Evolution of Design Patterns: Their Complexity and Usability

Hilay Khatri and Vivek Dodeja
Department of Computer Science
North Carolina State University
{hmkhatri, vdodeja}@ncsu.edu

Abstract

This paper investigates evolution of design patterns within the context of software complexity and usability. Companies often want to expedite their software development efforts in order to earn profits quickly and make up for the expenses made during development. In order to achieve these goals, they need to have flexible code, which is reusable and can be easily adapted to satisfy changing requirements. Implementing code with design patterns can help achieve such flexibility. As the software industry continues to evolve, new and unique ways of applying design patterns are emerging. In this paper, we discuss the importance of design patterns, its applications and the problems that it faces. Additionally, we explain how the extra efforts spent in deciding which design patterns to follow affect the overall development of project. Also, there are many automatic design pattern recognition tools available in market. In this paper, we also discuss their helpfulness and usability.

1. Introduction

Design patterns are "a particular prose form of recording design information such that designs which have worked well in the past can be applied again in similar situations in the future" [29]. Software developers with their experience have solved many design problems in their journey to the solutions. Their experience has been used to create a solution that can be implemented repeatedly even in other problem domains. These solutions, containing the intent and reasoning behind the design of solution without implementation details, were termed as design patterns. Christopher Alexander [29] proposed patterns in traditional architecture which were applied and used initially by Ward Cunningham and Kent Beck in 1987 while developing user interface for SmallTalk [28]. Later, design patterns became famous with Gang of Four (GoF) [15] works, where design patterns were proposed for use in design of object-oriented languages.

Design patterns have gained significant popularity in software engineering field because they: (1) provide an efficient way of communicating design problem and its solution , thus providing common vocabulary between developers which helps them to discuss about system design on abstract level [15, 19]; (2) have emerged from experienced developers and thus their experience can be reused to create and apply to complex design system thereby facilitating reusability [15]; (3) allow beginners to make use of practices of experienced developers thereby fostering sound software engineering and preventing them from implementing wrong solution [15]; and (4) are compact way of communicating design information motivating use of "best practices" [19].

Good design patterns are time-consuming to write, as it takes much prior experience and knowledge to reach to the conclusion of formulating a design pattern [19]. But are design patterns efficient for projects that have to be completed in short span of time considering the time overhead? Moreover design patterns are just a standard guidance of how to solve a particular problem and do not give any specification about how to implement the solution. In this paper we present a survey about how usable design patterns are and what complexity they add upon developers while implementing it. Over the years, many tools [1, 2, 6] and plugins [24] had been developed to simplify and automate design patterns recognition, but whether the added complexity in using these tools compensates the time and money spent? While surveying about evolution of design patterns, our paper focuses on these questions as well. In this paper, we evaluate design patterns to understand where they are leading to and what current progress is being made considering the programming environment has changed drastically over the last few years.

This paper is structured as follows: In Section 2, we discuss about the motivation and importance of design patterns. In Section 3, we discuss about application of design patterns in different domains. In Section 4, we evaluate design patterns, how they affect different phases of software development cycle by analyzing their merits and demerits. In Section 5, we discuss about design pattern detection tools. In Section 6, we discuss limitation of this paper, in context of topic covered for survey and papers referred. In Section 7, we discuss about possible future research work on design patterns. Finally, in

Section 8, we conclude by summarizing the contents of the paper.

2. Motivation

Industries have gained interest in design patterns as programmers and engineers try to reuse their past successful experience repeatedly. Design patterns are increasingly used in multiple types of software systems [5]. Design patterns are quite popular and are becoming integral part of framework and languages like Dependency Injection and MVC pattern in spring framework of Java [3]. They complement existing methods and processes of software design [18]. In recent years, design pattern had also been used in fields like HCI (Human Computer Interaction) and E-Learning [8].

2.1. Why study design patterns?

Design patterns provide an abstract solution to some general problem. These problems may be seen differently in different domain but have same characteristics. Design structure levitates the quality of product being developed. A developer having knowledge of design pattern can readily apply solution to the problems without spending time on re-discovering them. [15]. Design patterns also help in identifying interaction between classes and their objects which helps in maintenance of existing systems and can help in learning class library quickly [15]. A prior knowledge of design pattern can help identify how class library is coded and using that understanding if other classes have been implemented a similar design pattern, it would be easier to apply that knowledge thereby reducing the time required to learn class library.

Each design pattern has its own characteristic as they solve different problems distinctly. We describe briefly few design patterns and their helpfulness in solving a typical problems trait. *The Factory Method Pattern* is used in situation where many subclasses are to be created and the implementation of the method depends on the subclass in which it is implemented. Factory method therefore provides an excellent way to instantiate members in base classes from objects created from its subclasses [32]. *The Observer Pattern* works similar to publish-subscribe type exchange. It is generally used in systems which have objects and one or many observers. Observers react to the change in data of objects thereby controlling each observer individually in a way as desired [32]. *The Singleton Pattern* is useful in situation in which one needs to have only one instance of a class [32]. For example, it can be used in implementing website hit-counter. *The Decorator Pattern* can be used to augment functionality of the classes [32]. They are different from inheritance since the concept of inheritance leads to extension in hierarchical manner [32]. However, decorator pattern is useful when the extension depends on runtime and more than one extension can be active [32].

Design patterns help in simplifying what is to be done and how it should be done without considering the details of its implementation [19]. As design patterns provides a common vocabulary to communicate between people, the names used to describe design pattern also provide information of how the structure of the code is organized [19]. Design pattern helps in building a good project but it should not be used as a tool to revive broken projects and if a project is on the verge of failing, applying design patterns wouldn't be beneficial at all [19].

Design patterns are usually associated with object oriented languages but emergence of design patterns has nothing to do with them [19]. Design patterns were used in many companies. Initially, Motorola used design patterns, but implicitly, and implemented it for fault management in telephone system [19].

