TELETYPEx
MODEL 28 PAGE PRINTER

A new concept in Data Communications
It’s the technology, stupid!
Six easy pieces

Bertrand Meyer

LASER, Biodola, September 2006
Contracts and tests
The cluster model
Dealing with events
Dealing with Void
Open-sourcing EiffelStudio
Towards an O-O process
Program drives input:

```
from
  read_line
until end_of_file loop
  i := i + 1
  buffer [i] := last_line
read_line
end
```
Handling input with modern GUIs

User drives program:

“When a user presses this button, execute that action from my program”
Traditional input handling

```plaintext
from
  read_line
until end_of_file loop
  i := i + 1
  Result.put(last_line, i)
  read_line
end
```
Event-driven execution

Publishers

Subscribers

Routine

Routine

Routine

Routine

Routine
Handling input with modern GUIs

User drives program:

“When a user presses this button, execute that action from my program”
Event-driven programming

Specify that when a user clicks this button the system must execute

\[ \text{your\_procedure}(x, y) \]

where \( x \) and \( y \) are the mouse coordinates and \( \text{your\_procedure} \) is a specific procedure of your system.
Event-driven programming

Publishers

Subscribers

Routine

Routine

Routine

Routine

Routine
**Events Overview**

Events have the following properties:

1. The publisher determines when an event is raised; the subscribers determine what action is taken in response to the event.

2. An event can have multiple subscribers. A subscriber can handle multiple events from multiple publishers.

3. Events that have no subscribers are never called.

4. Events are commonly used to signal user actions such as button clicks or menu selections in graphical user interfaces.

5. When an event has multiple subscribers, the event handlers are invoked synchronously when an event is raised. To invoke events asynchronously, see [another section].

6. Events can be used to synchronize threads.

7. In the .NET Framework class library, events are based on the EventHandler delegate and the EventArgs base class.
Model-View Controller

(Trygve Reenskaug, 1979)
Architecture: avoiding glue code

Model View Controller (MVC) Design Pattern
A design pattern is an architectural scheme — a certain organization of classes and features — that provides applications with a standardized solution to a common problem.

Since 1994, various books have catalogued important patterns. Best known are by Gamma *et al.* and by Pree.
First solution: Observer Pattern

- Deferred (abstract)
- Effective (implemented)

Inherits from

Client (uses)
Observer pattern

Publisher keeps a list of observers:

\[ \text{subscribers : LINKED\_LIST [SUBSCRIBER]} \]

To register itself, an observer may execute

\[ \text{subscribe (some\_publisher)} \]

where \text{subscribe} is declared in \text{SUBSCRIBER}:

\[ \text{subscribe (p: PUBLISHER) is} \]

\[ \text{-- Make current object observe p.} \]

\[ \text{require} \]

\[ \text{publisher\_exists : p /= Void} \]

\[ \text{do} \]

\[ p.\text{attach} (\text{Current}) \]

\[ \text{end} \]
Attaching an observer

In class PUBLISHER:

\[
\text{attach} \ (s: \text{SUBSCRIBER}) \text{ is}
\]

\[
\begin{align*}
\text{-- Register } s \text{ as subscriber to current publisher.}
\text{require} \\
\text{subscriber\_exists: } p \neq \text{Void} \\
\text{do} \\
\text{subscribers.extend}(s) \\
\text{end}
\end{align*}
\]

The invariant of PUBLISHER includes the clause

\[
\text{subscribers} \neq \text{Void}
\]

(List subscribers is created by creation procedures of PUBLISHER)
publish is

-- Tell all observers to
-- react to current event.

do
  from subscribers.start
  until subscribers.after
  loop
    subscribers.item. handle
    subscribers.forth
  end
end

Every descendant of SUBSCRIBER defines its own version of handle
Observer pattern: some limitations

- Each publisher object knows about its observers

- Only one *handle* procedure in SUBSCRIBER:
  - At most one operation
  - Subscribe to at most one publisher!

- Very hard to take *event arguments* into account

- Not reusable — must be coded anew for each application

(This is the difference between patterns & components!)
Another approach: event-action table

**Event**\_**type** - **Action** Table

More precisely: **Event**\_**type** - **Context** - **Action** Table

<table>
<thead>
<tr>
<th>Event type</th>
<th>Context</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left_click</td>
<td>Yes_button</td>
<td><strong>Save</strong>_<strong>file</strong></td>
</tr>
<tr>
<td>Left_click</td>
<td>Cancel_button</td>
<td><strong>Reset</strong></td>
</tr>
<tr>
<td>Left_click</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Right_click</td>
<td></td>
<td><strong>Display</strong>_<strong>Menu</strong></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Action-event table

**Set of triples**

[Event, Context, Action]

- **Event**: any occurrence we track  
  **Example**: a mouse click

- **Context**: object for which the event is interesting  
  **Example**: a particular button

- **Action**: what we want to do when the event occurs in the context  
  **Example**: save the file

*Action-event table may be implemented as e.g. a hash table.*
Mechanisms in other languages

- C and C++: “function pointers”
- C#: delegates (more limited form of agents)
Problem scenario

