An Empirical Study of Design Patterns and Software Quality

Foutse Khomh and Yann-Gaël Guéhéneuc

Ptidej Team

GEODES – Research Group on Open, Distributed Systems, Experimental Software Engineering

University of Montreal
Montreal, Quebec, Canada
guehene@iro.umontreal.ca

January 29, 2008

Abstract

We present an empirical study of the impact of design patterns on quality attributes in the context of software maintenance and evolution. Our first hypothesis verifies software engineering lore: design patterns impact software quality positively. We show that, contrary to popular beliefs, design patterns *in practice* impact negatively several quality attributes, thus providing concrete evidence against common lore. We then study design patterns and object-oriented best practices by formulating a second hypothesis on the impact of these principles on quality. We show that results for some design patterns cannot be explained and conclude on the need for further studies on the relation between design patterns and object-oriented best practices. Thus, we bring further evidence that design patterns should be used with caution during development because they may actually impede maintenance and evolution.

Contents

1 Introduction

Many studies in the literature (including some by these authors) have for premise that design patterns [4] improve the quality of object-oriented software systems. Indeed, it is claimed that design patterns improve the quality of systems and that every well-structured object oriented designs contain pattern, for example [4, page xiii] or [12].

Yet, some studies, *e.g.*, [13], suggest that the use of design patterns do not always result in "good" designs. For example, a tangled implementation of patterns impacts negatively the quality that these patterns claim to improve [9]. Also, patterns generally ease future enhancement at the expense of simplicity.

Thus, evidence of quality improvements through the use of design patterns consists primarily of intuitive statements and examples. There is little empirical evidence to support the claims of improved reusability[1], expandability and understandability as put forward in [4] when applying design patterns. Also, the impact of design patterns on other quality attributes is unclear.

The lack of concrete evidence on the impact of design patterns on quality led us to carry an empirical study of the impact of these patterns on the quality of systems as perceived by software engineers in the context of maintenance and evolution. Our hypothesis verifies software engineering lore: design patterns impact software quality positively. Our objective is to provide evidence to confirm or refute the hypothesis.

We perform the study by asking respondents their evaluations of the impact of design patterns on several quality attributes after application, thus in the context of maintenance and evolution. We choose this approach because, as explained in Section 3, there exists no quality model that takes into account the use of design patterns. Existing quality models fail to assess correctly systems in which design patterns are used [5]. Thus, the evaluation of the perceived impact of design patterns on quality is a necessary step to build a model that takes into account design patterns.

We present detailed results for three design patterns: Abstract Factory, Composite, Flyweight and three quality attributes: reusability, understandability, and expandability, and global results for other patterns and quality attributes. We conclude that, contrary to popular beliefs, design patterns *in practice* do not always improve quality attributes, thus providing concrete evidence against common lore. We attempt to explain the results with respect to object-oriented best practices and show their limitations. We conclude that design patterns and object-oriented best practices require further study and that patterns should be used with caution during development because they may actually impede maintenance and evolution.

The contribution of this paper is three fold. First, we propose an empirical method to study the impact of design patterns on quality attributes. Second, we carry out a qualitative and quantitative study and show that some design

[1] Although reusability in [4] may refer to the reusability of the solutions of the design patterns, we consider reusability as the reusability of the piece of code in which a pattern is implemented.

patterns impact quality negatively. Third, we show that principles of object-oriented programming fail to explain previous results.

We organise our paper as follows: Section 2 presents related work and their limitations. Section 3 states the hypothesis and objective of the study. Section 4 describes our quantification method. Section 5 presents the results of our survey. Section 6 contains a discussion on the results of our survey. Section 7 concludes our research and introduces future work.

2 Related Work

Since their introduction by Gamma et al. [4] in 1994, there has been a growing interest on the use of design patterns. In particular, work has been carried out to study the potential impact of patterns on software systems. Yet, few work investigated empirically their impact on quality. We present here main lines of work.

Lange and Nakamura demonstrated [7] that design patterns can serve as guide in program exploration and thus make the process of program understanding more efficient. Through a trail of pattern execution, they showed that if patterns were recognized at a certain point in the understanding process, they could help in "filling in the blanks" and in further exploring a system, improving thus the understandability of the system. However this study was limited to a single quality attribute and to a little number of patterns.

Wydaeghe et al. [14] presented a study on the concrete use of six design patterns when building an OMT editor. They discussed the impact of these patterns on reusability, modularity, flexibility, and understandability. They also discussed the difficulty of the concrete implementation of these patterns. They concluded that although design patterns offer several advantages, not all patterns have a positive impact on quality attributes. However, this study is limited to the authors' own experience and thus their evaluations of the impact of these patterns on quality can hardly been generalized to other contexts of development.

Wendorff [13] evaluated the use of design patterns in large commercial software systems. The author concluded that patterns do not necessarily improve a system design. Indeed, a design can be over-engineered [6] and the cost of removing design patterns is high. He did not perform a study on the impact of patterns on quality and provide only qualitative arguments.

