

# On the Role of Software Metrics in Applying Design Patterns

**Niloofar Khedri**  
Database Research Group  
Fac. of ECE, School of  
Engineering  
University of Tehran, IRAN

**Masoud Rahgozar**  
Control Intelligent Processing  
Center of Excellence  
Fac. of ECE, School of Engineering  
University of Tehran, IRAN

**Mahmoud Reza Hashemi,**  
Database Research Group  
Fac. of ECE, School of  
Engineering  
University of Tehran, IRAN

**Abstract** - *Design patterns describe good solutions to common and reoccurring problems in program design. Applying design patterns in software design and implementation have significant effects on software quality metrics such as flexibility, usability, reusability, scalability and robustness. Applying design patterns in software systems does not have a specific rule. It is possible to apply two design patterns for a specific problem, so a decision should be taken on choosing one of the two design patterns. In this paper we propose a solution to choose and apply patterns according to the required quality metrics.*

**Keywords:** Design patterns, Software Quality Metrics, Software Engineering

## 1 Introduction

Design patterns [12] are high level building blocks that promote elegance in software by ordering proven and timeless solutions to common problems in software design. Design patterns convey the experience of software designers. Applying design patterns in software design and implementation have important effects on software quality metrics such as flexibility, usability, reusability, scalability and robustness [5]. In [8] design patterns' relationship are classified in 6 categories and then a new way for applying patterns in the software system is given. As we know studying software characteristics in the software design is an essential content but no consideration on applying patterns based on their expected software metrics in [5] has been studied yet.

In this paper we investigate situations in applying design patterns where two patterns can be applied. So we can decide on choosing similar patterns for a specific problem, according to the software metrics that the software system need to promote or increase. It means that software quality metrics may be the key for applying design patterns.

In this paper, first we talk about the quality metrics which design patterns are expected to bring. Second, we classify the design patterns based on their relationships. Then we study two relationships, "Similar" and "Conflict", and compare their design patterns' software quality metrics. At last we come to a new method on choosing patterns to apply in software systems.

## 2 Software Quality characteristics of Design Patterns

### 2.1 Quality Characteristics related with Design Patterns

Design Patterns are solutions for reoccurring problems, applying design patterns in software design and implementation have effect on software quality metrics such as flexibility, usability, reusability, scalability and robustness.

Gamma et al. in "Design Patterns: Elements of Reusable Object-Oriented Software" define design patterns as: "Patterns specific design problems and make object-oriented more flexible, elegant and ultimately reusable" and Design patterns help you chose design alternatives that make a system reusable and avoid alternatives that compromise reusability. [12].

Software elegance is defined as maximizing the information delivered through the simplest possible interface. When considering these definitions, design patterns are expected to bring:

- Flexibility: "effort required modifying an operational program" [4].
- Elegancy: Issues of elegance in software are reflected to robustness, scalability, flexibility, and usability.
  - Robustness: Robustness is the degree to which an executable work product continues to function properly under abnormal conditions or circumstances" [2] D.G. Firesmith. Common concepts underlying safety, security, and survivability engineering. December 2003.]. Also, the attributes related to the correct functioning of a software product in the case of invalid inputs or under stressful environmental conditions. [10]
  - Scalability: Scalability is the ease with which an application or component can be modified to expend its existing capacities" [2] [9] [6].
  - Flexibility: "effort required modifying an operational program" [4].
  - Usability: The capability of software product to be understood, learned, used and attractive to the user, when used under specified conditions" [2] [3] [11].

- Reusability: Reusability is the ease with which an existing application or component can be reused" [2] [7].

So design patterns are expected to increase the following quality characteristics: Flexibility, Reusability, Robustness, Scalability, and Usability.

Flexibility consists of the following quality characteristics:

- Expendability: The degree to which architectural, data or procedural design can be extended"[11].
- Generality: The breadth of potential application of program components" [11].
- Modularity: The functional independence of program components" [11].

Reusability consists of the following quality characteristics:

- Generality
- Hardware independence: The degree to which the software is decoupled from the hardware on which it operates"[11].
- Modularity
- Software system independence: The degree to which the program is independent of nonstandard programming language features, operating system characteristics, and other environmental constraints" [11].

