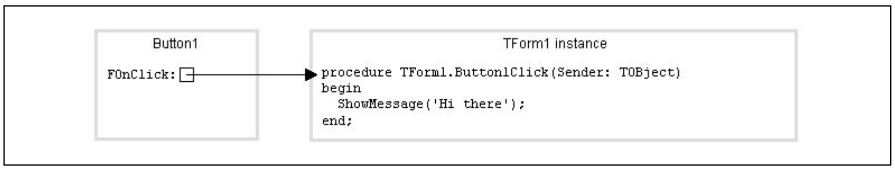
A Delphi Multicaster Class

by Max Rahder

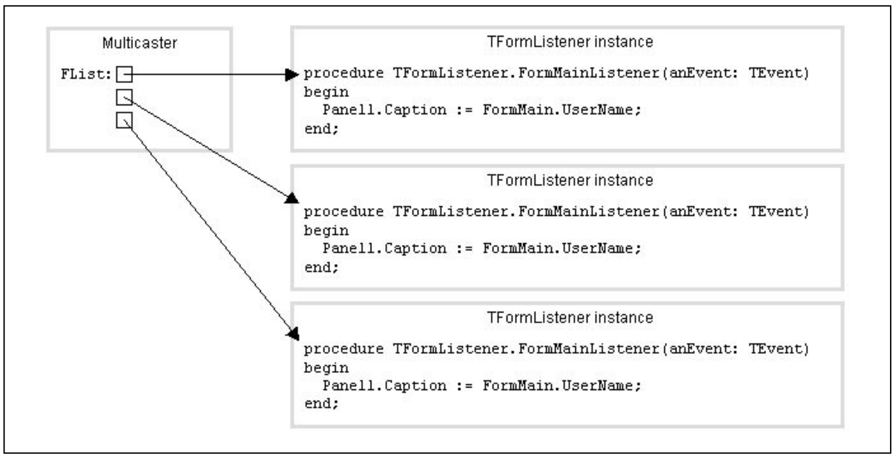
This article describes how to implement a multicaster class for Delphi. A multicaster allows many objects to detect events. The implementation is similar to exception handling: the multicaster 'listener' is passed an object and the listener checks the object's type to determine which event occurred. This design allows the programmer to listen to any event, such as a change in an object's state. For example, the multicaster could be used to have any object within an application detect changes to a form's caption. A multicaster can also be used to broadcast database changes, listeners can then determine whether their `TDataSet` objects need to be refreshed to reflect the changes.



```
procedure TForm1.Button1Click(Sender: TObject)
begin
  ShowMessage('Hi there');
end;
```

Above: Figure 1

Below: Figure 2



```
procedure TFormListener.FormMainListener(anEvent: TEvent)
begin
  Panel1.Caption := FormMain.UserName;
end;
```

```
procedure TFormListener.FormMainListener(anEvent: TEvent)
begin
  Panel1.Caption := FormMain.UserName;
end;
```

```
procedure TFormListener.FormMainListener(anEvent: TEvent)
begin
  Panel1.Caption := FormMain.UserName;
end;
```

The Delphi Event Model

Delphi events are 'singlecasters': an event is related to a single event handler. This is a good model because it is simple and meets the needs of most situations. Multicasting event notification can be found in Delphi. For example, a `TDataSource` detects changes to its associated dataset and notifies its associated data-aware controls. However, outside of creating one's own multicaster, there is no way to get this kind of notification for programmer-defined properties.

Figure 1 shows how a component refers to its event handler. In this example, a component named `Button1` runs a method named `Button1Click`. Each `TButton` component has its own set of instance variables, including the variable used to store the reference to the `OnClick` event handler.

Listing 1

```
TMethodReference = procedure of object;
TMethodReferenceList = class(TObject)
  private FList: TList;
  protected procedure Add(aMethodReference: TMethodReference);
  protected procedure Remove(aMethodReference: TMethodReference);
  protected procedure Clear;
  public constructor Create;
  public destructor Destroy; override;
  public procedure RemoveAllForAnObject(anObject: TObject);
```

Figure 2 shows how a multicaster stores references. An instance of `TEventMulticaster` has a `TList` which stores an arbitrary number of method references. Refer to this diagram as you read the implementation details in the following paragraphs.

