

Software Architecture

Bertrand Meyer

ETH Zurich, March-July 2007

Lecture 11: More patterns: bridge, composite, decorator, facade, flyweight

Last update: 5 June 2007

Today

Creational

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Structural

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

Behavioral

- > Chain of Responsibility
- > Command (undo/redo)
- > Interpreter
- > Iterator
- > Mediator
- > Memento
- > Observer
- > State
- > Strategy
- > Template Method
- > Visitor

Bridge pattern

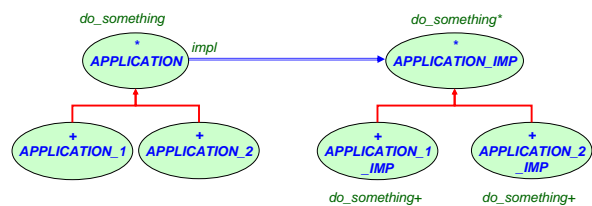
"Decouple[s] an abstraction from its implementation so that the two can vary." [GoF, p 151]

In other words:

It separates the class interface (visible to the clients) from the implementation (that may change later)



Bridge: Original pattern



Bridge: Classes

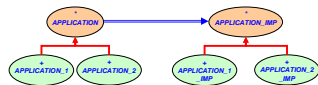
```
deferred class APPLICATION
  feature {NONE} -- Initialization
    make(an_impl: like impl) is
      -- Set impl to an_impl.
      require
        an_impl /= Void
      do
        impl := an_impl
      ensure
        impl_set: an_impl = impl
      end
  end

  feature {NONE} -- Implementation
    impl: APPLICATION_IMP -- Implementation
  end

  feature -- Basic operations
    do_something is
      -- Do something.
      do
        impl.do_something
      end
  end

  invariant
    impl_not_void: impl /= Void
end

deferred class APPLICATION_IMP
  feature -- Basic operations
    do_something is
      -- Do something.
      deferred
      end
  end
end
```



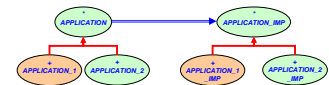
Bridge: Classes

```
class APPLICATION_1
  inherit APPLICATION

  create
    make
  end
end

class APPLICATION_1_IMP
  inherit APPLICATION

  feature -- Basic operations
    do_something is
      -- Do something.
      do
        ...
      end
  end
end
```



Bridge: Client view

class *CLIENT*

create
make

feature -- Basic operations

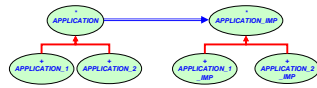
make is

local -- Do something.

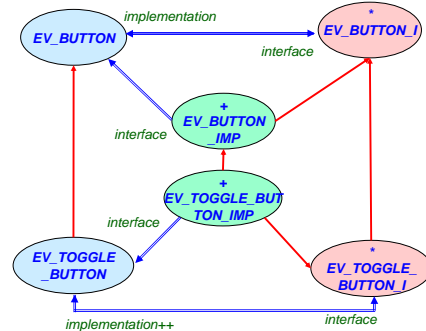
app1: APPLICATION_1
app2: APPLICATION_2

do
create app1.make (create {APPLICATION_1_IMP})
app1.do_something
create app2.make (create {APPLICATION_2_IMP})
app2.do_something

end



Bridge: A variation used in Vision2



Bridge: Vision2 example

class *EV_BUTTON*

...

feature {EV_ANY, EV_ANY_I} -- Implementation

implementation: EV_BUTTON_I -- Implementation

feature {NONE} -- Implementation

create_implementation is

-- Do something.

do
create {EV_BUTTON_IMP} implementation.make (Current)

end

end

Bridge: Advantages (or when to use it)

- No permanent binding between abstraction and implementation
- Abstraction and implementation extendible by subclassing
- Implementation changes have no impact on clients
- Implementation of an abstraction completely hidden from clients
- Implementation share with several objects, hidden from clients

Bridge: Componentization

- Non-componentizable (no library support)

