ISSN No. 0976-5697

Volume 5, No. 6, July-August 2014

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

Software Quality Improvement using Design Patterns

Ms. Renu Bala
Department of Computer Science and Application
Chaudhary Devi Lal University
Sirsa, Haryana, India

Mr. Kapil Kumar Kaswan
Department of Computer Science and Application
Chaudhary Devi Lal University
Sirsa, Haryana, India

Abstract— Design patterns usually describe abstract systems of interaction between classes, objects, and communication flows. So, a description of a set of interacting classes that provide a generalized solution framework to a generalized/specific design problem in a specific context can be said as a design pattern. There are many design patterns that can be used to solve real-life problems, but it remains very difficult to design, implement and reuse software for complex applications. Examples of these include enterprise system, real-time market data monitoring and analysis system. Design patterns provide an efficient way to create more flexible, elegant and ultimately reusable object-oriented software. 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". The solutions of the given problems are expressed in terms of objects and interfaces. Among 23 design patterns, Strategy pattern defines an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a Concrete Strategy. In accounting framework one thing is mostly needed that is tax calculation. To solve this problem author in the current study has chosen the strategy pattern.

Keywords: Design Pattern, Context, Abstract Factory,Object.

I. INTRODUCTION

A design pattern is a generic solution that has been observed in multiple instances to help resolve a particular problem within a known context. Design patterns provide an efficient way to create more flexible, elegant and ultimately reusable object-oriented software. Design methods are supposed to promote good design, to teach new designers how to design well and to standardize the way designs are developed. Typically, a design method comprises a set of syntactic notations usually graphical and a set of rules that govern how and when to use each notation. It will also describe problems that occur in a design, how to fix them, and how to evaluate a design**.** 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 [1]. The solutions of the given problems are expressed in terms of objects and interfaces. Design patterns are being increasingly used in software design. Design patterns are a good means for recording design experience as they systematically name, explain and evaluate important and recurrent designs in software systems. They describe problems that occur repeatedly, and describe the core of the solution to that problem, in such a way that this solution can be used many times in different contexts and applications. A good design is a good solution regardless of the technology. And no matter how good the technology may be, it is only as good as its design, and specifically the implementation of that design. In fact, a great design with older technology may still be good, but a bad design with new technology is usually just bad. A design pattern is a form of design information and the design that worked well in past will be used in future in any application similar to existing application which uses these designs. These design information can help both the experienced and the novice designer to recognize situations in which these designs can be reused. There are three categories of design patterns: Creational, structural and Behavioral.

II. NET FRAMEWORK

A .net is a new software platform for the desktop and the Web. The .NET Framework is an integral Windows component that supports building and running the next generation of applications. The .NET Framework has two main components: the common language runtime and the .NET Framework class library. The common language runtime is the foundation of the .NET Framework [2]. The .NET Framework is designed to fulfill the following objectives:

a. To provide a consistent object-oriented programming environment whether object code is stored and executed locally, executed locally but Internet-distributed, or executed remotely.
b. To provide a code-execution environment that minimizes software deployment and versioning conflicts.
c. To provide a code-execution environment that promotes safe execution of code, including code created by an unknown or semi-trusted third party.
d. To provide a code-execution environment that eliminates the performance problems of scripted or interpreted environments.
e. To make the developer experience consistent across widely varying types of applications, such as Windows-based applications and Web-based applications.
f. To build all communication on industry standards to ensure that code based on the .NET Framework can integrate with any other code [2].

III. ABSTRACT FACTORY PATTERN

The abstract factory pattern is a design pattern that provides a way to encapsulate a group of individual factories that have a common theme. This pattern provides interfaces for creating families of related or dependent objects without specifying their concrete classes. Modularization is a big issue in today's programming. Programmers all over the world are trying to avoid the idea of adding code to existing classes in order to make them support encapsulating more general information. Take the case of an information manager which manages phone number. Phone numbers have a particular rule on which they get generated depending on areas and countries. If at some point the application should be changed in order to support adding numbers form a new country, the code of the application would have to be changed and it would become more and more complicated. In order to prevent it, the Abstract Factory design pattern is used. Using this pattern a framework is defined, which produces objects that follow a general pattern and at runtime this factory is paired with any concrete factory to produce objects that follow the pattern of a certain country. In other words, the Abstract Factory is a super-factory which creates other factories (Factory of factories) the classes that participate to the Abstract Factory pattern are:

a. AbstractFactory - declares an interface for operations that create abstract products.
b. ConcreteFactory - implements operations to create concrete products.
c. AbstractProduct - declares an interface for a type of product objects.
d. Product - defines a product to be created by the corresponding ConcreteFactory; it implements the AbstractProduct interface.
e. Client - uses the interfaces declared by the AbstractFactory and AbstractProduct classes.

