Software Architecture

Lecture 9:
About design patterns

Reading assignment

- Chapter 20 & 21 of OOSC (Multi-panel & Undo-redo)
- Componentization of the visitor pattern:

Topics

- Patterns
- Abstract Factory Pattern
- Visitor
- Observer
- Chain of responsibility
- Command
- From patterns to components
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.

Some design patterns

- Creational
  - Abstract Factory
  - Builder
  - Factory Method
  - Prototype
  - Singleton
- Behavioral
  - Chain of Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template Method
  - Visitor
- Structural
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Façade
  - Flyweight
  - Proxy

Benefits of design patterns

- Capture the knowledge of experienced developers
- Teachable to newcomers
- Yield a better structure of the software
- Facilitate discussions between programmers and managers

Erich Gamma, Ralph Johnson, Richard Helm, John Vlissides: Design Patterns, Addison-Wesley, 1994
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

Some design patterns

- 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
**Abstract factory - Intent**

"Provide an interface for creating families of related or dependent objects without specifying their concrete classes." [Gamma et al.]

**Abstract Factory: example need**

Widget toolkit
- Different look and feel, e.g. for Unix & Windows
- Family of widgets: Scroll bars, buttons, dialogs...
- Want to allow changing look & feel

→ Most parts of the system need not know what look & feel is used
→ Creation of widget objects should not be distributed

**Architecture for widget example**
Architecture of a general example

Sketch of class FACTORY

defered class FACTORY
feature -- Basic operations
  new_button: BUTTON is
    deferred
    end
  new_checkbox: CHECKBOX is
    deferred
    end
end

Sketch of class WIN_FACTORY

class WIN_FACTORY
inher FACTORY
feature -- Basic operations
  new_button: BUTTON is
    do
      create (WIN_BUTTON) Result
    end
  new_checkbox: CHECKBOX is
    do
      create (WIN_CHECKBOX) Result
    end
end
Ancestor factory class

```ruby
class SHARED_FACTORY
 ...
  feature -- Basic operations
  factory: FACTORY is
    once
      if is_windows_os then
        create WIN_FACTORY Result
      else
        create UNIX_FACTORY Result
      end
  end
end
```

Usage of FACTORY

```ruby
class WIDGET_APPLICATION
  inherit SHARED_FACTORY
  ...
  feature -- Basic operations
  some_feature is
    local
      my_button: BUTTON
    do
      ...
      my_button := factory.new_button
    end
  ...
end
```

Reasons for using an abstract factory

- Most parts of a system should be independent of how its objects are created, represented and collaborating
- The 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
Abstract factory: pattern properties

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

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 XML documents
  - XML_NODE
  - XMLDOCUMENT
  - XML_ELEMENT
  - XML_ATTRIBUTE
  - XMLCONTENT
- One parser
- Many formatters
  - Pretty-print
  - Compress
  - Convert to different encoding
  - ...

Another example

AST of program
- Nodes: Class, Feature, instruction, ...
- Operations:
  - Compile
  - Pretty print
  - Generate documentation
  - Refactor

Library example

![Diagram of BORROWABLE, BOOK, and DVD]

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

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

**Visualization**

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

**Visitor pattern: a typical example**

Diagram showing the visitor pattern with various classes such as `BOOK`, `DVD`, `MAINTENANCE_VISITOR`, `DISPLAY_VISITOR`, and the `VISITOR` class, illustrating the `accept` method for each class.

Why is this approach bad?
### Class MAINTENANCE_VISITOR

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

    visit_dvd (d: DVD) is
      do
        d.check_surface
        if d.damaged then
          d.order_replacement
      end
    end
end
```

### Class BOOK

```plaintext
class BOOK
  inherit BORROWABLE
  feature
    accept (v: VISITOR) is
      do
        v.visit_book (Current)
      end
end
```

### Class DVD

```plaintext
class DVD
  inherit BORROWABLE
  feature -- Visitor pattern
    accept (v: VISITOR) is
      do
        v.visit_dvd (Current)
      end
end
```
### Visitor - Usage

```java
local
    item : BORROWABLE
    maintainer : MAINTENANCE_VISITOR
end

do
    item.accept (maintainer)
end
```

### 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
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. [Gamma et al., p 293]
### Book library example (1/4)

