
In unit BoxLargeU: TBoxLarge = class (TItem)
In unit BoxSmallU: TBoxSmall = class (TBoxLarge)

```

Notice also that each successive class definition must use the unit defining the higher level unit in order to get access to the definition of the higher unit.

```

In unit BoxLargeU: uses ItemU;
In unit BoxSmallU: uses BoxLargeU;

```

Although the original question specified an order for the hierarchy, we could also have reversed the order for defining the large and small boxes, and instead have derived TBoxSmall from TItem, and in turn derived TBoxLarge from TBoxSmall.

For the sake of simplicity, this code allows the number of items to drop below zero. In a real-life application, one may need to ensure that there could not be more items leaving than entering, but that would depend on the details of the application.

3. Implement the shallow hierarchy version.

No change is necessary in unit BoxCountU from the previous version.

```

unit ItemCountU;

interface

type

```

```

TItemCount = class(TObject)                                // derived from TObject
private
    FCount: integer;
    FTotal: integer;
protected
    procedure AddItem (NoOfItems: integer);                // not yet public
    procedure SubtractItem (NoOfItems: integer);          // not yet public
public
    function GetCount: integer;
    function GetTotal: integer;
    procedure ZeroCount;
end; // end TItemCount = class(TObject)

implementation

{ TItemCount }

procedure TItemCount.AddItem(NoOfItems: integer);
begin
    Inc(FCount, 1);
    Inc(FTotal, NoOfItems);
end; // end procedure TItemCount.AddItem(NoOfItems: integer);

procedure TItemCount.SubtractItem(NoOfItems: integer);
begin
    Inc(FCount, -1);
    Inc(FTotal, -NoOfItems);
end; // end procedure TItemCount.SubtractItem(NoOfItems: integer);

procedure TItemCount.ZeroCount;
begin
    FCount := 0;
    FTotal := 0;
end; // end procedure TItemCount.ZeroCount

function TItemCount.GetCount: integer;
begin
    Result := FCount;
end; // end function TItemCount.GetCount

function TItemCount.GetTotal: integer;
begin
    Result := FTotal;
end; // end function TItemCount.GetTotal

end. // end unit ItemCountU

unit ItemU;

interface

uses ItemCountU; // use TItemCount in type declaration below

type
    TItem = class(TItemCount) // derived from TItemCount
    public

```

```

    procedure AddItem;           // public at this level of the hierarchy
    procedure SubtractItem;     // public at this level of the hierarchy
end; // end TItem = class(TItemCount)

implementation

{ TItem }

procedure TItem.AddItem;
begin
    inherited AddItem (1);      // instead of overload
end; // end procedure TItem.AddItem

procedure TItem.SubtractItem;
begin
    inherited SubtractItem (1); // instead of overload
end; // end procedure TItem.SubtractItem

end. // end ItemU

unit BoxSmallU;

interface

uses ItemCountU;              // use TItemCount in type declaration below

type
    TBoxSmall = class(TItemCount) // change derivation: from TItemCount
    public
        procedure AddSmallBox;    // Public access methods
        procedure SubtractSmallBox;
    end; // end TItemBox = class(TItemCount)

implementation

{ TBoxSmall }

const
    NoInBox = 4;

procedure TBoxSmall.AddSmallBox;
begin
    AddItem (NoInBox);
end; // end procedure TBoxSmall.AddSmallBox

procedure TBoxSmall.SubtractSmallBox
begin
    SubtractItem (NoInBox);
end; // end procedure TBoxSmall.SubtractSmallBox

end. // end BoxSmallU

unit BoxLargeU;

interface

```

```

uses ItemCountU;           // use TItemCount in type declaration below

type
  TBoxLarge = class(TItemCount) // change derivation: from TItemCount
  public
    procedure AddLargeBox;           // Public access method
    procedure SubtractLargeBox;
  end; // end TItemBox = class(TItemCount)

implementation

{ TBoxLarge }

const
  NoInBox = 12;

procedure TBoxLarge.AddLargeBox;
begin
  AddItem (NoInBox);
end; // end procedure TBoxLarge.AddLargeBox

procedure TBoxLarge.SubtractLargeBox;
begin
  SubtractItem (NoInBox);
end; // end procedure TBoxLarge.SubtractLargeBox

