Software Architecture

Chair of Software Engineering
Lecture 9: More about patterns
Agenda for today

- Patterns
- Abstract Factory Pattern
- Visitor
- Observer
- Chain of responsibility
- Command
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.
Benefits of software design patterns:

- Capture the knowledge of experienced developers
- Newcomers can learn them and apply them to their design
- Yield a better structure of the software
- Facilitate discussions between programmers and managers
Abstract factory pattern
Creational patterns

- Hide the creation process of objects
- Hide the concrete type of these objects
- Allow dynamic and static configuration of the software system
Motivation

Widget toolkit

- Different look & feels of widgets, e.g. widgets for unix systems and widgets for windows
- Family of widgets: Scroll bars, buttons, dialogs...
- Want to allow changing look & feel

→ Most part of the system needs not know what look & feel is used
→ Creation of widget object should not be distributed

Solution: abstract factory pattern
Abstract factory - Intent

“Provide[s] an interface for creating families of related or dependent objects without specifying their concrete classes.” [GoF, p 87]
Architecture for widget example

* FACTORY
  + WIN_FACTORY
  + UNX_FACTORY

* BUTTON
  + UNX_BUTTON
  + WIN_BUTTON

* CHECKBOX
  + UNX_CHECKBOX
  + WIN_CHECKBOX

new_button* -> * FACTORY
new_button* -> * BUTTON
new_button* -> + UNX_BUTTON
new_button* -> + WIN_BUTTON
new_check_box* -> * CHECKBOX
new_check_box* -> + UNX_CHECKBOX
new_check_box* -> + WIN_CHECKBOX
Architecture of a general example

![Diagram of software architecture]

- **PRODUCT_A**
  - **PRODUCT_A1**
    - **FACTORY**
      - **FACTORY_1**
        - **PRODUCT_B**
          - **PRODUCT_B1**
          - **PRODUCT_B2**
        - **FACTORY_2**
          - **PRODUCT_B**
            - **PRODUCT_B1**
            - **PRODUCT_B2**
          - **PRODUCT_A**
            - **PRODUCT_A1**
            - **PRODUCT_A2**
          - **PRODUCT_A**
            - **PRODUCT_A1**
            - **PRODUCT_A2**

- **PRODUCT_A**
  - **PRODUCT_A1**
    - **FACTORY**
      - **FACTORY_1**
        - **PRODUCT_B**
          - **PRODUCT_B1**
          - **PRODUCT_B2**
        - **FACTORY_2**
          - **PRODUCT_A**
            - **PRODUCT_A1**
            - **PRODUCT_A2**
          - **PRODUCT_A**
            - **PRODUCT_A1**
            - **PRODUCT_A2**

- **PRODUCT_A**
  - **PRODUCT_A1**
    - **FACTORY**
      - **FACTORY_1**
        - **PRODUCT_B**
          - **PRODUCT_B1**
          - **PRODUCT_B2**
        - **FACTORY_2**
          - **PRODUCT_A**
            - **PRODUCT_A1**
            - **PRODUCT_A2**
          - **PRODUCT_A**
            - **PRODUCT_A1**
            - **PRODUCT_A2**

- **PRODUCT_A**
  - **PRODUCT_A1**
    - **FACTORY**
      - **FACTORY_1**
        - **PRODUCT_B**
          - **PRODUCT_B1**
          - **PRODUCT_B2**
        - **FACTORY_2**
          - **PRODUCT_A**
            - **PRODUCT_A1**
            - **PRODUCT_A2**
          - **PRODUCT_A**
            - **PRODUCT_A1**
            - **PRODUCT_A2**

- **PRODUCT_A**
  - **PRODUCT_A1**
    - **FACTORY**
      - **FACTORY_1**
        - **PRODUCT_B**
          - **PRODUCT_B1**
          - **PRODUCT_B2**
        - **FACTORY_2**
          - **PRODUCT_A**
            - **PRODUCT_A1**
            - **PRODUCT_A2**
          - **PRODUCT_A**
            - **PRODUCT_A1**
            - **PRODUCT_A2**

- **PRODUCT_A**
  - **PRODUCT_A1**
    - **FACTORY**
      - **FACTORY_1**
        - **PRODUCT_B**
          - **PRODUCT_B1**
          - **PRODUCT_B2**
        - **FACTORY_2**
          - **PRODUCT_A**
            - **PRODUCT_A1**
            - **PRODUCT_A2**
          - **PRODUCT_A**
            - **PRODUCT_A1**
            - **PRODUCT_A2**