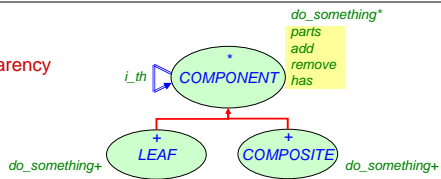
Composite pattern

"Way to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly." [GoF, p 163]

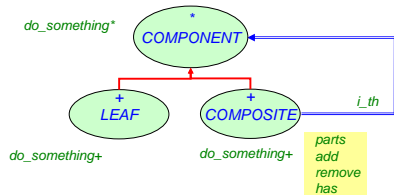


Composite: Original pattern

Transparency version



Safety version



Composite pattern, safety version (1/5)

```
deferred class
  COMPONENT
```

```
feature -- Basic operation
  do_something is
    -- Do something.
    deferred
    end
```

```
feature -- Status report
  is_composite: BOOLEAN is
    -- Is component a composite?
    do
      Result := False
    end
end
```

Composite pattern, safety version (2/5)

```
class
  COMPOSITE
inherit
  COMPONENT
  redefine
    is_composite
end
create
  make,
  make_from_components
feature {NONE} -- Initialization
  make is
    -- Initialize component parts.
    do
      create parts.make
    end
```

Composite pattern, safety version (3/5)

```
make_from_components(some_components: like parts) is
  -- Set parts to some_components.
  require
    some_components_not_void: some_components /= Void
    no_void_component: not some_components.has(Void)
  do
    parts := some_components
  ensure
    parts_set: parts = some_components
  end
feature -- Status report
  is_composite: BOOLEAN is
    -- Is component a composite?
    do
      Result := True
    end
```

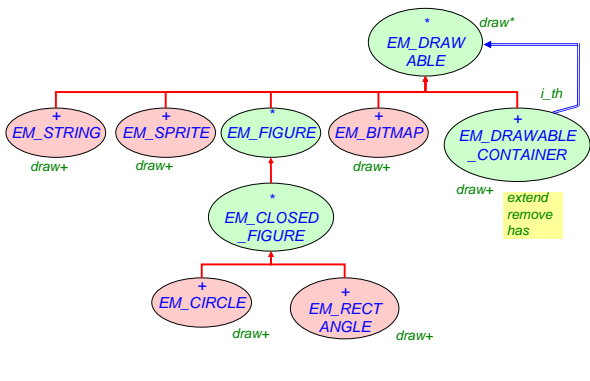
Composite pattern, safety version (4/5)

```
feature -- Basic operation
  do_something is
    -- Do something.
    do
      from parts.start until parts.after loop
        parts.item.do_something
      parts.forth
    end
  end
feature -- Access
  item: COMPONENT is
    -- Current part of composite
    do
      Result := parts.item
    ensure
      definition: Result = parts.item
      component_not_void: Result /= Void
    end
```

Composite pattern, safety version (5/5)

```
feature -- Others
  -- Access: i_th, first, last
  -- Status report: has, is_empty, off, after, before
  -- Measurement: count
  -- Element change: add
  -- Removal: remove
  -- Cursor movement: start, forth, finish, back
feature {NONE} -- Implementation
  parts: LINKED_LIST[like item]
  -- Component parts
  -- (which are themselves components)
invariant
  is_composite: is_composite
  parts_not_void: parts /= Void
  no_void_part: not parts.has(Void)
end
```

Composite: Variation used in EiffelMedia



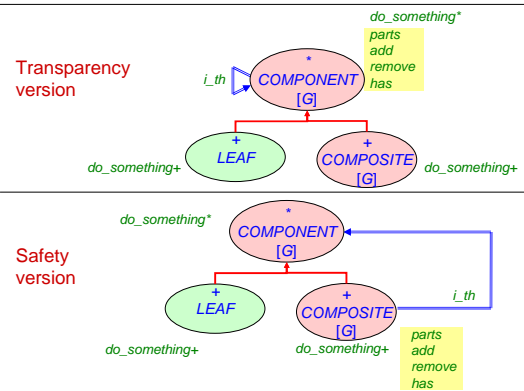
Composite: Advantages (or when to use it)

