5. Event-based Architectures

1. Motivation and Fundamental Concepts
2. Revisiting Object-Oriented Analysis, Design, and Implementation
3. Design Patterns
4. Pipes & Filter Architectures
5. **Event-based Architectures**
7. Framework Architectures
8. Component Architectures
Learning Objectives of Chapter 5

Students should be able

- to describe **event-based architectures** and their variations
- to explain the benefits and liabilities of event-based architectures
- to understand and compare the event-based architectures of popular GUI libraries
- to explain the observer, chain of responsibility and command pattern

**Recommended Reading**

- [ShGa96] Section 2.4
- [GHJV95] Observer (p. 293), Chain of Responsibility (p. 223), Command (p. 233)
- [Vli98] Chapter 2 + 3
- [BMRSS96] Section 2.4

Where Events are Needed

**Initial model of computation:**

Algorithms transform input (sequences) to output (sequences)

\[ f(e) = a \]

**Automaton**

\[ f(e_0, e_{-1}, \ldots, e_0) = a_0 \]

**Today’s model of computation:**

Computers are interaction machines (reactive systems) that continuously interact with their environment and that evolve with their environment
Events at Different Levels of Abstraction

- **Database Objects**
  - Mapped to: Article moved, Article dropped
  - Represented by: Character

- **Logical GUI Objects**
  - Mapped to: Menu item selected, Window
  - Represented by: Mouse moved

- **Application Objects**
  - Mapped to: Article window, News article list
  - Represented by: Message created

- **Physical I/O Objects**
  - Mapped to: Key pressed on keyboard, Mouse moved
  - Represented by: Character, Pixel

Popular Example: Events in Win32

Win32 is the Windows-98/ME/... Operating System API (application programming interface)

- Applications call the operating system
  - create, delete, modify resources (windows, files, connections, devices, ...)

- The operating system creates events to notify the applications
  - user input occurred (mouse, keyboard)
  - messages from the network arrived, ...
  - time-out happened, ...
  - application opened, system will be shut down, ...

- Applications may communicate with each other via events dispatched by the operating system:
  - open / close / print / copy / paste
Example: Win32 GUI-Events (1)

- GUI-Events are logically associated with GUI objects
- GUI elements are uniquely identifiable in a tree that represents the visual "parent / child" relationship
  - hierarchical name (button “New” in the window “NewsWindow” as child of the window “Cocoon Client” on the desktop)
  - unique ID assigned by the operating system (GUI object “43253323455”)
- The operating system is responsible for the translation from logical GUI elements to display representations (moving, resizing, overlapping of windows).

Example: Win32 GUI-Events (2)

- The Win32-API is not implemented in an object-oriented way.
- Conceptually, GUI elements are maintained in an inheritance hierarchy:
  - messages understood by “Window” are also understood by all its descendants
  - events created by “Window” can also be created by all its descendants.
- However, there are many ad-hoc exceptions to these rules.
Example: Win32 GUI-Events (3)

**Messages and events** are exchanged through two low-level operating system calls ("SendMessage" and "GetMessage"):

- Decoupling of sender and receiver
- Queuing of events by the operating system
- Flexible forwarding of events between application components
- Tracing of events
- Remote control of existing applications through GUI-messages, e.g.:

```c
char* GetText (HWND hParentWindow, int iControlId, char* sTextbuffer, int iMaxTextLen) {
    HWND hTextControl ;
    if (!(hTextControl = GetChildWindowHandle (hParentWindow, iControlId)))
        return "" ;
    SendMessage (hTextControl,WM_GETTEXT,iMaxTextLen,(LPARAM)sTextbuffer) ;
    return sTextbuffer ;
}
```

C, not Java (procedural)

Example: Java Applets (1)

An Applet is a Java program executed within a special environment, typically a web browser.

An Applet is visually embedded into an HTML page.

A web browser downloads the Applet byte code from the server.

Example:

```java
import java.applet.Applet ;
import java.awt.* ;

public class HelloWeb
    extends Applet
{
    public void paint (Graphics g) {
        g.setColor (Color.blue) ;
        g.setFont (new Font("Arial", Font.BOLD, 32)) ;
        g.drawString ("Hello, I'm a Java Applet!", 80, 25) ;
    } // paint

} // class HelloWeb
```
Example: Java Applets

Including the applet into an HTML page:

```html
My program says:

<P>
<APPLET CODE="HelloWeb.class" WIDTH=150 HEIGHT=25>
</APPLET>

Loading the HTML page with a Java-enabled web browser (e.g. Netscape):
```

Example: Java Applets – Sandbox Concept

Applets may not read or modify data on the computer on which they are being executed.
They may not establish network connections to servers other than the one they have been downloaded from.
Applets may not invoke other programs or use library functions on the computer on which they are executed.
Applets can access a constrained set of data about the properties of the client computer only.
Example: Java Applets: Applet Life Cycle (1)

Applets are programmed and used in a framework which offers the basic behavior of all Applets.

