Unidad 1

Elementos de Interfaces Gráficas

Programación Orientada a Eventos

Concepto de Evento

- Un evento es resultado de la ejecución de una acción dentro de un entorno gráfico.
- Los eventos pueden ser generados por interacción del usuario con la GUI o por disparo mediante programación.
- Los eventos son objetos. En una ventana, los eventos representan acciones como hacer click a un botón, el movimiento del ratón, la captura de texto en un campo, etc.
- Cualquier interfaz gráfica constantemente monitorea los eventos en el sistema e informa de estos a los programas que se están ejecutando. Cada programa determina como reaccionar en respuesta a esos eventos.

Esquema del Manejo de Eventos

Para el manejo de eventos se requiere un esquema compuesto de los siguientes elementos **modelados como objetos**.

- **Origen del evento:** Típicamente son los componentes gráficos (botones, opciones de menú, cuadros de texto, etc) con los que interactúa el usuario.
  Al efectuar una acción sobre el componente (dar click, dar enter, etc) se **dispara** (crea) un objeto que encapsula dicha acción.
- **Evento:** Objeto que encapsula la información relacionada al objeto que lo originó (su nombre, por ejemplo) y las condiciones de la acción realizada.
- **Oyente del evento:** Cumple la función de "estar atentos escuchando" si se dispara un evento. Cuando un evento se dispara y es escuchado por el oyente, este puede ejecutar una o varias acciones para responder al evento.

Debido a que en una interfaz gráfica pueden existir multiples origenes de eventos, también pueden existir varios oyentes para atender de forma particular a cada origen. Esto tiene como beneficio que se delimitan las responsabilidades de cada oyente con mayor claridad.

Por lo anterior, es muy importante asociar a cada origen su correspondiente oyente. Esta operación se conoce como "registro del oyente al origen". Es claro que un oyente puede ser registrado a uno o varios componentes.

