Trusted Components: From design patterns to components

Bertrand Meyer
After work by Karine Arnout
Benefits of design patterns

- Capture the knowledge of experienced developers
- Publicly available "repository"
- Newcomers can learn them and apply them to their design
- Yield a better structure of the software (modularity, extendibility)
- Common pattern language
- 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”
A successful story: the Observer pattern

**SUBSCRIBER**
- update*

**PUBLISHER**
- add_subscriber*
  - remove_subscriber*
  - notify_subscribers*

**MY_SUBSCRIBE**
- update+

**MY_PUBLISH**er
- add_subscriber+
  - remove_subscriber+
  - notify_subscribers+

Deferred (abstract) class
- *

Effective (concrete) class
- +

Deferred feature
- f*

Effective (implemented) feature
- f+
Handling traditional input

Program drives input:

```plaintext
from read_next_character
until last_character = Enter loop
  i := i + 1
  Result.put (last_character, i)
read_next_character
end
```
User drives program:

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

Publishers

Subscribers

Routine

Routine

Routine

Routine

Routine
A successful componentization: Observer

**Deferred (abstract) class**

**Effective (concrete) class**

**Deferred feature**

**Effective (implemented) feature**

**update**

**add_subscriber**

**remove_subscriber**

**notify_subscribers**
deferred class PUBLISHER feature -- Observer pattern

add_subscriber (s: SUBSCRIBER) is
    -- Add s to the list of subscribers.
    require
        not_yet_subscriber: not subscribers.has (s)
    do
        subscribers.extend (s)
    ensure
        added: subscribers.has (s)
        one_more: subscribers.count = old subscribers.count + 1
end

remove_subscriber (s: SUBSCRIBER) is
    -- Remove s from the list of subscribers.
    require
        is_subscriber: subscribers.has (s)
    do
        subscribers.search (s) ; subscribers.remove
    ensure
        removed: not subscribers.has (s)
        one_less: subscribers.count = old subscribers.count - 1
end
Class \textit{PUBLISHER}

\begin{verbatim}
notify_subscribers is
  -- Notify all subscribers.
  -- (Call \textit{update} on each subscriber.)
  do
    from
    until
    loop
      subscribers.item.update
      subscribers.forth
    end
  end

subscribers: \texttt{LINKED\_LIST [SUBSCRIBER]}
  -- List of subscribers

invariant
  subscribers_exist: subscribers /= \texttt{Void}
end
\end{verbatim}
deferred class
   SUBSCRIBER

feature  -- Observer pattern

   update is
      -- Update subscriber according to the state of
      -- publisher data.
   deferred
   end

   data: PUBLISHER
      -- Observable data
   end
class MY_DISPLAY
inherit SUBSCRIBER
  redefine data end
create make
feature -- Initialization
  make is
    -- Initialize GUI and register an subscriber of data.
    do
      create add_button.make_with_text_and_action
        (“Add”, agent on_add)
      create remove_button.make_with_text_and_action
        (“Remove”, agent on_remove)
      data.add_subscriber (Current)
    end
feature -- Access
  add_button: EV_BUTTON
    -- Button with label Add
  remove_button: EV_BUTTON
    -- Button with label Remove
data: MY_DATA
    -- Data to be observed

feature -- Event handling
  on_add is
    -- Action performed when add_button is pressed
    do data.add end

on_remove is
    -- Action performed when remove_button is pressed
    do data.remove end

feature -- Observer pattern
  update is
    -- Update GUI.
    do
        -- Something here
    end
end
Observer pattern: limitations

- Each publisher object knows about its observers
- Only one update procedure in *SUBSCRIBER*:
  - Subscribe to at most one publisher
  - At most one operation
- Not reusable — must be coded anew for each application
  (This is the difference between patterns and components)
A typical PUBLISHER

class
   MY_DATA

inherit
   PUBLISHER

feature -- Observer pattern

   add is
      do
         -- Add Current to data to be observed.
         -- Do something.
         notify_subscribers
      end

   remove is
      do
         -- Remove Current from data to be observed.
         -- Do something.
         notify_subscribers
      end

