Impact of Object Oriented Design Patterns on Software Development

R.Subburaj Professor, Gladman Jekese, Chiedza Hwata

Abstract —Software design patterns are a bonanza for building large Object Oriented (OO) software systems. They provide well-tested and proven solutions to recurring problems that developers address. There are several benefits of using patterns. They can speed up the software development process. Design patterns consolidate learning with an aim to make it easier for designers to use well-known and successful designs developed from expert experience. At the same time software design patterns are too abstract and remain an art that has to be mastered over time with experience. This paper seeks to evaluate the advantages and disadvantages of design patterns.

Index Terms— design patterns, analysis, selection, software development, object oriented, pattern catalog, pattern usage

1 INTRODUCTION

OBJECT oriented programming (OOP) has evolved to improve the quality of the programs. However, interesting concepts such as inheritance and polymorphism were also introduced in OOP to enhance reusability. Object oriented design patterns have been introduced in mid 1990s as a catalog of common solutions to common design problems, and are considered as standard of ''good'' software designs [1]. There are several methodologies in Object Oriented Development, of which design patterns is one of them. Due to the fact that Object Oriented Design is complex, it is of paramount importance that software developers build on the experience of others by making use of frameworks or design patterns. The notion of patterns was first introduced by Christopher Alexander, who identified and proposed solutions to common architectural problems. In his work he dealt with the question whether quality in architecture can be objective. By examining several architectural artifacts he discovered that ''good'' quality designs shared some common characteristics, or shared ''common solutions to common problems'' [2]. A design pattern represents a recurring solution to a design problem within a particular domain such as business data processing, telecommunications, graphical user interfaces, databases, and distributed communication software [3].

Design patterns facilitate architectural level reuse by providing "blueprints" that guide the definition, composition, and evaluation of key components in a software system.

- *Dr. R Subburaj is a Professor and Consultant in the Department of Information Technology at SRM University, Chennai., India,. E-mail: subburaj.spr@gmail.com*
- *Gladman Jekese is an M.Tech student in Information Technology at SRM University, Chennai, India, E-mail: jgman86@gmail.com*
- *Chiedza Hwata is an M.Tech student in Information Technology at SRM University, Chennai, India, E-mail: chiedza11@gmail.com*

In general, a large amount of experience reuse is possible at the architectural level. However, reusing design patterns does not necessarily result in direct reuse of algorithms, detailed designs, interfaces, or implementations [4], [5]. It systematically explains a general design that addresses a recurring design problem in object-oriented systems. Design patterns also describe the problem, the solution, when to apply it, plus the consequences associated with that specific solution. It also gives implementation hints and examples as well as a description or template for how to solve a problem that can be used in many different situations. The solution is customized and implemented to solve the problem in a particular context. In the Gang of Four (GoF) book, patterns typically have these major elements: Intent, Motivation, and Applicability, Structure, Participants, Collaborations, Consequences, Implementation, Sample Code, Known Uses and Related Patterns [6]. Although the terminology may differ with the author, all Object oriented design patterns are written following the aforementioned factors.

This paper is categorized into eight sections; with section 2 representing the related work outline, section 3 explaining basic classes of the design pattern catalog, section 4 design pattern selection, section 5 outlines how to use design patterns, section 6 describing the advantages and disadvantages of design patterns in 7, section 8 concluding the paper.

2 RELATED WORK

Object oriented design pattern is an active area of research in software development. Ackerman L and Gonzalez C explore the benefits of implementing patterns in software designs [7]. The article introduces developers and architects to the idea of a pattern implementation which is an artifact that allows the codification of a pattern specification for a specific environment and can be created and used for different phases in the software development lifecycle [7]. Rising. L analyzes an example from a small development team and discovers that a novice pattern, called the Mediator, is a perfect fit for the design challenges that they had just spent hours battling [8].

In [9], an analysis was done to verify the reusability of design patterns and software packages, which uncovered some advantages and disadvantages of design patterns. Three examples employing design patterns were developed, accompanied by alternative designs that solve the same problem, plus a tool to calculate the scores of extensibility was used and it suggested that in most scenarios the patterns provide a solution that is most extensible although the conclusion cannot be generalized for all patterns in [10]. A team at Erricson developed frameworks based on design patterns for two years, according to [4], [11]. The history of design patterns and some developments characterizing the advantages and disadvantages of design patterns are presented in [12]. Appostollos 2006 evaluates usage of object oriented design patterns in game development, proving maintainability although the research was biased towards game development only [13].

