Empirical Studies on the Impact of Design Patterns on Quality

Yann-Gaël Guéhéneuc

Professor

Canada Research Chair on Software Patterns and Patterns of Software

Leader of the Ptidej Team

Département de génie informatique et génie logiciel
École Polytechnique de Montréal
C.P. 6079, succ. Centre-Ville
Montreal, Quebec, Canada
H3C 3A7
1-514-340-4711 #7116
yann-gael.gueheneuc@polymtl.ca

February 12, 2010

Abstract

Design patterns are a form of documentation that proposes solutions to recurring object-oriented software design problems. Design patterns became popular in software engineering thanks to the book published in 1995 by the Gand of Four: Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Since the publication of the book "Design Patterns: Elements of Reusable Object-Oriented Software", design patterns have been used to design programs and ease their maintenance, to teach object-oriented concepts and related "good" practices in classrooms, to assess quality and help program comprehension in research. However, design patterns may also lead to over-engineered programs and may negatively impact quality. We recall the history of design patterns and present some recent development characterising the advantages and disadvantages of design patterns.

Keywords: Patterns, languages of patterns, design patterns, anti-patterns, code smells, software quality, implementation and identification, empirical studies.

Contents

1 A Brief History of Design Patterns **3**

2 Design Patterns and Quality **4**

2.1 Quality Without A Name 5

2.2 Languages of Patterns 7

2.3 Patterns Generate Architecture 7

2.4 Over-engineering and Refactorings 7

3 Research Work on Design Patterns **8**

3.1 Design Pattern Impact on Quality Characteristics 8

3.2 Design Pattern Impact on Program Comprehension 13

4 Conclusion **24**

1 A Brief History of Design Patterns

Many kinds of patterns exist to be used during the different phases of development and maintenance, *e.g.*, analysis [Fow96], architecture [BMR+96], design [GHJV94], implementation [Cop91]. The Pattern Languages of Programs conference series (PLoP) has received submissions of and published patterns encompassing all the phases of development and maintenance, including: development organisation, process, project planning, requirements engineering, configuration management, and so on, in addition to development and maintenance.

Patterns can be categorised along four axes:

1. Levels of abstraction, typically:
    - Code [BMR+96];
    - Design [GHJV94];
    - Architecture [BMR+96].
2. Artifacts on which their motifs apply, including:
    - Source code;
    - UML-like diagrams;
    - Execution traces;
    - Change sets.
3. Types of their motifs:
    - Positive, such as those proposed by design patterns;
    - Negative, such as those of code smells [Fow99] and antipatterns [BMB+98], which describe "poor" solutions to recurring design problems.
4. Application domains: some patterns, such as the ones in [GHJV94], are deemed general while others, such as the J2EE patterns [AMC03], are specific to one domain.

The most popular kind of patterns remains design patterns, introduced in 1994 [GHJV94] to gather "good" solutions to recurring object-oriented software design problems. The solutions proposed by design patterns are thought to increase, among other quality characteristics, expandability, reusability, and understandability. Design patterns already existed prior to the publication of the book "Design Patterns: Elements of Reusable Object-Oriented Software".

In 1987, Kent Beck and Ward Cunningham introduced the idea of patterns at the OOPSLA conference. In the following years, Beck, Cunningham and others kept working on this work. Beck and Cunningham got the idea of software defining patterns when they came across Christopher Alexander's work on architectural patterns. Alexander was a building architect who originated the concept of patterns in the 1960s. Alexander in particular wrote

the ground-breaking book “The Timeless Way of Building”, in which he explained the idea of patterns in architecture: “a pattern solves a specific problem by bringing two conflicting forces into balance”. The vocabulary used in software design patterns mostly stems from Alexander’s work.

Since the publication of the book “Design Patterns: Elements of Reusable Object-Oriented Software”, design patterns have been used to design programs and ease their maintenance. One of the first program designed using design patterns was JHotDraw, originally written by Erich Gamma and Thomas Eggenschwiler in the early 2000s. JHotDraw is a framework defining a GUI-based editor with tools in a tool palette, different views, user-defined graphical figures, and support for saving, loading, and printing drawings. The framework can be customized using inheritance and design patterns. Several programs have been built using the framework, in particular JavaDraw, a 2-D vector-based graphic editor, with some animation capabilities. JHotDraw includes 20 different patterns in its 5.1 version, which make it flexible, extensible, and reusable in many contexts. Although JHotDraw was conceived as an exercise, it is now, thanks to design patterns, a successfully open-source project, in its 7.2 version, with 24 active developers, and more than 52,000 downloads since October 2000.

Design patterns are also taught in classrooms to introduce object-oriented concepts and related “good” practices. Most computer science and engineering curricula now include “design and architecture” courses, in which design patterns are taught to students to illustrate “good” design practices and to acquaint them with solutions to recurring design problems. It is important for current and future developers to know design patterns because they form a vocabulary that eases communication: rather than explaining to a colleague that “this object can be a simple object or a complex composition of objects”, a developers would say that “this object is a Composite”, thus simplifying her explanations and conveying the solutions of her problem as well as her intent and the consequences of her design choice.

