Report of Design Patterns

Ligang Wang
lgwang@cs.concordia.ca

Wenhua Fan
wenhu_fa@cs.concordia.ca

Ping Wang
pollywang98@hotmail.com

Liyang Zhao
liyan_zh@cs.concordia.ca

Department of Computer Science, Concordia university
1455 de Maisonneuve Blvd. W. Montreal, Quebec, Canada

Abstract

Design pattern is a term of the abstraction from a concrete form that keeps recurring in specific non-arbitrary contexts. Design Patterns are devices that allow programs to share knowledge about their designs. It seems that the benefits of design pattern are so obvious. However, design pattern is so complicated that it makes the software even more difficult to be maintained. In this report, we show how we documented patterns, and the advantages and disadvantages of documenting design patterns. Finally, we get some conclusions.

1. Introduction

Design patterns are being considered to be a promising approach to system development. The main idea behind design patterns is to record experience in designing object-oriented software, thus allowing developers to communicate more effectively. Design patterns make it easier to reuse successful designs and architectures. Expressing proven techniques as design patterns makes them more accessible to developers of new systems. Design patterns help you choose design alternatives that make a system reusable and avoid alternatives that compromise reusability. Design patterns can even improve the documentation and maintenance of existing systems by furnishing an explicit specification of class and object interactions and their underlying intent.

Design patterns are widely and increasingly recognized as important software development methods. Their use as software understanding tools, though generally acknowledged has been scarcely explored. Patterns are most useful in understanding software when they are well documented. Sometimes they are described separately from code as design comments. Nevertheless they hold a strong relationship to the source code and thus they should be documented at the source level too. A pattern is not the reuse of implementation. Documenting a pattern allows ideas to be reused. However, There are many limitations and Pitfalls when using and documenting design pattern, such as over-engineering and the challenge of successful usage of patterns. We give some study to show that kind of problem.

2. Design Patterns

2.1. Definition of design pattern

Design pattern is a term of the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts. More specifically, the concrete form which recurs is that of a solution to a recurring problem, but a pattern is more than just a battle-proven solution to recurring problems. A design pattern is a named nugget of instructive information that captures the essential structure and insight of a successful family of proven solutions to a recurring problem that arises within a certain context and system of forces.

In general, a design pattern has four essential elements: the pattern name, the problem, the solution and the consequences [4].

The pattern name is a handle we can use to describe a design problem, its solutions, and consequences in a word or two. It lets us design at a higher level of abstraction.

The problem describes when to apply the pattern. It explains the problem and its context.

The solution describes the elements that make up the design, their relationships, responsibilities, and collaborations. The pattern provides an abstract description of a design problem and how a general arrangement of elements solves it.

The consequences are the results and trade-offs of applying the pattern. They are critical for evaluating design alternatives and for understanding the costs and benefits of

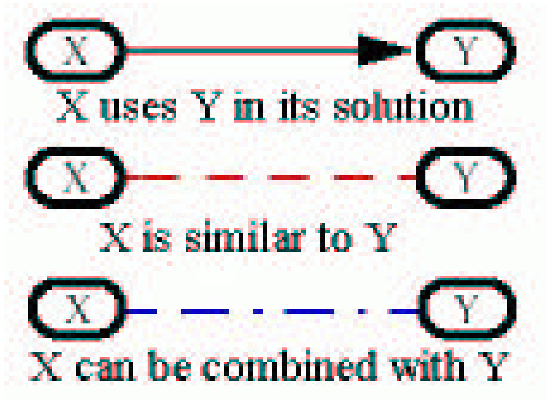


Figure 1. Classification of Relationships

applying the pattern. The consequences of a pattern include its impact on a system's flexibility, extensibility, or portability.

2.2. Relationship between Design Patterns

Relationships refer to different aspects of design patterns. We can classify relationships between design pattern X and design pattern Y into into three categories: Figure 1 describes the classification of the relationship.

"X uses Y" relationship: When building a solution for the problem addressed by X, one subproblem is similar to the problem addressed by Y. Therefore, the design pattern X uses the design pattern Y in its solution. Thus, the solution of Y (e.g. class structures) represents one part of the solution of X.

"X is similar to Y" relationship : They address a similar kind of problem (not a similar kind of solution). In several cases, these similarities are also expressed in the classification given in the catalogue, e.g. the catalogue classifies both Prototype and Abstract Factory in the category "Object - Creational".

"X can be combined with Y" relationship : A typical combination of design patterns is the combination of X and Y (e.g. Iterators traverse Composite structures). In contrast to "X uses Y", X does not use Y in its solution (or vice versa).

2.3. Categories of Design Patterns

Graphical notations, while important and useful, aren't sufficient to design patterns. To reuse the design, we must also record the decisions, alternatives, trade-offs, concrete examples that led to it, In addition, design patterns should be described using a consistent format, such as pattern name and classification, intent, also known as, motivation, applicability, structure, participants, collaborations, consequences, implementation, sample code, known uses, related patterns.

