Trusted Components
Reuse, Contracts and Patterns

Prof. Dr. Bertrand Meyer
Dr. Karine Arnout
Lecture 22: Decorator, Adapter
Componentizability classification

1. Componentizable
   - 1.1 Built-in
   - 1.2 Library-supported
   - 1.3 Newly componentized
   - 1.4 Possible component
     - 1.3.1 Fully componentizable
       - Prototype
     - 1.3.2 Componentizable but not comprehensive
       - Flyweight
       - Observer
       - Mediator
       - Abstract Factory
       - Factory Method
       - Visitor
       - Command
       - Composite
       - Chain of Responsibility
     - 1.3.3 Componentizable but unfaithful
       - Strategy
     - 1.3.4 Componentizable but useless
       - Memento

2. Non-componentizable
   - 2.1 Skeleton
   - 2.2 Possible skeleton
   - 2.3 Some library support
   - 2.4 Design idea
     - Singleton
     - Iterator
     - Facade
     - Interpreter

Design patterns:
- Flyweight
- Observer
- Mediator
- Abstract Factory
- Factory Method
- Visitor
- Command
- Composite
- Chain of Responsibility
- Builder
- Proxy
- State
- Strategy
- Memento
- Decorator
- Adapter
- Template Method
- Bridge
- Interpreter
Agenda for today

- Decorator
- Class Adapter
- Object Adapter
“Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.” [GoF, p 175]
Decorator: A typical example

- COMPONENT
  - MY_COMPONENT
    - SOME_TYPE
      - additional_attribute
  - DECORATED_COMPONENT
    - DECORATED_COMPONENT_A
    - DECORATED_COMPONENT_B

component

COMPONENT

SOME_TYPE

DECORATED_COMPONENT

DECORATED_COMPONENT_A

DECORATED_COMPONENT_B
Componentization attempts (1/5)

- Genericity
  - Idea: have a class `DECORATED_COMPONENT [G]`
  - Constraint: a `DECORATED_COMPONENT` must be a `COMPONENT`

```plaintext
class
  DECORATED_COMPONENT [G -> COMPONENT]
inherit
  G
...
end
```

Invalid code
Componentization attempts (2/5)

- Using mixins (∼ implementation inheritance) to implement the Decorator would mean:

  ```ruby
  class DECORATED_COMPONENT
    inherit COMPONENT
    DECORATION_MIXIN
  ...
  end
  ``

  e.g.
  ```ruby
  class COLORED_WINDOW
    inherit WINDOW
    COLORABLE
  end
  ```

  May help implementing a Decorator
  Does not help to componentize it
To componentize the pattern, same problems as with inheritance of classes:

- All instances of the class have the decoration
  \[ \Rightarrow \text{Does not allow decoration of a single object} \]

- Composing decorations yields a combinatorial explosion of classes:
  
  classes for each decoration
  + classes for all combinations of these decorations
  (e.g. `BOOK_WITH_BINDING`, `BOOK_WITH_COLOR` but also `BOOK_WITH_COLOR_AND_BINDING`, ...)
Componentization attempts (4/5)

- Automatic type conversion
  - Decoration added to a clone of the original object, not the object itself

- Agents
  - Cannot add an attribute to a given component

- Design by Contract
  - Improves a reusable component; does not make a component reusable

- Aspects
  - Cannot decorate only certain components
Componentization attempts (5/5)

- Automatic type conversion does not help componentizing Decorator because the decoration is not on the original object

- Would assignment attempts help?

  \[ db: \text{DECORATED\_COMPONENT [BOOK]} \]
  \[ b: \text{BOOK} \]
  
  ...  
  
  \[ b \ ?= \ db \]
  
  -- Assignment attempt will always fail because \[ \text{DECORATED\_COMPONENT [BOOK]} \] does not conform to \[ \text{BOOK} \]

  Non-componentizable pattern
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.

**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`

-- 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.

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 \texttt{COMPONENT} and implement them by
-- delegating calls to \texttt{component} as follows:
-- \texttt{do}
-- \texttt{component.feature\_from\_component}
-- \texttt{end}

-- For some of these features, you may want to do something more:
-- \texttt{do}
-- \texttt{component.feature\_from\_component}
-- \texttt{do\_something\_more}
-- \texttt{end}

**feature** \{\texttt{NONE}\} -- Implementation
  \texttt{component: COMPONENT}
  -- Component that will be used for the “decoration”

**invariant**
  \texttt{component\_not\_void: component /= Void}

\texttt{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*)
Decorator: why exporting additional features?

- Newly introduced features do not need to be visible to clients, but they may.
  e.g. Display a window with a border of a certain color

