Software Paradigms

Software Paradigms (Lesson 12)

Design Patterns

Table of Contents

1 Introduction.................................................................................................................. 3
  1.1 What is a Design Pattern? .................................................................................... 3
  1.2 Object-Oriented Design Patterns ....................................................................... 3
  1.3 Basic Elements of a Design Pattern ..................................................................... 4
  1.4 Describing Design Patterns .................................................................................. 4
  1.5 Classification of Design Patterns ......................................................................... 4

2 Examples of Design Patterns ....................................................................................... 6
  2.1 Creational Design Patterns .................................................................................. 6
  2.2 Creational Design Pattern Example: Abstract Factory ....................................... 6
    2.2.1 Intent.............................................................................................................. 6
    2.2.2 Motivation ...................................................................................................... 6
    2.2.3 Applicability ................................................................................................... 7
    2.2.4 Structure ......................................................................................................... 7
    2.2.5 Consequences ................................................................................................ 8
    2.2.6 Implementation ............................................................................................ 8
    2.2.7 Related patterns ............................................................................................ 9
  2.3 Structural Design Patterns ..................................................................................... 9
  2.4 Structural Design Pattern Example: Composite .................................................. 10
    2.4.1 Intent.............................................................................................................. 10
    2.4.2 Motivation ...................................................................................................... 10
    2.4.3 Applicability ................................................................................................... 11
    2.4.4 Structure ......................................................................................................... 11
    2.4.5 Consequences ................................................................................................ 12
    2.4.6 Implementation ............................................................................................ 12
    2.4.7 Related Patterns ............................................................................................ 13
  2.5 Behavioural Design Patterns .................................................................................. 13
  2.6 Behavioural Design Pattern Example: Visitor .................................................... 14
    2.6.1 Intent.............................................................................................................. 14
    2.6.2 Motivation ...................................................................................................... 14
    2.6.3 Applicability ................................................................................................... 15
    2.6.4 Structure......................................................................................................... 15
Software Paradigms

2.6.5 Consequences ........................................................................................................ 16
2.6.6 Implementation ........................................................................................................ 16
2.6.7 Related Patterns ....................................................................................................... 17

3 Properties of Design Patterns ...................................................................................... 18
  3.1 Solving Design Problems with Design Patterns .......................................................... 18
    3.1.1 Finding Appropriate Objects ................................................................................ 18
    3.1.2 Determining Object Granularity .......................................................................... 18
    3.1.3 Specifying Object Interfaces ............................................................................... 18
    3.1.4 Designing for Change ......................................................................................... 18
  3.2 Some Advantages of Design Patterns ....................................................................... 19
    3.2.1 Common Design Vocabulary .............................................................................. 19
    3.2.2 A Documentation Aid ......................................................................................... 19
1 Introduction

1.1 What is a Design Pattern?

As we build more complex computer systems, we face problems of construction rather than problems of analysis. The problems of construction are solved by designing programming solutions for such problems in the context of the computer application we are trying to build.

Some constructional problems occur over and over again across a wide range of different computer applications. Obviously, we can design a generic solution to such repeating problems, and then try to adjust such a generic solution to the specific need of the application we are currently building. Such generic solutions are usually referred to as design patterns.

Christopher Alexander first introduced patterns to encode knowledge and experience in designing buildings. He said: “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 many times over, without ever doing it the same way twice”.

Thus, a design pattern is the core of a solution to a problem in context. The solution can be applied in different situations, and has to be adapted to fit the needs of the specific situation.

1.2 Object-Oriented Design Patterns

Design patterns in programming were first introduced by the Gang of Four (Gamma, Helm, Johnson, Vlissides). They referred to design patterns always as to object-oriented design patterns, i.e., design patterns that occur in building object-oriented computer systems.

Thus, object-oriented design patterns might be defined as descriptions of communicating objects and classes that are customized to solve a general (object-oriented) design problem in a particular context.

They also stated that an object-oriented design pattern is not:

- A primitive building block, such as linked lists, or hash-tables that can be encoded in classes and reused in a wide range of applications
- A complex, domain specific design for an entire application or subsystem.

