Evaluation Amid different Software Design Patterns

Naseer Ahmad
Virtual University of Pakistan
M.A. Jinnah Campus, Defence
Road, Lahore, Pakistan

Muhammad Waqas Boota
Virtual University of Pakistan
M.A. Jinnah Campus, Defence
Road, Lahore, Pakistan

ABSTRACT
The software developers have been wholeheartedly implemented the various design patterns in current years. There is sufficient confirmation that patterns can have a valuable blow on software quality, on the other hand in some situations patterns have been improperly implemented because of having not too much experience. A software developer can make the design process simpler, well design to be reuse and most proficient with the help of various design patterns. But, a software developer should have a lot of experience and knowledge of using the design patterns. The design quality is a significant tool that is considered to find out the quality of a software product. In order to find out the quality of design in the premature stages of software process is an important task to extend and improve the quality of software. In order to make the software professional, the design patterns are useful tools for this purpose. This paper represents that how evaluation among different software design patterns with Chidamber-Kemerer (C&K) metrics is carried out.

Keywords
Design-pattern, C&K, metrics, evaluation

1. INTRODUCTION
Every pattern illustrates a problem that arises again and again in our surroundings, and then illustrates the main solution to that problem, in a particular way which we can make use of this solution number of times, lacking always doing the same thing two times. The fundamental elements of a pattern in which the first one is the pattern name in which we use to illustrate a particular problem related to the design. The second element is that the problem that illustrates when to implement the pattern. The third one is that the solution that illustrates the components that structure the design. The last one is the outcomes that are the final results and transactions of applying the pattern. We can say that the design patterns are not related to the designs such as hash tables that may be programmed in classes. These are not also like as the linked lists too. These are not too much complicated, domain-specific designs for whole application. These are explanations of communicating the classes as well as objects that are adapted to find out a solution of a specific design in a specific context.

In this paper it will be explained that how the evaluation on these patterns is carried out with C&K metrics and description of one design pattern type of each of these three design patterns is also explained. An evaluation of these patterns in which a description of one design pattern type of each of these three design patterns will be carried out [1]. Creational patterns offer a way to make objects as hiding the logic of creation of objects, instead of instantiating objects openly using fresh operator. It offers the program additional flexibility to decide that which particular object requires to be creating for the given use case [2]. These patterns provide the summary of instantiation process. These patterns are helpful to create an autonomous system that how can we create its objects, how its objects are composed as well as represented. These patterns become imperative as the development of system depend additional on the composition of the object as compared to the class inheritance. For a moment these patterns become opponents. For example, in some situations when also Prototype or Abstract Factory can be used beneficially. And in some situations these patterns become complementary: developer may use one of the other design patterns in order to implement which element gets created [1].

The composition of class and object is concerned in the structural design patterns. In order to get new features ways are defined for compositions of objects and also perception of inheritance is applied for the composition of interfaces [2]. In order to make large structures these patterns give perception that how the classes and objects are composed with these patterns. For example, number of inheritance merges the more than two classes into single class. The resultant class merges the properties of the parent class. Structural patterns are significantly helpful for constructing developed libraries work collectively but in parallel. Class structure of adapter pattern is another example of structural pattern. Usually, an adapter creates another interface from parent or original interface; in this manner it offers a smooth abstraction of different interfaces. A class adapter achieves this objective with the help of inheritance in confidential from an adaptee class. The adapter then articulates its own interface in terms of adaptee's class.

Communication among objects is related with behavioral patterns. These patterns are also concerned with task of jobs among objects and algorithms. These patterns express not only the classes and objects pattern but it also express communication among objects as describe earlier. Behavioral patterns also describe the complex flow of control which is not easy to trail at run time. They change our focus from control flow to let us focus only on the way objects are interrelated. Inheritance is used in these patterns to deal out behavior among classes. Composition of objects is used by behavioral objects patterns instead of inheritance [1, 2]. In ordered to provide evaluation among three main types of design patterns, we will describe one type of each design pattern and describe their implementation as well as their class diagrams. In ordered to provide evaluation that how through C&K metrics are implemented for evaluation purpose of various design patterns, are explained. The part 2 carries the related work, in the part 3 creational design patterns with its one type factory pattern is explained, in the part 4 structural design patterns with its one type adapter, in the part 5 behavioral design pattern with its type mediator pattern, part 6 explain how Chidamber-Kemerer (C&K) metrics are used to evaluation and analysis of different design patterns. Part 7 carries the concluded work of our research work.

