# AN ANALYTICAL APPROACH FOR ASSESSING SOFTWARE DESIGN PATTERNS AND EVALUATE DESIGN PATTERNS TO DETERMINE THEIR EFFECT ON SOFTWARE QUALITY

**[1] Deepa Kumari**
**Designation**- Research Scholar
**Department**- Mathematics
**University**- Magadh University, Bodhgaya
**Email**-Smiriti.Sinha7@Gmail.Com

**[2] Sunil Suman**
**Designation**- Assistant Professor
**Department**- Mathematics
**University**- Magadh University, Bodhgaya

**ABSTRACT**

*An essential part of developing high-quality software is evaluating and analyzing software design patterns. This analysis method aims to determine the impact of design patterns on various measures of software quality in a methodical manner. "The method involves learning about design patterns, defining evaluation criteria that are in line with desired software quality attributes, finding design patterns in the software, analyzing their impact on software quality attributes, performing code reviews, assessing relevant metrics, comparing alternative patterns, taking domain-specific factors into account, documenting findings, and continuously improving the evaluation process. Numerous studies have looked at the impact of design patterns on quality characteristics. However, since these studies come from different perspectives, have different goals, use different metrics, and focus on different quality attributes, their findings are sometimes contradictory and difficult to compare. "The authors hope that they can explain these results by addressing any confounding factors, procedures, measures, or implementation difficulties that affect quality." In addition, there needs to be more research that connects the dots between design pattern evaluations and pattern creation studies. They want to increase software quality by guiding how patterns might be better structured and implemented.*

***Keywords:*** *Software design patterns, Software quality, Evaluation, Design pattern impact, Software quality attributes*
**DOI Number: 10.48047/NQ.2022.20.18.NQ88172** **NeuroQuantology 2022; 20(18): 1630-1640**



## I. INTRODUCTION

Design patterns are a collection of answers to common issues in software development that may be used to produce high-quality code. In architecture, they were initially presented by Alexander et al. [1] in 1977. Gamma et al. [2] suggested implementing architectural patterns in OO programs in the middle of the last century. The "gang of four" (GoF) came up with 23 different patterns for design. "There are three types of GoF design patterns: structural, creative, and behavioral."

Design patterns are reusable strategies for addressing common challenges in the design and development of software parts. It has been argued that design patterns are helpful because they (i) help developers collaborate more effectively, (ii) make for more easily maintainable and reusable software, (iii) facilitate meeting the system's non-functional requirements, and (iv) reduce development time and expenses.

Numerous scholars have done studies to experimentally and analytically examine the assertions above. According to some research, design patterns make it harder to provide the necessary quality in the

program at hand [3,4]. However, other research has concluded that designers should use design patterns to enhance software quality. "The abstract factory pattern, for instance, has been shown to have mixed results: some research suggests it increases design extensibility (e.g. [5-7]), while others have discovered that it decreases it ([8-10]). Most of the assessed patterns have had such conflicting outcomes reported [11, 12]". As a result, there needs to be more disagreement over how much of an effect any given design pattern has on software quality.

The research was conducted to gather information from existing studies to help researchers agree on the impact of design patterns on specific quality characteristics. These analyses were either literature reviews or mapping studies (see, for example, [12, 13] and [11, 14, 15]). These findings suggest substantial disagreement over how design patterns influence software quality. "Riaz et al. [13] discovered that owing to variations in research design and implementation, it is difficult to compare the results of empirical investigations." The significant papers they reviewed led them to conclude that the existing empirical results must be better understood and have limited generalizability.

Fragments of code with code smells are notoriously difficult to update and maintain [11]. Fowler identified code smells in 1999 as indicators of design issues that might make it difficult to maintain software [11]. In contrast to design patterns, code smells are not patterns but indicators of where developers should hunt for a particular problem. Thus, code smells indicate sections of code that need to be scrutinized more closely. Researchers have found varying impacts of code smells on different quality elements, just as they have with design patterns. Li and Shatnawi [12] discovered a connection between code odors and the prevalence of bugs.

Nonetheless, research in [13–15] demonstrates that odors are helpful indications for explaining service problems. Code smells have been identified using a variety of programming contexts and programmer backgrounds [16, 17]. Researchers must thus develop new ways to identify unpleasant odors. Therefore, practitioners may benefit from a broader consideration of other elements when comprehending the connection between foul odors and code. "Design patterns and code smells have been the subject of many contemporary publications." A large percentage of the responses focused on refactoring considerations. Tools that identify code snippets that might benefit from refactoring have been suggested [18–20]. "Other research has also studied the structural connection between design patterns and code smells [21, 22]." The adverse effects of design patterns on traits like maintainability might also produce foul odors, as described by Wendorff [9]. "McNatt and Bieman [23] warned that improper usage of design patterns like the Command, Proxy, Bridge, and Observer might harm system performance and lead to unmanageable code." Good design patterns and code quality go hand in hand, whereas unpleasant odors point to software architecture or implementation problems. Since DPs and unpleasant odors are opposing structures, they are seldom studied together. "More research is warranted into the relationship between design patterns and code smells, with a focus on their co-occurrences, because there are contradictory findings about the impact of DPs on quality attributes, and ambiguous relations that relate code smells with maintainability aspects. In addition, not a ton of research has looked at how DPs and code smells are directly connected to one another [24]." As a result, there needs to be more research conducted to systematically examine and compare broad evidence of the effect of design patterns on code smells. In addition, we investigate whether or not there is a correlation between code smells and DPs across many granularities.