The Abstract Factory pattern has both benefits and flaws. On one hand it isolates the creation of objects from the client that needs them, giving the client only the possibility of accessing them through an interface, which makes the manipulation easier. The exchanging of product families is easier, as the class of a concrete factory appears in the code only where it is instantiated. Also if the products of a family are meant to work together, the Abstract Factory makes it easy to use the objects from only one family at a time. On the other hand, adding new products to the existing factories is difficult, because the Abstract Factory interface uses a fixed set of products that can be created. That is why adding a new product would mean extending the factory interface, which involves changes in the AbstractFactory class and all its subclasses. This section will discuss ways of implementing the pattern in order to avoid the problems that may appear.

A. Factories as singletons:

An application usually needs only one instance of the ConcreteFactory class per family product. This means that it is best to implement it as a Singleton.

B. Creating the products:

The AbstractFactory class only declares the interface for creating the products. It is the task of the ConcreteProduct class to actually create the products. For each family the best idea is applying the Factory Method design pattern. A concrete factory will specify its products by overriding the factory method for each of them. Even if the implementation might seem simple, using this idea will mean defining a new concrete factory subclass for each product family, even if the classes are similar in most aspects.

For simplifying the code and increase the performance the Prototype design pattern can be used instead of Factory Method, especially when there are many product families. In this case the concrete factory is initiated with a prototypical instance of each product in the family and when a new one is needed instead of creating it, the existing prototype is cloned. This approach eliminates the need for a new concrete factory for each new family of products.

C. Extending the factories:

The operation of changing a factory in order for it to support the creation of new products is not easy. What can be done to solve this problem is, instead of a CreateProduct method for each product, to use a single Create method that takes a parameter that identifies the type of product needed. This approach is more flexible, but less secure. The problem is that all the objects returned by the Create method will have the same interface, that is the one corresponding to the type returned by the Create method and the client will not always be able to correctly detect to which class the instance actually belongs[3].

IV. RELATED WORK

There are various design patterns that can be used to solve any of the industrial application. Here in this paper work, Singleton pattern, Abstract factory pattern is used to build a framework. Using these patterns, design solution of the industrial problem will be described. The father of the pattern concept, proposed a description template stating nine essential pattern elements. These patterns element describes the design patterns effectively; also describe how these patterns are useful to solve the problem. Industrial applications typically require different kinds of interfaces to the data they store and the logic they implement are data loaders, user interface and integration gateways and others. Instead of using for different purpose, these interfaces often need common interactions with the application to access and manipulate its data and invoke its business logic. These interactions may be complex, involving transaction across multiple resources and the coordination of several responses to an action. These interfaces decide the interaction between different layers of the application; how user interacts with middleware layer and the database layer. The framework is implemented in .Net. As we are using the design patterns to build this framework hence the developer can use this framework to build any kind of industrial application and can implement it in any other programming language using object-oriented concepts. Using the concept of design patterns, now we are proposing a class diagram of an industrial application. There are various classes with their methods and properties [5].

V. ANALYZE ABSTRACT FACTORY PATTERN BY REFLECTION

Reflection is simply a mechanism that allows components or classes to interrogate each other at run time and discover information that goes beyond the information gleaned from the publicly available interfaces the objects expose. In essence, reflection enables objects to provide information about themselves (metadata). The common language runtime loader manages application domains. This management includes loading each assembly into the appropriate application domain and controlling the memory layout of the type hierarchy within each assembly. The use of reflection in current study is to create an instance of a class once (Singleton pattern) and used that instance forever in that application. The reflection methods works as a singleton pattern in this framework.

