7.8. La classe “Mazzo di Carte” – Cenni alla gestione degli errori

Supponiamo di voler creare un programma che realizzi dei giochi di carte; a tal fine ci servirà una classe che rappresenti il mazzo di carte e, prima, ancora una classe che rappresenti una singola carta.
Il codice della classe Cls_Carta è mostrata di seguito:

```
Public Seme As Integer ' 1 Cuori, 2 Quadri, 3 Fiori, 4 Picche – In ordine d’importanza del Poker
Public Rango As Integer ' Asso = 1, Due = 2, …, 10 = 10, Jack = 11, Regina = 12 Re = 13
Private Semi(1 To 4) As String
Private Ranghi(1 To 13) As String

Private Sub Class_Initialize()
  ' Crea convenzionalmente la carta asso di cuori
  Rango = 1
  Seme = 1
  ' Crea i vettori dei semi e dei ranghi
  Semi(1) = "Cuori"
  Semi(2) = "Quadri"
  Semi(3) = "Fiori"
  Semi(4) = "Picche"
  Ranghi(1) = "Asso"
  For i = 2 To 10
    Ranghi(i) = CStr(i)
  Next i
  Ranghi(11) = "Fante"
  Ranghi(12) = "Donna"
  Ranghi(13) = "Re"
End Sub

Public Sub Crea(Sm As Integer, Rng As Integer)
  If Sm > 4 Or Sm < 1 Then Sm = 1
  Seme = Sm
  If Rng > 13 Or Rng < 1 Then Rnf = 1
  Rango = Rng
End Sub

Public Function Confronta(Carta As Cls_Carta) As Integer
  If Me.Seme < Carta.Seme Then Confronta = 1 'Sono in ordine inverso d'importanza (cuori = 1 più imp.)
  If Me.Seme > Carta.Seme Then Confronta = -1
  If Me.Seme = Carta.Seme Then
    If Me.Rango > Carta.Rango Then Confronta = 1
    If Me.Rango < Carta.Rango Then Confronta = -1
    If Me.Rango = Carta.Rango Then Confronta = 0
  End If
End Function

Public Sub Show()
  Debug.Print Ranghi(Me.Rango) & " di " & Semi(Me.Seme)
End Sub
```

La classe non ha nulla di particolare: ogni carta ha un seme ed un rango (proprietà), due variabili private a livello di classe che contengono la decodifica di seme e rango, un metodo di confronto e un metodo per la visualizzazione a video della carta. È inoltre presente l'evento Initialize e un semplice costruttore che imposta il valore delle due proprietà (seme e rango). Per quanto riguarda la codifica utilizzata, si noti che seme e rango sono definiti con i numeri naturali corrispondenti agli indici dei vettori Seme e Rango in cui è salvata la loro rappresentazione "letterale". Ad esempio Rango = 1 e Seme = 1 corrisponde all'Asso di cuori e, infatti, Rango(1) = Asso e Seme(1) = Cuore.

Vediamo ora <u>come usare la carta per generare un mazzo.</u>
L'idea di base è quella di usare una collection di carte contenente tutte e 52 le carte di un mazzo da poker. Quando si crea il mazzo ad ogni carta viene assegnata una chiave (key) composta da seme e rango tramite cui effettuare una ricerca nel mazzo (la collection). In altre parole la ricerca di una carta nel mazzo può essere fatta sia per indice che per chiave.
Una volta creato il mazzo è possibile: *(i)* verificare che una carta sia presente nel mazzo, *(ii)* inserire e rimuovere carte e, *(iii)* mescolare il mazzo; in quest'ultimo caso si pesca una carta in posizione casuale e la si reintroduce in un'altra posizione casuale e si ripete questo processo per un certo numero di volte.
Si noti che tre procedure (Prendi, Aggiungi e Is_In) fanno uso della gestione degli errori. Ad esempio, il metodo Prendi permette di specificare la posizione 'pos' del mazzo da cui prendere una carta. Tale parametro è opzionale e, di default, è settato a Null. All'interno del codice per prendere una carta si usa il metodo della collezione .Item(pos) che, come visto in precedenza, restituisce proprio l'oggetto in posizione 'pos' all'interno della collezione. Nel caso in cui 'pos' sia nullo e/o non corrisponda a nessun indice della collezione, tale metodo genera un errore e interrompe l'esecuzione del programma. Possiamo allora sfruttare tale errore per far sì che, tutte le volte in cui 'pos' è nullo o mal definito, venga automaticamente rimossa l'ultima carta del mazzo. A tal fine si usa l'espressione <u>**On Error Goto Err** che fa passare l'esecuzione del codice alla riga indicizzata con l'etichetta Err, non appena si genera un errore.</u> Nel nostro metodo, subito sotto tale etichetta abbiamo scritto le seguenti istruzioni che, come desiderato, specificano di restituire (e cancellare dal mazzo) l'ultima carta del mazzo:

```
Err:
  Set Prendi = Mazzo.Item(N_Carte) 'Pesco l'ultima
  Mazzo.Remove (N_Carte)
```