Yacoub, S. A., Ammar, H. H., & Mili A., teaches developers about the abstraction benefits of design patterns. The paper explains the benefits that include understanding the dependencies and collaboration between participating patterns while hiding implementation details. Douglas C. Schmidt and Paul Stephenson present a case study illustrating the implementation of design patterns (Reactor and Acceptor) in object-oriented telecommunication software framework across UNIX and Windows NT OS platforms and discuss the techniques, benefits, and limitations of applying a design pattern-based reuse strategy to commercial telecommunication software systems [4].

Fowler, M. (2003), explains the value of patterns as teaching tools. Written patterns help educate other team members for building and reviewing software and some developers declare patterns as good or bad instead of appropriate or not [14]. T Cline, M.P. (1996), focuses on the advantages and disadvantages of adopting and applying design patterns, which are a valuable tool for practicing software professionals. The paper presents the practical benefits of design patterns as well as inhibitors to pattern applications, which proves is beneficial to IT professionals who want to learn more or desire to promote the use of design patterns [15].

3 DESIGN PATTERN CATALOG

Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for developers and architects familiar with the patterns. It has taken years to establish common vocabulary on patterns. One of the drawbacks to design patterns is that the same names are used for different patterns. For example the Value Object pattern (Fowler, Value Object, 2002) described as a pattern for storing mutable data is preferred by Fowler for static and immutable data while he uses the Data Transfer Object pattern(Fowler, Data Transfer Object, 2002) for the former purpose [1], [16]. Design patterns can be grouped according to their usage. There are 3 basic classes of design patterns;

Structural design patterns

This group of design patterns eases software design by identifying a simple way to realize relationships between entities. Such patterns are all about Class and Object composition. Structural class-creation patterns use inheritance to compose interfaces while structural object-patterns define ways to compose objects to obtain new functionality. Examples of structural design patterns include adapter, decorator, bridge, composite, flyweight, façade and proxy.

Creational design patterns

These design patterns are basically concerned with class instantiation and are composed of two dominant ideas. One is encapsulating knowledge about which concrete classes the system uses and the other hides how instances of these concrete classes are created and combined [1]. Creational design patterns are further categorized into Object-creational patterns and Class-creational patterns, where Object-creational patterns deal with Object creation and Class-creational patterns deal with Class-instantiation. In greater details, Object-creational patterns defer part of its object creation to another object, while Class-creational patterns defer its object creation to subclasses Gang of Four [17]. The examples of creational design patterns are; abstract factory, builder, factory method, singleton and prototype.

Behavioral design patterns

Behavioral design patterns identify common communication patterns between objects and realize the assignment of responsibilities between objects. By doing so, these patterns increase flexibility in carrying out this communication because they shift focus away from flow of control and concentrate just on the way objects are interconnected. Observer, chain of responsibility, interpreter, iterator, memento, template method, command, visitor, strategy and mediator are some of the examples of behavioral design patterns.

4 HOW TO SELECT A DESIGN PATTERN

With many design patterns to choose from; it may be difficult to find one that addresses a particular problem, especially if the list is new and unfamiliar. The problem of searching patterns is defined as the effort of getting information about existing patterns; whereas pattern selection is described as the problem of deciding which pattern to choose among all available solutions. Below is an outline of approaches that can be used to find the design pattern that suits a particular problem.

According to the GoF book, there are five main approaches for searching and selecting design patterns namely; pattern repositories and pattern catalogues, recommendation systems, formal languages, search engines and other approaches [10]. In [18], [19], [20], [21], the authors create online pattern repositories in order to increase the availability of patterns. In such repositories patterns can be retrieved through searching criteria and by manual browsing among various patterns. Furthermore, in [22], [23] several recommendation systems are suggested in order to suggest the appropriate pattern,

according to the problem that the developer wants to solve. There are also several papers that describe approaches that use formal languages in order to represent design patterns and select patterns according to such a representation [24], [25] [26]. Selecting patterns through search engines corresponds to searches through keywords in engines that crawl and index pattern descriptions. Finally, there are several other approaches that cannot be classified in any of the above categories such as [6], [27].