- **PRODUCT_A**
  - **PRODUCT_A1**
    - **FACTORY**
      - **FACTORY_1**
        - **PRODUCT_B**
          - **PRODUCT_B1**
          - **PRODUCT_B2**
        - **FACTORY_2**
          - **PRODUCT_A**
            - **PRODUCT_A1**
            - **PRODUCT_A2**
          - **PRODUCT_A**
            - **PRODUCT_A1**
            - **PRODUCT_A2**

- **PRODUCT_A**
  - **PRODUCT_A1**
    - **FACTORY**
      - **FACTORY_1**
        - **PRODUCT_B**
          - **PRODUCT_B1**
          - **PRODUCT_B2**
        - **FACTORY_2**
          - **PRODUCT_A**
            - **PRODUCT_A1**
            - **PRODUCT_A2**
          - **PRODUCT_A**
            - **PRODUCT_A1**
            - **PRODUCT_A2**

- **PRODUCT_A**
  - **PRODUCT_A1**
    - **FACTORY**
      - **FACTORY_1**
        - **PRODUCT_B**
          - **PRODUCT_B1**
          - **PRODUCT_B2**
        - **FACTORY_2**
          - **PRODUCT_A**
            - **PRODUCT_A1**
            - **PRODUCT_A2**
          - **PRODUCT_A**
            - **PRODUCT_A1**
            - **PRODUCT_A2**

- **PRODUCT_A**
  - **PRODUCT_A1**
    - **FACTORY**
      - **FACTORY_1**
        - **PRODUCT_B**
          - **PRODUCT_B1**
          - **PRODUCT_B2**
        - **FACTORY_2**
          - **PRODUCT_A**
            - **PRODUCT_A1**
            - **PRODUCT_A2**
          - **PRODUCT_A**
            - **PRODUCT_A1**
            - **PRODUCT_A2**

- **PRODUCT_A**
  - **PRODUCT_A1**
    - **FACTORY**
      - **FACTORY_1**
        - **PRODUCT_B**
          - **PRODUCT_B1**
          - **PRODUCT_B2**
        - **FACTORY_2**
          - **PRODUCT_A**
            - **PRODUCT_A1**
            - **PRODUCT_A2**
          - **PRODUCT_A**
            - **PRODUCT_A1**
            - **PRODUCT_A2**

- **PRODUCT_A**
  - **PRODUCT_A1**
    - **FACTORY**
      - **FACTORY_1**
        - **PRODUCT_B**
          - **PRODUCT_B1**
          - **PRODUCT_B2**
        - **FACTORY_2**
          - **PRODUCT_A**
            - **PRODUCT_A1**
            - **PRODUCT_A2**
          - **PRODUCT_A**
            - **PRODUCT_A1**
            - **PRODUCT_A2**

- **PRODUCT_A**
  - **PRODUCT_A1**
    - **FACTORY**
      - **FACTORY_1**
        - **PRODUCT_B**
          - **PRODUCT_B1**
          - **PRODUCT_B2**
        - **FACTORY_2**
          - **PRODUCT_A**
            - **PRODUCT_A1**
            - **PRODUCT_A2**
          - **PRODUCT_A**
            - **PRODUCT_A1**
            - **PRODUCT_A2**

- **PRODUCT_A**
  - **PRODUCT_A1**
    - **FACTORY**
      - **FACTORY_1**
        - **PRODUCT_B**
          - **PRODUCT_B1**
          - **PRODUCT_B2**
        - **FACTORY_2**
          - **PRODUCT_A**
            - **PRODUCT_A1**
            - **PRODUCT_A2**
          - **PRODUCT_A**
            - **PRODUCT_A1**
            - **PRODUCT_A2**

- **PRODUCT_A**
  - **PRODUCT_A1**
    - **FACTORY**
      - **FACTORY_1**
        - **PRODUCT_B**
          - **PRODUCT_B1**
          - **PRODUCT_B2**
        - **FACTORY_2**
          - **PRODUCT_A**
            - **PRODUCT_A1**
            - **PRODUCT_A2**
          - **PRODUCT_A**
            - **PRODUCT_A1**
            - **PRODUCT_A2**

- **PRODUCT_A**
  - **PRODUCT_A1**
    - **FACTORY**
      - **FACTORY_1**
        - **PRODUCT_B**
          - **PRODUCT_B1**
          - **PRODUCT_B2**
        - **FACTORY_2**
          - **PRODUCT_A**
            - **PRODUCT_A1**
            - **PRODUCT_A2**
          - **PRODUCT_A**
            - **PRODUCT_A1**
            - **PRODUCT_A2**