McNatt and Bieman [9] examined the coupling between design patterns. They dressed a parallel between modularity and abstraction in software systems and modularity and abstraction in patterns. They showed that when patterns are loosely coupled and abstracted, maintainability, factorability, and reusability are well supported by the patterns. They concluded on the need for further studies to understand "good" pattern coupling methods. This study did not study the quantitative impact of patterns on quality.

Tahvildari et al. [11] studied the 23 design patterns from [4] and presented a layered classification of three primary relationships between these patterns:

use, refine, and conflict, and of three secondary relationships: similar, combine, and require, which can be expressed using the primary ones. They divided the patterns into two abstraction levels. They discussed how their classification can assist with understanding better the complex relationships among patterns, organising existing patterns as well as categorising and describing new patterns and building tool support for the application of patterns during restructuring. They did not investigate the impact of patterns on quality.

Bieman et al. [2, 3] examined common recommended programming styles on several different software systems, with and without patterns, and concluded that in contrast with common lore, the use of design patterns can lead to more change-prone classes rather than less change-prone classes during evolution.

Therefore, there is little evidence on the impact of design patterns on quality. Claims supporting the hypothesis are intuitive.

3 Objective and Hypothesis

The hypothesis of this study is that design patterns impact quality positively. Our objective is to qualify and quantify this impact on the overall quality of systems to confirm or refute the hypothesis.

There is a lack of a consensual framework to evaluate the quality of systems taking into account design patterns and other architectural characteristics. For example, the evaluation of a system with QMOOD [1] produces numerical values for some quality attributes that are difficult to interpret as there is no clear mapping between these numbers and object-oriented design principles.

Therefore, we chose an empirical study to collect, process, and analyse software engineers' evaluations of the impact of design patterns on quality attributes.

4 Collection and Processing

To collect evaluations of the impact of design patterns on quality attributes, we defined a questionnaire and carried out a survey electronically.

4.1 Definition of the Questionnaire

Following our previous work [5] and the work done in [1, 4], we chose the following set of quality attributes, based on their relevance to design patterns.

- Attributes related to design:
  - **Expandability:** The degree to which the design of a system can be extended.
  - **Simplicity:** The degree to which the design of a system can be understood easily.
  - **Reusability:** The degree to which a piece of design can be reused in another design.

- Attributes related to implementation:
  - **Learnability:** The degree to which the code source of a system is easy to learn.
  - **Understandability:** The degree to which the code source can be understood easily.
  - **Modularity:** The degree to which the implementation of the functions of a system are independent from one another.
- Attributes related to runtime:
  - **Generality:** The degree to which a system provides a wide range of functions at runtime.
  - **Modularity at runtime:** The degree to which the functions of a system are independent from one another at runtime.
  - **Scalability:** The degree to which the system can cope with large amount of data and computation at runtime.
  - **Robustness:** The degree to which a system continues to function properly under abnormal conditions or circumstances.

Each quality attribute was evaluated using a six-point Likert scale:

```
A - Very positive
B - Positive
C - Not significant
D - Negative
E - Very Negative
F - Not applicable
```

The sixth value allowed respondents to not answer a question if they did not know or are not sure about the impact of a design pattern on a quality attribute.

For every design pattern in [4] and for every quality attribute from our set, the respondents were asked to assess the impact of the patterns on the quality of a hypothetical system in which the pattern would be used appropriately, to assess the impact of this pattern on quality as they would during a technical review [10] or possibly while performing a program comprehension-related activity during maintenance and evolution.

The questionnaire is available on the Internet at `http://www.ptidej.net/downloads/`.

4.2 Data Collection

The questionnaire was sent out in January 2007 to experienced software engineers and posted on three mailing lists, *refactoring*, *patterns-discussion*, and *gang-of-4-patterns*. We collected respondents' evaluations during the period of January to April 2007.

Among the many answers that we received, we selected the questionnaires of 20 software engineers with a verifiable long experience in the use of design patterns in software development and maintenance.

Although among these selected 20 questionnaires, some respondents did not evaluate the quality of all design patterns, we argue that the sample is representative enough. Moreover, the number of collected evaluations is the largest of all studied previous work.

4.3 Data Processing

Due to the variations between answers, we felt that the differences between `Positive` and `Very Positive` answers were due to the fact that some respondents were less strict than others and thus, that their `Very Positive` evaluations were not directly relevant to the impact of the patterns. This fact has been confirmed in discussions with the respondents. For example, for Builder and expandability, we had 19% of respondents considering the pattern `Very Positive` while 63% considered it `Positive` and 18% considered it `Neutral`. Therefore, we chose to aggregate answers `A` and `B` and answers `D` and `E`:

```
Positive = A and B
Neutral  = C
Negative = D and E
```

Using the previous three-point Likert scale, we computed the frequencies of the answers on each quality attribute: `Positive`, `Neutral`, and `Negative` and we carried out a Null hypothesis test to assess the perceived impact of the patterns on the quality attributes.