According to [28], ways have been proposed that enable software developers to find the adequate design pattern(s) for a given design problem. It remains, thus, a measure avoiding the impact of the vast amount of design patterns present in different design pattern catalogues.

Consider how design patterns solve design problems: According to the GoF textbook discussion, design patterns help find appropriate objects, determine object granularity, specify object interfaces, and several other ways in which design patterns solve design problems.

Scan Intent sections from all the patterns in the catalog: Each pattern's intent is to be considered to find one or more that sound relevant to the current problem. A classification scheme can be used to narrow the search.

Study how patterns interrelate: Considering and studying relationships between design patterns can help direct developers to the right pattern or group of patterns.

Study patterns of like purpose: Each catalog classifies design patterns as creational, structural, or behavioral patterns. Each will include introductory comments on the patterns and concludes with a section that compares and contrasts them. These sections give insight into similarities and differences between patterns of like purpose.

Examine causes of redesign: Examining the causes of redesign in most designs to see if the current problem involves one or more of such and then looking at the patterns that help in avoiding the causes of redesign.

Consider what should be variable in the design: This approach is the opposite of focusing on the causes of redesign. Instead of considering what might force a change to a design, consideration is to be done, so as to realize what has to be changed without redesign. The focus is on encapsulating the concept that diverges from the themes of many design patterns. Consider aspects in which design patterns allow independent variance thereby letting the designer change them without redesign.

5 HOW TO USE DESIGN PATTERNS

Once a design pattern has been selected, the question is now on the approach that is to be used to apply the design pattern effectively. The following are the steps that can be employed in the use of a design pattern:

The design pattern has to be read through at least once for an overview. While reading, particular attention should be paid to the applicability and consequences sections to ensure that the pattern is right for that specific problem. The structure, participants and collaborations sections must be revisited and an understanding of how the classes and objects in the pattern relate to one another needs to be achieved. Studying the code helps in learning how to implement the pattern, and this is done by looking at the sample code section to find concrete examples of the design pattern in the code.

Choice of names for pattern participants is very important because they ought to be meaningful in the application context. They are usually too abstract to appear directly in an application. Nevertheless, it's useful to incorporate the participant name into the name that appears in the application, as this helps make the pattern more explicit during the implementation. For example, usage of the Strategy pattern for a text composition algorithm might mean having classes such as Simple-Layout-Strategy or Text-Layout-Strategy [1]. The next stage involves defining the classes, declaring their interfaces, establishing their inheritance relationships, and defining the instance variables that represent data and object references. Identifying existing classes in the application that the pattern will affect and modifying them accordingly can also be included.

Defining application-specific names for operations in the pattern generally depends on the application. Responsibilities and collaborations associated with each operation should be used as a guide. Consistency in the naming conventions is important, for example, using the "Create-" prefix consistently denotes a factory method. Implementing the operations is now done to carry out the responsibilities and collaborations in the pattern. The Implementation section offers hints to guidance in the implementation. The examples in the Sample Code section may be of help as well [1].

6 ADVANTAGES OF DESIGN PATTERNS

The following are some of the benefits of OO design patterns.

They can reduce development time as known solutions are used instead of reinventing the wheel thereby improving delivery speed.

Design patterns promote discovery and learning with an aim to make it easier for designers to use well-known and successful designs developed from expert experience (Chang, 2011) [5]. According to Rising (2010) [8], there is a debate about the merits of providing developers with formal training versus simply having access to a large searchable repository within which to search for a pattern to address some sticky problem. According to Bleistein (2003), engineers find themselves under increasing pressure to deliver solutions with a high degree of quality in a timely manner [29]. As the Project Triangle (2011) describes, project management can only focus on two of the following three facets of a development effort: (a) speed, (b) quality, and (c) cost [30]. 'Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use the solution a million times over, without ever doing it the same way twice' [31].

Design patterns provide flexibility and extensibility.

Tichy (1998) describes the concept of flexibility as how design patterns can improve the structure of software, speed up implementation, simplify maintenance, and help avoid

architectural drift. Future changes (extensibility) consistent with one of these hinges are relatively inexpensive, but forcing the software to change in other ways is like bending your elbow backwards; the system normally breaks [32].It is also beneficial to use known solutions that are tried and tested.