This research expands upon earlier efforts to illuminate how design patterns influence software quality. By investigating the variables, practices, and implementation challenges that impact quality characteristics while using design patterns, we want to explain the seemingly contradictory findings of different studies. We also compiled and analyzed a number of metrics for assessing design patterns and their implementation. "Furthermore, we seek advice on how design patterns, structure, and implementation may be improved to support the construction of high-quality software by explaining the link between the design pattern and quality that is based on several empirical research".

The purpose of this research is to provide a method for analyzing software design patterns and to analyze design patterns to learn how they affect software quality.

## II. RELATED WORK

There has been much interest in learning about design patterns since they were initially suggested. "Numerous research studies have been conducted to find, classify, use, and assess patterns empirically and analytically." Literature reviews help assess research

because they reveal the current status of the field and highlight significant advances.

Weiss [14] summarizes the findings of sixteen research on design patterns and their effect on system issues, published between 2000 and 2008. "The major research is broken down into three groups, with (i) studies that directly relate design patterns and system issues, (ii) studies that aim to promote the selection of patterns, and (iii) studies that explain the justification for architectural choices."

The SLR presented by Zhang and Budgen [15] is in the form of a geographical investigation. "The research examines which GoF design patterns have been studied empirically and what conclusions can be drawn from those studies". Eleven empirical research reported in 10 articles published between 1995 and 2009 are considered.

To summarize the current state of the art in the study of GoF design patterns, Ampatzoglou et al. [12] provide a mapping analysis comprising around 120 primary studies. They look into (a) if the study of design patterns can be broken down into distinct research subdomains, (b) which of these subdomains see the most significant activity, and (c) whether or not there is any evidence linking design patterns to certain qualities of software.

Ali and Elish [11] review the research literature to find examples of GoF design patterns and their effects on software quality. They summarized the effect of design patterns on software quality characteristics and assessed the extent to which empirical data covers quality attributes and design patterns. Seventeen publications published between 2001 and 2012 were analyzed.

In their analysis, Riaz et al. [13] analyzed data from 19 separate empirical investigations reported in 17 separate articles published between 2000 and 2009. In addition to 10 rating criteria and related observable measures, they gleaned helpful information on the participants' demographics, presenting patterns, and problems. They also pointed up problems that researchers can encounter while experimenting.

The authors of this paper, Mayvan et al. [16], mapped out many design patterns. They culled information from 637 publications. Their findings demonstrate the appeal of studying design patterns as a research topic. Furthermore, they discovered that pattern creation, pattern mining, and pattern use are the most often discussed, whereas pattern assessment and pattern definition received fewer publications.

Previous literature reviews and mapping studies highlight problems in evaluating the quality impact of design patterns. "Because of these issues, there are discrepancies in the findings of various research, In addition, considering the wide variety of elements that affect how software is built, it is far from straightforward to investigate the effect of design patterns or any design module, Deploying a solution requires a thorough knowledge of the problem, the solution, and the means of recognizing and quantifying the solution's impact."

## III. THE IMPACT OF DESIGN PATTERNS ON SOFTWARE QUALITY ATTRIBUTES

Researchers examined how DPs affected maintainability, adaptability, performance, and fault tolerance. "Prechelt et al [1] conducted a study to examine the complexity of software maintenance with and without using design patterns." They discovered that design solutions that use GoF's patterns are often simpler to maintain than their more straightforward counterparts. In several cases, however, the authors discovered that design patterns made the software harder to maintain [25]. They also revealed that, in comparison to a simple solution, design patterns may need more upkeep work. "Prechelt et al [1] found that the Visitor pattern resulted in a high cost in terms of development time and low accuracy; Vokac et al [26] confirmed these findings." They discovered that Decorator makes maintenance simpler while making it more challenging to comprehend the code because of how difficult it is to follow the program's control flow. So, it is excellent at being maintained but not outstanding at being understood. Several attempts [27–29] have been made to replicate the findings of [1]. "Others (e.g. [30]) found no apparent pattern of any influence and urged a deeper and a practical investigation be undertaken, while yet others discovered that DPs harmed maintainability [27–29]." Pattern-based designs are more challenging to comprehend and alter, according to Garzas et al. [31]. "The influence of various DPs in JHotDraw systems on maintainability was studied by Hegedus et al. [32], who concluded that DPs may affect maintainability favorably and lead to changes in the code. In order to determine whether or not a maintainer can work more efficiently, Lutz Prechelt et al. [33] ran a controlled experiment to see whether DPs in program code are explicitly described using comments." They reported less time and fewer issues with maintenance chores that relied on patterns. The connection between DPs and adaptability was investigated by Aversano et al. [34]. The findings indicate that DPs are more vulnerable to changes if and only if they play a crucial role in the system's operation. However, Gatrell