Answers `F` were not considered because they represented situations where the respondents did not know or did not want to evaluate the impact.

5 Analyses

We now present the detailed results of a qualitative and a quantitative analyses for three design patterns: Abstract Factory, Composite, and Flyweight, and the three quality attributes mentioned by the GoF [4, page xiii]: reusability, expandability, and understandability. Complete results then follows.

We choose the following three design patterns to illustrate our respondents' assessments first because of their popularity—they are among the most commonly used patterns and thus we felt that their evaluation would be more accurate—and second because they appear to be considered as globally positive, globally neutral, and globally negative.

5.1 Qualitative Analysis

We study qualitatively the results, without making comparison between the numbers of answers, which is the subject of the quantitative analysis in Section 5.4.

Attributes	Positive	Neutral	Negative
Expandability	100.00%	0.00%	0.00%
Simplicity	69.23%	15.38%	15.38%
Reusability	61.54%	23.08%	15.38%
Learnability	76.92%	7.69%	15.38%
Understandability	69.23%	15.38%	15.38%
Modularity	71.43%	21.43%	7.14%
Generality	76.92%	15.38%	7.69%
Mod. at Runtime	53.85%	38.46%	7.69%
Scalability	41.67%	41.67%	16.67%
Robustness	8.33%	91.67%	0.00%

Table 1: Impact of Composite (in percentage of the number of respondents.)

Attributes	Positive	Neutral	Negative
Expandability	100.00%	0.00%	0.00%
Simplicity	53.33%	13.33%	33.33%
Reusability	50.00%	42.86%	7.14%
Learnability	35.71%	28.57%	35.71%
Understandability	38.46%	30.77%	30.77%
Modularity	85.71%	7.14%	7.14%
Generality	78.57%	21.43%	0.00%
Mod. at Runtime	46.15%	38.46%	15.38%
Scalability	21.43%	64.29%	14.29%
Robustness	0.00%	72.73%	27.27%

Table 2: Impact of Abstract Factory.

5.2 Design Patterns

Composite. Table 1 presents the respondents' evaluations of the impact of the Composite pattern on the quality attributes. It appears that the Composite pattern is mostly perceived as having a positive impact on the quality of systems. All quality attributes are impacted positively but for scalability and robustness, which are consider neutral. Given the purpose of the Composite pattern, having a neutral impact on scalability is rather surprising.

Abstract Factory. Table 2 presents the respondents' evaluations of the impact of the Abstract Factory pattern on the quality attributes. It shows that half the quality attributes is considered as positively impacted while the other half is not. It is not surprising that the pattern is overall judged as neutral given its purpose and complexity. It is striking that learnability and understandability are felt negatively impacted.

Flyweight. Table 3 presents the respondents' evaluations of the impact of the Flyweight pattern on the quality attributes. It reports that this pattern is perceived as impacting negatively all quality attributes but scalability. Given the purpose of the pattern, it is not surprising that its impact on scalability is judged positively. The negative perception could be explained by the less frequent use of Flyweight in comparison with Composite and Abstract Factory.

Attributes	Positive	Neutral	Negative
Expandability	22.22%	44.44%	33.33%
Simplicity	0.00%	22.22%	77.78%
Reusability	37.50%	12.50%	50.00%
Learnability	0.00%	20.00%	80.00%
Understandability	0.00%	10.00%	90.00%
Modularity	33.33%	33.33%	33.33%
Generality	11.11%	44.44%	44.44%
Mod. at Runtime	11.11%	66.67%	22.22%
Scalability	77.78%	0.00%	22.22%
Robustness	22.22%	66.67%	11.11%

Table 3: Impact of Flyweight.

5.3 Quality Attributes

We choose the following three quality attributes because it is claimed in [4, 12] that they are improved by the use of design patterns.

Expandability. Table 4 presents the respondents' evaluations of the impact of the design patterns on expendability. All respondents felt that expandability is improved when using patterns, in conformance with the claims made in [4].

Reusability. Table 5 presents the respondents' evaluations of the impact of design patterns on reusability. Reusability is felt as being slightly more negatively impacted by design patterns, with 13 neutral or negative patterns and 10 positive patterns. This is rather surprising as the use of patterns is claimed to improve reusability.

Understandability. Table 6 presents the respondents' evaluations of the impact of design patterns on understandability. Similarly to reusability, respondents felt that understandability was rather slightly negatively impacted by the use of patterns.

5.4 Quantitative Analysis

Using the results presented in Tables 1, 2, 3, 5, 4 and 6, we carry out Null hypothesis tests to quantify the impact of the design patterns on the quality attributes and then confirm or refute the hypothesis that *design patterns impact software quality positively*. We use the frequencies of `Positive` and non-positive answers (combining `Neutral` and `Negative` answers) to decide on the impact of a pattern on a quality attribute.