2. RELATED WORK

Since 1994 various software design patterns detonated and at this current modern period hundred of patterns came into existence. There are number of warehouses that preserve and document the patterns in which the first one was created in 1995 by Ward Cunningham and it is called "Portland Pattern Repository". The major objective of this paper is to provide the fundamental knowledge about the evaluation of different design patterns belongs to creational, structural and behavioural patterns. In this paper the knowledge provided about design patterns is that how someone can evaluate these patterns. The main aim of design patterns is that how we can encourage the established design and frequently apply the successful solution pattern in various contexts. An important point is that we should keep in mind that we can not apply the design patterns blindly. We should keep in mind that the patterns are templates and these must be acclimatized to the specific solution in which these patterns belong. The matter of continues modification and designated fulfilled needs application modification in the code. Every implementation modification might need modification in the implementation of the existing design pattern. If complication in modification is much high then it should require another design pattern or developing of current design pattern to extra complicated implementation.

It can be much motivating to perform standard test on the code which is employed by make use of design patterns and in order to perform rapid optimized code is executed and this is executed without design patterns. These executions should implement the equivalent clarification; therefore it is easy to evaluate what are distinction in the execution time, and utilization of additional resources. We can expand this test with the utilization of various patterns and modifying the execution complexity. Composed results can be used to evaluate the cost which is associated to the design patterns. On the other hand, this type of evaluation cannot be consistent if there is no involvement of cost evaluation or analysis relative to the maintenance of code, correction of errors, and performance of modification arrangement and refactoring, enhanced design of current code [3]. The sketch of every design pattern has a segment where these are interconnected to other design patterns, of equivalent, of a privileged or worse granularity level are offered. These connections affect the construction process, since we should always see at relative patterns when we develop something; and we should always implement the higher level design patterns at initial stages. Taxonomy for design patterns, but this is not for their shared interconnection is given. Developers can use design patterns at various levels, and what is obtained at initial level can be assumed a fundamental pattern at secondary level. It is possibly representative of most architecture, several patterns will common and various will be precise to the area of application [4].

3. FACTORY METHOD DESIGN PATTERN

Factory pattern falls in the class of creational design patterns and this pattern is most popular and widely used in the design patterns in java. This is most useful pattern and has best methods for the creation of an object. In this pattern without revealing the logic of creating object to the user we can create an object and pass on to recently created object by using a general interface [2]. In other words we can say that factory itself is an object and it creates another object. We can also say that object factory is parent object and the objects that are created from it are child objects. This is mainly used in the frameworks and also in toolkits. These are used in the development based on test-driven that permits the classes to be tested before implementation. Factories also have benefits such as developers can reuse the objects with least duplication of code. While testing the objects, management of critical information can be passed prior to the information is specified. The duration of an object can central to make sure constant behaviour. These also have the capability of encapsulation because factories encapsulate the objects creation. Factories have limitations too because refactoring the current class to make use of other class causes a break in existing user. It cannot be expanded because it depends on using the private constructor in order to encapsulate the information. On the other hand it is expanded; the subclass should give its individual re-implementation [7].

3.1 Implementation

Now a *Barbeque* interface and the implementation of Barbeque interface with concrete classes will carried out. A factory class *BarbequeFactory* is defined next. *FactoryPatternDemo*, this demo will use *BarbequeFactory* in order to get a Barbeque object. This demo passes an information (ChickenTikka / MuttenTikka / BeefTikka) to *BarbequeFactory* to obtain the type of object that it requires. The design principle with the help of sample code and class diagram of the class Barbeque is given below:

```
Barbeque orderBarbeque(String type) {
Barbeque barbeque;
if (type.equals("chickentikka")) {
Barbeque = new ChickenTikkaBarbeque ();
 } else if (type.equals("muttentikka")) {
Barbeque = new MuttenTikkaBarbeque ();
} else if (type.equals("beeftikka")) {
Barbeque = new BeefTikkaBarbeque ();
}
barbeque.prepare();
barbeque.grill();
barbeque.box();
return barbeque;
}
```

