EVALUATING AND IMPROVING DESIGN PATTERNS APPLICABILITY WITH METRICS

Imène Issaoui¹, Nadia Bouassida², Hanène Ben-Abdallah¹,
Mir@cl Laboratory,
¹Faculty des Sciences Economiques et de Gestion
²Institut Supérieur d’Informatique et de Multimédia
Sfax University, Tunisia
imene.issaoui@gmail.com Nadia.Bouassida@isimsf.rnu.tn Hanene.BenAbdallah@fsegs.rnu.tn

Abstract
Design patterns are established solutions for recurring problems. They represent high-level abstractions that reflect domain experiences. Nowadays, design patterns use is widely spread and they are very well-known. Despite, their popularity, there is no method or way to verify the correct applicability of the pattern. In fact, in many cases, once a designer has reused a pattern in its application, he wants to know if the pattern instantiation is well-designed and if it can be improved. This paper presents an approach that evaluates the pattern instantiation through metrics. Then, it enhances the pattern design through suggesting pattern improvements.

Key Words
Design pattern, metric, design improvement, pattern evaluation, pattern instantiation quality.

1. Introduction

Design patterns have a great importance in software engineering since they provide general solutions to recurring problems. Moreover, they increase maintainability, reusability and understandability [1].

However, all the promised advantages of design patterns could not be attained, if they are not reused correctly in a certain application. In fact, in some cases patterns are inappropriately used due to the lack of experience or due to a bad comprehension of the problem. Till now, it is not clear how to measure the impact that has the application of design patterns and it is hard to judge whether a design pattern is applied appropriately or not. In fact, the evaluation of patterns application is somewhat neglected and quality measurements were practiced independently of patterns.

Thus, we argue that it is necessary to evaluate quality and analyze the correct application of design patterns. For this purpose and in order to control pattern instances quality we must provide a measurable basis which can be achieved with the use of metrics.

Software metrics are widely used in software engineering. They are exploited to measure different characteristics of a software system, mainly in order to have an approximate evaluation of its complexity and its quality. However, till now there is no real link between design patterns and metrics.

In this paper, we provide an approach that focuses on the relationship between design patterns, quality attributes, OO Design properties and metrics values. In fact, our approach verifies the efficiency of a pattern instantiation to realize the quality properties promised by a pattern and to ensure the objective or intention of the pattern. Once, we find that the pattern is not applied correctly, our approach suggests improvements, by indicating what is missing in the pattern and what elements have to be added restructured (e.g., a method lacks, the number of attributes is insufficient).

Many works were interested in design patterns, some of them were interested in their representation, to ensure a good comprehension of patterns and thus their correct instantiation. Others were interested in bringing assistance during design patterns reuse or instantiation. However, no work gives a feedback after the pattern was instantiated.

In fact, many researchers were interested in verifying the claimed efficiency of design patterns and the impact of their use on design quality, using metrics [2], [3] [4]. These works concluded that existing metrics are not adopted for design patterns evaluation and that there is a need to create other metrics that are adopted to design patterns.

Note that, evaluating the impact of design patterns on the quality is not the focus of this paper. In this paper we are not trying to judge if patterns improve the quality of software, we assume that and we try just to evaluate if a certain pattern instantiation is correct and if it can be improved.

We believe that patterns do not improve all the quality aspects, but each pattern is devoted to a certain quality aspect. For example, the pattern strategy aims to ensure a better polymorphism [2], flexibility [1] and reduce complexity. In addition it provides for high cohesion.

As a consequence, we believe that we cannot judge pattern’s quality in general, for example, if we apply the metrics on one pattern, we will find that the complexity of such class will increase whereas coupling will decrease. It is clear that it is judicious to consider each quality attribute independently and to consider, also, the intent behind the pattern applicability rather than to decide on the quality generally.
As result, if a pattern which goal is to decrease coupling is instantiated and the instance does not conform to this intent or the coupling metrics exceed the thresholds, our approach indicates to the designer the existence of an inappropriate instance of such pattern.