```plaintext
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
```

### Book library example (2/4)

```plaintext
feature -- Access
books: LINKED_LIST [BOOKS]
-- Books currently in the library
feature -- Element change
add_book (b: BOOK) is
  -- Add b to the list of books and notify all library observers.
  require
    b_not_void: b /= Void
    not_yet_in_library: not books.has (b)
  do
    books.extend (b)
    notify_observers
  ensure
    one_more: books.count = old books.count + 1
    book_added: books.last = b
  end

invariant
books_not_void: books /= Void
no_void_book: not books.has (Void)
end
```

### Book library example (3/4)

```plaintext
class APPLICATION
inherit OBSERVER
rename
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
```
### Book library example (4/4)

```plaintext
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 `SUBJECTs`.

### A refresher on agents

- objects representing potential computations

\[
b \int_a^b my\_function (x) \, dx
\]

my_integrator.integral (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
  \[ agent \ a_0.f(a_1, a_2, a_3) \]

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

- This represents the routine, ready to be called. To call it:
  \[ u.call(\[]) \]

- Recall original name of agents: "delayed calls".

Agent types: Kernel library classes

- *ROUTINE* \[ [BASE, ARGS \to\ TUPLE] \]
- *PROCEDURE* \[ [BASE, ARGS \to\ TUPLE] \]
- *FUNCTION* \[ [BASE, ARGS \to\ TUPLE, RES] \]

Creating vs. calling an agent

- Writing:
  \[ agent \ my\_feature \]
  creates an agent, i.e. an object of type *ROUTINE*.

- To call an agent, one needs to execute `call` (with the proper arguments) to this ROUTINE object, e.g:
  \[ my\_routine.call(\[\]) \]
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:

  \[ u := \text{agent} \ a0.f \ (a1, a2, a3) \]
  \[ w := \text{agent} \ a0.f \ (a1, a2, ?) \]
  \[ x := \text{agent} \ a0.f \ (a1, ?, a3) \]
  \[ y := \text{agent} \ a0.f \ (a1, ?, ?) \]
  \[ z := \text{agent} \ a0.f \ (?, ?, ?) \]

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:

  ```
  mouse_click: EVENT_TYPE [TUPLE [INTEGER, INTEGER]] is
  once create Result
  ensure mouse_click_not_void: Result /= Void
  end
  ```

- The publisher triggers the event:

  ```
  mouse_click.publish ((x_position, y_position))
  ```

- The subscribers (“observers”) subscribe to events:

  ```
  my_button.mouse_click.subscribe (agent my_procedure)
  ```
Publisher, subscriber, subscribed object (2/2)

Subscriber (APPLICATION)

Subscribed objects to events

Observer

Subject

Book library example with the Event Library (1/2)

```plaintext
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
```

Book library example with the Event Library (2/2)

```plaintext
feature -- Element change
add_book (b: BOOK) is
  -- Add b to the list of books and publish book_event.
  require
    b_not_void: b /= Void
    not_yet_in_library: not books.has (b)
  do
    books.extend (b)
    book_event.publish ([b])
  ensure
    one_more: books.count = old books.count + 1
    book_added: books.last = b
end

invariant
  books_not_void: books /= Void
  book_event_not_void: book_event /= Void
end
```
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

Chain of Responsibility: Intent

"Avoid[1] s coupling the sender of a request to its receiver by giving more than one object a change to handle the request. [11] chain[s] the receiving objects and pass[es] the request along the chain until an object handles it." \textit{[GoF, p. 223]}
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?
Class HANDLER [G] (2/3)

```plaintext
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
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)

```plaintext
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
```

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
  require -- ???
  not handled
end
```

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
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]
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.

To be completed
```

Example using the Command pattern (1/2)

```plaintext
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)

```plaintext
feature (NONE) -- Implementation
  history: HISTORY is
    -- History of executed commands
    once
      create Result.make
      ensure
        history_not_void: Result /= Void
  end
end
```
Command pattern (self-executable)

Command: class \textit{SHARED\_HISTORY}

**Diagram:**
- **Application**
  - **History**
    - Has execute, can\_undo, can\_redo
  - **COMMAND**
    - Has execute\+, undo\+, redo\+
    - \textit{is\_once\_command} execute\*, undo\*, redo\*

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

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

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