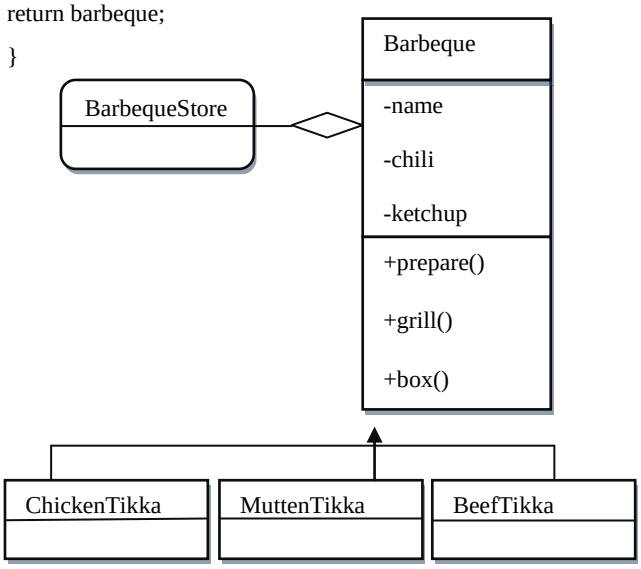

Fig 1: Class diagram of a class Barbeque [6]

Here is a problem in the design given above, now by performing some modifications in the code and the class diagram to get an accurate design:

Barbeque order*Barbeque* (String type) {

Barbeque barbeque;

if (type.equals("*chickentikka*")) {

barbeque = new *ChickenTikkaBarbeque* ();

} else if (type.equals**("***muttentikka***")) {**

barbeque **=** new *MuttenTikkaBarbeque* **();**

} else if (type.equals("*beeftikka*")) {

barbeque = new *BeefTikkaBarbeque*();\\encapsulation

} else if (type.equals("*shahitikka*")) {

barbeque **=** new *ShahiTikkaBarbeque***();**

}

barbeque.prepare();

barbeque.grill();

barbeque.box();

return barbeque;

}

A new object *shahitikkaBarbeque* is created as above in the code. Suppose that some people do not like Beef Tikka therefore now remove the object *beeftikkaBarbeque* and replaces it with the object *shahitikkaBarbeque*. In order to perform such type of modification in programming is difficult task. For encapsulation of the code that modifies will make the design much flexible by creating a SimpleFactory. Nowl remove the code that creates a barbeque and this forms a factory as follows:

public class *SimpleBarbequeFactory* {

public *Barbeque* create*Barbeque*(String type) {

Barbeque barbeque;

Barbeque order*Barbeque* (String type) {

Barbeque barbeque;

if (type.equals("*chickentikka*")) {

barbeque = new *ChickenTikkaBarbeque* ();

} else if (type.equals("*muttentikka*")) {

barbeque = new *MuttenTikkaBarbeque* ();

} else if (type.equals("*shahitikka*")) {

barbeque = new *ShahiTikkaBarbeque*();

}

return barbeque;

}

}

Now it is clear in this piece of code orderBarbeque is in order or in organized form.
public class *BarbequeStore* {

SimpleBarbequeFactory factory;

public BarbequeStore(*SimpleBarbequeFactory* factory) {

this.factory = factory;

}

public *Barbeque orderBarbeque*(String type) {

Barbeque barbeque;

barbeque = factory.create*Barbeque*(type);

barbeque.prepare();

barbeque.grill();

barbeque.box();

return barbeque;