Patterns ...	FCS	AT&T	Motorola	BNR	Siemens	IBM
are a good communications medium	√	√	√	√	√	√
are extracted from working designs	√	√	√	√	√	√
capture design essentials	√	√	√	√	√	√
enable sharing of "best practices"	√		√	√	√	√
are not necessarily object-oriented		√	√	√	√	
should be introduced through mentoring	√				√	√
are difficult/time-consuming to write			√	√	√	
require practice to write					√	√

Table 1: Companies and their Experience [19]

Table 1 shows a survey regarding the usefulness and complexity of design patterns, as used by some well-known companies like FCS, AT&T, Motorola, BNR, Siemens and IBM. From the Table 1, it can be concluded that companies agree with the facts that patterns: (1) provide a good medium for communication; (2) are obtained from designs that were successfully working and; (3) enables quality product development. Also, most of the companies believe that patterns may not be object oriented. However, a few companies do agree with the fact that writing design pattern is a tough task, requires experience and are time consuming.

When it comes to bridging the communication gap, design patterns did an excellent job to create a standard vocabulary which could be easily understood by senior architect and product developers [19]. As a result, there are less chances of misinterpreting the context of discussion. Design patterns can be very useful when defining a new architecture for a system. For e.g. design patterns like Observer, Strategy and Composite are very good at capturing changes that might happen as the software evolves [19].

It was found that writing a design pattern is not easy [19]. It should be represented in such a way that it is easy to understand and apply. In order to ensure easy understandability, modified design patterns are available

to ensure optimum quality [19]. Problems which are high in value should be worked upon by design patterns but this raises an issue in selecting these high-valued problems [19].

3. Applications of Design Patterns

Though design patterns were very well defined, they are hard to implement and select [19]. Therefore, there arises a need to define a way to select suitable design patterns for problem. As design patterns were used implicitly, it was only known by the senior level software developers [19]. Many authors have defined design patterns as the code structure they found in their company code [19]. They were able to deduce patterns from legacy code and tried to find where the pattern could be implemented in other aspects of system thereby implementing concept of re-usability [19].

Design patterns were initially described as a successful solution to design problem without being specific to domain [15]. This made the application of design pattern an industry wide solution. Now-a-days, design patterns are used in different fields to build Scientific Systems, Web Applications, Communication Systems and Real Time Systems [11, 18, 19].

3.1 Application in Scientific Systems

Inherently scientific programs are complex because they have a large number of lines of code and usually have a complex algorithm. It is beneficial to make such complex implementation reusable with other system for further enhancement, especially for physics simulation, which requires lots of complex computation [11]. In physics simulation application, scientists create model of real world objects having physical properties like mass, density, volume called physical models. These physics model are thereafter used in simulation. Simulation uses control logic which defines actions to be performed on physics model. Design pattern can help in managing complexity by separating physics models and control logic.

Strategy pattern helps in decoupling the system behavior with algorithms [15]. This reduces complexity per class for programming and can be used in scientific applications. However, not all the design patterns used in other systems can be used with scientific systems. Scientific systems are resource intensive, which may not always allow usage of patterns defined in Gang of Four [15]. There is very less work done in design patterns specific to scientific computing [11]. Design patterns like Semi-Discrete pattern, Strategy-and-Surrogate pattern and Template Class pattern were discovered to solve design problem for physics modeling [11].

Object oriented programming has gained popularity in industry, since it supports useful features like abstraction, polymorphism, and encapsulation. However, scientific systems tend to use older programming languages also like C, C++, Fortran 90/95 [11]. Programmers have developed a way to avoid complexity in single class by using the concept of abstraction which was not present in older languages like C and Fortran 90/95. Most design patterns defined by GoF [15] used these techniques of polymorphism and encapsulation. Design patterns requiring object oriented support can also be implemented even in procedural language, which was demonstrated using Fortran 90/95 [11]. Moreover, not all object oriented languages provide same set of features; but with modifications most of design patterns can be implemented.

Implementing scientific systems using design patterns have few potential drawbacks. Design patterns simplify implementation of solution by reducing systems complexity and adding flows between modules and abstraction [11]. One possible drawback is that compiler might not be able to optimize overall system performance, because (1) design patterns may not be implemented correctly and; (2) higher level of abstraction in different modules [11]. Incorrect implementation of design pattern can also lead to performance degradation. It was found that incorrectly implemented Semi-Discrete pattern could result in running additional clock cycles while performing scalar additions and multiplications [11]. Another possible drawback of using design pattern in scientific field is that it might use memory inefficiently [11]. These drawbacks are very critical for scientific systems because these systems are performance and memory intensive and waste or misuse of these resources can be costly.

3.2 Application in Real Time Systems

Real Time applications are different from other applications as they have to deal with time constraints and execute request within their deadlines. Air Traffic Control System and Freeway Management System are some example of Real Time systems, as they deal with real time data from flight and automobiles on freeway respectively [10]. Real time applications interact with real time database and transaction on this real time database puts constraint on both data and time, making real time application inherently complex [10].

Real Time patterns are described as patterns intended to provide behavior and structural solutions to the design problem of real time application [10]. Real time application needs to manage memory, resources, parallel computation and security which make design and development of these applications difficult. Real Time pattern provide solution to these problems. Recently in 2010, sensor pattern was proposed after reviewing that current real time patterns only solves the management of resource and security problem, but they don't solve modeling problem of Real Time systems like what essential data must be stored to fulfill the transaction [10]. Sensor pattern helps designer to build Real Time

application by expressing time constrained data and methods in reusable format [10].

3.3 Application in Communication Systems

Network speed and computer performance are increasing, however it is still complex to implement and design communication software [18]. Communication software is used to distribute work among multiple computers and makes it possible to communicate as per the requirements [18]. Communication softwares are used and researched by companies like Motorola, Ericsson and Kodak [18]. In their projects, they used design patterns to reuse software architecture in communication software. Example of one such pattern is Reactor Pattern that simplifies handling of requests for multiple events simultaneously. It was developed to support communication software which runs on Operating System with no multithreading support [18]. This reduced dependency on underlying operating system and thereby the architecture can make design pattern usable in many different scenarios.

4. Evaluating Design Patterns

Practitioners and researchers have been describing the advantages of using design patterns and they suggest implementing it in software [17]. However, there exist some problems with creation of new patterns as well as implementation of old patterns. In this section, we discuss about these problem faced in industry and solutions which can overcome these problems.