System.Reflection Namespace hierarchy

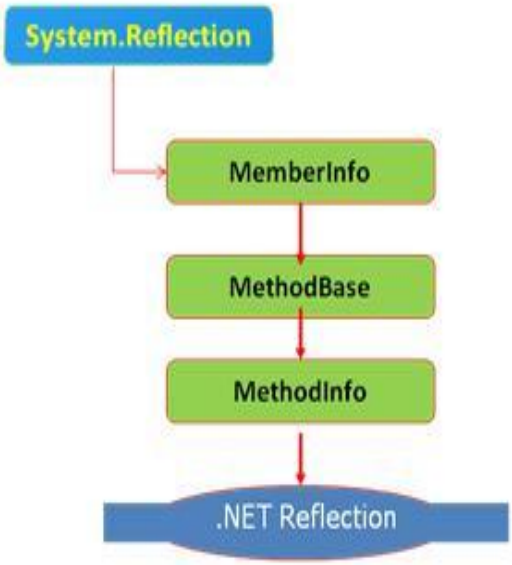

Figure: 1

VI. SIMULATION OF ABSTRACT FACTORY DESIGN PATTERN

The abstract factory pattern is a design pattern that provides a way to encapsulate a group of individual factories that have a common theme. This pattern provides interfaces for creating families of related or dependent objects without specifying their concrete classes.

In this research paper, we are building a framework using patterns that can be used to solve various industrial designs problem. This framework can be used by various programmers but its implementation may vary. There are various form created under this framework.

Industrial Application Model describes main classes and relationships which could be used during analysis phase to better understand domain area for any kind of Industrial Application.

ViewIfc interface is a user interface through which users can interact to the application model. There is a state class through which a user can decide in which state the ViewIfc will open the application i.e. to edit or to delete the entry in the application.

RequestHandler class is an intermediate class between the ViewIfc and WorkerIfc that how a user can add a new worker or can use existing workers. There are two interfaces which implements the WorkerIfc that are MasterWorkerIfc and TransactionWorkerIfc. Any action performed on these two interfaces is stored in Action enumeration.

All the workers stored in the DBInteraction enumeration. All the actions that are performed on DBInteraction are stored in DbActions enumeration.

The form ViewIfc interface form. Item form is Graphic user Interface (GUI) for interacting with user. Every GUI implements viewIfc Interface. Method GetWorker provides the workers by using singleton class object RequestHandler. The state enumeration describes the state in which it is open. There are various methods available in the state enumerations that are Add, Edit, Delete, and Query. That defines whether to add new records or edit or delete existing records.

The ViewIfc interface is a user interface form from which user can interact with application. There are three methods in the ViewIfc interface that are FillBag, GetWorker, and SetFilter. The FillBag method is used to fill the entries in the database. We can use the GetWorker method of ViewIfc interface to call the RequestHandler class. RequestHandler is a singleton class. It contains static reference variable minstance of type RequestHandler.

AddEventHandler method uses reflection to make object of worker classes and add them into _eventhandler collection using Actions enum as parameter and return the worker.

The framework is inspired from MVC model (Model, View, and Controller). The model consists of application data and business rules, and the controller mediates input, converting it to commands for the model or view. A view can be any output representation of data, such as a chart or a diagram. With the responsibilities of each component thus defined, MVC allows different views and controllers to be developed for the same model. It also allows the creation of general-purpose software frameworks to manage the interactions.