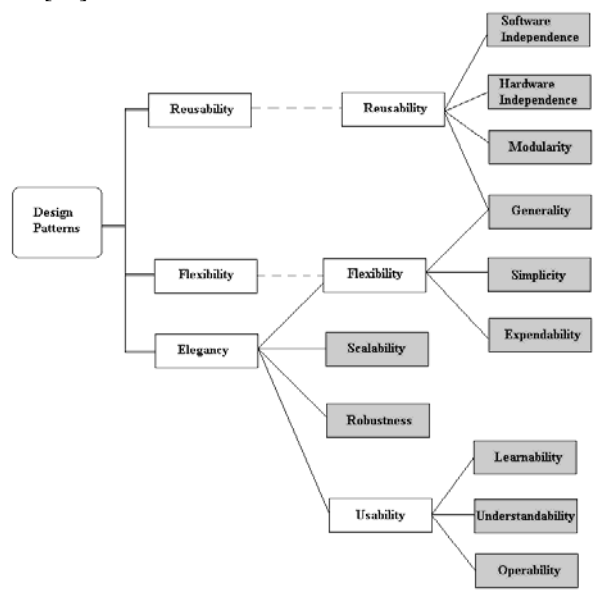


Figure 1 – Design Patterns and their Software Quality Characteristics

Usability consists of the following quality characteristics:

- Learnability: The capability of the software product to enable the user to learn its application" [1] [3].
- Operability: The capability of the software product to enable the user to operate and control it" [3]. Also, the ease of operation of a program"[1] [2] [11].
- Understandability: The capability of the software product to enable the user to understand whether the software is suitable, and how it can be used for particular tasks and conditions of use" [3].

Figure 1 shows the main characteristics and sub characteristics of the design patterns quality.

## 2.2 Quality Evaluation of Design Patterns

Khosravi and Gueheneuc in “A Quality Model for Design Patterns”[5] studied design patterns and evaluated manually their quality characteristics using five-levels scale (Excellent , Good, Fair, Bad and Very bad. Also, they used N/A for characteristics not applicable to some design patterns.)

Figure 2 summarizes the quality characteristics of the twenty-three design patterns. [5]

Design Patterns	Expendability	Simplicity	Generality	Modularity	Software Independence	Hardware Independence	Learnability	Understandability	Operability	Scalability	Robustness
Abstract Factory	Excellent	Excellent	Good	Good	Fair	Fair	Good	Good	Good	Good	Good
Builder	Good	Good	Fair	Fair	N/A	N/A	Fair	Good	Fair	Good	Good
Factory Method	Bad	Bad	Fair	Good	N/A	N/A	Good	Good	Good	Good	Good
Prototype	Excellent	Good	Fair	Good	N/A	N/A	Fair	Good	Fair	Excellent	Good
Singleton	Bad	Very Bad	Fair	Excellent	Fair	Fair	Fair	Fair	Fair	Good	Good
Adapter	Fair	Fair	Bad	Good	Good	N/A	Good	Fair	Fair	Fair	Fair
Bridge	Good	Fair	Good	Good	N/A	N/A	Fair	Fair	Good	Good	Good
Composite	Fair	Fair	N/A	Fair	N/A	N/A	Fair	Good	N/A	N/A	Good
Decorator	Excellent	Excellent	Good	Fair	Good	N/A	Good	Good	Good	Good	Fair
Facade	Good	Good	Good	Good	N/A	N/A	Fair	Good	Fair	Fair	Fair
Flyweight	Bad	Bad	Fair	Good	N/A	N/A	Good	Bad	Fair	Good	Good
Proxy	Good	Bad	Fair	Good	N/A	N/A	Fair	Bad	Good	Good	Fair
Chain of Responsibility	Good	Good	Good	Bad	N/A	N/A	Fair	Fair	Good	Bad	Fair
Command	Good	Bad	N/A	N/A	N/A	N/A	Bad	Very Bad	Good	Good	Good
Interpreter	Good	Fair	Good	Fair	N/A	N/A	Fair	Fair	Good	Good	Fair
Iterator	Excellent	Excellent	Good	Fair	Good	N/A	Good	Fair	Fair	Good	Good
Mediator	Good	Fair	Good	Good	N/A	N/A	Fair	Fair	Good	Good	Fair
Memento	Good	Fair	Fair	Very Bad	N/A	N/A	Bad	Fair	Good	Fair	Bad
Observer	Excellent	Good	Excellent	N/A	N/A	N/A	Fair	Good	Good	Good	Good
State	Good	Good	Fair	Bad	Good	N/A	Fair	Very Bad	Good	Good	Fair
Strategy	Good	Fair	Bad	Fair	Fair	N/A	Bad	Bad	Fair	Bad	Fair
Template Method	Excellent	Good	Fair	N/A	N/A	N/A	Good	Good	Good	Good	Good
Visitor	Excellent	Good	Good	Fair	Good	N/A	Good	Bad	Fair	Good	Fair