Design And Implementation

There are three classes in the design: `TMethodReferenceList` is a simple collection of method references, `TEvent` is the base class for all events sent through the multicaster, and `TEventMulticaster` subclasses from `TMethodReference-List`, adding awareness of the `TEvent` class. Let's look at each class in turn.

TMethodReferenceList

`TMethodReferenceList` is a simple collection class. It will be extended in `TEventMulticaster` to add awareness of `TEvent`.

To add a method to the collection, the programmer runs `Add` passing a method reference. This method is the multicaster 'listener'. This is analogous to regular component events: a component event stores a reference to a method, called the event handler. The multicaster stores several method references, called multicaster listeners: see Listing 1.

Procedure references are addresses. This contrasts with a method reference, which is a procedure or function associated with a class. It may be more accurate to say that a method is a procedure or function associated with an object, because the reference must store the address of the procedure and the address of the object being referenced by `Self` within the procedure. This means that a method reference is actually two addresses: the address of the procedure and the address of the object's instance.

`TMethodReference` defines the signature of a generic method. The `Add` method allocates memory to store a record holding the reference, copies the address information to the record, then

```
procedure TMethodReferenceList.Add(aMethodReference: TMethodReference);
var
  pMethodReference: ^TMethodReference;
  i: integer;
begin
  // Look at each method in the collection to see if aMethodReference has
  // already been added.
  for i := 0 to (FList.Count - 1) do begin
    // Put the untyped FList pointer into the typed method reference pointer.
    pMethodReference := FList.Items[i];
    // Don't do anything if the method reference has already been stored.
    if (TMethod(pMethodReference^).Code = TMethod(aMethodReference).Code)
      and (TMethod(pMethodReference^).Data = TMethod(aMethodReference).Data) then
      Exit;
  end;
  // If we get this far we're adding a new method reference. First allocate
  // space to store the saved reference.
  New(pMethodReference);
  pMethodReference^ := aMethodReference;  //save method ref data in new space
  // Finally, save the address of the method reference data in the TList.
  FList.Add(pMethodReference);
end;
```

➤ *Above: Listing 2*

➤ *Below: Listing 3*

```
procedure TMethodReferenceList.RemoveAllForAnObject(anObject: TObject);
var
  pMulticasterEventListener: ^TEventListener;
  i: integer;
begin
  // Look at each method in the collection.
  for i := (FList.Count - 1) downto 0 do begin
    // Put the untyped FList pointer into the typed method reference pointer.
    pMulticasterEventListener := FList.Items[i];
    // If any procedure or function reference is associated with the passed
    // object then de-allocate its memory and remove the reference from FList.
    if (TMethod(pMulticasterEventListener^).Data = anObject) then begin
      Dispose(pMulticasterEventListener);
      FList.Delete(i);
    end;
  end;
end;
```

adds the address of the record to the TList, see Listing 2.

TMethod is a Delphi-defined record composed of two pointers named Code and Data. Code is the address of the procedure. Data is the address of the instance. (This may make it clearer why it's possible to run a method on an object reference containing NIL. The reference's data type determines the address of the method to be run. The method will run until it tries to reference Self, which will be set to the instance's address, which in this example will be NIL. Methods that don't reference Self should probably be declared as class methods.)

To check if the method has already been added, the code has to check for both the procedure's address and the object's address. In other words, if two objects are adding the same listener method to the multicaster they will have the same address for the method, but a different address for the object. The Add routine has to differentiate between them.

The Remove method does the opposite: it searches for the method reference, de-allocates the record, and removes the reference from the TList.

I have included a catch-all method named RemoveAllForAnObject, which removes any method associated with the object. The idea here is that an object may add many listeners to the collection in order to listen to different events. When the object is freed, it is convenient to remove all listeners at once rather than run Remove for each listener separately, see Listing 3.

TEvent

In this design, the code that sends a broadcast passes an object describing an event. For example, to send the event that a change took place the code might read as follows:

```
MyMulticaster.Broadcast(
  TMyEventAfterChange.Create(
  <data of interest to the
  listener>));
```

The listener would look something like Listing 4.

Therefore, we need to define an ancestor event class used by the multicaster. Sub-classes can add properties used to pass data to the listeners, see Listing 5.

The Sender property is needed because we want to save a reference to the object creating the event. That way the listener can listen to the broadcasts of several multicasters.

Different types of event are created by sub-classing TEvent:

```
TEventAfterChange =
  class(TEvent);
```

These sub-classes could add properties if necessary. For example, TEventAfterChange could add a property named OldValue. A database update event could add properties to pass the name of the table and key of the record being updated.