Design patterns vary in their granularity and level of abstraction. The classification of design patterns helps you learn the patterns in the catalog faster, and it can direct efforts to find new patterns as well.

Design patterns can be classified by two criteria. The first criterion, called purpose that reflects what a pattern does, divided design patterns into creational, structural, or behavioral purpose. Creational patterns such as abstract factory, builder, factory method, prototype and singleton concern the process of object creation . Structural patterns such as adapter, bridge, composite, decorator, facade, flyweight and proxy deal with the composition of classes or objects. Behavioral patterns such as chain of responsibility, command, interpreter, iterator, mediator, memento, observer, state, strategy, template method and visitor characterize the ways in which classes or objects interact and distribute responsibility.

The second criterion of classification partitions design patterns into three semantically different layers: Basic design patterns and techniques, design patterns for typical software problems(middle layer), design patterns specific to an application domain(high layer).

Basic design patterns and techniques layer contains the design patterns, which are heavily used in the design patterns of higher layers and in object-oriented systems in general [13]. The problems addressed by these design patterns occur again and again when developing object-oriented systems. The design patterns are thus very general. When building a system, one would often look upon them more as basic design techniques than as patterns. The intentions of these design patterns(Table 1) are very general and applicable to a broad range of problems occurring in the design of object-oriented systems.

Design patterns for typical software problems are the middle layer of design patterns, which comprises design patterns used for more specific problems in the design of software. These design patterns are not used in design patterns from the basic layer, but in patterns from the application specific layer, and possibly from the same layer. The problems addressed by these design patterns are not typical of a certain application domain. Builder, Prototype and Abstract Factory address problems with the creation of objects; Iterator traverses object structures; Command objectifies an operation; and so on.

Design patterns specific to application domain are the high layer of design patterns. In this layer, design patterns are the most specific and can often be assigned to one or more application domains. The current catalogue contains almost no application specific design patterns. To be more

Design pattern	Purpose of the design pattern contexts
Adapter	Adapting a protocol of one class to the protocol of another class
Composite	Single and multiple, recursively composed objects can be accessed by the same protocol
Glue	Encapsulating a subsystem
Mediator	Managing collaboration between objects.
Memento	Encapsulating a snapshot of the internal state of an object
Objectifier	Objectifying behaviour
Proxy	Attaching additional properties to objects
Solitaire	Providing unique access to services or variables
Template Method	Objectifying behaviour (primitives will be varied in subclasses)

Table 1. Basic design patterns with their respective purposes

specific, interpreter is used to parse simple languages. The catalogue lists some of the known uses of Interpreter, e.g. parsing constraints and matching regular expressions. Compiler construction is the major application domain. Figure 2 shows the Arrangement of design pattern in layers.

Most patterns are generic and applicable to a broad range of problems. Frameworks [14] [6] can also be considered as high-level design patterns, usually consisting of many interrelated design patterns of lower levels.

2.4. Benefits of Application

It seems that the benefits of design pattern is so obvious, but some time, design pattern is used in situations where their flexibility is not needed. So the solution with design patter is more complicated than necessary. In order to find out whether or not design pattern is beneficial to software maintenance, we will introduce some experiences.

In our experiment, we propose two different, functionally equivalent versions program. One is PAT (pattern version),which employs one or more design patterns. The other is ALT (alternative version), which represents a simpler design using fewer design patterns or simplified versions of them. The experiment had two parts. At beginning, the participants have had no pattern knowledge at all. The first part (the pretest, PRE) was preformed at the first day. We call it as E1. Then a pattern course was taught. After that, the participants have significantly higher pattern knowledge than before. Then the second part (the posttest, POST) was performed. We call this part as E2. We also record the time that was taken for each maintenance task.

2.4.1 Composite and Abstract Factory: Graphics Library (GR)

The graphics library contains a library for creating manipulating, and drawing simple types of graphical objects on different types of graphical output devices [8]. Design Patterns used in the PAT version of this program are ABSTRACT FACTORY for the generator classes and COMPOSITE for hierarchical object grouping. The ALT version of the program realized the instantiation of the appropriate classes for each graphical output device by switch-statements in but a single generator class. The combination and manipulation of graphical object groups are realized with a quasi-COMPOSITE. The only difference is that groups are not treated as graphical objects as in the COMPOSITE.

Our work task is to add a third type of output device. Subjects maintaining the Pat program had to introduce a new concrete factory class, extend the factory selector method, and add two concrete product classes. Subjects in the Alt groups had to enhance the switch statements in all methods of the generator class. The appropriate classes of graphical objects for the new output device had to be added as for Pat.

Regarding the maintenance task, the time for finding the changes and additions is expected to be almost equal for the Pat and the Alt groups. So the main difference in time required for this task will be caused by program understanding. Here we expect the simpler Alt program to be easier to understand, at least in the pre-test. Pattern knowledge will help both groups because of the Composite structure in both programs. The pattern group may profit a little more from the pattern course, because it eases understanding the structure of the Abstract Factory.