- Represent part-whole hierarchies
- Clients treat compositions and individual objects uniformly

Composite: Componentization

- Fully componentizable
- Library support
- Main idea: Use genericity

Composite: Pattern library version

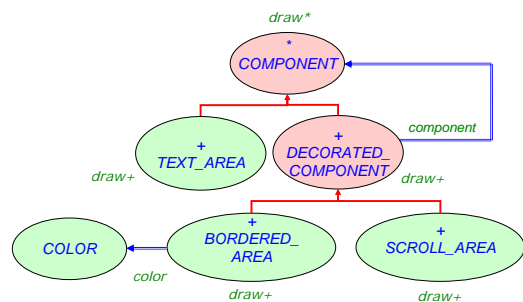


Decorator pattern

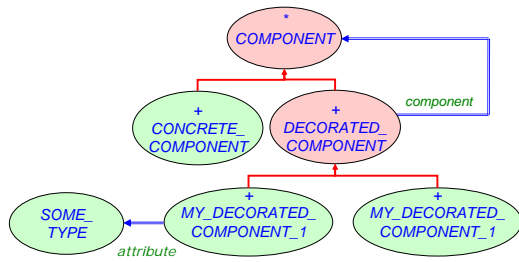
"Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality." [GoF, p 175]



Decorator: Example



Decorator: Original pattern



Decorator: Exporting additional features?

- Newly introduced features do not need to be visible to clients, but they may.

e.g. Display an area with a border of a certain color

```

class
  BORDERED_AREA
inherit
  DECORATED_COMPONENT
...
feature
  color: COLOR
  set_color(a_color: like color) is ...
  draw is
    do
      draw_border(color)
      component.draw
    end
end
end

```

Client can change the *color* by calling *set_color* if they have direct access to the *BORDERED_AREA*

Decorator: Advantages (or when to use it)

- Add responsibilities to individual objects dynamically and transparently
- Responsibilities can be withdrawn
- Omit explosion of subclasses to support combinations of responsibilities

Decorator: Componentization

- Non-componentizable
- Skeleton classes can be generated

Decorator skeleton, attribute (1/2)

```

indexing
  description: "Skeleton of a component decorated with additional attributes"
class
  DECORATED_COMPONENT -- You may want to change the class name.
inherit
  COMPONENT -- You may need to change the class name
  redefine
    -- List all features of COMPONENT that are not deferred.
  end
create
  make
  -- You may want to add creation procedures to initialize the additional attributes.
feature {NONE} -- Initialization
  make(a_component: like component) is
    -- Set component to a_component.
    require
      a_component_not_void: a_component /= Void
    do
      component := a_component
    ensure
      component_set: component = a_component
    end
  -- List additional creation procedures taking into account additional attributes.

```

Decorator skeleton, attribute (2/2)

```

feature -- Access
  -- List additional attributes.

feature -- To be completed
  -- List all features from COMPONENT and implement them by
  -- delegating calls to component as follows:
  -- do
  --   component.feature_from_component
  -- end

feature {NONE} -- Implementation
  component: COMPONENT
  -- Component that will be used decorated

invariant
  component_not_void: component /= Void
end

```

Decorator skeleton, behavior (1/2)

```

indexing
  description: "Skeleton of a component decorated with additional behavior"
class
  DECORATED_COMPONENT -- You may want to change the class name.
inherit
  COMPONENT -- You may need to change the class name
  redefine
    -- List all features of COMPONENT that are not deferred.
  end
create
  make
feature {NONE} -- Initialization
  make(a_component: like component) is
    -- Set component to a_component.
  require
    a_component_not_void: a_component /= Void
  do
    component := a_component
  ensure
    component_set: component = a_component
  end
  
```

Decorator skeleton, behavior (2/2)

```

feature -- To be completed
  -- List all features from COMPONENT and implement them by
  -- delegating calls to component as follows:
  -- do
  --   component.feature_from_component
  -- end

  -- For some of these features, you may want to do something more:
  -- do
  --   component.feature_from_component
  --   do_something_more
  -- end

feature {NONE} -- Implementation
  component: COMPONENT
  -- Component that will be used for the "decoration"

invariant
  component_not_void: component /= Void
end
  