In this work, we focus on making a link between metrics and patterns instantiation and also on fixing metrics thresholds. For example, the Observer pattern has a NOM (Number Of Methods) value greater than 2 based on the assumption that the Observer must have two methods “Update” and “Notify”. Thus, each application of the Observer pattern having the NOM value below 2 will be indicated to the designer advising him to improve his instance through adding methods to both classes: Observer and subject.

This paper is organized as follows: Section 2 summarizes related work; Section 3 introduces the OO software metrics used and our approach, which measures and evaluates the correct applicability of a pattern, since it presents the relations between each design pattern and some metrics values. Then, the approach is illustrated through an example. Section 4 presents a case study. Finally, section 5 presents a conclusion and presents future work.

2. Related Works

A large number of researchers have reported results on design patterns applications and also on finding design patterns in object-oriented software designs [5] [6] [7] [8] [9], however only a small amount of work on evaluation of the qualitative effectiveness has been reported.

Reißing [4] studies the impact of pattern use on design quality. For this purpose, he applies classic object-oriented design metrics to two similar designs A and B. Knowing that B uses design patterns and A does not use them, the metrics show that the design A is better than B (because it has less classes, operations, inheritance, associations, etc.). For this reason, Reißing suggests a more appropriate notion of quality that includes both views, the traditional design metric view based on size, coupling, and other complexity criteria and the flexibility considerations inherent in design patterns.

The interesting aspect in this work is that Reißing [4] challenges to the quality definition. We remark that assets the quality basing only on the design patterns is insufficient. Hence, a relationship between design patterns and metrics should be established and the quality basing on both of them have to be redefined. Nevertheless, the authors make their conclusion basing on just a simple example. Such conclusion should be done on important designs from open source systems. Besides, we find that metrics values calculated on both designs A and B are convergent. There is only little difference in the coupling metrics values. We question if this difference is enough to challenges the quality.

Abul Khaer et al. [5] propose an approach for measuring design quality of software using design patterns. The authors proved that the design quality of OO software systems can be affected by proper usage of design patterns through an experimental evaluation of the impact of design patterns on object oriented software system. The results obtained in this experiment are confronted with metrics values calculation. Furthermore, [5] believes that OO metric cannot describe OO systems properly. According to authors, a high quality implies a high number of design patterns is used. This assumption is derived from the high usage of design patterns in all standard object oriented systems.

One advantage of this work is that the usage of design pattern is expressed in term of quality i.e. patterns aim for reusability but there is not an precise guideline of how much the usage of design patterns will be useful. However, it is more interesting to combine between design patterns identification and metrics values, since a relationship exists between a design pattern and the categories of metrics. Moreover, metrics are a strong contribution in the oo systems measurement. Many works are based on metrics for detection and correction of design defects [6] [7] [8] [9]. For instance, a strategy promotes the coupling, this is why the CBO and RFC values are high. In addition, we think that "a high number of design patterns" and "high quality" are insufficient to developers. Instead of a number of whole design patterns, it is better to present the number of each design pattern detected because each one promotes a quality attribute and thus to present an assessment of the quality.

Masuda et al. [11] use the metrics suite of [12] to evaluate the efficiency of applying design patterns. They collect the metrics values for two kinds of applications for Knowledge discovery and data mining (KDD) developed by their research group. The results obtained showed that particular design patterns have a tendency to make a particular metric value poorer. It therefore does not necessarily mean that those design patterns have disadvantages. Instead they suggest that new metrics should be proposed for the evaluation of the efficiency of applying design patterns. The main limitation of this work is that the authors used small applications which are developed by their research group to validate their approach. In fact, they must use open source projects and software architectures reusing different and combined design patterns such as JHotDraw [13].

Hernandez et al. [14] propose an approach that selects metrics for predicting the appropriate application of design patterns. They present an experiment for measuring the utility of metrics and their success in predicting correct usage of design patterns. They make the hypothesis that the longer a design pattern stays in the code, the higher the possibility that it is a code of a high quality. According to Hernandez et al. if a design pattern is removed, then it was not applied correctly. The main drawback of this approach is that their assumption is unconvincing because the removal of design pattern may be due to any change of application requirements. In addition the chosen metrics are not adapted for the
prediction of correct usage of design pattern and they do not have thresholds.
We remark that all the previous work evoked the relationship between design patterns and metrics, but the direct impact of metrics values on the attribute quality of the design pattern is somewhat neglected. Furthermore, it is clear that the term “quality” is not yet well defined in the context of design patterns application and metrics values calculation.
Yet, it is interesting to propose a novel approach which does not only assess quality of design pattern instantiation but also makes recommendation to user on how he can improve the quality of patterns application.

