Last update: 18 April 2007

Software Architecture

Bertrand Meyer

ETH Zurich, March-July 2007

Lecture 8: More patterns: Visitor, Strategy, Chain, State, Command

(with material by Karine Arnout & by Yi Wei)

Some design patterns

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

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."

[Gamma et al., p 331]

- Static class hierarchy
- Need to perform traversal operations on corresponding data structures
- Avoid changing the original class structure

Visitor application example

Set of classes to deal with an Eiffel or Java program (in EiffelStudio, Eclipse ...)

Or: Set of classes to deal with XML documents (*XML_NODE, XML_DOCUMENT, XML_ELEMENT, XML_ATTRIBUTE, XML_CONTENT…)*

One parser (or several: keep comments or not…)

Many formatters:
- Pretty-print
- Compress
- Convert to different encoding
- Generate documentation
- Refactor
- …

Library example

*
BORROWABLE

+
BOOK

+
DVD

We want to add external functionality, for example:
- Maintenance
- Visualization

Visit operations

Why is this approach bad ?

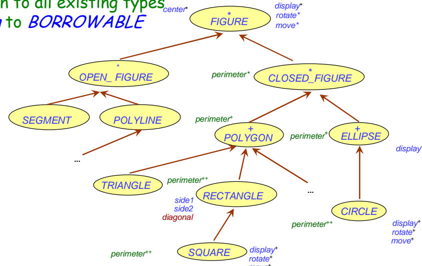
```
maintain (b: BORROWABLE) is
        -- Perform maintenance on b.
    require
        exists: b /= Void
    local
        book: BOOK
        dvd: DVD
    do
        book ?= b
        if book /= Void then
                ... Book maintenance ...
        end
        dvd ?= b
        if dvd /= Void then
                ... DVD maintenance ...
        end
    end
end
```

```
display (b: BORROWABLE) is
        -- Display b.
    require
        exists: b /= Void
    local
        book: BOOK
        dvd: DVD
    do
        book ?= b
        if book /= Void then
                ... Put book on display ...
        end
        dvd ?= b
        if dvd /= Void then
                ... Put DVD on display ...
        end
    end
end
```

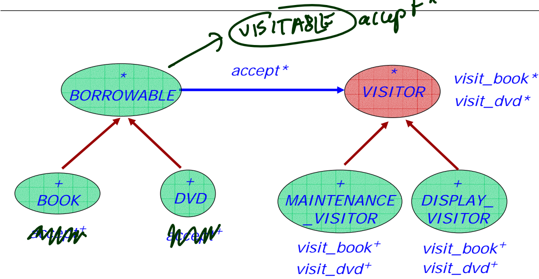
The O-O dilemma

Is it easy to

- Add types (e.g. *LOZENGE* to *FIGURE* hierarchy) with existing operations?
- Add operation to all existing types (e.g. *maintain* to *BORROWABLE* objects)?



Visitor: overall architecture



The maintenance visitor

```
class MAINTENANCE_VISITOR inherit
    VISITOR
feature -- Basic operations
    visit_book (b: BOOK) is
            -- Perform maintenance operations on b.
        do
            b.check_binding
            if b.damaged then b.repair end
        end

    visit_dvd (d: DVD) is
            -- Perform maintenance operations on d.
        do
            d.check_surface
            if d.damaged then d.order_replacement end
        end
end
```

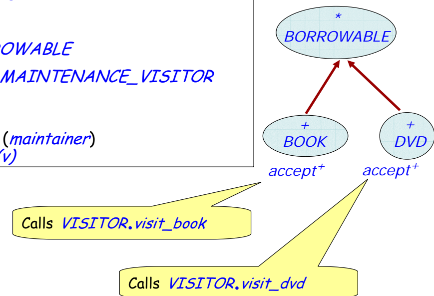

Visited objects

```
class
    BOOK
inherit
    BORROWABLE
feature
    accept (v: VISITOR) is
            -- Apply to v the book
            -- visit mechanism.
        do
            v.visit_book (Current)
        end
end
```

```
class
    DVD
inherit
    BORROWABLE
feature
    accept (v: VISITOR) is
            -- Apply to v the DVD
            -- visit mechanism.
        do
            v.visit_dvd (Current)
        end
end
```


Visitor - Usage

```
r(v: VISITOR) is
local
    item : BORROWABLE
    maintainer: MAINTENANCE_VISITOR
do
    …
    item.accept (maintainer)
    item.accept(v)
    …
end
```



Calls *VISITOR.visit_book*

Calls *VISITOR.visit_dvd*

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

Does the visitor pattern observe the Open-Closed Principle ?

