Chapter 6:
Using Design Patterns
6.1 Introduction to Patterns

The recurring aspects of designs are called *design patterns*.

- A *pattern* is the outline of a reusable solution to a general problem encountered in a particular context.
- Many of them have been systematically documented for all software developers to use.
- A good pattern should
  - Be as general as possible
  - Contain a solution that has been proven to effectively solve the problem in the indicated context.

*Studying patterns is an effective way to learn from the experience of others.*
Pattern description

Context:
• The general situation in which the pattern applies

Problem:
— A short sentence or two raising the main difficulty.

Forces:
• The issues or concerns to consider when solving the problem

Solution:
• The recommended way to solve the problem in the given context.
  — ‘to balance the forces’

Antipatterns: (Optional)
• Solutions that are inferior or do not work in this context.

Related patterns: (Optional)
• Patterns that are similar to this pattern.

References:
• Who developed or inspired the pattern.
6.2 The Abstraction-Occurrence Pattern

• **Context:**
  — Often in a domain model you find a set of related objects *(occurrences)*.
  — The members of such a set share common information
    - but also differ from each other in important ways.

• **Problem:**
  — What is the best way to represent such sets of occurrences in a class diagram?

• **Forces:**
  — You want to represent the members of each set of occurrences without duplicating the common information
Abstraction-Occurrence

• Solution:

```
<Abstraction> * <Occurrence>

TVSeries
  seriesName
  producer

Episode
  number
  title
  storySynopsis

Title
  name
  author
  isbn
  publicationDate
  libOfCongress

LibraryItem
  barCodeNumber
```
Abstraction-Occurrence

Antipatterns:

- **LibraryItem**
  - name
  - author
  - isbn
  - publicationDate
  - libOfCongress
  - barCodeNumber

- **LibraryItem**
  - name
  - author
  - isbn
  - publicationDate
  - libOfCongress
  - barCodeNumber

- **Title**
  - name
  - author
  - isbn
  - publicationDate
  - libOfCongress

- **GulliversTravels**
  - barCodeNumber

- **MobyDick**
  - barCodeNumber

Books mentioned:
- **GulliversTravels**
- **MobyDick**
Abstraction-Occurrence

Square variant

```
ScheduledTrain
  number

SpecificTrain
  date

ScheduledLeg
  scheduledDepTime
  scheduledArrTime

SpecificLeg
  actualDepTime
  actualArrTime

Station
  origin
  destination
```
6.3 The General Hierarchy Pattern

• Context:
  — Objects in a hierarchy can have one or more objects above them (superiors),
    - and one or more objects below them (subordinates).
  — Some objects cannot have any subordinates

• Problem:
  — How do you represent a hierarchy of objects, in which some objects cannot have subordinates?

• Forces:
  — You want a flexible way of representing the hierarchy
    - that prevents certain objects from having subordinates
  — All the objects have many common properties and operations
General Hierarchy

• Solution:

```
«Node»
  «subordinate»
  «NonSuperiorNode»
  «SuperiorNode»

«NonSuperiorNode»
  «subordinate»
  «SuperiorNode»

Employee
  * supervises
  Secretary
  Technician
  Manager

FileSystemItem
  * contains
  File
  Directory
```

© Lethbridge/Laganière 2001
General Hierarchy

Antipattern:

```
Recording
  /\                     /\
 VideoRecording         AudioRecording
   /\                     /\
 MusicVideo             JazzRecording ClassicalRecording BluesRecording RockRecording
```

Chapter 6: Using design patterns
6.4 The Player-Role Pattern

• **Context:**
  — A *role* is a particular set of properties associated with an object in a particular context.
  — An object may *play* different roles in different contexts.

• **Problem:**
  — How do you best model players and roles so that a player can change roles or possess multiple roles?
Player-Role

• **Forces:**
  — It is desirable to improve encapsulation by capturing the information associated with each separate role in a class.
  — You want to avoid multiple inheritance.
  — You cannot allow an instance to change class

• **Solution:**

![Diagram showing the relationships between Player, AbstractRole, Role1, and Role2.]
Player-Role

Example 1:

```
Animal
  ▲
  |  typeOfFood
  |  0..2
  |  HabitatRole
  |  ▲
  |  habitat
  |  ▲
  Carnivore  Herbivore  Omnivore
  |  AquaticAnimal  LandAnimal
```

Example 1:

Animal
  - typeOfFood
  - 0..2
  - HabitatRole
    - habitat
      - Carnivore
      - Herbivore
      - Omnivore
      - AquaticAnimal
      - LandAnimal
Player-Role