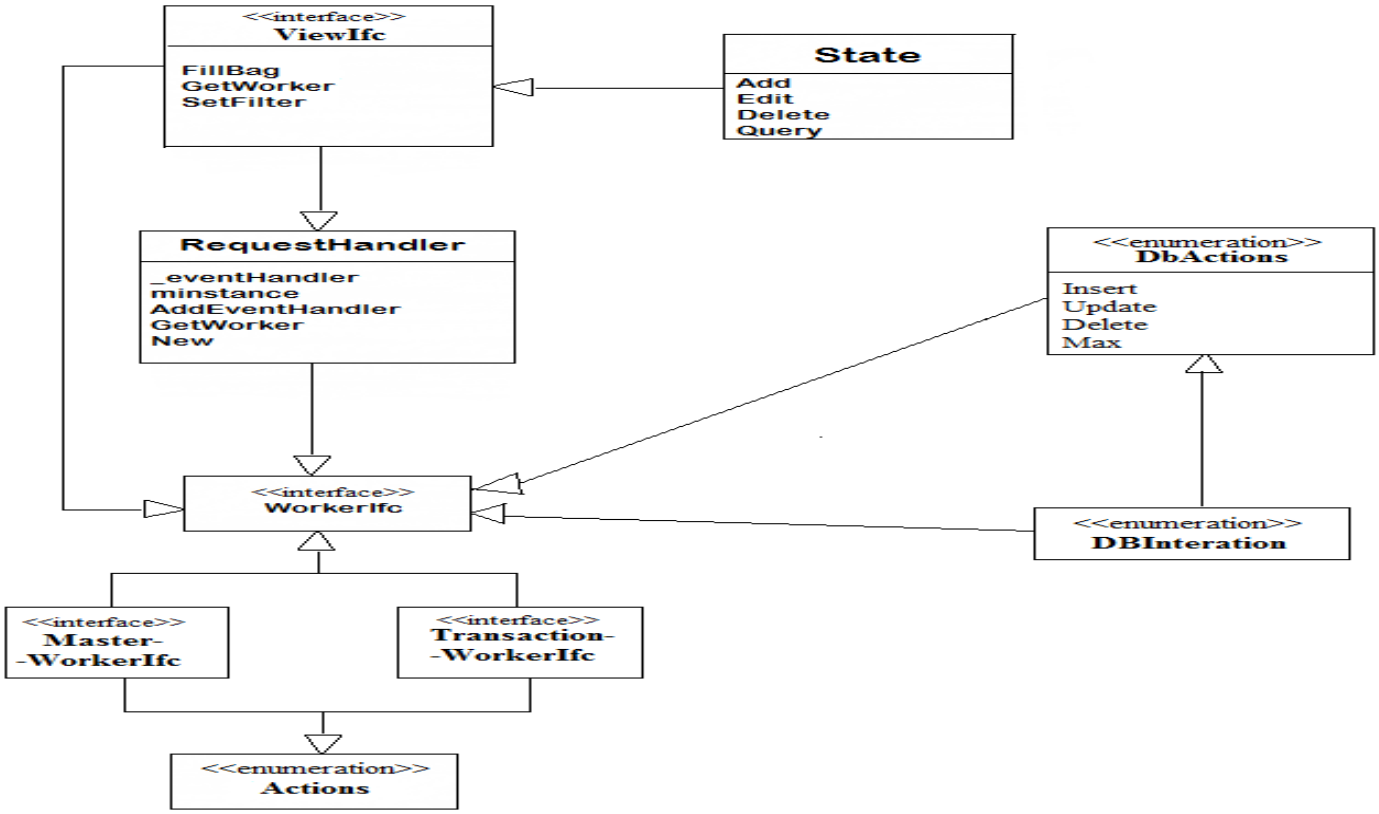

Figure:2 Framework of industrial application

The ViewIfc interface is a user interface form from which user can interact with application. There are three methods in the ViewIfc interface that are FillBag, GetWorker, and SetFilter. The FillBag method is used to fill the entries in the database. We can use the GetWorker method of ViewIfc interface to call the RequestHandler class. RequestHandler is a singleton class. It contains static reference variable minstance of type RequestHandler.