An Applet is defined by subclassing class Applet.

Some methods from Applet are redefined.

The framework determines when and with which parameters these methods are invoked.

The is no main Method (like in applications) which starts the Applet or creates an initial instance.

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
<th>Time of invocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>init ()</td>
<td>initialization of the Applet</td>
<td>after creating the Applet object</td>
</tr>
<tr>
<td>start ()</td>
<td>starting the Applet</td>
<td>after init() and when the Applet becomes visible</td>
</tr>
<tr>
<td>paint (Graphics g)</td>
<td>(re-)drawing the Applets</td>
<td>when redrawing becomes necessary</td>
</tr>
<tr>
<td>stop ()</td>
<td>stopping the Applet’s operations</td>
<td>when the applet leaves the visible area</td>
</tr>
<tr>
<td>destroy ()</td>
<td>destroying the Applet</td>
<td>when the Applet is unloaded</td>
</tr>
</tbody>
</table>

Example: Java Applets: Applet Life Cycle (2)

![Applet Life Cycle Diagram]

load Applet / MyApplet ()  
created

load Applet / init ()  
initialized

page with Applet visited / start ()  
running

Applet outside visible area / stop ()  
visible

cover Applet  
uncover Applet / update ()

page with Applet left or reloaded / destroy ()  
destroyed

remove Applet

covered

invisble

show page again / start ()  
page with Applet visited / start ()

Software Architectures Chapter 5: Event-based Architectures
Event-Based Architectures

- Communication between components not by explicit invocation (procedure call, message sending) but by implicit invocation (also called reactive integration or selective broadcast).
- One component (event source) announces (broadcasts) one or more events.
- Other components (event handlers) register an interest in an event by associating a procedure with it.
- When an event is announced, the system (event manager) invokes all of the procedures associated with this event.
- Component interfaces thus consist of:
  - data fields
  - actions (procedures, methods, ...)
  - events
- There are numerous design alternatives for event management policies (explicit and implicit chaining, broadcasting, ...).

Benefits of Event-Based Architectures

- Essential concept for reactive systems
  - user input
  - network communication
- Returning results from asynchronous execution
- Significantly increased potential for component reuse
- Improved system evolution
  - replace a component by another component
  - wrap a component by a similar component
  - add new events that are simply ignored by “old” clients
Limitations of Event-Based Architectures

- Distributed control through multiple event handlers makes synchronization, verification and debugging difficult.
- There is no guarantee that an event-handling component will be present at runtime (compare with type checking of actions).
- The event source cannot make assumptions about the temporal order of event handling code (e.g. concurrent execution). This may lead to the need for explicit synchronization.
- Complex event arguments may become difficult to administer
  - copy events between process spaces
  - release resources (open files, window handles)
  - detect “outdated” or “invalidated” events (e.g. from a window that has already been destroyed)
- Classical pre- and postcondition reasoning within a component becomes impossible in event-based systems.

Class Responsibility Cards

<table>
<thead>
<tr>
<th>Class: Event</th>
<th>Class: EventManager</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Responsibility</strong></td>
<td><strong>Responsibility</strong></td>
</tr>
<tr>
<td>• Encapsulates details of an event (reason, time, target, location, context)</td>
<td>• Decouples EventSource from EventHandler</td>
</tr>
<tr>
<td>• Classifies event</td>
<td>• Synchronizes Events</td>
</tr>
<tr>
<td><strong>Collaborators:</strong></td>
<td><strong>Collaborators:</strong></td>
</tr>
<tr>
<td>• EventManager</td>
<td>• EventSource</td>
</tr>
<tr>
<td>• EventHandler</td>
<td>• EventHandler</td>
</tr>
<tr>
<td>• Event</td>
<td>• Event</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class: EventSource</th>
<th>Class: EventHandler</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Responsibility</strong></td>
<td><strong>Responsibility</strong></td>
</tr>
<tr>
<td>• Collects Event details</td>
<td>• Expresses interest in (a class of) Events</td>
</tr>
<tr>
<td>• Delivers Events</td>
<td>• Accepts Events or rejects Events</td>
</tr>
<tr>
<td><strong>Collaborators:</strong></td>
<td><strong>Collaborators:</strong></td>
</tr>
<tr>
<td>• EventManager</td>
<td>• EventManager</td>
</tr>
<tr>
<td>• Event</td>
<td>• Event</td>
</tr>
</tbody>
</table>