TEventMulticaster

TEventMulticaster is where we put the method collection together with the event class. TEventMulticaster sub-classes from TMethodList is the abstract class ancestor of TEventMulticaster. Other than RemoveAllForAnObject, TMethodList methods are protected.

Listeners must match the signature defined by TEventListener, see Listing 6.

AddListener and RemoveListener just call the ancestor's Add and Remove methods, typecasting the listener as TMethodReference.

```
TMyClass.MyListener(anEvent: TMyEventAbstractClass);
begin
  if (anEvent is TMyEventAfterChange) then
    <do something>
end;
```

➤ *Above: Listing 4*

➤ *Below: Listing 5*

```
TEvent = class(TObject)
  private FSender: TObject;
  public property Sender: TObject read FSender;
  public constructor Create(aSender: TObject);
end;
```

```
TEventListener = procedure(anMulticasterEvent: TEvent) of object;
TEventMulticaster = class(TMethodReferenceList)
  public procedure AddListener(anMulticasterEventListener: TEventListener);
  public procedure RemoveListener(anMulticasterEventListener: TEventListener);
  public procedure Broadcast(anMulticasterEvent: TEvent);
end;
```

➤ *Above: Listing 6*

➤ *Below: Listing 7*

```
procedure TEventMulticaster.Broadcast(aMulticasterEvent: TEvent);
var
  i: integer;
  pMulticasterEventListener: ^TEventListener;
begin
  try
    // Look at each method in the collection.
    for i := 0 to (FList.Count - 1) do begin
      // Put the untyped FList pointer into the typed method reference pointer.
      pMulticasterEventListener := FList.Items[i];
      // Run the method, passing the event.
      pMulticasterEventListener^(aMulticasterEvent);
    end;
  finally
    // When all methods have been told of the event free the event object.
    aMulticasterEvent.Free;
  end;
end;
```

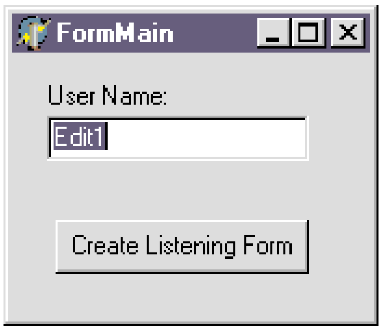
```
TFormMain = class(TForm)
    Edit1: TEdit;
    Button1: TButton;
    Label1: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure Edit1Change(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
    FMulticaster: TEventMulticaster;
    FUserName: string;
    procedure SetUserName(const aUserName: string);
  public
    public property UserName: string read FUserName write SetUserName;
    property Multicaster: TEventMulticaster read FMulticaster;
  end;
TEventUserNameChange = class(TEvent);
```

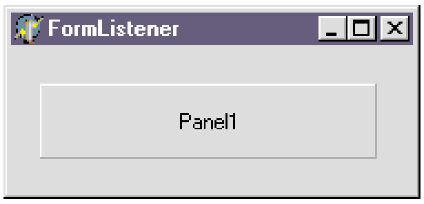
➤ *Above: Listing 8*

➤ *Below: Listing 9*

```
procedure TFormMain.SetUserName(const aUserName: string);
begin
  if (aUserName <> UserName) then begin
    FUserName := aUserName;
    Multicaster.Broadcast(TEventUserNameChange.Create(Self))
  end;
end;
```

➤ *Below: Figure 3*
Bottom: Figure 4





Broadcast iterates over the collection and runs each method, passing the TEvent object. After all methods are run, Broadcast destroys the object, see Listing 7.

The Broadcast method does something unusual: it deletes an object it doesn't create. Normally, good design practice dictates that the process that creates an object is responsible for destroying the object. The event model described in this article intentionally mimics the approach used by the exception handling model. Exceptions are raised (ie an exception object is created) to describe an event. The type of exception is checked in except on statements (ie they check the exception object's type). The destruction of exception objects is done as a side effect of the process, it's transparent to the programmer. This multicaster model 'raises' an event by creating a TEvent object in the argument of the Broadcast method. The event is detected using the as operator in the listener. The event object is freed automatically after all listeners have been run.