For a given question from our questionnaire, we consider the random variable X, that takes the value 0 when the impact of the pattern on the attribute is `Positive` and 1 when the impact is not positive. We defined P as the probability that the pattern does not impact positively the attribute. The probability that the pattern impacts positively the attribute is therefore $1 - P$. Considering the

Patterns	Positive	Neutral	Negative
A.Factory	100.00%	0.00%	0.00%
Builder	90.91%	9.09%	0.00%
F.Method	72.73%	9.09%	18.18%
Prototype	63.64%	27.27%	9.09%
Singleton	9.09%	27.27%	63.64%
Adapter	50.00%	41.67%	8.33%
Bridge	83.33%	16.67%	0.00%
Composite	100.00%	0.00%	0.00%
Decorator	90.91%	0.00%	9.09%
Facade	58.33%	16.67%	25.00%
Flyweight	22.22%	44.44%	33.33%
Proxy	45.45%	45.45%	9.09%
Ch.Of.Resp	91.67%	8.33%	0.00%
Command	66.67%	16.67%	16.67%
Interpreter	63.64%	27.27%	9.09%
Iterator	90.91%	9.09%	0.00%
Mediator	58.33%	25.00%	16.67%
Memento	33.33%	55.56%	11.11%
Observer	85.71%	7.14%	7.14%
State	72.73%	18.18%	9.09%
Strategy	76.92%	15.38%	7.69%
T.Method	84.62%	15.38%	0.00%
Visitor	71.43%	7.14%	21.43%

Table 4: Impact on expandability.

N respondents $j = 1, ..., N$ answering the question, we view their answers as occurrences of the random variable X and note them: $X_1, X_2, ...X_N$. Then, we set our Null hypothesis to be $H_0 : P \leq \frac{1}{2}$, which means that the impact of the pattern on the quality attribute is positive. The alternative hypothesis is then $H_1 : P > \frac{1}{2}$, which means that the pattern does not impact positively the attribute. Our decision rule is:

- We confirm H_0 if f_N is not high enough;
- We confirm H_1 if f_N is high enough;

where f_N is the frequency of the respondents who answered that the pattern impacts negatively or does not impact the attribute. By "high enough", we refer to a rate level that directly impacts the risk of making the decision at that level. For example if high enough is $\geq 80\%$, the risk encountered by deciding at that level is 0.37, while if high enough is $\geq 60\%$ the risk encountered by deciding at that level is 15.09. These values are computed using the Bernoulli distribution.

The risk that we encounter by rejecting the Null hypothesis H_0, *i.e.*, the pattern positively impacts the quality attribute, is then: $1 - F(f_N)$, where F is the cumulative density of the Bernoulli distribution $\beta(N, \frac{1}{2})$.

The Null hypothesis test yields the results summarized in Tables 7, 8, 9 and 10 for all design patterns and quality attributes. In these tables, the sign $+$ means that, with our Null hypothesis test, the impact of the pattern on the quality attribute is positive else the sign is $-$ (it can be negative or neutral). The number next to a sign represents the risk of making this decision.

Patterns	Positive	Neutral	Negative
A.Factory	46.15%	46.15%	7.69%
Builder	36.36%	45.45%	18.18%
F.Method	60.00%	20.00%	20.00%
Prototype	63.64%	0.00%	36.36%
Singleton	18.18%	54.55%	27.27%
Adapter	66.67%	25.0%	8.33%
Bridge	41.67%	16.67%	41.67%
Composite	58.33%	25.00%	16.67%
Decorator	36.36%	18.18%	45.45%
Facade	36.36%	45.45%	18.18%
Flyweight	37.5%	12.5%	50.00%
Proxy	45.45%	36.36%	18.18%
Ch.Of.Resp	54.55%	27.27%	18.18%
Command	30.00%	20.00%	50.00%
Interpreter	50.00%	0.00%	50.00%
Iterator	72.73%	9.09%	18.18%
Mediator	20.00%	50.00%	30.00%
Memento	28.57%	42.86%	28.57%
Observer	53.85%	23.08%	23.08%
State	20.00%	40.00%	40.00%
Strategy	41.67%	33.33%	25.00%
T.Method	58.33%	33.33%	8.33%
Visitor	28.57%	28.57%	42.86%

Table 5: Impact on reusability.

5.5 Complete Results

Tables 9 and 10 present the complete results of the impact of the 23 patterns on the quality attributes.

6 Impact of patterns on quality and object-oriented Programming

The analysis of the results of our study reveal that, in contrary to common lore, design patterns do not always impact quality attributes positively. Our respondents consider that, although patterns are useful to solve design problems, they do not always improve the quality of the systems in which they are applied. In particular, a large number of respondents considered that they sensibly decrease simplicity, learnability, and understandability. Some patterns, like Flyweight, are considered as impacting most attributes negatively.

We now attempt to explain these results by studying design patterns from the point of view of object-oriented software practices. We focus on some "famous" patterns as we consider their evaluations by the respondents more accurate. By "famous", we mean that all selected respondents fill the entire evaluations of these patterns.