Finally, the results support both expectations. Both groups maintaining the Alt program were faster than the corresponding Pat groups with the same pattern knowledge level, supporting E1 (15% faster, 32 minutes vs. 37.5 minutes, total significance p = 0:10). The improvement from Pre to Post (E2) is 17.3% (40.5 minutes vs. 33.5 minutes, significance p = 0:17) for the Pat group and 22.8% (36.4 minutes vs. 28.1 minutes, significance p = 0:031) for the Alt group. That is 21.2% overall (38.6 minutes vs. 30.4 minutes, significance p = 0:021).

2.4.2 Decorator: Communication Channels (CO)

Communication Channels is a wrapper library. A communication channel establishes a connection for transparently transferring arbitrary-length packets of data and one can turn on additional logging, data compression, and encryption functionality. The library does not implement the

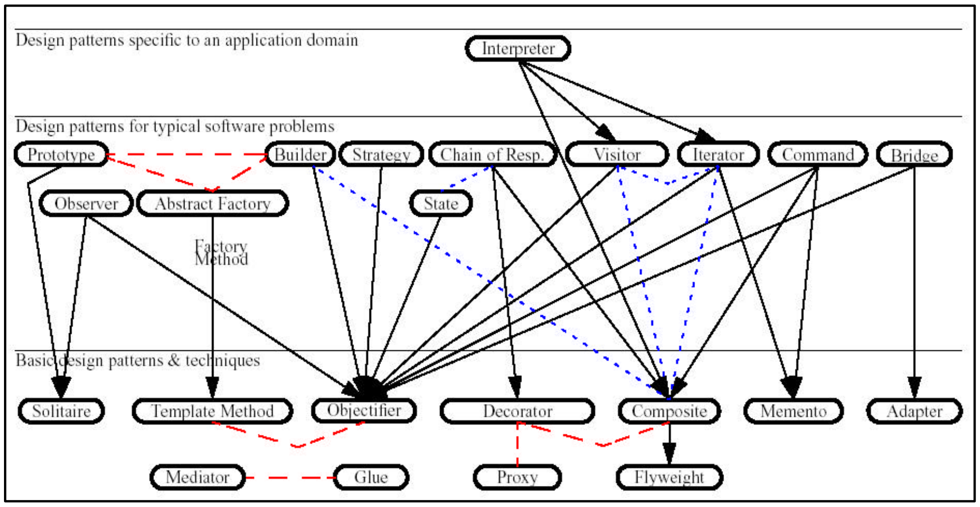


Figure 2. Arrangement of design pattern in layers

functionality itself, but only provides a Facade to a system library. However, this application of the Facade pattern is irrelevant to the experiment.

The Pat version is designed with a Decorator for adding the functionality to a bare channel, having the classes for logging, data compression, and encryption as decorator classes. The program consists of 365 lines in six classes.

The Alt version comprises but a single class, that uses flags and if-sequences for turning functionality on or off; the flags can be set when creating a channel. It consists of 318 lines. Communication channels is the only program where the Alt program has a structured (as opposed to object-oriented) design.

Now our work task is to enhance the functionality of the program such that error-correcting encoding (bit redundancy) can be added to communication channels.” The underlying functionality is again provided by a given class, so the subjects only had to integrate the new functionality into the program.

The Pat subjects had to add a new Decorator class while the Alt subjects had to make additions and changes at various points in the existing program.

We expect two influences of the Decorator on the subjects’ behavior. First the Alt version is easier to understand because its behavior is not delocalized as in the multiple decorator classes. This would lead to the conclusion that the Alt groups are faster than the Pa t groups, especially in the pretest. Second, a counter-influence results from the structure of the Decorator : the functionality is encapsulated in classes and one need hardly care about mutual influences. In particular, in the Alt version the subjects have to ensure they add the new functionality at the correct places in the program for proper sequencing of the various switchable functionalities; this will consume time and may lead to omissions and mistakes. We expect the second influence to be stronger than the first and hence the Pa t version to be preferable (E1), especially at higher levels of pattern knowledge (E2).

Finally, The result is that the Pat groups are indeed significantly faster than Alt groups (38% faster, 28.8 minutes vs. 46.2 minutes, significance $p<0{:}001$), confirming E1. The pattern-solution is clearly preferable. There is no significant difference between Pre - Alt and Post - Alt as expected (46.5 minutes vs. 45.9 minutes, significance $p = 0{:}46$), but also none between Pre-Pat and Post-Pat (27.5 minutes vs. 29.8 minutes, significance $p = 0{:}29$), thus rejecting E2. This means the positive effect of pattern use is even independent of pattern knowledge in this case! The pattern-solution is also superior in terms of correctness: Errors were made by 7 out of 8 Pre - Alt subjects and by 6 out of 7 Post - Alt while in the Pat group no errors occurred at all.

From the above, we can see that it is usually but not always useful to use a design pattern if there are simpler alternatives. And a thorough understanding of specific design patterns often helps when maintaining programs using them, even if these programs are neither very large nor very complicated.