}

}

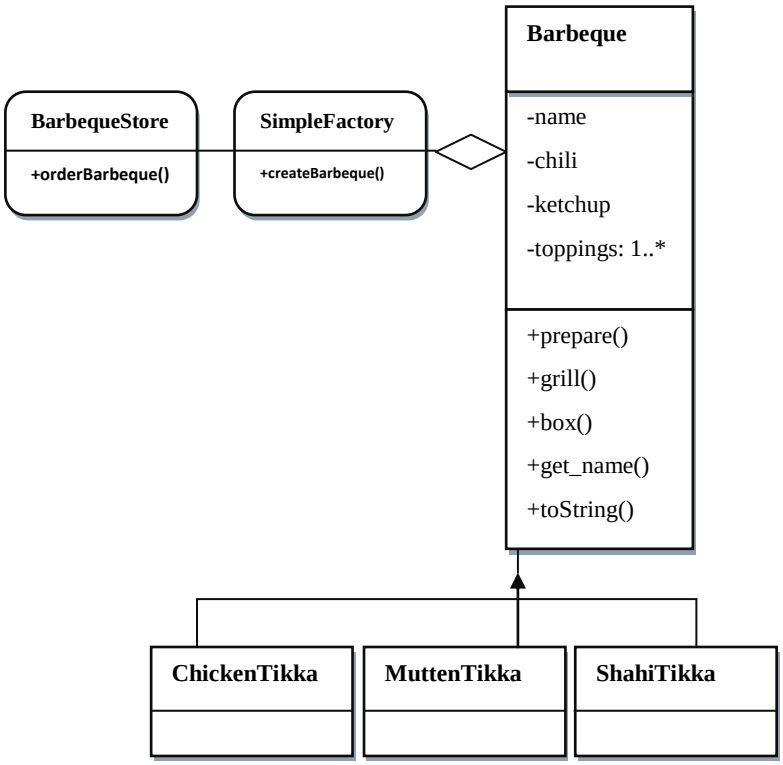

Fig 2: Revised class diagram of class Barbeque [6]

4. ADAPTER METHOD DESIGN PATTERN

The Adapter design pattern exchanges the interface of one class into another class. Programmers use adapter patterns whenever they desire to work unrelated classes jointly in a single program. The idea of this pattern is quite simple: a programmer writes a class that has preferred interface and after writing this class the programmer makes this able to communicate with another class that has different interface [8].

4.1 Intent

It is used to convert the interface of a particular class into another class for the expectation of the client. It makes the classes able to communicate each other that are different from their interfaces.

4.2 Also Known As

Adapter patter also knows as wrapper.

4.3 Elements

4.3.1 Target

It describes or defines the interface used by *Client*, interface is domain specific.

4.3.2 Client

It works together with objects that confirm the target interface.

4.3.3 Adapter

This element adjusts or adapt adaptee's interface to the target interface.

4.3.1 Adaptee

It defines the current interface that wants adapting.

4.4 Adapter Pattern Structure

Adapter pattern has class adapter as well as object adapter class adapter makes use of various inheritance in order to adapt one interface into another interface (Fig 3).

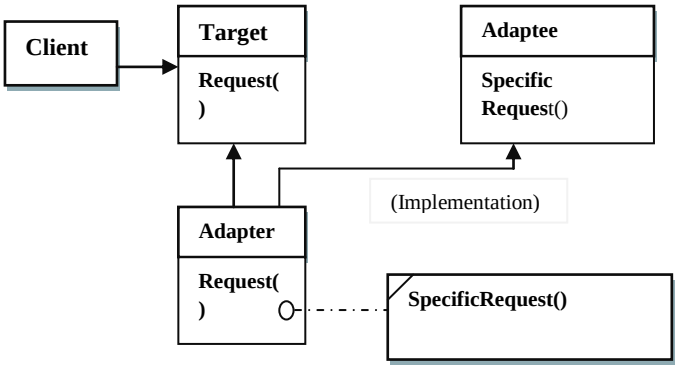

Fig 3: Structure of class Adapter design pattern [1].

On the other hand an object pattern depends on the Composition of different objects (Fig 4).