3. Design Pattern Instantiation Evaluation Approach

Our approach for the evaluation of a design pattern instantiation is based on OO software metrics. For this reason, we have selected a set of most pertinent metrics, measuring quantifiable properties, from the works of Chidamberer and Kemmerer [12]. In this section, we begin by presenting the different metrics. Afterwards, we explain their use to evaluate the correctness of a pattern instantiation. Thus, in order to provide a qualitative evaluation of the effectiveness of applying design patterns, we apply metrics on the class and sequence diagrams and use structural and behavioral information.

3.1 Useful metrics

Many researchers have been interested in software metrics and particularly in object-oriented specific metrics (c.f. [15], [16] and [12]). Chidamber, [12] presents a state of the art of object-oriented metrics and classifies them essentially into four categories: coupling, cohesion, complexity and inheritance.

- **Complexity**: Complexity measures the simplicity and understandability of a design. Many complexity measures have been proposed in the literature, we present the most useful ones which are NAtt and WMC (see Table 1 for more details).

- **Cohesion**: Cohesion is a measure of how strongly-related and focused the various responsibilities of a class. A cohesive class is a class which all its methods are tightly related to the attributes defined locally. The cohesion should be maximized to get a design with a good quality. The essential metrics measuring the cohesion is LCOM [12]. Note that the larger the number of similar methods, the more cohesive the class is [12].

- **Inheritance**: Inheritance measures the tree of inheritance and the number of children. In this category, we find DIT and NOC [12] (Table 1.)

- **Coupling**: Coupling measures the degree of interdependency between classes/objects. Two objects are coupled if and only if at least one of them acts upon the other, i.e. any action by methods of object X on methods or instance variables of the object Y and vice versa [12]. The coupling could, essentially, be measured with the CBO and RFC metrics. Note that, the CBO value should be minimized, first, because, when it increases, the sensibility to changes is higher and therefore maintenance is more difficult [12]. Moreover, it should also be minimized since the larger the number of methods that can be invoked by a class, the greater the complexity of the class is [12].

<table>
<thead>
<tr>
<th>Metric</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cohesion</strong></td>
<td>LCOM «Lack Of Cohesion in Methods » [12]</td>
</tr>
<tr>
<td></td>
<td>number of pairs of methods without shared instance variables, minus the number of pairs of methods with shared instance variables. Low cohesion increases complexity, and therefore maintenance is more difficult.</td>
</tr>
<tr>
<td><strong>Complexity</strong></td>
<td>NAtt Number of attributes in a class.</td>
</tr>
<tr>
<td></td>
<td>NOM [31] Number Of Methods in a class.</td>
</tr>
<tr>
<td></td>
<td>WMC [12] is a weighted sum of all the methods defined in a class. The larger the WMC value for a class, the more complicated and expensive it is to maintain the class.</td>
</tr>
<tr>
<td><strong>Inheritance</strong></td>
<td>DIT «Depth of Inheritance » [12]</td>
</tr>
<tr>
<td></td>
<td>is the length of the longest path from a given class to the root class in the inheritance hierarchy.</td>
</tr>
<tr>
<td></td>
<td>NOC «Number of children » [12]</td>
</tr>
<tr>
<td></td>
<td>is a count of the number of immediate child classes that have inherited from a given class.</td>
</tr>
<tr>
<td><strong>Coupling</strong></td>
<td>CBO «Coupling Between Objects » [12]</td>
</tr>
<tr>
<td></td>
<td>is a count of the number of other classes to which a given class is coupled and, hence, denotes the dependency of one class on other classes in the design. When the number of couplings is large, maintenance is more difficult.</td>
</tr>
<tr>
<td></td>
<td>RFC «Response For Call » [12]</td>
</tr>
<tr>
<td></td>
<td>is the count of the methods that can be potentially invoked in response to a message received by an object of a particular class. The larger the number of RFC values for a class, the greater the complexity of the class.</td>
</tr>
</tbody>
</table>