et al. [35] discovered that certain participation classes in DPs are more adaptable than non-participant classes. The link between software's adaptability and its architecture was studied by Bieman et al. [36]. The study relied on five different open-source operating systems. Recognizing design structure was achieved by the use of design patterns, class size, and class inheritance. Classes that make use of design patterns were shown to be more adaptable to change than other classes. Design patterns' contribution of crucial essential duties and functionality to the system may account for the observed outcome. Because of this, classes that take part in design patterns may be more adaptable than other classes. "Co-occurrence of design patterns with either the Divergent Change or Shotgun Surgery code smells, as described by Fowler et al. [11], may be a symptom of change-proneness in classes that engage in design patterns."Aversano et al. [34] came to the conclusion that using design patterns results in more stable programming. They also found that applications with frequent changes benefit most from using patterns.

## IV. DESIGN PATTERNS AND MAINTAINABILITY

Software maintainability is one of the primary quality concerns of researchers that examine the usage of GoF design patterns, as shown by two recent mapping studies on the research state of the art on GoF design patterns [37, 38]. "Specifically, only 40% (14 of 35) of the studies on the effect of GoF design patterns on software quality attributes investigate the effect of GoF design patterns on software maintainability, while Zhang and Budgen argue that GoF design patterns provide a framework for maintainability and that future research efforts should be more focused on maintainability [37]." In 2001 and 2004, Prechelt et al. and Vokac et al. conducted two of the most well-known controlled tests on the impact of GoF design patterns on software maintainability ([38], [39]). "Both looked at the effects of design patterns and their absence on the longevity of a system." Abstract Factory, Observer, Decorator, Composite, and Visitor were the patterns considered, and software developers were the test subjects [38]. "Based on the findings of the experiment, design patterns are recommended above simpler alternatives (for more information on the simpler solutions investigated, see articles [38], [39])." Later, Vokac et al. [39] replicated the study using the same patterns and subjects, but this time participants utilized an actual programming environment rather than just paper and pencil. Based on the findings, not all design patterns are created equal regarding their impact on maintenance, and designers should use their discretion when deciding whether to use a GoF design pattern or a more straightforward approach. In order to assess the readability and adaptability of Visitor design pattern examples, Jeanmart et al. [40] experimented with student volunteers. "Various understanding and modification tasks were used to evaluate the experiment's success, which employed three open-source projects as objects (containing canonical and non-canonical representations of the Visitor pattern)." According to their findings, people with solid knowledge of UML notations and who employ the canonical representation of the Visitor pattern spend less time on modification tasks. To help decide between using patterns and other design solutions for software maintenance, Ampatzoglou et al. [41] have sought to present an objective method. They presented a metric score that is a mathematical function of the number of classes that are part of the pattern, and they used a set of predictors for maintainability to arrive at this value. They used this technique to the Bridge and Abstract Factory design patterns and came up with a set of thresholds, or limits, on the number of pattern participating classes beyond which the solutions become less maintainable than the alternatives, and similarly for the other way around. Two studies, one by Ampatzoglou et al. [41] and another by Jeanmart et al. [40], identify factors that can predict whether a design pattern solution will be more maintainable than another. These factors are the number of classes and familiarity with UML notations. Many additional studies have empirically evaluated design patterns to keep the software running smoothly. Khomh and Guéhéneuc polled highly skilled software engineers to get their opinions on the eight quality features of the GoF design patterns. The degree to which a system's design may be expanded as the associated quality feature with maintainability. Based on the findings, 19 of the 23 GoF design patterns are helpful in terms of maintainability, while just four—the Singleton, Flyweight, Proxy, and Memento patterns—are detrimental.