We make the hypothesis that the principles help in improving the quality and thus should explain the results found on the impact of design patterns on

Patterns	Positive	Neutral	Negative
A.Factory	38.46%	30.77%	30.77%
Builder	81.82%	9.09%	9.09%
F.Method	45.45%	27.27%	27.27%
Prototype	58.33%	16.67%	25.00%
Singleton	91.67%	8.33%	0.00%
Adapter	50.00%	25.00%	25.00%
Bridge	50.00%	33.33%	16.67%
Composite	75.00%	16.67%	8.33%
Decorator	45.45%	9.09%	45.45%
Facade	81.82%	18.18%	0.00%
Flyweight	0.00%	10.00%	90.00%
Proxy	33.33%	50.00%	16.67%
Ch.Of.Resp	33.33%	33.33%	33.33%
Command	33.33%	33.33%	33.33%
Interpreter	63.64%	0.00%	36.36%
Iterator	50.00%	41.67%	8.33%
Mediator	58.33%	25.00%	16.67%
Memento	33.33%	55.56%	11.11%
Observer	42.86%	35.71%	21.43%
State	54.55%	0.00%	45.45%
Strategy	69.23%	23.08%	7.69%
T.Method	38.46%	38.46%	23.08%
Visitor	21.43%	21.43%	57.14%

Table 6: Impact on understandability.

quality.

7 Principles of Object Oriented Programming

Since the inception of object-oriented programming, several work provided guidelines and principles. We recall here some principles, summarised from [8]. We choose these principles among the many available ones because these are well-known, hard-won products of decades of experience in software engineering.

- **Open Close Principle:** Software entities like classes, packages, and methods should be open to extension but closed to modifications. The Open Close Principle encourages software engineers to design and write code so that adding new functionalities involve minimal changes. Most changes are handled as new methods and new classes. Designs following this principle are more resilient and do not break when adding new functionalities.
- **Liskov Substitution Principle:** An instance of a derived class must be able to take the place of an instance of the base class. For example, if a method has an object of a class as an argument, the same method must be able to work with an instance of a derived class.
- **Dependency Inversion Principle:** High-level modules should not depend upon low-level modules, both should depend upon abstractions. Abstractions should not depend upon details. Details should depend upon

Attributes	Composite		A.Factory		Flyweight	
	E	R(%)	E	R(%)	E	R(%)
Expendability	+	0.00	+	0.00	−	1.76
Simplicity	+	5.92	+	30.36	−	0.00
Reusability	+	15.09	+	50.00	−	15.09
Learnability	+	1.76	−	15.09	−	0.00
Understandability	+	5.92	−	15.09	−	0.00
Modularity	+	5.92	+	0.37	−	5.92
Generality	+	1.76	+	1.76	−	0.15
Mod. at Runtime	+	30.36	−	30.36	−	0.15
Scalability	−	30.36	−	1.76	+	1.76
Robustness	−	0.15	−	0.00	−	1.76
	8 + / 2 −		5 + / 5 −		1 + / 9 −	

Table 7: Estimation of the impact of the three design patterns on quality attributes.

abstractions. This principle "inverts" the conventional idea that high level modules should depend upon the lower level ones.

- **Interface Segregation Principle:** Clients should not be forced to depend upon interfaces that they do not use. Many client-specific interfaces are better than a general-purpose one.
- **Reuse/Release Equivalency Principle:** The granule of reuse is the same as the granule of release. Only components that are released through a tracking system can be effectively reused.
- **Common Reuse Principle:** Classes that are not reused together should not be grouped together.
- **Common Closure Principle:** Classes that change together, belong together.
- **Stable Abstractions Principle:** The more stable a category of classes is, the more it should consist of abstract classes. A completely stable category should consist of only abstract classes.
- **Least Astonishment Principle:** When two elements of an interface conflict or are ambiguous, the behavior should be that which least surprises the software engineer at the time of the conflict, because the least surprising behavior must be usually the correct one.
- **Deep Abstract Hierarchies Principle:** Class hierarchies should be deep and abstract.
- **The Acyclic Dependencies Principle:** There should be no cycles in the dependency graph.
- **The Stable Dependencies Principle:** Depend in the direction of stability. The dependencies between components in a design should be in the direction of stability. A component should only depend upon components that are more stable than it is.
- **Demeter Principle:** Each unit (class, method) should only use a limited set of other units: only units "closely" related to the current unit.