Prima di procedere oltre vediamo di formalizzare quanto detto, in relazione alla gestione degli errori. Osserviamo innanzitutto che, <u>ogni volta in cui si genera un errore</u>, se l'errore non è "gestito" via codice, <u>viene mostrato a video un messaggio d'errore che riporta due informazioni:</u>

- <u>Il numero dell'errore</u> (o meglio la proprietà Number dell'oggetto Err che si è generato)
- <u>La descrizione dell'errore</u> (o meglio la proprietà Description dell'oggetto Err)

Gestire gli errori significa in pratica: *(i)* anticipare i possibili errori che potrebbero verificarsi quando il codice verrà eseguito e *(ii)* scrivere del codice che permetta di gestire in maniera soft tali errori, senza che si verifichi un blocco del programma e senza che all'utente venga mostrata la finestra d'errore prima discussa.
<u>La prima cosa da fare per gestire gli errori è quella d'includere un'istruzione del tipo **On Error**</u>, possibilmente all'inizio della procedura e/o dell'istruzione che potrebbe generare l'errore.

In particolare, esistono quattro modi differenti di creare un'istruzione del tipo On Erro.
Queste sono:

1. **<u>On Error GoTo label</u>** – In questo caso, non appena si verifica un errore l'esecuzione del codice salta alla riga etichettata dalla parola label (che può essere scelta a piacere, nel caso del mazzo di carte, ad esempio, abbiamo utilizzato l'etichetta Err).
2. **<u>On Error Resume Next</u>** – In questo caso tutti gli errori vengono ignorati, le istruzioni che generano errori non vengono eseguite, e il codice prosegue alla linea di codice successiva.
3. **<u>On Error GoTo 0</u>** – Questa istruzione disabilita tutte le precedenti istruzioni di tipo On Error e la gestione degli errori viene nuovamente delegata al compilatore VBA.
4. **<u>On Error GoTo - 1</u>** – Questa istruzione "cancella l'errore" e permette di definire una nuova istruzione di tipo Or Error. In pratica le istruzioni On Erro valgono solo per il primo errore intercettato, se dopo la gestione dell'errore, se ne dovesse verificare un secondo questo non verrebbe gestito. Se volessimo farlo dovemmo prima "resettare" l'errore con l'istruzione On Erro Goto -1 e, successivamente, aggiungere ulteriori righe di gestione dell'errore.

Come detto, quando si usa l'espressione On Error Goto, in caso d'errore l'esecuzione passa alle righe di gestione errore specificate da un'etichetta di nostra scelta. Una volta che tali istruzioni sono state eseguite <u>è possibile specificare al codice da che punto del programma ricominciare. Ciò è possibile utilizzando l'istruzione **Resume**</u>. In pratica scrivendo:

- **<u>Resume</u>** - il codice torna al punto che aveva causato l'errore e prova ad eseguirlo nuovamente. Ad esempio, nel caso di una divisione per zero, potremmo modificare il valore (ad esempio mettendolo ad 1) e, solo a questo punto, far rieseguire la divisione;
- **<u>Resume Next</u>** – il codice torna al punto in cui si è verificato l'errore, salta le righe che avevano causato il problema, e prosegue con l'esecuzione di tutte le righe restanti;

5. **<u>Resume label</u>** – il codice riparte dalla riga etichettata con "label".