Finally, design patterns are the subject of much past and current research relating to software quality and program comprehension. Since their inception in 1995, design patterns have been said to improve software quality, in particular expandability, reusability, and understandability. Therefore, it is thought that using design patterns up-front in a design does improve its quality. Yet, design patterns may also lead to over-engineering a design, which impedes its development and maintenance. Design patterns, however, ease the communication among developers and can help in top-down and–or bottom-up comprehension approach by acting as “chunks” that speed the recognition process.

The following sections present concepts related to and studies on design patterns, which shed light on their advantages and limitations.

2 Design Patterns and Quality

A pattern, to be truly a pattern, must describe a solution to a problem that has occurred at least in three different programs, in three different contexts; its solution must be usable in these contexts and its use must lead to programs with “good” quality characteristics.

Therefore, patterns are often debated in Writers' workshops during which its authors and others debate the best way to both abstract the pattern and generalize its solution.

"Good" quality may have different meaning in different contexts, however anyone reading the solution of a pattern would feel that it is "good". This feeling is what Alexander named "Quality Without a Name". Patterns often do not work so well in isolation but need other patterns to complement and increase their benefit on quality. Language of patterns combine patterns that work "well" together, *i.e.*, they contribute to each other's quality without a name. Finally, patterns can be used to generate the architecture of a program but may lead to over-engineering this architecture that, consequently, must be refactored to be simplified.

2.1 Quality Without A Name

> This oneness, or the lack of it, is the fundamental quality for anything. Whether it is in a poem, or a man, or a building full of people, or in a forest, or a city, everything that matters stems from it. It embodies everything. — Alexander [Alexander Timeless way of Building]

There exist many quality characteristics. Internal characteristics relate to the inner working of a program while external characteristics relate to the quality visible by users of the programs. ISO 9126 and other standards define various such characteristics as well as their relations with one another and with the programs.

However, beyond these quality characteristics, there exists a feeling when a programmer and–or a user deals with a program. This feeling is often expressed unscientifically but none the less impacts considerably the relation of a programmer with the program: "nice" or "annoying" piece of code and design may ease or dramatically impede a programmers's ability to perform the tasks at hand.

This feeling relate to Alexander's concept of Quality Without a Name (QWAN). QWAN relates to the "good" feeling that applying patterns give to a programmer, the feeling of having applied the right solution to the right problem at the right time. It is difficult to define QWAN formally, but the Portland Pattern Repository's Wiki lists the following characteristics:

- Usability—Is the program something that people enjoy using? Would they miss it if it was no longer available?

- Readability—Is the intent of the program clear and well presented?

- Configurability—Can the user adapt the program to his or her needs?

- Reasonance-Does the program strike the user as special or unique, but at the same time, insightful and correct?

In addition to QWAN, there has been a growing interest in the literature on the use of design Patterns to improve quality. In particular, work has been carried out to study the

potential impact of patterns on programs. Yet, few works have investigated empirically their impact on quality.

Lange and Nakamura [LN95] demonstrated 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 program and, thus, in improving the understandability of the program. However this study was limited to a single quality attribute and to a small number of patterns.

Wydaeghe *et al.* [WVM+98] 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 [Wen01] evaluated the use of design patterns in a large commercial programs. The author concluded that patterns do not necessarily improve a program design. Indeed, a design can be over-engineered [Ker04] and the cost of removing design patterns is high. He did not perform a study on the impact of patterns on quality and provides only qualitative arguments.

McNatt and Bieman [MB01] examined the coupling between design patterns. They drew a parallel between modularity and abstraction in programs, and modularity and abstraction in patterns. They showed that when patterns are loosely coupled and abstracted, maintainability, factorability, and reusability are well supported. They concluded on the need for further studies to understand "good" pattern coupling methods. This work did not study the quantitative impact ofpatterns on quality.

Tahvildari *et al.* [TK02] studied the 23 design patterns from [GHJV94] and presented a layered classification of three primary relationships among these patterns: use, refine, and conflict, and 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.* [BSW+03, BAIM01] examined common recommended programming styles on several different programs, 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.

2.2 Languages of Patterns

Often, a pattern considered and used in isolation does not fulfill all the programmer's needs or does not give all the potential of its QWAN. A pattern must be used by programmers in collaboration with other patterns. Such consistent set of patterns form language of patterns. A well-known language of patterns is the language defined by Alexander in his work, for example the language of pattern related to building houses or the one related to planning a city. In software engineering, there exist many languages of patterns, even though none is currently more well-known as the language defined in the GoF's book. Languages of patterns exist also for different functional and non-functional contexts: for example security patterns form a language of patterns dedicated to preventing security issues in programs [YWM08].

2.3 Patterns Generate Architecture

In addition to improving quality, patterns and languages of patterns are also used to generate architecture of programs [BJ94]. In their seminal work on the subject, Beck and Johnson show how a language of patterns can be used by programmers to generate the architecture of a program and give it "good" quality. They illustrated their point by designing and implementing HotDraw, the Smalltalk ancestor to JHotDraw. HotDraw includes several patterns that make it easy to extend and adapt to different contexts by programmers.