Sketch of class FACTORY

defferred class FACTORY

feature -- Basic operations

  new_button: BUTTON is
    -- New button
    deferred
  end

  new_checkbox: CHECKBOX is
    -- New checkbox
    deferred
  end

... end
class
    WIN_FACTORY
inherit
    FACTORY

feature  -- Basic operations
    new_button: BUTTON is
        -- New windows button
        do
            Result := create \{WIN_BUTTON\}
        end
    new_checkbox: CHECKBOX is
        -- New windows checkbox
        do
            Result := create \{WIN_CHECKBOX\}
        end
    ...
end
class \textit{SHARED\_FACTORY} 
...
\textbf{feature} -- Basic operations

\textit{factory: FACTORY is} 

\hspace{1cm} -- Factory used for widget instantiation

\hspace{1cm} \textbf{once}

\hspace{1cm} \textbf{if} \ \textit{is\_windows\_os} \ \textbf{then}

\hspace{2cm} \textbf{Result} := \textbf{create} \ \{\textit{WIN\_FACTORY}\}

\hspace{1cm} \textbf{else}

\hspace{2cm} \textbf{Result} := \textbf{create} \ \{\textit{UNX\_FACTORY}\}

\hspace{1cm} \textbf{end}

\hspace{1cm} \textbf{end}

...

\hspace{1cm} \textbf{end}
class
    WIDGET_APPLICATION
inherit
    SHARED_FACTORY
...
feature -- Basic operations

    a_feature is
        -- Generate a new button and use it.
        local
            my_button: BUTTON
        do
            ...
            my_button := factory.new_button
            ...
        end
    ...
end
Use the abstract factory pattern if...

- a system should be independent of how its objects are created, represented and collaborating
- a system needs to be configured with one of multiple families
- a family of objects is to be designed and only used together
- you want to support a whole palette of products, but only want to show the public interface
Consequences

- Isolates concrete classes
- Makes exchanging product families easy
- Promotes consistency among products
- Supporting new kinds of products is difficult
Visitor
Visitor - Example

- Set of classes to deal with XML documents
  - `XML_NODE`
  - `XML_DOCUMENT`
  - `XML_ELEMENT`
  - `XML_ATTRIBUTE`
  - `XML_CONTENT`
- One parser
- Many formatters
  - Normalize (Pretty Print)
  - Compressed
  - Convert to different encoding
  - ...
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.

[GoF, p 331]
Visitor - Motivation

- Static class hierarchy
- Many operations to be performed on it

- Another Example:
  - AST of program
    - Nodes: Class, Feature, instruction, ...
  - Operations:
    - Compile
    - Pretty print
    - Generate documentation
    - Refactor
On books and VCRs

- We want to add external functionality
- Example
  - Maintenance
  - Visualization
**Maintenance**

`maintain (an_item: BORROWABLE) is`  
-- Maintain `an_item`.

**require**

`an_item_not_void: an_item /= Void`

**local**

`book: BOOK`
`vcr: VIDEO_RECORDER`

**do**

`book == an_item`
`if book /= Void then`
-- Do book maintenance.
`end`

`vcr == an_item`
`if vcr /= Void then`
-- Do VCR maintenance.
`end`

`end`
Visualization

\[
display \ (an\_item: \ BORROWABLE) \ is \\
\quad \quad \quad -- \ Maintain \ an\_item.
\]
\[
\text{require} \\
an\_item\_not\_void: \ an\_item \ /= \ Void
\]
\[
\text{local} \\
\quad \text{book: BOOK} \\
\quad \vcr: \ VIDEO\_RECORDER
\]
\[
do \\
\quad \text{book} \ ?= \ an\_item \\
\quad \text{if} \ \text{book} \ /= \ Void \ \text{then} \\
\quad \quad \quad -- \ Display \ book.
\quad \text{end} \\
\quad \vcr \ ?= \ an\_item \\
\quad \text{if} \ \vcr \ /= \ Void \ \text{then} \\
\quad \quad \quad -- \ Display \ VCR.
\quad \text{end}
\]
\[
\text{end}
\]