```

Decorator skeleton: Limitations

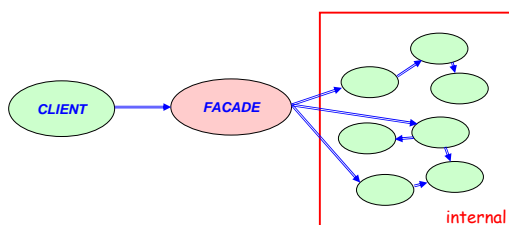
- feature** -- To be completed
- List all features from *COMPONENT* and implement them by
 - delegating calls to *component* as follows:
 - do
 - *component.feature_from_component*
 - end
- Does not work if *feature_from_component* is:
 - an **attribute**: cannot redefine an attribute into a function (Discussed at ECMA)
 - a **frozen feature** (rare): cannot be redefined, but typically:
 - Feature whose behavior does not need to be redefined (e.g. *standard_equal*, ... from *ANY*)
 - Feature defined in terms of another feature, which can be redefined (e.g. *clone* defined in terms of *copy*)

Facade

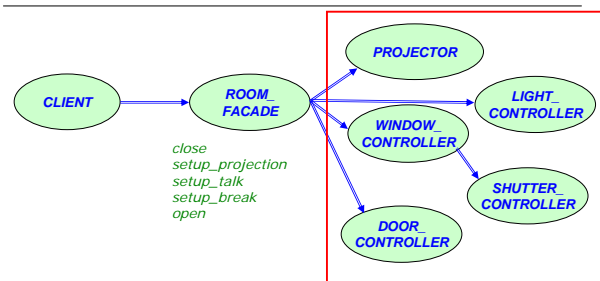
"Provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use." [GoF, p 185]



Facade: Original pattern



Facade: Example



Other example: Compiler, where clients should not need to know about all internally used classes.

Facade: Advantages (or when to use it)

- Provides a simple interface to a complex subsystem
- Decouples clients from the subsystem and fosters portability
- Can be used to layer subsystems by using facades to define entry points for each subsystem level

Facade: Componentization

- Non-componentizable

Flyweight pattern

"Use sharing to support large numbers of fine-grained objects efficiently." [GoF, p 195]

Without the Flyweight pattern (1/2)

```
class
  CLIENT
...
feature -- Basic operation
  draw_lines is
    -- Draw some lines in color.
    local
      line1, line2: LINE
      red: INTEGER
    do
      ...
      create line1.make (red, 100, 200)
      line1.draw
      create line2.make (red, 100, 400)
      line2.draw
    ...
  end
...
end
```

Creates one LINE object for each line to draw

Without the Flyweight pattern (2/2)

```
class interface
  LINE
create
  make
feature -- Initialization
  make (a_color, x, y: INTEGER)
    -- Set color to a_color, x as x_position, and y as y_position.
  ensure
    color_set: color = a_color
    x_set: x_position = x
    y_set: y_position = y
feature -- Access
  color: INTEGER
    -- Line color
  x_position, y_position: INTEGER
    -- Line position
feature -- Basic operation
  draw
    -- Draw line at position (x_position, y_position) with color.
end
```

With the Flyweight pattern (1/3)

```
class
  CLIENT
feature -- Basic operation
  draw_lines is
    -- Draw some lines in color.
    local
      line_factory: LINE_FACTORY
      red_line: LINE
      red: INTEGER
    do
      ...
      red_line := line_factory.new_line (red)
      red_line.draw (100, 200)
      red_line.draw (100, 400)
    ...
  end
...
end
```

Creates only one LINE object per color

With the Flyweight pattern (2/3)

```
class interface
  LINE_FACTORY

feature -- Initialization
  new_line(a_color: INTEGER): LINE
    -- New line with color a_color
  ensure
    new_line_not_void: Result /= Void
...
end
```