Figure 2 – Patterns Quality Characteristics

We can come to conclude from Figure 2 as below:

1. We can not gain hardware independence by using design patterns.
2. Software Independence can be achieved only through using Adapter, Decorator, State and Visitor.
3. In creational patterns (Abstract Factory, Builder, Factory Method, Prototype and Singleton):
  - Expendability and Simplicity can not be achieved through using Factory Method and Singleton and other creational patterns give a reasonable view for Expendability and Simplicity.
  - So except from Factory Method and Singleton, other creational patterns give flexibility to the system design.
  - Using creational design patterns give the system design, a good reusability, usability, scalability and robustness.
4. In structural patterns (Adapter, Bridge, composite, Decorator, Façade, flyweight, Proxy):
  - The only design pattern that has three bad points in software metrics is flyweight which can not achieve a good result in expendability, simplicity and understandability.
  - Except from the Flyweight, All other structural patterns have reasonable result in expendability and simplicity.
  - All the structural patterns have a good view in generality.

- By using Adapter and Decorator can achieve a better Reusability because of the good result of them in software independence.
- Except from the Flyweight and Proxy can achieve usability because of the good result in understandability.
- All the structural patterns have a good view in scalability and robustness.

5. In Behavioral patterns (Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, and Visitor):

- Except from the Command, all other behavioral patterns give the system design a good simplicity result.
- Except from the Strategy, all other behavioral patterns give the system design a good generality result.
- System design can achieve a good result in flexibility through using behavioral patterns except from the Command and Strategy.
- Software independency can be achieved by using State and Visitor.
- There is not any metrics in this field that has a good result by using Mediator pattern.
- All the behavioral patterns has a good view in operability , but because of the bad result of Command , Memento and Strategy in learnability and bad result of Command , state and Strategy in understandability , nothing can be concluded for he usability metric.
- Except from the Chain of Responsibility and Command, all other behavioral patterns have a good result in scalability.
- Except from the Memento, all other behavioral patterns have a good result in robustness.
- By using Strategy, Memento and Command can not achieve at least to three metrics.

### 3 A way to applying patterns based on software quality characteristics

#### 3.1 Classification of Design Patterns Relationships

According to [8] design patterns relationships can be classified through 6 categories:

1. Use: A pattern uses another pattern.
2. Refine: A more specific pattern refines a more general and abstract pattern.
3. Conflict: One pattern conflicts with another pattern when they both provide mutually exclusive solutions to similar problems.
4. Similar: This relationship is often used to describe patterns which are similar because they address the same problem. The similarity relationship seems to be much broader than just conflicts and as it is also used to describe patterns which have a similar solution technique such as Strategy and State.

5. Combine: Two patterns are combining to solve a single problem.
6. Require: One pattern requires a second pattern if the second pattern is a prerequisite for solving the problem addressed by the first pattern.

In software system, during the applying design patterns to a specific problem, there are some cases to choose one pattern form the patterns that can be applied for a problem. In these cases on pattern which is similar to another one or one pattern which is conflicts with another should be chosen.

In next part, patterns which have similar or conflicts with relationship to other patterns are considered to study the effect of choosing one patterns between two on the quality metrics. Figure 3 and Figure 4 show the brief form of the patterns and the software characteristics.

