ISSN No. 0976-5697

Volume 2, No. 1, Jan-Feb 2011

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

Reducing Efforts in Refactoring Process Using Design Patterns

Dharmendra Pathak*
School for Computer Science & IT
Devi AhilyaVishwavidyalaya
Indore (M.P.), India
dharam_241086@yahoo.co.in

Ugrasen Suman
School for Computer Science & IT
Devi AhilyaVishwavidyalaya
Indore (M.P.), India
ugrasen123@yahoo.com

Abstract: Refactoring is required to improve the overall design of software code, which provides better stability and more efficiency to particular software. Design patterns are useful to solve some common software development problems within a particular context. In this paper, we will describe the comparative introduction of various design patterns and refactoring techniques used in object oriented paradigm. Thereafter, we will consolidate these two concepts and observe the impact of design patterns in refactoring with the efficient implementation using appropriate examples. The proposed research can help to reduce efforts required in refactoring by a large extent.

Keywords: Design Patterns, Refactoring, Factory Pattern, Adapter Pattern, Bridge Pattern, Extract method, Template method, Move method.

I. INTRODUCTION

Design patterns are part of the cutting edge of object-oriented technology. Design patterns represent solutions to problems that arise when developing software within a particular context. Patterns capture the static and dynamic structure and collaboration among key participants in software designs. Patterns facilitate reuse of successful software architectures and designs. Design patterns provide a higher perspective on analysis and design [5].

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code. It improves the internal structure of code. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence refactoring improves the design of the code after it has been written. With refactoring unstructured design converted into well-designed code [1].

Refactoring is performed after the development of software, which takes extra time and cost the design patterns are implemented during developing software. Refactoring cost and time can be reduced at a large extent if proper design patterns were implemented at the time of software development. Further, we have observed the impact of various design patterns on different refactoring methods.

The rest of the paper is categorized into four main sections. The Section I introduces the basic concept of design patterns and refactoring. The consolidation of design patterns and refactoring techniques, and the impact of various design patterns over different refactoring methods with examples are discussed in Section II. The Section III summarizes the paper and presents the future research work.

II. CONSOLIDATION OF DESIGN PATTERNS AND REFACTORING

There are various design patterns such as facade, adapter, bridge, strategy, abstract factory, singleton pattern etc. The common attributes of these patterns are intent, problem, solution, participation and collaborators, consequences and implementation. Also, there are various refactoring methods such as extract class, move method, substitute algorithm method, replace method with method object, local extension method, replace constructor with factory method etc. We will discuss the usefulness of various design patterns to reduce the efforts required in refactoring process.

A. *Facade Pattern*

Facade pattern can be used to create a simpler interface in terms of method calls as well as to reduce the number of objects that a client object must deal with [5]. The Facade pattern can be characterized as follows:

[a] Intent: It is used to simplify how to use an existing system to define our own interface.
[b] Problem: Problem is to use only a subset of a complex system and to interact with the system in a particular way.
[c] Solution: The Facade presents a new interface for the client of the existing system to use.
[d] Participants and Collaborators: It presents a simplified interface to the client that makes it easier to use.
[e] Consequences: Since the Facade is not complete, certain functionality may be unavailable to the client.
[f] Implementation: Define a new class (or classes) that have the required interface. Have this new class use the existing system [2].

We are required to use all the functionality of a complex system and can create a new class that contains all the rules for accessing that system [6].

B. *Impact on Extract Class Method*

Extract method is used to decompose the complex classes into simpler classes. In this method of refactoring, new class is created and relevant fields and methods from the old class can be moved into the new class. The implementing Facade pattern instead of Extract class refactoring method can reduce effort required for software design.

C. *Implementation*

A new class is created to express the split-off responsibilities and then move methods from old to new class. We start with lower-level methods (called rather than calling) and build to the higher level.

Review and reduce the interfaces of each class and decide whether to expose the new class. If we expose the class then decide whether to expose it as a reference object or as an immutable value object [1].

