Trusted Components

Reuse, Contracts and Patterns

Prof. Dr. Bertrand Meyer
Dr. Karine Arnout
Lecture 19: Command, Chain of Responsibility
Agenda for today

- Command pattern
- Command Library
- Chain of Responsibility pattern
- Chain of Responsibility Library
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
  - commands

- **HISTORY**
  - 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
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**

**COMMAND_1**

**COMMAND_2**

**SHARED HISTORY**

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

- *****
- **is_once_command**
- **execute***
- **undo***
- **redo***

- **+**
- **execute+**
- **undo+**
- **redo+**

- **+**
- **execute+**
- **undo+**
- **redo+**
**Command: class** *SHARE\_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.)
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
      - 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 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
   - Decorator
   - Adapter
   - Template Method
   - Bridge
Agenda for today

- Command pattern
- Command Library
- Chain of Responsibility pattern
- Chain of Responsibility Library
Command Library (history-executable)

- COMPONENT
  - parent
  - item

- COMPOSITE
  - do_something
  - undo_something

- APPLICATION
  - history
    - execute
    - can_undo, can_redo
    - undo, redo
    - undo_all, redo_all
    - commands, arguments
    - extend

- HISTORY

- COMMAND
  - action
    - undo_action
    - set_undo_action
    - is_once_command
    - execute
    - execute_with_args
    - undo, redo
class COMMAND
inherit COMPONENT [COMMAND]

rename

  do_something as execute
redefine

  execute

end

create

make, make_with_undo

feature {NONE} -- Initialization

make (an_action: like action; a_value: like is_once_command) is

  -- Set action to an_action and is_once_command to a_value.

  require

    an_action_not_void: an_action /= Void

  do

    action := an_action

    is_once_command := a_value

  ensure

    action_set: action = an_action

    is_once_command_set: is_once_command = a_value

end
make_with_undo (an_action: like action;
an_undo_action: like undo_action;
a_value: like is_once_command) is
-- Set action to an_action, undo_action to an_undo_action.
-- Set is_once_command to a_value.

require
an_action_not_void: an_action /= Void
an_undo_action_not_void: an_undo_action /= Void

do
action := an_action
undo_action := an_undo_action
is_once_command := a_value

ensure
action_set: action = an_action
undo_action_set: undo_action = an_undo_action
is_once_command_set: is_once_command = a_value

end
Class COMMAND (3/6)

feature -- Access

action: PROCEDURE [ANY, TUPLE]
  -- Action to be executed

undo_action: PROCEDURE [ANY, TUPLE]
  -- Action to be executed to undo the effects of
  -- calling action

feature -- Status report

is_once_command: BOOLEAN
  -- Can this command be executed only once?

valid_args (args: TUPLE): BOOLEAN is
  -- Are args valid arguments for
  -- execute_with_args and redo?

do
  Result := action.valid_operands ([args])
end
Class **COMMAND (4/6)**

**feature** -- Status setting

setUndoAction (an_action: like undo_action) is

-- Set undo_action to an_action.

**require**

an_action_not_void: an_action /= Void

do

undo_action := an_action

**ensure**

undo_action_set: undo_action = an_action

end

**feature** {HISTORY} -- Command pattern

execute is

-- Call action with an empty tuple as arguments.

do

if action.valid_operands ([]) then
    action.call ([]) 
end

end
execute_with_args (args: TUPLE) is
   -- Call action with args.
   require
      args_not_void: args /= Void
      valid_args: valid_args ([args])
   do
      action.call ([args])
   end

feature {HISTORY} -- Undo
undo (args: TUPLE) is
   -- Undo last action. (Call undo_action with args.)
   require
      undo_action_not_void: undo_action /= Void
      args_not_void: args /= Void
      valid_args: undo_action.valid_operands ([args])
   do
      undo_action.call ([args])
   end
feature \{HISTORY\} -- Redo

redo (args: TUPLE) is
    -- Redo last undone action.
    -- (Call action with args.)

require
    args_not_void: args /\= Void
    valid_args: valid_args ([args])

do
    action.call ([args])
end

invariant
    action_not_void: action /\= Void
end
How to use the Command Library

- Creation of a command:
  ```python
create a_command.make_with_undo (  
    agent do_something,  
    agent undo_something,  
    True  
  )
```

- Execution of a command:
  ```python
history.execute (  
  a_command,  
  [some_arguments_for_feature_do_something]  
)
```
class

SHARED_HISTORY

feature \{NONE\} -- Implementation

history: HISTORY is

-- History of executed commands

once

create Result.make

ensure

history_not_void: Result /= Void

end

end
class COMMAND
...
feature -- Command pattern

do
execute is

-- Call action with an empty tuple as arguments.

if action.valid_operands ([]) then
  if is_once_command and then
    if action.valid_operands ([]) then
      history.has (Current) then
        history.extend (clone (Current), [])
      else
        history.extend (Current, [])
      end
    end
  else
    history.extend (Current, [])
  end
action.call ([])
execute_with_args (args: TUPLE) is
  -- Call action with args.
  require
    args_not_void: args /= Void
    valid_args: valid_args ([args])
  do
    if is_once_command and then
      history.has (Current) then
        history.extend (clone (Current), args)
      else
        history.extend (Current, args)
    end
    action.call ([args])
  end
...
How to use the Command Library