## V. DESIGN PATTERNS AND STABILITY

program stability is defined in many ways, including by ISO-9126 and by Yau and Collofello ([42], [43]), who describe it as resistance to the propagation of changes (ripple effect) that the program would have when it was updated. "The primary purpose of this research is to assess how GoF design patterns affect stability, but we will not ignore related studies that focus on GoF design patterns and change proneness because (a)

there has not been much done on the topic of patterns and stability, and (b) instability is a subset of change proneness."Bieman et al. did some of the first research on design patterns and class switching. A 2001 industrial case study by the authors examined the relationship between code modifications, reusability, design patterns, and the number of classes in use. The study's findings imply that the likelihood of modifications increases with class size [44] and that classes with pattern-related or inherited reuse are more likely to undergo modifications. "In 2003, the same authors utilized three commercial and two open-source programs to replicate the case study and achieve the same goals." There needs to be more consistency in the findings of the two studies. Despite this, the connections between class size, engagement in design patterns, and adaptability are weakening [45]. Using the same GoF design principles, Gatrell et al.'s 2009 work on proprietary C# apps is a carbon copy of Beeman's [44, 45]. Other than the language used to code, the most notable distinction was the unit of analysis used for modifications. Gatrell measured modifications per class, whereas Bieman tracked modifications per operation. Replication findings, however, confirmed that classes with higher rates of GoF pattern occurrences are more adaptable than those with lower rates. Di Penta et al. [46] looked for links between the propensity of a class to change, the significance of that class's participation in the pattern, and the nature of that class's transformation. "Abstract Factory, Adapter, Command, Composite, Decorator, Factory Method, Observer, Prototype, Singleton, State, Strategy, and Template Method are the design patterns analyzed in this paper." Three open-source initiatives were analyzed. Intuitively, the paper's findings apply to most possible class roles in a design pattern implementation. "Classes in more abstract roles, such as the Abstract Factory in the Abstract Factory design or the Product in the Command pattern, tend to undergo fewer frequent changes than their concrete counterparts." The Command pattern is another typical design pattern where classes in the Receiver role are more likely to be replaced than classes in the Command role. It is also recommended that designers consider the different functions a class may do since the relative ease with which interface responsibilities vary can reduce the system's overall flexibility. Based on this work, Aversano et al. [47] explored how GoF design patterns have changed over time regarding the actual modifications made to pattern occurrences in various versions. In particular, the authors repeated and expanded upon Di Penta et al. [46] work by studying the effects of pattern alterations on client and target classes.

## VI. ANALYSIS AND DISCUSSION

According to the United States Department of Defense, the software's complexity is the system's greatest threat. According to Bruce Schnier (Schneier, 2008), "The Future of digital systems is complexity, and complexity is the deadliest adversary of security." Consequently, researchers ignore security concerns in favor of design complexity matrices to gauge the generated model's efficacy. Metrics for the complexity of designs are calculations used to assess certain types of complexity.



AutoSBP

| No | File Name | No of States | No of Transition | Time Taken | Time per State | State Transition Complexity | Overall Design Complexity |
|---|---|---|---|---|---|---|---|
| 1 | ad_bank.xmi | 10 | 12 | 724 | 72.4 | 0.166666666666663 | 0.1818181818181817 |
| 2 | ad_laboratory | 8 | 13 | 729 | 91.125 | 0.5 | 0.3846153846153846 |
| 3 | ad_book_issue.xmi | 10 | 11 | 610 | 61.0 | 0.166666666666663 | 0.0909090909090904 |
| 4 | ad_discharge.xmi | 8 | 11 | 506 | 63.25 | 0.272727272727272 | 0.2727272727277272 |
| 5 | ad_patient_Admission | 24 | 31 | 1286 | 53.5833333333333 | 0.27272727272727 | 0.225806451612903 |

*Figure1. The complexityofvarious domainmodels*

To understand complexity, one must first understand the interdependencies between various things (www.jot.fm). A set's size is equal to the sum of its element counts. The more connections there are in a set, the more complicated it is. A system's complexity grows in proportion to the number of its parts and the degree of dependence between them (McCabe & Butler 1989). The number of its constituent parts and the nature of their interdependence define the entity's degree of complexity. The following categories of complexity have been developed for UML models, and they all take into account state and their relationships.

Complete design complexity (Cd) is defined as follows: Equation 5.3.

$$C_d = \frac{N(D_e)}{N(D_r)} \qquad (5.3)$$

WhereN(De)representsthenumberofdesign entitiesand N(Dr)representsthenumberofrelationshipsbetweendesignentities.ActivityControlFlowComplexity(Cacf)is givenby,

$$C_{acf} = 1 - \frac{N(AD_a)}{N(AD_f)+N(AD_c)} \qquad (5.4)$$

The activity diagram's activity count (N (ADa)), flow count (N (ADf)), and condition count (N (ADc)).

The lines connecting the various tasks in an activity diagram show how decisions are made and communicated. In other words, they might be conditional or unconditional. It is easier to model a process when conditional flows are included. Any other activities may follow in the footsteps of the original activity. The method becomes more complicated as the number of conditions grows.