Rather, an object-oriented design pattern just describes a recurring object-oriented design structure.

Thus, an object-oriented design pattern:

- Names, abstracts, and identifies the key aspect (classes, objects and their relations) of a common design structure
- Creates a reusable object-oriented design
- Focuses on a particular object-oriented design by describing when it applies, whether it can be applies in view of other design constraints, and the consequences and trade-offs of its use
- Implements the design idea in an object-oriented programming language.

In the rest of this paper we will examine solely object-oriented design patterns. Thus, whenever we say a design pattern we actually refer to an object-oriented design pattern.
1.3 Basic Elements of a Design Pattern

Each pattern has four essential elements:

- The pattern name is a handle we can use to describe a design problem, its solutions and consequences in a word or two. Naming a pattern immediately increases the design vocabulary of programmers. Having a vocabulary for patterns enables to talk about patterns with other programmers, in the documentation, etc.

- The problem describes when to apply the pattern. It explains the problem and its context. It might describe specific design problems such as how to represent algorithms as objects. It might describe class or object structures that are symptomatic of an inflexible design. Sometimes the problem includes also a list of conditions that must be met before it makes sense to apply the pattern.

- The solution describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution doesn’t describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations. Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects) solves it.

- The consequences are the results and trade-offs of applying the pattern. They may address language and implementation issues as well. Since reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system’s flexibility, extensibility, or portability.

1.4 Describing Design Patterns

The Gang of Four used a consistent format to describe patterns. They developed a template for describing a design pattern. The template lent a uniform structure to the information and made design patterns easier to learn, compare and use. This template describes a design pattern with:

- Pattern name and classification (according to the classification schema, see the next section)

- Intent of the design pattern

- Motivation; describes a typical scenario that illustrates the design problem

- Applicability; when we can apply the design pattern

- Structure; a graphical representation of the classes and their collaborations in the pattern

- Consequences of the design pattern

- Implementation and the sample code

- Related patterns.

We will apply this template to show few examples of design patterns in the next chapter.

1.5 Classification of Design Patterns

Design patterns can be classified by two criteria:

- Purpose, which reflects what a pattern does. Patterns can have creational, structural or behavioural purpose;

  o Creational patterns concern the process of object creation
Software Paradigms

- Structural patterns deal with the composition of classes and objects
- Behavioural patterns characterize the ways in which classes and objects interact and distribute responsibility.

- **Scope**, which specifies whether the pattern applies primarily to classes or to objects:
  - Class patterns deal with relationships between classes and their subclasses. These relationships are established through inheritance, so they are static.
  - Object patterns deal with object relationships, which can be changed at run-time and are more dynamic.

The following table shows the classification of design patterns. Note, that those are the patterns introduced by the Gang of Four.

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Design Patterns</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creational</td>
<td>Abstract Factory</td>
<td>Object</td>
</tr>
<tr>
<td></td>
<td>Builder</td>
<td>Object</td>
</tr>
<tr>
<td></td>
<td>Factory Method</td>
<td>Class</td>
</tr>
<tr>
<td></td>
<td>Prototype</td>
<td>Object</td>
</tr>
<tr>
<td></td>
<td>Singleton</td>
<td>Object</td>
</tr>
<tr>
<td>Structural</td>
<td>Adapter</td>
<td>Class</td>
</tr>
<tr>
<td></td>
<td>Bridge</td>
<td>Object</td>
</tr>
<tr>
<td></td>
<td>Composite</td>
<td>Object</td>
</tr>
<tr>
<td></td>
<td>Decorator</td>
<td>Object</td>
</tr>
<tr>
<td></td>
<td>Facade</td>
<td>Object</td>
</tr>
<tr>
<td></td>
<td>Flyweight</td>
<td>Object</td>
</tr>
<tr>
<td></td>
<td>Proxy</td>
<td>Object</td>
</tr>
<tr>
<td>Behavioral</td>
<td>Chain of Responsibility</td>
<td>Object</td>
</tr>
<tr>
<td></td>
<td>Command</td>
<td>Object</td>
</tr>
<tr>
<td></td>
<td>Interpreter</td>
<td>Class</td>
</tr>
<tr>
<td></td>
<td>Iterator</td>
<td>Object</td>
</tr>
<tr>
<td></td>
<td>Mediator</td>
<td>Object</td>
</tr>
<tr>
<td></td>
<td>Memento</td>
<td>Object</td>
</tr>
<tr>
<td></td>
<td>Observer</td>
<td>Object</td>
</tr>
<tr>
<td></td>
<td>State</td>
<td>Object</td>
</tr>
<tr>
<td></td>
<td>Strategy</td>
<td>Object</td>
</tr>
<tr>
<td></td>
<td>Template Method</td>
<td>Class</td>
</tr>
<tr>
<td></td>
<td>Visitor</td>
<td>Object</td>
</tr>
</tbody>
</table>
2 Examples of Design Patterns