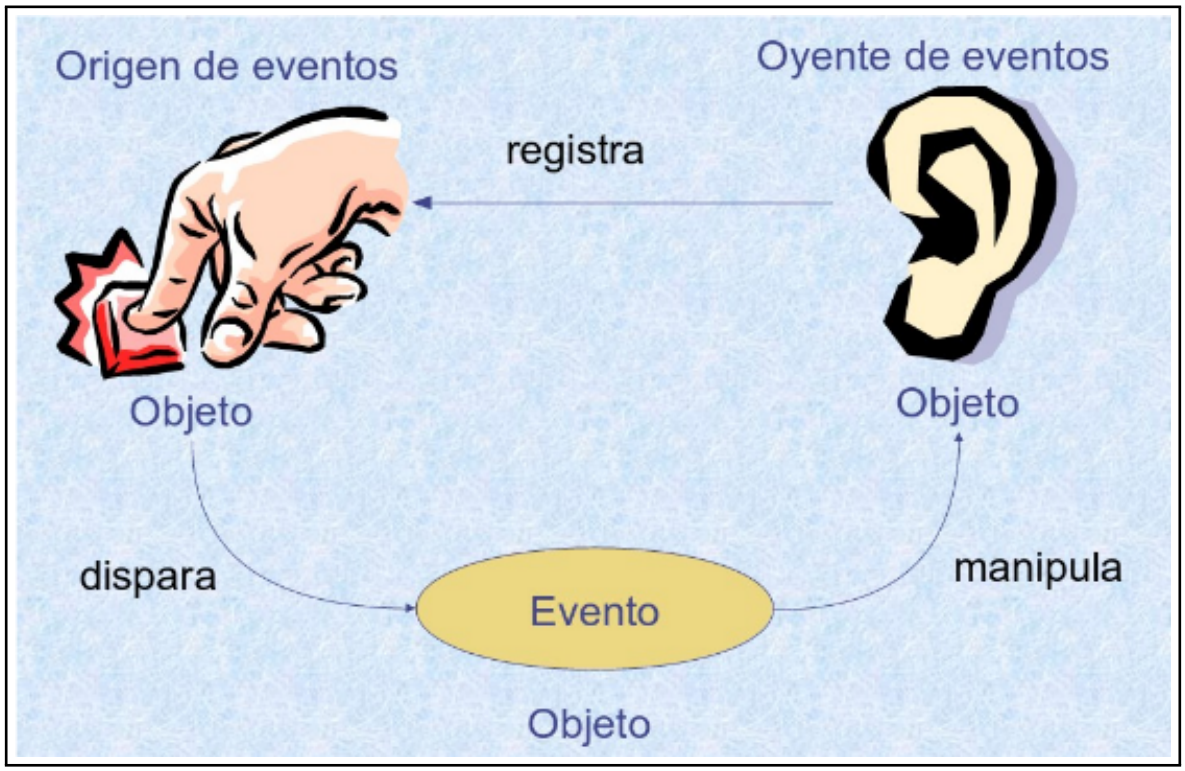

Figura 1.1: Esquema general para manejar los eventos

Clasificación de Eventos

El esquema de eventos de JavaFX es distinto en diseño y funcionamiento al de sus predecesores Swing y AWT. Se integra de dos clases: Event e EvenType, y una interface (clases con prototipos de métodos), EventHandler.

Event es la clase base que modela los eventos. Todos los eventos definidos en JavaFX extienden de Event, véase la figura 1.2. Toda instancia de Event tendrá asociado su origen (objeto que desencadenó el evento), un target (camino que seguirá el evento al ser disparado) y uno o más subtipos que son modelados por la clase EventType.

Cada subclase de Event es en sí un tipo de evento distinto, los tipos modelados por EventType son útiles para clasificar específicamente eventos que pertenecen a una misma subclase de

Event. Por ejemplo, la clase MouseEvent que modela los eventos de mouse que suceden sobre un componente puede especificarse en los eventos: botón presionado, botón liberado, click en mouse, movimiento del mouse, etc. Esta forma de subclasificar eventos se emplea agregando atributos estáticos a la subclase de Event en cuestión, véase la figura 1.3. La tabla 1.1 tiene una relación de los eventos y sus subtipos más relevantes.