Example 2:

```
AttendanceRole

Student

LevelRole

FullTimeStudent PartTimeStudent

GraduateStudent UndergraduateStudent
```

- AttendanceRole
- Student
- LevelRole
- FullTimeStudent
- PartTimeStudent
- GraduateStudent
- UndergraduateStudent

- attendance
- level
Player-Role

Antipatterns:

- Merge all the properties and behaviours into a single «Player» class and not have «Role» classes at all.
- Create roles as subclasses of the «Player» class.
6.5 The Singleton Pattern

• **Context:**
  — It is very common to find classes for which only one instance should exist (*singleton*)

• **Problem:**
  — How do you ensure that it is never possible to create more than one instance of a singleton class?

• **Forces:**
  — The use of a public constructor cannot guarantee that no more than one instance will be created.
  — The singleton instance must also be accessible to all classes that require it
Singleton

- Solution:

```
«Singleton»

theInstance
getInstance

Company

theCompany

Company «private»
getInstance

if (theCompany==null)
theCompany= new Company();
return theCompany;
```

```
6.6 The Observer Pattern

• **Context:**
  — When an association is created between two classes, the code for the classes becomes inseparable.
  — If you want to reuse one class, then you also have to reuse the other.

• **Problem:**
  — How do you reduce the interconnection between classes, especially between classes that belong to different modules or subsystems?

• **Forces:**
  — You want to maximize the flexibility of the system to the greatest extent possible
Observer

- **Solution:**

```
Observers are notified when a new prediction is ready

Forecaster

WeatherViewer
```
Observer

Antipatterns:

• Connect an observer directly to an observable so that they both have references to each other.
• Make the observers \textit{subclasses} of the observable.
6.7 The Delegation Pattern

• **Context:**
  — You are designing a method in a class
  — You realize that another class has a method which provides the required service
  — Inheritance is not appropriate
    - E.g. because the isa rule does not apply

• **Problem:**
  — How can you most effectively make use of a method that already exists in the other class?

• **Forces:**
  — You want to minimize development cost by reusing methods
Delegation

• Solution:

```
<Delegator>
  delegatingMethod
<Delegate>
  method
```

```
Stack
  push
  pop
  isEmpty
```

```
LinkedList
  addFirst
  addLast
  addAfter
  removeFirst
  removeLast
  delete
  isEmpty
```

```
delegatingMethod()
{  
delegate.method();
}
```

```
push()
{  
  list.addFirst();
}
```
Delegation

Example:

```java
class Booking {
    void flightNumber() {
        return specificFlight.flightNumber();
    }
}

class SpecificFlight {
    void flightNumber() {
        return specificFlight.flightNumber();
    }
}

class RegularFlight {
    void flightNumber() {
        return regularFlight.flightNumber();
    }
}
```
Delegation

Antipatterns

• Overuse generalization and *inherit* the method that is to be reused

• Instead of creating a *single* method in the «Delegator» that does nothing other than call a method in the «Delegate»
  — consider having many different methods in the «Delegator» call the delegate’s method

• Access non-neighboring classes
  
  return `specificFlight.regularFlight.flightNumber()`;

  return `getRegularFlight().flightNumber()`;
6.8 The Adapter Pattern

• **Context:**
  — You are building an inheritance hierarchy and want to incorporate it into an existing class.
  — The reused class is also often already part of its own inheritance hierarchy.

• **Problem:**
  — How to obtain the power of polymorphism when reusing a class whose methods
    - have the same function
    - but *not* the same signature
  as the other methods in the hierarchy?

• **Forces:**
  — You do not have access to multiple inheritance or you do not want to use it.
Adapter

- **Solution:**

```java
polymorphicMethod()
{
    return adaptee.adaptedMethod();
}
```
Adapter

Example:

```java
volume() {
    return adaptee.calcVolume();
}
```

![Class diagram](image)
6.9 The Façade Pattern

- **Context:**
  - Often, an application contains several complex packages.
  - A programmer working with such packages has to manipulate many different classes.

- **Problem:**
  - How do you simplify the view that programmers have of a complex package?

- **Forces:**
  - It is hard for a programmer to understand and use an entire subsystem.
  - If several different application classes call methods of the complex package, then any modifications made to the package will necessitate a complete review of all these classes.
Façade

• **Solution:**

- «Facade»
- «PackageClass1»
- «PackageClass2»
- «PackageClass3»
- Airline
  - findFlight
  - makeBooking
  - deleteBooking
- RegularFlight
- Person
6.10 The Immutable Pattern

• **Context:**
  — An immutable object is an object that has a state that never changes after creation

• **Problem:**
  — How do you create a class whose instances are immutable?

• **Forces:**
  — There must be no loopholes that would allow ‘illegal’ modification of an immutable object

• **Solution:**
  — Ensure that the constructor of the immutable class is the *only* place where the values of instance variables are set or modified.
  — Instance methods which access properties must not have side effects.
  — If a method that would otherwise modify an instance variable is required, then it has to return a *new* instance of the class.
6.11 The Read-only Interface Pattern

- **Context:**
  - You sometimes want certain privileged classes to be able to modify attributes of objects that are otherwise immutable

- **Problem:**
  - How do you create a situation where some classes see a class as read-only whereas others are able to make modifications?

- **Forces:**
  - Restricting access by using the `public`, `protected` and `private` keywords is not adequately selective.
  - Making access `public` makes it public for both reading and writing
Read-only Interface

• Solution:

```
<interface>
<ReadOnlyInterface>
getAttribute
</ReadOnlyInterface>
```

```
<Mutable>
attribute «private»
getAttribute
setAttribute
</Mutable>
```

```
<UnprivilegedClass>*
```

```
<Mutator>*
```

© Lethbridge/Laganière 2001
Read-only Interface

Example:

```
«interface» Person

getName

Mutableperson

firstName
lastName

setFirstName
setLastName
getName
```
Read-only Interface

Antipattern:

- Make the read-only class a *subclass* of the «Mutable» class
  - Override all methods that modify properties
    - such that they throw an exception
6.12 The Proxy Pattern

• **Context:**
  — Often, it is time-consuming and complicated to create instances of a class (*heavyweight* classes).
  — There is a time delay and a complex mechanism involved in creating the object in memory

• **Problem:**
  — How to reduce the need to create instances of a heavyweight class?

• **Forces:**
  — We want all the objects in a domain model to be available for programs to use when they execute a system’s various responsibilities.
  — It is also important for many objects to persist from run to run of the same program
Proxy

• Solution:

```
   «interface»
   «ClassIF»

«Client» * «Proxy» * «HeavyWeight»
```
Proxy

Example:

The list elements will be loaded into local memory only when needed.

```
@interface ListIF
@end

@interface PersistentList
@end

@interface Student
@end

@interface PersistentStudent
@end

@interface ListProxy
@end

@interface StudentProxy
@end
```
6.13 Detailed Example: The Observable layer of OCSF

AbstractClient

AdaptableClient
connectionEstablished
connectionClosed
handleMessageFromServer

Observable

ObservableClient

ObservableServer

AbstractServer

AdaptableServer
clientConnected
clientDisconnected
serverStarted
serverStopped
handleMessageFromClient

ConnectionToClient
The Observable layer of OCSF (continued)

```
ObservableClient
openConnection
closeConnection
sendToServer
isConnected
getPort
setPort
getHost
setHost
getInetAddress
handleMessageFromServer
connectionClosed
connectionEstablished

Observable

ObservableServer
listen
stopListening
close
sendToAllClients
isListening
clientConnections
getNumberOfClients
getPort
setPort
clientConnected
clientDisconnected
serverStarted
serverStopped
handleMessageFromClient

AdaptableClient

AdaptableServer
```

© Lethbridge/Laganière 2001
Chapter 6: Using design patterns 39
Using the Observable layer

1. Create a class that implements the **Observer** interface.
2. Register it as an observer of the **Observable**:
   ```java
   public MessageHandler(Observable observable) {
       client.addObserver(this);
       ...
   }
   
3. Define the **update** method in the new class:
   ```java
   public void update(Observable obs, Object message) {
       if (message instanceof SomeClass) {
           // process the message
       }
   }
   ```
6.14 Difficulties and Risks When Working with Patterns

• **Patterns are not a panacea:**
  
  — Whenever you see an indication that a pattern should be applied, you might be tempted to blindly apply the pattern. However this can lead to unwise design decisions.

• **Resolution:**
  
  — *Always understand in depth the forces that need to be balanced, and when other patterns better balance the forces.*
  
  — *Make sure you justify each design decision carefully.*
Difficulties and Risks When Working With Patterns

• Developing patterns is hard
  — Writing a good pattern takes considerable work.
  — A poor pattern can be hard to apply correctly

• Resolution:
  — Do not write patterns for others to use until you have considerable experience both in software design and in the use of patterns.
  — Take an in-depth course on patterns.
  — Iteratively refine your patterns, and have them peer reviewed at each iteration.