2.1 Creational Design Patterns

Creational design patterns abstract the instantiation process. They help make a system independent of how its objects are created, composed, and represented.

A class creational pattern uses inheritance to vary the class that’s instantiated, whereas an object creational pattern will delegate instantiation to another object.

The creational design patterns allow configuring of a software system as a system with “product” objects that vary widely in structure and functionality. Such configuration can be static, i.e., specified at compile-time, or dynamic, i.e., specified at run-time.

2.2 Creational Design Pattern Example: Abstract Factory

2.2.1 Intent

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

2.2.2 Motivation

Consider a toolkit to allow applications with GUI interfaces to run on multiple platforms (Mac, PC, OS/2, Unix Motif) using the look and feel of each platform. To be portable across look-and-feel standards, an application should not hard-code its widgets for a particular look and feel. Instantiating look-and-feel-specific classes of widgets throughout the application makes it hard to change look and feel later.

To solve this problem we can define an abstract WidgetFactory class that declares an interface for each basic kind of widget. There is also an abstract class for each kind of widget, and concrete subclasses implement widgets for specific look-and-feel standards. WidgetFactory’s interface has an operation that returns a new widget object for each abstract widget class. Clients call these operations to obtain widget instances, but clients are not aware of the concrete classes they are using. Thus, clients stay independent of the prevailing look and feel. There is a concrete subclass of WidgetFactory for each look-and-feel standard. Each subclass implements the operations to create the appropriate widget for the look and feel.

![Diagram](Image)

Figure 1 Abstract MenuWidget class and its concrete subclasses
2.2.3 **Applicability**

We use the Abstract Factory design pattern if:

- A system should be independent of how its products are created, composed, and represented
- A system should be configured with one of multiple families of products
- A family of related product objects is designed to be used together, and you need to enforce this constraint
- You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

2.2.4 **Structure**

Participants of this design pattern are as follows:

- AbstractFactory (e.g. WidgetFactory), which declares an interface for operations that create abstract product objects
Software Paradigms

- ConcreteFactory (e.g. MotifWidgetFactory, MacWidgetFactory, etc.), which implements the operations to create concrete product objects
- AbstractProduct (e.g. Menu, Window), which declares an interface for a type of product object.
- ConcreteProduct (e.g. MacWindow, Win95Window, etc.), which defines a product object to be created by the corresponding concrete factory by implementing the AbstractProduct interface.
- Client, which uses only interface, declared by AbstractFactory and AbstractProduct classes.

Usually, a single instance of a ConcreteFactory class is created at run-time. This concrete factory creates product objects having a particular implementation. To create different product objects, clients should use a different concrete factory.

2.2.5 Consequences

The Abstract Factory pattern has the following consequences:

- Isolates concrete classes by helping programmers to control the classes of objects that an application creates.
- Makes exchanging product families easy because the class of a concrete factory appears only once in an application, at the place of its instantiation. This makes it easy to change the concrete factory that an application uses.
- Promotes consistency among products by enforcing to use objects from the same family of objects.
- Supporting new kinds (in each family) of products is difficult since we need not only to define new product objects but also to extend all factories to be able to create those new product objects.