D. Example

Consider the following example that exhibits the impact of design pattern into refactoring method:

[a] Problem: Consider a class Student that has implemented methods such as fetchData() to fetch student records, connectData() to store all those records into databases and etrWindow() method to provide GUI.

[b] Solution provided by Facade pattern: This condition can be removed by placing all three methods to three classes Fetch, Connect and Window. StudentFacade class will contain object of all these classes and let this class to be used in the system (Student class) as follows:

```
class Fetch
{       void fetchData(){//method body }
}
        class Connect
{       void connectData(){//method body}
}
        class Window
{       void etrWindow(){//method body}
}
        class StudentFacade implements interfaces….
{       Fetch f = new Fetch ();
        Connect ct = new Connect ();
        Window w = new Window ();
}
        class Student extends StudentFacade
{       //remaining code
}
```

Now, these methods can be accessed by creating Student class object as follows:

```
Student s =new Student ();
s.etrWindow();
s.fetchData();
s.connectData();
```

[c] Solution provided by refactoring method: Similar solution can be provided by Extract class method, in which Student class is divided into three classes as in Facade Pattern and then create object of these three classes into Student class as follows:

```
class Student implements interfaces….
{       Fetch f = new Fetch ();
        Connect ct = new Connect ();
        Window w = new Window ();
}
```

Thereafter, these methods can be accessed by creating Student class object as follows:

```
Student s =new Student ();
s.etrWindow();
s.fetchData();
s.connectData();
```

Hence, using this example it is observed that Facade pattern can be used to eliminate Extract class method.

E. Adapter Pattern

It is used to convert the interface of a class into another interface that the clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces [5]. Adapter pattern can be characterized as follows:

[a] Intent: Match an existing object beyond control to a particular interface.

[b] Problem: A system has the right data and behavior but the wrong interface. Typically used to make something a derivative of an abstract class.

[c] Solution: The Adapter provides a wrapper with the desired interface.

[d] Participants and Collaborators: The Adapter adapts the interface of an Adaptee to match that of the Adapter's target (the class it derives from). This allows the client to use the Adaptee as if it were a type of target.

[e] Consequences: The Adapter pattern allows for preexisting objects to fit into new class structures without being limited by their interfaces.

[f] Implementation: It contains the existing class in another class that contains the class match the required interface and calls the methods of the contained class [2].

We are required to use all the functionality of a complex system and can create a new class that contains all the rules for accessing that system [6]. In Adapter pattern following functions can be adapted:

[g] Those functions that are implemented in the existing class can be adapted.

[h] Those functions that are not present can be implemented in the wrapping class [3].

F. Impact on Move Refactoring Method

Move method is used when a method or interface is, or will be, using or used by more features of another class than the class on which it is defined. Create a new method or interface with a similar body in the class it uses most. Either the old method or interface can be turned into a simple delegation or it can be removed altogether. By implementing the Adapter Pattern, the efforts required in Move refactoring Method can be reduced at very high extent.

G. Implementation

All features can be examined, which are used by source method/ interface that are defined on the source class. The sub and super classes can be checked in the source class for other declarations of the method/ interface.

The method can be declared in the target class, copy the code from the source method to the target, and then adjust the method/interface to make it work in its new home [1].

H. Example

The following example of image animation can help us to show the impact of design pattern:

[a] Problem: Assume that a class ImageAnimation which is implementing two interfaces animation and image directly, which is not compatible to it.

[b] Solution provided by Adapter pattern: This condition can be removed by placing both the interfaces into a single class ComInter and then ImageAnimation will extend this class as follows:

```
class ComInter implements image, animation
{       void run();
        void animation();
        void takeImage();
}
class ImageAnimation extends ComInter
{       void takeImage()
        {//method body}
        void animation()
        {//method body}
}
```

[c] Solution provided by refactoring method: Similar solution can be provided by Move refactoring method by moving both the incompatible interfaces into a new class ComInter and extending it by ImageAnimation as follows:

```
class ImageAnimation extends ComInter
{       // similar code
}
```

Thus, using this example it is clear that Adapter pattern can be used to eliminate Move method.

I. Strategy Pattern

It can be used to define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

The Strategy pattern is based on a few principles:

[a] Objects have responsibilities.