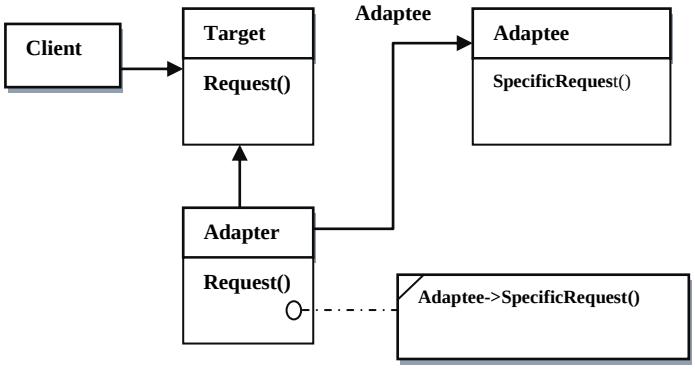

Fig 4: Structure of object adapter design pattern [1].

In ordered to understand Adapter patterns a simple example is illustrated here that is real life example. Assume a mobile phone requires 12 Volts of power supply for its battery charging but the main power supply produces current of 220 Volts. These 220 Volts are too much high for charging of mobile phone and not suitable for requirement because it can damage the battery or mobile phone. Since, to charge battery it needs 12 Volts instead of 220 Volts.

In order to solve this issue mobile phone charger act as an adapter and this adapter make a relationship among the interface of mobile phone and main power supply that are unrelated to each other. According to this mechanism an Adapter works [9]. This Adapter pattern falls in the category of structural design patterns. Another real life example of Adapter is suppose some Pakistani researchers go to Chinese university in order to complete their research work. The official language of China is Mandarin and Pakistani researchers do not understand Mandarin language. In order to solve this problem Chinese university provide a translator for Pakistani researchers. The translator first teaches the researchers Mandarin language to the researchers in order to understand lectures that are delivered by professor.

4.5 Implementation

In this example researchers are Target, the translator is Adapter and the Pakistani researchers are adaptee [11]. The class diagram and code synopsis of above example or scenario is given below:

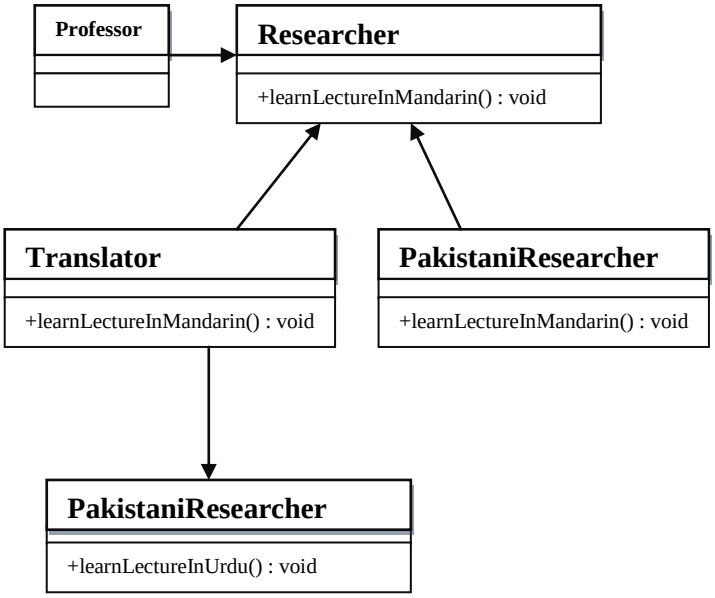

Fig 5: Class diagram of Class Researcher [11].

4.5 The Professor Class

```
public class Professor {
List<Researcher> listResearcher;
public Professor() {
listResearcher = new ArrayList<Researcher>();
}
public void teachResearchers() {
for (Researcher researcher : listResearcher) {
researcher.learnLectureInMandarin ();
}
}
}
```

4.6 The Researcher Class

```
public abstract class Researcher {
public abstract void
learnMandarin ();
}
```

4.7 The Translator Class

```
public class Translator extends Researcher {

PakistaniResearcher pakistaniResearcher;

public Translator(PakistaniResearcher pakistaniResearcher)
{

super();

this.pakistaniResearcher = pakistaniResearcher;

}

@Override
public void learnLectureInMandarin () {
System.out.println("Translate to urdu");
pakistaniResearcher.learnLectureInUrdu();
}

}
```