2.4 Over-engineering and Refactorings

However, using patterns and languages of patterns to generate architectures may lead to over-engineering the design of a program [Ker04]. Over-engineering happens when a design or a piece of code is more flexible or sophisticated that it should be, most likely in preparation for future extensions that may or may not come. Over-engineering is the opposite of under-engineering, which occurs when a programmer chooses the path of least resistance to design and implement a program, leading to a solution that is suboptimal and that must be changed to adapt to foreseeable changes.

On the one hand, under-engineering is much more frequent than over-engineering, because programmers often work under time and cost pressures and, thus, cannot spend the required time to carefully think and craft their changes. Time and cost pressures often lead to the decay of the program design and implementation. On the other hand, some programmers try to fight the decay by using design patterns as much as possible, in case parts of a program need to change in the future or be adapted to a different context. Both under- and over-engineering arise from misunderstood requirements and uncarefully planned design and development. Both of them are problematic either because most of the program much be changed to adapt to new requirements and contexts (under-engineering) or because its sophistication makes it adaptable at a high cost and unnecessary complexity.

A solution to both under- and over-engineering is to apply refactorings prior to modifying the design or the code of the program. The refactoring step is necessary to cleanup the program and make it ready for change, at the right time and with the right amount of work.

Refactoring to/away from patterns is thus a preliminary step before modifying the program, like dusting a room is before painting it. It contributes to the change by making it easy and safe to perform, even though it is not the change *per se*.

3 Research Work on Design Patterns

The following sections introduce some work related to design patterns that illustrate their importance during program evaluation and comprehension.

3.1 Design Pattern Impact on Quality Characteristics

Many studies in the literature are based on the premise that design patterns improve the quality of object-oriented programs [GHJV94]. Indeed, it is claimed that design patterns improve the quality of programs and that every well-structured object-oriented design contains patterns (see for example [GHJV94, page xiii] or [Ven05]).

Yet, some studies, *e.g.*, [Wen01], suggest that the use of design patterns does not always result in "good" designs. For example, a tangled implementation of patterns impacts negatively the quality that these patterns claim to improve [MB01]. 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 [GHJV94] 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 programs as perceived by software engineers in the context of maintenance and evolution [KG08]. 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 programs in which design patterns are used [GGKS05]. 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 now 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. Contrary to popular beliefs,

[1] Although reusability in [GHJV94] 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.

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.

design patterns in practice do not always improve quality attributes, thus providing concrete evidence against common lore.

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 programs. 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 patterns is perceived as impacting negatively all quality attributes but scalability. Given the purpose of the pattern, it is

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.

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.

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 [GHJV94].

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.

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 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

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.

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.

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

Conclusion. 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 programs 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.

This study is the largest to date in term of the number of collected evaluations on design patterns and quality of programs. 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.

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.

3.2 Design Pattern Impact on Program Comprehension

No study has tried to establish so far the impact of design patterns on program comprehension. Consequently, we performed a study to determine whether the Visitor pattern is useful during maintenance [sGSH09]. Based on ISO-9126 [iso01], we focus on two major characteristics of maintainability: analysability and changeability. Consequently, we study the impact of the Visitor pattern on tasks representative of those characteristics: comprehension and modification tasks and we propose:

1. To compare developers' effort in the presence and absence of the Visitor pattern when performing comprehension and modification activities.

2. To compare developers' effort when the Visitor pattern is shown as described in [GHJV94] and with a different layout when performing comprehension and modification activities.

We design experiments to collect data to compare the developers' efforts when performing comprehension and modification tasks using different yet *semantically equivalent* UML class diagrams. We define and measure efforts as *the amount of attention* that developers must spend to perform the tasks: less attention and less time means less effort. We measure the developers' efforts using data collected with an eye-tracker to decide whether a class

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.

diagram decrease their efforts with respect to the others. We choose UML class diagrams because UML is a *de-facto* standard and the main constructs of class diagrams are well understood by developers. The class diagrams in our experiments either include classes playing roles in a design pattern or not and either show the design pattern following its *canonical* representation (*i.e.*, as described in [GHJV94]) or not. We only use the Visitor pattern because this pattern is widely used and is a good example for which several design alternatives exist. Moreover, using more than one pattern would have made it difficult to isolate their respective impact on the developers' efforts.

The design of our experiments is directed by our use of an eye-tracker to collect relevant data to assess the subjects' efforts when performing comprehension and modification tasks.

Eye-trackers. Many eye-trackers exist on the market. We use the EyeLink II from SR Research[2]. EyeLink II has the highest resolution (noise-limited at $< 0.01°$) and fastest data rate (500 samples per second) of any head mounted eye-tracker. Such an eye-tracker is composed of two computers and a head-band. The headband includes two cameras and an infra-red emitter. The cameras use infra-red rays that are reflected on the subject's cornea to register eye-movements. Four sensors are placed on the subject's screen. These sensors work with the infra-red emitter to gather the position of the head-band with respect to the screen. Along with the head-band cameras, these four sensors allow the system to compute precisely the position of the subject's gaze on the screen.

The data collected from the eye-trackers are of two types: raw positions and parsed positions. Parsed positions are expressed in terms of fixations and saccades based on physiological thresholds. Fixations are stabilisations of the eye during a gaze while saccades are movements from one fixation to another. The numbers of fixations and saccades can vary from a dozen to several hundreds depending on the size and complexity of the class diagram. These two pieces of data can be combined to measure the subjects' efforts.

