Trusted Components

Prof. Dr. Bertrand Meyer
October 2006 - February 2007

From design patterns to components

After work by Karine Arnout
Patterns in software development

Document that describes a general solution to a design problem that recurs in many applications. Developers adapt the pattern to their specific application.
Some design patterns

**Creational**
- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

**Structural**
- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

**Behavioral**
- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Erich Gamma, Ralph Johnson, Richard Helms, John Vlissides: *Design Patterns*, Addison-Wesley, 1995
Benefits of design patterns

Capture the knowledge of experienced developers
Publicly available repository
Common pattern language
Newcomers can learn them and apply them to their design
Yield a better structure of the software (modularity, extendibility)
Facilitate discussions between programmers and managers
However: 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."

  Erich Gamma et al., *Design Patterns*, 1995

NOT REUSABLE
A step backwards from reuse:

- No available “pattern libraries”
- Programmers need to implement them each time anew
- A pedagogical tool, not a reuse tool

“A successful pattern cannot just be a book description: it must be a software component”
An example: event-driven development

- The issue
- The Observer pattern solution
- Limitation of the pattern-based solution
- .NET approach
- Componentization using Eiffel agents
Handling input through traditional techniques

Program drives input:

from
  read_next_character
until last_character = Enter loop
  i := i + 1
  Result.put (last_character, i)
  read_next_character
end
Handling input with modern GUIs

User drives program:

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

Publishers

Subscribers

Routine

Routine

Routine

Routine
Event-driven programming: example

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

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

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

Save file?

[OK!]
[Cancel]
Avoiding glue code

Model View Controller (MVC) Design Pattern

Event producer (e.g. GUI)

Direct subscription

Connection objects

Business model (application logic)
A solution: the Observer Pattern

SUBJECT / PUBLISHER

LIBCLASS

attach detach

OBSERVER / SUBSCRIBER

APPCLASS

subscribe+
unsubscribe+

update*

update+

Deferred (abstract)

Effective (implemented)

Inherits from Client (uses)
Observer pattern

Publisher keeps a list of subscribers:

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

To register itself, an subscriber may execute

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

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

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

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

\[ \text{require} \]

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

\[ \text{do} \]

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

\[ \text{end} \]
In class *PUBLISHER*:

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

\[
\quad \text{-- Register } s \text{ as subscriber to current publisher.}
\]

\[
\text{require}
\]

\[
\quad \text{subscriber}\_\text{exists}: \ s \neq \text{Void}
\]

\[
\quad \text{do}
\]

\[
\quad \text{subscribed}.\text{extend} \ (s)
\]

\[
\quad \text{end}
\]

Note that invariant of *PUBLISHER* includes the clause

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

(List *subscribed* is created by creation procedures of *PUBLISHER*)
Triggering an event

trigger is

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

do

do from

subscribed.start

until

subscribed.after

loop

subscribed.item.update

subscribed.forth

end

end

Each descendant of **OBSERVER** defines its own version of **update**
Observer pattern

- Publishers know about subscribers
- Subscriber may subscribe to at most one publisher
- May subscribe at most one operation
- Not reusable — must be coded anew for each application
Another approach: 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.
The EiffelVision style

my_button.click.action_list.extend(agent my_procedure)
Mechanisms in other languages

C and C++: “function pointers”

C#: delegates (more limited form of agents)
P1. Introduce new class `ClickArgs` inheriting from `EventArgs`, repeating arguments types of `myProcedure`:

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

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

```csharp
public void delegate ClickDelegate (ClickArgs e)
```

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

```csharp
public event ClickDelegate Click
```
P4. Write new procedure `OnClick` to wrap handling:

```csharp
protected void OnClick (int x, int y)
{
    if (Click != null) {Click (this, x, y)}
}
```

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

```csharp
ClickArgs myClickargs = new ClickArgs (h, v)
```

P6. For every event occurrence, trigger event:

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

D2. Instantiate it with **myProcedure** as argument:

```
myDelegate = new ClickDelegate (myProcedure)
```