Visitor vs dynamic binding

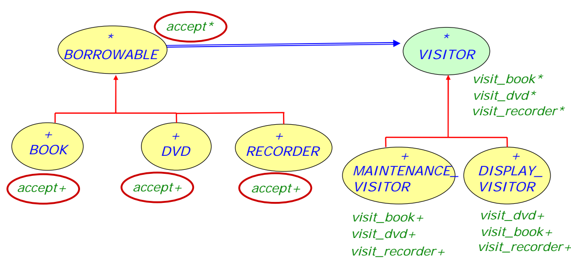
Dynamic binding:

- Easy to add types
- Hard to add operations

Visitor:

- Easy to add operations
- Hard to add types

The original visitor pattern



Can we make it easier for the application developer?

The Visitor Library

One generic class *VISITOR* [*G*]

e.g. *maintenance_visitor*: *VISITOR* [*BORROWABLE*]

Actions represented as agents

actions: *LIST* [*PROCEDURE* [*ANY*, *TUPLE* [*G*]]]

No need for *accept* features

- *visit* determines the action applicable to the given element

For efficiency

- Topological sort of actions (by conformance)
- Cache (to avoid useless linear traversals)

Visitor Library interface (1/2)

```
class
    VISITOR [G]
create
    make
feature {NONE} -- Initialization
    make
        -- Initialize actions.
feature -- Visitor
    visit (an_element: G)
            -- Select action applicable to an_element.
        require
            an_element_not_void: an_element /= Void
feature -- Access
    actions: LIST [PROCEDURE [ANY, TUPLE [G]]]
            -- Actions to be performed depending on the element
```

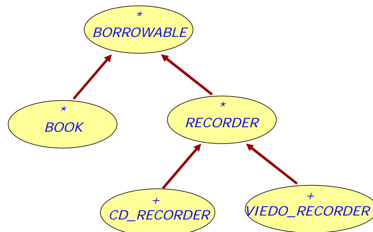

Visitor Library interface (2/2)

```
feature -- Element change
    extend (an_action: PROCEDURE [ANY, TUPLE [G]])
            -- Extend actions with an_action.
        require
            an_action_not_void: an_action /= Void
        ensure
            one_more: actions.count = old actions.count + 1
            inserted: actions.last = an_action

    append (some_actions: ARRAY [PROCEDURE [ANY, TUPLE [G]]])
            -- Append actions in some_actions
            -- to the end of the actions list.
        require
            some_actions_not_void: some_actions /= Void
            no_void_action: not some_actions.has (Void)
invariant
    actions_not_void: actions /= Void
    no_void_action: not actions.has (Void)
end
```

Using the Visitor Library

```
maintenance_visitor: VISITOR [BORROWABLE]

create maintenance_visitor.make
maintenance_visitor.append ([
                    agent maintain_book,
                    agent maintain_dvd ,
                    agent maintain_recorder
                ])

maintain_book (a_book: BOOK) is ...
maintain_video_dvd (a_dvd: DVD) is ...
maintain_recorder (a_recorder: RECORDER) is ...
```

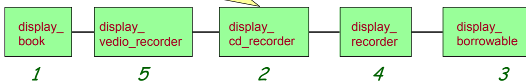

Topological sorting of agents (1/2)



Topological sorting of agents (2/2)

```
display_visitor.extend (agent display_book)
display_visitor.extend (agent display_cd_recorder)
display_visitor.extend (agent display_borrowable)
display_visitor.extend (agent display_recorder)
display_visitor.extend (agent display_video_recorder)
```

For agent *display_a* (*a*: A) and *display_b* (*b*: B), if A conforms to B, then position of *display_a* is before position of *display_b* in the agent list



display_visitor.visit (a_cd_recorder)

Strategy

Strategy - Intent

Purpose

"Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it".
[Gamma et al., p 315]