2.2.6 Implementation

Here is a simple implementation in Java of the above example.

```java
abstract class WidgetFactory{
    public Window createWindow();
    public Menu createMenu();
    public Button createButton();
}

class MacWidgetFactory extends WidgetFactory{
    public Window createWindow()
    { code to create a mac window }

    public Menu createMenu()
    { code to create a mac Menu }

    public Button createButton()
```
Here is the code from a client. We just need to ensure that the application creates a proper ConcreteFactory object.

```java
public void installDisneyMenu(WidgetFactory myFactory){
    Menu disney = myFactory.createMenu();
    disney.addItem( "Disney World" );
    disney.addItem( "Donald Duck" );
    disney.addItem( "Mickey Mouse" );
    disney.addGrayBar( );
    disney.addItem( "Minnie Mouse" );
    disney.addItem( "Pluto" );
    etc.
}
```

### 2.2.7 Related patterns

AbstractFactory classes are often implemented with factory methods (Factory Method pattern), but they can be also implemented using Prototype design pattern. A concrete factory is often a singleton (Singleton design pattern).

### 2.3 Structural Design Patterns

Structural patterns are concerned with how classes and objects are composed to form larger structures.

Structural class patterns use inheritance to compose interfaces or implementations. For example, multiple inheritance can be seen as a kind of structural design patterns, since it uses inheritance to mix two or more classes into a new one.

Rather than composing interfaces or implementations, structural object patterns describe ways to compose objects to realize new functionality. The added flexibility of object composition comes from the ability to change the composition at run-time, which is impossible with static class composition.
2.4 Structural Design Pattern Example: Composite

2.4.1 Intent

Composite design pattern composes objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

2.4.2 Motivation

Abstract Window Toolkits (AWT) lets user build complex graphical user interfaces. The user can group a number of simple user interface elements (such as, radio buttons, menu items, etc.) to form larger components (such as radio button group, menu, etc.). A simple implementation of an AWT could implement classes for user interface primitives and other classes that act as containers for these primitives. But there is a problem with this approach: Code that uses these classes must treat primitive and container objects differently, even if the most of the time the user treats them identically. Having to distinguish these objects makes the application unnecessarily more complex. The Composite pattern describes how to use recursive composition so that clients don’t have to make this distinction.

![Figure 4 Typical Abstract Window Toolkit](image)

The key to the Composite pattern is an abstract class that represents both primitives and their containers.

![Figure 5 Application of the Composite pattern for an AWT](image)

For the Abstract Window Toolkit this class is Component class. Component declares operations like Draw, Show, etc. that are specific for all graphic user interface elements. It also declares operations that all composite objects share, such as operations for accessing and managing its children.
For example, the standard Java AWT uses the Composite pattern.

![Figure 6 Standard Java AWT](image)

2.4.3 Applicability

Use the Composite pattern when:
- You want to represent part-whole hierarchies of objects
- You want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

2.4.4 Structure

![Figure 7 Structure of the Composite pattern](image)

Participants of the Composite pattern are as follows:
Component (Component), which declares the interface for objects in the composition. It also implements the default behavior for the interface common to all classes. Further, it should declare an interface for accessing and managing its child components.

Leaf (Button, Menu, etc.), which represents leaf objects in the composition. A leaf has no children.

Composite (WidgetContainer), which defines behavior for components having children. At run-time the Container stores child components. Container also implements child-related operation in the Component interface.

Client, which manipulates objects in the composition through the uniform Component interface.

2.4.5 Consequences

The Composite pattern has the following consequences:

- Makes the client simple, since clients treat composite structure and individual objects uniformly.
- Makes it easier to add new kinds of components.
- Makes it hard to restrict the components of a composite. Sometimes you want a composite to have only certain components. However, with the Composite pattern you have to implement run-time checks to ensure this restriction.