One problem is the detection of new patterns as "Patterns explicitly capture knowledge that experienced developers already understand implicitly" [18]. As patterns are described in an abstract way, it can be confusing sometimes for developers to relate or to select design pattern for current problem. Even if design pattern solves very complex design problem easily it would not be used by developers if they cannot relate it with the problem. This can be solved by providing more number of examples with patterns [18].

Patterns are useful but should not be used for implementing each and every part of software system [18]. Sometimes developers use design pattern even for very simple problem and at very granular level like implementation of specific algorithm. This is redundant use of pattern and doesn't add any value.

Design pattern emerges from the knowledge of experience programmers [18]. If they are not awarded for their knowledge, they might be reluctant in sharing it with others [18]. Another reason of having this reluctance is to have an edge over other programmers [18]. Developers who develop design patterns should not work alone; but they should work with domain experts. Working alone may result in development of an over simplified pattern, resulting into a less useful design [18].

One of the most beneficial uses of design patterns is the reuse of software design as they provide an abstract solution which can be implemented in different domains [18]. However, knowledge of design patterns in context of domain is required to understand pros and cons for its implementation [18]. Implementing pattern correctly in software is not an easy task and it requires extra development effort and time [18]. Also, design patterns can sometime give false belief to developer that they know more about the solutions [18]. This happens due to once developer knows about structure of patterns; he/she disregards the challenges in implementation. Hence developers should strengthen their design and coding skills [18].

Selection of language also affects the performance of the implementation used for the solutions [11, 18]. Language selection could be deciding point for selection of design pattern as there can be features in language which will reduce performance. Compiler cannot optimize polymorphic abstraction is one such case [11]. If performance and programming language cannot be changed then not using or tweaking design pattern could be beneficial [11]. One such concern was also found in work of Schmidt [18] where in communication system programmers were concerned about performance. Developers overcame this concern by implementing patterns using parameterized type of C++ instead of inheritance and dynamic binding [18].

Design patterns provide information of classes, objects and how they interact with each other. Hence information of design pattern also helps in understanding the structure and characteristics of the source program which can be helpful to developers and can be used in maintenance or re-engineering of the system [24]. It was found that developers spent more than half of their time understanding source code and the maintenance phase itself cost about 50% of the entire programs cost [12]. A review on the effect of adding comment lines with information of pattern usage in software was carried out which deduced that adding extra pattern comment lines resulted in maintenance task faster than maintenance of software with no additional pattern comments [9].

Design pattern also helps in product life cycle management. One important requirement for product cycle management is keeping track of information associated with products. With the increasing demand for product customization it becomes necessary to keep track of information on the component level rather than the product type level [21]. Also, products components may be made by different manufactures; hence sometime it may become difficult to retrieve such information quickly resulting in information management problem [21]. In order to address this issue, two industrial solutions model were provided for: (1) Composite products, which deals with sub-components of the products and; (2) Observer, which deals with information other than product's

components, like how the product needs to be transported [21].

Design pattern with similar names (Composite pattern and Observer pattern) are defined by GoF [15] and they can be used for information management. One use of composite design patterns is to group objects together and then to apply operations on these group as a whole [21]. This concept is similar to real world products where parts of the products are combined resulting in information between parts being interconnected to provide information at a general level rather than at specific [21]. This information at the top hierarchy being grouped as a whole among sub-components, can be used to provide details about the information located at a hierarchy below, thereby managing information flow. Observer pattern can be used as well in management because Observer pattern is used in setups where a change in one object affects the change in other objects [21]. Design patterns are used in Graphical User Interface (GUI) where interacting with one element of GUI affects the other [21]. This concept can also be applied to managing product information. A change in information in one sub-component can be provided to other manufacturers [21]. Another property of observer pattern is that only subscribed observers can receive the information from the services to which they are subscribed. Using this property, information can be transferred to only those manufacturers whose production depends on the information received.

Design patterns were written in abstract nature [15]. To use them in any domain one needs to instantiate them with domain information [5]. Software development sometimes uses multiple patterns for an application. Applying multiple patterns is susceptible to error in integration and often difficult to implement [5]. Therefore integrated patterns need extensive testing. Instantiation of design pattern result in more number of class because implementation contain domain specific classes [5]. For example Observer Pattern once instantiated will have more number of classes, as it will contain concrete implementation of observer. Therefore testing integration of design pattern before instantiation is simpler as it will have less number of classes to integrate. However question arises whether instantiation after integration of design pattern is always possible? Are they commutable? It was observed that in certain cases instantiation and integration are commutable [5]. Therefore under conditions in which instantiation and integration of design patterns are commutable, testing effort can be reduced by integrating before instantiating.

During the development of software, it may be possible that new requirement gets added or the existing one gets modified. In either case, it may be required to add new classes or delete existing ones to conform to the change. It was found that there is direct relation between class size and changes made to evolve the system [27]. Bigger the class size, it is more susceptible to changes [27]. Also, classes taking part in pattern are more prone to change then classes not taking part. This is unexpected because theoretically design pattern classes should be extended if system evolves [27]. This was explained through informal analysis that design patterns are used for implementing crucial functionalities which made them more susceptible to change [27].

Experiences of expert designer are captured by design pattern to help or reuse design solutions by other programmer. However Design patterns usage gets restricted because it's hard to find or visualize which pattern will help in solving design problem while implementation [7]. This requires experience and knowledge on developer's parts [7]. Therefore, to simplify selection of design pattern, problem domain for patterns should be described thoroughly. This can be done by providing details about problem domain where design pattern is already implemented successfully. Problem domain of design pattern when described using precise notation, can help in automation of evaluation of pattern applicability [7].

In the next sub-section, we evaluate design patterns in different aspects of: (1) Software Quality; (2) Software testing; (3) Evolution and Robustness and; (4) Maintenance compared with non-design pattern solutions.

4.1 Software Quality

Design pattern improves design quality of software by using knowledge of experienced developer [26]. Software quality depends on validation and verification of software. Therefore Validation and verification of design patterns have direct effect on software quality. But it is hard to test completeness and correctness of design pattern by any kind of testing. Schmidt [18] suggested validating patterns on continuous basis as they can be tested more efficiently by experienced developers then by any other testing methodology.