A fini di debug è anche possibile scrivere a video il numero e la descrizione dell'errore (le stesse che vedremmo nella finestra d'errore qualora non gestissimo l'errore stesso) mediante le seguenti istruzioni:

```
Debug.Print Err.Number
Debug.Print Err.Description
```

È anche possibile <u>visualizzare il numero della riga che ha generato l'errore mediante l'espressione **Erl**</u> (error line)

```
Debug.Print Erl
```

Tornando alla classe che stavamo discutendo, il codice della classe Cls_Mazzo è mostrato di seguito:

```
Private Mazzo As Collection

Private Sub Class_Initialize()
   Set Mazzo = New Collection
End Sub

Private Sub Class_Terminate()
   Set Mazzo = Nothing
End Sub
```

```
Public Sub Crea() ' Le 52 carte generate a partire dall’asso di cuori sino al re di picche
Dim s As Integer, r As Integer
Dim Carta As Cls_Carta
Dim Key As String
  For s = 1 To 4
    For r = 1 To 13
      Set Carta = New Cls_Carta
      Carta.Crea s, r
      Key = CStr(s) & "-" & CStr(r) ' La chiave di ricerca all’interno della collection
      Mazzo.Add Carta, Key
      Set Carta = Nothing
    Next r
  Next s
End Sub

Public Sub Show()
  If N_Carte = 0 Then
    Debug.Print "Il mazzo è vuoto"
    Exit Sub
  End If

  For Each Carta In Mazzo
    Carta.Show
  Next Carta
End Sub

Public Function N_Carte() As Integer
  N_Carte = Mazzo.Count
End Function

Public Sub Rimuovi(Seme As String, Rango As String)
  If Is_In(Seme, Rango) Then Mazzo.Remove (Seme & "-" & Rango) 'Metodo che usa la gestione degli errori
End Sub

Public Function Prendi(Optional pos As Variant = Null) As Cls_Carta
Dim Position As Integer
On Error GoTo Err: 'Se la posizione è errata o nulla si rimuove l’ultima carta del mazzo
  If Not IsNull(pos) Then Position = CInt(pos)
  Set Prendi = Mazzo.Item(Position)
  Mazzo.Remove (Position)
  Exit Function
Err:
  Set Prendi = Mazzo.Item(N_Carte) 'Pesco l'ultima
  Mazzo.Remove (N_Carte)
End Function
```

```
Public Sub Aggiungi(Carta As Cls_Carta, Optional pos As Variant = Null) 'Usa la gestione degli errori
Dim Key As String
Dim Position As Integer
On Error GoTo Err:
   If Carta Is Nothing Or Is_In(Carta.Seme, Carta.Rango) Then Exit Sub
   If Not IsNull(pos) Then Position = CInt(pos)
   Key = CStr(Carta.Seme) & "-" & CStr(Carta.Rango)
   Mazzo.Add Carta, Key, Position
   Exit Sub
Err:
   Mazzo.Add Carta, Key
End Sub

Public Function Is_In(Seme As String, Rango As String) As Boolean 'Metodo che usa la gestione degli errori
Dim Key As String
On Error GoTo Err: 'Se key esiste, Mazzo(key).Rango restituisce il rango, viceversa genera un errore
   Key = Seme & "-" & Rango
   A = Mazzo(Key).Rango
   Is_In = True 'Se non c'è errore vuol dire che la carta c'è
   Exit Function
Err:
   Is_In = False
End Function

Public Sub Mescola(Optional N As Integer = 3)
Dim i As Integer
Dim p As Variant
Dim C As Cls_Carta
   For i = 1 To 3 * N_Carte
      p = Application.WorksheetFunction.RandBetween(1, N_Carte)
      Set C = Prendi(p) 'La carta "pescata" che verrà reintrodotta in una nuova posizione
      p = Application.WorksheetFunction.RandBetween(1, N_Carte)
      Call Aggiungi(C, p)
      Set C = Nothing
   Next i
End Sub
```