Patterns make the communication of development teams easier.

Because design patterns capture distilled experience, they can provide a communication tool throughout the software development lifecycle and across diverse communities of designers and programmers (Cline, 1996) [15]. This improved communication among software developers is a benefit that can empower less experienced developers to produce high-quality designs. According to Fowler (2003) [33], an expert on a team can use written design patterns to help educate other team members as they work through software requirement, design, and review.

Patterns are underspecified

Design patterns are underspecified since they generally do not over constrain implementations. This is beneficial since patterns permit flexible solutions that are customizable to account for application requirements and constraints imposed by the application development. Because they are underspecified, implementing a pattern on your own for a particular purpose is one of the best ways to learn about it [4].

Design patterns capture knowledge that is implicitly understood

Once developers are exposed to, and properly motivated by, design patterns they are eager to adopt the pattern nomenclature and concepts. This stems from the fact that patterns codify knowledge that is already understood intuitively. Therefore, once basic concepts, notations, and pattern template formats are mastered, it is straightforward to document and reason about many portions of a system's architecture and design using patterns [34].

They promote a structured means of documenting software architectures

This is done by capturing the structure and collaboration of participants in a software architecture at a higher level than source code. This abstraction process is beneficial since it captures the essential architectural interactions while suppressing unnecessary details [4]. 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 in the choice of 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. Put simply, design patterns help a designer get a design "right" faster [1]. Patterns improve the documentation and maintenance of existing systems by furnishing an explicit specification of class and object interactions and their underlying intent.

Patterns are a known solution for building software systems, and provide solutions to recurring problems

Developers employ design patterns because they make use of tested solutions. According to Bansyia et al; "Software reusability reflects the presence of object-oriented characteristics that allow a system to be reapplied to a new problem without significant effort", (Bansiya and Davis, 2002) [35]. Design patterns capture expertise and make it accessible to non-experts in a standard form. Therefore patterns help in identifying common solutions to recurring problems. The solution is really what the pattern is, yet the problem is a vital part of the context. A pattern cannot be fully understood without understanding the problem, and the problem is essential to helping people find a pattern when they need it [15].

Design patterns can be used to provide a software hinge or adaptability point

Design patterns constrain maintenance programmers by reducing the chance of breaking a design pattern's adaptability point (or software hinge). Object-oriented design patterns specify the relationships between the participating classes and determine their collaboration. Such solutions are especially geared to improve adaptability, by modifying the initial design in order to ease future changes [1]. Each pattern allows some aspect of the system structure to change independently of other aspects.

They make it easier to reuse successful designs and avoid alternatives that diminish reusability

Design patterns capture the static and dynamic structures of solutions that occur repeatedly when developing applications in a particular context [1], [17], [36]. Systematically incorporating design patterns into the software development process helps improve software quality since patterns address a fundamental challenge in large-scale software development: communication of architectural knowledge among developers. Patterns can also be used in software architecture and, if applied properly, increase flexibility and reusability of the underlying system. Each pattern allows some aspect of the system structure to change independently of other aspects [13].

Improved IT process and communication

Design patterns coordinate the entire process and community through a common vocabulary. They assist in improving software communication quality since they address a fundamental challenge in large-scale software development; communication of architectural knowledge among developers [4].

Constrain maintenance programmers

Require maintenance programmers to understand and preserve the integrity of the design patterns during maintenance changes. This therefore ensures preservation of the credibility of the quality of design patterns.

Design patterns can be used reactively and proactively through fragmenting and abstraction of design.

Design patterns can be used reactively as a documentation tool to classify the fragments of a design and proactively to build robust designs with design-level parts that have well understood trade-offs [15].

Design patterns can turn a trade-off into a win-win situation by allowing multiple facets of quality that are often viewed as mutually exclusive.

The notion of patterns was first introduced by Christopher Alexander, who identified and proposed solutions to common

architectural problems. In his work he dealt with the question whether quality in architecture can be objective. By examining several architectural artifacts he discovered that ''good'' quality designs shared some common characteristics, or shared ''common solutions to common problems'' [2]. Adaptability must be explicitly designed into the software in designated places.

