Chapter 9: Behavioural Design Patterns

King Fahd University of Petroleum & Minerals
SWE 316: Software Design & Architecture
Semester: 072
Objectives

- To introduce behavioral design patterns
  - Interpreter
  - Iterator
  - Mediator
  - Observer
  - State
  - Chain of Responsibility
  - Command
  - Template
Interpreter Design Pattern

- Design Purpose
  - Interpret expressions written in a formal grammar.

- Design Pattern Summary
  - Represent the grammar using a Recursive design pattern form: Pass interpretation to aggregated objects.
There are two parts relating to the Interpreter design pattern:

- Parsing
  - The process of converting an input, usually a string, into a tree of objects consistent with the class model.

- Interpreting
  - Performing useful functionality with the result of the parsing phase.
Interpreter: Class Model

```
Client

AbstractExpression
  interpret()

   1..n

TerminalExpression
  interpret()

NonTerminalExpression
  interpret()
```
Interpreter: Sequence Diagram

![Sequence Diagram](image)

- **Client**: AbstractExpression
- **NonterminalExpression**: ...
- **TerminalExpression**: create & interpret()
Network Assembly Example

Please describe a network on one line using the following grammar for 'component.' Blank spaces are ignored.

```
component ::= net system | computer
net system ::= { component } { component } | { component }
computer ::= ( cpu ram )
cpu ::= integer
ram ::= integer
```

Example: `{ {((400 4)){ (900 3)}} }((600 3)) } { (750 10) }
An input with a syntactic error will be ignored without comment.

```
{ {((111 11)){ (222 22)}} }((333 33)) } { (444 44) }
You chose { {((111 11)){ (222 22)}} }((333 33)) } { (444 44) }
```
Network Assembly Example (Cont’d)
Comments on Interpreter

- It's easy to change and extend the grammar.

- *Interpreter* has features in common with the *Composite* design pattern.
  - They are certainly both in recursive form.
  - *Composite* emphasizes the *representation* of a data structure (a *static* viewpoint), whereas *Interpreter* emphasize the *functional* viewpoint (interpreting the expression).
Iterator Design Pattern

- **Design Purpose**
  - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

- **Design Pattern Summary**
  - Encapsulate the iteration in a class pointing (in effect) to an element of the aggregate.
Purpose of Iterator

- Given a collection of objects
  
  e.g.,
  
  - the videos in a video store
  - a directory

- Having specified ways to progress through them
  
  e.g.,
  
  - “list in alphabetical order”
  - “list all videos currently on loan”

- Encapsulate each of these ways
Functions of Iterator

// Iterator "points" to first element:
void setToFirst();

// true if iterator "points" past the last element:
boolean isDone();

// Causes the iterator to point to its next element:
void increment();

// Return the element pointed to by the iterator:
getCurrentElement();
Using Iterator Functions

/*
To perform desiredOperation() on elements of the aggregate according to the iteration (order) i:
*/

for( i.setToFirst(); !i.isDone(); i.increment() )
    desiredOperation( i.getCurrentElement() );
Iterator: Class Model

- **Iterator**
  - `setToFirst()`
  - `increment()`
  - `isDone()`
  - `getCurrentItem()`

- **ConcreteIterator**

- **Aggregate**

- **ConcreteAggregate**

- **AggregatedElement**: 1..n
Comments on Iterator

- Iterator separates the “visiting” procedure from the processing of individual aggregated elements.

- Every collection must have an iterator of one kind or another.
  - A collection in which we can not access all of its individual element is useless.
Mediator Design Pattern

- **Design Purpose**
  - Avoid references between dependent objects.

- **Design Pattern Summary**
  - Capture mutual behavior in a separate class.
Mediator design pattern consists of a base class `Mediator`, each subclass of which encapsulates desired interaction among objects of `ConcreteColleague1`, `ConcreteColleague1`, etc.

All interacting objects belong to subclasses of a base class `Colleague` that references `Mediator`.