Before development of any system, requirements are gathered. These requirements can be divided in two types, namely functional requirements (FR) and non-functional requirements (NFR). Functional requirements are the functionality which system must perform and they are the reason for the existence of the system. However non-functional requirement are qualities or properties system must have. There are ways in which design pattern's quality can be quantified using these requirement [26]. Design pattern provides way to solve a design problem, which is usually related to functional requirement, in such a way that non-functional requirements can be met [26]. In past few years, there has been work done to find "how design pattern enhance non-functional requirement" thereby improving software quality [26].

Another way to measure software quality is by finding out answer to the following questions: Is code maintainable? Is software modularized? Are modules reusable? All these factors are affected by Cohesion and

coupling in code. Cohesion tells about how closely components are related in single module and coupling tells about degree of relation or linkage between multiple modules [37]. Good quality system tries to achieve high cohesion and lowest coupling [37]. High cohesion provides a specific purpose to the module [37]. This makes module reusable for same purpose anywhere because if modules achieves multiple purpose, then it might require changes before getting reused. Loose coupling among modules provides easy replacement making system easy to maintain or upgrade [37]. Design patterns try to make use of these details about cohesion and coupling which also helps in satisfying non-function requirements [26]. *The observer pattern* has functional intent of notifying subject's state change to observers. Non-functional intent of the observer pattern is in not knowing concrete types of observer, just notifying them about changes. This pattern decouples subject from observers, thereby being coupling improver [26]. Similar coupling improvement is achieved by the *Abstract Factory Pattern* where client requesting object from factory doesn't know about its concrete type. However, these are not the only way to improve quality. Other patterns, like the *Strategy Pattern* improves usage of polymorphism in code, thereby improving flexibility of code because it allows easy replacement of algorithm. There also exist design pattern which improves different aspect of quality like encapsulation, abstraction and complexity [26].

Design patterns do improve design quality but not all of them [26]. Quality-improver design pattern can be defined "as a design pattern which intents to address quality requirements and has a better structure for addressing them" [26]. In other word, design pattern which satisfy non-functional requirement and improves software quality are quality improver [26].Other patterns which don't fall under quality improver category, fall under categories like design problem solver and design conflict solver [26].

Design patterns can be used successfully to analyze quality of software [22]. In one approach, numbers of design patterns per 10 classes (P10C) was used as measure of software quality. This was termed as Design Quality Level (DQL); DQL was termed as high when P10C were greater than 6, as medium when P10C was between 5 to 3 and as low when P10C was less than 3 [22]. This approach was validated using two Object Oriented metrics, namely Coupling Between Object (CBO) and Weighted Method per Class (WMC). CBO tells about coupling by providing average number classes to which each class is coupled [22]. Higher CBO is bad for system quality as higher coupling makes system hard to maintain. WMC measures sum complexity of all the methods in class [22]. Therefore lesser WMC means lesser complexity in class to maintain and thereby easy to maintain. Experiment was done on different projects to find out validity of this analysis technique [22]. Table 2 shows analysis result of 9 open source projects. It can be seen that projects like Java Util, JHot Draw and Java Swing which have high DQL, had low CBO and WMC while Projects like Card Game, Net.Protocol and Mobile Game with low DQL have high CBO and WMC. This result shows that higher design pattern usage improves design quality and reduces complexity [22]. This approach, if used with automatic pattern detection tools, can be beneficial for software engineer to analyze quality of software faster. However, there is more for design than design patterns which impacts quality [22].

SL. No.	Projects	*DQL* by pattern	*WMC*	*CBO*
1	Java Util	High	9.95	12.44
2	JHot Draw	High	6.84	6.91
3	Javax Swing	High	1.93	2.39
4	Fitness	Medium	10.14	19.67
5	JDOM	Medium	25.77	14.33
6	Deputy	Medium	61.02	84.35
7	Card Game	Low	715.31	414
8	Net. Protocol	Low	84.10	210.26
9	Mobile Game	Low	600.85	128.76

Table 2: Design quality level (DQL by pattern) [22]

4.2 Software Testing

Testing is one of the crucial and time consuming parts of software life cycle. Design pattern increases code reusability. However it adds complexity in code by adding lots of polymorphism. This extra interaction adds more chances for error, therefore if not implemented correctly; design pattern can result in many bugs. Not only it's tough to write test cases for polymorphic interaction, they are time consuming, expensive and error prone [23]. This can be explained with an example where abstract child classes are inherited from abstract parent class. Same hierarchy is followed with concrete classes. Therefore, higher the number of concrete implementation, more are the number of classes. Usually testing each class requires access from interface but in case if more classes are added, testing becomes complicated.

However, features of object oriented language can be beneficial for testing in some complex cases [23]. One of such example is of Abstract Factory Pattern which is used for creation of new object using abstract interface. These abstract interfaces are called abstract factories and they provide object reference to newly created objects. These abstract factories are extended by classes called concrete factories, which create concrete objects. While testing, abstract factory can dynamically refer to concrete factories which creates object specific for testing. By doing this, pattern decouples module under test from other dependency and provides ideal scenario for testing any module [23]. Other pattern which can benefit testing is Decorator Pattern which is used for adding and removing properties of an object at runtime. Decorate

Pattern when used in testing environment allows reuse of test code without requiring any changes in source code [23].

4.3 Evolution and Robustness

Softwares are getting complex on daily basis while industrial demands are increasing for fast and timely delivery of the product. However, quick delivery doesn't mean low quality software. This is because once software is accepted it becomes part of "legacy" system [3]. Legacy systems are not only used by other systems but other systems are built on top of them. Therefore software engineer should not give up on quality for time constraints. This makes robustness of legacy product very important factor in today's world. Designs are at times improperly represented [3]. Also with change in requirements, design needs to be updated. Hence not only robustness, evolution of legacy system design also plays an important role [3].