7 DISADVANTAGES OF DESIGN PATTERNS

The following provides the views of critics;

Remains an art which can only be mastered after using it for many years

Developers can still make a mistake of employing a design pattern where it is not necessary, for example given a subsystem does not necessarily qualify the usage of a proxy pattern. It is dependent on experience of usage by the developers. According to Manolescu (2007), adoption rates are still low for IT organizations due to lack of discovery and limited education around how to apply design patterns to specific domain contexts. A survey from Manolescu (2007) indicates that only half of the developers and architects in IT organizations use patterns. Additionally, ninety percent of respondents that claim to be pattern practitioners hadn't taken any educational courses on design patterns. Manolescu attributes this low adoption rate to the fact that finding patterns relevant to a particular problem isn't trivial and because the design pattern world doesn't have an authoritative index [16]. This challenge is, in part, due to the nature of how patterns often match a problem domain and each domain needs a distinct approach (Bleistein, 2003) [29].

Over engineering and under-engineering

Using patterns and languages of patterns to generate architectures may lead to over-engineering the design of a program. Over-engineering happens when a design or a piece of code is more flexible or sophisticated than 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. 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. A solution to both under- and over-engineering is to apply refactoring prior to modifying the design or the code of the program. The refactoring step is necessary to clean up the program and makes it ready for change, at the right time and with the right amount of work. Refactoring to or 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 [12].

They target the wrong problem sometimes

The need for patterns results from using computer languages or techniques with insufficient abstraction ability [37]. Under ideal factoring, a concept should not be copied, but merely referenced. But if something is referenced instead of copied, then there is no "pattern" to label and catalog. Conway's law suggests that "organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations" (Conway, 1968) [38]. Although Conway's law was not verified at the time of its publishing it was heavily cited for decades and recent studies from Harvard Business School (McCormack et al., 2008) and Microsoft research (Agape et al., 2008) confirm it [39].

Use of design patterns does not necessarily improve quality

Many studies in the literature are based on the premise that design patterns improve the quality of object-oriented programs. Yet, some studies suggest that the use of design patterns does not always result in "goo" designs. For example, a tangled implementation of patterns impacts negatively the quality that these patterns claim to improve. 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, expandability and understandability as put forward in when applying design patterns. Also, the impact of design patterns on other quality attributes is unclear.

Developers and managers must recognize that learning a collection of patterns is no substitute for design and implementation skills

In fact, patterns often lead team members to think they know more about the solution to a problem than they actually do. For example, recognizing the structure and participants in a pattern (such as the Reactor or Acceptor patterns)

Design patterns may increase complexity

Design pattern flexibility often comes at a price of complexity as dynamic, highly parameterized software is harder to understand and document (Wendorff, 2001). Patterns are said to be half-baked, that means one always have to finish the development and adapt it to their own environment. The research indicates that the poor judgments of individual software developers often add complexity without benefit which results in designs that become difficult to alter or remove [40].

Conventional pattern catalogs are too abstract

One of the main concerns with conventional pattern catalogs [1], [17], however, is that they are too abstract. According to research, overly abstract pattern descriptions make it difficult for developers to understand and apply a particular pattern to systems they were building in most cases.

Use of inappropriate design patterns

As usual, however, restraint and a good sense of aesthetics are required to resist the temptation of elevating complex concepts and principles to the level of hyperbole. There is a tendency for some developers to adopt a tunnel vision where they would try to apply patterns that were inappropriate, simply because they were familiar with the patterns. For

example, the Reactor pattern may be an inefficient event de-multiplexing model for a multi-processor platform since it serializes application concurrency at a fairly coarse-grained level [33].

Pattern overload

A drawback to the intuitive nature of patterns is a phenomenon we termed pattern overload. In this situation, so many aspects of the project are expressed as patterns that the concept becomes diluted. This situation occurs when existing development practices are relabeled as patterns without significantly improving them. Likewise, developers may spend their time recasting mundane concepts (such as binary search or building a linked list) into pattern form. Although this is intellectually satisfying, it becomes counter-productive if it does not lead to software quality improvements [41].

Expectation management

Many of the problems with patterns discussed above are related to managing the expectations of development team members. As usual, patterns are no silver bullet that will magically absolve developers from having to wrestle with tough design and implementation issues.