Example application

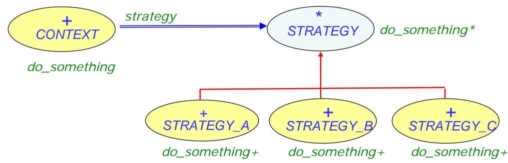
selecting a sorting algorithm on-the-fly

Life without strategy: a sorting example

```
feature -- Sorting

  sort (a_list: LIST [INTEGER]; a_strategy: INTEGER) is
        -- Sort a_list using algorithm indicated by a_strategy.
    require
        a_list_not_void: a_list /= Void
        a_strategy_valid: is_strategy_valid (a_strategy)
    do
        inspect
            a_strategy
        when binary then ...
        when quick then ...
        when bubble then ...
        else ...
        end
    ensure
        a_list_sorted: ...
  end
```

What if a new algorithm is needed ?

Strategy pattern: overall architecture



Class STRATEGY

```
deferred class
    STRATEGY

feature -- Basic operation

    do_something
            -- Do something (perform some algorithm).
        deferred
        end

end
```


Class CONTEXT (1/3)

```
class
    CONTEXT

create
    make

feature {NONE} -- Initialization

    make (a_strategy: like strategy) is
            -- Set strategy to a_strategy.
        require
            a_strategy_not_void: a_strategy /= Void
        do
            strategy := a_strategy
        ensure
            strategy_set: strategy = a_strategy
        end
```


Class CONTEXT (2/3)

```
feature – Basic operations

    do_something
            -- Do something (call algorithm cooresponding to strategy ).
        do
            strategy.do_something
        end

    set_strategy (a_strategy: like strategy)
            -- Set strategy to a_strategy.
        require
            a_strategy /= Void
        do
            strategy := a_strategy
        ensure
            strategy = a_strategy
        end
```


Class CONTEXT (3/3)

```
feature {NONE} – Implementation

    strategy: STRATEGY
            -- Strategy to be used

invariant
    strategy_not_void: strategy /= Void

end
```


Using strategy pattern

```
sorter_context: SORTER_CONTEXT

bubble_strategy: BUBBLE_STRATEGY
quick_strategy: QUICK_STRATEGY
hash_strategy: HASH_STRATEGY

create sorter_context.make (bubble_strategy)
sorter_context.sort (a_list)

sorter_context.set_strategy (quick_strategy)
sorter_context.sort (a_list)


sorter_context.set_strategy (hash_strategy)
sorter_context.sort (a_list)
```

Now, what if a new algorithm is needed ?

Strategy - Consequences

- Families of related algorithms
- An alternative to subclassing
- Eliminate conditional statements
- A choice of implementations

- Clients must be aware of different Strategies
- Communication overhead between Strategy and Context
- Increased number of objects

Strategy - Participants

Strategy

declares an interface common to all supported algorithms.

Concrete strategy

implements the algorithm using the Strategy interface.

Context

- is configured with a concrete strategy object.
- maintains a reference to a strategy object.

Chain of responsibility

Chain of responsibility - Intent

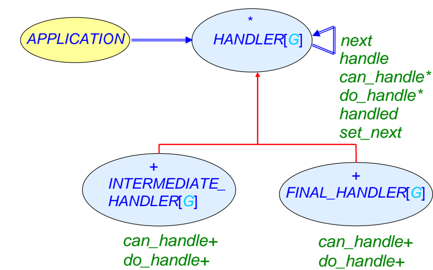
Purpose

> *"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."*
>
> [Gamma et al., p 223]

Example application

> *A GUI event is passed from level to level (such as from button to dialog and then to application)*

Chain of responsibility: overall architecture



Class *HANDLER* [*G*] (1/3)

```
deferred class
    HANDLER [G]
create default_create, make

feature {NONE} -- Initialization
    make (n : like next) is
                    -- Set next to n.
        do
                    next := n
        ensure
                    next_set: next = n
        end

feature -- Access
    next : HANDLER [G]
                    -- Successor in the chain of responsibility

feature -- Status report
    can_handle (r : G): BOOLEAN is deferred end
                    -- Can this handler handle r?

    handled: BOOLEAN
                    -- Has request been handled?
```

Class *HANDLER* [*G*] (2/3)