Software can also evolve in terms of scalability and other non-functional requirements [3]. If evolution of software becomes complex then it will make it costly and the customer might abandon it and approach a new system [3]. Design patterns can significantly help in achieving non-functional requirement, as previous experience of design pattern can inform easily about tradeoff with non-functional requirement prior to implementation. Systems having layered and modular architecture implemented by design pattern are flexible and easy to scale [3].

Patterns like decorator pattern can help in reducing complexity, as it extends responsibilities without defining too many classes [3]. For improving flexibility, Adapter pattern can also be used. Adapter pattern is applied when interface doesn't match what user needs. Adapter pattern then transforms needed interface according to user needs, providing flexibility. Pattern when applied in system provide less complexity making system more testable and robust. Patterns which make system more flexible, also makes its evolution process easy [3].

4.4 Maintenance Compared With Non-Design Pattern Solutions

Design pattern uses techniques likes polymorphism, abstraction and encapsulation [18, 20]. However, it is not always necessary to solve problem using design pattern, as they consume time to understand and requires precise knowledge. The solutions which do not use design pattern are termed as simpler solutions as they are implemented in adhoc manner rather than using subtle experience of previous programmer like design pattern. While deciding for whether to use design pattern or simpler solution, there is always a question whether implementing design pattern make solution more complicated than actually required. It was found that maintenance time is reduced if patterns are used compared to simpler solution [9]. But in a study it was found that at-times simpler solutions were easy and fast to maintain in comparison to design pattern [9]. Sometimes software industry requires fast and continuous change in software with the ever changing requirement from customer. Extra feasibility provided by design pattern and fast maintenance can be very useful. Comments in code specific to the patterns, makes maintenance task easier and faster for developers [14]. So using design pattern (without overdoing it) should be primary choice unless it is known that its simpler solutions can be beneficial [9].

5. Tools and Automation

Looking at the benefits of design patterns, many tools and plug-ins have been developed to utilize their advantages. In this section we discuss about two kinds of tools: (1) For detecting design pattern and; (2) For tracking evolution of design based on patterns.

5.1 Detection tools

For automated detection of design patterns, one needs an organization of classes and details about how they interact with each other. This input is required to compare with the existing organization of classes of design patterns [16]. Design pattern solutions are "design motifs, prototypical micro-architectures from which developers draw inspiration to design and implement their programs" [12]. Using this knowledge, many tools were developed to detect design patterns.

Design pattern tools detect patterns by using concept of complete occurrence and incomplete occurrence [12]. If classes highly conform to the design motifs, they are called complete occurrence [12]. Generally it is harder to find complete occurrence in the program since programmers have to adapt their codes according to the software [12]. In incomplete occurrence, the organization of classes and structure does not strictly follow the design motif as stated [12]. However, identification of incomplete occurrence is difficult and time consuming because it requires; (1) searching through all combination of classes for matching design motif and; (2) returns false positives [12].

There are many algorithms available for design pattern recognition. An algorithm based on bit vector was developed to detect design patterns in small to medium scale programs [16]. For pattern matching, bit vector algorithms are generally used [35, 36] as they help in limiting the number of operations required to find a solution, making them independent of the size of the program. Also, only those operations are allowed that can be used to exploit the parallelism in processor, thereby implementing them efficiently. The bit vector algorithm discussed two aspects of identification: (1) Quality of micro-architecture, i.e., the precision of finding a correct design pattern and; (2) Quality of Identification process, i.e., time, resources, human interaction required [16]. The

bit vector algorithm was applied on three small scale programs and it was able to detect design pattern successfully [16]. However, the algorithm could only determine those design patterns that were present in PADL [13] design pattern repository. In comparison with Ptidej [13], it was found that bit wise vector approach is more efficient and better than constrained based approaches. According to one study, regarding identification of design motifs in object oriented codes, it was found that using numerical and structural approaches, improves the performance and precision in detection of complete or incomplete occurrence of design pattern [13].

There are some tools that detect design pattern by dividing the detection process in two phases. In one such tool, the first phase detects the design patterns using visual language parsing and the second phase inspects the source codes [30]. The tool was developed as an extension from previous work [30, 31] thereby increasing

Authors/ Reference	Recovery strategy	Tool	Manual (M) Semi-Automatic (SA) Automatic (A)	Input source code	Recovered design patterns	Software case studies	Precision
Keller et al. (1999)	Minimum key structure	SPOOL	All	C++	Template method, factory method and bridge	2 industrial systems (telecommunications domain), ET++	–
Philippow et al. (2005)	Minimum key structure, with negative and positive criteria	–	A	C++	GoF patterns Gamma et al. (1995)	Student projects	100% for Singleton and Interpreter
Kramer and Prechelt (1996)	Class structure	Pat	A	C++	Adapter, bridge, proxy, composite, decorator	NME, LEDA, zApp	From 14% to 50%
Beyer and Lewerentz (2003) and Beyer et al. (2005)	Predicate Calculus	Crocopat	A	Java/C++	Composite, mediator	Mozilla, JWAM, wxWindows	–
Dong et al. (2007) and Dong and Zhao (2007)	Matrix and weights	DP-Miner	A	Java	Adapter/command, bridge, composite, strategy/state	Java AWT, JEdit, JHotDraw 6.0b1, JUnit	From 91% to 100%
Balanyi and Ferenc (2003)	Class structure	Columbus	A	C++	Reclassified GoF patterns Gamma et al. (1995)	Jikes, Leda, StarOffice, StarOffice Writer	<60%
Heuzeroth et al. (2003)	Predicates on abstract syntax trees	–	A	Java	Observer, mediator, chain of responsibility, visitor, and decorator	Swing	–
Niere et al. (2002, 2003)	Cliché recognition and graph transformation, with fuzzy logic	FUJABA	SA	Java	GoF patterns Gamma et al. (1995)	AWT library	–
Antoniol et al. (2001)	Cliché matching with software metrics on the class structure	–	A	C++	Adapter, bridge, proxy, composite, decorator	LEDA, libg++, galib, mec, socket	30%
Kim and Boldyreff (2000)	Metrics	–	A	C++	GoF patterns Gamma et al. (1995)	3 Systems (no info)	Avg 43%
Olsson and Shi (2006)	Class structure, exploiting inter-class relationships	PINOT	A	Java	Reclassified GoF patterns Gamma et al. (1995)	ANT, AWT, JHotDraw, Swing	–
Gueheneuc et al. (2006)	Bit-vector based on string representation	–	A	Java	Abstract factory and composite	JHotDrawm, QuickUML, Juzzle	–
Smith and Stotts (2003)	Elemental design patterns and rho-calculus	SPQR	A	Java	Decorator	No case study	–
Tsantalis et al. (2006a)	Class structure expresses as matrices, exploiting Graph similarity algorithm	–	A	Java	Composite, adapter/command, decorator, observer, state/ strategy, prototype, visitor	JHotDraw, JRefactory, JUnit	100%
Proposed approach	XPG formalism and LR- based parsing	DPRE	A	Java	Adapter, bridge, composite, façade, proxy, and decorator	JHotdraw 5.1, JHotdraw 6.0b1, QuickUML, Apache Ant, Swing, and Eclipse JDT (components UI 3.3.2 and CORE 3.3.3).	From 62% to 97%

