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

Structural
- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Behavioral
- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

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
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
  inherit 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

class SHARED_FACTORY
  ...
feature -- Basic operations
  factory: FACTORY is
    once
      if is_windows() then
        create {WIN_FACTORY} Result
      else
        create {UNIX_FACTORY} Result
      end
    end
  end
end

Usage of FACTORY

class WIDGET.APPLICATION
  inherit SHARED_FACTORY
  feature -- Basic operations
    some_feature is
      local
        my_button: BUTTON
      do
        ...
        my_button := factory.new_button
      end
    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.
"Testat"

To take the exam, you must be part of a group that has submitted a project that
- Runs
- Has a requirements document, or a user’s manual, or both
- Has a design document

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
- XML_DOCUMENT
- XML_ELEMENT
- XML_ATTRIBUTE
- XML_CONTENT

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

**Maintenance**

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

**Visualization**

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

Why is this approach bad?
Visitor: overall architecture

Class MAINTENANCE_VISITOR

Class BOOK
**Software Architecture - Lecture 10**

**Chair of Software Engineering**

**Class DVD**

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

**Visitor - Usage**

```ruby
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

Observers

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." [Gamma et al.]

Terminology:
- Objects being watched: SUBJECTS or PUBLISHERS
- Objects that need to be notified of changes: OBSERVERS or SUBSCRIBERS

Rest of this presentation uses "publish-subscribe" terminology

**General publisher (1/2)**

class PUBLISHER
feature -- Access
  subscribers: LINKED/List [SUBSCRIBER]
  -- Observers subscribed to this publisher
feature -- Element change
  add_subscriber (s: SUBSCRIBER)
  -- Register s as subscriber to this publisher.
  do
    subscribers.extend (s)
  end
feature – Observer pattern
notify_subscribers is
-- Broadcast change to all registered subscribers.
do
  from Subscribers.start
  until Subscribers.after
  loop
    Subscribers.item.update
    Subscribers.forth
  end
end

class LIBRARY
inherit PUBLISHER
redefine default_create
end

feature {NONE} -- Initialization
default_create is
-- Create and initialize library with empty book list.
do
  Precursor {PUBLISHER}
  create books.make
end

...
Library subscriber

class APPLICATION
inherit SUBSCRIBER
  rename update as display_book
  redefine default_create
end

feature (NONE) -- Initialization
  default_create is
  do
    create library
    library.add_subscriber (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.subscribers.has (Current)
end

Drawbacks of Observer structure

The publisher internally knows its subscribers

No information passing from publisher to subscriber when an event occurs

An subscriber can register with at most one publisher

- Could pass the PUBLISHER as argument to update but would yield many assignment attempts to distinguish between the different PUBLISHERs.
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: Intent**

"Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. [It] chain[s] the receiving objects and pass[es] the request along the chain until an object handles it." [Gamma et al.]

**Class HANDLER**

defined class HANDLER

feature = Basic operation
handle =
  handle request if can_handle otherwise forward it to next.
  If next is void, set handled to False.
  do
    if can_handle then
      do_handle
      handled := True
    else if next /= Void then
      next_handle
      handled := next.handled
    else
      handled := False
  end
ensure
  can_handle implies handled
  (not can_handle and then next /= Void) implies handled = next.handled
  (not can_handle and then next = Void) implies not handled
end

**Chain of Responsibility**
Class HANDLER [G]

defered class HANDLER [G]

feature (NONE) -- Initialization
    make (n: like next) is
        -- Set next to n.
        do
            next := n
        ensure
            next_set: next = n
        end

feature -- Access
    next: HANDLER [G]
        -- Successor in the chain of responsibility

feature -- Status report
    can_handle (r: G): BOOLEAN is deferred end
        -- Can this handler handle r?
    handled: BOOLEAN
        -- Has request been handled?

feature -- Basic operations
    handle (r: G) is
        -- Handle r if can_handle otherwise forward it to next.
        -- If no next, set handled to False.
        do
            if can_handle (r) then
                do_handle (r)
                handled := True
            else
                if next /= Void then
                    next.handle (r)
                    handled := next.handled
                else
                    handled := False
                end
            end
        ensure
            can_handle (r) implies handled
            (not can_handle (r) and next /= Void)
            implies not handled
            (not can_handle (r) and next = Void)
            implies not handled
        end

feature -- Element change
    set_next (n: like next) is
        -- Set next to n.
        do
            next := n
        ensure
            next_set: next = n
        end

feature (NONE) -- Implementation
    do_handle (r: G) is
        -- Handle r.
        require
            can_handle: can_handle (r)
    deferred end

Chain of Responsibility: feature handle