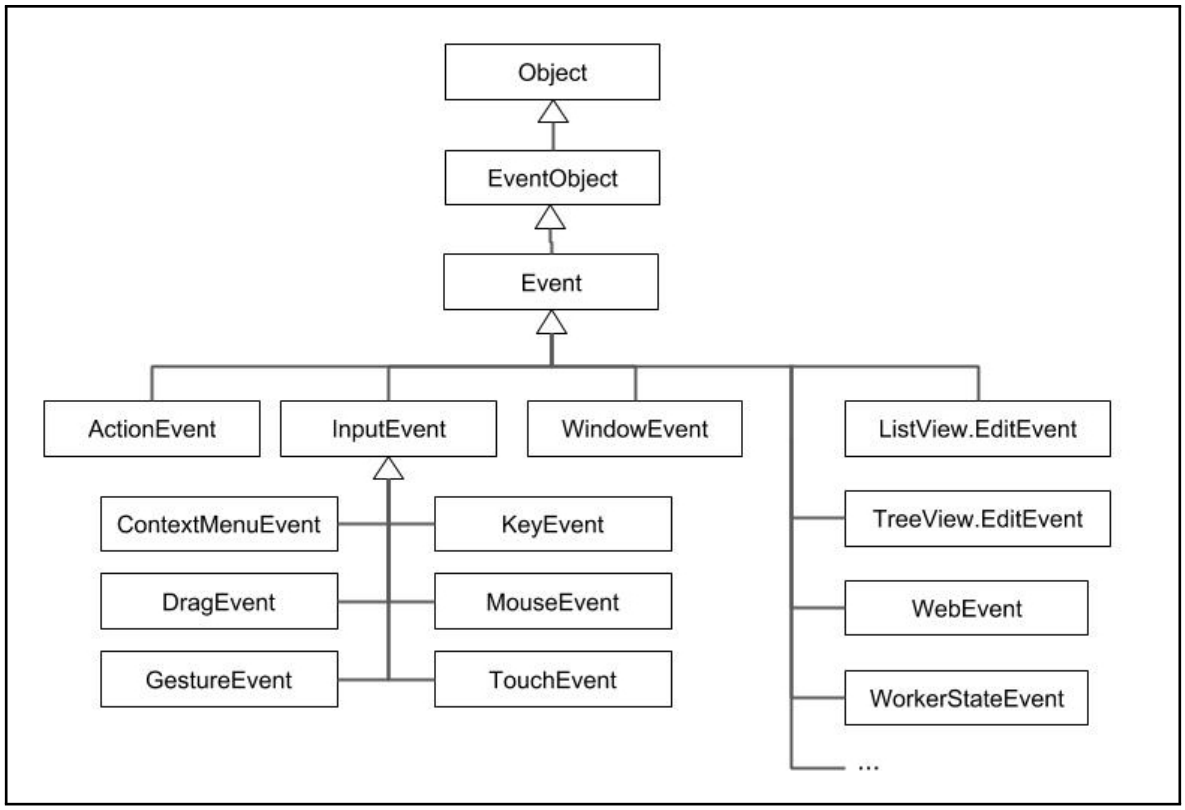

Figura 1.2: Diagrama de clases de eventos en JavaFX

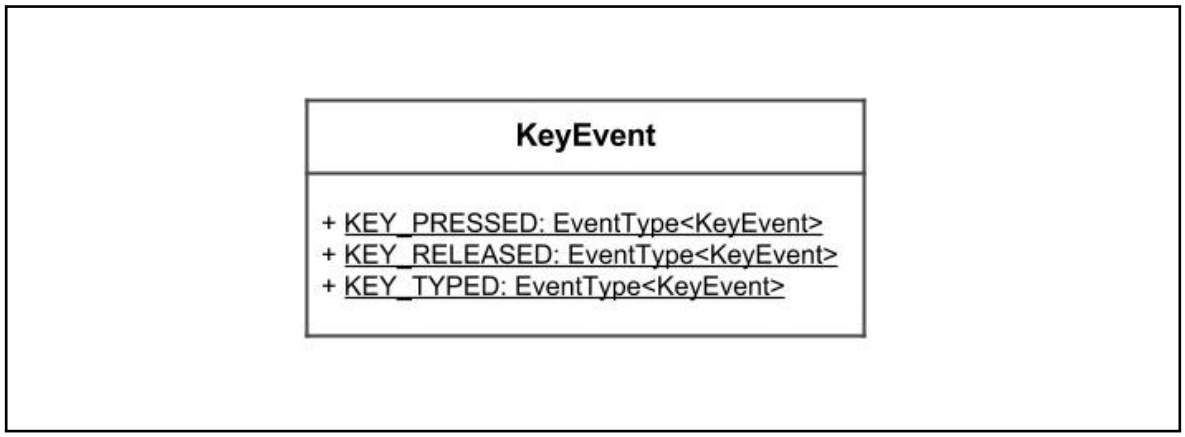

Figura 1.3: Diagrama de clase KeyEvent

Tabla 1.1: Relación Event - EvenType

Subclase	**Subtipos**
MouseEvent	MOUSE_DETECTED, MOUSE_CLICKED, MOUSE_DRAGGED, MOUSE_MOVED, MOUSE_ENTERED, MOUSE_EXITED, MOUSE_MOVED, MOUSE_PRESSED y MOUSE_RELEASED
KeyEvent	KEY_PRESSED, KEY_RELEASED y KEY_TYPED
TouchEvent	TOUCH_MOVED, TOUCH_PRESSED, TOUCH_RELEASED y TOUCH_STATIONARY

Tipos de Oyentes

Las clases oyentes representan objetos que pueden manipular los eventos que escuchan. JavaFX utiliza la interface EventHandler para el manejo de todos los tipos de eventos posibles, el código 1.1 muestra su definición.

Código 1.1: Definición de la interface EventHandler

```
1 public interface EventHandler<T extends Event> extends EventListener {
2
3    void handle(T event);
4
5 }
```

EventHandler es una interface paramétrizada, esto significa que su definición es aún más general que el de una interface normal, dando posibilidad al programador de especificar los tipos de retorno o argumentos de sus métodos mediante el parámetro que se encuentra entre los símbolos < >.

En la línea 1 del código 1.1, se indica que el parámetro T tiene que ser subclase de Event, y este parámetro modificará el tipo del argumento de la función handle() de la línea 3. Es el uso del parámetro de la interface lo que vuelve a EventHandler lo suficiente flexible y puede ser utilizada para escuchar y atender cualquier tipo de evento.