4.8 The Chinese Researcher Class

```
public class ChineseResearcher extends Researcher {

@Override

public void learnMandarin() {

System.out.println("Learning mandarin");

}

}
```

4.9 The Pakistani Researcher Class

```
public class PakistaniResearcher {

public void learnLectureInUrdu() {

System.out.println("Learn in urdu");

}
```

5. MEDIATOR METHOD DESIGN PATTERN

In order to minimize communication complexity among various objects as well as in classes Mediator design patterns are used. Mediator pattern offers a mediator class that basically manages entire communications among multiple classes and maintains simple maintainability of the code with the help of loose coupling and this pattern falls in the category of behavioural patterns [2]. This pattern indicates that an object that encapsulates and how a group of objects can work together with other objects of the group. As mentioned earlier it also supports loose coupling by remaining objects from submitting to each other unambiguously, and it allows the developers differ their communications in parallel. Following diagram shows the communication among two objects [1].

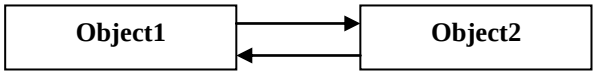

Fig 6: Point-to-Point Communication among Two Objects [10]

Multiple objects can also communicate each other and following figure shows the communication among multiple objects.

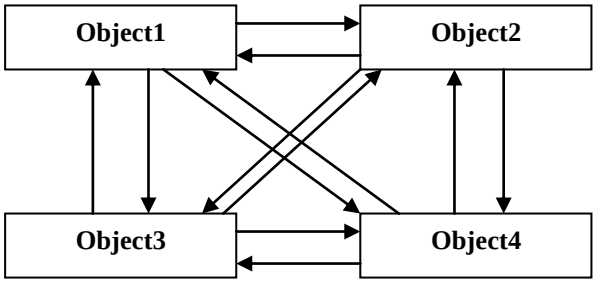

Fig 7: Point-to-Point Communication among multiple Objects [10]

5.1 Implementation

Here is an example of Class *FaceBook* that describes how facebook friend communicate each other. This example is explained with help of code synopsis and class diagram as follow:

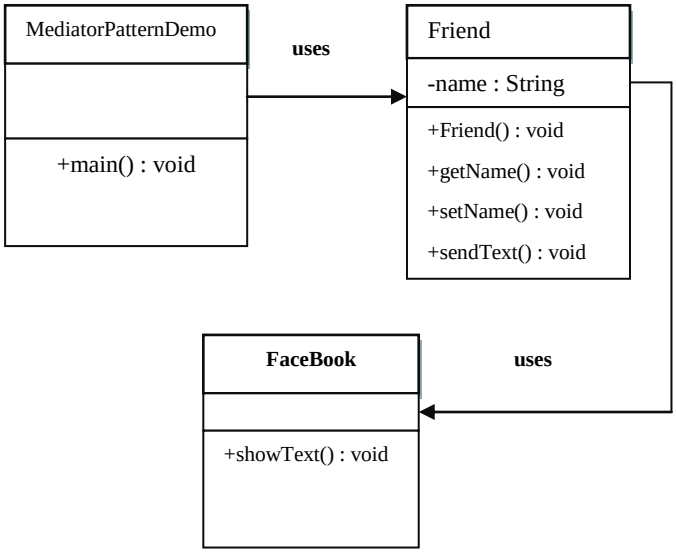

Fig 8: Class diagram of Class FaceBook [2]

5.2 Create Mediator Class

```
import java.util.Date;

public class FaceBook {

public static void showText(Friend friend, String text){

System.out.println(new Date().toString()

+ " [" + friend.getName() +"] : " + text);

}

}
```

In the above code synopsis mediator Class is created and the utility *Date* indicates the communication period among objects.

5.3 Create Friend Class

```
public class Friend {

private String name;

public String getName() {

return name;
```

```
}
public void setName(String name) {
this.name = name;
}
public Friend(String name){
this.name = name;
}
public void sendText(String text){
FaceBook.showText(this,text);
}
}
```

Class *friend* is created from above code synopsis and finally by using the *Friend* object we can show communication between two *facebook friends* as follow in the code [2].