3. Documenting Design Pattern

A pattern is not the reuse of implementation. Documenting a pattern allows ideas to be reused. Patterns can be seen as a literary form used to document ideas and concepts that are in common use. Patterns are not invented, that is, they are discovered. A pattern gives us a vocabulary to better express ourselves. Documenting using patterns allows for reuse of documentation. Simply refer to a previously documented pattern. Patterns catalogues are not simply listings of solutions to problems. The pattern documents the context the of the problem, the forces involved and gives a rationale for the solution.

The usefulness of patterns for software understanding purposes has been scarcely investigated. There is only one empirical study that is described in [3]. This work suggests that there is empirical evidence of such usefulness. The work of Prechelt and colleagues has the merit of stressing two important aspects of design patterns. First they can be used as a mean to understand a system (or to ”explain it” using Coplien's perspective [2]). Second the mere presence of patterns is not enough; they should be documented to be effective in understanding.

What lacks in their work is an analysis of how patterns are used in software system and how they can be documented. There is no agreement on how to document pattern use. And is such pattern documentation helpful for understanding a program more quickly and designing better solutions for given maintenance tasks? The following will address these drawbacks by providing four main contributions:

- An analysis of the use of pattern
- A proposal of how to document the use of pattern, implemented by means of a standard code documentation system
- If pattern documentation helpful for understanding a program
- Fallacies and Pitfalls for the use of design patterns and documentation

3.1. Documenting

How to document the pattern well? There is the standard solution that is an English description structured using a template and collected together into a catalogue. In fact, the template-based documentation strongly resembles one form of knowledge representation technique from the AI/Cog Science communities: knowledge schemas (which are logic analogues of data schemas in databases). Design pattern forms are similar to essay structures. When you write an essay, you try to structure the text (introduction, conclusion, etc.) so that it is easier to read. The design pattern forms are similar: they provide standard places for you to pour your pattern descriptions into. The important part is the pattern description: the pattern form merely helps in the writing and reading. Obviously having standardized forms makes writing and reading patterns easier, hence the reason for the attention paid to pattern forms.

3.1.1 Pattern Language

In The Timeless Ways of Building (Alexander, 1979), Alexander identified three essential elements that should exist within any pattern, they are:

1. A rule to identify the relationship between the pattern and its context.
2. A system of forces which arises within the context.
3. A solution or configuration to resolve the forces within the context.

In the software engineering community, a different format that had been used to describe patterns, depending on the author of different ”patterns handbook”. Appleton (1997) had highlighted the following elements.

1. Name - A unique and meaningful name to identify the pattern. The pattern name should form a vocabulary for communication among software engineers; it should also reflect the structure or knowledge that the pattern describes. Patterns writers should be aware of other aliases commonly used in the industry and document them too.
2. Problem - statement to describe the intention of the pattern, such as its aims, objectives and motivation.
3. Context - a description of the environment in which the pattern is applicable; this may be viewed as the preconditions for applying the solution.
4. Forces - the notion of force generalizes the kind of criteria that software engineers use to justify designs and implementations (Lea, 1997). This component contains a description of the forces within the context and how they affect each other. Evaluating the effect of these forces, software engineers would consider various trade-offs for applying the solution.
5. Solutions - Patterns describe solutions as a collection of static and dynamic rules that should be applied for solving the problem while maintaining a balance among all forces. The pattern should describe not only the static structure but also the dynamic behavior of the solution. It is possible that variants of the solution exist, which should be described in the pattern too.

6. Examples - Examples of how to apply the pattern would help pattern users understand the pattern's use and applicability better, this is especially important to new pattern users.

7. Resulting Context - Applying solutions onto an existing context would transform the system into a resulting context; pattern users should understand the consequences of applying the solution, as well as other problems or patterns that might arise in the new context. In another word, this component captures the post- conditions and side effects (Appleton, 1997) of the pattern.

8. Rationale - Since patterns exist together with a set of forces within a particular context, there must be a rationale to justify the use of the proposed solution as well as the way the solution resolves these forces. This component tells why a solution works, why it is "good" and how it works.

9. Related Patterns - this element identifies any dynamic or static relationship with other related patterns. These relationships may exist because of similar initial or resulting context, sequence of applicability, similar forces, or dependence among the patterns.

10. Known Uses - The Rule of Three (Hohmann, 1998) says that unless the solution described in the pattern has been used in at least three systems, it is rarely considered a pattern. Since patterns are "proven solutions to solve recurring problems", it is very important to document known uses to prove that the solution proposed is indeed proven; on the other hand, such documentation would serve as examples too.

This will allow a broader range of people within your organization to consume, contribute, and create new patterns, ultimately growing your design knowledge base. A common approach to documenting patterns will be emerging using a "common pattern language". Currently, the most common language has been based on working by architect Christopher Alexander and his book.

3.1.2 Pattern use

We are interested in investigating how an intentional and documented use of design patterns can help to understand a software system. In particular we will focus on the documentation of patterns at the source code level. Our work addresses the use of patterns in the Java programming language, Javadoc tool and to generate HTML documentation. However the proposed approach can be adapted to other programming languages, such as C++, using a suitable code documentation application [12]. Design patterns usually describe an idea at a high level of abstraction. Although sample reference implementations are usually provided, there are several possible variations in a pattern implementation. In addition, often patterns must be modified and adapted to serve the needs of the software system where they are used.