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

```
YES_button.Click += myDelegate
```
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
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):
Example using the Event library

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

```plaintext
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:

```plaintext
mouse_click.publish ([x_position, y_position])
```

The subscribers ("observers") subscribe to events:

```plaintext
my_button.mouse_click.subscribe (agent my_procedure)
```
The EiffelVision style

\[ \text{YES\_button.click.action\_list.extend(agent my\_procedure)} \]
Event Library style

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)
```
Subscriber variants

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

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

```python
click.subscribe(agent your_procedure (a, ?, ?, b))
```

```python
click.subscribe(agent other_object.other_procedure)
```
Another example of using agents

\[
\int_{a}^{b} my\_function\ (x)\ dx
\]

\[
\int_{a}^{b} your\_function\ (x, u, v)\ dx
\]

\[
my\_integrator\_integral\ (\text{agent } my\_function, a, b)
\]

\[
my\_integrator\_integral\ (\text{agent } your\_function (?, u, v), a, b)
\]
Applications of agents

- Undo-redo
- Iteration
- High-level contracts
- Numerical programming
- Introspection (finding out properties of the program itself)
An encouraging success

- A book idea: the Observer pattern
- A reusable library: the Event library

Let’s go further and explore all design patterns...
Results

A new classification of design patterns:

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

“Pattern library” and “Pattern Wizard”
Results

- Not Componentizable (2) 9%
- Wizard or Library Support (6) 26%
- Partially Componentizable (4) 17%
- Fully Componentizable (11) 48%
### Creational design patterns

<table>
<thead>
<tr>
<th>Artificial design patterns</th>
<th>Reusable design patterns</th>
<th>Remaining design patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>Abstract Factory</td>
<td>Builder</td>
</tr>
<tr>
<td></td>
<td>Factory Method</td>
<td>Singleton</td>
</tr>
</tbody>
</table>
Prototype: an artificial pattern

Intent:

- "Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype." [Gamma 1995]

In Eiffel, every object is a prototype!
Abstract Factory: fully componentizable

Intent:

- "Provide an interface for creating families of related or dependent objects without specifying their concrete classes." [Gamma 1995, p 87]
deferred class FACTORY

feature -- Factory methods

  new_product_a : PRODUCT_A is
    -- New product of type PRODUCT_A
    deferred
    ensure
      exists: Result /= Void
    end

  new_product_b : PRODUCT_B is
    -- New product of type PRODUCT_B
    deferred
    ensure
      exists: Result /= Void
    end

end
Class FACTORY_1

class FACTORY_1

inherit FACTORY

feature -- Factory methods

new_product_a: PRODUCT_A1 is
  -- New product of type PRODUCT_A1
  do
    create Result
  end

new_product_b: PRODUCT_B1 is
  -- New product of type PRODUCT_B1
  do
    create Result
  end

end
Using the Abstract Factory pattern

With:

```ruby
car_factory: CAR_FACTORY is
  -- Factory of cars
  once
  create Result
  ensure
  exists: Result /= Void
end
```

```ruby
simulated_traffic.add_vehicle(
  car_factory.new_car(p, d, w, h))
```
Criticism

Code redundancy:
- The factory classes, e.g. `CAR_FACTORY` and `BUS_FACTORY` will be similar

Lack of flexibility:
- `FACTORY` fixes the set of factory functions `new_product_a` and `new_product_b`
The Factory library

class FACTORY[G]

create

make

feature -- Initialization

make (f: like factory_function) is
    -- Initialize with factory_function set to f.
    require
        exists: f /= Void
    do
        factory_function := f
    ensure
        set: factory_function = f
    end

feature -- Access

factory_function: FUNCTION[ANY, TUPLE[], G]
    -- Factory function creating new instances of type G
feature - Factory methods

new: G is
     -- New instance of type G
     do
         factory_function.call([])
         Result := factory_function.last_result
     ensure
         exists: Result /= Void
     end

new_with_args(args: TUPLE): G is
     -- New instance of type G initialized with args
     do
         factory_function.call(args)
         Result := factory_function.last_result
     ensure
         exists: Result /= Void
     end

invariant

   exists: factory_function /= Void

end
**Sample application**

```
simulated_traffic: TRAFFIC

simulated_traffic.add_vehicle(...)```
With the Abstract Factory pattern

