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 design patterns

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

Architecture of a general example

Sketch of class FACTORY
Sketch of class *WIN_FACTORY*

```plaintext
class WIN_FACTORY
inherit FACTORY

feature -- Basic operations
    new_button: BUTTON is
        do
            Result := create {WIN_BUTTON}
        end
    end

    new_checkbox: CHECKBOX is
        do
            Result := create {WIN_CHECKBOX}
        end
    end

```

The one and only factory

```plaintext
class SHARED_FACTORY

feature -- Basic operations
    factory: FACTORY is
        once
            if is_windows_os then
                Result := create {WIN_FACTORY}
            else
                Result := create {UNIX_FACTORY}
            end
        end
    end
```

Usage of *FACTORY*

```plaintext
class WIDGET_APPLICATION
inherit SHARED_FACTORY

feature -- Basic operations
    a_feature is
        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 - 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

```plaintext
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

```plaintext
display (an_item: BORROWABLE) is
  -- Display an_item.
  require
    an_item_not_void: an_item /= Void
  local
    book: BOOK
    vcr: VIDEO_RECORDER
  do
    book ?= an_item
    if book /= Void then
      -- Display book.
    end
    vcr ?= an_item
    if vcr /= Void then
      -- Display VCR.
    end
  end
```

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

**Class MAINTENANCE_VISITOR**

```plaintext
class MAINTENANCE_VISITOR
inherit VISITOR
feature -- Basic operations
  visit_book (a_book: BOOK) is
    do
      a_book.check_binding
      if a_book.damaged then
        a_book.repair
      end
    end
  visit_video_recorder (a_recorder: VIDEO_RECORDER) is
    do
      a_recorder.check_reading_heads
      if a_recorder.damaged then
        a_recorder.send_to_reparation
      end
    end
end
```

**Class BOOK**

```plaintext
class BOOK
inherit BORROWABLE
feature -- Visitor pattern
  accept (a_visitor: VISITOR) is
    do
      a_visitor.visit_book (Current)
    end
end
```

**Class VIDEO_RECORDER**

```plaintext
class VIDEO_RECORDER
inherit BORROWABLE
feature -- Visitor pattern
  accept (a_visitor: VISITOR) is
    do
      a_visitor.visit_video_recorder (Current)
    end
end
```

Visitor - Usage

```plaintext
local
  item: BORROWABLE
  maintainer: MAINTENANCE_VISITOR
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

Observer pattern

"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)

Book library example (1/4)

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)

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

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 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
```

**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
  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

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

**Normal call vs. agent call**

- Normal call
  \[ a0.f (a1, a2, a3) \]
- Agent call (expression): preface it by keyword `agent`, yielding
  \[ agent a0.f (a1, a2, a3) \]
- For example:
  \[ u := agent a0.f (a1, a2, a3) \]
- This represents the routine, ready to be called. To call it:
  \[ u.call () \]
  -- For type of \( u \), see next
- Recall original name of agents: "delayed calls".

**Agent types: Kernel library classes**

- ROUTINE
  \[ \text{BASE, ARGS} \rightarrow \text{TUPLE} \]
  \[ \text{call} \]
- PROCEDURE
  \[ \text{BASE, ARGS} \rightarrow \text{TUPLE} \]
- FUNCTION
  \[ \text{BASE, ARGS} \rightarrow \text{TUPLE, RES} \]
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 ([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 := agent a0.f (a1, a2, a3) -- All closed (as before)
  w := agent a0.f (a1, a2, ?)
  x := agent a0.f (a1, ?, a3)
  y := agent a0.f (a1, ?, ?)
  z := 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]]
  ```
  - Mouse click event type
  - `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)

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

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

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

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

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, Command
The Chain of Responsibility design pattern is used to avoid the coupling of the sender of a request to its receiver by giving more than one object a chance to handle the request. The Chain of Responsibility pattern allows for deferred class and deferred methods. It is often implemented as a chain of objects, where each object in the chain has a "can handle" method. If an object can handle a request, it handles the request; otherwise, it passes the request to the next object in the chain. This ensures loose coupling between the sender of a request and its receiver, as the sender does not need to know the exact implementation details of the receiver.
Chain of Responsibility: feature handle

```java
defered class HANDLER

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

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

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)

```java
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 (1/2)
feature (NONE) -- Implementation

history: HISTORY is
    -- History of executed commands
    once create Result.make
    ensure history_not_void: Result /= Void
end
end

Example using the Command pattern (2/2)

Command: class SHARED_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