Note that CBO and RFC metrics are calculated using sequence diagram, while DIT, NOC and WMC are calculated using the class diagram. The CBO is number of objects (in the class diagram) with which the class
exchanges messages. In addition, the RFC is the number of methods of the class plus number of methods of other classes called. Note also, that the WMC and NOM are the same since we are at the design level.

3.2 The threshold values for metrics

Choosing useful metrics and relating them to patterns is not enough to ensure that our approach is applicable; it is necessary to fix the threshold values which highly influence the efficiency of the evaluation. We should caution that, even in the software engineering field, in general, there is not yet a precise guideline for how to fix thresholds. In fact, the threshold problem is far from being new.

Marinescu [6] proposed three means to fix metric thresholds: 1) experience and hints from the literature, it consists in using metrics from the literature with already predefined thresholds; this might require the adaptation of the thresholds to the system size [6] 2) a tuning machine that finds automatically the proper threshold values. Note that, this requires a large repository of design fragments containing patterns; 3) analysis of multiple versions of designs.

In our case, we have chosen the first mean to fix thresholds. In fact, there are several defined threshold metrics; for instance, DIT is fixed to six according to [17]. The (WMC) threshold limit is set to 15 per class. Chandra et al. [18] suggest a threshold value of 6 for DIT metric. The NOC threshold limit is set to 6 as for DIT. The threshold limit for CBO metric is set to 8 per class. For the RFC metric the threshold limit is set to 35 per class, finally the threshold limit for the LCOM metric is set to 1 per class.

### Table 2

<table>
<thead>
<tr>
<th>Metric</th>
<th>Threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>WMC</td>
<td>0-15</td>
</tr>
<tr>
<td>DIT</td>
<td>0-6</td>
</tr>
<tr>
<td>NOC</td>
<td>0-6</td>
</tr>
<tr>
<td>CBO</td>
<td>0-8</td>
</tr>
<tr>
<td>RFC</td>
<td>0-35</td>
</tr>
<tr>
<td>LCOM</td>
<td>0-1</td>
</tr>
</tbody>
</table>

Note that, a metric value is considered high if it is greater than the half of the threshold defined in Table 2 and it is considered low if it is lower than the half of the threshold fixed in Table 2. Indeed, it is essential to make an empirical study on corpus containing patterns to determine the right thresholds values.

3.3 The relation between metrics and patterns

In this section, we show how design patterns are related to quality attributes and OO design properties. Table 3 presents, only, some of the GoF patterns, the other design patterns will be presented in a future work.

### Table 3

Design patterns relationship with C&K metrics

<table>
<thead>
<tr>
<th>D. P</th>
<th>OO Design properties</th>
<th>DesignQuality attribute</th>
<th>Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mediator</td>
<td>decrease coupling</td>
<td>increase reusability</td>
<td>high for the Class</td>
</tr>
<tr>
<td></td>
<td>decrease complexity</td>
<td>increase reusability</td>
<td>Colleague</td>
</tr>
<tr>
<td>Command</td>
<td>decrease complexity</td>
<td>increase flexible</td>
<td></td>
</tr>
<tr>
<td></td>
<td>increase cohesion</td>
<td>increase reusability</td>
<td></td>
</tr>
<tr>
<td>Strategy</td>
<td>increase cohesion</td>
<td>increase flexibility</td>
<td>high for the class</td>
</tr>
<tr>
<td></td>
<td>decrease complexity</td>
<td>increase flexibility</td>
<td>Strategy</td>
</tr>
<tr>
<td>State</td>
<td>increase cohesion</td>
<td>increase flexibility</td>
<td>high for the class</td>
</tr>
<tr>
<td>Factory Method</td>
<td>decrease complexity</td>
<td>increase flexibility</td>
<td></td>
</tr>
<tr>
<td>Visitor</td>
<td>decrease complexity</td>
<td>increase polymorphism</td>
<td>high for the class</td>
</tr>
<tr>
<td>Abstract Factory</td>
<td>decrease complexity</td>
<td>increase reusability</td>
<td></td>
</tr>
</tbody>
</table>
In the literature, quality metrics for object oriented software are mostly used to identify design defects [23] [24] [7] [8] [9]. But, there are other works that apply metrics on patterns, in order to justify the benefits of the pattern in terms of quality [4] [5] [2] [10] [14].