Design Patterns	Flexibility	Reuseability	Usability	Scalability	Robustness
Abstract Factory & Builder	Abstract Factory	Abstract Factory	Abstract Factory	Same	Same
Adapter & Bridge	Bridge	According to software	According to software	Same	Bridge
Proxy & Decorator	Proxy	According to software	Decorator	Same	Same
Façade & Mediator	Façade	Same	According to software	Mediator	Same
State & Strategy	State	According to software	According to software	State	Same

Figure 3 – Comparing Quality Characteristics of Similar Patterns

Design Patterns	Flexibility	Reuseability	Usability	Scalability	Robustness
Builder & Composite	Builder	Builder	Builder	Builder	Same
Factory Method & Prototype	Factory Method	Same	Factory Method	Prototype	Same
Prototype & Template Method	Same	Prototype	Template Method	Prototype	Same

Figure 4 – Comparing Quality Characteristics of Conflict Patterns

### 3.2 Similarity Relationship and Software Metrics

When two patterns are solutions for a problem, they are similar because they addressed the same problem. Patterns with similar relationship and their effect on software quality metrics are shown in Figure 5 and Figure 6.

Pattern Similar to Pattern	
Abstract Factory	Builder
Adapter	Bridge
Decorator	Proxy
Façade	Mediator
State	Strategy

Figure 5 – Patterns with Similarity Relationship

Design Patterns	Expendability	Simplicity	Generality	Modularity	Software Independence	Hardware Independence	Learnability	Understandability	Operability	Scalability	Robustness
Abstract Factory	Excellent	Excellent	Good	Good	Fair	Fair	Good	Good	Good	Good	Good
Builder	Good	Good	Fair	Fair	N/A	N/A	Fair	Good	Fair	Good	Good
Adapter	Fair	Fair	Bad	Good	Good	N/A	Good	Fair	Fair	Good	Fair
Bridge	Good	Fair	Good	Good	N/A	N/A	Fair	Fair	Good	Good	Good
Proxy	Good	Bad	Fair	Good	N/A	N/A	Fair	Bad	Good	Good	Fair
Decorator	Excellent	Excellent	Good	Fair	Good	N/A	Good	Good	Good	Good	Fair
Façade	Good	Good	Good	Good	N/A	N/A	Fair	Good	Fair	Fair	Fair
Mediator	Good	Fair	Good	Good	N/A	N/A	Fair	Fair	Good	Good	Fair
State	Good	Good	Fair	Bad	Good	N/A	Fair	Very Bad	Good	Good	Fair
Strategy	Good	Fair	Bad	Fair	Fair	N/A	Bad	Bad	Fair	Bad	Fair

Figure 6 - Quality Characteristics of Similar Patterns

### 3.2.1 Choosing between Abstract Factory and Builder

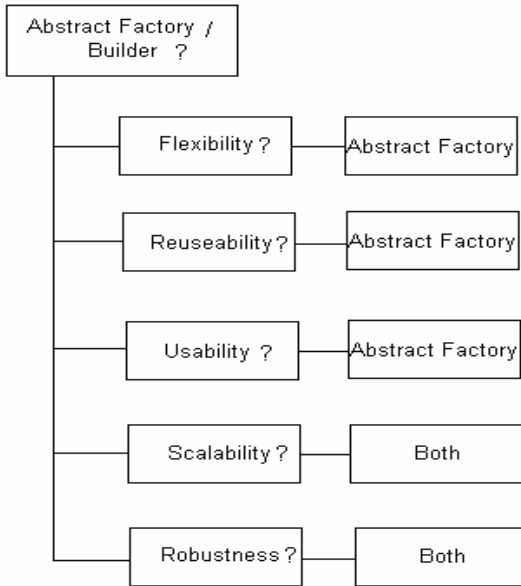


Figure 7- Choosing between Abstract Factory and Builder

Abstract Factory and Builder are two creational patterns. Abstract Factory, provides an interface for creating families of related or dependent objects without specifying their concrete classes. Builder separates the construction of a complex object from its representation so that the same construction process can create different representations. Abstract Factory is similar to Builder in that it too may construct complex objects. The primary difference is that the Builder pattern focuses on constructing a complex object step by step. Abstract Factory's emphasis is on families of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory pattern is concerned, the product gets returned immediately [12]. It seems that Abstract Factory achieve better results in software quality metrics than the Builder (Figure 7).

- Adapter has better results to gain learnability and Bridge has better results to gain operability and both has the same result in understandability, so according to the software system use can choose both of them to achieve usability.
- Bridge has better results to gain robustness than Adapter.

So we can use Abstract Factory in order to gain software quality characteristics instead of Bridge where ever there is no need to construct a complex object step by step.

