Trusted Components
Reuse, Contracts and Patterns

Prof. Dr. Bertrand Meyer
Dr. Karine Arnout
Lecture 14: Observer, Mediator
Agenda for today

- Observer pattern
- Event Library
- Mediator pattern
- Mediator Library
- Observer vs. Mediator
Componentizability classification

1. Componentizable
   1.1 Built-in
   1.2 Library-supported
   1.3 Newly componentized
   1.4 Possible component

   1.3.1 Fully componentizable
   1.3.2 Componentizable but not comprehensive
   1.3.3 Componentizable but unfaithful
   1.3.4 Componentizable but useless

   Prototype
   Flyweight
   Observer
   Mediator
   Abstract Factory
   Factory Method
   Visitor
   Command
   Composite
   Chain of Responsibility

2. Non-componentizable
   2.1 Skeleton
   2.2 Possible skeleton
   2.3 Some library support
   2.4 Design idea

   2.1.1 Method
   2.1.2 No method
   2.2.1 Singleton
   2.2.2 Iterator
   2.2.3 Facade
   2.2.4 Interpreter
   2.3.1 Decorator
   2.3.2 Adapter
   2.3.3 Template Method
   2.3.4 Bridge

Design 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]
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
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

observers: LINKED_LIST [OBSERVER]
   -- List of observers
invariant
   observers_not_void: observers /= Void
end
Class \textit{OBSERVER}

\begin{verbatim}
deferred class
  \textit{OBSERVER}

feature -- Observer pattern
  \textit{update is}
  -- Update observer according to the state of
  -- the subject it is subscribed to.
  deferred
    end
end
\end{verbatim}
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
endo
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
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

...
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

```haskell
class
    MY_DATA

inherit
    SUBJECT

feature -- Observer pattern
    add is
        do
            -- Add Current to data to be observed.
            notify_observers
        end

    remove is
        do
            -- Remove Current from data to be observed.
            notify_observers
        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 (in the “pull” version)

- An observer can register to at most one subject (in the “pull” version)
  - Could pass the \textit{SUBJECT} as argument to \textit{update} but would yield many assignment attempts to distinguish between the different \textit{SUBJECT}s.
In `java.util`, interface `Observer` and class `Observable` → Rarely used for lack of multiple inheritance (subjects must inherit from `Observable`)

- Common scheme (e.g. Java Swing, AWT): event-based implementation
  - Subjects define the registration methods:
    ```java
    void addXxxListener (XxxListener l)
    void removeXxxListener (XxxListener l)
    ```
  - Whenever a property being observed changes, the subject iterates over its listeners and calls the method defined by the `XxxListener` interface
Smalltalk supports the Observer pattern in the kernel library: class **Object**, shared by all objects, has messages (features) for both observer and subject objects.

- `update: anAspectSymbol
  update: anAspectSymbol with: aParameter
  update: anAspectSymbol with: aParameter from: aSender`  
  // Receive an update message from a Model (Subject).

- `changed
  changed: anAspectSymbol
  changed: anAspectSymbol with: aParameter`  
  // Receiver changed.

- `addDependent: anObject
  removeDependent: anObject`  
  dependents
  // Return collection of all dependents."
Agenda for today

- Observer pattern
- Event Library
- Mediator pattern
- Mediator Library
- Observer vs. Mediator
Event Library

- Basically:
  - One generic class: \textit{EVENT\_TYPE}
  - Two features: \textit{publish} and \textit{subscribe}

- For example: A button \textit{my\_button} that reacts in a way defined in \textit{my\_procedure} when clicked (event \textit{mouse\_click}):
Example using the Event Library

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

```haskell
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:

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

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

```haskell
my_button.mouse_click.subscribe (agent my_procedure)
```
Subscriber variants

\texttt{click.subscribe (agent my\_procedure)}

\texttt{my\_button.click.subscribe (agent my\_procedure)}

\texttt{click.subscribe (agent your\_procedure (a, ?, ?, b))}

\texttt{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 are connected to publishers.
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 *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
  - E.g. EiffelVision vs. EiffelVision2
Observer: Componentization outcome

- Completeness
  - All cases of the Observer pattern (and more)

- Usefulness
  - Easy-to-use, extendible
  - Event-driven programming
  - Already used in practice (JMLC paper, ESDL)

- Faithfulness
  - Agents instead of inheritance

- Type-safety
  - Type-safe (constrained genericity, agents)

