Trusted Components
Reuse, Contracts and Patterns

Prof. Dr. Bertrand Meyer
Dr. Karine Arnout
Lecture 23: Template Method, Bridge
Componentizability classification

1. Componentizable
   - 1.1 Built-in
   - 1.2 Library-supported
   - 1.3 Newly componentized
   - 1.4 Possible component

   1.1.1 Fully componentizable
   - Prototype
   - Flyweight
   - Observer
   - Mediator
   - Abstract Factory
   - Factory Method
   - Visitor
   - Command
   - Composite
   - Chain of Responsibility

   1.1.2 Componentizable but not comprehensive
   - Builder
   - Proxy
   - State

   1.1.3 Componentizable but unfaithful
   - Strategy

   1.1.4 Componentizable but useless
   - Memento

   1.2 Library-supported

   1.3 Newly componentized

   1.4 Possible component

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

   2.1.1 Method
   - Singleton
   - Iterator
   - Facade
   - Interpreter

   2.1.2 No method

   2.2.1 Design idea
   - Decorator
   - Adapter
   - Template Method
   - Bridge
Agenda for today

- Template Method
- Bridge
Template Method pattern

Explains how to “define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure.” [GoF, p 325]
class APPLICATION
feature -- Template method
  frozen do_something is
    -- Do something.
    require
    ready: ready
    do
      do_something_imp_1
      do_something_imp_2
    ensure
    done: done
  end
feature -- Status report
  ready: BOOLEAN is
    -- Are all conditions met for do_something to be called?
    deferred
  end
  done: BOOLEAN is
    -- Has do_something done its job?
    deferred
  end
feature \{NONE\} -- Status report
  first_part_done: BOOLEAN is
    -- Has do_something_imp_1 done its job?
    deferred
    end
feature \{NONE\} -- Implementation ("Hook" features)
  do_something_imp_1 is
    -- Do something.
    require
    ready: ready
    deferred
    ensure
    first_part_done: first_part_done
    end
  do_something_imp_2 is
    -- Do something.
    require
    first_part_done: first_part_done
    deferred
    ensure
    done: done
    end
An important technique of software engineering

“This technique is part of a general approach that we may dub “don’t call us, we’ll call you”: rather than an application system that calls out reusable primitives, a general-purpose scheme lets application developers “plant” their own variants at strategic locations.”

“What the O-O method offers, thanks to behavior classes, is systematic, safe support for this technique, through classes, inheritance, type checking, deferred classes and features, as well as assertions that enable the developer of the fixed part to specify what properties the variable replacements must always satisfy.”

“With the techniques just discussed we are at the heart of the object-oriented method’s contribution to reusability: offering not just frozen components (such as found in subroutine libraries), but flexible solutions that provide the basic schemes and can be adapted to suit the needs of many diverse applications.”

deferred class
  \( \text{LINEAR} \ [G] \)

feature -- Status report
  \( \text{has} \ (v: \text{like item}): \text{BOOLEAN} \ is \)
  -- Does structure include an occurrence of \( v \)?
  
  do
  from
  until
  loop
  end
  Result := not after

\( \text{has is a template method}: \) it is implemented in terms of other features that are deferred (the “hook” features)
Typical example in EiffelBase (2/3)

feature -- Cursor movement

    start is
      -- Move to first position if any; after otherwise.
      deferred
      ensure
        first_position_if_not_empty:
          not is_empty implies index = 1
        after_if_empty: is_empty implies after
      end

    forth is
      -- Move to next position.
      require
        not_after: not after
      deferred
      ensure
        moved: index = old index + 1
      end
feature -- Access

item: G is
    -- Item at current position
    require
        not_off: not off
        deferred
        end

feature -- Status report

after: BOOLEAN is
    -- Is there no valid position to the right of current one?
    deferred
    ensure
        definition: Result = (index = count + 1)
        end

... end
Template Method with agents

**frozen** do\_something is
  -- Do something.
  do
    from
    implementation\_procedures.start
  until
    implementation\_procedures.after
  loop
    implementation\_procedures.item.call ([]) [implementation\_procedures.forth]
  end
end

With:

implementation\_procedures:
  LINKED\_LIST [PROCEDURE [ANY, TUPLE]]

Up to the client to provide the implementation procedures
⇒ Goes against the *Information Hiding principle*

More difficult to ensure that implementation steps are performed in the right order (no contract any more)
“Shows one of the classic forms of reuse in object-oriented programming” [Martin 2002a] but it is not componentizable.

Skeleton classes (Pattern Wizard)
Class APPLICATION (1/2)

defered class APPLICATION

feature -- Template method
  frozen do_something is
    -- Do something.
    require
      ready: ready
    do
      do_something_imp_1
      do_something_imp_2
    ensure
      done: done
  end

feature -- Status report
  ready: BOOLEAN is
    -- Are all conditions met for do_something to be called?
    deferred
  end

done: BOOLEAN is
  -- Has do_something done its job?
  deferred
  end
Class APPLICATION (2/2)

```plaintext
feature {NONE} -- Status report
  first_part_done: BOOLEAN is
    -- Has do_something_imp_1 done its job?
    deferred
end

feature {NONE} -- Implementation ("Hook" features)
  do_something_imp_1 is
    -- Do something.
    require
    ready: ready
    deferred
    ensure
      first_part_done: first_part_done
  end

  do_something_imp_2 is
    -- Do something.
    require
      first_part_done: first_part_done
    deferred
    ensure
      done: done
  end
end
```