With the Flyweight pattern (3/3)

```
class interface
  LINE
create
  make
feature -- Initialization
  make(a_color: INTEGER) is
    -- Set color to a_color.
  ensure
    color_set: color = a_color
feature -- Access
  color: INTEGER
    -- Line color
feature -- Basic operation
  draw(x, y: INTEGER)
    -- Draw line at position (x, y) with color.
end
```

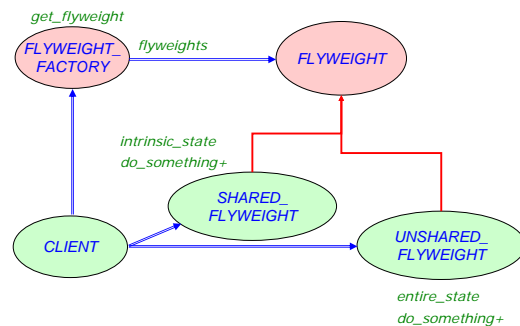
Shared/unshared and (non-)composite objects

- Two kinds of properties:
 - Intrinsic characteristics stored in the flyweight
 - Extrinsic characteristics moved to the client (typically a "flyweight context")

The color of the LINE

The coordinates of the LINE

Flyweight: Original pattern



Flyweight pattern: Description

- Intent:** "Use sharing to support large numbers of fine-grained objects efficiently."
- Participants:**
 - FLYWEIGHT:** Offers a service *do_something* to which the extrinsic characteristic will be passed
 - SHARED_FLYWEIGHT:** Adds storage for intrinsic characteristic
 - COMPOSITE_FLYWEIGHT:** Composite of shared flyweight; May be shared or unshared
 - FACTORY:** Creates and manages the flyweight objects
 - CLIENT:** Maintains a reference to flyweight, and computes or stores the extrinsic characteristics of flyweight

Shared/unshared and (non-)composite objects

- Two kinds of flyweights:
 - Composites (shared or unshared)
 - Non-composites (shared)

Flyweight: Advantages (or when to use it)

- Reduces storage costs if a large number of objects is used
 - through reducing the number of objects by using shared objects
 - the amount of intrinsic state
 - through computation of the extrinsic state

Flyweight pattern: Adapted pattern



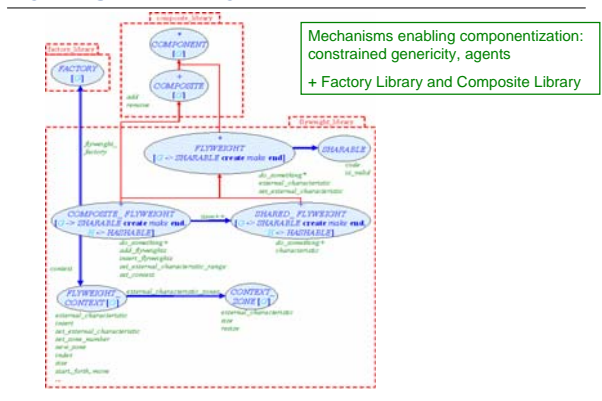
Flyweight context

- External characteristics are not stored in the flyweight object → client must handle them
- A possibility is to create a *FLYWEIGHT_CONTEXT* describing a list of flyweights grouped by *FLYWEIGHT_ZONES* with the same external characteristic (e.g. characters with the same color in a row of a book page)

Flyweight: Componentization

- Fully componentizable

Flyweight Library



Flyweight Library: use of the Composite Library

- Two kinds of flyweights:
 - Non-composites (shared)
 - Composites (shared or unshared)
- Use the Composite Library
 - *FLYWEIGHT[G]* inherits from *COMPONENT[FLYWEIGHT[G]]*
 - *COMPOSITE_FLYWEIGHT[G, H]* inherits from *FLYWEIGHT[G]* and *COMPOSITE[FLYWEIGHT[G]]*

Uses the safety version of the Flyweight Library where the *COMPONENT* does not know about its parent to allow a same flyweight object to be part of different composites

What we've seen today



- Bridge: Separation of interface from implementation
- Composite: Uniform handling of compound and individual objects
- Decorator: Attaching responsibilities to objects without subclassing
- Facade: A unified interface to a subsystem
- Flyweight: Using flyweight objects for reducing storage costs