Table 3: Design pattern Tools [4]

the precision and reducing false positives. However this tool is only applicable for structural design patterns and hence it is not useful when detecting creational and behavioral design patterns [30].

Table 3 shows a comprehensive comparison of different tools [4]. The table summarizes 15 different tools, some of which are dated as early as 1996. Column 1 describes about the author and the year in which the tool was developed. Column 2 tells about strategy used to retrieve design patterns. Column 3 contains the name of the tool developed. Column 4 states whether the tool works manually, semi-automatic or automatic, i.e. the level of human interaction required. Column 5 states about the type of input source code as supported by the tool for design pattern detection. Column 6 states the design patterns which the tool is able to recover. Column 7 contains software used for testing the tool and column 8 shows precision level provided by the tool. As seen from the table, during the late 1990's, tools were developed that used strategies like minimum key structure and class structure to detect design patterns. It was created to address design patterns detection in object oriented language like C++. Later, with the emergence of JAVA, a subtle transition can be seen in the development of design pattern detection tools directed to help programmers in detecting pattern detection in programs coded in JAVA. More tools started emerging up addressing JAVA platform from 2002-2003 and onwards. Over the years, not only the source code (JAVA, C++) used to detect design pattern changed, but different strategies were developed to detect design patterns automatically. It is also observed from Table 3 that precision value varies widely when one tool is compared to other. It can also be seen that most of the tools are developed to address and detect a few specific design patterns with high accuracy. Thus there still exists a need for a tool which can detect all design patterns with high precision.

SPOOL was the tool developed to retrieve design pattern from programs coded in C++ by using structural description of design patterns [4]. It was extended later to provide high precision and to address design pattern detection of all patterns as defined by Gang of Four (Gof) [15]. The PAT systems used information of pattern class structure to retrieve design patterns [4]. However, even though this system works automatically, false positives need to be taken care manually [4]. CROCOPAT uses relational expression to describe system and automatically recovers design patterns [4]. DP-Miner retrieves design pattern by building a matrix. The elements of these matrices are classes obtained from the source codes and the relation between two elements (classes) is expressed as prime number while the combination of relations is expressed as product of these prime numbers [4]. Thus computationally, detecting design patter is now reduced to performing arithmetic computation [4]. SPQR uses a rho-calculus, a formal language to describe relationship among the source code and design patterns for retrieving design patterns [4]. The Columbus tool creates Abstract Semantic graph from codes which is compared with the XML DOM tree to detect design patterns [4]. FUJUBA uses fuzzy logic along with abstract semantic graph for retrieving design pattern [4]. The tool PINOT, automatically identifies design pattern by analyzing symbol table and Abstract syntax tree (AST) [4].

As we have seen, there are many tools developed to detect design patterns. However there were some tools developed which also aimed at helping software developers to identify existing defect in the design which can be improved [13]. A combined use of PatternBox, which helps in designing architecture, and Ptidej, which helps in detecting design pattern; in the existing architecture was proposed [13].

Most of the object oriented programming is done in JAVA based environment [2]. Hence to address a wide majority of developers in helping out with maintenance and re-constructing, many plugins were developed. There are many plugins developed for Eclipse because it is highly used open source framework, supported by majority of the developers and its functionality can be easily extended by using plugins [2]. An Eclipse plugin called MARPLE (Metrics and Architecture Reconstruction Plug-in for Eclipse), was developed to detect design patterns and help in reconstruction of software design. The architecture of MARPLE is such that it can be used with any programming language [2]. Another eclipse plugin called ePAD (eclipse plug-in for design Pattern Analysis and Detection) was developed which can detect design pattern and provides many visual functions for configuring the functionalities of the design pattern.

Thus, there are many tools developed to extract design pattern information from source code. All these tools seem to work pretty well but there are some constrains, as they are developed only for a particular environment or are still in development phase with their working prototype model (PatternBox, Ptidej, and DPRE) available [4, 13]. Also, a thorough investigation is required to identify which tools are efficient. However, not much work has been to evaluate such tools since there were no common benchmarking system available which can grade these tools according to their usability and precision.

The primary reason behind lack of information on how good a design patterns tool works, was because there was no approved benchmarking tool [33]. Some people tried to came up with a solution by developing a benchmark tool called DEEBEE (Design patterns Evaluation Benchmark Environment) to compare performance of different design pattern tools which is programming language independent, thereby providing support to a large number of programming frameworks [33]. Some developed a framework addressing design pattern

recovery tools [34] to compare different characteristics like "context, input, technique, output, intent, users, and implementation aspects of the tool" [34]. A benchmarking tool that addresses wide aspect of characteristics is very important since the automated detection tools available have different characteristics, represents their output differently and are implemented by different methodologies and algorithms [33].

5.2 Evolutions tools

Design pattern evolution is the concept/process of adapting the design pattern while retaining their properties [1]. During the software development process, the developer may have to modify some components of code to conform to the requirement constraints and thus it may lead to some inconsistency to the design pattern that was originally applied [1]. This inconsistency may be difficult to find and sometimes hard to correct. To address this issue, a tool was developed which help developers in validating their design pattern as the code evolves over time thereby helping in its maintenance [1].