*sometimes called EventTarget*
Example: GUI Event Processing

- **Event**: "Button" "double-clicked" "17:31:22"
- **EventSource**: Button managed by the GUI subsystem of the operating system
- **EventHandler**: Notification method in the application code
- **EventManager**: Operating system or GUI library code

```
>User aButton:Button
<EventManager aHandler:ClickHandler
<Application

registration
<new()
<mouseClick
anEvent:Event
<service>
<getInfo ()

concurrency!
```

Example: Event Handling in AWT of Java 1.0 (1)

**Improvements over Win32**

- GUI objects are represented as objects (with true subclassing)
- Message dispatch to GUI objects is fully handled by dynamic message binding
- Events are represented as first-class objects (however, there is no subclassing)

```
Component
getX (), getY (), getW (), getH ()
action (evt : Event, o : Object)

Event
target : Object
...
```

AWT = Abstract Window Toolkit
Example: Event Handling in AWT of Java 1.0 (2)

Limitations

- GUI objects are event handlers ⇒ tight coupling of GUI and application logic
- There is an `action()` method for a component
  - cascaded if-statements to identify the actual source of the event within the window by
    - string comparison of the name of the sender `arg`
      (slow because of string comparison, unsafe, not shown here)
    - or by an object identity test on `evt.target`
      (faster and safer because due to knowledge on the type of object)
  - inside each if-branch identify the details of the event by inspecting the attributes of `evt`
- There are other methods for other event types (key up, key down, ...).
- Each method returns a boolean value to indicate whether the event has been handled. Otherwise, the superclass implementation should be called.
- Each of these methods has a default “no op” implementation.

Example: Event Handling in AWT of Java 1.0 (3)

```java
public class Button2 extends Applet {
  Button b1 = new Button("Button 1"),
  b2 = new Button("Button 2") ;

  public void init () {
    add (b1) ;
    add (b2) ;
  } // init

  public boolean action (Event evt, Object arg) {
    if (evt.target.equals (b1))
      getAppletContext ().showStatus ("Button 1 pressed") ;
    else if (evt.target.equals (b2))
      getAppletContext ().showStatus ("Button 2 pressed") ;
    else
      return super .action (evt, arg) ;

    return true ; // we have handled the event
  } // action

} // class Button2
```

This is the status line
Example: Event Handling in JFC of Java 1.1 (1)

The "new event model":

- **Event**: One class per type of event. Inheritance captures specialization of events.
- **Event source**: An object that implements an event source interface.
- **Event listener**:
  - An event listener is an object of a class that implements a particular event listener interface.
  - An event listener is created dynamically and attached to an event source with an `addXListener()` method call.
  - Listeners can be added & removed freely.
  - There can be multiple listeners attached to one event source.

Clean separation between GUI and application logic.

See also **Typed Message / Multicast** [Vli98], Chapter 4.

Example: Event Handling in JFC of Java 1.1 (2)

**EventListener**

- `target : Object`

**ActionEvent**

- `actionPerformed (a : ActionEvent)`

**KeyEvent**

- `keyPressed (k : KeyEvent)`
- `keyReleased (k : KeyEvent)`
- `keyTyped (k : KeyEvent)`

**KeySource**

- `gui components constitute EventSources`

**MyActionListener**

**YourActionListener**

**MyKeyListener**

**YourKeyListener**

**KeyEvent**

**ActionEvent**

**EventListener**

**GUI source**

**Event source**

JFC = Java Foundation Classes
Example: Event Handling in JFC of Java 1.1

```java
public class Button2New extends Applet {
    Button b1 = new Button("Button 1");
    Button b2 = new Button("Button 2");
    public void init () {
        b1.addActionListener (new B1 (this));
        b2.addActionListener (new B2 (this));
        add (b1);
        add (b2);
    } // init
} // Button2New

class B1 implements ActionListener {
    private Applet applet;
    B1 (Applet applet) {
        this.applet = applet;
    }
    public void actionPerformed (ActionEvent a) {
        applet.getAppletContext ().showStatus ("Button 1 pressed");
    }
} // class B1

class B2 implements ActionListener {
    private Applet applet;
    B2 (Applet applet) {
        this.applet = applet;
    }
    public void actionPerformed (ActionEvent a) {
        applet.getAppletContext ().showStatus ("Button 2 pressed");
    }
} // class B2
```

Event Handling for MVC Pattern in Java Swing

The “new” event model is the basis for the MVC pattern used in most of the components from the “Swing” library.

Swing Components have a model assigned:

```java
ListModel model = new MyListModel (...);
JList list = new JList (model);
```

The communication between the component (as view and controller) and the model is done through Events and Listeners.