Infine, per potere “giocare a carte”, è necessario generare delle “mani”. In pratica una mano è un mazzo con qualche proprietà e metodo addizionale. È pertanto intuitivo pensare di generare tale classe partendo dalla classe mazzo sviluppata in precedenza. I linguaggi orientati agli oggetti veri e propri facilitano questo compito implementando l’ereditarietà, tramite cui una nuova classe può ereditare tutte le proprietà e i metodi di un’altra classe (ancestor) e implementarne di nuovi e specifici. VBA non implementa tale approccio e, pertanto, il modo migliore per generare una classe a partire da un’altra è quella di includere tra le proprietà private della nuova classe il riferimento alla classe ancestor. In pratica la classe mano avrà una proprietà privata di tipo Cls_Mazzo.
Ciò è mostrato nel codice della classe Cls_Mano mostrato di seguito:

```
Private Mano As Cls_Mazzo
Public Nome As String

Private Sub Class_Initialize()
   Set Mano = New Cls_Mazzo
End Sub

Private Sub Class_Terminate()
            Set Mano = Nothing
End Sub

Public Sub Pesca(Da_Mazzo As Cls_Mazzo, Optional N As Integer = 1)
Dim i As Integer
   If Da_Mazzo.N_Carte > 0 Then
      For i = 1 To Application.WorksheetFunction.Min(N, Da_Mazzo.N_Carte)
         Mano.Aggiungi Da_Mazzo.Prendi
      Next i
   End If
End Sub

Public Sub Scarta(Seme As String, Rango As String)
   If Mano.Is_In(Seme, Rango) Then Mano.Rimuovi Seme, Rango
End Sub

Public Sub Show()
   If Mano.N_Carte = 0 Then
      Debug.Print "La mano è vuota"
      Exit Sub
   End If
   Mano.Show
End Sub

Public Function Gioca_Carta(pos As Variant) As Cls_Carta
   Set Gioca_Carta = Mano.Prendi(pos)
End Function
```

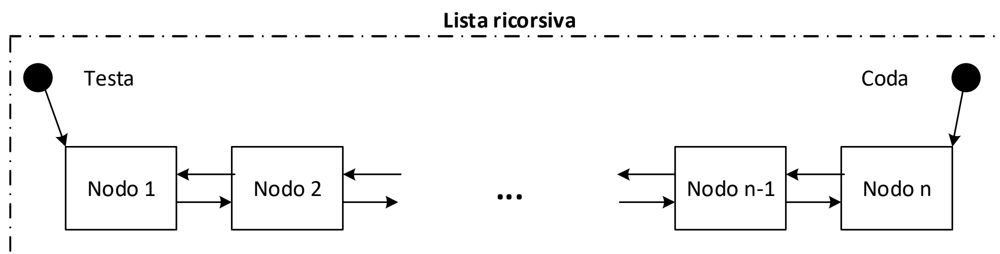
7.9. La classe "Coda LIFO"

Concludiamo gli esempi sulle classi mostrando l'implementazione di una coda LIFO (Last In First Out), un altro tipico esempio che ci permetterà anche di far nuovamente uso del concetto di ricorsione. In pratica dobbiamo immaginare la coda come una sorta di Array, ossia come una serie di elementi dello stesso tipo raggruppati secondo una certa logica d'impilamento. Nel caso della coda LIFO la regola d'impilamento e d'estrazione degli elementi contenuti nella coda è molto semplice: ogni volta che aggiungeremo un elemento alla coda tale elemento viene piazzato in cima alla coda e, pertanto, sarà anche il primo che lascerà la coda (sempre che, nel frattempo non si siano verificati nuovi arrivi). Per quanto detto quindi l'unico elemento che possiamo estrarre dalla coda è sempre quello in cima alla coda. Un esempio può essere un magazzino a catasta in cui i pallet vengono impilati uno sull'altro: per forza di cose, in questo caso, è possibile prelevare solo il pallet in testa alla coda.

Per implementare tale struttura ovviamente potremmo ricorrere ad una collection modificandola un po' i suoi metodi; tuttavia, a fini didattici, partiremo da zero ed implementeremo la coda mediante una lista ricorsiva.

Una lista ricorsiva è definita da una serie di nodi (che definiscono gli elementi della coda), ciascuno dei quali contiene un valore (o un oggetto) e ha un riferimento al nodo precedente e al nodo successivo (se esistenti). La lista avrà un nodo di testa (l'inizio della coda) e un nodo di coda (la fine della coda) che possono essere entrambi nulli (se la coda dovesse essere vuota), uguali (se la coda dovesse avere un solo elemento) e distinti (qualora la coda dovesse avere più elemento).

Una rappresentazione grafica della lista ricorsiva è mostrata nella figura seguente:



Per creare la nostra coda iniziamo a creare la classe nodo. In particolare, l'oggetto nodo avrà quattro proprietà pubbliche: il suo Id, il suo contenuto (valore) e il riferimento ad altri due nodi (variabili di tipo oggetto). Tali nodi sono il nodo di monte e il nodo di valle, rispettivamente. L'oggetto avrà anche un metodo per effettuare il collegamento (a valle o a monte) con un altro nodo e un metodo per visualizzare a video la sua descrizione.

