Design Patterns

1. Objectives.................................................................................................................. 2
2. Definitions.................................................................................................................... 3
   2.1. What is a Pattern? ................................................................................................. 3
   2.2. Categories of Patterns ......................................................................................... 4
3. Pattern Characteristics [Buschmann]: ................................................................. 5
4. Essential Elements of a Design Pattern [Gamma et al.] ........................................ 6
5. Describing Design Patterns [Gamma et al.] ............................................................ 7
6. Design Pattern Catalog ............................................................................................. 9
7. Examples of design patterns include the following ..... 11
   7.1. Singleton ............................................................................................................. 11
   7.2. Adapter (or Wrapper) pattern .............................................................................. 12
   7.3. Bridge (Also known as Handle/Body) ................................................................. 14
   7.4. Strategy (or Policy) Pattern ................................................................................ 20
8. Design Pattern Selection ......................................................................................... 23
9. How to Use a Design Pattern? ................................................................................ 24
10. Idioms ...................................................................................................................... 25
11. Summary .................................................................................................................. 26
1. **Objectives**

- Solutions are reusable.
- There are recurring patterns of classes and communicating objects that solve specific design problems and make these designs more flexible and reusable.
- A design pattern systematically names, explains and evaluates an important and recurring design in object-oriented systems.
- It is possible to collect these patterns into a catalog for designers to use to select and evaluate alternatives.
- Where are we in the software life cycle?

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Design</th>
<th>Implementation</th>
<th>.....</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Software Architectures</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Components</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Software Component Architecture</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSSA: Domain-Specific Software Architectures</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Frameworks</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Design Patterns</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Which Programming language?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Advanced Software Paradigms (A. Bellaachia)
2. Definitions

2.1. What is a Pattern?

- The concept of a pattern as used in software architecture is borrowed from the field of (building) architecture, in particular from the writings of architect Christopher Alexander.

- Christopher Alexander says, “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”.

- When experts need to solve a problem, they seldom invent a totally new solution. More often they will recall a similar problem they have solved previously and reuse the essential aspects of the old solution to solve the new problem. They tend to think in problem-solution pairs.

- "A pattern for software architecture describes a particular recurring design problem that arises in specific design contexts and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships,
and the ways in which they collaborate."
[Buschmann].

- Where software architecture is concerned, the concept of a pattern described here is essentially the same concept as an architectural style or architectural idiom in the Shaw and Garlan book.

2.2. Categories of Patterns

- Patterns can be grouped into three categories according to their level of abstraction [Buschmann]:
  - Architectural patterns or styles
  - Design patterns
  - Idioms
3. Pattern Characteristics [Buschmann]:

- A pattern describes a solution to a recurring problem that arises in specific design situations.
- Patterns are not invented; they are distilled from practical experience.
- Patterns describe a group of components (e.g., classes or objects), how the components interact, and the responsibilities of each component. That is, they are higher-level abstractions than classes or objects.
- Patterns provide a vocabulary for communication among designers. The choice of a name for a pattern is very important.
- Patterns help document the architectural vision of a design. If the vision is clearly understood, it will less likely be violated when the system is modified.
- Patterns provide a conceptual skeleton for a solution to a design problem and, hence, encourage the construction of software with well-defined properties.
- Patterns are building blocks for the construction of more complex designs.
- Patterns help designers manage the complexity of the software. When a recurring pattern is identified, the corresponding general solution can be implemented productively to provide a reliable software system.
4. Essential Elements of a Design Pattern [Gamma et al.]

- **Pattern Name**
  - ✓ A handle to describe a design problem, its solutions and consequences
  - ✓ Allows for design at a higher level of abstraction
  - ✓ Communication mechanism for designers

- **Problem**
  - ✓ Describes when to apply the pattern
  - ✓ Explains the problem and its context
  - ✓ May include a list of conditions that must be met before it makes sense to apply the pattern

- **Solution**
  - ✓ Describes the elements that make up the design, their relationships, responsibilities and collaborations
  - ✓ It is not the design or implementation
  - ✓ A pattern is like a template
  - ✓ Provides an abstract description of a design problem and how a general arrangement of elements solves it

- **Consequences**
  - ✓ Results and trade-offs of applying the pattern
5. **Describing Design Patterns [Gamma et al.]**

- **Pattern Name and Classification**
  
  ✓ Important because it becomes part of your design vocabulary.