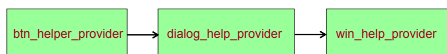
```
feature -- Basic operations
    handle (r : G) is
            -- Handle r if can_handle otherwise forward it to next.
            -- If no next, set handled to False.
        do
            if can_handle (r ) then
                do_handle (r ) ; handled := True
            else
                if next /= Void then
                    next.handle (r ) ; handled := next.handled
                else
                    handled := False
                end
            end
        ensure
            can_handle (r ) implies handled
            (not can_handle (r ) and next /= Void) implies handled = next.handled
            (not can_handle (r ) and next = Void) implies not handled
        end
```


Class *HANDLER* [*G*] (3/3)

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

feature {NONE} – Implementation

    do_handle (r : G) is
            -- Handle r.
        require
            can_handle: can_handle (r)
        deferred
        end

end
```


Using chain of responsibility

```
btn_help_provider: BUTTON_HELP_HANDLER [CLICK_REQUEST]
dialog_help_provider: DIALOG_HELP_HANDLER [CLICK_REQUEST]
win_help_provider: WINDOW_HELP_HANDLER [CLICK_REQUEST]

create win_help_provider
create dialog_help_provider.make (win_help_provider)
create btn_help_provider.make (dialog_help_provider)

btn_help_provider.handle (a_click_request)
```



Chain of responsibility - Consequences

Reduced coupling

An object only has to know that a request will be handled "appropriately". Both the receiver and the sender have no explicit knowledge of each other

Added flexibility in assigning responsibilities to objects

Ability to add or change responsibilities for handling a request by adding to or otherwise changing the chain at run-time

Receipt isn't guaranteed

the request can fall off the end of the chain without ever being handled

Chain of responsibility - Participants

Handler

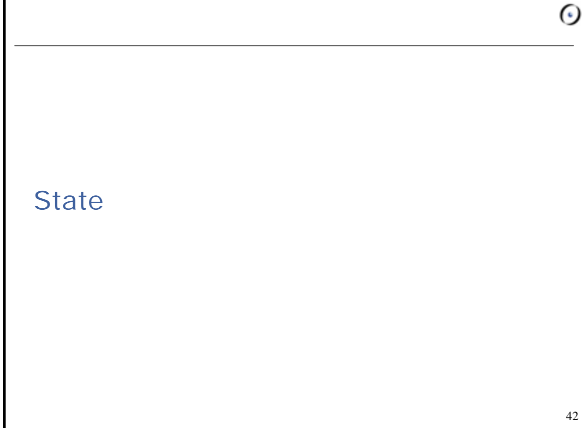
- defines an interface for handling requests.
- (optional) implements the successor link.

Concrete handler

- handles requests it is responsible for.
- can access its successor.
- if the Concrete handler can handle the request, it does so; otherwise it forwards the request to its successor.

Client

initiates the request to a Concrete handler object on the chain.



State

State - Intent

Purpose

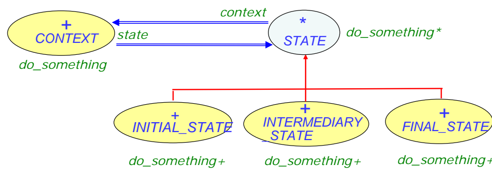
"*Allows an object to alter its behavior when its internal state changes. The object will appear to change its class*".

[Gamma et al., p 305]

Application example:

- add attributes without changing class
- state machine simulation

State: overall architecture



What's the difference between state and strategy ?

Class CONTEXT

```
class
   CONTEXT
…
feature – Basic operations

   do_something is
            -- Do something depending on the state.
      do
            state .do_something
      end

feature {NONE} -- Implementation

    state: STATE
      -- Current state

    some_other_state: STATE
      -- Other managed state
end
```


Class STATE

```
deferred class
   STATE
…

feature – Basic operations

   do_something is
               -- Do something
      deferred
      end

feature -- Access

    context: CONTEXT
      -- Context where current is used
…
end
```


Class CONCRETE_STATE

```
class
   CONCRETE_STATE

inherit
   STATE

feature – Basic operations

   do_something is
      -- Do something
      do
         …
         context.set_state (context.some_other_state)
      end

end
```


Using state (1/5): Class WEBSITE_CONTEXT

```
class WEBSITE_CONTEXT
feature – Basic operations
   login is
         -- Login.
      do
         state.login
      end

   logout is
         -- Logout.
      do
         state.logout
      end

   visit_url (a_url: URL) is
         -- Visit URL a_url.
      do
         state.visit (a_url)
      end
```

Using state (2/5): Class WEBSITE_CONTEXT