Example 1: Broadcasting Changes To An Object

This example has two forms. The main form, TFormMain, has a button used to create instances of the second form, TFormListener (which is not autocreated). See Figures 3 and 4 respectively. In a real world situation you'd associate the multicaster with a business object and the listener with any object that needs to detect changes to the business object's state, see Listing 8.

The constructor creates an instance of TEventMulticaster and stores it in FMulticaster. The Edit1Change event handler assigns the edit field's Text property to UserName. The UserName setter sends the broadcast, see Listing 9.

TFormListener has a single panel, see Listing 10. The TFormListener constructor adds its listener method, FormMainListener, to the FormMain multicaster, see Listing 11. The FormMainListener event checks to see what event was sent and updates the panel's caption to reflect the new UserName value, Listing 12.

If you run the application you'll see that each time you press the button on FormMain a new instance of TFormListener appears. If you type in the FormMain edit field, each listener form reflects the change as you type.

Example 2: A Multicaster On An Object Property

In practice it's sometimes useful to create a set of object oriented versions of Delphi's fundamental data types, such as TStringObject and TIntegerObject. These data types include a Multicaster property. In this scenario, the TFormMain UserName property is of type

TStringObject. The listeners would listen to UserName directly, rather than having a single multicaster to broadcast all of the TFormMain events. Doing this also means TFormMain no longer needs a UserName setter, because the multicaster takes care of broadcasting changes. See the code in Listing 13.

In this example, the TFormListener constructor references the UserName property's multicaster:

```
(Owner as TFormMain).UserName.
  Multicaster.AddListener(
  UserNameListener);
```

The TFormMain constructor needs to create and save an instance of TStringObject. The string object doesn't need to be explicitly destroyed because TStringObject sub-classes from TComponent. TComponent instances are automatically destroyed as their owning object's are destroyed. See Listing 14.

```
type
  TFormListener = class(TForm)
    Panel1: TPanel;
    procedure FormCreate(Sender: TObject);
  private
  private
    procedure FormMainListener(aMulticasterEvent: TEvent);
  public
  end;
```

➤ *Above: Listing 10*

➤ *Below: Listing 11*

```
procedure TFormListener.FormCreate(Sender: TObject);
begin
  (Owner as TFormMain).Multicaster.AddListener(FormMainListener);
end;
```

```
procedure TFormListener.FormMainListener(aMulticasterEvent: TEvent);
begin
  if (aMulticasterEvent is TEventUserNameChange) then
    Panel1.Caption := FormMain.UserName;
end;
```

➤ *Listing 12*

Example 3: A Global Database Update Multicaster

In some applications you need to be able to detect table changes throughout the application. For example, several forms may contain a TDBGrid showing customer orders. If the user opens a form to edit an order record and saves those changes, then it can be confusing to the user if the grids don't reflect the change. To handle this with a multicaster, create a global singleton object that contains a multicaster. Also create a new event class that has a TableName property that holds the name of the affected table, an

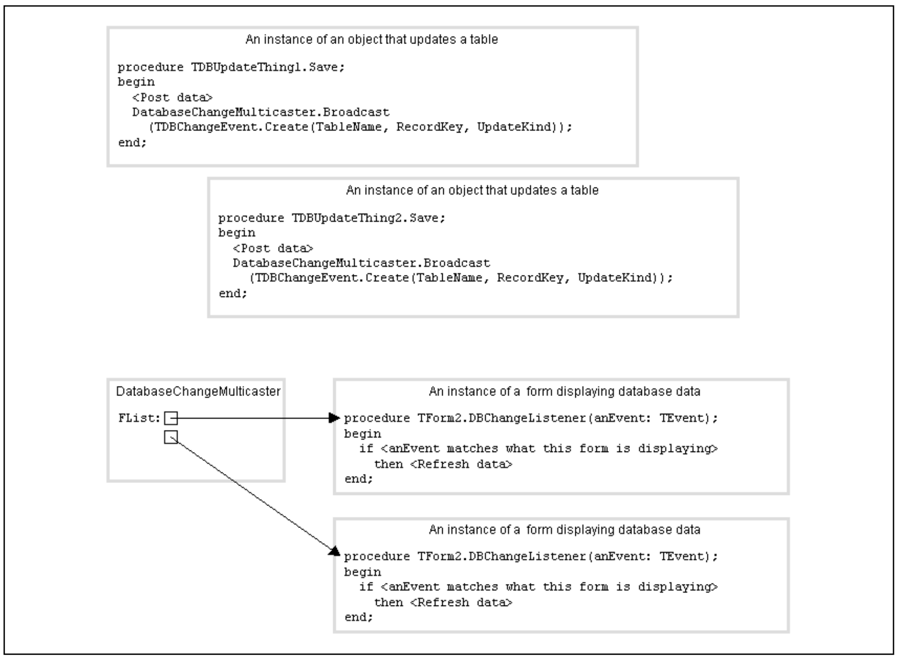
```
TEventAfterChange = class(TEvent);
  TStringObject = class(TComponent)
  private
    FValue: string;
    FMulticaster: TEventMulticaster;
    procedure SetValue(const aValue: string);
  public
    property Multicaster: TEventMulticaster read FMulticaster;
    property Value: string read FValue write SetValue;
    constructor Create(Owner: TComponent);
    destructor Destroy; override;
  end;
constructor TStringObject.Create(Owner: TComponent);
begin
  inherited;
  FMulticaster := TEventMulticaster.Create;
  FValue := '';
end;
destructor TStringObject.Destroy;
begin
  FMulticaster.Destroy;
  inherited;
end;
procedure TStringObject.SetValue(const aValue: string);
begin
  if (aValue <> Value) then begin
    FValue := aValue;
    Multicaster.Broadcast(TEventAfterChange.Create(Self));
  end;
end;
```

