

Last update: 18 April 2007

Software Architecture

Bertrand Meyer

ETH Zurich, March-July 2007

Lecture 7: Patterns, Observer, MVC

References

Erich Gamma, Ralph Johnson, Richard Helms, John Vlissides: *Design Patterns*, Addison-Wesley, 1994

Jean-Marc Jezequel, Michel Train, Christine Mingins: *Design Patterns and Contracts*, Addison-Wesley, 1999

Karine Arnout: *From Patterns to Components*, 2004 ETH thesis, <http://se.inf.ethz.ch/people/arnout/patterns/>

Patterns in software development

Design pattern:

- > A document that describes a general solution to a design problem that recurs in many applications.

Developers adapt the pattern to their specific application.

Benefits of design patterns

- > Capture the knowledge of experienced developers
- > Publicly available repository
- > Common pattern language
- > Newcomers can learn & apply patterns
- > Yield better software structure
- > Facilitate discussions: programmers, managers

Some design patterns

<p>Creational</p> <ul style="list-style-type: none"> > Abstract Factory > Builder > Factory Method > Prototype > Singleton <p>Structural</p> <ul style="list-style-type: none"> > Adapter > Bridge > Composite > Decorator > Façade > Flyweight > Proxy 	<p>Behavioral</p> <ul style="list-style-type: none"> > Chain of Responsibility > Command (undo/redo) > Interpreter > Iterator > Mediator > Memento > Observer > State > Strategy > Template Method > Visitor
---	---

A pattern is not a reusable solution

Solution to a particular recurring design issue in a particular context:

- > *"Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to this problem in such a way that you can use this solution a million times over, without ever doing it the same way twice."*

Gamma et al.

NOT REUSABLE

A step backwards?

Patterns are not reusable solutions:

- You must implement every pattern every time
- Pedagogical tools, not components

We have done work at ETH to correct this situation:

"A successful pattern cannot just be a book description: it must be a software component"

Result: Pattern Library and Pattern Wizard
(see following lectures)

Our first set of patterns & componentization

Observer pattern

Model-View Controller

Improving on Observer: a more general & flexible approach

Implementing the solution in C#/ .NET

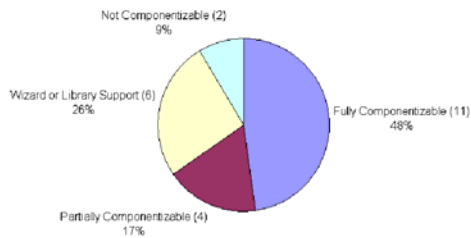
Implementing the solution in Eiffel

Pattern componentization

Karine Arnout
ETH PhD, 2004

Classification of design patterns:

- Fully componentizable
- Partially componentizable
- Wizard- or library-supported
- Non-componentizable



Handling input through traditional techniques

Program drives user:

```
from
  read_line
  count := 0
until exhausted loop
  count := count + 1
  -- Store last_line at
  -- position count in Result
  Result[count] := last_line
  read_line
end
```

End of input



Pattern componentization: references

Bertrand Meyer: *The power of abstraction, reuse and simplicity: an object-oriented library for event-driven design*, in *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*, Lecture Notes in Computer Science 2635, Springer-Verlag, 2004, pages 236-271

se.ethz.ch/~meyer/ongoing/events.pdf

Karine Arnout and Bertrand Meyer: *Pattern Componentization: the Factory Example*, in *Innovations in Systems and Software Technology (a NASA Journal)* (Springer-Verlag), 2006

se.ethz.ch/~meyer/publications/nasa/factory.pdf

Bertrand Meyer and Karine Arnout: *Componentization: the Visitor Example*, in *Computer* (IEEE), vol. 39, no. 7, July 2006, pages 23-30

se.ethz.ch/~meyer/publications/computer/visitor.pdf

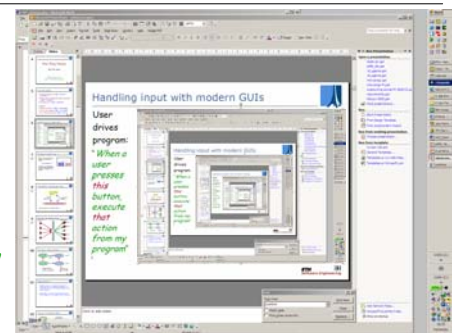
Karine Arnout's thesis: *From Patterns to Components*, March 2004