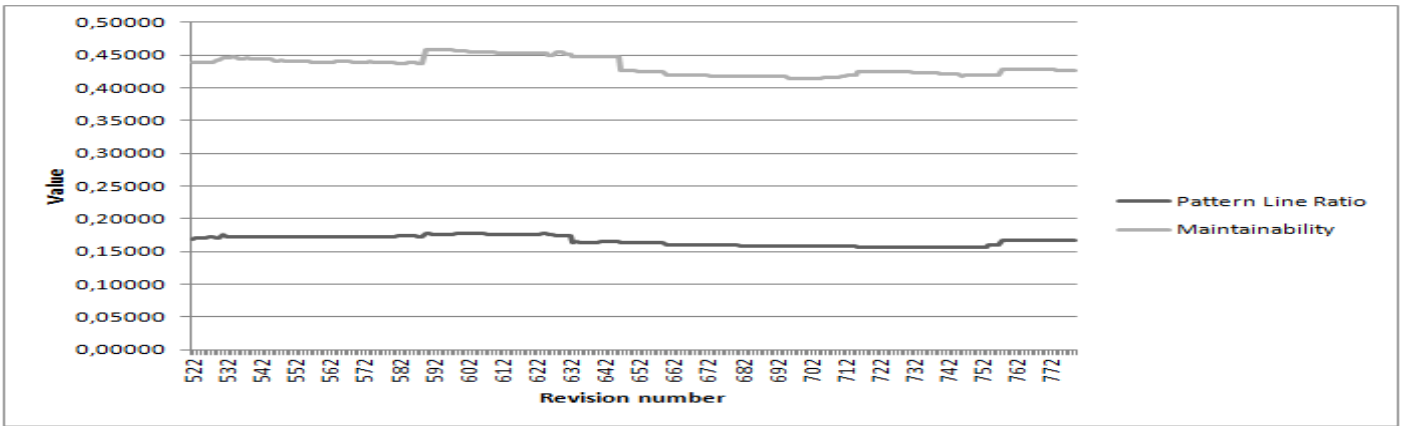

Figure 3 The tendencies of pattern line density and maintainability.

Table 1. The tendency of the quality attributes in case of design pattern changes

Revision (r)	pattern	Pattern line density(*PDensr*)	maintainablity(*Mr*)	testability	analyzability	stability	Change ability
531	+3	↑	↑	↑	↑	↑	↑
574	+1	↑	↑	↑	↑	↑	↑
609	-1	↓	-	-	-	-	-
716	+1	↓	↑	↑	↑	↑	↑
758	+1	↑	↑	↑	↑	↑	↑

VII. CONCLUSION

Although the belief of utilizing design patterns to create better quality software is fairly widespread, there is relatively little research objectively indicating that their usage is indeed beneficial. In this paper we try to reveal the connection between design patterns and software maintainability. It is very hard to understand better what the patterns are and how they relate to each other. At this point there is a fundamental picture as reacting to an event to produce accounting entries. We used our probabilistic quality model for estimating the maintainability. We found that every introduced pattern instance caused an improvement in the different quality attributes. Moreover, the average design pattern line density showed a very high, 0.89 Pearson correlations with the estimated maintainability values. Design patterns are outstanding communication tool and help to make the design process faster. This allows solution providers to take the time to concentrate on the business implementation. Patterns help the design to make it reusable. Reusability not just applies to the component, but also the stages of the design that must go from a problem to final solution. The ability to apply a pattern that provides a repeatable solution is worth the little time spent learning formal patterns. There are some promising results showing that applying design patterns improve the different quality attributes according to our maintainability model. In addition, the ratio of the source code lines taking part in some design patterns in the system has a very high correlation with the maintainability. However, these results are only a small step towards the empirical analysis of design patterns and software quality [4].Design patterns shall support reuse of a software architecture in different application domains as well as reuse of flexible components[6].

VIII.REFERENCES

[1] http://blogs.infragistics.com/blogs/ux/archive/2009/02/03/what-is-a-design-pattern-and-why-use-them-for-quince.aspx

[2] Bertrand Meyer, Karine Arnout, Componentization: The Visitor Example, to appear in Computer (IEEE), 2006.

[3] http://www.oodesign.com/abstract-factory-pattern.html

[4] P´eter Heged˝us, D´enes B´an, Rudolf Ferenc, and Tibor Gyim´othy University of Szeged, Department of Software Engineering ´Arp´ad t´er 2. H-6720 Szeged, Hungary {hpeter,zealot,ferenc,gyimothy}@inf.u-szeged.hu

[5] Meyer, Bertrand "Componentization: The Visitor Example". *IEEE computer* (IEEE) **39** (7): 23–30.

[6] Jurgen Dorn and Tabbasum Naz,Institute of Information Sysytems 184/2 Technicla University Vienna ,Favoritenstrabe 9-11, Vienna A-1040, Austria {dorn/naz}@dbai.tuwien.ac.at