This ensures that the interacting objects need not know about each other.
Mediator: Sequence Diagram

1. Initiation by ConcreteMediator

2. Initiation on a ConcreteColleague
Customer Information Example

**Basic Information**

Please select customer type:
- regular customer
- volume customer
- select customer

**Name and Location**

- Name
- Street
- City
Customer Information Example (Cont’d)

Please select customer type:
- regular customer
- volume customer
- select customer

Account Information:
- Customer ID
- Total business
- Amount due

Additional information
Customer Information Example (Cont’d)
Mediator: Consequences

- It decouples colleagues.
- It centralizes control.
Observer Design Pattern

- Design Purpose
  - Arrange for a set of objects to be affected by a single object.
  - i.e. keep a set of objects up to date with the state of a designated object.

- Design Pattern Summary
  - The single object aggregates the set, calling a method with a fixed name on each member.
Observer: Motivation

The diagram illustrates the Observer pattern, where a subject (in the center) notifies observers (the three windows) of changes. The subject contains statistics for 'a', 'b', and 'c' with values of 50%, 30%, and 20% respectively. The windows display different types of data, with one showing a bar chart and another a pie chart. The arrows indicate change notifications and requests for modification.
Observer: Class Model

1. Client
2. Source
   - notify()
3. Observer
   - update()
   - for all Observer's o:
     - o.update();

ConcreteSource
- state

ConcreteObserver
- observerState
- update()
International Hamburger Co. Example

```java
if (abs(hq.demand - marketingDemand) > 0.01)
{
    marketingDemand = hq.getDemand();
    doDisplay();
}
```
Observer: Consequences

- *Observer* allows the addition and removal of observers without disrupting existing observer code.

- *Observer* may be disadvantageous if very few of the observers need to react to changes (in which case the possibly frequent notifications waste resources).

- *Observer* can’t work if the observable cannot have a reference to all of the observers (in client-server, the server should maintain a reference to all clients).
State Design Pattern

- Design Purpose
  - Cause an object to behave in a manner determined by its state.

- Design Pattern Summary
  - Aggregate a State object and delegate behavior to it.
State: Class Model

Client

Target
doRequest()

TargetState

{ targetState.handleRequest(); }

TargetState

handleRequest()

......

TargetStateA

handleRequest()

TargetStateB

handleRequest()
Role-Playing Game Example

```
MyGame
  setCharacteristics()
  {
    state.handleClick();
  }

MyGameState
  handleClick()

SettingUp
  handleClick()
  {
    showWindow();
    .... // more
  }

Waiting
  handleClick()
  {
    showWindow();
    .... // more
  }

SettingCharacteristics
  handleClick()
  {
    // already responding
    .... // display message
  }

Engaging
  handleClick()
  {
    // do nothing
    .... // display message
  }
```
State: Consequences

- It is particularly beneficial when new states are likely to be needed in the future.
Chain of Responsibility
Design Pattern

- Design Purpose
  - Allow a set of objects to service a request. Present clients with a simple interface.

- Design Pattern Summary
  - Link the objects in a chain via aggregation, allowing each to perform some of the responsibility, passing the request along.
Chain of Responsibility: Object Model

entryHandler: HandlerSubclassX

(next element): HandlerSubclassY

successor

(next element): HandlerSubclassZ

successor
Chain of Responsibility: Class Model

```
respondA(); // handle some of the required functionality
successor.handleRequest(); // pass along remaining responsibility
```
GUI For *Customer Information* Application

```
<customer>
  <professionalInfo>
    <company>
      <address>
        ABCDEFGHI
      </address>
      <company>
    </professionalInfo>
  </customer>
```
Class Model For User Information Collection

- TextFieldListener
  - «client»

- CustomerInfoElement
  - container

- CustomerPersonal
  - CustomerAddress
  - Company
  - CustomerProfessional
    - CustomerName
    - CustomerInfo
    - CompanyName

- CustomerInfoApp
  - «setup»
Object Model Fragment for *Customer Information* Example
Chain of Responsibility: Comments

- The class model for *Chain of Responsibility* is very similar to *Decorator*. The difference is in the viewpoint taken:
  - *Decorator* emphasizes the structural value of stringing objects together, primarily the static viewpoint.
  - *Chain of Responsibility* emphasizes the dynamic viewpoint: sharing functionality among a community of objects.