Estos son los cuatro modos de implementar un oyente con la interface EventHandler:

Implementación externa: Una clase independiente (clase que es ajena a cualquier otra) implementa la interface EventHandler, posteriormente se registra una nueva instancia del oyente en el objeto que origina los eventos. Vea el código 1.4.

Implementación interna: La interface EventHandler se implementa en alguna de las otras clases de la aplicación, una clase contenedora por ejemplo y posteriormente se registra la instancia de la clase que implentó la interface como oyente en el objeto que origina los eventos. Vea los códigos 1.5 y 1.6.

Clase anónima: Se registra directamente una instancia de la interface EventHandler implementando el método abstracto durante el instanciamiento. Vea el código 1.7.

Lambda: Se registra una función lambda (función anónima) definiendo su cuerpo durante el registro. Vea el código 1.8.

Métodos de Registro de Oyentes

Toda subclase de Node tiene métodos de conveniencia para registrar los eventos más generales definidos en JavaFX, setOnMouseClicked(), setOnKeyTyped(), setOnTouchMoved(), setOnRotate(), entre muchos otros. El código 1.2 presenta, de forma general, una firma de tipo de los métodos para registrar oyentes de eventos. Las letras en negritas son ilustrativas.

Código 1.2: Firma de tipo de los métodos de la clase Node para registrar oyentes

```
public void setOnEventName(SubclassOfEvent event)
```

Adicionalmente Node tiene un método para registrar cualquier tipo de evento, su firma de tipo se muestra en el código 1.3.

Código 1.3: Firma de tipo del método addEventHandler()

```
public final <T extends Event> void addEventHandler(EventType<T> eventType,
                                    EventHandler<? super T> eventHandler)
```

El primer parámetro del método indica el tipo de evento que va a estar escuchando, el segundo parámetro es el oyente.

Código 1.4: Implementación de oyente en clase externa

```
//Implementación del Oyente
public interface ButtonListener implements EventHandler<ActionEvent> {

   @Override
   public void handle(ActionEvent event) {
      ...
      System.out.println("Do␣some␣cool␣stuff");
      ...
   }
}

public class Layout extends BorderPane {

   public void someMethod() {
      Button btn = new Button("Click␣Me!");
      btn.setOnAction(new ButtonListener()); //Registro del oyente
      ...
   }
}
```

Código 1.5: Implementación de un oyente en clase contenedora del origen de eventos

```
public class Layout extends BorderPane implements EventHandler<ActionEvent> {

   public void someMethod() {
      Button btn = new Button("Click␣Me!");
      btn.setOnAction(this); //Registro del oyente
      ...
   }

   //Implementación del Oyente
   @Override
   public void handle(ActionEvent event) {
      System.out.println("Do␣some␣cool␣stuff");
   }
}
```

Código 1.6: Implementación de un oyente en clase interna

```
public class Layout extends BorderPane {

   public void someMethod() {
      Button btn = new Button("Click␣Me!");
      btn.setOnAction(new ButtonListener()); //Registro del oyente
      ...
   }

   class ButtonListener implements EventHandler<ActionEvent> {

      //Implementación del Oyente
      @Override
      public void handle(ActionEvent event) {
         System.out.println("Do␣some␣cool␣stuff");
      }
   }
}
```

Código 1.7: Implementación de oyente con clase anónima

```
public class Layout extends BorderPane {

   public void someMethod() {
      Button btn = new Button("Click␣Me!");
      //Registro del Oyente
      btn.setOnAction(new EventHandler<ActionEvent>() {

         //Implementación del Oyente
         @Override
         public void handle(ActionEvent event) {
            System.out.println("Do␣some␣cool␣stuff");
         }
      });
      ...
   }
}
```

Código 1.8: Implementación de oyente con lambda

```
public class Layout extends BorderPane {

   public void someMethod() {
      Button btn = new Button("Click␣Me!");
      //Registro del Oyente
      btn.setOnAction((ActionEvent e) -> {
         System.out.println("Do␣some␣cool␣stuff");
      });
      ...
   }
}
```

Ejemplos Prácticos

Ejemplo 1: Manejo de Eventos de Botón

En este ejemplo se manejan eventos de botón mediante un esquema de oyente con clase externa.