- Why is this approach bad?
Visitor pattern: a typical example

- BORROWABLE
  - BOOK
    - accept+
  - VIDEO_RECORDER
    - accept+

- VISITOR
  - accept*
  - visit_book*
  - visit_video_recorder*

- MAINTENANCE_VISITOR
  - visit_book+
  - visit_video_recorder+

- DISPLAY_VISITOR
  - visit_book+
  - visit_video_recorder+
Class MAINTENANCE_VISITOR

class MAINTENANCE_VISITOR
inherit VISITOR

feature -- Basic operations
  visit_book (a_book: BOOK) is
    -- Maintain a_book.
    do
      a_book.check_binding
      if a_book.damaged then
        a_book.repair
      end
    end
  end

  visit_video_recorder (a_recorder: VIDEO_RECORDER) is
    -- Maintain a_recorder.
    do
      a_recorder.check_reading_heads
      if a_recorder.damaged then
        a_recorder.send_to_reparation
      end
    end
  end
Class **BOOK**

class
    **BOOK**
inherit
    **BORROWABLE**
feature -- Visitor pattern
    accept (a_visitor: VISITOR) is
        -- Accept visitor a_visitor and call the specialized
        -- visit_* feature applicable to books.
        do
            a_visitor.visit_book (Current)
        end
    end
end
class VIDEO_RECORDER

inherit BORROWABLE

feature -- Visitor pattern

accept (a_visitor: VISITOR) is

-- Accept visitor a_visitor and call the specialized
-- visit_* feature applicable to video recorders.

do

  a_visitor.visit_video_recorder (Current)

end

end
Visitor - Usage

local

  item: BORROWABLE
  maintainer: MAINTAINANCE_VISITOR

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

Calls VISITOR.visit_book

Calls VISITOR.visit_video_recorder
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.
Visitor - Consequences

- Makes adding new operations easy
- Gathers related operations, separates unrelated ones
- Avoids assignment attempts
  - Better type checking
- Adding new concrete element is hard
Observer
Observer

Observers

Subject

A = 50%
B = 30%
C = 20%
“Define[s] a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.” [GoF, p 293]
class
  LIBRARY
inherit
  SUBJECT
  redefine
    default_create
end
feature \{NONE\} -- Initialization
  default_create is
    -- Create and initialize the library with an empty
    -- list of books.
    do
      Precursor \{SUBJECT\}
      create books.make
    end
feature -- Access
books: LINKED_LIST[BOOKS]
   -- Books currently in the library

feature -- Element change
add_book (a_book: BOOK) is
   -- Add a_book to the list of books and notify all library observers.
   require
   a_book_not_void: a_book /= Void
   not_yet_in_library: not books.has (a_book)
   do
   books.extend (a_book)
   notify_observers
   ensure
   one_more: books.count = old books.count + 1
   book_added: books.last = a_book
   end

...
class APPLICATION
inherit OBSERVER
rename
    update as display_book
redefine
default_create
end

feature {NONE} -- Initialization
default_create is
    -- Initialize library and subscribe current application as
    -- library observer.
    do
        create library
        library.add_observer (Current)
    end

...
feature -- Observer pattern

library: LIBRARY
   -- Subject to observe

display_book is
   -- Display title of last book added to library.

   do
      print (library.books.last.title)
   end

invariant

library_not_void: library /= Void
consistent: library.observers.has (Current)
end
Drawbacks of the Observer

- The subject knows its observers

- No information passing from subject to observer when an event occurs

- An observer can register to at most one subject

  - Could pass the *SUBJECT* as argument to *update* but would yield many assignment attempts to distinguish between the different *SUBJECT*s.
A refresher on agents

- objects representing potential computations

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

\textit{my\_integrator}\.\textit{integral} \ (\texttt{agent my\_function, a, b})
Normal call vs. agent call

- Normal call
  \[ a_0.f(a_1, a_2, a_3) \]

- Agent call (expression): preface it by keyword `agent`, yielding
  \[ \text{agent } a_0.f(a_1, a_2, a_3) \]

- For example:
  \[ u := \text{agent } a_0.f(a_1, a_2, a_3) \]

- This represents the routine, ready to be called. To call it:
  \[ u.call([]) \]
  -- For type of \text{\textit{u}}, see next

- Recall original name of agents: “delayed calls”.
Agent types: Kernel library classes

- ROUTINE
  - BASE, ARGS -> TUPLE

- PROCEDURE
  - BASE, ARGS -> TUPLE

- FUNCTION
  - BASE, ARGS -> TUPLE, RES

Inherits from
- Deferred
Creating vs. calling an agent

- Writing:
  
  \[
  \text{agent my\_feature}
  \]

  creates an agent, i.e. an object of type \textit{ROUTEINE}.