2.4.6 Implementation

Here is a simple implementation of the above example.

abstract class Component{
    private Component parent;
    
    abstract public void update();
    abstract public void add();
    public void setParent(Component par){
        parent = par;
    }
}

class WidgetContainer extends Component{
    Component[] myComponents;
    
    public void update(){
        if ( myComponents != null )
        {
            for ( int k = 0; k < myComponents.length(); k++ )
            {
            }
        }
Software Paradigms

```java
myComponents[k].update();
}
}
}

public add( Component aComponent ){
    myComponents.append( aComponent );
aComponent.setParent( this );
}
}

class Button extends Component{
    public void update(){
        etc.
    }
    etc.
}

2.4.7 Related Patterns

Decorator pattern is often used with Composite. When decorators and composites are used together, they will usually have a common parent class. So decorators will have to support the Component interface with operations like Add, Remove, and GetChild.

Iterator pattern can be used to traverse composites.

Visitor pattern localizes operations and behaviour that would otherwise be distributed across Composite and Leaf classes.

2.5 Behavioural Design Patterns

Behavioural patterns are concerned with algorithms and the assignment of responsibility between objects. Behavioural patterns describe not just patterns of objects or classes but also the patterns of communication between them. These patterns characterize complex control flow that is difficult to follow at run-time. They shift your focus away from flow of control to let you concentrate just on the way objects are interconnected.

Behavioural class patterns use inheritance to distribute behaviour between classes.

Behavioural object patterns use object composition rather than inheritance. For example, a behavioural object pattern can describe how a group of object might cooperate to perform a task that no single object can carry out by itself. A typical example is the Observer pattern from the Smalltalk (Model/View/Controller paradigm). Views are used to show the state of data (contained in Model) and they are observers of this data. Whenever a model changes its state all views are notified and they can update the representation of the data in views.
2.6 Behavioural Design Pattern Example: Visitor

2.6.1 Intent

Visitor design pattern represents an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing classes of the elements on which it operates.

2.6.2 Motivation

Consider the following implementation of integer lists, written in Java.

```java
interface List {}

class Nil implements List {}

class Cons implements List{
    int head;
    List tail;
}
```

Suppose, that we need to write a program, which computes the sum of all components of a given List-object. The first obvious implementation would be to check for each object its exact type (in Java we can do this with the `instanceof` operator). Then if the object is of type Cons-object than the fields are accessed via type caste and the loop is repeated, otherwise if the type is Nil-object we just would proceed to the next object.

The advantage of this code, which is that it can be written without touching the classes Nil and Cons.

However, the drawback is that the code constantly uses type casts and `instanceof` to determine what class of objects it is considering.

Another possible implementation is to define an abstract method in the List interface, e.g. `sum` method. Then all subclasses, i.e., Cons and Nil implement the sum method to provide the exact sum of its elements. Thus, Mil class returns 0, whereas Cons class iterates through its objects.

The advantage of this code is that the type casts and `instanceof` operations have disappeared.

The disadvantage is that every time we want to perform a new operation on List-objects, then new-dedicated methods have to be written for all the classes, and the classes must be recompiled.

Finally, let us consider it as a Visitor pattern. We can implement by packaging related operations from each class in a separate object, called a `visitor`, and passing it to elements of the List object. When an element of the List “accepts” the visitor, it sends a request to the visitor that encodes the element’s class. It also includes this element as an argument. The visitor will execute the operation for that element and compute the correct sum for that element.

The advantage is that one can write code that manipulates objects of existing classes without recompiling those classes.

The price is that all objects must have an accept method.
2.6.3 Applicability

Use the Visitor pattern when:
- An object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.
- Many distinct and unrelated operations need to be performed on objects in an object structure.
- The classes defining the object structure rarely change, but you often want to define new operations over the structure.

2.6.4 Structure

Figure 8 Structure of the Visitor pattern

The participants of the Visitor pattern are as follows:
- Visitor (ListVisitor), which declares a Visit operation for each class of ConcreteElement in the object structure.
- ConcreteVisitor (SumVisitor), which implements each operation declared by Visitor.
- Element (ListElement), which defines an Accept operation that takes a visitor as an argument.
- ConcreteElement (Cons, Nil), which implements an Accept operation that takes a visitor as an argument.
- ObjectStructure (List), which can enumerate its elements.

Thus, when an element is visited, it calls the Visitor operation that corresponds to its class. The element supplies itself as an argument to this operation so let the visitor access its state, if necessary.
2.6.5 Consequences

The consequences of the Visitor pattern are as follows:

- Visitor makes adding new operations easy.
- A visitor gathers related operations and separates unrelated ones.
- Adding new ConcreteElement class is hard, since each new ConcreteElement gives rise to a new abstract operation on Visitor and corresponding implementation in every ConcreteVisitor class.
- Visiting is possible across class hierarchies.
- Visitor sometimes breaks the encapsulation of a ConcreteElement class, since we need to provide public operations that let Visitor check the internal state of the ConcreteElement.

2.6.6 Implementation

Here is a simple implementation of the above example.