[2]http://www.eyelinkinfo.com/

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

Hypotheses. We want to assess the following four null hypotheses, two for comprehension task (HC) and two for modification task (HM):

- HC_{0_1}: A class diagram with the Visitor pattern does not reduce the subjects' efforts during program comprehension when compared to a class diagram without Visitor pattern.
- HC_{0_2}: A class diagram using the canonical representation of the Visitor pattern does not reduce the subjects' efforts during program comprehension when compared to a class diagram using the Visitor pattern with another layout.
- HM_{0_1}: A class diagram with the Visitor pattern does not reduce the subjects' efforts during program modification when compared to a class diagram without Visitor pattern.
- HM_{0_2}: A class diagram using the canonical representation of a design pattern does not reduce the subjects' efforts during program modification when compared to a class diagram using the Visitor pattern with another layout.

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).

If the previous null hypotheses are rejected, we could assume (threats to the validity are presented in Section 3.2) that the following alternative hypotheses are true:

- HC_{a_1}: A class diagram with the Visitor pattern reduces the subjects' efforts during program comprehension.
- HC_{a_2}: A class diagram using the canonical representation of the Visitor pattern reduces the subjects' efforts during program comprehension.
- HM_{a_1}: A class diagram with the Visitor pattern reduces the subjects' efforts during program modification.
- HM_{a_2}: A class diagram using the canonical representation of the Visitor pattern reduces the subjects' efforts during program modification.

Variables. From the previous hypotheses, we identify the following independent variables:

- Design Alternative: CP, MP, and NP are the possible values for this variable. Each class diagram conveys the same semantics. NP (*no pattern*) diagrams do not

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).

use the pattern, as shown in Figure 2; CP (*canonical pattern*) show the canonical representation of the pattern, for example see Figure 1; MP (*modified pattern*) show the pattern with a modified layout, as illustrated in Figure 2. (No systematic layout modification has been performed because we only want to assess the familiar layout against a unfamiliar layout.)

- TASKS: Two different tasks are compared, one program comprehension task and one modification task: C or M.

In the rest of this paper, the different experiments performed by the subjects are named as:
"DESIGN_ALTERNATIVE"_"TASK", for example CP_C.

We retain two mitigating variables to put into perspective and better understand the results of our experiments:

- UML KNOWLEDGE: The subjects' knowledge of UML, established using a questionnaire. Value range is $[1, 2]$ where 2 means that a subject has a very good knowledge of UML and 1 that the subject knows UML.

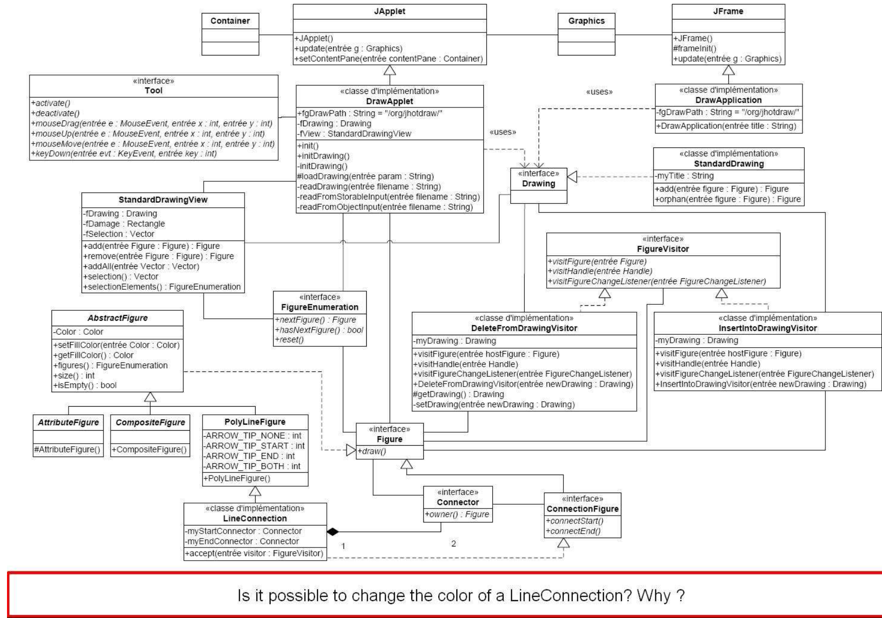


Figure 1: JHotDraw – Classical Design (CP)

- DP Knowledge: The subjects' knowledge of design patterns. This knowledge is also established using a questionnaire and follows the same value range as UML Knowledge.

We use fixations as the concrete manifestation of a subject's focus of attention in an area of the screen. We choose the normalised rate of relevant fixations [GK99] (NRRF) as dependent variable because it measures the subjects' efforts when performing tasks on diagrams. NRRF is an indicator of the subjects' efforts: a *higher* rate means *less* efforts (and vice versa).