- **Intent**

  ✓ What does the design pattern do?
  ✓ What is its rationale and intent?
  ✓ What particular design issue or problem does it address?

- **Also Known As**

  ✓ Other well-known names for the pattern, if any.

- **Motivation**

  ✓ A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem.

- **Applicability**

  ✓ When should the pattern be applied?
✓ How can you recognize these situations?

• **Structure**
  ✓ A graphical representation of the classes in the pattern using a notation such as Object Modeling Technique (OMT) or UML to illustrate sequences of requests and collaborations.

• **Participants**
  ✓ The classes and/or objects participating in the design pattern and their responsibilities.

• **Collaborations**
  ✓ How the participants collaborate to carry out their responsibilities.

• **Consequences**
  ✓ How does the pattern support its objectives?
  ✓ What are the trade-offs and results of using the pattern?
  ✓ What aspect of system structure does it let you vary independently?

• **Implementation**
  ✓ What pitfalls, hints, or techniques should you be aware of when implementing the pattern?
  ✓ Are there language-specific issues?
• Sample Code

✓ Code fragments that illustrate how you might implement the pattern in a particular object-oriented language.

• Known Uses

✓ Examples of the pattern found in real systems.

• Related Patterns

✓ What design patterns are closely related to this one?
✓ What are the important differences?
✓ With which other patterns should this one be used?

6. Design Pattern Catalog

• The design pattern catalog contains 23 design patterns, which are distributed over three categories:

  1. **Creational patterns:** These concern the process of object creation.
  2. **Structural patterns:** These deal with the composition of classes or objects.
  3. **Behavioral patterns:** These characterize the ways in which classes or objects interact and distribute responsibility.
This categorization is further refined by specifying the scope of a pattern. It can be either class or object dependent on which a pattern applies primarily. The design pattern space is shown in the following table:

**Table: Design pattern space.**

<table>
<thead>
<tr>
<th>Scope</th>
<th>Class</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Factory Method</td>
<td>Creator</td>
</tr>
<tr>
<td></td>
<td>Adapter (class)</td>
<td>Structural</td>
</tr>
<tr>
<td></td>
<td>Interpreter</td>
<td>Behavioral</td>
</tr>
<tr>
<td>Object</td>
<td>Abstract Factory</td>
<td>Template Method</td>
</tr>
<tr>
<td></td>
<td>Adapter (object)</td>
<td>Chain of Responsibility</td>
</tr>
<tr>
<td></td>
<td>Chain of Responsibility</td>
<td>Command</td>
</tr>
<tr>
<td></td>
<td>Command</td>
<td>Iterator</td>
</tr>
<tr>
<td></td>
<td>Iterator</td>
<td>Mediator</td>
</tr>
<tr>
<td></td>
<td>Mediator</td>
<td>Memento</td>
</tr>
<tr>
<td></td>
<td>Memento</td>
<td>Observer</td>
</tr>
<tr>
<td></td>
<td>Observer</td>
<td>State</td>
</tr>
<tr>
<td></td>
<td>State</td>
<td>Strategy</td>
</tr>
<tr>
<td></td>
<td>Strategy</td>
<td>Visitor</td>
</tr>
</tbody>
</table>
7. **Examples of design patterns include the following**

7.1. **Singleton**

- Make sure that there is a single instance of a class in the whole application.
- Have an encapsulated static variable holding the single instance of a class.
- Provide a static get-operation that creates the single instance once and returns it from then on.

```
public class Singleton {
    private static Singleton instance = null;
    public static Singleton getSingleton() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

- **Java Code (Use static member functions for C++):**

```
public class Singleton {
    private static Singleton instance = null;
    public static Singleton getSingleton() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```
7.2. Adapter (or Wrapper) pattern.

- **Intent**: adapt the interface of one existing type of object to have the same interface as a different existing type of object.