Figure 1.shows the complexity of AutoSBP applied to source models from various domains. The system's intricacy is determined by considering models from the banking, laboratory, and library sectors. Proof that the system can conduct complexity measurements on models from different domains. Since the produced models are well-designed, complexity metrics may be used to assess the model's quality.

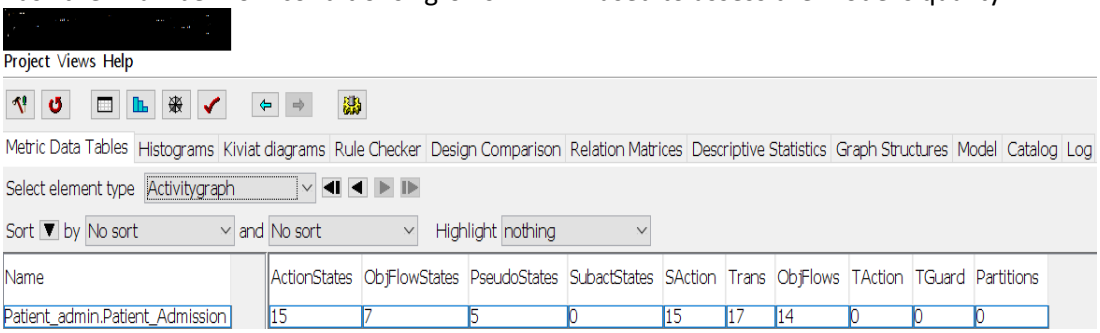


*Figure2. EvaluatedActivitymetrics*

We may use SD Metrics (www.sdmetrics.com) to determine the diagram's size, coupling, and complexity to determine its overall quality. During the model transformation, missing states and transitions may be spotted with the help of the following metrics, which are created to examine the structure of the UML model.

Quantity of Activity Diagram Action and Call States Physical Flow States of Objects (Size) A count of the activity diagram's item Flow states, Disguised Government (Size) Subact State Count, Number of Pseudo States in Activity Diagram (Size) When looking at an activity diagram, the number of sub-activities, Trans (Complexity) Count of Activity Diagram Transitions Without Object Flow,

Table1. EnduserEvaluation

| EvaluationPoints | Excellent(%) | Good(%) | Fair(%) | Poor(%) |
|---|---|---|---|---|
| Clarityoftheoutputmodel | 61.07 | 24.16 | 10.07 | 4.7 |
| Retrievalofsourcestates | 69.13 | 26.85 | 4.03 | 0 |
| Retrieval sourcetransactions | of65.77 | 26.17 | 8.05 | 0 |

| | | | | |
|---|---|---|---|---|
| Retaininginformation | 46.31 | 42.95 | 6.04 | 4.7 |
| Logicalconsistency | 36.91 | 44.97 | 11.41 | 6.71 |
| Securityspecification | 44.3 | 39.6 | 10.07 | 6.04 |

ObjFlows (Complexity) (Complexity) TheTAction diagram's number of item flows (Complexity). TGuard is the number of transitions in an activity diagram (Complexity). Each guard in the transition of activity diagram has a specific number, determining the flow of activities. Partitions (Size) (Size) How many sections an activity graphic has Data Driven Metrics' output for the final stop activity model is shown in Fig. 2. The depicted model is guaranteed by the complexity calculation that is performed. All of the original model's properties are preserved in the final Product, and the UML model's quality is enhanced by including the security requirement. “An assessment of the resulting model in terms of model transformation and security was conducted by providing outcome models and rubrics to 137 end users with profiles ranging from students to software engineers and academics.” Twelve specialists in the field were used to assess the same. Models were scored on their ability to be understood, to retrieve original state and transition data, to preserve original data and logic, and to avoid compromising security. Table 1.and Figure 3, which display the assessment findings, demonstrate the reliable outcome model.