```
simulated_traffic.add_vehicle (car_factory.new_car (p, d, w, h))
```

With:

```
car_factory: CAR_FACTORY is
   -- Factory of cars
   once
      create Result
   ensure
      exists: Result /= Void
   end
```
With the Factory library

```
simulated_traffic.add_vehicle(
    car_factory.new_with_args([p, d, w, h]))
```

With:

```
car_factory: FACTORY [CAR] is
    -- Factory of cars
    once
        create Result.make(agent new_car)
    ensure
        exists: Result /= Void
end
```
With the Factory library

and:

\[\text{new\_car}(p, d, w, h: \text{INTEGER}): \text{CAR} \text{ is}\]
\[
\quad \text{-- New car with power engine } p,
\quad \text{-- wheel diameter } d,
\quad \text{-- door width } w, \text{ door height } h
\]
\[
\text{do}
\quad \text{-- Create car } \text{engine, wheels, and doors.}
\quad \text{create } \text{Result.make (engine, wheels, doors)}
\]
\[
\text{ensure}
\quad \text{exists: } \text{Result} \neq \text{Void}
\]
\[
\text{end}
\]
Factory pattern vs. library

Benefits:
- Get rid of some code duplication
- Fewer classes
- Reusability

One caveat though:
- Likely to yield a bigger client class (because similarities cannot be factorized through inheritance)
Visitor
Visitor - Intent

“Represents an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.”

[Gamma et al., p 331]

- Static class hierarchy
- Need to perform traversal operations on corresponding data structures
- Avoid changing the original class structure
Visitor application example

Set of classes to deal with XML documents
- `XML_NODE`
- `XMLDOCUMENT`
- `XML_ELEMENT`
- `XML_ATTRIBUTE`
- `XML_CONTENT`

One parser
Many formatters
- Pretty-print
- Compress
- Convert to different encoding
- ...

Another example

AST of program
- Nodes: Class, Feature, instruction, ...
- Operations:
  - Compile
  - Pretty print
  - Generate documentation
  - Refactor
We want to add external functionality, for example:

- Maintenance
- Visualization
**Maintenance**

\[
maintain(b: \text{BORROWABLE}) \text{ is} \]

\[\begin{array}{l}
\text{-- Perform maintenance operations on } n. \\
\text{require} \\
\text{exists: } b \neq \text{Void} \\
\text{local} \\
\text{book: } \text{BOOK} \\
\text{dvd: } \text{DVD} \\
\text{do} \\
\text{book }\neq b \\
\text{if } \text{book }\neq \text{Void} \text{ then} \\
\quad \text{... Book maintenance ...} \\
\text{end} \\
\text{dvd }\neq b \\
\text{if } \text{dvd }\neq \text{Void} \text{ then} \\
\quad \text{... DVD maintenance ...} \\
\text{end} \\
\text{end} \\
\text{end}
\end{array}\]
Visualization

\textit{display (}\textit{b: BORROWABLE}) \textit{is}

\begin{verbatim}
  -- Display \textit{b}.
  require
  exists: \textit{b} \neq \textit{Void}
  local
  \textit{book: BOOK}
  \textit{dvd: DVD}
  do
    \textit{book ?= b}
    if \textit{book} \neq \textit{Void} then
      \ldots \text{Put book on display} \ldots
    end
    \textit{dvd ?= b}
    if \textit{dvd} \neq \textit{Void} then
      \ldots \text{Put DVD on display} \ldots
    end
  end
end
\end{verbatim}

Why is this approach bad?
Visitor: overall architecture

- **BORROWABLE**
  - + **BOOK**
    - accept^+
  - + **DVD**
    - accept^+

- **VISITOR**
  - + **MAINTENANCE_VISITOR**
    - visit_book^+
    - visit_dvd^+
  - + **DISPLAY_VISITOR**
    - visit_book^+
    - visit_dvd^+
class MAINTENANCE_VISITOR

inherit VISITOR

feature -- Basic operations

visit_book (b: BOOK) is
  -- Perform maintenance operations on b.
  do
    b.check_binding
    if b.damaged then
      b.repair
    end
  end

visit_dvd (d: DVD) is
  -- Perform maintenance operations on d.
  do
    d.check_surface
    if d.damaged then
      d.order_replacement