- **Example**: [http://www.cs.wm.edu/~noonan/java/stackwrap/](http://www.cs.wm.edu/~noonan/java/stackwrap/)
  - A Stack class implemented using the Adapter Pattern.
  - The class makes use of the functionality of the existing **Vector** class, wrapping or adapting it to implement a stack.
  - This has the advantage of adding lots of extra useful methods at little cost, like `toString` and `elements`. 
o **Code:**

```java
import java.util.*;

public class Stack {

    private Vector stack = new Vector();

    public boolean empty() { return stack.size() == 0; }

    public Object top() { return stack.lastElement(); }

    public Object pop() {
        Object rslt = stack.lastElement();
        stack.removeElementAt(stack.size() - 1);
        return rslt;
    }

    void push(Object v) { stack.addElement(v); }

    public String toString() { return stack.toString(); }

    public Enumeration elements() { return stack.elements(); }
}
```
7.3. Facade Design Pattern

- When designing good programs, programmers usually attempt to avoid excess coupling between module/classes.
- It decouples client from complex system.
- It makes the task of accessing a large number of modules much simpler by providing an additional interface layer.
- Using this pattern helps to simplify much of the interfacing that makes large amounts of coupling complex to use and difficult to understand: It hides the complexities of the system and provides an interface to the client from where the client can access the system.
- For example, the Java interface JDBC can be called a facade.
- Facade: A single class to access a collection of classes.
• Classes:
  o One client class
  o The facade class: it has a little of code to call lower layers most of the time.
  o The classes underneath the facade.

• Advantages/Disadvantages
  o Decouple the interfacing between many modules or classes.
  o One possible disadvantage to this pattern is that you may lose some functionality contained in the lower level of classes, but this depends on how the facade was designed.

• Example: http://sourcemaking.com/design_patterns/facade
  o Consumers encounter a Facade when ordering from a catalog. The consumer calls one number and speaks with a real or virtual customer service representative. The customer service representative acts as a Facade, providing an interface to the order fulfillment department, the billing department, and the shipping department.
• Example: (http://en.wikipedia.org/wiki/Façade_pattern)

- Façade: interacts Packages 1, 2, and 3 with the rest of the application.
- Clients: The objects using the Facade Pattern to access resources from the Packages.
- Packages: Software library / API collection accessed through the Facade Class.

- Java Example:
  - The following example hides the parts of a complicated calendar API behind a more user friendly facade. The output is:

```java
import java.text.*;

class1 c1 = new Class1();
class2 c2 = new Class2();
class3 c3 = new Class3();
c1.doStuff(c2);
c3.setX(c1.getX());
return c3.getY();
```
import java.util.*;

/** "Facade" /**
class UserfriendlyDate
{
    Calendar cal = Calendar.getInstance();
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");

    public UserfriendlyDate (String isodate_ymd) throws ParseException {
        Date date = sdf.parse(isodate_ymd);
        cal.setTime(date);
    }

    public void addDays (int days) {
        cal.add (Calendar.DAY_OF_MONTH, days);
    }

    public String toString() {
        return sdf.format(cal.getTime());
    }
}

/** "Client" /**
class FacadePattern
{
    public static void main(String[] args) throws ParseException {
        UserfriendlyDate d = new UserfriendlyDate("1980-08-20");
        System.out.println("Date: " + d.toString());
        d.addDays(20);
        System.out.println("20 days after: " + d.toString());
    }
}
7.4. Bridge (Also known as Handle/Body)

- **Intent:** Decouple an abstraction from its implementation so that the two can vary independently. A Bridge pattern is a design time Design Pattern.

- **Motivation:**
  - When an abstraction can have several implementations inheritance is used to accommodate them.
  - But inheritance binds an implementation to the abstraction permanently, hence its difficult to modify, extend and reuse abstraction and implementations *independently*.

- **Use** the Bridge pattern when:
  - You want run-time binding of the implementation
  - You have a proliferation of classes resulting from a coupled interface and numerous implementations
  - You want to share an implementation among multiple objects

- **Structure:** *Example*
• **Client**  
  o The Object using the bridge pattern

• **Abstraction**  
  o Defines the abstraction's interface.  
  o Maintains a reference to an object to type Implementor.

• **Refined Abstraction**  
  o Extends the interface defines by Abstraction.