Since, each design pattern has its own way of representing class's structure, most of the design pattern allows changes or evolutions thereby changing its application domain [6]. Detecting evolution in design pattern can be difficult for developer if program includes large number of classes and objects [6]. In order to address this issue an approach to automate this evolution process of design pattern based on QVT (Query, View, Transformation) technique was developed [6].

Thus, it can be seen that many design pattern tools for automation detection have been developed for fostering system maintenance and re-engineering. Since its introduction, the way in which design pattern are applied has changed vastly from writing and applying manually to using automated detection tool as seen from above discussion. Thus it can be seen that from object oriented paradigm, design pattern have evolved in many different domains with its wide variety of applications and tools.

6. Limitations

The number of design patterns that have been discovered and used in the industries have increased manifold. Industrial work on design patterns covered in reviewed papers are very selective and from last century. This might not give insight of work done in last few year in industries. Another limitation of this paper is that it doesn't review practices in design pattern with very recent software languages like Ruby and C#.

This paper does not contain methods of describing design pattern in detail which can affect understanding and implementation. In this paper we have reviewed applications of design patterns in three types of real world systems namely Scientific, Real Time and Communication, but it does not review about their specific subparts or subsystem like user interface. Design patterns were initially implemented for user interface in SmallTalk [29] but our paper doesn't contain information about how far it has advanced in the field of Human Computer Interaction (HCI).

Moreover, new design patterns are being found and implemented with time. However, we have not reviewed how hard it is for user to find or come across new design patterns. This paper doesn't contain details about design pattern repositories and how easily new patterns can be found and used by programmer in industry.

There are large numbers of tools available in the market which helps in detecting and automating design patterns. Because of time constraint, we were not able to discuss all the tools that may also be implemented in frameworks other than object oriented. As we have seen from discussion in Section 5, some tools like SPOOL, PAT, CROCOPAT, DR-MINER, SPQR and DPRE helps in detecting a subset of design patterns, while tools like COLUMBUS and PINOT helps in detecting reclassified GoF [15] patterns. Hence it becomes difficult to arrive at a conclusion to decide which tools are more efficient and helpful.

Although some work has been done on developing a benchmarking tool [33, 34], but there is no standard available metrics or approved benchmarking tool which can evaluate all available design pattern detection tool and provide a grading of such tools. Hence until this issue is addressed, it becomes difficult to measure the superiority of one tool over other.

7. Future Directions

In recent years, aspect oriented programming paradigm (AOP) had gained interest and gradually replacing object oriented paradigm. Object oriented design pattern is used as synonyms for design pattern because it's mostly used with objected oriented programming paradigm [11]. Due to more work done on design patterns in object oriented paradigm, the paper focused more in this area. However, design pattern can be successfully adapted in aspect oriented paradigm [20]. Most of pattern described by GoF [15] for object oriented were successfully transformed to Aspect Oriented Paradigm using new entity called aspect in AOP (Aspect oriented programming) [20]. In future, we plan to work on evolution of design pattern with respect to aspect oriented programming paradigm.

Interesting future works has been proposed to improve the current standard of tools to automate design pattern detection. Tools can be implemented using metrics as it is known to reduce noise thereby will making the tool to detect design patterns more precisely and efficiently [16]. Expressing design motif's string representation as regular expression; and then building an automaton which recognizes this regular expression [16], extending the tool to support all the design patterns [15] using metrics and basic elements and making user

experience better are some of the interesting proposal. Thus, our future work will include study of these tools regarding how usable in terms of functionality and visually appealing they are in making user experience better. We also plan to review impacts on non-functional and conflicting requirement [25], which we feel deserves further research.

8. Discussion & Summary

Design patterns, being an abstract solution, provides an excellent means of communication, improves system design, helps in maintenance, promotes re-usability, helps in learning about programs structure, thus helps in fostering quality product development. Patterns are getting used industry wide on various project sizes and in multiple domains. Software engineers need to have experience in design patterns to select required pattern and implement them correctly. Although, design pattern adds complexity in software testing; they make system robust and flexible for evolution and reduce time for maintenance.

Design patterns are complicated, consumes time and requires prior knowledge about its structure and characteristics. As design pattern requires good knowledge and experience, many automated tools were developed that made it easier for developers to understand and implement design pattern. These tools addresses the design pattern detection problem pretty well, but they tools are still in developing phase. In coming years, developers may be able to extract full power from these tools, making design pattern detection and implementation easier.

References

[1] C. Zhao, J. Kong, J. Dong and K. Zhang. Pattern-based design evolution using graph transformation. *Journal of Visual Languages and Computing 18(4).* pp. 378-398. 2007.

[2] F. Arcelli Fontana and M. Zanoni. A tool for design pattern detection and software architecture reconstruction. *Information Sciences 181(7).* pp. 1306-1324. 2011.

[3] C. Chang, C. Lu and P. Hsiung. Pattern-based framework for modularized software development and evolution robustness. *Information and Software Technology 53(4).* pp. 307-316. 2011.

[4] A. De Lucia, V. Deufemia, C. Gravino and M. Risi. Design pattern recovery through visual language parsing and source code analysis. *Journal of Systems and Software 82(7).* pp. 1177-1193. 2009.

[5] J. Dong, T. Peng and Y. Zhao. On instantiation and integration commutability of design pattern. *Computer Journal 54(1).* pp. 164-184. 2011.

[6] J. Dong, Y. Zhao and Y. Sun. Design pattern evolutions in QVT. *Software Quality Journal 18(2).* pp. 269-297. 2010

[7] D. Kim and C. El Khawand. An approach to precisely specifying the problem domain of design patterns. *Journal of Visual Languages and Computing 18(6).* pp. 560-591. 2007.

[8] S. L. Pauwels, C. Huebscher, J. A. Bargas-Avila and K. Opwis. Building an interaction design pattern language: A case study. *Computers in Human Behavior 26(3).* pp. 452-463. 2010.

[9] L. Prechelt, B. Unger-Lamprecht, M. Philippsen and W. F. Tichy. Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. *IEEE Transactions on Software Engineering 28(6).* pp. 595-606. 2002.