MediatorPatternDemo.java

```
public class MediatorPatternDemo {
public static void main(String[] args) {
Friend ali = new Friend("Ali");
Friend ahmad = new Friend("Ahmad");
ali.sendText("Hi! Ahmad!");
ahmad.sendText("Hello! Ali!");
}
}
```

Place Tables/Figures/Images in text as close to the reference as possible (see Figure 1). It may extend across both columns to a maximum width of 17.78 cm (7”).

Captions should be Times New Roman 9-point bold. They should be numbered (e.g., “Table 1” or “Figure 2”), please note that the word for Table and Figure are spelled out. Figure’s captions should be centered beneath the image or picture, and Table captions should be centered above the table body

6. CHIDAMBER-KEMERER (C&K) METRICS

In this section it is explained that how evaluation of design patterns is performed by C&K metrics, because it is most famous software metrics for object-oriented languages that is proposed by Chidamber and Kemerer [6]. C&K metrics suite originally contains six metrics that are as follows one by one:

6.1 Weighted Method per Class (WMC)

WMC can be described as the total of all method’s complexity of a class.

WMC = number of methods defined in class

If WMC has large value for a particular class, then it is much complicated and extremely expensive to maintain such class. In C&K metrics approach the method’s complexity is not described in particular to permit for most common application of C&K metrics. We can define the complexity of a method as ratio of method’s weight to 128 bytes that is standard weight of a particular method. The “Method’s weight” can be defined as byte size of entire objects contained in a method. Classes that contain large number of methods, limit the possibility of reuse. While evaluating the design patters with C&K metrics we can came know that Mediator design pattern makes coupling among collaborator free. However, it may centralize control in the CnocreteMediator class. The granularity can be low while applying the design patterns.

6.2 Depth of Inheritance Tree (DIT)

DIT can be defined as highest inheritance path from the class to the root class.

DIT = Highest inheritance path from the class to the root class

If a class is much deeper in the given hierarchy, large number of variables and methods to be inherit, that makes a class much complex. The trees that have much depth, indicate larger will be the design complexity of a class. In order to manage such type of complexity, inheritance tool is used. A higher DIT means that design of a class has large number of faults. The recommended value of DIT should be5 or less than 5. The recommended value of DIT for Visual Studio .NET documentation is <= 5, since excessively deeper class hierarchies are much difficult to develop. But some sources recommend the DIT value up to 8. While using the design patterns or without design patterns, it may be no significant variation among these two situations. However, we can promote composition of objects by using the design patterns before inheritance of the class.

6.3 Number of Children (NOC)

NOC can be defined as number of instant children or subclasses.

NOC = Number of immediate subclasses of a class

The larger the value of NOC for a particular class, larger will be the influence on that class; therefore addition testing will be required for the methods in that class. NOC is used to measure class hierarchy’s breadth, while DIT is used to measure depth of the tree. Depth is better as compared to breadth, as it promotes reusability via inheritance. As we discussed number of design patterns in previous sections, and some of these design patterns occupy abstract classes that necessarily have number of children or subclasses such as class Strategy in the Strategy design pattern, class command in the Command design pattern, and class Visitor in the Visitor design pattern. These classes increase value of NOC.

6.4 Coupling Between Objects (CBO)

CBO can be defined as number of classes or objects to be coupled through use of attribute or method.

CBO = Number of classes to which a class is coupled

Larger the number of coupling between objects or classes, higher will be the sensitivity to modification in other components of a design; therefore maintenance will be more complex too. Low CBO is desirable, since more coupling among object classes is harmful to modular design and avoids reusability. So, to improve modularity and reusability it is desirable to reduce coupling among classes. We can apply C&K metric for calculating CBO, and we also know that in the Creational design patterns such as Abstract Factory, Factory Method, and Builder design patterns have the responsibility to create objects, requires knowing class name of a particular object. Therefore, class ConcreteFactory in the design pattern Abstract Factory, class ConcreteCreator in the design pattern Factory Method, class ConcreteBuilder in the design pattern Builder, and the class Mediator of Mediator

design pattern from Behavioral patterns have high CBO value, which is not desirable. As design patterns represent which classes have the responsibility of creation of objects, it is not complex to manage such classes, opposite to C&K's calculations.