E.g., ListDataListeners can be registered for a ListModel, which define operations:

- void contentsChanged (ListDataEvent e)
- void intervalAdded (ListDataEvent e)
- void intervalRemoved (ListDataEvent e)

and a ListDataEvent is of one of the types: CONTENTS_CHANGED, INTERVAL_ADDED, or INTERVAL_REMOVED.
Variations: SELECT in POSIX

- System programs often have to be able to wait for (one of) simultaneous events
  - time-out, network packet on socket A arrived, user-input received, lock released, ...
- The application submits multiple requests, each identified by a request-id.
- The application calls the SELECT procedure from the POSIX API (portable operating system interface for Unix systems) indicating the requests it is interested in.
- The operating system blocks the execution of the process executing the SELECT procedure until a reply for one of the matching requests has been received.
- The operating system resumes execution of the calling process with a return value.
- The application program examines the return value to decide which event occurred and accesses further data structures to obtain event-specific details.

Note:
- Event parameters and handler buried somewhere in the code.
- The SELECT procedure tends to be the most complicated piece of code in any application.

Variations: ECA-Rules in Databases (1)

Some modern database systems (active databases) permit to attach ECA-rules to databases or database classes:

<table>
<thead>
<tr>
<th>on</th>
<th>person.father.delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>when</td>
<td>person.mother is null</td>
</tr>
<tr>
<td>do</td>
<td>delete person</td>
</tr>
</tbody>
</table>

- **Event:** Triggered at the end of a transaction / during a database update
  - Object of a certain class in the database created, destroyed, updated
  - Attribute of an object of a certain class updated, created, destroyed
  - Object referenced by another object deleted or updated
- **Condition:** Boolean expression on the object affected by the event
- **Action:** Operation executed (as a separate transaction!)
  - propagate change
  - abort triggering transaction
  - start arbitrary procedural code (fixup action)
Variations: ECA-Rules in Databases (2)

Note: Users and Transactions interact implicitly.

Variations: ECA-Rules in Databases (3)

Advantages:
- Catch relevant events from any application (including interactive user input)
- Protect integrity of mission-critical data

Disadvantages:
- Limited expressiveness of the action part (e.g., has to be expressed in PL SQL)
- Limited knowledge of the session context
- Limited control over the overall order of execution of cascading triggers (e.g. phase 1, phase 2 and phase 3 trigger processing, priority of triggers, ...)
- Hard to understand and to maintain
  - effect
  - termination
  - ...

Advantages:
- Catch relevant events from any application (including interactive user input)
- Protect integrity of mission-critical data

Disadvantages:
- Limited expressiveness of the action part (e.g., has to be expressed in PL SQL)
- Limited knowledge of the session context
- Limited control over the overall order of execution of cascading triggers (e.g. phase 1, phase 2 and phase 3 trigger processing, priority of triggers, ...)
- Hard to understand and to maintain
  - effect
  - termination
  - ...

These Are Similar to But Different From Events

- **Exceptions** are mechanisms for structured error handling **within** a single software component:
  - **Example** in Java: catch division by zero error
    ```java
    try {
        int x = 1 / 0;
    } catch (Exception e) {
        System.err.println (e.getClass () + ": "+ e.getMessage () ;
    }
    ```
  - Exception object resembles event object. Exception handler resembles event handler.
  - However: event handler determined by program flow only, exception handling changes control flow in original program

- **Unix Signals** are a mechanism to asynchronously notify another running process. The process can be suspended or aborted, a flag can be set, ...

Related Design Patterns (1)

**Command** [GHJV95, p 233]:
- Turn a message with arguments into a command object with data fields
  ```java
  listener.buttonPressed (300, 400) ;
  ```
- There is a common superclass **Command** that defines
  - methods on commands (execute, undo, redo, copy)
  - functions on commands (changesState, isUndoable, ...)
- There are tools that work on commands
  - command dispatcher, command tracer, recovery tool, macro recorder

See also **Observer** [GHJV95, p. 293].
```java
Command

ButtonPressed
Command

listener.buttonPressed (300, 400) ;
c = new ButtonPressedCommand (300, 400) ;
c.execute () ;
```
Related Design Patterns (2)

**Chain of Responsibility** [GHJV95, p. 223]
- There is a hierarchy of EventTargets (often represented visually by nesting)
- Parents are also responsible for the events of their children
- The bottom elements of the hierarchy perform very specific event handling.
- If there is no specific EventHandler, the EventManager automatically forwards the event to the next higherEventHandler

- **Example:** Context-sensitive help in MS Windows
  - Help on a character in a text, help on a text frame, help on an editor window, help on the editor commands, help on MS Windows

- **Example:** Command processing in Hypercard
  - Command processed by a card element
  - Command processed by a card
  - Command processed by a stack of cards
  - Command processed by AppleScript

This is a text in a text editor window that may illustrate ...

F1 pressed