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 (Quality Without A Name). 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 [41].

8 Conclusion and Future Work

From the analysis carried out based on the previous published work, a conclusion is reached that design patterns are not a panacea. In as much as there are advantages, they also tend to pose a lot of disadvantages if not applied correctly. In this paper a brief overview of OO design patterns is given with catalog of design patterns, how to select and use them and their advantages and disadvantages in OO analysis and design. Much work is still to be done on design patterns to make them understandable to new users because currently usage depends on the developer's expertise.

References

1. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, 'Design Patterns: Elements of Reusable Object-Oriented Software', Addison-Wesley, 1995.
2. C. Alexander, S. Ishikawa and M. Silverstein, 'A Pattern Language – Town, Buildings, Construction', Oxford University Press, New York, 1977.
3. E. Gamma, R. Helms, R. Johnson and J. Vlissides, 'Design Patterns: Elements of Reusable Object-Oriented Software', Addison-Wesley Professional, MA, 1994.
4. Douglas C. Schmidt and Paul Stephenson, 'Experience using Design Patterns to Evolve Communication Software across Diverse OS Platforms', Proceedings of the 9th European Conference on Object-Oriented Programming, 1995.
5. C H Chang, C W Lu and P A Hsiung, 'Pattern-based framework for modularized software development and evolution robustness', Information & Software Technology, 2011.
6. D.C. Kung, H. Bhambhani, R. Shah and G. Pancholi, 'An expert system for suggesting design patterns: a methodology and a prototype', Series in Engineering and Computer Science: Software Engineering with Computational Intelligence, Kluwer International, 2003.
7. Ackerman. L and Gonzalez. C, 'The value of pattern implementations', The World of Software Development Journal, Computer Science Vol.32 Issue. 6, pp. 28-32, 2011.
8. L. Rising, 'The benefits of patterns', IEEE software, Vol. 27 Issue. 5, 2011.
9. Apostolos Ampatzoglou, Apostolos Kritikos, George Kakarontzas and Ioannis Stamelos, 'An Empirical Investigation on the Reusability of Design Patterns and Software Packages', Department of Informatics, Journal of Systems and Software, Vol. 84, 2011.
10. Apostolos Ampatzoglou, Georgia Frantzeskou, Ioannis Stamelos, 'A Methodology to Assess the Impact of Design Patterns on Software Quality', Department of Informatics, Information and Communication Systems Engineering, University of the Aegean, Samos, Greece, 2012.
11. D. C. Schmidt and T. Suda, 'An Object-Oriented Frameworkfor Dynamically Configuring Extensible Distributed Communication Systems', IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems), Vol. 2, pp. 280–293, 1994.
12. Yann-GaÄel Gueheneuc Professor, 'Empirical Studies on the Impact of Design Patterns on Quality', Departement de genie informatique et genie logiciel Ecole Polytechnique de Montreal, Quebec, Canada, 2010.
13. Apostolos Ampatzoglou and Alexander Chatzigeorgiou, 'Evaluation of object-oriented design patterns in game development', Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece, 2006.
14. Fowler. M, 'Patterns', IEEE software, 20(2).Retrieved May 8, 2011 from http://www.se.rit.edu/~se362/Misc/FowlerOnPatterns-IEEESoftware-Mar-2003.
15. Cline M. P, 'The pros and cons of adopting and applying design patterns in the real world', Communications of the ACM, 2011.
16. D Manolescu, W Kozaczynski, A Miller and J Hogg, 'The growing divide in the patterns world', IEEE software, Vol. 24 Issue. 4, 2011.
17. F. Buschmann, R. Meunier, H. Rohnert and M. Stal, 'Pattern-Oriented Software Architecture - A Pattern System', Wileys and Sons, 1995.
18. M. Weiss and A. Birukou, 'Building a pattern repository: Benefiting from the open, lightweight, and participative nature of wikis' International Symposium on Wikis (WikiSym), 2007.
19. L. Welicki, J.M.C. Lovelle and L.J. Aguilar, 'Patterns meta-specification and cataloging: Towards a more dynamic patterns life cycle' International Workshop on Software Patterns, IEEE, 2007.
20. J. Deng, E. Kemp and E.G. Todd, 'Managing UI pattern collections' Proceedings of the 6th ACM SIGCHI New Zealand Chapter's International Conference on Computer-Human Interaction: Making CHI Natural (CHINZ '05), ACM, pp. 31–38.
21. L. Rising, 'The Pattern Almanac', Addison-Wesley Longman Publishing, 2000.
22. P. Gomes, F.C. Pereira, P. Paiva, N. Seco, P. Carreiro, J.L. Ferreira and C. Bento, 'Using CBR for automation of software design patterns', 6th European