In our turn, we find that the instantiation of patterns leads to quality deterioration when the instance is not appropriate. In our opinion, an appropriate instance is one that respects the values of metrics required as shown in table3. For example, an appropriate instance of the Mediator pattern implies a high coupling for the class Mediator since it plays a central role in the application and it needs to know about other Colleagues, the CBO value therefore tends to be high. Moreover, the Mediator pattern centralizes control in class ConcreteMediator. This increases the complexity of this class. The WMC value of class ConcreteMediator increases. In addition, the Mediator pattern involves the abstract classes Colleagues which essentially have a large number of children ConcreteColleague, for this reason the NOC Values of the class Colleague becomes high. Also the RFC value of ConcreteMediator tends to be high because it communicates with many other colleague objects via message passing. Furthermore, the class ConcreteMediator receives a lot of different requests in place of its colleague objects. As a result, it has various methods which are unrelated to each other; therefore the LCOM value tends to be very high when instantiating design patterns.

Note that, we consider a metric value as high if it is superior to the half of the threshold defined in Table 2 [18] and we consider it as low if it is inferior than the half of the threshold fixed in Table 2. In fact, it is essential to make an empirical study on open source projects and software architectures reusing different and combined design patterns such as JHotDraw [13] determine the right thresholds values.

3.4 Example

To illustrate the relation between patterns use and metrics, as it was presented in Table 1, we suppose we have a design without the use of patterns and then, the same design is restructured using the state design pattern and we want to verify that the metrics characterizing the state pattern have been improved, as stated in Table 1.

Figure 1 shows the class diagram of the design, before applying the state pattern, figure 2 shows its sequence diagram. Figure 3 introduces the State pattern into the design of figure 1, figure 4 shows its sequence diagram. The class diagram of Figure 1 illustrates an example of a car rental. In this example, the charge for each rental is calculated by the Car Class which knows the price code and which is forwarded the number of days rented from Rental. The price codes are modeled by an enumeration type. In figure 2, the enumeration type for the price code is replaced by the abstract class Price and its subclasses. For each price code there is a subclass of Price that calculates the charge when given the number of days the car was rented.

State pattern allows an object to change its behavior when its internal state changes [1]. The object will behave as if it is changing its class. This pattern is used especially when there is a state dependency of an object. If an object is required to change its behavior at runtime depending on its state this pattern is very suitable. Moreover, if there are large numbers of conditional statements which are about the states of an object, this pattern should be used. State pattern solves these problems by implementing each state as a separate class.

State pattern provides flexibility [1] and promotes High Cohesion, Polymorphism just like Strategy pattern [20]. The main difference between strategy and state pattern is that, state pattern focuses on the state changes of an object whereas strategy pattern focuses on the type variations of an object. If a class has different states and behaviors, state pattern decreases the complexity of that class. As a result, weighted method per class (WMC) decreases. However, this pattern adds new associations to the objects, therefore coupling between objects (CBO) slightly increases.
Table 4 confirms the metric values, presented in Table 3. We remark that the NOC and CBO of the class Price which plays the role of the state increased, in addition the RFC values of the context class increased, but the WMC values of the ConcreteState are low (equal to 1). However the DIT and LCOM values do not change while applying the state design pattern.