• **Implementor**  
  o Defines the interface for implementation classes. This interface doesn't have to  
    correspond exactly to Abstraction’s interface; in fact the two interfaces can be quite different.  
    Typically the Implementation interface provides only primitive operations, and  
    Abstraction defines higher-level operations based on these primitives.

• **Concrete Implementor**  
  o It Implements the Implementor interface and  
    defines its concrete implementation.
7.5. Strategy (or Policy) Pattern

- The goal of this pattern is to allow any one of a family of related algorithms to be easily substituted in a system.
- A client object can access the different algorithms through a single class.
- Strategy contains a library of algorithms and makes code reuse easier.
- The following is a class diagram of the Strategy design pattern (Kremer 1998).

```
Context
  contextInterface()

Strategy
  algorithmInterface()

ConcreteStrategyA
  algorithmInterface()

ConcreteStrategyB
  algorithmInterface()

ConcreteStrategyC
  algorithmInterface()
```

- Participants:
  - The **Context** class;
    - It creates and maintains a reference to a Concrete Strategy object, via a Strategy object.
  - The **Strategy** class:
    - An **abstract class** or possibly an interface in Java, Strategy provides the common programming interface for all of the algorithms.
• It is the parent class for all the different algorithms to be encapsulated.
• It provides an interface for client objects to access the different algorithms, which are implemented as the different ConcreteStrategy classes.
  o The **ConcreteStrategy** classes:
    • They contain the actual implementation of the different algorithms.
    • They **implements** the interface defined by the Strategy class.
  o The **Client** class:
    • The user of the Strategy often knows about and selects the Concrete Strategy.
    • A client object can choose to use one of the ConcreteStrategy algorithms at run time. For example, if a client object want to use an algorithm in ConcreteStrategyA class, it can use the following C++ syntax to instantiate it:

```
Strategy *a = new Strategy(new ConcreteStrategyA());
```

• Strategy is a good pattern for experimenting different algorithms. Algorithms can easily be added or removed from the Strategy class.
• Examples:
  o Saving files in different formats: Word, ODT, RTF, HTML plain text, Excel, CSV, HTML.
  o Compress files using different compression algorithms.
- Capture video using different video compression algorithms.
- Plot the same data using different formats (points, line chart, bar chart, etc.)
8. Design Pattern Selection

- Consider how design patterns solve design problems.
  - Find appropriate objects
  - Determine object granularity
  - Specify object interfaces

- Scan Intent sections.
- Study how patterns interrelate.
- Study patterns of like purpose.
  - Study similarities and differences

- Examine a cause of redesign.
  - Consider what might force a change to your design

- Consider what should be variable in your design.
  - Consider what you want to be able to change without redesign - encapsulate the concept that varies
9. **How to Use a Design Pattern?**

- Read the pattern once through for an overview.
  - Pay special attention to the Applicability and Consequences sections
  - Ensure the pattern is right for your problem

- Go back and study the Structure, Participants and Collaborations sections
- Look at the sample code.
- Choose names for pattern participants that are meaningful in the application context.
- Define the classes.
- Define application-specific names for operations in the pattern.
- Implement the operations to carry out the responsibilities and collaborations in the pattern.
10. Idioms

- Definition: "An idiom is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language." [Buschmann]
- An idiom is a low-level abstraction. It is usually a language-specific pattern that deals with some aspects of both design and implementation.
- In Java, the language-specific iterator defined to implement the java.util Enumeration interface can be considered an idiom. It is a language-specific instance of the more general Iterator design pattern.
- Example 1:

```java
void cancelAll(Collection c) {
    for (Iterator i = c. Iterator(); i.hasNext(); ) {
        TimerTask tt = (TimerTask) i.next();
        tt.cancel();
    }
}
```

- Example 2: A better implementation in Java 5.0 uses the for-each construct:

```java
void cancelAll(Collection<TimerTask> c) {
    for (TimerTask t : c)
        t.cancel();
}
```
11. Summary

- Definitions
- What is a Pattern?
- Categories of Patterns
  - Pattern Characteristics
  - Essential Elements of a Design Pattern
  - Examples of design patterns:
    - Singleton
    - Adapter
    - Facade
    - Bridge
    - Strategy
  - Design Pattern Selection
  - How to Use a Design Pattern?
  - Idioms