se.inf.ethz.ch/people/arnout/patterns/

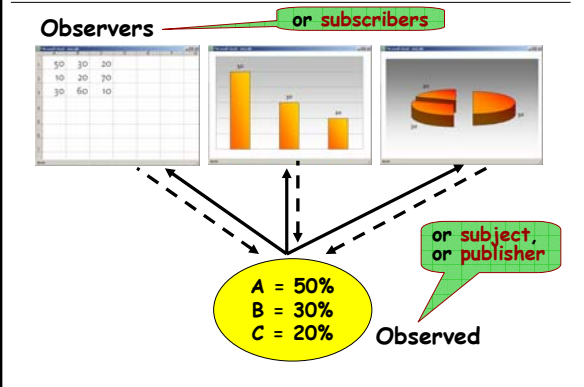
Handling input with modern GUIs

User drives program:

"When a user presses this button, execute that action from my program"



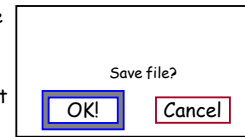
Multiple observers



Event-driven programming: example scenario

One of your classes has a routine
my_procedure

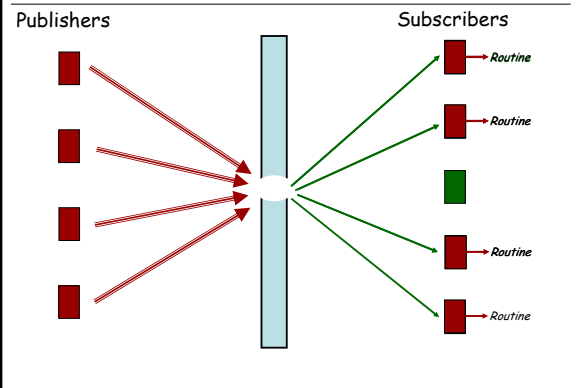
Your application has a GUI object
OK_button



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

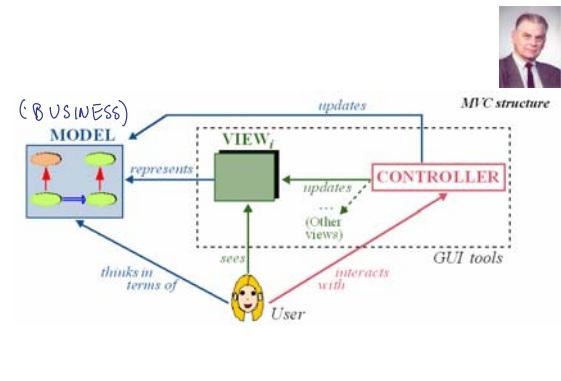
You want to ensure that a mouse click at coordinates $[h, v]$ calls *my_procedure(h, v)*

Event-driven design



Model-View Controller

(Trygve Reenskaug, 1979)



Confusion

Event Event type Uncertain

Events Overview (from .NET documentation)

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.

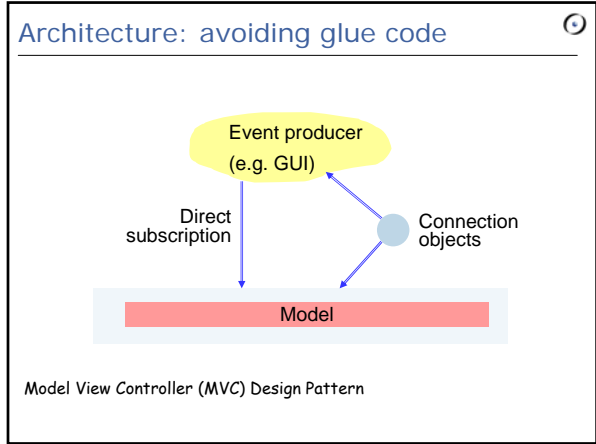
MVC references

Reenskaug's MVC page:

heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html

His original MVC paper:

heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf



Attaching an observer

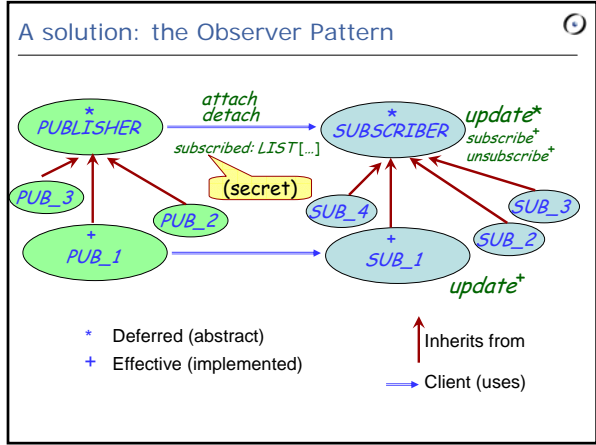
```

In class PUBLISHER:
  feature {SUBSCRIBER}
  attach (s: SUBSCRIBER)
    -- Register s as subscriber to current publisher.
  require
    subscriber_exists: s /= Void
  do
    subscribed.extend(s)
  end
  
```

(selective export)

sub1 sub2 sub_n
subscribers

The invariant of PUBLISHER includes the clause
subscribed /= Void
(subscribed is created by creation procedures of PUBLISHER)



Triggering an event

```

publish is
  -- Ask all observers to
  -- react to current event.
do
  from
  until
  subscribed.start
  subscribed.after
  loop
  subscribed.item.update
  subscribed.forth
  end
end
  
```

Dynamic binding!

sub1 sub2 sub_n
subscribers

Each descendant of SUBSCRIBER defines its own version of update

Observer pattern

Each publisher keeps a list of subscribers:

```

feature {NONE}
  subscribed: LINKED_LIST[SUBSCRIBER]
  
```

To register itself, a subscriber may execute:

```

subscribe (some_publisher)
  
```

where subscribe is defined in SUBSCRIBER as:

```

subscribe (p: PUBLISHER) is
  -- Make current object observe p.
  require
    publisher_exists: p /= Void
  do
    p.attach (Current)
  end
  
```

- ### Observer pattern
- > Subscriber may subscribe to at most one publisher
 - > May subscribe at most one operation
 - > Publishers internally know about subscribers
 - > Not reusable — must be coded anew for each application

Another approach: event-context-action table

Set of triples

[Event type, Context, Action]

Event type: any kind of event we track
Example: left mouse click

Context: object for which these events are interesting
Example: a particular button

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

Event-context-action table may be implemented as e.g. a hash table.



Example scenario (reminder)

One of your classes has a routine
my_procedure

Your application has a GUI object
known as *OK_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 $[h, v]$ calls *my_procedure(h, v)*

Event-context-action table

Event type	Context	Action
Left_click	OK_button	Save_file
Left_click	Cancel_button	Reset
Left_click
Right_click		Display_Menu
...

With .NET delegates: publisher (1)

P1. Introduce new class *ClickArgs* inheriting from *EventArgs*, repeating arguments types of *my_procedure*:

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

P2. Introduce new type *ClickDelegate* (delegate type) based on that class

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

P3. Declare new type *Click* (event type) based on the type *ClickDelegate*:

```
public event ClickDelegate Click
```

Language mechanisms

C and C++: function pointers

C#: delegates

Eiffel: agents

With .NET delegates: publisher (2)

P4. Write new procedure *OnClick* to wrap handling:

```
protected void OnClick (ClickArgs ca)
{if (Click != null) {Click (this, ca.x, ca.y);}}
```

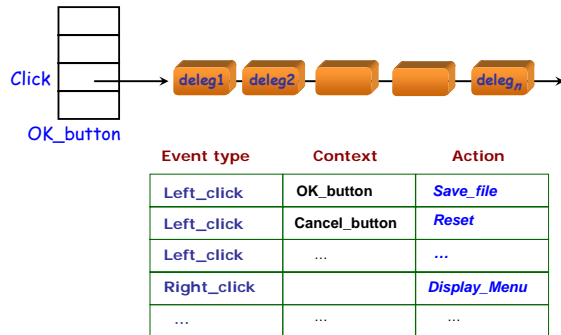
P5. To publish an event of the given type, create new object (instance of *ClickArgs*), passing arguments to constructor:

```
ClickArgs myClickArgs = new ClickArgs (h, v);
```