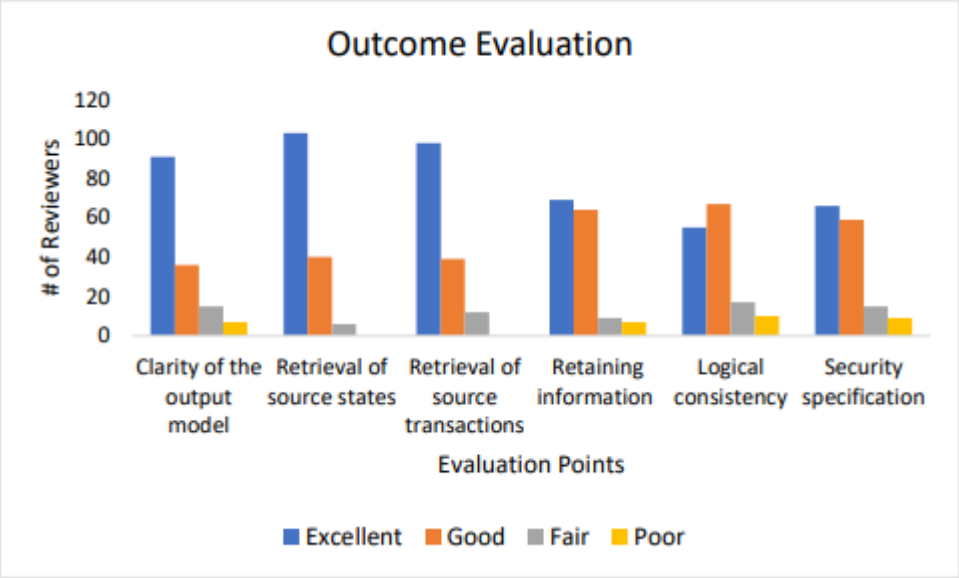


*Figure3. Qualityevaluationonmodeltransformationandsecurity*

Table 2 compares the AutoSBP proposal to many current methods drawn from the literature. This side-by-side analysis reveals that the first studies either concentrate on automated model transformation of heterogeneous models or model transformation, including security. Therefore, the proposal of a paradigm for autonomous endogenous security model transition. Another aspect that the suggested system aims to automate is the implementation of security measures. In addition to improving upon preexisting systems in novel ways, this study proposes a novel approach to securing information through a decision-learning algorithm.

AutoSBP is a framework introduced in this thesis that can automatically inject a predefined set of security criteria into a business process model. Security needs may be outlined in UML 2.0 activity diagrams for business processes. Adding security features (such as Authentication, Privacy, Non-repudiation, Integrity, and Attack Harm Detection) boosts the business

process model's flexibility.

BPM is a significant resource for learning about an application's security requirements in this study. When compared to other standard notations, this one has the added benefit of being able to describe security-related components where they belong in the business model. The decision-learning method automates this process, ensuring that the security requirement is consistently enforced across models. Complexity measurements are calculated to check whether adding security features degrades the quality of the business process model.

**Table2.Comparison with existing methodologies**

| References | Model Transformation | SecurityIncorporation | Security Optimization | Automation |
|---|---|---|---|---|
| Alotaibi& Liu(2014) | x | ✓ | x | ✓ |
| Rodríguez*etal.*(2011) | ✓ | ✓ | x | x |
| Haley*etal.* (2008) | x | ✓ | x | x |
| Mellado*etal.*(2010) | x | ✓ | x | x |
| Rodríguez*etal.*(2010) | ✓ | ✓ | x | x |
| Shin&Gomaa(2007) | x | ✓ | x | x |
| DanielMellado*etal.*(2011) | ✓ | ✓ | x | x |
| Castro*etal.*(2011) | ✓ | X | x | ✓ |
| Al-Amin*et al.*(2018) | x | ✓ | x | x |
| Lee*et al.*(2017a);Lee *etal.* (2017b) | ✓ | x | x | x |
| Lano*etal.*(2018) | ✓ | ✓ | x | x |
| Hamid& Weber(2018) | ✓ | ✓ | x | ✓ |
| Mohammed*et al.*(2017) | ✓ | ✓ | x | x |
| AutoSBP | ✓ | ✓ | ✓ | ✓ |



## VII. CONCLUSION

This research concludes that software design patterns help fix software design issues in general. Based on the results of prior studies, we examine the effects of various design patterns on several aspects of software quality. The proposed analytical technique combines quantitative and qualitative approaches to analyze design patterns and evaluate quality qualities. Based on the data, design patterns do not always lead to a better quality result. Every facet of quality is both favorably and poorly impacted by design patterns.

## VIII. REFERENCES

[1]. Alexander, C., Ishikawa, S., Silverstein, M.: ' *A pattern language: towns, buildings, construction*,' vol. **2** (Oxford University Press, New York, NY, USA, 1977)

[2]. Gamma, E., Helm, R., Johnson, R., *et al*.: ' *Design patterns: elements of reusable object-oriented software* (Addison-Wesley, Boston, MA, USA, 1994)

[3]. Schmidt, D., Stal, M., Rohnert, H., *et al*.: ' *Pattern-oriented software architecture, volume 1: a system of patterns*' (John Wiley & Sons, Chichester, UK, 1996)

[4]. Schmidt, D., Stal, M., Rohnert, H., *et al*.: ' *Patterns for concurrent and networked objects, volume 2 of pattern-oriented software architecture*' (Wiley, Chichester, UK, 2000)

[5]. Prechelt, L., Unger, B., Tichy, W.F., *et al*.: 'A controlled experiment in maintenance: comparing design patterns to simpler solutions,' *IEEE Trans. Softw. Eng.*, 2001, **27**, (12), pp. 1134– 1144

[6]. Aversano, L., Cerulo, L., Di Penta, M.: 'Relating the evolution of design patterns and crosscutting concerns.' Seventh IEEE Int. Working Conf. on Source Code Analysis and Manipulation (SCAM 2007), Paris, France, 30 September–1 October 2007, pp. 180– 192