Design Patterns	Expendability(%)		Understandability(%)		Reusability(%)	
	E	R(%)	E	R(%)	E	R(%)
A.Factory	+	0.00	−	15.09	+	50.00
Builder	+	0.15	+	0.37	−	15.09
F.Method	+	1.76	−	30.36	+	15.09
Prototype	+	30.36	+	30.36	+	30.36
Singleton	−	0.15	+	0.15	−	0.37
Adapter	+	30.36	−	30.36	+	5.92
Bridge	+	0.37	+	50.00	−	30.36
Composite	+	0.00	+	5.92	+	15.09
Decorator	+	0.15	−	30.36	−	5.92
Facade	+	30.36	+	1.76	−	5.92
Flyweight	−	1.76	−	0.00	−	15.09
Proxy	−	30.36	−	5.92	+	50.00
Ch.Of.Resp	+	0.15	−	5.92	+	30.36
Command	+	5.92	−	5.92	−	5.92
Interpreter	+	5.92	+	5.92	+	30.36
Iterator	+	0.15	+	50.00	+	5.92
Mediator	+	30.36	+	30.36	−	1.76
Memento	−	5.92	−	30.36	−	15.09
Observer	+	0.15	−	30.36	+	50.00
State	+	5.92	+	30.36	−	1.76
Strategy	+	1.76	+	15.09	−	30.36
T.Method	+	0.37	−	15.09	+	30.36
Visitor	+	5.92	−	1.76	−	1.76
	19 + / 4 −		11 + / 12 −		11 + / 12 −	

Table 8: Estimation of the impact of design patterns on the three quality attributes

8 Design Patterns and Principles

We now study the evaluations of the three design patterns, shown in Tables 9 and 10, with respect to object-oriented principles.

Composite. The Composite pattern allows an instance of a class to be treated in the same way as a group of objects. It makes it easy to add new kinds of objects. It makes clients simpler, because they do not have to know if they are dealing with a leaf or a composite object. Thus its use in a system impacts positively the expandability and the simplicity, which is in accordance with the evaluations of our respondents. However, the Composite pattern makes it harder to restrict the type of objects in a composite and may lead to large amount of objects being instantiated and referenced, thus possibly explaining the neutral evaluations of its impact on robustness and scalability.

Abstract Factory. The intent of the Abstract Factory pattern is to separate the creation of objects from their uses. It allows for new derived types to be introduced with no change to the code that uses the base objects. This pattern thus respects the Open Close Principle and improves the expandability, the simplicity, and the generality of systems. Also, it makes it possible to

Design Patterns	Expendability(%)		Simplicity(%)		Generality(%)		Modularity(%)		Mod. at runtime(%)	
	E	R(%)	E	R(%)	E	R(%)	E	R(%)	E	R(%)
A.Factory	+	0.00	+	30.36	+	1.76	+	0.37	−	30.36
Builder	+	0.15	+	30.36	+	30.36	+	0.15	−	30.36
F.Method	+	1.76	+	30.36	+	30.36	+	5.92	−	30.36
Prototype	+	30.36	+	30.36	+	1.76	−	15.09	+	30.36
Singleton	−	0.15	+	0.15	−	5.92	−	0.37	−	0.37
Adapter	+	30.36	−	30.36	+	15.09	+	1.76	+	30.36
Bridge	+	0.37	−	0.37	+	5.92	+	1.76	+	50.00
Composite	+	0.00	+	5.92	+	1.76	+	5.92	+	30.36
Decorator	+	0.15	−	5.92	+	0.15	+	5.92	+	5.92
Facade	+	30.36	+	0.37	+	50.00	+	50.00	−	15.09
Flyweight	−	1.76	−	0.00	−	0.15	−	5.92	−	0.15
Proxy	−	30.36	+	15.09	+	15.09	+	1.76	−	15.09
Ch.Of.Resp	+	0.15	+	5.92	+	0.37	+	5.92	+	30.36
Command	+	5.92	−	30.36	+	15.09	+	15.09	−	30.36
Interpreter	+	5.92	+	50.00	+	15.09	+	30.36	−	30.36
Iterator	+	0.15	+	5.92	+	5.92	+	30.36	+	50.00
Mediator	+	30.36	−	5.92	−	30.36	+	50.00	−	5.92
Memento	−	5.92	+	50.00	−	30.36	−	0.15	−	0.15
Observer	+	0.15	+	15.09	+	1.76	+	1.76	+	5.92
State	+	5.92	+	15.09	+	15.09	+	15.09	+	30.36
Strategy	+	1.76	+	5.92	+	5.92	+	0.37	+	1.76
T.Method	+	0.37	+	5.92	+	1.76	−	5.92	−	5.92
Visitor	+	5.92	−	0.15	+	1.76	+	1.76	+	30.36
	19 + / 4 −		16 + / 7 −		19 + / 4−		18 + / 5 −		11 + / 12 −	

Table 9: Estimation of the impact of design patterns on quality attributes (Part 1).

interchange concrete classes without changing the code that uses them, even at runtime, thus improving modularity and reusability, in accordance with the our respondents' evaluations.

However, due to the flexibility of interchanging concrete classes at runtime, the pattern should improve the modularity at runtime and the scalability of systems in which it is used, a position which contradicts our respondents' results. We believe that this unexpected results can be explain by the difficulty of writing optimal implementations of this pattern. The use of this pattern, as with similar design patterns, induces the risk of unnecessary complexity and extra work in the initial design and implementation, thus decreasing understandability and learnability.