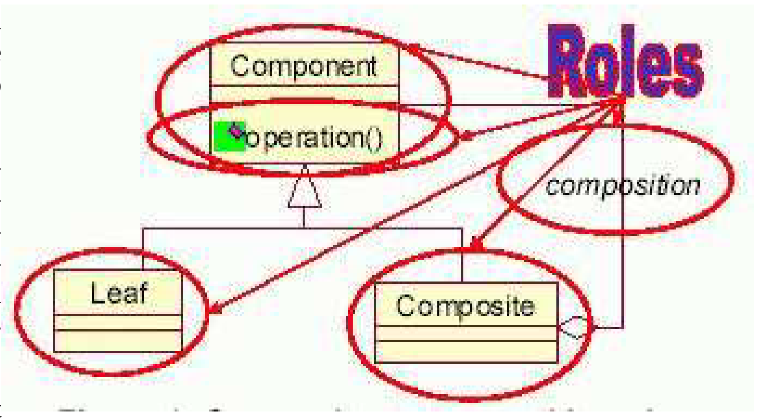


Figure 3. Composite Pattern and its Roles

Often a design pattern is described by means of a class diagram. For instance Figure 3 shows the composite design pattern described in [4]. This pattern is used to represent part-whole hierarchies while minimizing the difference between composite and leaf objects.

3.1.3 Javadoc

We focus now on the problem of documenting Java programs. The standard documentation of all the class libraries in the Java environment conforms to the Javadoc format. Thus we investigate this tool and its customization capabilities.

Javadoc [9] is a tool that parses the declarations and documentation comments in a set of source files and produces a group of cross linked HTML pages describing the classes, inner classes, interfaces, constructors, methods, and fields. The documentation produced by Javadoc can be customized using tags. A tag is a special keyword within a doc comment that Javadoc can process. Javadoc has standard tags, which appear as @tag, and in-line tags, which appear within braces, as @tag.

As of version 1.4, the Javadoc tool can be customized in two ways: using doclets or using taglets. It is possible to use doclets to customize Javadoc output. A doclet is a program written with the doclet API that specifies the content and format of the output to be generated by the Javadoc tool. Doclets allow to define the overall structure and format of the documentation, for instance they can be used to produce documentation is other formats than HTML.

```
/**
* This class purpose is&
* @pat.name Composite {@pat.role Leaf}
* @pat.task it represents a variable,
* {@link #evaluate()} gives the value
*/ class Variable { }
```

Figure 4. Code Example

```
The class purpose is&
Pattern:
        Composite, role: Leaf
Pattern task:
        It represents a variable, evaluate() gives the
        value.
```

Figure 5. Comments

3.1.4 Pattern Taglets

Pattern in Java code are documented by means of additional tags. The programmer can use these new tags in addition to the standard ones. The tags we propose to document the use of patterns are the following:

- @pat.name: the name of the pattern. This is a standard tag that applies to an element of a pattern instance. For the time being only the patterns described in the GoF book [4] are valid. The name will be represented in the documentation as a link to online description of the pattern.
- @pat.role: pattern role. This is an in-line tag that describes the role played by the element of the pattern instance; it must be nested inside a @pat.name tag.
- @pat.task: pattern task. This is a standard tag that is used to describe the task performed by and instance of a pattern of by one of its parts.
- @pat.use: pattern use. This is a standard tag that is used to describe the use of a pattern instance by one of its clients.

The following code(see Figure 4), is an example of documentation of a class that plays the role of Leaf in the Composite pattern.

The above fragment of comment, when processed by the Javadoc tool enhanced with the proposed pattern specific tags produces the following documentation(see Figure 5):

The first sentence in the documentation is used as the generic class description. Then the pattern specific tags generate their output. The Composite link brings to an online description of the composite pattern. The evaluate () link, obtained through the default in-line tag link, brings to the documentation of the method evaluate within the current class [12].

Our proposal is based on a simple extension of a widely used standard; therefore it can be easily adopted. The overhead required to document pattern is very low: a few lines in addition to the usual documentation. The resulting documentation is well structured and is linked to pattern description, thus making it easy to understand the program.

3.2. Documenting Design Patterns in Code Eases Program Maintenance

Most of the people documenting patterns are motivated by the following values: The longer a pattern has been used successfully, the more valuable it tends to be, because new techniques are often untested. Finding a pattern is a matter of discovery and experience, not an invention. A new technique can be documented as a pattern, but its value is known only after it has been tested.