### 3.2.2 Choosing between Adapter and Bridge

Adapter and bridge are two structural patterns. Adapter converts the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. Bridge, decouples an abstraction from its implementation so that the two can vary independently. Bridge has a

structure similar to an object adapter, but Bridge has a different intent: It is meant to separate an interface from its implementation so that they can be varied easily and independently. An adapter is meant to change the interface of an existing object [12]. We can compare Bridge and Adapter as below (Figure 8):

- Bridge has better results to gain flexibility than the Adapter.
- Bridge has better results to gain software independence and Adapter has better results to gain generality, so according to the software system use can choose both of them to achieve reusability.

Now, we can come to conclusion that if we want to emphasis on learnability and software independence in our software system, we should use Adapter; in other cases we can apply Bridge.

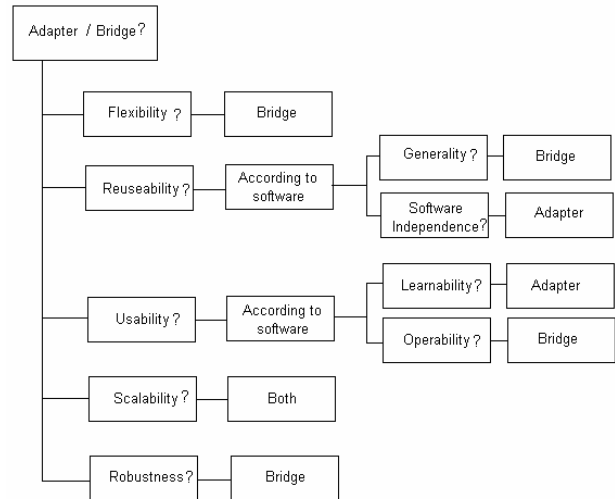


Figure 8 – Choosing between Adapter and Bridge

### 3.2.3 Choosing between Decorator and Proxy

Decorator and Proxy are two structural patterns. Decorator attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality. Proxy provides a surrogate or placeholder for another object to control access to it. Although decorators can have similar implementations as proxies, decorators have a different purpose.

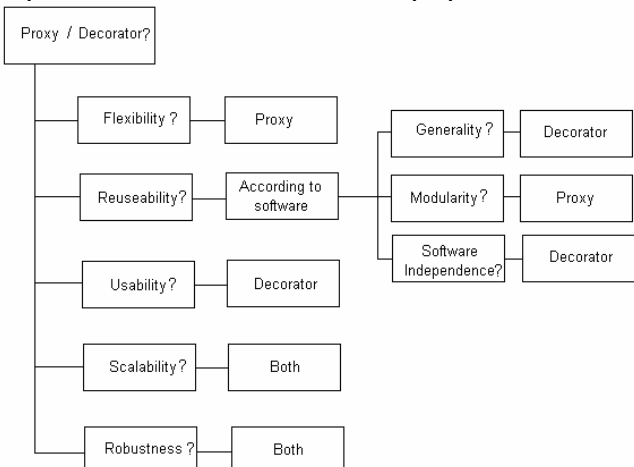


Figure 9– Choosing between Decorator and Proxy

A decorator adds one or more responsibilities to an object, whereas a proxy controls access to an object. Proxies vary in the degree to which they are implemented like a decorator [12]. We can compare these two patterns as below (Figure 9):

- Proxy has better results to gain flexibility than Decorator.
- Decorator has better results to gain software independence than Proxy, so it seems that by using Decorator can achieve better reusability.

Now, we can come to conclusion that if we want to emphasis on generality, software independence and usability in our software system, we should use Decorator and if we want to emphasis on flexibility and modularity in our software system, we should use Proxy.