- One of your classes has a routine \textit{your\_procedure}

- Your application has a GUI object known as \textit{Yes\_button}

- Whenever the user clicks the mouse, the underlying GUI library returns the mouse coordinates

- You want to ensure that a mouse click at coordinates \([h, v]\) in calls \textit{your\_procedure}(h, v)
P1. Introduce **new class** `ClickArgs` inheriting from `EventArgs`, repeating arguments types of `yourProcedure`:

```csharp
public class ClickArgs {... int x, y; ...}
```

P2. Introduce **new delegate type** `ClickDelegate` based on `ClickArgs`:

```csharp
public void delegate ClickDelegate (Object sender, e);
```

P3. Introduce **new event type** `Click` based on `ClickDelegate`:

```csharp
public event ClickDelegate Click;
```
P4. Introduce **new procedure** `OnClick` to wrap handling:

```csharp
protected void OnClick (Clickargs c)
    {if (Click != null) {Click (this, c);}}
```

P5. For every event occurrence, create **new object** (instance of `ClickArgs`), passing arguments to constructor:

```csharp
ClickArgs yourClickargs = new Clickargs (h, v);
```

P6. For every event occurrence, publish event:

```csharp
OnClick (yourClickargs);
```
D1. Declare a delegate \texttt{myDelegate} of type \texttt{ClickDelegate}.
(Usually combined with following step.)

D2. Instantiate it with \texttt{yourProcedure} as argument:

\begin{verbatim}
ClickDelegate myDelegate = new ClickDelegate (yourProcedure);
\end{verbatim}

D3. Add it to the delegate list for the event:

\begin{verbatim}
YES_button.Click += myDelegate;
\end{verbatim}
Using the Eiffel approach (Event Library)

- **Event**: each event type will be an object
  
  **Example**: mouse clicks
  
  May have **arguments**

- **Context**: an object, usually representing a user interface element
  
  **Example**: a particular button

- **Action**: an agent representing a routine
  
  **Example**: your_procedure

---

Yes!
Action-event table

Set of triples

[Event, Context, Action]
Using the Event Library

The basic class is \textit{EVENT\_TYPE}

On the publisher side, e.g. GUI library:

- (Once) declare event type:
  
  \textit{click}: \textit{EVENT\_TYPE} [\textit{TUPLE}[\textit{INTEGER}, \textit{INTEGER}]]

- (Once) create event type object:
  
  \texttt{create click}

- To publish one occurrence of the event:
  
  \texttt{click.publish ([x-coordinate, y-coordinate])}

On the subscriber side, e.g. an application:

\texttt{click.subscribe (agent my_procedure)}
Subscriber variants

\[ \text{click.subscribe (agent your\_procedure)} \]

\[ \text{my\_button. click.subscribe (agent your\_procedure)} \]

\[ \text{click.subscribe (agent my\_procedure (a, ?, ?, b))} \]

\[ \text{click.subscribe (agent other\_object.\_other\_procedure)} \]
MVC

MODEL

represents

think in terms of

VIEW_i

updates

sees

CONTROLLER

GUI tools

interacts with

User

(MVC structure)

(Other views)
MVC: Direct subscription
MVC: using a controller

MODEL

EVENT TYPES

VIEW_i

CONTROLLER

(Action)

(Event type)

publish

subscribe

(Context)

Optional client link
The context of this work

Componentization: turning patterns into components
Thesis by Karine Arnout at ETH Zurich

Google: karine arnout patterns components eth

For the publish-subscribe example, see


Google: bertrand meyer abstraction event dahl

(Also articles by Volkan Arslan and others)
Componentization: turning patterns into components

The conjecture: can we get rid of patterns?
The testbed: all the patterns in Gamma et al

Karine’s classification, and componentization results: (for Eiffel)

- 1. Fully componentizable 65%
- 2. Partially componentizable 26%
- 3. Recalcitrant 9%

Downloadable from http://se.ethz.ch:

- Pattern library (for categories 1 and 2)
- Pattern Wizard (for 3)
Language constructs that make this possible

- **Genericity**
- **Inheritance, polymorphism, dynamic binding**
- **Multiple inheritance**
- **Constrained genericity**
- **Covariance**
- **Design by Contract, esp. class invariants**
- "Once" routines

- **Tuples**
- **Agents**
Using the Eiffel Event Library

The basic class is **EVENT_TYPE**

On the publisher side, e.g. GUI library:

- (Once) declare event type:
  
  ```
  click: EVENT_TYPE [ TUPLE [ INTEGER, INTEGER ] ]
  ```

- (Once) create event type object:
  
  ```
  create click
  ```

- To publish one occurrence of the event:
  
  ```
  click.publish ([x_coordinate, y_coordinate])
  ```

On the subscriber side, e.g. an application:

```
click.subscribe (agent my_procedure)
```
Subscriber variants

\texttt{click.subscribe (agent your\_procedure)}

\texttt{my\_button. click.subscribe (agent your\_procedure)}

\texttt{click.subscribe (agent my\_procedure (a, ?, ?, b))}

\texttt{click.subscribe (agent other\_object.\_other\_procedure)}