```
feature {NONE} -- Implementation
    state: ACCOUNT_STATE
        -- Current state

feature {ACCOUNT_STATE} -- Implementation
    loggedin_state,
    loggedout_state: like state
        -- States

    set_state (a_state: like state) is
            -- Logout state
        require
            a_state_not_void: a_state /= Void
        do
            state := a_state
        ensure
            state_set: state = a_state
        end
end
```


Using state (3/5): Class ACCOUNT_STATE

```
deferred class ACCOUNT_STATE

feature – Basic operations
    login is deferred end

    logout is deferred end

    visit_url (a_url: URL) is
        require
            a_url_not_void: a_url /= Void
        deferred
        end

feature{NONE} – Implementation
    context: WEBSITE_CONTEXT
        -- Context
…
end
```


Using state (4/5): Class LOGIN_STATE

```
class
    LOGGEDIN_STATE
inherit
    ACCOUNT_STATE

feature – Basic operations
    login is do end

    logout is
        do
            context.set_state(context.loggedout_state)
        end

    visit_url (a_url: URL) is
        do
            -- Go to URL.
        end
end
```


Using state (5/5): Class LOGOUT_STATE

```
class
    LOGGEDOUT_STATE
inherit
    ACCOUNT_STATE

feature – Basic operations
    login is
        do
            context.set_state(context.loggedin_state)
        end

    logout is do end

    visit_url (a_url: URL) is
        do
            -- Disallow visit.
        end
end
```


State - Consequences

It localizes state-specific behavior and partitions behavior for different states

It makes state transitions explicit

State objects can be shared

State - Participants

Context

- defines the interface of interest to clients.
- maintains an instance of a Concrete state subclass that defines the current state.

State

defines an interface for encapsulating the behavior associated with a particular state of the Context.

Concrete state

each subclass implements a behavior associated with a state of the Context

Command

Command pattern - Intent

Purpose

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."

[Gamma et al., p 233]

Application example

EiffelStudio

The problem

Enabling users of an interactive system to cancel the effect of the last command.

Often implemented as "Control-Z".

Should support multi-level undo-redo, with no limitation other than a possible maximum set by the user

A good review of O-O techniques

Working example: text editor

Notion of "current line". Assume commands such as:

- Insert line after current position
- Insert line before current position
- Delete current line
- Replace current line
- Swap current line with next if any
- ...

This is a line-oriented view for simplicity, but the discussion applies to more sophisticated views

Underlying class (from "business model")

```
class EDIT_CONTROLLER feature
    text : LINKED_LIST [STRING]

    remove is
            -- Remove line at current position.
        require
            not off
        do
            text.remove
        end

    put_right (line: STRING) is
            -- Insert line after current position.
        require
            not after
        do
            text.put_right (line)
        end
    ...
end
```


Key step in devising a software architecture

Finding the right abstractions

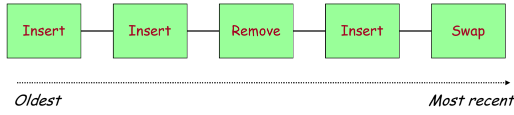
(Interesting object types)

Here:

The notion of "command"

Keeping the history of the session

The history list:



history: *LINKED_LIST* [*COMMAND*]

What's a "command" object?

An instance of *COMMAND* includes information about one execution of a command by the user, sufficient to:

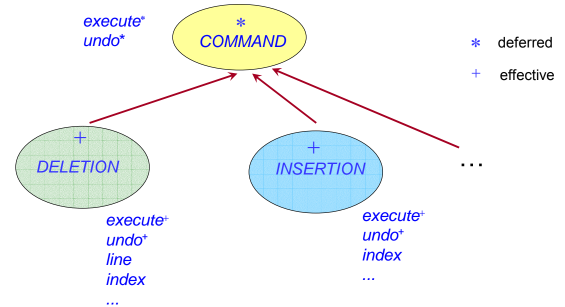
- Execute the command
- Cancel the command if requested later

For example, in a delete command object, we need:
- The position of the line being deleted
- The content of that line

General notion of command