Conference on Advances in Case-Based Reasoning, Springer, pp. 534–548.

23. G. Shu-Hang, L. Yu-Qing, J. Mao-Zhong, G. Jing and L. Hong-Juan, 'A requirement analysis pattern selection method for e-business project situation', IEEE International Conference on E-Business Engineering (ICEBE'07), IEEE, pp. 347–350, 2007.
24. U. Zdun, 'Systematic pattern selection using pattern language grammars and design space analysis', Software: Practice & Experience 37 pp. 983–1016, 2007.
25. M. Weiss and H. Mouratidis, 'Selecting security patterns that fulfill security requirements', 16th International Conference on Requirements Engineering (RE'08), IEEE, pp. 169–172, 2008.
26. H. Albin-Amiot, P. Cointe, Y.-G. Gueheneuc and N. Jussien, 'Instantiating and detecting design patterns: putting bits and pieces together', International Conference on Automated Software Engineering (ASE' 01), ACM, pp. 166–173, 2001.
27. O. Zimmermann, U. Zdun, T. Gschwind and F. Leymann, 'Combining pattern languages and reusable architectural decision models into a comprehensive and comprehensible design method', Proceedings of the 7th Working IEEE/IFIP Conference on Software Architecture (WICSA 2008), IEEE, 2008.
28. Zakaria Moudama and Noureddine Chenfoura, 'Design Pattern Support System: Help Making Decision in the Choice of Appropriate Pattern', Computer Science Department Faculty of Sciences Dhar Mehraz, University Sidi Mohammed Ben Abdellah,Fez, Morocco, 2012.
29. Bleistein, S. J, Aurum, A., Cox, K., and Ray, P. K, 'Linking requirements goal modelling techniques to strategic e-business patterns and best practices', AWRE, 3., 2011.
30. Project triangle. Wikipedia (2011). Retrieved May 1,2011,from,http://en.wikipedia.org/wiki/Project_triangle.
31. Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl and King Shlomo Angel, 'A Pattern Language: Towns, Buildings, Construction', 1977.
32. W. Tichy, 'A catalogue of general-purpose software design patterns', UIUC Pattern Group V, 2011.
33. Martin Fowler, 'Thoughts Design Patterns', IEEE Computer Society, pp., 2003.
34. G. Goos, J. Hartmains and J.van Leeuwen, 'ECOOP, Object-oriented Programming: 9th European Conference', Vol. 9, pp. 421, 1995.
35. Bansiya J and Davis C, 'A hierarchical model for object-oriented design quality assessment', Transaction on Software Engineering, IEEE Computer Society 28, Vol. 1, 2002.
36. Zakaria Moudama and Noureddine Chenfour, 'Design Pattern Support System: Help Making Decision in the Choice of Appropriate Pattern', Published by Elsevier Ltd, 2011.
37. http://www.paulgraham.com/icad.html... Revenge of the nerds. Essay, Paul Graham.
38. Melvin E. Conway and M Conway, 'How do Committees Invent Datamation', Vol. 14, pp28-31, 1968.
39. Apostolos Ampatzoglou, Apostolos Kritikos, George Kakarontzas and Ioannis Stamelos, 'An Empirical Investigation on the Reusability of Design Patterns and Software Packages', Department of Informatics, Journal of Systems and Software, Vol. 84, Greece, 2011.
40. P Wendorff, 'Assessment of design patterns during software reengineering: Lessons learned from a large commercial project', IEEE Computer Society, 2001.
41. Nobukazu Yoshioka, Hironori Washizaki and Katsuhisa Maruyama, 'A suvery on security patterns', Department of Computer Science, Ritsumeikan University, National Institute of Informatics, 2008.