Chair of Software Engineering
deferred class
   \texttt{<TEMPLATE\_METHOD\_APPLICATION\_CLASS\_NAME>}
feature -- Template method
   frozen \texttt{<TEMPLATE\_METHOD\_NAME>} is
      -- Do something.
       \textbf{require} \texttt{ready: ready}
       \textbf{do}
       \textbf{<TEMPLATE\_METHOD\_IMPLEMENTATION\_FEATURES\_NAME>}
       \textbf{ensure} \texttt{done: done}
\end

feature -- Status report
   \texttt{ready: BOOLEAN is}
      -- Are all conditions met for feature \texttt{do\_something} to be called?
   \textbf{deferred}
\end

\texttt{done: BOOLEAN is}
   -- Has \texttt{do\_something} done its job?
\textbf{deferred}
\end

feature \{NONE\} -- Status report
   \texttt{<TEMPLATE\_METHOD\_PARTS\_DONE>}

feature \{NONE\} -- Implementation ("Hook" features)
   \texttt{<TEMPLATE\_METHOD\_IMPLEMENTATION\_FEATURES\_CODE>}
\end

Chair of Software Engineering
Componentizability classification

1. Componentizable
   1.1 Built-in
     Prototype
   1.2 Library-supported
   1.3 Newly componentized
     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
   1.3.3 Componentizable but unfaithful
     Strategy
   1.3.4 Componentizable but useless
     Memento

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

Design pattern

1. Componentizable
2. Non-componentizable

Prototype
Builder
Strategy
Singleton
Adapter
Decorator
Bridge
Agenda for today

- Template Method
- Bridge
“Decouple an abstraction from its implementation so that the two can vary independently.” [GoF, p 151]
Deferred class \texttt{APPLICATION}

defered class \texttt{APPLICATION}

\textbf{feature} \{\texttt{NONE}\} -- Initialization

\textit{make} (\textit{an\_implementation}: \texttt{like impl}) \textbf{is}

\hspace{1em} -- Set \textit{impl} to \textit{an\_implementation}.

\hspace{1em} \textbf{require}

\hspace{2em} \textit{an\_implementation\_not\_void}: \textit{an\_implementation} \neq \texttt{Void}

\hspace{1em} \textbf{do}

\hspace{2em} \textit{impl} := \textit{an\_implementation}

\hspace{1em} \textbf{ensure}

\hspace{2em} \textit{impl\_set}: \textit{impl} = \textit{an\_implementation}

\hspace{1em} \textbf{end}

\textbf{feature} -- Basic operation

\textit{do\_something} \textbf{is}

\hspace{1em} -- Do something.

\hspace{1em} \textbf{do}

\hspace{2em} \textit{impl\_do\_something}

\hspace{1em} \textbf{end}

\textbf{feature} \{\texttt{NONE}\} -- Implementation

\textit{impl}: \texttt{APPLICATION\_IMP}

\hspace{1em} -- Implementation

\textbf{invariant}

\hspace{1em} \textit{impl\_not\_void}: \textit{impl} \neq \texttt{Void}

\hspace{1em} \textbf{end}
class CLIENT

create make

feature {NONE} -- Initialization
make is

-- Illustrate how to create and use composite components.

local
application_1: APPLICATION_1
application_2: APPLICATION_2

do
create application_1.make (create {APPLICATION_1_IMP})
application_1.do_something
create application_2.make (create {APPLICATION_2_IMP})
application_2.do_something

doe
Common variation of the Bridge pattern
Bridge with non-conforming inheritance

```
(* APPLICATION *)
  + APPLICATION_1
  + APPLICATION_2

(* APPLICATION_IMP *)
  + APPLICATION_2_IMP
  + APPLICATION_1_IMP
```

- do_something
- do_something++
class APPLICATION
inherit ANY
expanded APPLICATION_IMP
export
  {NONE} all
end
feature -- Basic operation
do_something is
  -- Do something.
do
do_something_imp
end
end
class APPLICATION_1
inherit APPLICATION
undefine do_something_imp
def
expanded APPLICATION_1_IMP
export
{NONE} all
def
end
end
Client of an application using a Bridge with non-conforming inheritance

class CLIENT
create make

feature {NONE} -- Initialization
make is

-- Perform an operation.

local

application_1: APPLICATION_1
application_2: APPLICATION_2

do

create application_1

application_1.do_something

create application_2

application_2.do_something

end

end
# Client vs. inheritance

<table>
<thead>
<tr>
<th>Nº</th>
<th>Criterion</th>
<th>Client</th>
<th>Inheritance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>No permanent binding between abstraction and implementation</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>Abstract and implementation extendible by subclassing</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>3</td>
<td>Implementation changes have no impact on clients</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td>4</td>
<td>Implementation of an abstraction completely hidden from clients</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td>5</td>
<td>Possibility to split numerous classes into two parts: abstraction and implementation</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>6</td>
<td>Implementation share with several objects, hidden from clients</td>
<td>+</td>
<td>++</td>
</tr>
</tbody>
</table>
Bridge pattern vs. Bridge with non-conforming inheritance

- 2 different application needs:
  - Bridge pattern: enables selecting an implementation at run time
    - e.g. Gobo XML has two parsers (pure Eiffel and wrapper of C expat) that can be selected dynamically
  - Bridge with non-conforming inheritance: enables selecting an implementation at compile time
    - e.g. GUI application must select the widgets (Windows or Unix) to be used at compile time
class

APPLICATION [G -> APPLICATION_IMP]

feature -- Basic operation
f is
-- Delegate call to impl.
do
implementation.f
end

implementation: G
-- Implementation

do
Non-componentizable
end
Skeleton classes (Pattern Wizard)

• How can we specify a reusable APPLICATION_IMP class?
• Clients should not know that APPLICATION uses a Bridge
Complementary material (1/2)

- From Patterns to Components:
  - Chapter 17: Template Method and Bridge

- Further reading:
    - 14.8 The role of deferred classes, p 500-506.
Further reading:


End of lecture 23