P6. To publish an event of the given type, trigger event:

```
OnClick (myClickArgs)
```

Event-context-action table in .NET



The Event library

Basically:

- > One generic class: *EVENT_TYPE*
- > Two features: *publish* and *subscribe*

For example:

A button *my_button* that reacts in a way defined in *my_procedure* when clicked (event *mouse_click*)

With .NET delegates: subscriber

D1. Declare a delegate *myDelegate* of type *ClickDelegate*.
(Usually combined with following step.)

D2. Instantiate it with *my_procedure* as argument:

```
ClickDelegate = new ClickDelegate (my_procedure)
```

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

```
OK_button.Click += myDelegate
```

Example using the Event library

The publisher ("subject") creates an event type object:

```
mouse_click: EVENT_TYPE [ TUPLE [ INTEGER, INTEGER ] ] is
  -- Mouse click event type
  once
    create Result
  ensure
    exists: Result /= Void
  end
```

The publisher triggers the event:

```
mouse_click.publish ( [ h, v ] )
```

The subscribers ("observers") subscribe to events:

```
my_button.mouse_click.subscribe (agent my_procedure)
```

Abstractions behind the Eiffel Event Library

Event: each event *type* will be an object
Example: mouse clicks

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

Action: an agent representing a routine
Example: routine to save the file

Event Library specification

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

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

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

Observer pattern vs. Event Library

In case of an existing class *MY_CLASS*:

- > **With the Observer pattern:**
 - Need to write a descendant of *SUBSCRIBER* and *MY_CLASS*
 - ⇒ Useless multiplication of classes
- > **With the Event Library:**
 - Can reuse the existing routines directly as agents

`some_event_type.subscribe (agent existing_routine)`

In a concurrent context (SCOOP)

Volkan Arslan

Use "separate" events:

`temperature_change: separate EVENT_TYPE [TUPLE[REAL]`

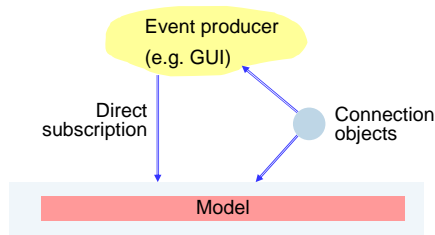
`temperature_change.subscribe (agent my_operation)`

Library provides periodic events, synchronous events...

See Volkan Arslan, Piotr Nienaltowski, and Karine Arnout. "Event library: an object-oriented library for event-driven design". JMLC 2003

- > se.ethz.ch/people/arslan/data/scoop/conferences/Event_Library_JMLC_2003_Arslan.pdf

Architecture: avoiding glue code



Model View Controller (MVC) Design Pattern

Towards a theory of event-driven computation

Execution is solution of

$$h = \text{root} + \text{consequences} (h)$$

Actor
Event_type, abbreviated E
Finite_history = P (E)*
History = P (E)[∞]

$$h1 + h2 = \lambda i | h1 (i) \cup h2 (i)$$

$$h \setminus t = \lambda i | h (i - t)$$

exec: Actor → Finite_history
subscribers: E → P (Actor)
root: Actor

consequence: N × E → Finite_history
consequence (t, e) = $\sum_{s: \text{subscribers}(e)} \text{exec}(e) \setminus t$

consequences: History → History

$$\text{consequences} (h) = \lambda i | \sum_{e: h (i)} \text{consequence} (i)$$

Subscriber variants

`click.subscribe (agent my_procedure)`

`my_button.click.subscribe (agent my_procedure)`

`click.subscribe (agent my_procedure (a, ?, ?, b))`

`click.subscribe (agent other_object.other_procedure)`

Lessons

Simplicity

Search for the right abstractions

Language matters