[7]. Gatrell, M., Counsell, S., Hall, T.: 'Design patterns and change proneness: a replication using proprietary C# software.' 16th Working Conf. on Reverse Engineering (WCRE'09), Lille, France, October 2009, pp. 160– 164

[8]. Vokáč, M., Tichy, W., Sjøberg, D.I., *et al*.: 'A controlled experiment comparing the maintainability of programs designed with and without design patterns—a replication in a real programming environment,' *Empir. Softw. Eng.*, 2004, **9**, (3), pp. 149– 195

[9]. Ampatzoglou, A., Frantzeskou, G., Stamelos, I.: 'A methodology to assess the impact of design patterns on software quality,' *Inf. Softw. Technol.*, 2012, **54**, (4), pp. 331– 346

[10]. Nanthaamornphong, A., Carver, J.C.: 'Design patterns in software maintenance: an experiment replication at the University of Alabama.' Second Int. Workshop on Replication in Empirical Software Engineering Research (RESER), Banff, AB, Canada, September 2011, pp. 15– 24

[11]. Ali, M., Elish, M.O.: 'A comparative literature survey of design patterns impact on software quality.' Int. Conf. on Information Science and Applications (ICISA), Suwon, South Korea, June 2013, pp. 1– 7

[12]. Ampatzoglou, A., Charalampidou, S., Stamelos, I.: 'Research state of the art on GoF design patterns: a mapping study,' *J. Syst. Softw.*, 2013, **86**, (7), pp. 1945– 1964

[13]. Riaz, M., Breaux, T., Williams, L.: 'How have we evaluated software pattern application? A systematic mapping study of research design practices', *Inf. Softw. Technol.*, 2015, **65**, pp. 14– 38

[14]. Weiss, M.: 'Patterns and their impact on system concerns.' 13th Annual European Conf. on Pattern Languages of Programming (EuroPLoP), Irsee, Germany, July 2008, pp. S2-1–S2-10

[15]. Zhang, C., Budgen, D.: 'What do we know about the effectiveness of software design patterns?', *IEEE Trans. Softw. Eng.*, 2012, **38**, (5), pp. 1213– 1231

[16]. Mayvan, B.B., Rasoolzadegan, A., Yazdi, Z.G.: 'The state of the art on design patterns: A systematic mapping of the literature,' *J. Syst. Softw.*, 2017, **125**, pp. 93– 118

[17]. Kitchenham, B.: ' Procedures for performing systematic reviews.' TR/SE-0401, Keele University, Keele, UK, 2004

[18]. Jabangwe, R., Börstler, J., Šmite, D., *et al*.: 'Empirical evidence on the link between object-oriented measures and external quality attributes: a systematic literature review,' *Empir. Softw. Eng.*, 2015, **20**, (3), pp. 640– 693

[19]. Dyba, T., Dingsoyr, T., Hanssen, G.K.: 'Applying systematic reviews to diverse study types: an experience report.' First Int. Symp. on Empirical Software Engineering and Measurement (ESEM 2007), Madrid, Spain, September 2007, pp. 225– 234

[20]. Scopus: abstract and citation database of peer-reviewed literature. Elsevier, 2019. Available at https://www.elsevier.com/solutions/scopus

[21]. Aridor, Y., Lange, D.B.: 'Agent design patterns: elements of agent application design.' Proc. Second Int. Conf. on Autonomous Agents, Minneapolis, MN, USA, May 1998, pp. 108–115

[22]. Izurieta, C., Bieman, J.M.: 'How software designs decay: a pilot study of pattern evolution.' First Int. Symp. on Empirical Software Engineering and Measurement (ESEM) 2007, Madrid, Spain, September 2007, pp. 449– 451

[23]. Izurieta, C., Bieman, J.M.: 'A multiple case study of design pattern decay, grime, and rot in evolving software systems', *Softw. Qual. J.*, 2013, **21**, (2), pp. 289– 323

[24]. Kothari, C.R.: ' *Research methodology: methods and techniques* (New Age International, New Delhi, India, 2004)

[25]. Runeson, P., Höst, M.: 'Guidelines for conducting and reporting case study research in software engineering,' Empir. Softw. Eng., 2008, 14, (2), pp. 131– 164

[26]. Guéhéneuc, Y.G.: 'P-MARt: pattern-like micro architecture repository'. Proc. 1st EuroPLoP

Focus Group on Pattern Repositories, 2007, pp. 1– 3

[27]. Dong, J., Zhao, Y., Peng, T.: 'A review of design pattern mining techniques,' Int. J. Softw. Eng. Knowl. Eng., 2009, 19, (6), pp. 823– 855