We create for each diagram (NP, CP, and MP) two lists of *area of interest* and *area of relevant interest*, which include *all classes* and *all relevant classes*. An area of interest is any class in our diagrams that could be the focus of the subjects' attention to perform their comprehension or modification tasks (C or M). An area of relevant interest is any class that is *relevant* for the task at hand and, therefore, should be *more* the focus of the subject' attention during the task. Then, NRRF is defined as:

$$NRRF = \frac{\frac{\sum_{c \in \{Relevant\ Classes\}} F(c)}{\#\{Rel.\ Classes\}}}{\frac{\sum_{c \in \{Rel.\ Classes\}} F(c)}{\#\{Rel.\ Classes\}} + \frac{\sum_{c \in \{Non-Rel.\ Classes\}} F(c)}{\#\{Non-Rel.\ Classes\}}}$$

where F is a function that, given a class, returns the number of fixations performed by a subject in that class. Design alternatives (CP, MP, and NP) do not all have the same number of classes and therefore we normalise the number of relevant fixations (numerator) by the numbers of fixations in relevant and irrelevant classes.

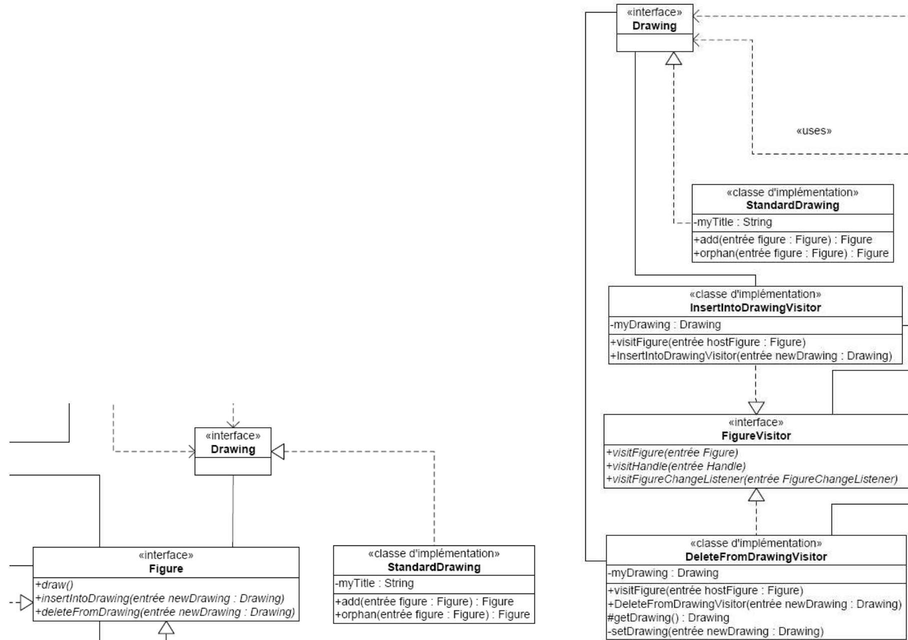


Figure 2: JHotDraw – No Pattern Design (NP, left) and Modified Design (MP, right)

Objects and Subjects. We choose three open-source programs as objects: JHotDraw, JRefactory and PADL. JHotDraw is a framework to implement technical and structured drawings. This framework provides support for the creation of geometric and user defined shapes. JRefactory is a code refactoring tool for Java programs. PADL is a meta-model for describing patterns and object-oriented programs. These programs all use the Visitor pattern. Since full documentation was not available, partial reverse engineering has been done on each of these three programs to represent their design using UML class diagrams. For each of these three programs, three semantically-equivalent versions of their structure have been designed.

The experimentation was performed with 24 subjects: 7 post-graduate and 17 graduate students at the Department of Informatics and Operations Research at University of Montreal. All students have designed software architectures and used UML class diagrams. These 24 subjects are placed into 3 balanced groups. The balancing simplifies and strengthen our statistical analysis of the collected data [WRH+99].

Questions. Appropriate questions must trigger the subjects' mental processes when performing their tasks [BT06]. We ask specific questions to the subjects with each design alternatives, as illustrated on Figure 1. Half the questions focus on the comprehension the diagram, the other half on a modification task. Questions were designed in such a way that each class diagram contains enough information to answer the questions. Typically, questions involved two or three classes and took no more than five (seven) minutes to answer for

comprehension (modification) tasks.

Analysis of Comprehension Task Data. We use the Mann-Whitney test to check our hypotheses because we cannot assume that our dependent variables are normal. We attempt to reject the null-hypotheses in favor of the alternative hypotheses. We also explore the two mitigating variables, UML KNOWLEDGE and DP KNOWLEDGE.

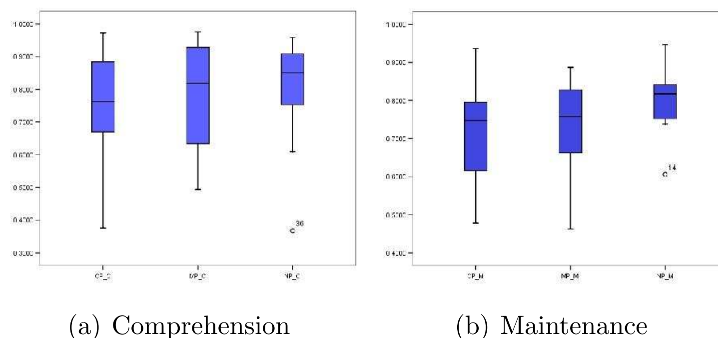
Table 11 summarises the effect of using the Visitor pattern on the comprehension of UML class diagrams. NRRF is given respectively for the three design alternatives CP_C, MP_C, and NP_C. The p-value columns report the statistical significances of the differences between CP_C and NP_C (HC_{0_1}) and MP_C and NP_C (HC_{0_2}).