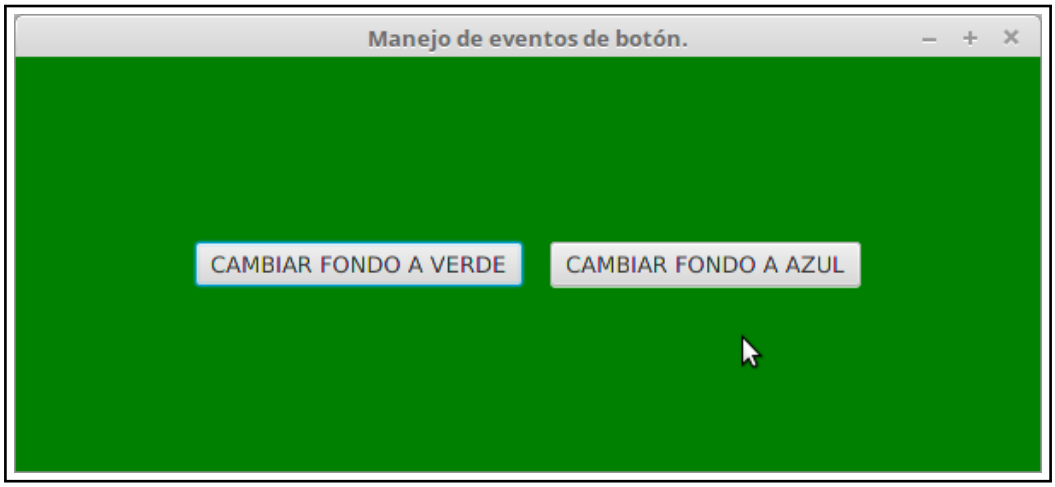

Figura 1.4: Aplicación con eventos de botón

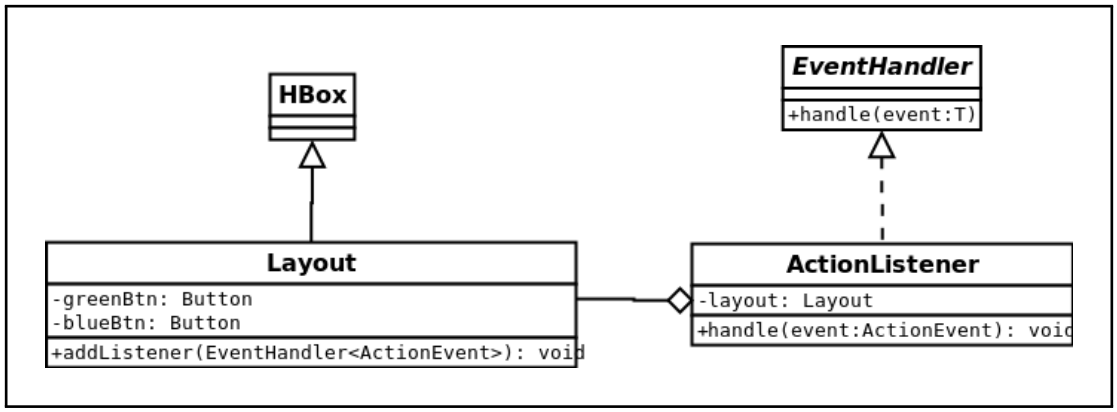

Figura 1.5: Diagrama de clases de la aplicación para el manejo de eventos de botón con oyente externo

Código 1.9: Clase principal de la aplicación

```
public class Main extends Application {

   @Override
   public void start(Stage primaryStage) {
      Layout root = new Layout();
      EventHandler<ActionEvent> listener = new ActionListener(root);
      root.addListener(listener);

      primaryStage.setScene(new Scene(root, 600, 400));
      primaryStage.setTitle("Manejo de eventos de botón.");
      primaryStage.show();
   }

   public static void main(String [] args) { launch(); }
}
```