[b] Different, specific implementations of these responsibilities are manifested through the use of polymorphism.

[c] There is a need to manage several different implementations of what is, conceptually, the same algorithm [3].

The Strategy pattern can be characterized as follows:

[d] Intent: It enables us to use different business rules or algorithms depending on the context in which they occur.

[e] Problem: The selection of an algorithm that needs to be applied depends on the client making the request or the data being acted on.

[f] Solution: Separates the selection of algorithm from the implementation of the algorithm. It allows for the selection to be made based upon context.

[g] Participants and Collaborators: It specifies how the different algorithms are used which is implemented by Concrete strategies.

[h] Consequences: The Strategy pattern defines a family of algorithms. Switches and/or conditionals can be eliminated. We must invoke all algorithms in the same way.

[i] Implementation: It has the class that uses the algorithm (Context) contains an abstract class (Strategy) that has an abstract method specifying how to call the algorithm. Each derived class implements the algorithm as needed [2].

It is required by Strategy pattern that the algorithms (business rules) being encapsulated now lie outside of the class that is using them (the Context) It is assume to be a good design practice to separate behaviors that occur in the problem domain from each other that is, to decouple them [5].

J. Impact on Substitute Algorithm Refactoring Method

Substitute algorithm method is used to replace an algorithm with one that is clearer. This can be done by replacing the body of the method with the new algorithm.

Strategy pattern can be implemented to reduce the efforts required in Substitute Algorithm refactoring method.

K. Implementation

Prepare alternative algorithm and then run the new algorithm against tests, if the results are the same, then finished else if the results aren't the same, uses the old algorithm for comparison in testing and debugging [1].

L. Example

Following Shape class example will justify this concept:

[a] Problem: Consider different shape classes redundantly declaring similar methods i.e. area (), display () and variables length, width etc. in their bodies.

[b] Solution provided by Strategy pattern: This condition can be removed by creating a abstract class Shape and placing common methods and variables into its body as follows:

```
abstract class Shape
{       float length, width;
        Shape(float l, float w )
{
        length=l;
        width=w;
}
        void display();
        void area();
        void volume();
        //remaining code
}
        class Rectangle extends Shape
{       Rectangle(float l, float w)
{
super(l,w);
}
void area()
{       //method body }
}
```

Similarly class Square extends Shape etc.

[c] Solution provided by refactoring method: Similar solution is provided by Substitute algorithm method by declaring single abstract class Shape and extending its methods area(), display() by creating Rectangle, Square etc. child classes objects as follows:

```
        Rectangle r = new Ractangle (4.9f, 3.2f);
        r.area();
        r.display();
```

Thus, it shows that Strategy pattern can be used to eliminate
Substitute algorithm method.

M. Bridge Pattern

Bridge pattern is basically used to decouple an abstraction from its implementation because of this two can vary independently [5].

Bridge is most useful to decouple abstraction from its implementation unless the consideration of whether the Bridge pattern applied. This can be used to abstract out the implementations that are present in problem domain [3].

Bridge pattern can be characterized as follows:

[a] Intent: It decouples a set of implementations from the set of objects using them.

[b] Problem: The derivations of an abstract class must use multiple implementations without causing an explosion in the number of classes.

[c] Solution: Define an interface for all implementations to use and have the derivations of the abstract class to use that interface.

[d] Participants and Collaborators: Classes derived from abstraction use classes derived from implementer without knowing which particular Concrete Implementer is in use.
[e] Consequences: The decoupling of the implementations from the objects that use them increases extensibility.
[f] Implementation: It encapsulates the implementations in an abstract class and contains a handle to it in the base class of the abstraction being implemented [2].

N. Impact on Replace Method with Method Object

Replace method with method object is used when we have a long method that uses local variables in such a way that we cannot apply Extract method.

This can be done by turning the method into its own object due to this all the local variables become fields on that object. We can then decompose the method into other methods on the same object.