### 3.2.4 Choosing between Façade and Mediator

Façade is a structural pattern and Mediator is a Behavioral pattern. Façade provides a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use. Mediator defines an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently. Mediator is similar to Façade in that it abstracts functionality of existing classes. However, Mediator's purpose is to abstract arbitrary communication between colleague objects, often centralizing functionality that doesn't belong in any one of them. A mediator's colleagues are aware of and communicate with the mediator instead of communicating with each other directly. In contrast, a facade merely abstracts the interface to subsystem objects to make them easier to use; it doesn't define new functionality, and subsystem classes don't know about it. Façade differs from Mediator in that it abstracts a subsystem of objects to provide a more convenient interface. Its protocol is unidirectional, that is, Façade objects make requests of the subsystem classes but not vice versa. In contrast, Mediator enables cooperative Behavior that colleague objects don't or can't provide, and the protocol is multidirectional [12]. We can compare Façade and Mediator based on their quality characteristics as below (Figure 10):

- Façade can achieve better simplicity than Mediator and both have the same results in expendability and generality so Façade can achieve better flexibility than Mediator.
- Both Façade and Mediator have the same results in reusability.
- Façade can achieve better understandability and Mediator can achieve better operability , so choosing between these two patterns should be according to the software system and the metrics needed.
- Mediator can achieve better scalability than the Façade.

Now, we can come to conclusion that if we want to emphasis on flexibility and understandability in our software system, we should use Façade and if we want to emphasis on operability and scalability in our software system, we should use Mediator.

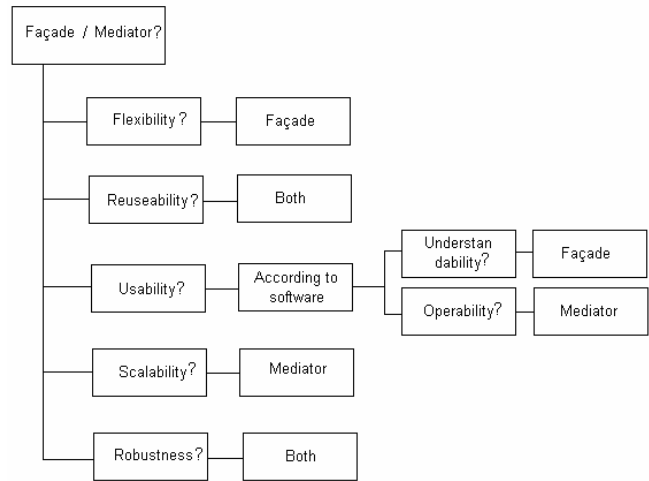


Figure 10– Choosing between Façade and Mediator

### 3.2.5 Choosing between State and Strategy

State and Strategy are two Behavioral patterns. State, allows an object to alter its Behavior when its internal state changes. The object will appear to change its class. Strategy defines a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it [12]. We can compare their quality characteristics as below (Figure 11):

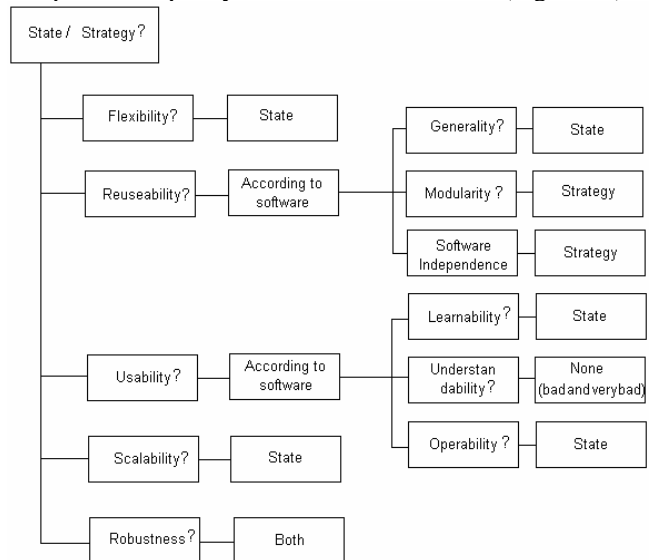


Figure 11 – Choosing between State and Strategy

- State can achieve better results in generality and Simplicity than the Strategy, so State can achieve better results in flexibility.
- State can achieve software independence, but according to the fair value for the State generality and Strategy modularity, there is no result for reusability gained by using these to patterns.

- State can achieve operability but it seems that no result is exists for the usability.
- State can achieve better scalability.
- It seems that by using State can achieve better results in software quality metrics.

Now, we can come to conclusion that if we want to emphasis on modularity and software independence in our software system, we should use Strategy. Neither of the State and Strategy improves the understandability. In other case we can apply State in the software.