- To call an agent, one needs to execute \textit{call} (with the proper arguments) to this \textit{ROUTEINE} object, e.g.:

  \[
  \text{my\_routine\_call ([args])}
  \]
Keeping arguments open

- An agent can have both “closed” and “open” arguments.

- Closed arguments set at time of agent definition; open arguments set at time of each call.

- To keep an argument open, just replace it by a question mark:

  \[
  u := \text{agent } a0.f (a1, a2, a3) \\
  \quad \quad \quad \text{-- All closed (as before)}
  \]

  \[
  w := \text{agent } a0.f (a1, a2, ?)
  \]

  \[
  x := \text{agent } a0.f (a1, ?, a3)
  \]

  \[
  y := \text{agent } a0.f (a1, ?, ?)
  \]

  \[
  z := \text{agent } a0.f (?, ?, ?)
  \]
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
        mouse_click_not_void: 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)
```
Publisher, subscriber, subscribed object (2/2)

Subscriber (APPLICATION) subscribes objects to events.

Subscribed objects:
- Subject
- Publisher

Observer
class
  LIBRARY
...

feature -- Access

  books: LINKED_LIST [BOOK]
    -- Books in library

feature -- Event type

  book_event: EVENT_TYPE [TUPLE [BOOK]]
    -- Event associated with attribute books
feature -- Element change
  \textit{add\_book} (\textit{a\_book}: \textit{BOOK}) is
  -- Add \textit{a\_book} to the list of books and
  -- publish \textit{book\_event}.

\begin{verbatim}
require
  \textit{a\_book\_not\_void}: \textit{a\_book} \neq \textit{Void}
  \textit{not\_yet\_in\_library}: \textit{not books\.has} (\textit{a\_book})
\end{verbatim}

\begin{verbatim}
\textbf{do}
  \textit{books}\.\textbf{extend} (\textit{a\_book})
  \textit{book\_event}\.\textbf{publish} ([\textit{a\_book}])
\end{verbatim}

\begin{verbatim}
\textbf{ensure}
  \textit{one\_more}: \textit{books\.count} = \textit{old books\.count} + 1
  \textit{book\_added}: \textit{books\.last} = \textit{a\_book}
\end{verbatim}

\begin{verbatim}
\textbf{end}
\end{verbatim}

\begin{verbatim}
\textbf{invariant}
  \textit{books\_not\_void}: \textit{books} \neq \textit{Void}
  \textit{book\_event\_not\_void}: \textit{book\_event} \neq \textit{Void}
\end{verbatim}

\begin{verbatim}
\textbf{end}
\end{verbatim}
Observer pattern vs. Event Library

- In case of an existing class `MY_CLASS`:
  - With the Observer pattern:
    - Need to write a descendant of `OBSERVER` and `MY_CLASS`
    - ⇒ Useless multiplication of classes
  - With the Event Library:
    - Can reuse the existing routines directly as agents
Chain of responsibility, Command
Behavioral design patterns (1/2)

- 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
Behavioral design patterns (2/2)

- **Deal with:**
  - Algorithms
  - Assignment of responsibilities between objects
  - Communication between objects

- **How:**
  - Through inheritance or composition
Chain of Responsibility pattern
“Avoid[s] coupling the sender of a request to its receiver by giving more than one object a change to handle the request. [It] chain[s] the receiving objects and pass[es] the request along the chain until an object handles it.” [GoF, p 223]
deferred class
  HANDLER
...

feature -- Basic operation
  handle is
     
     -- Handle request if can_handle otherwise forward it to next.
     -- If next is void, set handled to False.
     
     do
     if can_handle then
       do_handle
       handled := True
     else
       if next /= Void then
         next.handle
         handled := next.handled
       else
         handled := False
       end
     end

   ensure
     can_handle implies handled
     (not can_handle and then next /= Void) implies handled = next.handled
     (not can_handle and then next = Void) implies not handled
   end
end

Chair of Software Engineering
Chain of Responsibility

**APPLICATION** -> **HANDLER** -> **INTERMEDIATE_HANDLER** -> **FINAL_HANDLER**

- **APPLICATION**:
  - can_handle+
  - do_handle+

- **HANDLER**:
  - can_handle*
  - do_handle*
  - handled
  - set_next

- **INTERMEDIATE_HANDLER**:
  - can_handle+
  - do_handle+

- **FINAL_HANDLER**:
  - can_handle+
  - do_handle+
deferred class
    HANDLER [G]
feature {NONE} -- Initialization
    make (a_successor: like next) is
        -- Set next to a_successor.
        do
            next := a_successor
        ensure
            next_set: next = a_successor
        end
feature -- Access
    next: HANDLER [G]
        -- Successor in the chain of responsibility
feature -- Status report
    can_handle (a_request: G): BOOLEAN is deferred end
        -- Can current handle a_request?
    handled: BOOLEAN
        -- Has request been handled?
feature -- Basic operation
handle (a_request: G) is
  -- Handle a_request if can_handle otherwise forward it to next.
  -- If next is void, set handled to False.

  do
    if can_handle (a_request) then
      do_handle (a_request)
      handled := True
    else
      if next /= Void then
        next.handle (a_request)
        handled := next.handled
      else
        handled := False
      end
    end
  end

ensure
  can_handle (a_request) implies handled
  (not can_handle (a_request) and then next /= Void) implies handled = next.handled
  (not can_handle (a_request) and then next = Void) implies not handled
end
Class HANDLER [G] (3/3)

feature -- Element change
    set_next (a_successor: like next) is
        -- Set next to a_successor.
        do
            next := a_successor
        ensure
            next_set: next = a_successor
        end

feature {NONE} -- Implementation
    do_handle (a_request: G) is
        -- Handle a_request.
        require
            can_handle: can_handle (a_request)
        deferred
        end
end
deferred class HANDLER [G]

... feature -- Basic operation
handle (a_request: G) is
  do
    if can_handle (a_request) then
      do_handle (a_request)
      handled := True
    else
      if next /= Void then
        next.handle (a_request)
        handled := next.handled
      else
        handled := False
      end
    end
  end
ensure
  ...
end

require -- ???
  not handled

Would mean that a HANDLER that has handled a request cannot handle any other request; one would need to create another HANDLER object
⇒ Not very useful

Chair of Software Engineering

Software Architecture - Lecture 10
Command pattern
Command pattern: Intent

- Way to implement an undo-redo mechanism, e.g. in text editors. [OOSC, p 285-290]

- “Way to encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.” [GoF, p 233]
Command pattern (history-executable)

- APPLICATION
- HISTORY
- COMMAND

**APPLICATION**
- `history`

**HISTORY**
- `commands`
- `execute`
- `can_undo`, `can_redo`
- `undo`, `redo`
- `undo_all`, `redo_all`
- `commands`, `arguments`
- `extend`

**COMMAND**
- `is_once_command`
- `execute*`
- `undo*`
- `redo*`

**COMMAND_1**
- `execute+`
- `undo+`
- `redo+`

**COMMAND_2**
- `execute+`
- `undo+`
- `redo+`
How to use the Command pattern

- Create a descendant of `COMMAND` and effect its features `execute`, `undo`, and `redo`

