Object Oriented Software Construction

Prof. Dr. Bertrand Meyer

Lecture 9: Introduction to Patterns, Model View Controller and Observer Pattern

Till G. Bay
Introduction to Patterns, Model View Controller and Observer Pattern

- What is a Pattern?
- Model View Controller Pattern
- Observer Pattern
- Event Library
“A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design.”

Erich Gamma et al., Design Patterns: Elements of Reusable Object-Oriented Software, 1995, p 3.
Design pattern: A definition

A design pattern is a set of domain-independent architectural ideas — typically a design scheme describing some classes involved and the collaboration between their instances — captured from real-world systems that programmers can learn and apply to their software in response to a specific problem.

Karine Arnout, From Patterns to Components, 2004
A design pattern is given by one or more of
- A description of the pattern’s intent
- Use cases
- A software architecture for typical implementations
GoF’s description of a design pattern

- Pattern name and classification
- Intent
- Also known as
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample code
- Known uses
- Related patterns

Categorization by intent
The GoF 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
Creational 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
Creational design patterns (2/2)

- **Goal:**
  - Put more flexibility into the instantiation process

- **How:**
  - Through inheritance or delegation

- **What:**
  - Defer parts of object creation
Structural 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
Goal:
- Compose software elements into bigger structures

How:
- Through inheritance (static binding) or composition (flexibility)
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
Model View Controller

Views

Model

A = 50%
B = 30%
C = 20%
**Model View Controller**

<table>
<thead>
<tr>
<th>Model</th>
<th>View</th>
<th>Controller</th>
</tr>
</thead>
</table>
| - Encapsulates application state  
- Exposes application functionality  
- Notifies view of changes | - Renders the model  
- Sends user gestures to controller  
- Allows controller to select view | - Defines application behavior  
- Maps user actions to model updates  
- Selects view for response  
- One for each functionality |

- Change  
- Notification  
- State change  

- Feature calls  
- Events  
- View selection  
- User gestures

*Chair of Software Engineering*  
OOSC - Summer Semester 2005
Observer

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]
deferred class
  SUBJE merchant
inherit
  ANY
     redefine
       default_create
     end
feature {NONE} -- Initialization
default_create is
  -- Initialize observers.
do
    create observers.make
end

feature -- Observer pattern
  add_observer (an_observer: OBSERVER) is
    -- Add an_observer to the list of observers.
    require
      not_yet_an_observer: not observers.has (an_observer)
do
    observers.extend (an_observer)
ensure
  observer_added: observers.last = an_observer
  one_more: observers.count = old observers.count + 1
end
Class **SUBJECT (2/2)**

```
remove_observer (an_observer: OBSERVER) is
    -- Remove an_observer from the list of observers.
    require
        is_an_observer: observers.has (an_observer)
    do
        observers.search (an_observer)
        observers.remove
    ensure
        observer_removed: not observers.has (an_observer)
        one_less: observers.count = old observers.count - 1
    end

notifyObservers is
    -- Notify all observers. (Call update on each observer.)
    do
        from observers.start until observers.after loop
            observers.item.update
            observers.forth
        end
    end

observers: LINKED_LIST [OBSERVER]
    -- List of observers

invariant
    observers_not_void: observers /= Void
end
```
deferred class

Observer

feature -- Observer pattern

update is

-- Update observer according to the state of
-- the subject it is subscribed to.

defered
end

deferred
end
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
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)
   notifyObservers
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

Chair of Software Engineering
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
...

Chair of Software Engineering
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
A typical *SUBJECT*

class
  *MY_DATA*

*inherit* *SUBJECT*

*feature* -- Observer pattern
  
  *add is*
  
  -- Add Current to data to be observed.
      
      **do**
      
      -- Do something.
      
      **notifyObservers**
      
      **end**

  
  *remove is*
  
  -- Remove Current from data to be observed.
      
      **do**
      
      -- Do something.
      
      **notifyObservers**
      
      **end**

  
  **Redundancy:**
  
  → Hardly maintainable
  
  → Not reusable
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 \textit{SUBJECT} as argument to \textit{update} but would yield many assignment attempts to distinguish between the different \textit{SUBJECT}s.
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)
```
Subscriber variants

`click.subscribe (agent my_procedure)`

`my_button.click.subscribe (agent my_procedure)`

`click.subscribe (agent your_procedure (a, ?, ?, b))`

`click.subscribe (agent other_object.other_procedure)`
Publisher: Responsible for triggering ("publishing") events. (Corresponds to the subject of the Observer pattern.)

Subscribed object: Notified whenever an event (of the event type they are subscribed to) is published. (Corresponds to the observer of the Observer pattern.)

Subscriber: Registers subscribed objects to a given event type. (Corresponds to the class, which registers the observers to the subjects.)
Publisher, subscriber, subscribed object (2/2)

Subscriber (APPLICATION)

Subscribes objects to events

Subscribed objects

Publishers

Subscribed objects to events
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
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 \textit{MY\_CLASS}:
  - With the Observer pattern:
    - Need to write a descendant of \textit{OBSERVER} and \textit{MY\_CLASS}
    - ⇒ Useless multiplication of classes
  - With the Event Library:
    - Can reuse the existing routines directly as agents
End of lecture 9