- Creation of a command:
  ```python
  create a_command.make_with_undo (agent do_something,
  agent undo_something,
  True
  )
  ```

- Execution of a command:
  ```python
  a_command.execute
  a_command.execute_with_args ( ["Command"])
  ```

- (Instead of:
  ```python
  history.execute (a_command,
  [some_arguments_for_feature_do_something]
  )
  ```
  in the history-executable version)
If existing class `THIRD_PARTY_COMMAND` with a feature `execute_command`:

- With the Command pattern:
  - Need to create a new class `MY_COMMAND` inheriting from `THIRD_PARTY_COMMAND` and `COMMAND`.

- With the Command Library:
  - Can use the existing code directly:
    
    ```agent
    execute_command
    ```
Combining the two approaches

CLIENT_1 \rightarrow MY_COMMAND

CLIENT_2 \rightarrow AGENT_COMMAND

COMMAND

execute* undo* redo*

* action
undo_action
set_undo_action
is_once_command
execute_agent
execute_agent_with_args
undo_agent
redo_agent
Command: Componentization outcome

- Completeness
  - All cases of the Command pattern

- Usefulness
  - Reusable
  - Easy-to-use

- Faithfulness
  - Different from a traditional implementation of Command (agents instead of inheritance)

- Type-safety
  - Type-safe (agents, Composite Library)

- Performance
  - Same order as the Command pattern

- Extended applicability
  - No more cases
Agenda for today

- Command pattern
- Command Library
- Chain of Responsibility pattern
- Chain of Responsibility Library
“Avoid[s] coupling the sender of a request to its receiver by giving more than one object a chance 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]
Class HANDLER

defered 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.handle
                else
                    handled := False
                end
            end
        end
    ensure
        can_handle implies handled
        not can_handle and then next /= Void) implies handled = next.handle
        not can_handle and then next = Void) implies not handled
    end

...
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
       - Flyweight
       - Observer
       - Mediator
       - Abstract Factory
       - Factory Method
       - Visitor
       - Command
       - Composite
     - 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 Skeleton
   - 2.2 Possible skeleton
   - 2.3 Some library support
   - 2.4 Design idea
     - Singleton
     - Iterator
     - Facade
     - Interpreter
     - Decorator
     - Adapter
     - Template Method
     - Bridge
     - Chain of Responsibility

Design pattern
Agenda for today

- Command pattern
- Command Library
- Chain of Responsibility pattern
- Chain of Responsibility Library
Chain of Responsibility Library

Mechanisms enabling componentization: unconstrained genericity, contracts

- APPLICATION
- HANDLER
- INTERMEDIATE_HANDLER
- FINAL_HANDLER

Method signatures:
- \texttt{can\_handle} +
- \texttt{do\_handle} +
- \texttt{handled}
- \texttt{set\_next}
- \texttt{handle}
- \texttt{can\_handle}*  
- \texttt{do\_handle}*  
- \texttt{handled}
- \texttt{set\_next}

\texttt{next}
deferred class
  \texttt{HANDLER} [G]
feature \{NONE\} -- Initialization
  \textit{make} (a\_successor: \texttt{like next}) is
  -- Set \textit{next} to \textit{a\_successor}.
  require
    a\_successor\_not\_void: a\_successor \neq \texttt{Void}
  do
    next := a\_successor
  ensure
    next\_set: next = a\_successor
end
feature -- Access
  next: \texttt{HANDLER} [G]
    -- Successor in the chain of responsibility
feature -- Status report
  can\_handle (a\_request: G): BOOLEAN is deferred end
  -- Can current handle \textit{a\_request}?
handled: BOOLEAN
  -- Has request been handled?
Class **HANDLER** [G] (2/3)

**feature** -- Basic operation

```plaintext
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
**Chain of Responsibility: feature handle**

```plaintext
deferred class HANDLER [G]

... 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
    ...
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
Design Patterns and Contracts, by Jézéquel et al. (p 142) suggests having a class `HANDLER [G]`

But their class is not so complete as in the Chain of Responsibility Library and does not express contracts (e.g. No notion of `can_handle`, although it is an important part of the pattern)
Chain of Responsibility: Componentization outcome

- **Completeness**
  - All cases of the Chain of Responsibility pattern

- **Usefulness**
  - Reusable
  - Easy-to-use
  - Semantics of the pattern captured into the assertions
    
- **Faithfulness**
  - Similar to a traditional implementation of Chain of Responsibility
    (with genericity)

- **Type-safety**
  - Type-safe (unconstrained genericity, assertions)

- **Performance**
  - Same order as the Chain of Responsibility pattern

- **Extended applicability**
  - No more cases
Complementary material

- From Patterns to Components:
  - Chapter 12: Command and Chain of Responsibility

- Further reading:
  - Erich Gamma: *Design Patterns*, 1995. (Command, p 233-242; Chain of Responsibility, p 223-232)
End of lecture 19