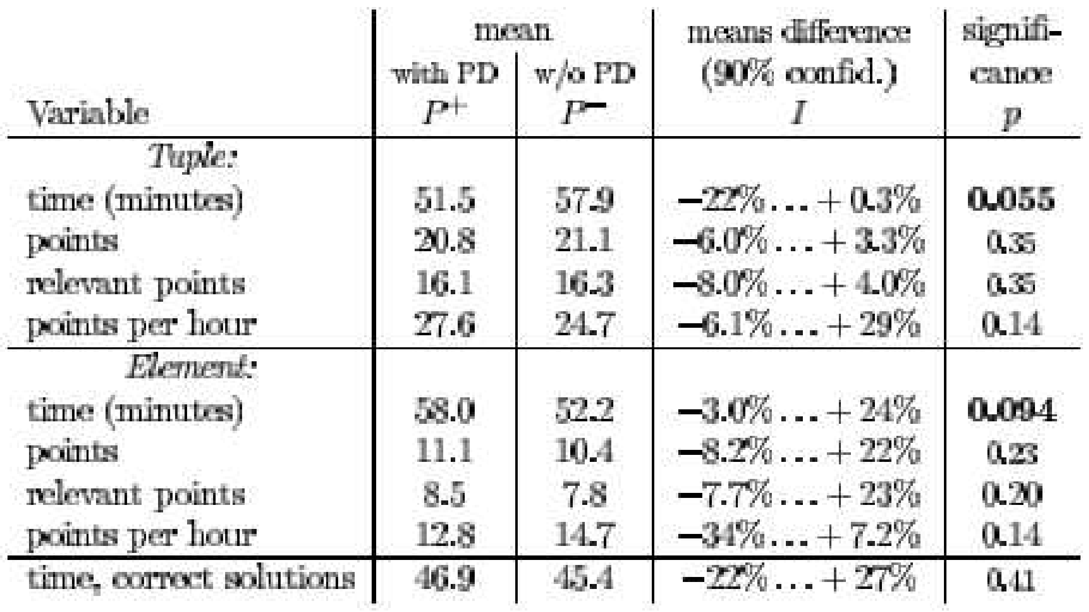
The idea of design patterns is appealing and practitioners report subjectively that design patterns simplify communication between designers by providing a concise common vocabulary, can be used to record and reuse best practices, and “capture the essential parts of a design in compact form” [7].

However, no rigorous test has yet been presented that design patterns are useful. Now, we decide to reference the experiment that investigates the benefits obtained from explicit use documentation of design patterns, which is described in [10]. You will find an extensive description and evaluation of the experiment in [10]. (See table 6)

As we see, pattern documentation helped to improve either task completion time or solution quality, depending on the kind of task and program. In a quantitative sense, these results are not dramatic. But given the conservative design of experiment [10], they indicate that documenting design patterns in software can improve program maintenance.

It is unclear how these results will scale to more experienced software engineers and to programs of industrial size and complexity, but we believe that in situations where design patterns are relevant during maintenance, one would see benefits from pattern documentation that are at least as large as in the experiment.

Therefore, from this experiment, we recommend that when design patterns are used they should be explicitly documented in the program code. In addition, we also may



Variable	mean with PD P^+	mean w/o PD P^-	means difference (90% confid.) I	signifi-cance p
Tuple:				
time (minutes)	51.5	57.9	$-22\% \ldots +0.3\%$	**0.055**
points	20.8	21.1	$-6.0\% \ldots +3.3\%$	0.35
relevant points	16.1	16.3	$-8.0\% \ldots +4.0\%$	0.35
points per hour	27.6	24.7	$-6.1\% \ldots +29\%$	0.14
Element:				
time (minutes)	58.0	52.2	$-3.0\% \ldots +24\%$	**0.094**
points	11.1	10.4	$-8.2\% \ldots +22\%$	0.23
relevant points	8.5	7.8	$-7.7\% \ldots +23\%$	0.20
points per hour	12.8	14.7	$-34\% \ldots +7.2\%$	0.14
time, correct solutions	46.9	45.4	$-22\% \ldots +27\%$	0.41

Figure 6. Table1

find other widely cited benefits gained by using patterns and document design patterns are [11]:

- Improved communication between (and within) project teams: The improvements in communication come at two levels: (1) when discussing designs, developers can speak in terms of the patterns they wish to use, rather than in terms of the specifics of the implementation; (2) because the patterns are language-independent, experience gained by one project team can potentially be reused by another in a different context. Thus, an organization that has integrated design patterns into its development strategy can expect improved communication because of a shared vocabulary and concepts.

- Better design documentation: Patterns facilitate design documentation by providing a high level of abstraction.

- Since pattern descriptions explicitly enumerate consequences, they serve to record engineering tradeoffs and design alternatives: One of the intrinsic benefits of design patterns is that they not only document good designs, but also why the designs are good and explicitly what consequences result from its implementation.

- Patterns explicitly capture knowledge that experienced developers understand implicitly: Because of patterns written by developers knowledgeable in a given area, they provide a means whereby the knowledge gained can be disseminated to others. This leads to many of the other advantages, including aiding new developers to get up to speed, and eliminating some of the loss of knowledge as team members leave the group.

- Standardized solutions to recurring design problems: Code maintenance is potentially easier on code developed using design patterns because the same strategies are used to resolve recurring problems.