[10] S. Rekhis, N. Bouassida, R. Bouaziz, C. Duvallet and B. Sadeg. (2010). Modeling real-time applications with reusable design patterns. *International Journal of Advanced Science & Technology 23.* pp. 41-55.

[11] D. W. I. Rouson, H. Adalsteinsson and J. Xia. Design patterns for multi-physics modeling in Fortran 2003 and C++. *ACM Transactions on Mathematical Software 37(1).* pp. 3:1-3:30. 2010.

[12] Y. Guéhéneuc, J. Guyomarc'h and H. Sahraoui. Improving design-pattern identification: A new approach and an exploratory study. *Software Quality Control 18(1).* pp. 145-174. 2010.

[13] H. Albin-Amiot, P. Cointe, Y.-G. GueÏheÏneuc and N. Jussien. Instantiating and Detecting Design Patterns: Putting Bits and Pieces Together. In *Proceedings of the 16th International Conference on Automated Software Engineering (ASE'01)*. pp. 166-173. 2001.

[14] L. Prechelt, B. Unger, W. F. Tichy, P. Brossler and L. G. Votta. A controlled experiment in maintenance: Comparing design patterns to simpler solutions. *IEEE Transactions on Software Engineering 27(12)*. pp. 1134-1144. 2001.

[15] E. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse in object-oriented designs. In *Proceedings of the 7th European Conference on Object-oriented Programming (ECOOP'93)*. pp. 406-431. 1993.

[16] O. Kaczor, Y.-G. Gueheneuc and S. Hamel. Efficient identification of design patterns with bit-vector algorithm. In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering*. Bari, Italy. pp. 175-184. 2006.

[17] T. H. Ng, Y. T. Yu and S. C. Cheung. Factors for effective use of deployed design patterns. In *Proceedings of the 10th International Conference on Quality Software (QSIC)*. Zhangjiajie, China. pp. 112-121. 2010.

[18] D. C. Schmidt. Using design patterns to develop reusable object-oriented communication software. (1995). *Communications of the ACM 38(10).* pp. 65-74. 1995

[19] K. Beck, *et al.* Industrial Experience with Design Patterns. In *Proceedings of the 18th International Conference on Software Engineering*. Berlin, Germany. pp. 103-114. 1996.

[20] Z. Vaira and A. Caplinskas. Software engineering paradigm independent design problems, GoF 23 design patterns, and aspect design. *Informatica 22(2)*. pp. 289-317. 2011.

[21] K. Främling, T. Ala-Risku, M. Kärkkäinen and J. Holmström. Design patterns for managing product life cycle information. *Communications of the ACM 50(6)*. pp. 75-79. 2007.

[22] M. Abul Khaer, M. M. A. Hashem and M. Raihan Masud. On use of design patterns in empirical assessment of software design quality. In *Proceedings of the International Conference on Computer and Communication Engineering (ICCCE)*. Kuala Lumpur, Malaysia. pp. 133-137. 2008.

[23] P. Dasiewicz. Design patterns and object-oriented software testing. In *Proceedings of the Canadian Conference on Electrical and Computer Engineering.* Saskatoon, Canada. pp. 904-907. 2005.

[24] A. De Lucia, V. Deufemia, C. Gravino and M. Risi. An eclipse plug-in for the detection of design pattern instances through static and dynamic analysis. In *Proceedings of the International Conference on Software Maintenance (ICSM)*. Timisoara, Romania. pp. 1-6. 2010.

[25] Nien-Lin Hsueh and Wen-Hsiang Shen. Handling nonfunctional and conflicting requirements with design patterns. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference*. pp. 608- 615. 2004.

[26] Nien-Lin Hsueh, Peng-Hua Chu and William Chu. A quantitative approach for evaluating the quality of design patterns. *Journal of Systems and Software*. pp. 1430-1439. 2008.

[27] J.M. Bieman, G. Straw, H. Wang, P.W. Munger, R.T. Alexander. Design patterns and change proneness: an examination of five evolving systems. In *Proceedings of the 9th International Software Metrics Symposium*. pp. 40- 49. 2003.

[28] Kent Beck. Using a pattern language for programming. In *Addendum to the Proceedings* of *OOPSLA'87*. vol. 23. pp. 16. 1988.

[29] Christopher Alexander. *The Timeless Way* of *Building.* Oxford University Press. New York. 1979.

[30] Costagliola, G., De Lucia, A., Deufemia, V., Gravino, C., Risi, M. Design pattern recovery by visual language parsing. In *Proceedings of European Conference* on *Software Maintenance and Reengineering (CSMR'05)*. Manchester, UK. pp. 102-111. 2005.

[31] Costagliola, G., De Lucia, A., Deufemia, V., Gravino, C., Risi, M. Case studies ofvisual language based design pattern recovery. In *Proceedings of EuropeanConference* on *Software Maintenance and Reengineering (CSMR'06)*. Bari, Italy. pp. 165-174. 2006.

[32] Vokac, M. Defect frequency and design patterns: an empirical study of industrial code. *IEEE Transactions on Software Engineering 30(12)*. pp. 904- 917. 2004.

[33] Fulop, L.J.; Ferenc, R.; Gyimothy, T. Towards a Benchmark for Evaluating Design Pattern Miner Tools. In *Proceedings of 12th European Conference* on *Software Maintenance and Reengineering (CSMR'08)*. pp.143-152. 2008.

[34] Y.-G. Gueheneuc, K. Mens, and R. Wuyts. A Comparative Framework for Design Recovery Tools. In *Proceedings of the 10th Conference* on *Software Maintenance and Reengineering (CSMR'06)*. pp. 123–134. 2006.

[35] R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. In *Communications of the ACM 35(10)*. pp. 74–82. 1992.

[36] J. Holub and B. Melichar. Implementation of nondeterministic finite automata for approximate pattern matching. In *proceedings of the 3rd International Workshop* on *Implementing Automata*. London, UK. pp. 92–99. 1998.

[37] Woodward M. R. Difficulties using cohesion and coupling as quality indicators. *Software Quality Journal* 2 (2). pp. 109. 1993.