    end
  end
end
Class BOOK

class
    BOOK
inherit
    BORROWABLE
feature
    accept (v: VISITOR) is
        -- Apply to v the book visit mechanism.
        do
            v.visit_book (Current)
        end
end
class
  DVD
inherit
  BORROWABLE
feature -- Visitor pattern
  accept (v: VISITOR) is
    do
      v.visit_dvd (Current)
    end
end
end
Visitor - Usage

local
  item: BORROWABLE
  maintainer: MAINTENANCE_VISITOR

do
  ...
  item.accept (maintainer)
  ...
end

Calls VISITOR.visit_book

Calls VISITOR.visit_dvd
Visitor - Participants

**Visitor**

*Common ancestor for all concrete visitors.*

**Concrete Visitor**

*Represents a specific operation, applicable to all elements.*

**Element**

*Common ancestor for all concrete elements.*

**Concrete Element**

*Represents a specific element in class hierarchy.*
Makes adding new operations easy
Gathers related operations, separates unrelated ones
Avoids assignment attempts
  ➢ Better type checking
Adding new concrete element is hard
Intent:

- “Ensure a class only has one instance, and provide a global point of access to it.”

[Gamma 1995, p 127]
Wrong approach

class  
  SINGLETON  

feature {NONE} -- Implementation

  frozen the_singleton: SINGLETON is
    -- The unique instance of this class
    once
      Result := Current
    end
  
  invariant

    only_one_instance: Current = the_singleton
  
end
Wrong approach

defered class
    \texttt{SHARED\_SINGLETON}
feature \{NONE\} -- Implementation

    \texttt{singleton:}\ \texttt{SINGLETON is}
        -- Access to unique instance
        deferred
    end

    \texttt{is\_real\_singleton:}\ \texttt{BOOLEAN is}
        -- Do multiple calls to singleton return the same result?
        do
            Result := singleton = singleton
        end

invariant
    \texttt{singleton\_is\_real\_singleton:}\ \texttt{is\_real\_singleton}
end
What’s wrong?

If one inherits from `SINGLETON` several times:

- The inherited feature `the_singleton` keeps the value of the first created instance.
- Violates the invariant of class `SINGLETON` in all descendant classes except the one for which the singleton was created first.

There must only be one singleton per system.
A correct Singleton example

class MY_SHARED_SINGLETON feature -- Access
  singleton: MY_SINGLETON is
    -- Singleton object
    do
      Result := singleton_cell.item
      if Result = Void then create Result.make end
    ensure
      created: singleton_created
      exists: Result /= Void
    end
end

feature -- Status report
  singleton_created: BOOLEAN is
    -- Has singleton already been created?
    do Result := singleton_cell.item /= Void end

feature {NONE} -- Implementation
  singleton_cell: CELLS [MY_SINGLETON] is
    -- Cell containing the singleton if already created
    once
      create Result.put (Void)
    ensure
      exists: Result /= Void
    end
end
class MY_SINGLETON inherit MY_SHARED_SINGLETON

create
make

feature {NONE} -- Initialization
make is
  -- Create a singleton object.
  require
  singleton_not_created: not singleton_created
do
  singleton_cell.put (Current)
end

invariant
  singleton_created: singleton_created
  singleton_pattern: Current = singleton
A Singleton in Eiffel: impossible?

Having frozen classes (from which one cannot inherit) would enable writing singletons in Eiffel

But it would still not be a reusable solution
References: Design patterns

Gamma et al.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

References: From patterns to components


Karine Arnout: From Patterns to Components, PhD thesis, ETH, 2004 (see SE Web site)

Exercises

Undo-redo
Factory method