```plaintext
class DECORATED_WINDOW
inherit WINDOW

... feature
  color: COLOR
  set_color (a_color: like color) is ...
  display is do
    draw_border (color)
  end
end
```

Client can change the color by calling `set_color` if they have direct access to the `DECORATED_WINDOW`
Componentizability classification

1. Componentizable
   - 1.1 Built-in
     - Prototype
     - 1.3.1 Fully componentizable
       - Flyweight
       - Observer
       - Mediator
       - Abstract Factory
       - Factory Method
       - Visitor
       - Command
       - Composite
       - Chain of Responsibility
     - 1.3.2 Componentizable but not comprehensive
       - Builder
       - Proxy
       - State
   - 1.2 Library-supported
   - 1.3 Newly componentized
   - 1.4 Possible component

2. Non-componentizable
   - 2.1 Possible skeleton
     - Singleton
   - 2.2 Some library support
   - 2.3 Design idea
     - Facade
     - Interpreter
     - 2.4 No method
     - Decorator
     - Template Method
     - Bridge
     - Adapter

Design pattern

1. Componentizable
2. Non-componentizable

Componentizability classification
Adapter pattern: Intent

“Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.” [GoF, p 139]

2 flavors: Class Adapter and Object Adapter
Agenda for today

- Decorator
- Class Adapter
- Object Adapter
Class Adapter pattern

\[ \text{APPLICATION} \rightarrow \text{TARGET} \rightarrow \text{ADAPTEE} \]

- \( f \) from TARGET and \( g \) from ADAPTEE may have compatible signatures and contracts or not.
class

TARGET

feature -- Basic operation

f (i: INTEGER; s: STRING) is
  -- Do something with i and s.
  require
  s_not_void: s /= Void
do
  ...
end
end
Class **ADAPTEE**

class

**ADAPTEE**

**feature** -- Basic operation

\texttt{g (s: STRING; i: INTEGER) is}

-- Do something with \( s \) and \( i \).

**require**

\texttt{s\_not\_void: s /= Void}

\texttt{s\_not\_empty: not s.is\_empty}

**do**

\ldots

**end**

**end**
Class ADAPTER

class ADAPTER
inherit TARGET
  redefine f end
expanded ADAPTEE
export
  {NONE} all
end
feature -- Basic operation
  f (i: INTEGER; s: STRING) is
    -- Do something with i and s.
    require
      s_not_void: s /= Void
    do
      if not s.is_empty then
        g (s, i)
      end
    end
end
Class *ADAPTER*, particular case

- When *TARGET* and *ADAPTEE* have the same signatures and contracts:

```plaintext
class  
   ADAPTER
inherit  
   TARGET
   undefined  
      f
   end
expanded  ADAPTEE
rename  
      g as f
export  
   {NONE} all
end
end
```
Application using the Class Adapter

class
  APPLICATION
create
  make
feature {NONE}  -- Initialization
    make is
      -- Do something.
      -- (Show typical use of the class adapter pattern.)
      do
        -- Call the version of TARGET.
        do_something (create {TARGET})
        -- Call the version of ADAPTEE.
        do_something (create {ADAPTEE})
      end
feature  -- Basic operations
  do_something (a_target: TARGET) is
    do
      a_target.f
    end
end
Componentization attempts (1/4)

class ADAPTER [$G$, $H$ -> $TARGET$ create default_create end]

create
   make

convert
   to_target: \{H\}

feature -- Conversion
   to_target: $H$ is
      -- Target corresponding to given adaptee
      do
         create Result
      ensure
         target_not_void: Result /= Void
      end
Componentization attempts (2/4)

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

feature -- Access
  impl: PROCEDURE [ANY, TUPLE]
  -- Procedure ready to be called by f

feature -- Basic operations
  f is
    -- Perform an operation.
    do
      impl.call ([])
    end
  invariant
    impl_not_void: impl /= Void
end
Componentization attempts (3/4)