Código 1.10: Layout de la interfaz gŕáfica de la aplicación

```
public class Layout extends HBox {

   private Button greenBtn;
   private Button blueBtn;

   public Layout() { init(); }

   private void init() {
      setAlignment(Pos.CENTER);
      setSpacing(16);

      greenBtn = new Button("CAMBIAR FONDO A VERDE");
      blueBtn = new Button("CAMBIAR FONDO A AZUL");

      greenBtn.setId("green-btn");
      blueBtn.setId("blue-btn");

      getChildren().addAll(greenBtn, blueBtn);
   }

   public void addListener(EventHandler<ActionEvent> listener) {
      greenBtn.setOnAction(listener);
      blueBtn.setOnAction(listener);
   }
}
```

Código 1.11: Clase que implementa EventHandler

```
public class ActionListener implements EventHandler<ActionEvent> {

   private final Pane layout;

   public ActionListener(Pane layout) {
      this.layout = layout;
   }


   @Override
   public void handle(ActionEvent event) {
      String sourceId = ((Node) event.getSource()).getId();

      switch(sourceId) {
         case "green-btn":
            layout.setBackground(new Background(
                  new BackgroundFill(Color.GREEN, null, null)));
            break;

         case "blue-btn":
            layout.setBackground(new Background(
                  new BackgroundFill(Color.BLUE, null, null)));
            break;
      }

   }
}
```

Ejemplo 2: Oyente Interno - Implementación de la Interface en Contenedor

En este ejemplo se obtiene una aplicación con el mismo comportamiento que la del ejemplo anterior pero diferente en implementación de oyentes. Este oyente se implementa en la clase contenedora de los origenes de eventos (los botones).

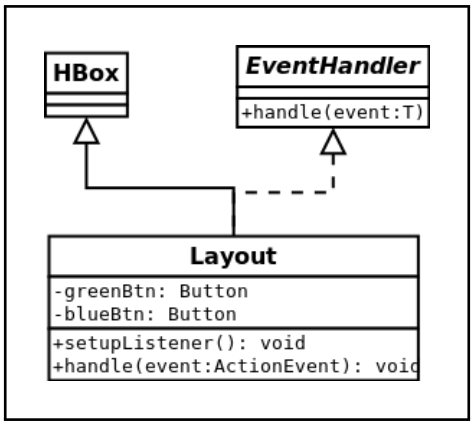

Figura 1.6: Diagrama de clases de la aplicación para el manejo de eventos de botón con oyente interno

Código 1.12: Clase principal de la aplicación

```
public class Main extends Application {

   @Override
   public void start(Stage primaryStage) {
      Layout root = new Layout();

      primaryStage.setScene(new Scene(root, 600, 400));
      primaryStage.setTitle("Manejo de eventos de botón.");
      primaryStage.show();
   }

   public static void main(String [] args) { launch(); }
}
```

Código 1.13: Oyente implementado en la clase Layout

```
public class Layout extends HBox implements EventHandler<ActionEvent> {

   private Button greenBtn;
   private Button blueBtn;

   public Layout() { init(); setupListener(); }

   private void init() {
      //Mismo código que en el ejemplo 1
   }

   private void setupListener() {
      greenBtn.setOnAction(this);
      blueBtn.setOnAction(this);
   }

   @Override
   public void handle(ActionEvent event) {
      String sourceId = ((Node) event.getSource()).getId();

      switch (sourceId) {
         case "green-btn":
            this.setBackground(new Background(
                  new BackgroundFill(Color.GREEN, null, null)));
            break;

         case "blue-btn":
            this.setBackground(new Background(
                  new BackgroundFill(Color.BLUE, null, null)));
            break;
      }
   }
}
```

Ejemplo 3: Oyente Interno - Implementación de la Interface en Clase interna

Similar al ejemplo 2, en este ejemplo se define el oyente dentro de la misma clase contenedora de los origenes de eventos pero se implementa en una clase interna. En el diagrama de la figura 1.7 se indica la clase interna mediante el conector con punta circular. El conector con punta romboidal indica que dentro de la clase Layout se hace uso de la clase ActionListener.