Il codice completo è mostrato di seguito:

```
' **** LA CLASSE NODO ****
Public ID As String
Public Content As Variant
Public P_Node As Cls_Node
Public N_Node As Cls_Node

Private Sub Class_Initialize()  ' Un nodo vuoto senza riferimenti ad altri nodi né contenuto
   Set P_Node = Nothing  ' Il riferimento al nodo precedente
   Set N_Node = Nothing   ' e a quello successivo
   Content = Null
End Sub
```

```
Public Sub Link_Node(N As Cls_Node, Optional Nxt As Boolean = True)
  If Nxt Then
    Set N_Node = N 'Il nodo viene linkato a valle
    Set N.P_Node = Me 'Il nodo linkato vede Me ossia il nodo originario come nodo a monte
  Else
    Set P_Node = N
    Set N.N_Node = Me
  End If
End Sub

Public Function Representation() As String
Dim S As String
  On Error Resume Next
  If IsNull(Content) Or IsEmpty(Content) Then
    S = "Missing"
  Else
    S = CStr(Content)
  End If
  Representation = "Node " & ID & ", Value is : " & S
End Function

Public Sub Show()
  Debug.Print Representation
End Sub
```

Si noti l'uso dell'evento Class_Initialize usato per settare a Nothing i riferimenti ai due nodi di valle e di monte (quando il nodo viene creto, ovviamente, non è collegato ad altri).
Si noti anche l'uso del prefisso Me nel metodo usato per collegare un nodo ad un altro. Me (in altri linguaggi sostituito dal prefisso Self) serve ad indicare l'oggetto in questione. Supponiamo di aver creato un nodo N1 e un nodo N2 e di voler collegare N1 con N2, mettendo N2 a valle di N2. Usando il metodo Link_Node possiamo effettuare il collegamento nel modo seguente:

```
  N1.Link_Node(N2, True)
```

Ora tale metodo dovrà lavorare su entrambi i nodi, dato che N1 diventerà il nodo di monte di N2 e, analogamente, N2 diverrà il nodo di valle di N1. In pratica le proprietà P_Node e N_Node, rispettivamente, di N2 e di N1, andranno settate così:

```
Set N2.P_Node = N1
Set N1.N_Node = N2
```

Sorge però un problema, al metodo è stato passato (in input) solo N2, non N1. Tuttavia, N1 è l'oggetto stesso che invoca il metodo e quindi può essere richiamato dal metodo usando il prefisso Me che sta proprio ad indicare "la variabile oggetto che ha invocato il metodo". Questo spiega l'effettiva scrittura del codice (inerente al metodo Link_Node) riportato di seguito per completezza:

```
[…]
If Nxt Then
    Set Me.N_Node = N 'Il nodo viene linkato a valle (qui si può anche evitare di scrivere Me)
    Set N.P_Node = Me 'Il nodo linkato vede Me ossia il nodo originario come nodo a monte
Else
[…]
```

A questo punto possiamo creare la coda Lifo come una classe costituita da due nodi, la testa e la coda; ciò non vuol dire che la coda avrà solo due elementi, tutt'altro! Dato che ogni nodo ne referenzia al massimo altri due (quello precedente e quello successivo) è sufficiente tener traccia del primo e dell'ultimo per poter accedere a tutti gli altri (in effetti basterebbe tenere traccia del primo, ma la ricerca sarebbe più onerosa).
Il codice complessivo è mostrato di seguito:

```
' **** LA CLASSE CODA LIFO ****
Private Head As Cls_Node
Private Tail As Cls_Node

Private Sub Class_Initialize()
   Set Head = Nothing
   Set Tail = Nothing
End Sub

' Push aggiunge un nodo in fondo: i nuovi elementi si accodano
Public Sub Push(Node As Cls_Node)
   If Head Is Nothing Then
      Set Head = Node
      Set Tail = Node
   Else
      Tail.Link_Node Node
      Set Tail = Node
   End If
   Set Node = Nothing 'Che succederebbe senza questa riga?!? Provare a capire
End Sub

' Pop restituisce l'ultimo della coda. In una LIFO l'ultimo arrivato esce per primo
Public Function Pop() As Cls_Node
   If Tail Is Nothing Then
      Set Pop = Nothing
   Else
      Set Pop = Tail 'Si toglie l'ultimo
      If Pop.P_Node Is Nothing Then 'Se la coda ora è vuota …
         Set Head = Nothing
         Set Tail = Nothing
      Else
         Set Tail = Pop.P_Node 'La coda diventa il nodo a monte di quello tolto (pop)
      End If
   End If
End Function

' Remove cancella l'ultimo (ossia l'elemento di coda), senza restituire nulla
Public Sub Remove_Last()
Dim Last_Node As Cls_Node
   Set Last = Me.Pop
   Set Last = Nothing
End Sub
```

```
'La Vista Backward, che mostra a video la descrizione di tutti i nodi partendo dall'ultimo al primo
Public Sub Bkw_View()
Dim S As String
  If Tail Is Nothing Then
    Debug.Print "The List is empty"
  Else
    S = View(Tail)
    Debug.Print S 'Chiamata ad una funzione ricorsiva
  End If
End Sub

Private Function View(Node As Cls_Node) As String
  View = Node.Representation
  If Not (Node.P_Node Is Nothing) Then
    View = View & vbCrLf & View(Node.P_Node)
  Else  'L'assenza di altri nodi è la condizione di chiusura
    View = View & vbCrLf & "--- End ---" ' vbCrLf è il carattere di andata a capo
  End If
End Function
```