	CP_C	MP_C	NP_C	p-value (HC_{01})	p-value (HC_{02})
NRRF (%)	75.77	77.95	81.49	0.221	0.663

Table 11: Effect of Visitor on comprehension

We observe almost the same NRRF for CP_C and MP_C (76% and 78%). The NRRF for NP_C is higher (81%) than for CP_C and MP_C. The distributions presented in Figure 3(a) show that the medians for NRRF are around 75% and 85% for all tasks. However, there is a large variance for NRRF between subjects working on MP_C, *i.e.*, a wider box plot, compared to subjects working on CP_C and NP_C. There is also a wide variance for the first quartile of CP_C. Values range from 38% to 67%. This variance could indicate that the subjects' levels of knowledge in UML and–or design patterns play a role when the Visitor is present in the diagrams.

In conclusion, the significance tests presented in Table 11 indicate that the presence of the Visitor pattern as well as its layout do not have a significant impact on the comprehension of UML class diagrams.



(a) Comprehension (b) Maintenance

Figure 3: Comprehension and maintenance data distribution (abscissa: DESIGN ALTERNATIVES, ordinate: NRRF values)

Analysis of Modification Task Data. The differences for NRRF between tasks CP_M, MP_M, and NP_M are important for diagrams without the Visitor pattern when compared to diagrams with the pattern: roughly $7\% - -8\%$ more, as shown in Table 12.

	CP_M	MP_M	NP_M	p-value (HM_{01})	p-value (HM_{02})
NRRF (%)	71.97	73.46	80.18	0.027	0.725

Table 12: Effect of Visitor on maintenance

Figure 3(b) shows that subjects working on diagrams with the Visitor pattern and its classical layout have clearly lower values for NRRF. Moreover, subjects working on diagrams without the Visitor pattern have a uniform effort, *i.e.*, flat box plots, compared to those of the two other groups.

We conclude that the Visitor pattern, when present in diagrams with the classical layout, has a significant impact on the modification task (p-values around 0.026): tasks using CP_M require less effort than using MP_M or NP_M. However, the impact is not significant when the Visitor pattern is presented with its modified layout (p-values 0.195 and 0.126). We conjecture that when its layout is modified, the Visitor pattern is difficult to recognise in the diagram and then the impact of its presence on the subjects' efforts is attenuated.

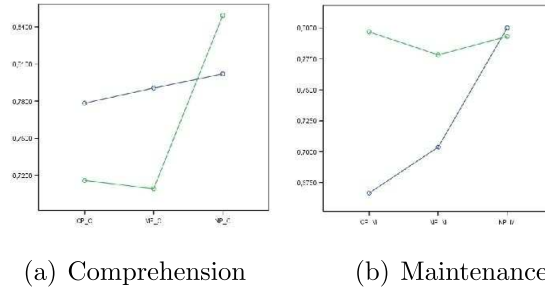
Impact of Mitigating Variables. The results presented show that there is a positive impact of the Visitor pattern on the subjects' efforts for modification tasks only when it is displayed using its classical layout. For the comprehension task, no significant impact at all can be reported.

Considering the variances for NRRF for the program comprehension task, we decided to investigate if the subjects' levels of knowledge in UML and design patterns could mitigate the results. Figure 4 (left) shows the impact of UML knowledge on comprehension tasks. On NP_C diagrams, subjects who have a *very good* knowledge of UML (level 2) perform better than those having only a *good* knowledge (level 1).

For the modification task, see Figure 4 (right), subjects with *very good* UML knowledge (level 2) perform better than other subjects. This result might explain why subjects with better UML knowledge might maintain programs more easily.

However, the impact of the UML knowledge cannot be confirmed by a 2-way ANOVA test. Table 13 (left) gives the test significance values for the impact of UML Knowledge and the combined impact (Design Alternative $\times$ UML Knowledge) for comprehension and modification tasks. All p-values are greater to 0.05.