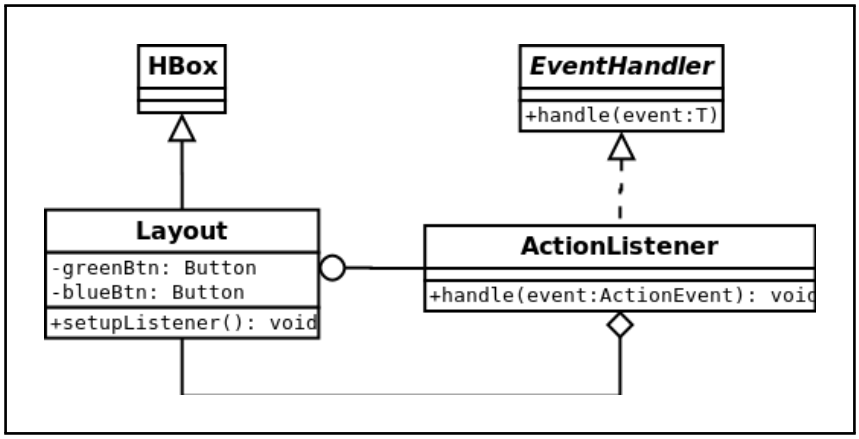

Figura 1.7: Diagrama de clases de la aplicación para el manejo de eventos de botón con oyente en clase interno

Código 1.14: Clase principal de la aplicación

```
public class Main extends Application {

   @Override
   public void start(Stage primaryStage) {
      Layout root = new Layout();

      primaryStage.setScene(new Scene(root, 600, 400));
      primaryStage.setTitle("Manejo de eventos de botón.");
      primaryStage.show();
   }

   public static void main(String [] args) { launch(); }
}
```

Código 1.15: Clase interna como oyente

```
public class Layout extends HBox {

   private Button greenBtn;
   private Button blueBtn;

   public Layout() { init(); setupListener(); }

   private void init() {
      //Mismo código que en el ejemplo 1
   }

   private void setupListener() {
      ActionListener listener = new ActionListener();
      greenBtn.setOnAction(listener);
      blueBtn.setOnAction(listener);
   }

   class ActionListener implements EventHandler<ActionEvent> {

      @Override
      public void handle(ActionEvent event) {
         String sourceId = ((Node) event.getSource()).getId();

         switch (sourceId) {
            case "green-btn":
               Layout.this.setBackground(new Background(
                     new BackgroundFill(Color.GREEN, null, null)));
               break;

            case "blue-btn":
               Layout.this.setBackground(new Background(
                     new BackgroundFill(Color.BLUE, null, null)));
               break;
         }
      }
   }
}
```

Ejemplo 4: Implementación de oyente en clase anónima

Este método también se puede considerar una implementación interna pero se hace una distinción debido a que la implementación carece de una clase receptora de la interface.

Código 1.16: Clase principal de la aplicación

```
public class Main extends Application {

   @Override
   public void start(Stage primaryStage) {
      Layout root = new Layout();

      primaryStage.setScene(new Scene(root, 600, 400));
      primaryStage.setTitle("Manejo de eventos de botón.");
      primaryStage.show();
   }

   public static void main(String [] args) { launch(); }
}
```

Código 1.17: Clase contenedora con oyente anónimo implementado

```
public class Layout extends HBox {

   private Button greenBtn;
   private Button blueBtn;

   public Layout() { init(); setupListener(); }

   private void init() {
      //Mismo código que en el ejemplo 1
   }

   private void setupListener() {
      EventHandler listener = new EventHandler<ActionEvent>() {
         @Override
         public void handle(ActionEvent event) {
            String sourceId = ((Node) event.getSource()).getId();

            switch (sourceId) {
               case "green-btn":
                  Layout.this.setBackground(new Background(
                        new BackgroundFill(Color.GREEN, null, null)));
                  break;

               case "blue-btn":
                  Layout.this.setBackground(new Background(
                        new BackgroundFill(Color.BLUE, null, null)));
                  break;
            }
         }
      };
      greenBtn.setOnAction(listener);
      blueBtn.setOnAction(listener);
   }
}
```

Ejemplo 5: Lambda como Oyente

Las interfaces que poseen un sólo método son conocidas como interfaces funcionales, debido a que su definición como clases anónimas pueden ser sustituidas por funciones lambda.