Anche in questo caso viene usato l'evento Class_Initialize per settare a nothing la testa e la coda della coda LIFO. Sono poi presenti due metodi Push e Pop rispettivamente per aggiungere, in fondo alla coda un nuovo arrivato e per far uscire dalla coda l'ultimo arrivato (la coda LIFO, come detto funziona proprio così).

Interessante l'ultimo metodo Bkw_View che mostra a video la descrizione di tutti i nodi partendo dall'ultimo (quello di coda) fino a risalire al primo (quello di testa). Tale metodo si basa su una funzione di stampa ricorsiva denominata View, che di fatto è molto simile alla funzione Impila considerata in un precedente paragrafo.

In questo caso, dato che stiamo operando su una struttura ripetitiva e simmetrica, la ricorsione funziona bene. In pratica, la funzione View riceve in input un nodo e, all'inizio, setta la variabile d'output al valore restituito dal metodo Node.Representation (che come abbiamo visto crea una stringa contenente tutte le sue informazioni). A questo punto si verifica se a monte del nodo considerato (indichiamolo con N(i)) ci sia un altro nodo (indichiamolo con N(i-1)) oppure no. Nel primo caso si concatena al valore della variabile d'output (che al momento contiene la descrizione di N(i)), con il carattere d'andata a capo e con la stringa descrittiva prodotta da una chiamata (ricorsiva) alla funzione View applicata a (N(i-1)). Viceversa, se N(i-1) non esiste, abbiamo raggiunto la condizione di chiusura (Escape) e la funzione si limita a restituire la descrizione di N(i) seguita dalla stringa “End”

7.10. Eventi

Gli eventi sono subroutine che esistono al di fuori del codice associato con la classe, che possono essere chiamate da istruzioni all'interno della classe. Gli eventi forniscono alla classe un modo per interrompere il programma che ha creato un'istanza dell'oggetto dalla classe, permettendo quindi al programma di eseguire le proprie operazioni in risposta a una situazione incontrata dall'oggetto. È importante tenere a mente che il codice associato a un evento in realtà risiede al di fuori della classe. Le uniche informazioni memorizzate all'interno della classe sono la definizione dell'evento con i parametri che saranno passati al programma esterno.

Detta così, la definizione di evento appare un po' fumosa. Vediamo di chiarire il tutto con un semplice esempio illustrativo.

Supponiamo di voler gestire dei prodotti commercializzati in un punto vendita. Ogni prodotto avrà un nome, un prezzo di listino e un prezzo minimo al quale può essere venduto. A ciascun prodotto può essere applicato uno sconto e deve essere possibile associare un prezzo totale a ciascun prodotto, pari al prodotto della quantità ordinata e del prezzo applicato (di listino o scontato). Per farlo è possibile definire la classe Cls_Prodotto che avrà the proprietà (nome, prezzo di listino, prezzo minimo) e due metodi calcola totale e sconta. Il primo metodo sarà pubblico, l'altro privato e verrà richiamato dal metodo pubblico per calcolare il prezzo complessivo da applicare. Osserviamo ora che, siccome ogni prodotto ha un prezzo minimo, non tutti gli sconti saranno ammissibili: in pratica lo sconto massimo applicabile sarà quello che fa scendere il prezzo di listino al prezzo minimo di vendita. Pertanto, nel caso in cui metodo che calcola il totale dovesse essere lanciato con uno sconto troppo alto si genererebbe un'incongruenza. In precedenza, abbiamo visto come sia possibile usare le routine di proprietà per effettuare controlli sugli input, ma in questo caso l'utilizzo di un evento è più corretto e intuitivo. In pratica vogliamo fare in modo che, qualora lo sconto applicato dovesse essere inammissibile, la classe mandasse un segnale di allarme intercettabile dal codice esterno alla classe che sta utilizzando un'istanza (un oggetto) della classe.