O. Bridge Pattern can be used to reduce the efforts required in replace method with method object. Implementation

P. Implementation

Create a new class; give the new class a constructor that takes the source object and each parameter and give the new class a method and copy the body of the original method into compute then, replace the old method with one that creates the new object and calls that method [1].

Q. Example

Consider the following Encryption example that helps to consolidate the two concepts:

[a] Problem: Assume that the class Encryption has coupled algoImplement() method abstraction to its implementation by declaring and describing its body at one place, which causes other classes to declare their own encryption method separately.
[b] Solution provided by Bridge pattern: This condition can be removed by decoupling abstraction with its implementation. This can be done by creating a abstract class algorithm and place algoImplement() method there. Now every other class can override this method accordingly as follows:

```
abstract class Algorithm
{       void algoImplement();
        //remaining code
}
        class Encryption extends Algorithm
{       void algoImplement()
{       //method body
}
}
```

[c] Solution provided by refactoring method: Similar solution can be provided by Replace method with method object. In this approach all the implemented algorithms will be divided into smaller methods and placed into a new class and call it accordingly as needed as follows:

```
class Algorithm
{       void algoImplement()
{
        //method body
}
}
```

```
class Encryption
        {void takeAlgo()
{       Algorithm a = new Algorithm();
        a.algoImplement();
        // remaining code
}
}
```

Hence, Replace method can be eliminated by using Bridge design pattern.

R. Abstract Factory Pattern

Abstract Factory pattern can be used to provide an interface for creating families of related or dependent objects without specifying their concrete classes.

[a] It is known to client object that who to ask for the objects it required and how to use them.
[b] The Abstract Factory class can be used to specify which objects can be instantiated by defining a method for each of these different types of objects.
[c] The concrete factories are used to specify which objects are to be instantiated [3].

The Abstract factory pattern can be characterized as follows:

[d] Intent: It is used to have families or sets of objects for particular clients (or cases).
[e] Problem: Families of related objects required to be instantiated.
[f] Solution: It coordinates the creation of families of objects and gives a way to take the rules of how to perform the instantiation out of the client object that is using these created objects.
[g] Participants and Collaborators: The Abstract Factory defines the interface for how to create each member of the family of objects required.
[h] Consequences: The pattern isolates the rules of which objects to use from the logic of how to use these objects.
[i] Implementation: It defines an abstract class that specifies which objects are to be made, then implement one concrete class for each family [2].

Using the Abstract Factory is indicated when the problem domain has different families of objects present and each family is used under different circumstances [5].

S. Impact on Introduce Local Extension Method

Local extension method is used when a class needs several additional methods, but modification in the class is not possible.

To implement this concept create a new class that contains these extra methods and make this extension class a subclass or a wrapper of the original.

T. Abstract Factory pattern can be used to reduce the efforts required in Introduce local extension method

U. Implementation

Create an extension class either as a subclass or a wrapper of the original and then add converting constructors to the extension and new features to the extension then replace the original with the extension where required and move any foreign methods defined for this class onto the extension [1].

V. Example

The following TimeTable class example will justify this concept:

[a] Problem: Assume that the class TimeTable is accessing two different methods upload() and download() by creating two different class objects of class UploadFile and class DownloadFile separately in its body.

[b] Solution provided by Abstract Factory pattern: This condition can be removed by creating both classes objects into single AbstractFactory Class and extending this class into TimeTable class as follows:

```
class AbstractFactory
{       UploadFile uf = new UploadFile();
        DownloadFile df = new DownloadFile();
}
        class TimeTable extends AbstractFactory
{       uf.upload();
        df.download();
}
```

[c] Solution provided by refactoring method: Similar solution can be provided by Introduce local extension method by placing both the objects into single class and accessing them using extending that class as follows:

```
class EnhanceAbility
{       UploadFile uf = new UploadFile();
        DownloadFile df = new DownloadFile();
}
```

Similarly this class will be extended by TimeTable class.

Hence, this example clears that Abstract factory pattern can be used to eliminate Local extension method.