```
defered class HANDLER

feature -- Basic operation
handle (r) as

do
    if can_handle(r)
        do_handle(r)
        handled := True
    else
        if next /= Void
            next_handle(r)
        else
            handled := False
        end
    end
ensure
    ...
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

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." [Gamma et al., p 233]
Command pattern (history-executable)

- APPLICATION
- HISTORY
- COMMAND

create
make
feature
undo
redo
extend

How to use the Command pattern

Create a descendant of COMMAND and effect its features execute, undo, and redo.

class COMMAND_1
inherit COMMAND
create
make
feature
execute (args: TUPLE) is do end
undo (args: TUPLE) is do end
redo (args: TUPLE) is do end

end

Example using the Command pattern (1/2)

class APPLICATION
create
make
feature
make is
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)

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.
### Observer pattern: some limitations

Each publisher object knows about its observers

Only one `update` procedure in `SUBSCRIBER`:
- At most one operation
- Subscribe to at most one publisher!

Not reusable — must be coded anew for each application

(This is the difference between patterns & components!)

### 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

### Beyond patterns

A pattern is an architectural solution
Each programmer who needs the pattern has to learn it (externals and internals) and reimplement it for every application
Similar to traditional learning of algorithms and data structures

Can we do better: use components instead?

It's better to reuse than do redo.

(if reuse occurs through an API)
**Components over patterns**

Easier to learn

No need to learn implementation

Professional implementation will be better than manually crafted one

But: do we lose generality?

**Some references**

Karine Arnout & Bertrand Meyer: Componentization, the Factory Pattern (Innovations in Systems and Software Engineering, 2006)


Bertrand Meyer & Karine Arnout: Pattern componentization, the Visitor example, IEEE Computer, to appear, 2006


Karine Arnout: From Patterns to Components, PhD thesis, ETH, 2004 (see SE Web site)

**The key to pattern componentization**

Genericity (unconstrained and constrained)

Contracts

Once routines

Inheritance, in particular multiple inheritance

Tuples

Agents
A refresher on agents

objects representing potential computations

\[ \int_{a}^{b} \text{my\_function}(x) \, dx \]

\textit{my\_integrator.integral(agent my\_function, a, b)}

Normal call vs. agent call

Normal call

\[ a0.f(a1, a2, a3) \]

Agent call (expression): preface it by keyword \texttt{agent}, yielding

\[ \text{agent} \ a0.f(a1, a2, a3) \]

For example:

\[ u := \text{agent} \ a0.f(a1, a2, a3) \]

This represents the routine, ready to be called. To call it:

\[ u.\text{call}() \]

-- For type of \( u \), see next

Recall original name of agents: “delayed calls”.

Agent types: Kernel library classes

- \texttt{PROCEDURE}: \texttt{BASE, ARGS -> TUPLE}
- \texttt{FUNCTION}: \texttt{BASE,ARGS -> TUPLE, RES}
- \texttt{ROUTINE}: \texttt{BASE,ARGS -> TUPLE}
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])
```

Iterators

In class `LINEAR [G]`, ancestor to all classes for lists, sequences etc., you will find:

```
for_all
there_exists
do_all
do_if
do_while
do_until
```

Iterating on either the target or an argument

```
all_positive := my_integer_list.for_all (agent is_positive (?))
```

```
all_married := my_employee_list.for_all (agent EMPLOYEE.is_married)
```

This assumes

class C with:

```
is_positive (n : INTEGER): BOOLEAN
```

class `EMPLOYEE` with:

```
is_married : BOOLEAN
```
Applications of agents

Undo-redo
Iteration
High-level contracts
Numerical programming
Reflection

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:

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

Observer pattern: some limitations

Each publisher object knows about its observers

Only one update procedure in SUBSCRIBER:
- At most one operation
- Subscribe to at most one publisher!

Not reusable — must be coded anew for each application

(This is the difference between patterns & components!)
Another approach: event-action table

Event_type – Action Table

More precisely: Event_type - Context – Action Table

<table>
<thead>
<tr>
<th>Event type</th>
<th>Context</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left_click</td>
<td>Yes_button</td>
<td>Save_file</td>
</tr>
<tr>
<td>Left_click</td>
<td>Cancel_button</td>
<td>Reset</td>
</tr>
<tr>
<td>Left_click</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Right_click</td>
<td>...</td>
<td>Display_Menu</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Action-event table

Set of triples

\[
\{\text{Event, Context, Action}\}
\]

Event: any occurrence we track
Example: a mouse click

Context: object for which the event is interesting
Example: a particular button

Action: what we want to do when the event occurs in the context
Example: save the file

Action-event table may be implemented as e.g. a hash table.

Mechanisms in other languages

C and C++: “function pointers”

C#: delegates (more limited form of agents)
Problem scenario

One of your classes has a routine \textit{your\_procedure}

Your application has a GUI object known as \textit{Yes\_button}

Whenever the user clicks the mouse, the underlying GUI library returns the mouse coordinates

You want to ensure that a mouse click at coordinates \([h, v]\) in calls \textit{your\_procedure}(h, v)

\section*{With .NET delegates: publisher (1)}

P1. Introduce new class \textit{ClickArgs} inheriting from \textit{EventArgs}, repeating arguments types of \textit{yourProcedure}:

\begin{verbatim}
public class ClickArgs { ...
    int x, y;
    ...
}
\end{verbatim}

P2. Introduce new delegate type \textit{ClickDelegate} based on \textit{ClickArgs}:

\begin{verbatim}
public void delegate ClickDelegate (Object sender, e);
\end{verbatim}

P3. Introduce new event type \textit{Click} based on \textit{ClickDelegate}:

\begin{verbatim}
public event ClickDelegate Click;
\end{verbatim}

\section*{With .NET delegates: publisher (2)}

P4. Introduce new procedure \textit{OnClick} to wrap handling:

\begin{verbatim}
protected void OnClick (ClickArgs c)
    { if (Click != null) { Click (this, c); } }
\end{verbatim}

P5. For every event occurrence, create new object (instance of \textit{ClickArgs}), passing arguments to constructor:

\begin{verbatim}
ClickArgs yourClickargs = new ClickArgs (h, v);
\end{verbatim}

P6. For every event occurrence, trigger event:

\begin{verbatim}
OnClick (yourClickargs);
\end{verbatim}
With .NET delegates: subscriber

D1. Declare a delegate myDelegate of type ClickDelegate. (Usually combined with following step.)

D2. Instantiate it with yourProcedure as argument:

```csharp
ClickDelegate myDelegate = new ClickDelegate(yourProcedure);
```

D3. Add it to the delegate list for the event:

```csharp
YES_button.Click += myDelegate;
```

Event Library

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

For example: A button \texttt{my\_button} that reacts in a way defined in \texttt{my\_procedure} when clicked (event \texttt{mouse\_click}):  

Using the Event Library

The publisher creates an event type object:

```csharp
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:

```csharp
mouse_click.publish ((x\_position, y\_position))
```

The subscribers ("observers") subscribe to events:

```csharp
my_button.mouse_click.subscribe (agent my\_procedure)
```
Publisher, subscriber, subscribed object (2/2)

Subscriber
(APPLICATION)

Subscribed objects

Publishers

Observer

SubjectObserver

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

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

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 `SUBSCRIBER` and `MY_CLASS`
  - Useless multiplication of classes

- With the Event Library:
  - Can reuse the existing routines directly as agents

---

Observer pattern

---

Exercise

Investigate the use of agents to improve:
- Visitor
- Undo-redo