The same analysis was conducted for the impact of DP Knowledge. We notice an important difference in behavior on NP_C between subjects with a very good knowledge (level 2) and a good knowledge (level 1) of design patterns. The same behavior can be seen for CP_M. Figures 5 (left) and (right) show differences of roughly 10% for NRRF. We observe also that for modification tasks, in Figure 5 (right), there is a difference between groups of subjects (levels 1 and 2) on all the design alternatives. This difference is an indication



(a) Comprehension (b) Maintenance

Figure 4: Impacts of UML Knowledge and Visitor (abscissa: Design Alternatives, ordinate: NRRF estimated mean variances for level 1 (blue) and 2 (green))

(a) UML Knowledge

NRRF	Comprehension	Maintenance
UML Knowledge	0.494	0.061
Combined	0.471	0.267

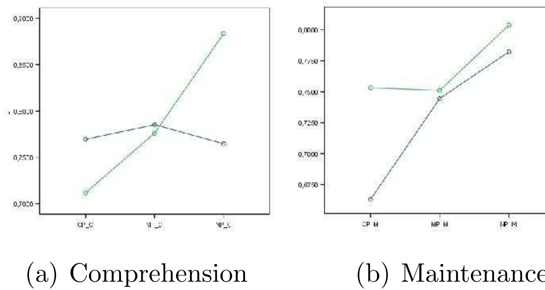
(b) DP Knowledge

NRRF	Comprehension	Maintenance
DP Knowledge	0.692	0.245
Combined	0.217	0.546

Table 13: 2-way ANOVA tests of the tmpacts of UML Knowledge and DP Knowledge

that the level of knowledge of design patterns has an impact on modification tasks. Yet, 2-way ANOVA tests give p-values greater than 0.05, as shown in Table 13 (right). However, p-values of 0.06 indicate that our observation on the impact on modification need to be explored in a replication study.

Threats to Validity. We identified three possible threats to **internal validity**: maturation, instrumentation, and diffusion of the treatments. In the case of *maturation*, we addressed the learning and fatigue effects by presenting the tasks (questions to answer) in random and different orders to subjects. We reduced the fatigue effect also by limiting the subject effort performing all the tasks (estimated time of 20 to 30 minutes). The *instrumentation* threat is related to the use of the eye-tracker. Subjects had to wear a headband with the infrared camera. They had to minimise their head movements to avoid decalibration. To circumvent this threat, we analysed eye-movement movies recorded during the experiment of each subject. We only detected one case of decalibration and the corresponding data was discarded. Finally, to prevent subjects from learning the treatments before hand, we gave instructions to each subject to not talk about the experiment before a fixed date corresponding to the end of all the experiments with subjects. We are confident that the instructions were followed by the subjects considering their perceived motivation.



(a) Comprehension (b) Maintenance

Figure 5: Impacts of DP Knowledge and Visitor (abscissa: Design Alternatives, ordinate: NRRF estimated mean variances for level 1 (blue) and 2 (green))

During the study, we specifically addressed four threats to **construct validity** : mono-operation bias, mono-method bias, hypothesis guessing, and apprehension. Concerning the two first threats, we used the diagrams extracted from three programs with different application domains. However, we only used one dependent variable, NRRF, which could introduce a bias. We did not inform the subject about the goal of the study before hand to anticipate the hypothesis guessing. They were not aware about presence of the patterns in the diagrams. We just explained them in the tutorial that they had to perform tasks on different diagrams coming from different programs. None from the subjects had more than one version of the same diagram. Finally, different actions were performed to prevent the apprehension threat. First, the eye-tracker was explained to the subjects and they were reassured about the absence of risks related to infrared camera on their eyes. Second, the subject were assigned identifiers to assure that no relation could be traced between them and their data Finally, although the subjects were asked to perform the task diligently, we did not set a predefined time to avoid time pressure.

For **external validity** threats, we considered the interactions of selection and setting with the treatments. The issue of whether student subjects are representative of software professionals has been discussed in several studies (see [BLPYB05] for example). In our case, we used graduate and postgraduate students that have knowledge of UML and design patterns comparable to most professionals. For the setting interaction, we considered the size and the complexity of the diagrams. We reverse-engineered them from open-source programs that are extensively used. The diagrams presented to the subjects contains roughly 20 classes, which is usual for comprehension and maintenance tasks.

Threats to **conclusion validity** relates to random irrelevancies in setting and heterogeneity of subjects. To prevent the first threat, we used a quiet laboratory. We performed the experiment several times with some subjects (not included in the study sample) to detect any problems. Regarding the choice of subjects, our sample is quite heterogeneous. In addition, we used UML and DP Knowledge as mitigating variables and verified that their impacts are less important than the presence of patterns.

Conclusion. The analysis of the data collected using an eye-tracker showed that the Visitor pattern does not reduce the subjects' efforts for comprehension tasks. However, cognitive factors related to the subjects' familiarity with patterns and UML (in general) have observable (even though not significant) influences on comprehension and modification tasks. The analysis showed also that the Visitor pattern with its *canonical* representation reduces the developers' efforts for modification tasks.

The largely-admitted intuition that patterns help developers during comprehension and maintenance tasks seemed confirmed only for modification tasks when using the Visitor pattern. Therefore, other experiments are required to confirm our results and also to conclude for other design patterns.

4 Conclusion

Design patterns are an important and multi-purpose tool in the tool box of software engineers. They are important because they embody expert knowledge in the form of reusable solution to recurring design problems. There are multi-purpose because they can be used to generate architecture, to improve quality characteristics, and to improve program comprehension.

Yet, design patterns must not be over-used or used in the wrong context. Indeed, using design patterns can lead to over-engineering an architecture, giving it unnecessary flexibility at the cost of impeded comprehension and negative impact on other quality characteristics. Also, the use of design patterns remains so far an art, which can only be mastered after many years of using/removing patterns in programs.