- Performance
  - Same order as the Observer pattern

- Extended applicability
  - Event-driven programming in general
Agenda for today

- Observer pattern
- Event Library
- Mediator pattern
- Mediator Library
- Observer vs. Mediator
Componentizability classification

1. Componentizable
   - 1.1 Built-in
   - 1.2 Library-supported
   - 1.3 Newly componentized
     - 1.3.1 Fully componentizable
     - 1.3.2 Componentizable but not comprehensive
       - Flyweight
       - Observer
       - Mediator
       - Abstract Factory
       - Factory Method
       - Visitor
       - Command
       - Composite
       - Chain of Responsibility
     - 1.3.3 Componentizable but unfaithful
       - Strategy
     - 1.3.4 Componentizable but useless
       - Memento
   - 1.4 Possible component

2. Non-componentizable
   - 2.1 Skeleton
   - 2.2 Possible skeleton
   - 2.3 Some library support
   - 2.4 Design idea
     - Singleton
     - Iterator
     - Facade
     - Interpreter

Design pattern

Prototype

1. Componentizable
   - 1.1 Built-in
   - 1.2 Library-supported
   - 1.3 Newly componentized
     - 1.3.1 Fully componentizable
     - 1.3.2 Componentizable but not comprehensive
       - Flyweight
       - Observer
       - Mediator
       - Abstract Factory
       - Factory Method
       - Visitor
       - Command
       - Composite
       - Chain of Responsibility
     - 1.3.3 Componentizable but unfaithful
       - Strategy
     - 1.3.4 Componentizable but useless
       - Memento
   - 1.4 Possible component

2. Non-componentizable
   - 2.1 Skeleton
   - 2.2 Possible skeleton
   - 2.3 Some library support
   - 2.4 Design idea
     - Singleton
     - Iterator
     - Facade
     - Interpreter
Mediator pattern

“Define[s] an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.” [GoF, p 273]

“Anarchy” style

“Monarchy” style

Send and receive requests
Mediator pattern

* MEDIATOR
  + MY_MEDIATOR
  + COLLEAGUE_1
  + COLLEAGUE_2

* COLLEAGUE
  + COLLEAGUE_1
  + COLLEAGUE_2

mediator
update_colleagues

colleague_1
colleague_2

notify_mediator
class MY_MEDIATOR
inherit MEDIATOR
create
make
feature {NONE} -- Initialization
make is
  -- Create colleague_1 and colleague_2.
  do
    create colleague_1.make (Current)
    create colleague_2.make (Current)
  end
feature -- Access
  colleague_1: COLLEAGUE_1
    -- First colleague of mediator
  colleague_2: COLLEAGUE_2
    -- Second colleague of mediator
Class MY_MEDIATOR (2/2)

feature -- Basic operations
update_colleagues (a_colleague: COLLEAGUE) is
    -- Update colleagues because a_colleague changed.
    do
        if a_colleague = colleague_1 then
            colleague_2.do_something_2
        elseif a_colleague = colleague_2 then
            colleague_1.do_something_1
        else
            -- Colleague not known
        end
    end

end

invariant
    colleague_1_not_void: colleague_1 /\= Void
    colleague_2_not_void: colleague_2 /\= Void

end
deferred class
  COLLEAGUE
feature {NONE} -- Initialization
  make (a_mediator: like mediator) is
    -- Set mediator to a_mediator.
    require
      a_mediator_not_void: a_mediator /= Void
    do
      mediator := a_mediator
    ensure
      mediator_set: mediator = a_mediator
    end
  feature -- Access
    mediator: MEDIATOR
    -- Mediator
Class COLLEAGUE (2/2)

feature -- Mediator pattern

    notify_mediator is
        -- Notify mediator that Current has changed.
        do
            mediator.update_colleagues (Current)
        end
    end

invariant

    mediator_not_void: mediator /= Void

end
Class COLLEAGUE_1

class COLLEAGUE_1
  inherit COLLEAGUE
  create make
  feature -- Basic operations
  change_1 is
    do
      -- Do something that changes Current's state.
      notify_mediator
    end
  do_something_1 is
    -- Do something.
    do
      ...
    end
end