end. // end BoxLargeU

```

After having implemented these two versions you may find that you want to add to or modify your comments in part 1 of this question.

Comment:

Having implemented both these versions, let's look at the public methods for each declared class. First, we list the classes in the *deep hierarchy* version:

TItem	TBoxLarge	TBoxSmall
<i>(Own)</i>	<i>(Inherited from TItem)</i>	<i>(Inherited from TItem)</i>
procedure AddItem;	*procedure AddItem;	*procedure AddItem;
procedure SubtractItem;	*procedure SubtractItem;	*procedure SubtractItem;
function GetCount: integer;	function GetCount: integer;	function GetCount: integer;
function GetTotal: integer;	function GetTotal: integer;	function GetTotal: integer;
procedure ZeroCount;	procedure ZeroCount;	procedure ZeroCount;
	<i>(Own)</i>	<i>(Inherited from TBoxLarge)</i>
	procedure AddLargeBox;	*procedure AddLargeBox;
	procedure SubtractLargeBox;	*procedure SubtractLargeBox;

TItem	TBoxLarge	TBoxSmall
		<i>(Own)</i> procedure AddSmallBox; procedure SubtractSmallBox;

Methods shown with an asterisk are methods that that class exposes but that are not appropriate to it, and the further down the hierarchy one moves, the more these inappropriate methods accumulate. Using the asterisked methods for that class can lead to serious (semantic) errors. For example, TBoxSmall should not have an AddItem or AddLargeBox method since these operations are not associated with a small box.

We now create a similar table for the *shallow hierarchy* version:

TItem	TBoxLarge	TBoxSmall
<i>(Inherited from TItemCount):</i> function GetCount: integer; function GetTotal: integer; procedure ZeroCount;	<i>(Inherited from TItemCount):</i> function GetCount: integer; function GetTotal: integer; procedure ZeroCount;	<i>(Inherited from TItemCount):</i> function GetCount: integer; function GetTotal: integer; procedure ZeroCount;
<i>(Own)</i> procedure AddItem; procedure SubtractItem;	<i>(Own)</i> procedure AddLargeBox; procedure SubtractLargeBox;	<i>(Own)</i> procedure AddSmallBox; procedure SubtractSmallBox;

With the shallow hierarchy approach, each class has only the methods that are appropriate to it, and no class exposes inappropriately the methods of any other class. So now TBoxSmall no longer has an AddItem or AddLargeBox method. This is an important factor and is discussed in subsequent chapters of the notes.

Problem 3.4 Protected entry

a) Write a simple password-protected access system that initially has a single access gate.

The user interface for Gate 1 (figure 24) has:

- a text entry section, which show just asterisks when you type into it,
- a button to submit the password, and
- a response area.

If the password is correct, the response is 'Proceed' (figure 24). If the password is incorrect the response is 'Report to main desk'.



Figure 24 First level password

In accordance with the Separation of Concerns Principle, the user interface relies on a separate, application class (TVerify) to determine whether the password is correct or not (figure 26). The user interface sends a message to the application object giving a key (which represents the correct password) and the word entered by the use. The password verification object then returns True or False. Currently the verification algorithm is very simple: the user must enter the reverse of what the key is. So if the key is Mary, the user must enter yraM as the password. Figure 25 shows the components on the user interface and figure 26 gives a class diagram. Write this program. The password key should be Mary, with separate units for TfrmPassword (figure 24) and for TVerify.



Figure 25 The user interface components

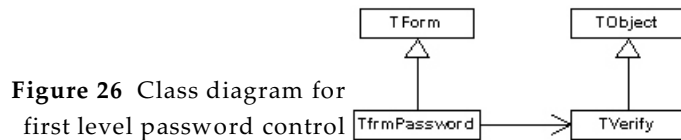


Figure 26 Class diagram for first level password control

b) This system must now be extended. The first access gate remains as before (figure 24). A second gate with a second, separate user interface like the interface of the first gate is added. This second access gate has a two stage password check. If the user enters the first password successfully at the second gate (similarly to part a), the system then displays an additional form requesting the user for a second, different password (figure 27). Only if this too is correct is the user given permission to proceed. Should the first password be incorrect, or if the first password is correct but the second is wrong, the user is told 'Report to the main desk' as in part a. The class diagram is shown in figure 28. Write this program. The first password key is Mary and the second key is George. The update program now has four

separate units, one each for the first gate (TfrmPassword, figure 24), for the second gate (TfrmPassword1 derived from TfrmPassword), for the additional form TfrmPassword2 (figure 27) and for TVerify.



Figure 27 The additional entry screen (for the second password)

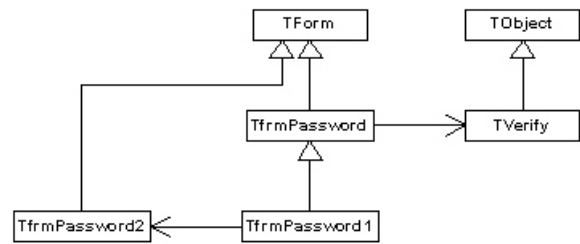


Figure 28 Class diagram for second level of password control