Per fare questo è necessario dichiarare, insieme alle proprietà pubbliche, l'evento che vogliamo generare. Si usa la seguente sintassi:

```
Public Event Nome_Evento([parametro di input 1], …)
```

Come si può vedere è una dichiarazione che sembra una via di mezzo tra la dichiarazione di una variabile e quella di una procedura. Infatti, se e quando l'evento si attiverà, esso dovrà essere generato con un certo numero di valori di input. Valori che potranno essere utilizzati dal codice esterno alla classe per prendere le dovute contromisure. Detto in altri termini, l'evento manda, all'esterno della classe, un segnale contenente delle variabili già valorizzate.

Nel nostro caso d'esempio, dato che l'evento verrà generato nel caso in cui sia stato applicato uno sconto troppo elevato, potrebbe aver senso includere nel segnale una variabile contenente il prezzo troppo (troppo basso) che risulterebbe dallo sconto e un messaggio d'errore. La dichiarazione dell'evento risulta allora la seguente:

```
Public Event ErrSconto(Value As Currency, Msg As String)
```

Fatto ciò, dobbiamo stabilire in che punto del codice introdurre l'istruzione che farà partire l'evento. La sintassi è la seguente:

```
Raise Event Nome_Evento([parametro di input 1], …)
```

Nel nostro caso metteremo il codice di attivazione nella procedura che calcola il totale, a condizione che il prezzo calcolato risulti troppo basso. In pratica andremo a scrivere qualcosa del genere:

```
  If Prezzo_calcolato < Prezzo_min Then
     RaiseEvent Errore_Sconto(Prezzo_calcolato, "Riduci Sconto")
     Exit Function
  End If
```

In questo modo rendiamo noto all'esterno il prezzo calcolato (troppo basso) e rilasciamo anche un messaggio d'errore.

Il codice completo è mostrato di seguito:

```
' CLASSE PRODOTTO – CHIAMATA Cls_Prodotto
Public Nome As String
Public Prezzo_listino As Currency
Public Prezzo_min As Currency
Public Event ErrSconto(value As Currency, Msg As String)  ' La dichiarazione dell'evento

Private Function Sconta(Optional Sconto As Double = 0.1) As Currency
Dim P_Scontato As Currency
   If Sconto < 1 And Sconto > 0 Then
      P_Scontato = Prezzo_listino * (1 - Sconto)
   Else
      P_Scontato = Prezzo_listino
   End If
Sconta = P_Scontato
End Function

Public Function Totale(Quantità As Integer, Optional Sconto As Double = 0)
Dim P As Currency
   P = Sconta(Sconto)
   If P < Prezzo_min Then
      RaiseEvent ErrSconto(P, "Riduci Sconto") ' Se avviene questa condizione, viene attivato l'evento
      Exit Function
   End If
   Totale = P * Quantità
End Function
```

Per chi ha progettato la classe il compito è finito. L'evento è stato definito ed è stato inserito un codice d'attivazione. Tuttavia, l'evento di per sé stesso non fa nulla, se non segnalare (tramite il rilascio di due variabili valorizzate) che qualcosa non è andato a buon fine[9]. Perché l'evento faccia qualcosa di produttivo è necessario intercettarlo scrivendo una funzione o una procedura esterna alla classe. Vediamo come fare. In pratica le cose da fare sono fondamentalmente due:

1. Creare un oggetto della classe che genera l'evento;
2. Creare una Sub che gestisce l'evento.

Per quanto riguarda il punto 1, si tratta di dimensionare l'oggetto usando, in aggiunta, l'operatore WithEvents (che fa sì che gli eventi generati siano anche visibili), come mostrato di seguito:

[9] Si osservi che gli eventi non sono necessariamente legati a fatti negativi e/o errori, possono essere legati a qualsiasi tipo di occorrenza, negativa o positiva che sia.

```
WithEvents Nome_Oggetto As Nome_Classe
```