```plaintext
class
    COMMAND_1
inherit
    COMMAND
create
    make
feature
    {HISTORY} -- Command pattern
    execute (args: TUPLE) is do ... end
    -- Execute command with args.
feature
    {HISTORY} -- Undo
    undo (args: TUPLE) is do ... end
    -- Undo last action.
feature
    {HISTORY} -- Redo
    redo (args: TUPLE) is do ... end
    -- Redo last undone action.
end
```

To be completed
class APPLICATION
create make feature {NONE} -- Initialization
make is
--- Create a command and execute it.
--- (Use the undo/redo mechanism.)

local
command_1: COMMAND_1
command_2: COMMAND_2

do create command_1.make (True)
create command_2.make (False)
history.execute (command_1, [])
history.execute (command_2, [])
history.undo
history.redo

end
Example using the Command pattern (2/2)

```
feature {NONE} -- Implementation

history: HISTORY is
  -- History of executed commands
  once
  create Result.make
  ensure
    history_not_void: Result /= Void
  end
end
```
Command pattern (auto-executable)

- **APPLICATION**
- **HISTORY**
- **COMMAND**
  - `execute*`
  - `undo*`
  - `redo*`
  - `is_once_command`

**SHARED_HISTORY**
- `history`
- `has`
  - `execute`
  - `can_undo`, `can_redo`
  - `undo`, `redo`
  - `undo_all`, `redo_all`
  - `commands`
  - `arguments`
  - `extend`
**Command: class** **SHARE** _**D_HISTORY**_

**Common scheme in Eiffel:** Inherit from a class containing the data to be shared among different objects.

**Not compulsory:** `COMMAND` could have an attribute `history` initialized at creation and one would always pass the same `HISTORY` object as argument; hence sharing.

**Advantage:** enables having several histories; e.g. keep 2 histories of commands corresponding to 2 editor windows.)
End