- Documentation of successful software: Design patterns have the potential to accomplish much more than merely document how to design software. Because they are the result of actual experiences, they serve as documentation of successful software projects - which has been sorely missed in the traditional computer science literature. While studying patterns trains developers to design quality software, merely reading them has an equally significant benefit - providing an interesting perspective into a successful software project.

- Developing and maintaining a list of patterns: Many of the advantages discussed in the previous section require that all developers in a project team (or organization) share the same set of patterns. This entails that someone (or some group of people) be responsible for selecting which patterns to start with - and over time maintain this list by adding and removing patterns as needs change.

Therefore, the use of design patterns has the potential to deliver some significant benefits to the development environment. Design patterns are a form of high-level reuse, mainly the reuse of design. By reusing proven designs, designers need not solve every problem from first principles. They need not design all solutions from scratch; thus, it is possible to reduce time required for design. Reusing proven designs also helps to maintain characteristics of good design too.

3.3. Limitations and Pitfalls

3.3.1 Over-engineering

Commonly, most of people believe that design pattern can help them to develop flexible frameworks and build robust and extensible software systems. Hence, the system developer will make their code more flexible or sophisticated than system needs to be. As a result, they are over-engineering their work. So they waste development time and money.

Over-engineering tends to happen quietly: Many architects and programmers aren't even aware they do it. And while their organizations may discern a decline in team productivity, few know that over-engineering is playing a role in the problem [1].

We give the following example about over-engineering. Suppose that a man have a program to do is: validation on user input, checking whether the input value is numeric value. One of team member considered which patterns could help. Building a class for the handle validation came to mind, and he suggested build a class for each different type of validation. For example, user id validations have a class, validation on email address or numeric have a class. His partner's response with this suggestion: "Building a class for validation here would be like applying a big and heavy hammer when a few light taps with a small hammer would do." His solution was to create a common class to group all validation functions together. Building a common class for validation took no time to program, since it was less than 20 simple lines of code. If we build a different class for different type of validation, it would have involved creating more than 50 lines of code. Actually, two approaches can solve the problem. But, the latter approach do some repetitive code on the program, it is unnecessary to do much on this stage. Therefore, this approach is over-engineering ring the works.

Therefore, we give the following solution to over-engineering. To compensate for triggering problem after learning pattern, two methods should be carried out to avoid over-engineering: Test-first programming and merciless refactoring.

Test-first programming and merciless refactoring, which are two of the many excellent Extreme Programming(XP) practices, can dramatically improved the way people using design pattern in software development. Test-first programming is that you start with a small test, then write just enough code to implement it, and continue with the next test until the code is done. This test-first programming enabled us to make a primitive piece of behavior work correctly before evolving it to the next necessary level of sophistication. Merciless refactoring can help us to weed out inessentials, clarifying ambiguities and consolidating ideas. When we mercilessly refactor, you relentlessly poke and prod your code to remove duplication, clarify and simplify.

The above two ideas have ability to keep track what we are going on and what are the next step in software development cycle.

3.3.2 Limitation of Documenting Patterns

We found out that only a very small fraction of projects use design patterns for documenting the changes in the source code. And, design patterns are more likely to be used in projects with bigger developer teams (¿7 developers). This is due to design pattern can facilitate communication between developers.

For the number of different design patters used in a project and their position in the software life-cycle only a very small correlation was found. Therefore, in the projects in our data set we found no evidence that design patterns are widely used for refactoring. Additionally, we found evidence that developers who mainly develop new functionality are more likely to use patterns than developers who specialize in modifying existing code. Reasons for this behavior could be that the analyzed projects are still too early in their life-cycle to make major restructuring necessary. Another reason could be that open source development favors more flexible design by frequent modifications and expansion of the code and therefore does not need explicit refactoring as some custom-made systems do [5].

This first study of the application of design patterns in real-world software development projects has many limitations, e.g. it does not include additional information on the quality of the produced code, the code itself is not analyzed using object-oriented metrics and the actual effort used for the projects is unknown. Also the projects are open source projects which means that the development process differs significantly from industrial projects. But even with this limitation the quantitative results confirm the most important claim of design patterns, namely that design patterns are used to facilitate communication between developers which, without doubt, is also vital for industrial software development.

3.3.3 Fallacies and Pitfalls (Patterns are dead?)

How do you know that you have a new pattern to document? You don't unless you read through all the pattern literature to see if it's already there or not. That's onerous. Patterns are often misapplied. I don't know how many design specs I've seen now that treat them as clipart. They don't work as clipart, but people who apply them that way soon lose the desire to pursue them any further.

One of the major threats to successful usage of patterns is that like almost all hot topics, it is over hyped. Designers should not treat design patterns as ”must have” and use patterns regardless of applicability. This leads to the misuse of patterns in situations where designers do not know when to use or not to use patterns. An organization needs to have an organizational-wide design pattern program in order to enjoy the full potential of patterns. If one of the benefits of using patterns is to provide a common vocabulary for communication among designers, then all designers should be well verse with the vocabulary. In order to fulfill this benefit, all developers have to be familiar with the same collection of patterns to a certain degree. This leads to another concern: there is not yet a methodological support for organizational-wide use of patterns. Issues like documentation, training, and assessment for the development team has to be dealt with.