Similar to the Observer pattern

Event Library
Original Mediator pattern
Mediator pattern with the Event Library

![Diagram showing mediator pattern with Event Library](image-url)
Class COLLEAGUE_1

class
  COLLEAGUE_1
inherit
  COLLEAGUE
...
feature -- Basic operations
  change_1 is
    -- Change something on Current.
    do
      -- Do something that changes Current's state.
      event_1.publish ([])
    end
...
end
class MEDIATOR
create make
feature {NONE} -- Initialization
  make is
    do
      create colleague_1.make (Current)
      create colleague_2.make (Current)
      colleague_1.event_1.subscribe (agent colleague_2.do_something_2)
      colleague_2.event_2.subscribe (agent colleague_1.do_something_1)
    end
  ...
end
Agenda for today

- Observer pattern
- Event Library
- Mediator pattern
- Mediator Library
- Observer vs. Mediator
class interface
  MEDIATOR [G -> COLLEAGUE]
create
  make
    -- Initialize colleagues.
feature -- Access
  colleagues: LINKED_LIST [G]
    -- Colleagues of mediator
feature -- Element change
  extend (a_colleague: G)
    -- Extend colleagues with a_colleague.
    -- Update event subscription of colleagues.
require
  a_colleague_not_void: a_colleague /= Void
  not_a_colleague: not colleagues.has (a_colleague)
ensure
  one_more: colleagues.count = old colleagues.count + 1
  is_last: colleagues.last = a_colleague
  subscribed: colleagues.for_all (agent is_colleague_subscribed)
remove \( (a\_colleague: \ G) \) is
  -- Remove \( a\_colleague \) from \( \text{colleagues} \).
  -- Update event subscription of remaining \( \text{colleagues} \).

require
  a\_colleague\_not\_void: \ a\_colleague \ /= \ Void
  has\_colleague: \ colleagues\_has \ (a\_colleague)

ensure
  one\_less: \ colleagues\_count = \ old \ colleagues\_count \ − \ 1
  not\_has\_colleague: \ not \ colleagues\_has \ (a\_colleague)
  unsubscribed: \ a\_colleague\_unsubscribed

invariant

  colleagues\_not\_void: \ colleagues \ /= \ Void
  no\_void\_colleague: \ not \ colleagues\_has \ (Void)

end
deferred class interface
   COLLEAGUE

feature {NONE} -- Initialization
make (a_mediator: like mediator) is
   -- Set mediator to a_mediator.
   require
      a_mediator_not_void: a_mediator /= Void
   ensure
      mediator_set: mediator = a_mediator

feature -- Access
   mediator: MEDIATOR [COLLEAGUE]
      -- Mediator

   event: EVENT_TYPE [TUPLE]
      -- Event
feature -- Basic operations

  change is

    -- Do something that changes current colleague's
    -- state.

    do

      do_change

      event.publish ([]) 

    end

...

invariant

  mediator_not_void: mediator /= Void

  event_not_void: event /= Void

end
class USER
inherit COLLEAGUE
create
make

feature -- Status report
may_borrow: BOOLEAN
    -- May user borrow books from the library?

feature -- Element change
do_something is
do
    may_borrow := False
ensure then
    may_not_borrow: not may_borrow
end
feature \{NONE\} -- Implementation

  do_change is

  -- Borrow a book from the library.
  do
  if may_borrow then
    -- Borrow a book from the library.
    end
  end

end
Mediator: Componentization outcome

- **Completeness**
  - All cases of the Mediator pattern

- **Usefulness**
  - Reusable
  - Easy-to-use (Event Library)

- **Faithfulness**
  - Different from a traditional implementation of Mediator
  - Similar to an implementation of Mediator using the Event Library (with genericity)

- **Type-safety**
  - Type-safe (constrained genericity, agents)

- **Performance**
  - Same order as the Mediator pattern

- **Extended applicability**
  - No more cases
Agenda for today

- Observer pattern
- Event Library
- Mediator pattern
- Mediator Library
- Observer vs. Mediator
Mediator Library vs. Event Library (1/3)

- Mediator pattern: Every colleague “observes” all other colleagues → A particular use of the Event Library

- Mediator Library: Removes from clients the burden of (un)registering colleagues when a new colleague is added (removed) → Moved to the MEDIATOR

Mediator Library encapsulates a common use of the Event Library to facilitate the programmers’ job. It relies on the Event Library.
With the Event Library:
With the Mediator Library:
Complementary material

- From Patterns to Components:
  - Chapter 7: Observer and Mediator

- About the Event Library and Event-Driven programming:
  - [http://www.inf.ethz.ch/~meyer/ongoing/events.pdf](http://www.inf.ethz.ch/~meyer/ongoing/events.pdf)

- Further reading:
End of lecture 14