



## Object Oriented Software Construction

Prof. Dr. Bertrand Meyer

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

Till G. Bay



## Design pattern: Gang of Four's description

*"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.



## Introduction to Patterns, Model View Controller and Observer Pattern

- What is a Pattern?
- Model View Controller Pattern
- Observer Pattern
- Event Library



## 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

## Description of a design pattern

5

- 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

## The GoF design patterns

7

- **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

## GoF's description of a design pattern

6

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

Categorization  
by intent

## Creational design patterns (1/2)

8

- **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)

9

- **Goal:**
  - Put more flexibility into the instantiation process
- **How:**
  - Through inheritance or delegation
- **What:**
  - Defer parts of object creation

## Structural design patterns (2/2)

11

- **Goal:**
  - Compose software elements into bigger structures
- **How:**
  - Through inheritance (static binding) or composition (flexibility)

## Structural design patterns (1/2)

10

- **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 (1/2)

12

- **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

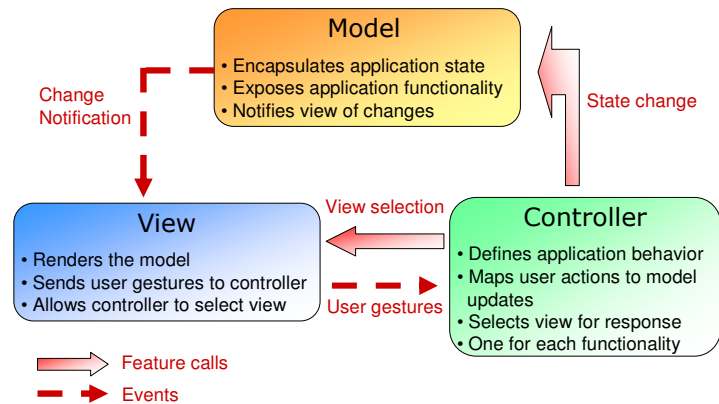
## Behavioral design patterns (2/2)

13

- Deal with:
  - Algorithms
  - Assignment of responsibilities between objects
  - Communication between objects
- How:
  - Through inheritance or composition

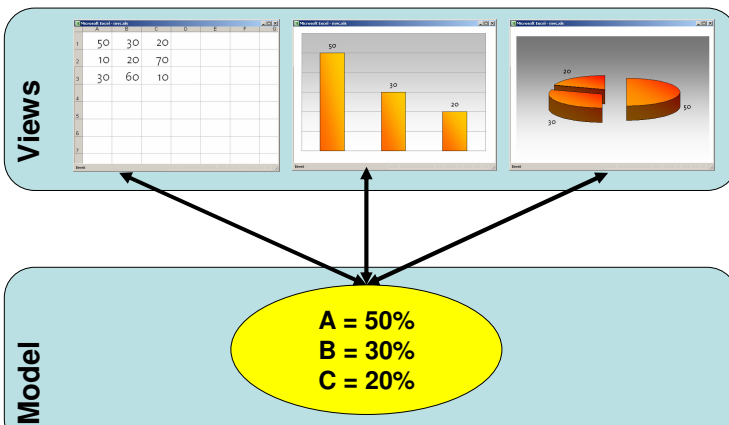
## Model View Controller

15



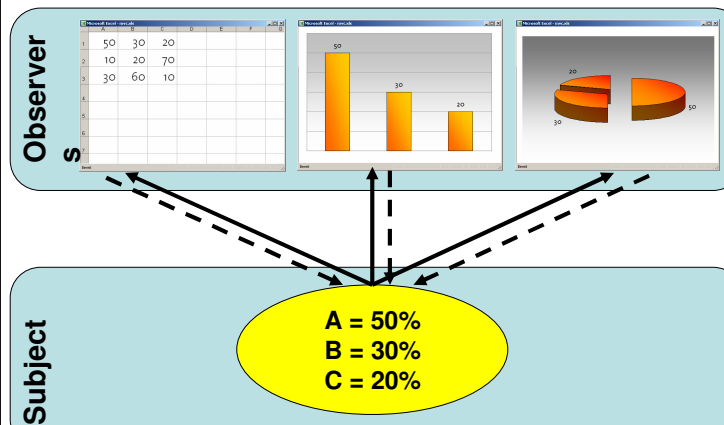
## Model View Controller

14



## Observer

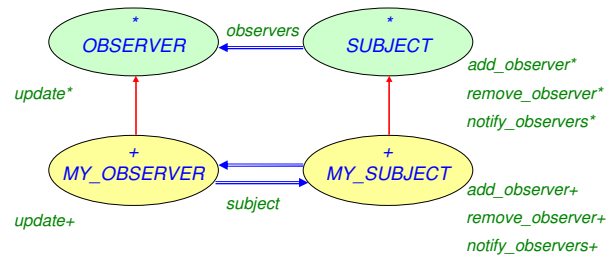
16



## Observer pattern

17

"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]



## Class SUBJECT (2/2)

19

```

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
notify_observers 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
  
```

## Class SUBJECT (1/2)

18

```

deferred class
  SUBJECT
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 OBSERVER

20

```

deferred class
  OBSERVER
feature -- Observer pattern
  update is
    -- Update observer according to the state of
    -- the subject it is subscribed to.
  deferred
  end
end
  
```

## Book library example (1/4)

21

```
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 (3/4)

23

```
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 (2/4)

22

```
feature -- Access
  books: LINKED_LIST [BOOK]
  -- 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 (4/4)

24

```
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*

25

```
class
  MY_DATA
inherit
  SUBJECT
feature -- Observer pattern
  add is
    -- Add Current to data to be observed.
    do
      -- Do something.
      notify_observers
    end
  remove is
    -- Remove Current from data to be observed.
    do
      -- Do something.
      notify_observers
    end
end
```

**Redundancy:**  
→ Hardly maintainable  
→ Not reusable

## Event Library

27

- 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*):

## Drawbacks of the Observer

26

- 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*.

## Example using the Event Library

28

- The publisher ("subject") creates an event type object:

```
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:

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

- The subscribers ("observers") subscribe to events:

```
my_button.mouse_click.subscribe (agent my_procedure)
```

## Subscriber variants

29

```
click.subscribe (agent my_procedure)
```

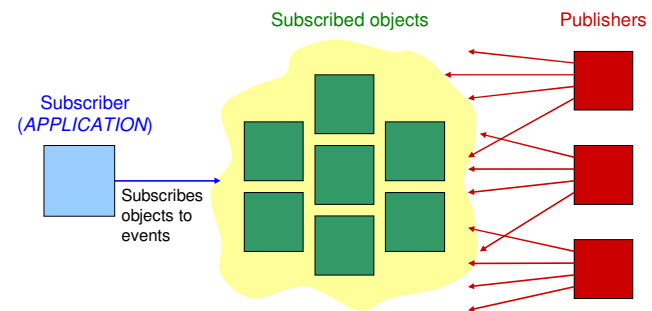
```
my_button.click.subscribe (agent my_procedure)
```

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

```
click.subscribe (agent other_object.other_procedure)
```

## Publisher, subscriber, subscribed object (2/2)

31



## Publisher, subscriber, subscribed object (1/2)

30

- **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.)

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

32

```
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
```



End of lecture 9



- 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