The methodological support is imperative in a multi-designer environment, where a uniform level of knowledge is preferred. Imagine if an organisation has fifty developers but only eight of them know patterns. The method describes processes and methods required to achieving such environment. However, there is not yet a methodological support for using patterns which also deals with patterns mining, documenting and the use of patterns. Successful organisational-wide patterns program such as those in AG Communication Systems (Goldfedder and Rising, 1996, Rising, 1996) are mainly due to very well planned approaches.

Recent rush to patterns causes increasing effort to patterns mining. One possible consequence will be the increasing number of patterns discovered. The problem that designers will face is getting familiar with design patterns and choosing which one to use. One solution is to select and use only a few common design patterns and learn more slowly over time.

Designers have to keep in mind that patterns are not design strategies. The aim of design patterns is to solve a design problem concerning a particular context. For example, ”Reduce coupling among classes” (which is a design strategy) is not a design pattern. One most important thing that all pattern users should be aware of - patterns are not the long-sought-after silver bullet. While patterns are good ways of experience reuse, they are still too high-level to support direct code reuse, a good pattern may still rendered useless when implemented wrongly. In addition, the issue of choosing the right pattern depends on the judgement of designers.

4. Conclusions

Design patterns are widely recognized as important software development methods. Patterns are most useful in understanding software when they are well documented. In this report, we first classify the design pattern and indicate the relationship between different design pattern. We also give some examples of applying design pattern in projects. Furthermore, we introduce that how we document the design pattern well. Finally, we make some investigation on the limitations and pitfalls on using and documenting design pattern.

5. Future Work

What can you do if you are interested in patterns? Use them and look for other patterns that fit the way you design. Moreover, look for patterns you use, and write them down. Make them a part of your documentation. Show them to other people. In fact, finding relevant patterns is nearly impossible if you don't have practical experience.

A body of literature on design patterns has emerged. These patterns identify, document, and catalog successful solutions to common software problems. The patterns captured by this literature have already had a significant impact the construction of commercial software. In these systems, patterns have been used to enable widespread reuse of communication software architectures, developer expertise, and object-oriented framework components.

Over the next few years, we think a wealth of software design knowledge will be captured in the form of patterns and frameworks. These patterns and frameworks will span domains and disciplines such as concurrency, distribution, organizational design, software reuse, real-time systems, business and electronic commerce, and human interface design.

Focus in the future:

- Integration of design patterns with frameworks and other design paradigms.
- Integration of design patterns to form pattern languages.
- Integration with current software development methods and software process models.

References

[1] S. Chan. Design pattern: concept and application.

[2] J. Coplien. Software design patterns: Common questions and answers. *in The Patterns Handbook: Techniques, Strategies, and Applications, L. Rising, Ed. New York: Cambridge University Press*.

[3] A. Cornils and G. Hedin. Statically checked documentation with design patterns. *in Proc. of 33rd International Conference on Technology of Object-Oriented Languages (TOOLS 33), Mont-Saint-Michel, France*, 2000.

[4] R. J. J. V. Erich Gamma, Richard Helm. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Longman, Inc., Addison Wesley Longman, One Jacob Way, Reading, MA 01867, 1995.

[5] M. Hahsler. A quantitative study of the application of design patterns in java.

[6] R. E. Johnson and V. F. Russo. Reusing object-oriented designs. *Technical Report Technical Report UIUCDCS 911696. , University of Illinois*, May 1991.

[7] R. C. L. D. G. M. F. P. K. Beck, J.O. Coplien and J. Vlissides. Industrial experience with design patterns. *In 18th Intl. Conf. on Software Engineering, Berlin, IEEE CS press*, pages 103–114, March 1996.

[8] W. F. T. P. B. e. a. Lutz Prechelt, Barbara Unger. A controlled experiment in maintenance comparing design patterns to simpler solutions. *IEEE Transactions on Software Engineering*, 2000.

[9] S. Microsystem. In one task the average time reduced from 55 to 47 javadoc tool home page. *available at http://java.sun.com/j2se/javadoc/*, 2002.

[10] L. Prechelt. An experiment on the usefulness of design patterns: Detailed description and evaluation. *Technical Report,Fakultat fur Informatik, Universitat Karlsruhe, Germany, ftp.ira.uka.de/pub/techreports*, September 1997.

[11] D. T. S. Ross A. McKegney. Patterns for pattern integration. *Queen's University, Kingston, Ontario Canada*, 2000.

[12] M. Torchiano. Documenting pattern use in java programs. *Norwegian University of Science and Technology (NTNU), Sem Slands vei 7-9, N-7491 Trondheim, Norway*.

[13] B. P. Walter Zimmer, Forschungszentrum Informatik. Relationships between design patterns. *In J. O. Coplien and D. C. Schmidt, editors, Pattern Languages of Program Design*, page 345364, 1995.

[14] R. J. Wirfs-Brock and R. E. Johnson. Surveying current research in object-oriented design. *CACM, 33(9)*, page 105123, September 1990.