### 3.3 Conflict Relationship and Software Metrics

Two pattern conflict with each other when they both provide mutually exclusive solutions to similar problems. Patterns with conflict relationship and their effect on software quality metrics are shown in Figure 12 and Figure 13 :

Pattern Conflict to Pattern	
Builder	Composite
Factory Method	Prototype
Prototype	Template Method

Figure 12 - Patterns with Conflict Relationship

Design Patterns	Expendability	Simplicity	Generality	Modularity	Software Independence	Hardware Independence	Learnability	Understandability	Operability	Scalability	Robustness
Builder	Good	Good	Fair	Fair	N/A	N/A	Fair	Good	Fair	Good	Good
Composite	Fair	Fair	N/A	Fair	N/A	N/A	Fair	Good	N/A	N/A	Good
Factory Method	Bad	Bad	Fair	Good	N/A	N/A	Good	Good	Good	Good	Good
Prototype	Excellent	Good	Fair	Good	N/A	N/A	Fair	Good	Fair	Excellent	Good
Template Method	Excellent	Good	Fair	N/A	N/A	N/A	Good	Good	Good	Good	Good

Figure 13 – Quality Characteristics of Conflict Patterns

#### 3.3.1 Choosing between Builder and Composite

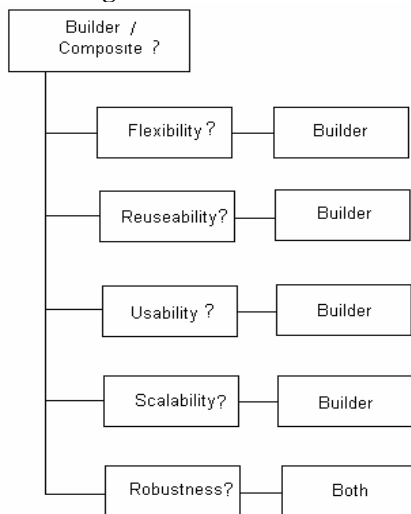


Figure 14 – Choosing between Builder and Composite

Builder is a creational pattern and Composite is a structural pattern. Builder separates the construction of a complex object from its representation so that the same construction process can create different representations. Composite composes objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. A Composite is what the builder often builds

[12]. We can compare Builder and Composite as below (Figure 14):

- Builder can achieve better results in expendability, Simplicity and generality than the Composite, so Builder has better results in flexibility.
- It seems that Builder has better results in reusability and usability, because of several not applicable values for software quality metrics for Composite.
- Builder can achieve better results in scalability than the Composite.

Now, we can come to conclusion that if we want to compose objects into tree structures to represent part-whole hierarchies, we should use Composite, but if there is not such a need in the software we can use Builder instead of Composite in order to gain better software quality characteristics.

#### 3.3.2 Choosing between Factory Method and Prototype

Factory Method and Prototype are two creational patterns. Factory Method defines an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. Prototype specifies the kinds of objects to create using a prototypical instance, and creates new objects by copying this prototype. Prototype doesn't require sub classing Creator. However, they often require an Initialize operation on the Product class. Creator uses Initialize to initialize the object. Factory Method doesn't require such an operation [12]. The comparison between these two patterns is listed below (Figure 15):Factory Method has better results to gain expendability and simplicity than the Prototype, so Factory Method has better results to gain flexibility.

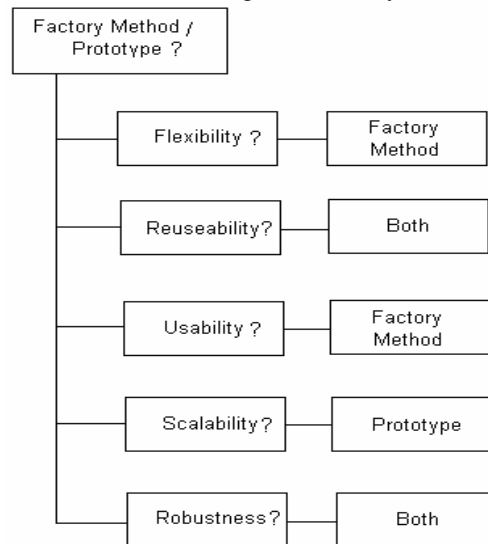


Figure 15 – Choosing between Factory Method and Prototype

- Factory Method can achieve better results in usability than the Prototype.
- Prototype can achieve better results to gain

scalability.