Código 1.18: Clase principal de la aplicación

```
public class Main extends Application {

   @Override
   public void start(Stage primaryStage) {
      Layout root = new Layout();

      primaryStage.setScene(new Scene(root, 600, 400));
      primaryStage.setTitle("Manejo de eventos de botón.");
      primaryStage.show();
   }

   public static void main(String [] args) { launch(); }
}
```

Código 1.19: Clase contenedora con oyente anónimo implementado

```
public class Layout extends HBox {

   private Button greenBtn;
   private Button blueBtn;

   public Layout() { init(); setupListener(); }

   private void init() {
      //Mismo código que en el ejemplo 1
   }

   private void setupListener() {
      EventHandler<ActionEvent> listener = (ActionEvent event) -> {
         String sourceId = ((Node) event.getSource()).getId();

         switch (sourceId) {
            case "green-btn":
               Layout.this.setBackground(new Background(
                     new BackgroundFill(Color.GREEN, null, null)));
               break;

            case "blue-btn":
               Layout.this.setBackground(new Background(
                     new BackgroundFill(Color.BLUE, null, null)));
               break;
         }
      };
      greenBtn.setOnAction(listener);
      blueBtn.setOnAction(listener);
   }
}
```

Ejemplo Extra: FXML y Controladores

El esquema de construcción de interfaces mediante archivos FXML permite otra forma de atender los eventos de los componentes de la GUI.

FXML es un lenguaje declarativo, no de programación, y es utilizado para describir la interface de usuario: los componentes que la conforman, su estructura y orden. Para definir el comportamiento de la aplicación cuando se produce algún eventos desde la GUI se utiliza código Java.

En FXML se pueden definir uno o más controladores (oyentes) en las distintas secciones de la GUI. La vinculación de un controlador con la sección de GUI que controlará se hace mediante el atributo fx:controller.

A continuación se presentan códigos de ejemplo para realizar la misma aplicación de los ejemplos anteriores utilizando FXML.

Código 1.20: Clase principal donde se construye la GUI desde el archivo FXML

```
package fxml;

//Se omiten imports
import javafx.fxml.FXMLLoader;

public class Main extends Application {

   @Override
   public void start(Stage stage) throws Exception {
      Parent root = FXMLLoader.load(getClass().getResource("layout.fxml"));

      Scene scene = new Scene(root);
      stage.setScene(scene);
      stage.show();
   }

   public static void main(String[] args) { launch(args); }
}
```

Código 1.21: Arhivo FXML que define la GUI de la aplicación

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Button?>
<?import javafx.scene.layout.HBox?>

<HBox xmlns="http://javafx.com/javafx/8.0.111" xmlns:fx="http://javafx.com/fxml/1"
     maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity" minWidth="-Infinity"
     prefHeight="331.0" prefWidth="557.0" spacing="16.0" alignment="CENTER"
     fx:id=”container” fx:controller=”fxml.Controller”>
   <children>
     <Button mnemonicParsing="false" onAction=”#changeToGreen” text="CAMBIAR FONDO A VERDE" />
     <Button mnemonicParsing="false" onAction=”#changeToBlue” text="CAMBIAR FONDO A AZUL" />
   </children>
</HBox>
```

Código 1.22: Clase controladora de los eventos de la GUI

```
1  package fxml;
2
3  //Se omiten imports
4  import javafx.fxml.FXML;
5
6  public class Controller {
7
8     @FXML
9     private HBox container;
10
11    @FXML
12    private void changeToGreen(ActionEvent event) {
13       container.setBackground(new Background(
14                         new BackgroundFill(Color.GREEN, null, null)));
15    }
16
17    @FXML
18    private void changeToBlue(ActionEvent event) {
19       container.setBackground(new Background(
20                         new BackgroundFill(Color.BLUE, null, null)));
21    }
22
23 }
```

En el código 1.22 se utiliza la anotación @FXML para dos propósitos. En la línea 8 vincula el atributo que le sucede con un componente de la interface definido en el archivo FXML, el pegamento que los une son el nombre del atributo de la clase y del atributo fx:id del elemento HBox en el archivo FXML. La segunda función de la anotación es "exponer" los métodos de las líneas 12 y 18 del controlado en el archivo FXML permitiendo su asociación como oyentes de los eventos de los botones.