```
deferred class COMMAND feature
    done: BOOLEAN is
            -- Has this command been executed?

    execute is
            -- Carry out one execution of this command.
        deferred
        ensure
            already: done
        end

    undo is
            -- Cancel an earlier execution of this command.
        require
            already: done
        deferred
        end
end
```


Command class hierarchy

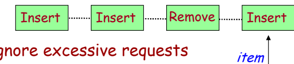


A command class (sketch, no contracts)

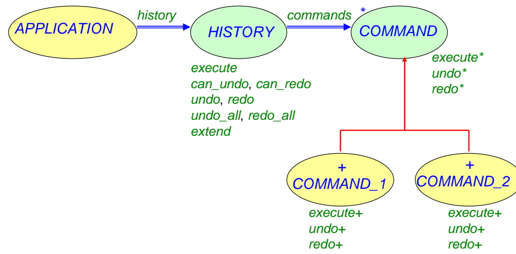
```
class DELETION inherit COMMAND feature
    controller: EDIT_CONTROLLER
            -- Access to business model
    line: STRING
            -- Line being deleted
    index: INTEGER
            -- Position of line being deleted
    execute is
            -- Remove current line and remember it.
        do
            line := controller.item ; index := controller.index
            controller.remove ; done := True
        end
    undo is
            -- Re-insert previously removed line.
        do
            controller.go_ith (index)
            controller.put_left (line)
        end
end
```


Executing a user command

```
decode_user_request
if "Request is normal command" then
    "Create command object c corresponding to user request"
    history.extend (c)
    c.execute
elseif "Request is UNDO" then
    if not history.before then        -- Ignore excessive requests
        history.item.undo
        history.back
    end
elseif "Request is REDO" then
    if not history.is_last then -- Ignore excessive requests
        history.forth
        history.item.execute
    end
end
```



Command pattern: overall architecture



The undo-redo pattern

Has been extensively used (e.g. in Eiffel tools)

Fairly easy to implement

Details must be handled carefully (e.g. some commands may not be undoable)

Elegant use of O-O techniques

Disadvantage: explosion of small classes

In Java, can use "inner" classes.

Using agents

For each user command, have two routines:

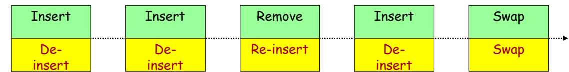
- The routine to do it
- The routine to undo it!

The history list using agents

The history list simply becomes a list of agents pairs:

```
history: LINKED_LIST [TUPLE
                [PROCEDURE [ANY, TUPLE],
                PROCEDURE [ANY, TUPLE]]
```

Basic scheme remains the same, but no need for command objects any more; the history list simply contains agents.



Executing a user command (before)

```
decode_user_request
if "Request is normal command" then
    "Create command object c corresponding to user request"
    history.extend (c)
    c.execute
elseif "Request is UNDO" then
    if not history.before then -- Ignore excessive requests
        history.item.undo
        history.back
    end
elseif "Request is REDO" then
    if not history.is_last then -- Ignore excessive requests
        history.forth
        history. item.execute
    end
end
```

Insert — Insert — Remove — Insert

item

Executing a user command (now)

```
"Decode user_request giving two agents do_it and undo_it"
if "Request is normal command" then
    history.extend ([do_it, undo_it])
    do_it.call ([])
elseif "Request is UNDO" then
    if not history.before then
        history.item.item (2) .call ([])
        history.back
    end
elseif "Request is REDO" then
    if not history.is_last then
        history.forth
        history.item.item (1) .call ([])
    end
end
```

Insert	Insert	Remove	Swap
De-insert	De-insert	Re-insert	Swap

Command - Consequences

Command decouples the object that invokes the operation from the one that knows how to perform it.

Commands are first-class objects. They can be manipulated and extended like any other object.

You can assemble commands into a composite command.

It's easy to add new Commands, because you don't have to change existing classes.

Command - Participants

Command

declares an interface for executing an operation.

Concrete command

- defines a binding between a Receiver object and an action.
- implements Execute by invoking the corresponding operation(s) on Receiver.

Client

creates a ConcreteCommand object and sets its receiver.

Invoker

asks the command to carry out the request.

Receiver

knows how to perform the operations associated with carrying out a request. Any class may serve as a Receiver.