```java
interface List{
    void accept(Visitor v);
}

class Nil implements List{
    public void accept(Visitor v)
    {
        v.visitNil(this);
    }
}

class Cons implements List{
    int head;
    List tail;
    public void accept(Visitor v){
        v.visitCons(this);
    }
}

interface Visitor{
    void visitNil(Nil x);
    void visitCons(Cons x);
}
```
class SumVisitor implements Visitor{
    int sum = 0;
    public void visitNil(Nil x) {}
    public void visitCons(Cons x){
        sum = sum + x.head;
        x.tail.accept(this);
    }
}

2.6.7 Related Patterns
Patterns related to the Visitor pattern are the Composite pattern because Visitors are often used to apply an operation over an object structure defined by the Composite pattern.
3 Properties of Design Patterns

3.1 Solving Design Problems with Design Patterns

3.1.1 Finding Appropriate Objects

Object-oriented programs are made up of objects. The hard part about object-oriented design is decomposing a system into objects. Design patterns can help in this process by identifying less obvious abstractions and the objects that capture them.

For example, objects that represent a process or algorithm don’t occur in nature, but they can be crucial in a flexible design. The Strategy pattern describes how to implement families of algorithms to solve a particular problem. Those algorithms can be interchanged at run-time, since they are objects now and are subject to polymorphism for example.

3.1.2 Determining Object Granularity

Usually, objects vary in size and number. They can represent everything down to the hardware or all the way up to entire applications.

Design patterns can help to determine proper object granularity. For example, the Facade pattern describes how to represent complete subsystems as objects, and the Flyweight pattern describes how to support huge numbers of objects at the finest granularities.

A number of other patterns, such as Composite, describe how to decompose an object into smaller objects.

3.1.3 Specifying Object Interfaces

Design patterns help programmers to define interfaces by identifying their key elements and the kind of data that get sent across an interface. A design pattern can also tell what not to put in the interface.

For example, the Memento pattern describes how to encapsulate and save the internal state of an object so that the object can be restored to that state later. The pattern stipulates that Memento objects must define two interfaces:

- A restricted one that lets clients hold and copy mementos
- A privileged one that only the original objects can use to store and retrieve state in the memento.

3.1.4 Designing for Change

Designing a system that is robust to changes is a rather hard task to do. However, a design that doesn’t take changes into account risks major redesigns in the future. Design patterns can ensure that a system can change in specific ways.

Each design pattern lets some aspect of the system structure vary independently of other aspects, thereby making a system more robust to a particular kind of change.

Let us look on some examples:

- Creating an object by specifying a class explicitly commits to a particular implementation instead of a particular interface. That means if we in the future want to change the object...
that we use we need also to implement the client code again. On the other hand using the
design patterns, such as Abstract Factory pattern lets you avoid this problem.

- Dependence on hardware and software platform. Clients that know how an object is
  represented, stored, located, or implemented might need to be changed when the object
  changes. Hiding this information from clients keeps changes from cascading. An example
  is again Abstract Factory used to create different look-and-feel components across
different operating systems.

3.2 Some Advantages of Design Patterns

3.2.1 Common Design Vocabulary

Computer scientists and programmers create names for algorithms and data structures. Similarly,
design patterns should provide a common vocabulary for designers. Such vocabulary can be used
to communicate, document and explore design decisions.

Design patterns make a system less complex, since we can talk about it in the terms form
different design patterns rather then in terms of programming languages.

3.2.2 A Documentation Aid

Describing a system by applying a common design vocabulary makes this system easier to
understand. Having such a common vocabulary means you don't have to describe the whole
design pattern; programmers just can name it and expect that the reader already knows what a
specific pattern means. Therefore, writing the documentation can become much more interesting
and easier☺