- The difference between the *Chain of Responsibility* and *Decorator* is that the principal aggregation in *Decorator* is with the base class whereas the aggregation is with the class itself in the case of *Chain of Responsibility*

- *Chain of Responsibility* has flexibility in that one can easily add or remove functionality from the collective group responsibility.
Command Design Pattern

- **Design Purpose**
  - Increase flexibility in calling for a service e.g., allow undo-able operations.

- **Design Pattern Summary**
  - Capture operations as classes.
**Command: Class Model**

Diagram showing the class model for the Command design pattern:

- **Client**
- **Target1**
  - action1()
- **Target2**
  - action2()
- **Command**
  - execute()
- **Action1Command**
  - execute()
- **Action2Command**
  - execute()
Menu Example

Menu Example diagram:
- **MenuItem**
  - `handleClick`

- **Command**
  - `execute`

- **CutCommand**
  - `execute`
  - `document.cut()`

- **CopyCommand**
  - `execute`
  - `document.copy()`

- **Document**
  - `cut`
  - `copy`
Menu Example (Cont’d)
**Undo Example**

Pick from one of the following:
undo
paste
quit
cut

Please specify index where the paste must start:
0
You chose 0
Please specify text to be inserted:
Mary had a little lamb.
You chose Mary had a little lamb.

Text is now as shown between the pair of arrows.
-->Mary had a little lamb.<--

Pick from one of the following:
undo
cut

Text is now as shown between the pair of arrows.
--><--

Pick from one of the following:
undo
cut

Text is now as shown between the pair of arrows.
 --><--

Pick from one of the following:
undo
cut

Text is now as shown between the pair of arrows.
 --><--

Pick from one of the following:
undo
quit
cut

quit

----- TERMINATING APPLICATION ----
Undo Example: Class Model

WordProcessor

Command
execute()
undo()

Document

text: String

document

DocumentCommand

getInputFromUser()

UndoCommand
execute()

QuitCommand
execute()

CutCommand
textCut: String
offset: int
execute()
undo()

PasteCommand
offset: int
length: int
execute()
undo()
We use *Command* whenever a method call alone is not sufficient: the ability to undo a command is one reason.

Menus often have submenus, sub-submenus or even more levels of menus.

- This means that there is a hierarchy of commands (a paste, special kind of paste, etc.).
- This hierarchy translates directly into an inheritance hierarchy of *Command* classes.
Command: Consequences

- *Command* affords us flexibility in dealing with operations, making it easier to modify, isolate, remove, or add operations to an application.

- *Command* promotes modularity, and thus correctness and reusability, because it separates functionality (e.g. a cut operation) from the object to which the functionality applies (e.g. a document whose text is to be cut).

- One downside of using *Command* is the fact that, as is common with design patterns, we are replacing what would be a single function call with multiple function calls.
Template Design Pattern

- **Design Purpose**
  - Allow runtime variants on an algorithm.

- **Design Pattern Summary**
  - Express the basic algorithm in a base class, using method calls where variation is required.
Example of Template Motivation

- Required to solve equations of the form \( ax^2 + bx + c = 0 \).

- Must be able to handle all input possibilities for \( a \), \( b \), and \( c \).

- This is a tutorial application that must provide full explanations to users about the solutions for all values for \( a \), \( b \), and \( c \).
A Basic Quadratic Algorithm

- Report progress
- Display number of solutions
- Display first solution, if any
- Display second solution, if any
Please enter 'a' in the form ....m.n....: 1
Please enter 'b' in the form ....m.n....: 3
Please enter 'c' in the form ....m.n....: 2
solving the equation ....
There are two solutions to this quadratic equation.
The reason is that b**2 - 4ac is positive.
There are two solutions because b**2 > 4ac.
The first solution is obtained from the formula \((-b + \sqrt{b^2 - 4ac}) / 2a\)
It is -1.0
There are two solutions because b**2 > 4ac.
The second solution is obtained from the formula \((-b - \sqrt{b^2 - 4ac}) / 2a\)
It is -2.0