Redundancy:
→ Hardly maintainable
→ Not reusable
The Event library

- Basically:
  - One generic class: \textit{EVENT\_TYPE}
  - Two features: \textit{publish} and \textit{subscribe}

- For example: A button \textit{my\_button} that reacts in a way defined in \textit{my\_procedure} when clicked (event \textit{mouse\_click}):
Example using the Event library

- The publisher ("publisher") 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 ("subscribers") subscribe to events:

```plaintext
my_button.mouse_click.subscribe (agent my_procedure)
```
An encouraging success

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

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

- A new classification of design patterns:
  - Artificial patterns
  - Reusable patterns
  - Resistant patterns

- A “pattern library” made of the reusable components obtained from design patterns

- Code templates otherwise
Creational design patterns

<table>
<thead>
<tr>
<th>Artificial design patterns</th>
<th>Reusable design patterns</th>
<th>Resistant 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>
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: a reusable pattern

- 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
class FACTORY_1

inherit

FACTORY

feature -- Factory methods

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

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

- **Code redundancy:**
  - `FACTORY_1` and `FACTORY_2` will be similar

- **Lack of flexibility:**
  - `FACTORY` fixes the set of factory functions `new_product_a` and `new_product_b`
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
simulated_traffic: TRAFFIC

simulated_traffic.add_vehicle (...)

Simulated Traffic: TRAFFIC

+ SIMULATION
  + TRAFFIC
    * VEHICLE
      + CAR
      + BUS
      + METRO

Chair of Software Engineering
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
```
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
and:

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

- Prototype
- Abstract Factory
- Factory Method
- Builder
- Singleton
**Intent:**

“Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.” [Gamma 1995, p 107]

A special case of the Abstract Factory
Builder: a remaining pattern

- Intent:
  - “Separate the construction of a complex object from its representation so that the same construction process can create different representations.” [Gamma 1995, p 97]
deferred class
BUILD

feature -- Access

last_product: PRODUCT is
   -- Product under construction
deferred
end

feature -- Basic operations

build is
   -- Create and build last_product.
do
   build_product
   build_part_a
   build_part_b
ensure
   exists: last_product /= Void
end

...
A reusable builder?

- Issue:
  - How to know how many parts the product has?
    - Not reusable
  - Handle some usual cases, e.g. a “two part builder” by reusing the Factory library:

```java
class TWO_PART_BUILDER [F -> BUILDABLE, G, H]
  -- Build a product of type F
  -- composed of two parts:
  -- the first part of type G,
  -- the second part of type H.
```
Class *BUILDABLE*

deferred class *BUILDABLE* feature -- Access

\[
g: \text{ANY}
\]
-- First part of the product to be created

\[
h: \text{ANY}
\]
-- Second part of the product to be created

feature \{ *TWO_PART_BUILDER* \} -- Status setting

-- set\_g
-- set\_h

end
Singleton: a remaining pattern

- **Intent:**
  - “Ensure a class only has one instance, and provide a global point of access to it.”
  
  [Gamma 1995, p 127]

Harder than it looks...
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
  SHARED_SINGLETON

feature {NONE} -- Implementation

  singleton: SINGLETON is
    -- Access to unique instance
    deferred
  end

  is_real_singleton: BOOLEAN is
    -- Do multiple calls to singleton return the same result?
    do
      Result := singleton = singleton
    end

invariant

  singleton_is_real_singleton: 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: CELL [MY_SINGLETON] is
        -- Cell containing the singleton if already created
        once
            create Result.put (Void)
        ensure
            exists: Result /= Void
        end
end

Chair of Software Engineering
A correct Singleton example

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
end

In fact, one can still break it by:
- Cloning a singleton.
- Using persistence.
- Inheriting from MY_SHARED_SINGLETON and putting back Void to the cell after the singleton has been created.
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
### Structural design patterns

<table>
<thead>
<tr>
<th>Artificial design patterns</th>
<th>Reusable design patterns</th>
<th>Resistant design patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>▪ Composite</td>
<td>▪ Proxy</td>
</tr>
<tr>
<td></td>
<td>▪ Flyweight</td>
<td>▪ Decorator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▪ Adapter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▪ Bridge</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▪ Facade</td>
</tr>
</tbody>
</table>
References: Design patterns

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

References: From patterns to components