6.5 Response for a Class (RFC)

RFC can be defined as number of methods in a group of entire methods, which can be invoked in reaction to a message sent to an object of a particular class.

RFC = M + R (Initial Step Measure)

RFC' = M + R' (Complete Measure)

Where "M" represents number of methods in a class, "R" represents number of remote methods that are openly by class's methods, and "R'" represents the number of remote methods that are recursively called via complete call tree. If the RFC value for a particular class will be high, then the complexity of that class will also high, therefore much complex to maintenance. The class ConcreteMediator in Mediator design patterns has high RFC value, and class ConcreteVisitor in Visitor design pattern often invokes the methods that are described in the object "ConcreteElement", therefore class ConcreteVisitor also have the high RFC value.

6.6 Lack of Cohesion in Methods (LCOM)

LCOM defined as by deducting the number of methods with shared instance variables from number of methods without shared instance variables. Initially this C&K metric is set to zero whenever the calculation is negative. High cohesion increases maintainability, while low cohesion maximizes the complexity and hence, difficult for maintenance. LCOM values are desired to be high.

7. CONCLUSION

In this paper we illustrate differentiation and evaluation among some types of each creational, structural and behavioral pattern. Creational patterns are useful for the creation of objects and these permits the objects to be came into existence in a system devoid of to recognize a particular class type in the code, so programmers do not need to write large and complex code in order to instantiate an object. As discussed in structural patterns form bigger structures from individual parts, usually of different classes and these patterns show a discrepancy a big deal in which the dependency on what kind of structure is being created for a particular task. The interaction among different objects is related to behavioral patterns. Behavioral patterns have the objective that in what manner communication is take place among different objects. These patterns decrease the communication complexity flow charts to simple interconnections among objects of different classes [5]. Design patterns recommend the relationships between different classes. In the described C&K metrics suite, the NOC, WMC, LCOM, and DIT are the metrics necessarily for a single class, not appropriate for the measurements of relationships between classes. For this purpose the metrics RFC and CBO are used to capture the degree of communication among such classes [11, 12]. The table given below summarizes the above discussed metrics and additional metrics such as number of classes and lines of code.

Table 1: Summary of C&K Metrics [11, 12]

Metric	Desirable Value
Weighted Methods Per Class (WMC)	Low
Depth of the Inheritance Tree (DIT)	Low
Number Of Children (NOC)	Low
Coupling Between Objects (CBO)	Low
Response For a Class (RFC)	Low
Lack of Cohesion in Methods (LCOM)	Low
Number of Classes	High
Lines of Code	Low

8. REFERENCES

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1993 Design Patterns – Elements of Reusable Object-Oriented Software.

[2] Design Patterns in Java Tutorial, www.tutorialspoint.com/design_pattern/design_pattern_tutorial.pdf

[3] Aleksandar Bulajic. 2011. Design Patterns Past and Future.

[4] Walter Zimmer. Relationships between Design Patterns.

[5] Design Patterns.,

http://www.gofpatterns.com/design patterns/module2/three-types-design-patterns.php.

[6] The Simple Factory Pattern and Factory Method Pattern. 2011.

http://web.cs.dal.ca/~jin/3132/lectures/dp-07.pdf

[7] Factory Design Pattern. 2011

web.engr.oregonstate.edu/~cscaffid/courses/CS361.../more patterns.pdf

[8] James W. Cooper. 2000. Java Design Patterns: A Tutorial. ISBN: 0-201-48539-7

[9] Adapter design pattern in java. 2012.

http://javapostsforlearning.blogspot.com/2012/09/adapter-design-pattern-in-java.html

[10] Partha Kuchana. 2004. Software Architecture Design Pattern in Java.

[11] Basili, V. R, Braid, L. and Melo, W.L. 1995. A Validation of Object-Oriented Design Metrics as Quality Indicators.

[12] Chidamber & Kemerer object-oriented metrics suite. Project Analyzer v10.2.

http://www.aivosto.com/project/help/pm-oo-ck.html