Type quit if you don't want to solve more quadratic equations, otherwise any character.
more
Please enter 'a' in the form ....m.n....: 1
Please enter 'b' in the form ....m.n....: 2
Please enter 'c' in the form ....m.n....: 3
solving the equation ....
There are no solutions to this quadratic equation.
The reason is that b**2 < 4ac.
I/O For Quadratic Example (Cont’d)

Type quit if you don’t want to solve more quadratic equations, otherwise any character.
more
Please enter ‘a’ in the form ....m.n....
0
Please enter ‘b’ in the form ....m.n....
3
Please enter ‘c’ in the form ....m.n....
4
solving the equation ..... 
There is one solutions to this quadratic equation.
There is one solution because a is zero and so the equation is bx + c = 0.
The solution is thus obtained from the formula - c / b.
The solution is -1.3333333333333333

Type quit if you don’t want to solve more quadratic equations, otherwise any character.
more
Please enter ‘a’ in the form ....m.n....
0
Please enter ‘b’ in the form ....m.n....
0
Please enter ‘c’ in the form ....m.n....
1
solving the equation ..... 
There are no solutions to this quadratic equation.
The reason is that this equation is of the form 1.0 = 0 and no value for x can make this true.

Type quit if you don’t want to solve more quadratic equations, otherwise any character.
more
Please enter ‘a’ in the form ....m.n....
0
Please enter ‘b’ in the form ....m.n....
0
Please enter ‘c’ in the form ....m.n....
0
solving the equation ..... 
There are infinitely many solutions to this equation.
The reason is that this equation is of the form 0 = 0 and this is true regardless of what value x has.

Type quit if you don’t want to solve more quadratic equations, otherwise any character.
Class Model for Quadratic Problem

**QuadraticEquation**
- `float a, b, c`
- `solveQuadratic()`

**Setup**
- `getABC()`

**Client**

**QuadraticAlgorithm**
- `displayWhatsHappening()`
- `displaySolution()`
- `inputMakesSense()`
- `displayNumSolutions();`
- `displayFirstSolution();`
- `displaySecondSolution();`

**NoSolutions**
- `displayNumSolution();`
- `displayFirstSolution();`
- `displaySecondSolution();`

**OneSolution**
- `displayNumSolution();`
- `displayFirstSolution();`
- `displaySecondSolution();`

**TwoSolutions**
- `displayNumSolution();`
- `displayFirstSolution();`
- `displaySecondSolution();`

**InfinitelyManySolutions**
- `displayNumSolutions();`
- `displayFirstSolution();`
- `displaySecondSolution();`
Quadratic Display Algorithm

def displaySolutions()
{
    displayWhatsHappening();
    displayNumSolutions(); \textit{Override}
    displayFirstSolutions(); \textit{Override}
    displaySecondSolution(); \textit{Override}
}

Template: Class Model

```
Client

Subject
doRequest()

{ algorithm.handleRequest(); }

Algorithm1
calledMethod1()
calledMethod2()

Algorithm2
calledMethod1()
calledMethod2()

Algorithm3
calledMethod1()
calledMethod2()

TemplateAlgorithm
handleRequest()
nonInheritedMethod()
calledMethod1()
calledMethod2()
....
```
Template: Consequences

- *Template* promotes modularity (and thus correctness and reusability) by separating the core of a set of algorithms from its variants.

- The downside of using *Template* is the usual one with design patterns: We are introducing additional function calls, which reduces time efficiency.

- By separating algorithms into a hierarchy, we promote modularity (and thus the ability to verify the design correctness), maintainability (keeping track of the various algorithms more effectively) and flexibility (allowing us to change algorithms easily).
Summary of Behavioral Patterns

- **Behavioral Design Patterns** capture behavior among objects:
  - *Interpreter* handles expressions in grammars
  - *Iterator* visits members of a collection
  - *Mediator* captures behavior among peer objects
  - *Observer* updates objects affected by a single object
  - *State* allows method behavior to depend on current status
  - *Chain of Responsibility* allows a set of objects to provide functionality collectively
  - *Command* captures function flexibly (e.g. undo-able)
  - *Template* captures basic algorithms, allowing variability