W. Singleton Pattern

It ensures that a class only has one instance, and provides a global point of access to it. The Singleton pattern works by having a special method that is used to instantiate the desired object. Some interesting concepts about this special method:

[a] When this method is called, it checks to observe whether the object has already been instantiated. If it has, the method just returns a reference to the object. If not, the method instantiates it and returns a reference to the new instance.

[b] To ensure that this is the only way to instantiate an object of this type, we have to define the constructor of this class to be protected/private [5].

The Singleton pattern can be characterized as follows:

[c] Intent: Intent is to have only one of an object, but there is no global object that controls the instantiation of this object and to ensure that all entities are using the same instance of this object, without passing a reference to all of them.

[d] Problem: Several different client objects require referring to the same thing, and we have to ensure that we do not have more than one of them.

[e] Solution: It guarantees one instance.

[f] Participants and Collaborators: Clients create an instance of the Singleton solely through the getInstance method.

[g] Consequences: Clients doesn't require concerning themselves whether an instance of the Singleton exists. This can be controlled from within the Singleton.

[h] Implementation: Add a private static member of the class that refers to the desired object. (Initially, it is null.), then add a public static method that instantiates this class if this member is null (and sets this member's value) and then returns the value of this member and last set the constructor's status to protected or private due to this no one can directly instantiate this class and bypass the static constructor mechanism [2].

The essence of the Singleton pattern is that every object in the application uses the same instance of the Singleton [4].

X. Impact on Replace Constructor with Factory Method

Replace constructor with factory method is used to provide better features using simple construction when creating an object.

Singleton pattern can be used to reduce the efforts required in Replace constructor with Factory refactoring method can be reduced at very high extent.

Y. Implementation

Create a factory method and make its body a call to the current constructor then replace all calls to the constructor with calls to the factory method and declare the constructor private/protected [1].

Z. Example

The following FileInstance class example will illustrate this concept:

[a] Problem: Consider FileInstance class has multiple accesses point to instantiated its object.

[b] Solution provided by Singleton pattern: This can be removed by providing single global access point to FileInstance class object.class OpenFile provides the single global point to access and instantiate FileInstance class object as follows:

```
class FileInstance
{       FileInstance f=new File ("Test.txt");
        protected  FileInstance (File fl)
{       f=fl;
}
}
class OpenFile extends File
{       File newFile;
        OpenFile(newFile)
{       super(newFile);
}
}
```

Now, FileInstance class object can only be instantiated using single global point provided by OpenFile class.

[c] Solution provided by refactoring method: Similar solution can be provided by replacing constructor of FileInstance class using Factory method as follows:

```
protected static FileInstance instant (File fl)
{       f=fl;
}
```

This factory constructor can only be called and instantiated using its child class OpenFile as follows:

```
FileInstance fit = FileInstance.instant(newFile);
```

Thus, this example shows that Singleton pattern can be used to eliminate Replace constructor with factory method.

III. CONCLUSION AND FUTURE RESEARCH WORK

In this paper, we have discussed various design patterns as well as various refactoring techniques. We have also discussed the importance of various design patterns solving

the common software development problems. It can reduce the efforts required in most common refactoring methods to a large extent with different examples. It will help to improve overall software design.

Our future research work will focus on the case study of large projects to show the impact of design patterns over refactoring process.

IV. REFERENCES

[1] M.Fowler, K.Beck, J.Brant, W. Opdyke, D.Roberts: Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[2] Alan Shalloway, James R. Trott: Design Patterns Explained, A New Perspective on Object Oriented Design, Second Edition, Addison Wesley Professional 2004.

[3] S. Demeyer, S. Ducasse, O. Nierstrasz : Object Oriented Reengineering Patterns, Morgan Kaufmann, 2002.

[4] Steven John Metsker: Design Patterns Java Workbook, Addison Wesley, 2002.

[5] Douglas C. Schmidt: Design Patterns and Frameworks for Object Oriented Communication Systems, Washington University, 1997.

[6] Gamma et al.: Design Patterns: Elements of Reusable Object – Oriented Software, Addison Wesley, Reading, MA, 1994.

[7] James O. Coplien and Douglas C. Schmidt: Pattern Languages of Program Design, Addison Wesley, Reading, MA, 1995.