➤ *Above: Listing 13* ➤ *Below: Listing 14*

```
constructor TStringObject.Create(Owner: TComponent);
begin
  inherited;
  FMulticaster := TEventMulticaster.Create;
  FValue := '';
end;
```

```
interface
...
function DatabaseChangeMulticaster: TEventMulticaster;
...
implementation
...
var PrivateDatabaseChangeMulticaster: TEventMulticaster;
...
function DatabaseChangeMulticaster: TEventMulticaster;
begin
  if not(Assigned(PrivateDatabaseChangeMulticaster) then
    PrivateDatabaseChangeMulticaster := TEventMulticaster.Create;
  Result := PrivateDatabaseChangeMulticaster;
end;
...
finalization
  PrivateDatabaseChangeMulticaster.Free
```

➤ *Listing 15* ➤ *Below: Figure 5*


An instance of an object that updates a table
An instance of an object that updates a table
DatabaseChangeMulticaster
FList:
An instance of a form displaying database data
An instance of a form displaying database data

```
procedure TDBUpdateThing1.Save;
begin
  <Post data>
  DatabaseChangeMulticaster.Broadcast
    (TDBChangeEvent.Create(TableName, RecordKey, UpdateKind));
end;
```

```
procedure TDBUpdateThing2.Save;
begin
  <Post data>
  DatabaseChangeMulticaster.Broadcast
    (TDBChangeEvent.Create(TableName, RecordKey, UpdateKind));
end;
```

```
procedure TForm2.DBChangeListener(anEvent: TEvent);
begin
  if <anEvent matches what this form is displaying>
    then <Refresh data>
end;
```

```
procedure TForm2.DBChangeListener(anEvent: TEvent);
begin
  if <anEvent matches what this form is displaying>
    then <Refresh data>
end;
```

`UpdateKind` property that reflects whether the change was an insert, update, or delete, and a `RecordKey` property which stores the key of the changed record. Any form that updates a record would send a broadcast through the global multicaster. Forms containing the grids would add listeners to the multicaster. In their listeners, those forms would check to see if the changed table is being shown on the form, and if so, refresh the grid's data source.

In Delphi, singletons are created via an interface function (or class function). Using a function makes the reference read-only, see Listing 15.

Conclusion

The multicaster class makes it easy to integrate parts of an application by allowing any number of listeners to be aware of state changes in objects.

This can be used as the need arises, or can be built into an application's architecture at a more fundamental level. For example, if you had good separation between business objects and their user interface, then the business object's properties could be simple object types that include multicasters. The user interface would then add listeners for each property of interest, and update the interface as the property changes (in a user interface using data-aware controls this functionality is provided by the `TDataSource`).

Note that multicaster functionality can also be implemented through messaging. However, the approach described in this article is easier to debug because the code can be stepped through as it executes, and besides, this implementation should work fine with the Linux version of Delphi!

Max Rahder (www.rahder.org/max) is an independent consultant living in Madison, Wisconsin. He is a certified Delphi and JBuilder instructor. You can email Max at max@rahder.org