Table 4
Metric values for the designs before and after applying patterns

<table>
<thead>
<tr>
<th></th>
<th>DIT</th>
<th>NOC</th>
<th>CBO</th>
<th>RFC</th>
<th>WMC</th>
<th>LCOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before</td>
<td>State=0, State=0</td>
<td>State=2, State=0</td>
<td>context=0, context=1</td>
<td>concreteState=0, concreteState=1</td>
<td>0 for all class</td>
<td>0 for all class</td>
</tr>
<tr>
<td>After</td>
<td>State=1, State=3</td>
<td>State=1, State=1</td>
<td>context=1, context=1</td>
<td>concreteState=1, concreteState=1</td>
<td>0 for all class</td>
<td>0 for all class</td>
</tr>
</tbody>
</table>

Figure 5 presents the class diagram of the example, and figure 6 shows its sequence diagram. In this example, we remark that the Billing component delegates to a tax calculator to decide how much sales tax to add to the invoice. This object determines if California tax should be computed.

Figure 5 A design example

4. Case Study: using metrics to identify inappropriate application of design pattern

In this section we introduce one illustration that presents an inappropriate application of design pattern. To demonstrate our approach, let us consider the design fragment (see figure 5) which involves the Strategy design pattern [1]. This pattern is useful when there are different algorithms of the same behavior. The clients are not aware of the algorithm, they just know the behavior. By this way, algorithms are not coupled to the client and may be implemented independently. The Strategy pattern mainly promotes polymorphism [19].

Moreover, each created subclass is focused on only one job, this in turn increases cohesion. In addition, the Strategy Pattern promotes the High cohesion principle and it reduces complexity [19]. Thus, the Strategy pattern reduces the WMC [20] and it increases CBO and RFC. Figure 5 shows its sequence diagram. In this example, we remark that the Billing component delegates to a tax calculator to decide how much sales tax to add to the invoice. This object determines if California tax should be computed.

As illustrated in figure 5 the class Billing plays the role of Context, the TaxCalculator class plays the role of Strategy and CaliforniaTax class plays the role of ConcreteStrategyA. These different roles are obtained thanks to the structural identification proposed in our previous work [25].
To evaluate the quality of this design fragment, we calculate the quality metrics defined by [12]; Table 5 shows the values of these metrics.

### Table 5
Calculated metrics on the Billing example

<table>
<thead>
<tr>
<th>Design shown in fig 7</th>
<th>DIT</th>
<th>NOC</th>
<th>CBO</th>
<th>RFC</th>
<th>WMC</th>
<th>LCOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tax Calculator = 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>California Tax = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Billing = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Billing = 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>California Tax = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The NOC value of the Strategy class is equal to one. Such value does not reflect the intent behind the reuse of the strategy pattern, since it enhance the polymorphism. Therefore, we recommend to the designer to study the Strategy behaviors and designing missed behaviors as subclasses (ConcreteStrategy) of the Strategy class without exceeding the threshold already set which is equal to 6.

Concerning the DIT value, it does not become very high when instantiating the strategy design pattern. The CBO value of the Context class is low. This value does not conform to Table 3. The Context Class plays a central role since it calls the convenient ConcreteStrategy to handle some behavior. Thus, the Context class needs to know about other ConcreteStrategy which increases the coupling between Context and ConcreteStrategy classes. Consequently, the CBO value must be high. Note that, when increasing the NOC of the Strategy class, implicitly the CBO value will increase.

On the other hand, the coupling increases through message passing, therefore the RFC value of the Billing class playing the role of Context class must be high which is not realized in this example.

When separating behaviors of the Strategy class into ConcreteStrategy classes for each behavior, it makes them small classes with a low value of WMC. For these reasons, we suggest to the designer to add method for ConcreteStrategy taking into account the threshold of low WMC.

### 5. Conclusion
Design patterns capitalize the knowledge of expert designers and offer reuse that provides overall faster development and a higher design quality. Each pattern has a certain objective and is devoted to a certain quality aspect that could be measured with metrics. Thus, any instantiation of a certain pattern must satisfy certain metric values. As a result, if the quality aspect that must result for the pattern instantiation is not satisfied then the design is a bad pattern instantiation. In this work we propose an approach that verifies and evaluates the correct pattern instantiation through metrics. Then, it enhances the pattern design through suggesting pattern improvements.

Our future work includes an empirical study of different open source systems using patterns, in order to validate our proposal. Moreover, we are also interested in proposing new metrics appropriate for design patterns.

### References