Per quanto riguarda il punto 2, è necessario scrivere una subroutine con nome uguale a quello dell'oggetto creato seguito dall'underscore seguito dal nome dell'evento. Tale subroutine dovrà inoltre ricevere in input gli stessi parametri rilasciati (passati) dall'evento. Lo scheletro di tale Subroutine è pertanto il seguente:

```
Public|Private Sub NomeOggeto_NomeEvento(Lista parametri dell'evento)
        […]
End Sub
```

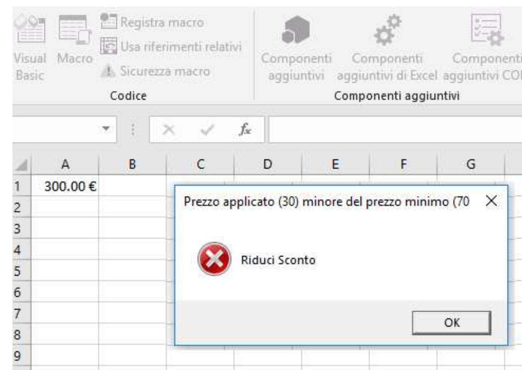
Vediamo d'implementare quanto detto in riferimento al nostro esempio. Per farlo, scriviamo una macro[10] nello spazio di lavoro di un foglio di lavoro, ridenominato, ad esempio, col nome "Prodotti".
In particolare, vogliamo che tale macro crei un oggetto prodotto (P1) e che in seguito calcoli il prezzo totale relativo a tre transizioni (definite in termini di numero di prodotti acquistati e di sconto effettuato). Il prezzo totale di tali transazioni dovrà essere scritto nelle celle A1, A2 e A3 del foglio di lavoro. Inoltre, nel caso in cui lo sconto sia eccessivo vogliamo che la macro intercetti l'evento e mostri un messaggio d'errore.
Il codice complessivo è mostrato di seguito:

```
Private WithEvents P1 As Cls_Prodotto ' In questo modo è possibile accedere all'evento
' Il codice che gestisce l'evento, una volta attivato.
Private Sub P1_ErrSconto(value As Currency, min As Currency, msg As String)
Dim Errore As String
  Errore = "Prezzo applicato (" & value & ") minore del prezzo minimo ("
  Errore = Errore & min
  MsgBox msg, vbCritical, Errore
End Sub

Public Sub Prova_Evento()
Dim T As Currency
  Range("A1:A3").Clear 'Range è un oggetto che rappresenta un range. Clear è un metodo per pulire il range
  'Creiamo il prodotto
  Set P1 = New Cls_Prodotto
  P1.Nome = "Prodotto 1"
  P1.Prezzo_listino = 100
  P1.Prezzo_min = 70
  'Calcoliamo il totale per 3 prodotti con sconto nullo
  T = P1.Totale(3, 0)
  Range("A1").value = T 'Tutto ok, in A1 viene scritto il valore della transazione eseguita
  'Facciamo la stessa cosa con uno sconto troppo alto
  T = P1.Totale(3, 0.7) 'Qui si scatena l'evento e viene richiamata la funzione P1_ErrSconto
  Range("A2").value = T 'Nella cella A2 viene scritto 0. Perché?
  'Facciamo la stessa cosa con uno sconto corretto
  T = P1.Totale(3, 0.1)
  Range("A3").value = T 'Tutto ok
  Debug.Print T
End Sub
```

[10] Una macro Excel è una sub pubblica, che non riceve parametri d'input, scritta nello spazio riservato di un foglio di lavoro

In pratica la prima e la terza transazione sono corrette e, pertanto, nelle celle A1 e A3 viene scritto il valore di tali transazioni (rispettivamente 300 e 270). Viceversa, lo sconto della transazione 3 è troppo alto e, pertanto, si scatena l'evento. Si noti, è importante, che non appena si scatena l'evento, dato che abbiamo creato l'oggetto con la clausola WithEvents, l'esecuzione del codice salta alla procedura legata all'evento (la procedura di gestione dell'evento), ossia la Private Sub P1_ErrSconto. Tale procedura provvede a scrivere un codice d'errore che viene mostrato a video tramite una message box, come mostrato nella figura seguente. Infine, una volta visualizzata la message Box, il codice riprende fino alla conclusione. In particolare, viene stampato il valore della seconda transazione, pari a zero (capire il perché è lasciato come esercizio al lettore) e, quello dell'ultima.



Esempio di esecuzione del codice