Flyweight. The Flyweight pattern is considered by our respondents as impacting negatively most quality attributes. The Flyweight pattern is tied to a very specific problem and thus is not expandable. Yet, it allows thousands of objects to work together improving thus scalability. It is not simple and not generalizable, it decreases learnability, understandability, and reusability, as software engineers must know the specific solved problem to be able to understand the implementation. This pattern violates the Open Close Principle as engineers cannot extend the piece of code in which it is used without almost

Design Patterns	Learnability(%)		Understandability(%)		Reusability(%)		Scalability(%)		Robustness(%)	
	E	R(%)	E	R(%)	E	R(%)	E	R(%)	E	R(%)
A.Factory	−	15.09	−	15.09	+	50.00	−	1.76	−	0.00
Builder	+	30.36	+	0.37	−	15.09	−	0.00	−	0.00
F.Method	+	50.00	−	30.36	+	15.09	−	5.92	−	0.00
Prototype	−	30.36	+	30.36	+	30.36	−	5.92	−	0.15
Singleton	+	0.37	+	0.15	−	0.37	−	15.09	−	0.37
Adapter	+	5.92	−	30.36	+	5.92	−	0.00	−	0.00
Bridge	−	5.92	+	50.00	−	30.36	−	0.15	−	0.15
Composite	+	1.76	+	5.92	+	15.09	−	30.36	−	0.15
Decorator	−	30.36	−	30.36	−	5.92	−	0.37	−	0.00
Facade	+	5.92	+	1.76	−	5.92	−	1.76	−	1.76
Flyweight	−	0.00	−	0.00	−	15.09	+	1.76	−	1.76
Proxy	−	30.36	−	5.92	+	50.00	−	5.92	−	0.15
Ch.Of.Resp	+	15.09	−	5.92	+	30.36	−	0.15	−	1.76
Command	−	15.09	−	5.92	−	5.92	−	1.76	−	5.92
Interpreter	+	5.92	+	5.92	+	30.36	−	5.92	−	0.15
Iterator	+	50.00	+	50.00	+	5.92	−	0.37	−	30.36
Mediator	+	30.36	+	30.36	−	1.76	−	1.76	−	0.15
Memento	−	30.36	−	30.36	−	15.09	−	0.00	−	0.15
Observer	+	50.00	−	30.36	+	50.00	−	0.37	−	0.15
State	+	30.36	+	30.36	−	1.76	−	0.37	−	0.15
Strategy	+	1.76	+	15.09	−	30.36	−	1.76	−	5.92
T.Method	+	30.36	−	15.09	+	30.36	−	1.76	−	0.37
Visitor	−	0.37	−	1.76	−	1.76	−	1.76	−	0.00
	14 + / 9 −		11 + / 12 −		11 + / 12−		1 + / 22 −		0 + / 23 −	

Table 10: Estimation of the impact of design patterns on quality attributes (Part 2).

rewriting it all.

Other Patterns. The Observer pattern does not make a system more modular but improves the modularity of other patterns [14]. It enables, for example, the separation between model, view, and controller in the MVC pattern. The use of this pattern makes a system more flexible: by removing dependencies between modules, the Observer pattern makes it possible to replace these modules seamlessly. It is also easy to extend functionalities by adding one or more entities that cooperates with existing ones. This pattern respects the Open Close Principle. Thus, we could expect that it impacts positively reusability, expandability, and simplicity, which is the case according to our respondents' evaluations, but negatively learnability and understandability because the complexity of the structure and the implicit coupling make it difficult for a maintainer to identify at first the relationships among classes. The Observer pattern weakens the encapsulation and decreases robustness because the implicit coupling makes the correction of bugs difficult.

Visitor promotes "good" programming because it encapsulates operations in classes instead of having this operation spread over a number of classes, thus improving modularity. It also improves reusability because one can choose to drop unnecessary visitors when reusing the implementation and because writ-

ing new visitors is eased by reusing common functions of existing visitors. The implementation of this pattern respect the Common Reuse and the Common Closure Principles. However, the internal working of the pattern may be hard to understand thus impacting understandability negatively. The pattern implies many indirections and thus weakens encapsulation. The cost of the many indirections may become too much for the cache and thus decrease scalability.

Chain of Responsibility does not make a system more modular as it adds dependencies between previously separated objects. Spreading functionality through the code does not make the code more readable thus decreasing learnability. Understandability is also decreased as it is very dynamic. Our survey showed that in practice this pattern is not consider to be simple and, in case of bugs, it is sometimes hard to fix so it decreases the robustness of systems. These weaknesses are compensated by a gain in flexibility because with a chain of responsibility, it is easy to insert a new object to handle new events.

Singleton is considered as impacting negatively the expandability of systems because it is a sort of euphemism for a global variable, which weakens the encapsulation, and it is reported to produce high coupling, thus violating the Dependency Inversion principle. It is reported robust due to its simplicity.