Now, we can come to conclusion that if we want to gain scalability we should use Prototype and if we want to emphasis on flexibility and usability we should use Factory Method, in other case we can use either of them.

### 3.3.3 Choosing between Prototype and Template Methods

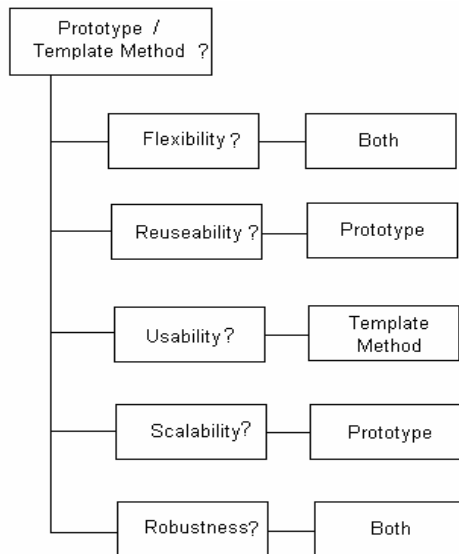


Figure 16 - Choosing between Prototype and Template Method

Prototype is a creational pattern and Template Methods is a behavioral pattern. Prototype specifies the kinds of objects to create using a prototypical instance, and creates new objects by copying this prototype. Template Method defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure [12]. We can compare the quality characteristics of these two patterns as below (Figure 16):

- Template Method and Prototype has the same results to gain flexibility.
- Prototype can achieve better results in modularity.
- Template Method can achieve better results in usability.
- Prototype can achieve better results in scalability.

Now, we can come to conclusion that if we want to gain scalability and reusability we should use Prototype and if we want to emphasis on usability we should use Template Method, in other case we can use either of them.

## 4 Conclusions

In this paper, we suggested to take benefits of design patterns to increase the Reusability, Flexibility, Usability, Scalability and Robustness. Then we focused on two relationships which lead us to decide between different design patterns. Our studies on “Similar” and “Conflict” relationships showed that patterns are not the same at

gaining the software quality metrics. Also there are some evidences that design patterns do not intrinsically promote every quality. For example none of the two patterns, State and Strategy, improve understandability.

The method we presented for “similar” and “conflict” patterns guided us to obtain a decision tree for applying the patterns in similar cases. This tree is base on the software quality characteristics that that are required for a software system.

In our future work we will try to study the other design patterns relationships than “similar” and “conflict”.

## 5 References

- [1] A.A. Aaby. Software: a fine art. Jan 2004. <http://cs.wvc.edu/aabyan/FAS/book>.
- [2] D.G. Firesmith. Common concepts underlying safety, security, and survivability engineering. December 2003. <http://www.sei.cmu.edu/pub/documents/03.reports/pdf/03tn033.pdf>.
- [3] International Standard. ISO/IEC 9126-1. Institute of Electrical and Electronics Engineers, 2001. <http://www.iso.ch>.
- [4] J. E. Gaffney. Metrics in software quality assurance. Proceedings of the ACM '81 conference, March 1981. <http://portal.acm.org>.
- [5] K. Khosravi, Y.G. Gueheneuc. A Quality Model for Design Patterns. Summer 2004
- [6] L. G. Williams, C. U. Smith. Introduction to Software Performance Engineering. Addison Wesley, Nov 2001. <http://www.awprofessional.com/articles/article.asp?p=24009>
- [7] L. J. Arthur. Software evolution, the software maintenance challenge. John Wiley and sons, 1951.
- [8] L. Tahvildari, K. Kontogiannis. On the Role of Design Patterns in Quality-Driven Re-engineering. Proceedings of the Sixth European Conference on Software Maintenance and Reengineering (CSMR02), 2002.
- [9] M. B. Nilles. A hard look at quality management software. Quality Digest, 2001. <http://www.dofactory.com/patterns/Patterns.aspx>.
- [10] O. Balci. Credibility assessment of simulation results. Proceedings of the 18th conference on winter simulation, 1986.
- [11] R. S. Pressman. Software Engineering a practitioner's Approach. McGraw-Hill, Inc, 1992.
- [12] R. Johnson, E. Gamma, R. Helm and J. Vlissides. Design Patterns Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.