Consequently, much work remains on design patterns. In particular, it would be important to define a framework of pattern-based software engineering to understand and leverage the use of patterns to reduce development and maintenance costs. The framework would allow putting patterns into perspective, relating them with one another, understanding them as a whole, and assessing their impact on quality and program comprehension, thus achieving three benefits:

- **Benefit 1**: Cast in a consistent framework patterns at different level of abstraction, on different artifacts, of different types, and for different application domains.
- **Benefit 2**: Understand the roles and impact of patterns on quality, program comprehension and development and maintenance costs through the framework.
- **Benefit 3**: Recast existing software engineering techniques in the framework, including feature location, software evolution, and software quality evaluation.

Acknowledgment

The author gratefully and respectfully thanks Giuliano Antoniol, Massimiliano Di Penta, Naji Habra, Foutse Khomh, Naouel Moha, and Stéphane Vaucher for their help on part(s)

of this entry. This entry has been partly funded by NSERC, in particular through the Canada Research Chair on Software Patterns and Patterns of Software and the Discovery Grant #293213.

References

[AMC03] Deepak Alur, Dan Malks, and John Crupi. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall, 2nd edition, May 2003.

[BAIM01] James M. Bieman, Roger Alexander, P. Willard Munger III, and Erin Meunier. Software design quality: Style and substance. In *Proceedings of the* 4th *Workshop on Software Quality*. ACM Press, March 2001.

[BJ94] Kent Beck and Ralph E. Johnson. Patterns generate architectures. In Mario Tokoro and Remo Pareschi, editors, *Proceedings of* 8th *European Conference for Object-Oriented Programming*, pages 139–149. Springer-Verlag, July 1994.

[BLPYB05] Lionel C. Briand, Yvan Labiche, Massimiliano Di Penta, and Han Yan-Bondoc. An experimental investigation of formality in UML-based development. *Transaction on Software Engineering*, 31(10):833– –849, October 2005.

[BMB+98] William J. Brown, Raphael C. Malveau, William H. Brown, Hays W. McCormick III, and Thomas J. Mowbray. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, 1st edition, March 1998.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons, 1st edition, August 1996.

[BSW+03] 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* 9th *international Software Metrics Symposium*, pages 40–49. IEEE Computer Society Press, September 2003.

[BT06] Roman Bednarik and Markku Tukiainen. An eye-tracking methodology for characterizing program comprehension processes. In *2006 symposium on Eye tracking research & applications*, pages 125–132, 2006.

[Cop91] James O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1st edition, August 1991.

[Fow96] Martin Fowler. *Analysis Patterns : Reusable Object Models*. Addison-Wesley – Object Technology Series, 1st edition, October 1996.

[Fow99] Martin Fowler. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, 1st edition, June 1999.

[GGKS05] Yann-Gaël Guéhéneuc, Jean-Yves Guyomarc'h, Khashayar Khosravi, and Houari Sahraoui. Design patterns as laws of quality. University of Montreal, 2005.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1st edition, 1994.

[GK99] Joseph H. Goldberg and Xerxes P. Kotval. Computer interface evaluation using eye movements: Methods and constructs. *International Journal of Industrial Ergonomics*, 24(6):631–645, October 1999.

[iso01] *ISO/IEC 9126- Software Engineering - Product Quality*. International Organization for Standardization - ISO, 2001.

[Ker04] J. Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 1st edition, August 2004.

[KG08] Foutse Khomh and Yann-Gaël Guéhéneuc. Do design patterns impact software quality positively? In Christos Tjortjis and Andreas Winter, editors, *Proceedings of the* 12th *Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, April 2008.

[LN95] Danny B. Lange and Yuichi Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Proceedings of the* 10th *annual conference on Object-oriented programming systems, languages, and applications*, pages 342 – 357. ACM Press, 1995.

[MB01] William B. McNatt and James M. Bieman. Coupling of design patterns: Common practices and their benefits. In T.H. Tse, editor, *Proceedings of the* 25th *Computer Software and Applications Conference*, pages 574–579. IEEE Computer Society Press, October 2001.

[sGSH09] Sébastien Jeanmart, Yann-Gaël Guéhéneuc, Houari Sahraoui, and Naji Habra. Impact of the visitor pattern on program comprehension and maintenance. In James Miller and Rick Selby, editors, *Proceedings of the* 3rd *International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE Computer Society Press, October 2009. 10 pages.

[TK02] 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*6th *European Conference on Software Maintenance and Reengineering*, pages 230–240. IEEE Computer Society, March 2002.

[Ven05] Bill Venners. How to use design patterns – A conversation with Erich Gamma, part I, May 2005. http://www.artima.com/lejava/articles/gammadp.html.

[Wen01] 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* 5th *Conference on Software Maintenance and Reengineering*, pages 77–84. IEEE Computer Society Press, March 2001.

[WRH+99] Claes Wohlin, Per Runeson, Martin Host, Magnus C. Ohlsson, Bjorn Regnell, and Anders Wesslen. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, 1st edition, December 1999.

[WVM+98] 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.

[YWM08] Nobukazu Yoshioka, Hironori Washizaki, and Katsuhisa Maruyama. A suvery on security patterns. *Progress in Informatics*, (5):35–47, 2008.