9 Discussion

From this study, we remark that most design patterns respect the principles of object-oriented programming. Hence, according to our hypothesis that object-oriented best practices help producing systems with good quality, it is surprising that their use seems to decrease quality. A possible explanation could be that, for a pattern, many implementations are possible and that the concrete implementations may not be conform to the principles of object-oriented programming. Also, it may be that these best practices are necessary but not sufficient to build systems with good quality. In addition, we notice that, for the studied design patterns, several principles do not seem to apply or explain the results of the study, thus calling for further studies on the impact of these principles on quality.

10 Threats to Validity

This empirical study has the advantage to focus on concrete implementations of design patterns and thus produce accurate evaluations of the impact of the patterns on quality as perceived by software engineers. However, it has some limitations.

There is a difficulty in collecting a large number of data. In this study, we focused on the answers of 20 software engineers with a long experience in the use of patterns. Choosing experienced engineers introduces a bias against novice engineers.

The subjectivity of a respondent's evaluation may affect the results: some respondents are more strict than others, even though we tried to lessen this risk.

Among the 23 patterns from the GOF, some are not frequently used in systems. Thus, the negative evaluations may be just an *a priori* on the pattern because our respondents considered the pattern to be not suitable and then their evaluation may not reflect the real impact of the implementation of the pattern on quality.

There is no threat to the validity of the conclusion of this study as there is a direct relationship between the design of a system and its quality. The design of a system directly impacts the quality attributes presented in Section 4.1 thus we can say that there is no threat to the construct and internal validities of our study. However, the results of our study may not be fully generalisable to any software engineers, patterns, and systems. Future work will enrich the number of answers and provide more generalisable results.

11 Conclusion

With this study, we show that design patterns do not always improve the quality of systems. Some patterns are reported to decrease some quality attributes and to not necessarily promote reusability, expandability, and understandability. Therefore, we bring further evidence that design patterns should be used with caution during development because they may actually impede maintenance and evolution. This study also reveals that object-oriented principles may not be so "good" as they may not necessarily result in systems with good quality. Thus, there is a need for studies to assess the impact of these principles on the quality of systems.

This study is the largest to date in term of the number of collected evaluations on design patterns and quality of systems. However, we plan to continue collecting evaluations to improve the accuracy of our results and to generalise our conclusions to different software context. The questionnaire is available on the Internet at `http://www.ptidej.net/downloads/` (it may take some minutes to load as it weighs 4 MB). We are looking forward receiving more evaluations.

Acknowledgments

We are grateful to Kim Mens for the fruitful discussions. We would like to thank all the respondents for their evaluations and comments. This work has been partially funded by NSERC and the VINCI program of University of Montreal.

References

[1] Jagdish Bansiya and Carl G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28:4–17, January 2002.

[2] James Bieman, Greg Straw, Huxia Wang, P. Willard Munger, and Roger T. Alexander. Design patterns and change proneness: An examination of five evolving systems. In Michael Berry and Warren Harrison, editors, *Proceedings of the* 9^{th} *international Software Metrics Symposium*, pages 40–49. IEEE Computer Society Press, September 2003.

[3] James M. Bieman, Roger Alexander, P. Willard Munger III, and Erin Meunier. Software design quality: Style and substance. In *Proceedings of the* 4^{th} *Workshop on Software Quality*. ACM Press, March 2001.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1^{st} edition, 1994.

[5] Yann-Gaël Guéhéneuc, Jean-Yves Guyomarc'h, Khashayar Khosravi, and Houari Sahraoui. Design patterns as laws of quality. University of Montreal, 2005.

[6] J. Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 1st edition, August 2004.

[7] Danny B. Lange and Yuichi Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Proceedings of the* 10^{th} *annual conference on Object-oriented programming systems, languages, and applications*, pages 342 – 357. ACM Press, 1995.

[8] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. 2002.

[9] William B. McNatt and James M. Bieman. Coupling of design patterns: Common practices and their benefits. In T.H. Tse, editor, *Proceedings of the* 25^{th} *Computer Software and Applications Conference*, pages 574–579. IEEE Computer Society Press, October 2001.

[10] Roger S. Pressman. *Software Engineering – A Practitioner's Approach*. McGraw-Hill Higher Education, 5^{th} edition, November 2001.

[11] Ladan Tahvildari and Kostas Kontogiannis. On the role of design patterns in quality-driven re-engineering. In Tibor Gyimothy and Fernando Brito e Abreu, editors, *Proceedings of the*6^{th} *European Conference on Software Maintenance and Reengineering*, pages 230–240. IEEE Computer Society, March 2002.

[12] Bill Venners. How to use design patterns – A conversation with Erich Gamma, part I, May 2005. `http://www.artima.com/lejava/articles/gammadp.html`.

[13] Peter Wendorff. Assessment of design patterns during software reengineering: Lessons learned from a large commercial project. In Pedro Sousa and Jürgen Ebert, editors, *Proceedings of* 5^{th} *Conference on Software Maintenance and Reengineering*, pages 77–84. IEEE Computer Society Press, March 2001.

[14] B. Wydaeghe, K. Verschaeve, B. Michiels, B. Van Damme, E. Arckens, and V. Jonckers. Building an OMT-editor using design patterns: An experience report. 1998.