[28]. Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., et al.: 'Design pattern detection using similarity scoring,' IEEE Trans. Softw. Eng., 2006, 32, (11), pp. 896– 909

[29]. Guéhéneuc, Y.G., Antoniol, G.: 'DeMIMA: a multilayered approach for design pattern identification,' IEEE Trans. Softw. Eng., 2008, 34, (5), pp. 667– 684

[30]. Guéhéneuc, Y.G.: 'Ptidej: a flexible reverse engineering tool suite'. IEEE Int. Conf. on Software Maintenance, 2007 (ICSM 2007), Paris, France, October 2007, pp. 529– 530

[31]. Shi, N., Olsson, R.A.: 'Reverse engineering of design patterns from Java source code.' 21st IEEE/ACM Int. Conf. on Automated Software Engineering (ASE'06), Tokyo, Japan, September 2006, pp. 123– 134

[32]. Wohlin, C., Höst, M., Henningsson, K.: ' Empirical research methods in software engineering in R. Conradi, A.I. Wang (Eds.): ' Empirical methods and studies in software engineering (Springer, Verlag, Berlin & Heidelberg, 2003), pp. 7– 23

[33]. Kitchenham, B.A., Pfleeger, S.L., Pickard, L.M., et al.: 'Preliminary guidelines for empirical research in software engineering,' IEEE Trans. Softw. Eng., 2002, 28, (8), pp. 721– 734

[34]. Robson, C.: ' Real world research: a resource for social scientists and practitioners– researchers' (John Wiley & Sons, Chichester, UK, 1993)

[35]. Forward, A., Lethbridge, T.C.: 'The relevance of software documentation, tools, and technologies: a survey.' Proc. 2002 ACM Symp. on Document Engineering, McLean, VA, USA, November 2002, pp. 26– 33

[36]. deSouza, S.C.B., Anquetil, N., de Oliveira, K.M.: 'A study of the documentation essential to software maintenance.' Proc. 23rd Annual Int. Conf. on Design of Communication: Documenting & Designing for Pervasive Information, Coventry, UK, September 2005, pp. 68– 75

[37]. C. Zhang and D. Budgen, "What do we know about the effectiveness of software design patterns," Transactions on Software Engineering, IEEE Computer Society, 38 (5), pp. 1213–1231, September - October 2012.

[38]. L. Prechelt, B. Unger, W. F. Tichy, P. Bressler, and L. G. Votta, "A controlled experiment in maintenance comparing design patterns to simpler solutions," Transactions on Software Engineering, IEEE Computer Society, 27 (12), pp. 1134–1144, December 2001.

[39]. M. Vokác, W. Tichy, D. I. K. Sjøberg, E. Arisholm, and M. Aldrin, "A controlled experiment comparing the maintainability of programs designed with and without design patterns: A replication in a real programming environment," Empirical Software Engineering, Springer, 9 (3), pp. 149–195, September 2004.

[40]. A. Ampatzoglou, S. Charalampidou, and I. Stamelos, "Research state of the art on GoF design patterns: A Mapping Study," Journal of Systems and Software, 86 (2), pp. 1945–1964, July 2013.

[41]. A. Ampatzoglou, G. Frantzeskou, and I. Stamelos, "A methodology to assess the impact of design patterns on software quality," Information and Software Technology, Elsevier, 54 (4), pp. 331–346, April 2012.

[42]. S. Yau and J. Collofello, "Design stability measures for software maintenance," Transactions on Software Engineering, IEEE Computer Society, 11 (9), pp. 849-856, September 1985.

[43]. R. K. Yin, "Case study research: Design and methods," 3rd Edition, Sage Publications, Los Angeles, California, USA, 2003.

[44]. J. M. Bieman, D. Jain, and H. J. Yang, "OO design patterns, design structure, and program changes: an industrial case study," 17th International Conference on Software Maintenance (ICSM'01), IEEE Computer Society, pp. 580–589, 7–9 November 2001, Italy.

[45]. J. M. Bieman, G. Straw, H. Wang, P. W. Munger, and R. T. Alexander, "Design patterns and change proneness: an examination of five evolving systems," 9th International Software Metrics Symposium (METRICS'03), IEEE Computer Society, pp. 40–49, 3-5 September 2003, Sydney, Australi

[46]. M. Di Penta, L. Cerulo, Y.-G. Guéhéneuc, and G. Antoniol, "An empirical study of relationships between design pattern roles

and class change proneness," 24th International Conference on Software Maintenance (ICSM'08), IEEE Computer Society, pp. 217–226, 28 September–4 October 2008, Beijing, China.

**[47].** L. Aversano, G. Canfora, L. Cerulo, C. Del Grosso, and M. Di Penta, "An empirical study on the evolution of design patterns," 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE '07), ACM, pp. 385-394, 3-7 September 2007, Dubrovnik, Croatia