```ruby
class ADAPTER [G, H -> TARGET create default_create end]

- Typical application:

  ```ruby
  an_adapter: ADAPTER [ADAPTEE, TARGET]
  ...
  create an_adapter.make (agent {ADAPTEE}.f)
  do_something (an_adapter)
  -- Call agent corresponding to the version of f from
  -- ADAPTEE. Equivalent to do_something (an_adapter.to_target)
  -- because of automatic type conversion.
  ```

Adapter not really transparent to clients

When we go to the US with a Swiss plug, we need to know it's a Swiss plug to find the right adapter
```
Componentization attempts (4/4)

- Problem with this approach: we would like to write
  
  **books**: `LINKED_LIST [BOOK]`
  
  **an_adapter**: `ADAPTER [TEXTBOOK, BOOK]`
  
  ...  
  
  `create an_adapter.make (  
    agent \{TEXTBOOK\}.borrow_textbook)  
  books.extend (an_adapter)`

Requires `BOOK` to have `default_create` as creation procedure

⇒ Would require a complete refactoring of the book library example
- Constrained genericity, agents, conversion, and aspects do not help (see previous)
  ⇒ Unconstrained genericity would not help more

- Contracts could only improve an existing reusable component

- Inheritance enables implementing the pattern but does not make it componentizable (*TARGET* and *ADAPTEE* are context-dependent)
Class Adapter skeleton

class
   
ADAPTER

inherit
   
TARGET

   undefine
      -- To be completed
   
end

expanded
   
ADAPTEE

rename
   -- To be completed

export
   
{NONE} all

end

create
   -- List creation procedure(s) here.

feature
   -- List features here.

end
Agenda for today

- Decorator
- Class Adapter
- Object Adapter
Object Adapter pattern

APPLICATION \rightarrow TARGET \rightarrow ADAPTEE

\text{APPLICATION} \quad \text{TARGET} \quad \text{ADAPTEE}

\begin{align*}
\text{TARGET} & \quad f \\
\text{ADAPTEE} & \quad g \\
\text{ADAPTER} & \quad f++ \quad \text{adaptee}
\end{align*}
class
  \textit{ADAPTER}
inherit
  \textit{TARGET}
redefine \textit{f} end
create
  \textit{make}
feature \{\textit{NONE}\} \quad \text{Initialization}
  \textit{make} (\textit{an\_adaptee}: \textit{like\ adaptee}) is
    \begin{itemize}
    \item \text{Set} \textit{adaptee} \text{to} \textit{an\_adaptee}.
    \end{itemize}
require
  \begin{itemize}
  \item \textit{an\_adaptee\_not\_void}: \textit{an\_adaptee} = \textit{Void}
  \end{itemize}
do
  \textit{adaptee} := \textit{an\_adaptee}
ensure
  \begin{itemize}
  \item \textit{adaptee\_set}: \textit{adaptee} = \textit{an\_adaptee}
  \end{itemize}
end
Class **ADAPTER (2/2)**

**feature** -- Access

\[ \textit{adaptee}: \textit{ADAPTEE} \]

-- Object to be adapted to \textit{TARGET}

**feature** -- Basic operations

\[ \textbf{f is} \]

-- Do something. (Delegate work to \textit{adaptee}.)

\[ \textit{do} \]

\[ \textit{adaptee}\.g \]

\[ \textit{end} \]

**invariant**

\[ \textit{adaptee\_not\_void}: \textit{adaptee} \neq \textit{Void} \]

\[ \textit{end} \]
class APPLICATION
create make
feature {NONE} -- Initialization
    make is
        -- Do something. (Show a typical use of the object Adapter pattern.)
        local
        an_adaptee: ADAPTEE
        do
            -- Call the version of TARGET.
            do_something (create {TARGET})
            create an_adaptee
                -- Possibly perform some operations on an_adaptee.
                -- Do something using existing object adaptee.
                -- Call the version of ADAPTEE.
                do_something (create {ADAPTER}.make (an_adaptee))
        end
    feature -- Basic operations
        do_something (a_target: TARGET) is
            -- Do something on a_target.
            do
                a_target.f
        end
end

Chair of Software Engineering

Trusted Components: Reuse, Contracts and Patterns - Lecture 22
Componentization attempts (1/6)

class ADAPTER [G -> ADAPTABLE]
inherit TARGET
  redefine f end

... feature -- Access
  adaptee: G

  -- Object to be adapted to TARGET

feature -- Basic operations
  f is
    -- Perform an operation. (Delegate work to adaptee.)
    do
    adaptee.f
    end

... end

• How can we write the classes TARGET and ADAPTABLE?
• Need to conform to ADAPTABLE ⇒ ADAPTEE likely to require changes
Componentization attempts (2/6)

- More appealing scheme:

```eiffel
class ADAPTER [G, H]  -- G: ADAPTEE
                         -- H: TARGET

inherit H

...  
end

But illegal in Eiffel to inherit from a formal generic parameter
```
Componentization attempts (3/6)

- First scheme:
  ```ruby
  class TARGET
    create
      make_from_adaptee
    convert
      make_from_adaptee ({ADAPTEE})
  ...
  end
  ```

- Second scheme:
  ```ruby
  class ADAPTEE
    convert
      to_target: {TARGET}
  ...
  end
  ```

May not have access to the source code of TARGET
Otherwise would not need an adapter…

Will create a new object
Componentization attempts (4/6)

class ADAPTER
inherit TARGET
redefine f end
create
make
feature {NONE} -- Initialization
make (an_impl: like impl)
   -- Set impl to an_impl.
require
   an_impl_not_void: an_impl /= Void
ensure
   impl_set: impl = an_impl

feature -- Access
   impl: PROCEDURE [ANY, TUPLE]
   -- Procedure ready to be called by f

feature -- Basic operations
  f is
     -- Do something.
     do
        impl.call ([])
     end

invariant
   impl_not_void: impl /= Void
end

Replace a call on the ADAPTEE’s routine by a call on the agent given at creation
Componentization attempts (5/6)

- Typical client code would be:
  ```
  create {ADAPTER}.make (agent {ADAPTEE}.f)
  ```

- Works, but changes the goal of the Object Adapter
  - Pattern: Find an adapter to use an adaptee
  - Agents: Give impression we have an adapter and we need to find an adaptee to use it
Componentization attempts (6/6)

```java
aspect Adapter {
    // Apply aspect whenever f of class Target is called.
    pointcut adapterPointcut ():
        call (Target.f)

    // Adaptee providing the new implementation of f declared in Target
    public Adaptee adaptee;

    // New implementation of f declared in Target
    around ():
        adapterPointcut (){
            adaptee.g()
        }
}
```

⇒ Break dynamic dimension of the Object Adapter:
   no object-scope on the aspect, either it is applied to all instances created at run time or none
Object Adapter: Componentization outcome

- Genericity (constrained or not), agents, conversion, and aspects do not help (see previous)

- Contracts could only improve an existing reusable component

- Inheritance does not help because it is static mechanism and object adaptation should happen at run time
Object Adapter skeleton (1/2)

class
  \textit{ADAPTER}
inherit
  \textit{TARGET}
redefine
  -- List all features from \textit{TARGET}
  -- that have a direct counterpart in \textit{ADAPTEE}
end
create
make
feature \{\textit{NONE}\} -- Initialization
make (an\_adaptee: \textit{like} \textit{adaptee}) is
  -- Set \textit{adaptee} to \textit{an\_adaptee}.
require
  an\_adaptee\_not\_void: an\_adaptee /= \textit{Void}
do
  adaptee := an\_adaptee
ensure
  adaptee\_set: adaptee = an\_adaptee
end
Object Adapter skeleton (2/2)

**feature** -- Access

*adaptee: ADAPTEE*

-- Object to be adapted to TARGET

**feature**

-- List all features from TARGET and implement them
-- by calling the version from ADAPTEE if applicable
-- *(adaptee.feature_from_adaptee)*
-- otherwise leave an empty body.

**invariant**

adaptee_not_void: adaptee /= Void

end
Complementary material (1/2)

- **From Patterns to Components:**
  - Chapter 16: Decorator and Adapter
  - Chapter 5: Turning patterns into components: A preview
    - 5.3 A non-componentizable pattern: Decorator

- **Further reading:**
    (Decorator, p 175-184; Adapter, p 139-150)
Further reading:


End of lecture 22