```

unit ProtectedEntryU;

interface

uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
    Forms, Dialogs, StdCtrls, VerifyU;

type
    TfrmPassword = class(TForm)
        edtPassword: TEdit;
        btnPassword: TButton;
        lblResult: TLabel;
        procedure btnPasswordClick(Sender: TObject);
        procedure FormCreate(Sender: TObject);
        procedure FormShow(Sender: TObject);
        procedure edtPasswordChange(Sender: TObject);
    protected
        FKey: string;
        Verifier: TVerify;
        procedure DisplayResult (Success: Boolean);
        function GetKey: string;
    end;

var
    frmPassword: TfrmPassword;

implementation

uses ProtEntryVFIu;

{$R *.dfm}

procedure TfrmPassword.btnPasswordClick(Sender: TObject);
begin

```

```

    DisplayResult(Verifier.Check (GetKey, edtPassword.Text));
end; // end procedure TfrmPassword.btnPasswordClick

procedure TfrmPassword.DisplayResult(Success: Boolean);
begin
    edtPassword.Clear;
    edtPassword.SetFocus;
    if Success then
        lblResult.Caption := 'Proceed'
    else
        lblResult.Caption := 'Report to main desk';
end; // end procedure TfrmPassword.DisplayResult

procedure TfrmPassword.FormCreate(Sender: TObject);
begin
    Verifier := TVerify.Create;
    FKey := 'Peter';
end; // end procedure TfrmPassword.FormCreate

function TfrmPassword.GetKey: string;
begin
    Result := FKey;
end; // end function TfrmPassword.GetKey

// only for VFI case
procedure TfrmPassword.FormShow(Sender: TObject);
begin
    // if (Sender as TForm).Name = 'frmPassword' then
    if Self = frmPassword then
        frmPassword1.Show;
end; // end procedure TfrmPassword.FormShow

procedure TfrmPassword.edtPasswordChange(Sender: TObject);
begin
    lblResult.Caption := '';
end; // end procedure TfrmPassword.edtPasswordChange

end. // end ProtectedEntryU

unit VerifyU;

interface

type
    TVerify = class(Tobject)
    public
        function Check (aKey, anEntry: string): Boolean;
    end; // end TVerify = class(Tobject)

implementation

{ TVerify }

function TVerify.Check(aKey, anEntry: string): Boolean;
{ This function checks whether anEntry is the reverse of aKey
(case sensitive) }

```

```

var
    Required: string;
    I: integer;
begin
    // Reverse aKey
    Required := '';
    for i := 1 to Length(aKey) do
        begin
            Required := aKey[I] + Required;
        end;
    // Check against anEntry
    Result := (anEntry = Required);
end; // end function TVerify.Check

(*
function TVerify.Check(aKey, anEntry: string): Boolean;
{ Alternative implementation:
  This function checks whether anEntry is the reverse of aKey
  (case sensitive) }
begin
    Result := (anEntry = ReverseString(aKey));
end; // end function TVerify.Check
*)

end. // end VerifyU

unit ProtEntryVFIu;

interface

uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
    Forms, Dialogs, ProtectedEntryU, StdCtrls, VerifyU;

type
    TfrmPassword1 = class(TfrmPassword)
        procedure btnPasswordClick(Sender: TObject);
        procedure FormCreate(Sender: TObject);
    protected
        FKey1: string;
        Password2: string;
    public
        procedure SetPassword2 (aPassword: string);
    end; // end TfrmPassword1 = class(TfrmPassword)

var
    frmPassword1: TfrmPassword1;

implementation

uses GetPwr2u;

{$R *.dfm}

procedure TfrmPassword1.btnPasswordClick(Sender: TObject);
begin

```

```

    if Verifier.Check (FKey, edtPassword.Text) then
    begin
        frmPassword2.ShowModal;
        DisplayResult (Verifier.Check(FKey1, Password2));
    end
    else
        DisplayResult(False);
end; // end procedure TfrmPassword1.btnPasswordClick

procedure TfrmPassword1.FormCreate(Sender: TObject);
begin
    inherited;
    FKey1 := 'Mary'; // hard coded for this problem
end; // end procedure TfrmPassword1.FormCreate

procedure TfrmPassword1.SetPassword2(aPassword: string);
begin
    Password2 := aPassword;
end; // end procedure TfrmPassword1.SetPassword2

end. // end ProtEntryVFIu

unit GetPwr2u;

interface

uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
    Forms, Dialogs, StdCtrls;

type
    TfrmPassword2 = class(TForm)
        edtPassword2: TEdit;
        btnPassword2: TButton;
        procedure btnPassword2Click(Sender: TObject);
    end; // end TfrmPassword2 = class(TForm)

var
    frmPassword2: TfrmPassword2;

implementation

uses ProtEntryVFIu;

{$R *.dfm}

procedure TfrmPassword2.btnPassword2Click(Sender: TObject);
begin
    frmPassword1.SetPassword2(edtPassword2.Text);
    edtPasword2.Text := '';
    Close;
end; // end procedure TfrmPassword2.btnPassword2Click

end. // end unit GetPwr2u

```