Design Patterns Impact on Software Quality: Where Are the Theories?

Foutse Khomh
Department of Computer and Software Engineering
Polytechnique Montréal
Montréal, Canada
foutse.khomh@polymtl.ca

Yann-Gaël Guéhéneuc
Department of Computer Science and Software Engineering
Concordia University
Montréal, Canada
yann-gael.gueheneuc@concordia.ca

***Abstract*—Software engineers are creators of habits. During software development, they follow again and again the same patterns when architecting, designing and implementing programs. Alexander introduced such patterns in architecture in 1974 and, 20 years later, they made their way in software development thanks to the work of Gamma et al. Software design patterns were promoted to make the design of programs more "flexible, modular, reusable, and understandable". However, ten years later, these patterns, their roles, and their impact on software quality were not fully understood. We then set out to study the impact of design patterns on different quality attributes and published a paper entitled "Do Design Patterns Impact Software Quality Positively?" in the proceedings of the 12^{th} European Conference on Software Maintenance and Reengineering (CSMR) in 2008. Ten years later, this paper received the Most Influential Paper award at the 25^{th} International Conference on Software Analysis, Evolution, and Reengineering (SANER) in 2018.**

In this retrospective paper for the award, we report and reflect on our and others' studies on the impact of design patterns, discussing some key findings reported about design patterns. We also take a step back from these studies and re-examine the role that design patterns should play in software development. Finally, we outline some avenues for future research work on design patterns, e.g., the identification of the patterns really used by developers, the theories explaining the impact of patterns, or their use to raise the abstraction level of programming languages.

***Index Terms*—Design patterns, software quality, quantitative studies, qualitative studies, retrospective.**

I. Introduction

Developers like most human beings are creatures and creators of habit. When developing software systems, they follow the same patterns again and again and reuse the same solutions for similar problems. During development and maintenance activities, they also look for occurrences of patterns in software artifacts, as telltales of other developers' architecture, design, and implementation choices. Yet, these patterns, their roles, and their impact on software quality are not fully understood.

There is an abundant literature on habits and patterns in human behavior, e.g., [1, 2]. Without going into the interesting debate between Cartesians and pragmatists [2], we point out that several theories have been formulated [1] in attempts to explain the formation, use, and changes of habits and patterns in human behavior as well as the dangers and–or usefulness of changes in these habits and patterns, e.g., the Theory of Reasoned Action [3], the Social Cognitive Theory [4], the Theory of Interpersonal Behavior [5], or the Theory of Planned Behavior [6].

Alexander introduced such architectural patterns in architecture in 1974 [7] and, 20 years later, they made their way in software engineering thanks to the work of Gamma et al. [8]. In software engineering, design patterns encode "good practices" guiding developers in solving recurring design problems. They were promoted to make designs more "flexible, modular, reusable, and understandable". Guéhéneuc [9] has proposed a theory that try to explain how developers rely on patterns to gather knowledge about programs. This theory and other anecdotal evidence support the importance of encoding patterns, whether programming language features, recurring programming idioms, or patterns of developers' usages of their IDEs. They also support the roles of these patterns for providing a common vocabulary among software engineers and for teaching software engineers best practices [10, 11]. Thus, they support patterns as improving the overall quality of software development and software systems.

However, at the time of publication of our paper entitled "Do Design Patterns Impact Software Quality Positively?" [12] in 2008, the impact of patterns on software quality was little researched and not fully understood. Few studies had investigated the role of design patterns in software development activities and there were hints that some patterns may actually be harmful to software quality [13]. Consequently, we surveyed the research community to understand experts' perception on design patterns and their impact on different quality attributes: the four attributes put forward by Gamma et al. [8], i.e., flexibility (expandability), modularity, reusability, and understandability, and six other interesting attributes: simplicity, learnability, generality, modularity at runtime, scalability, and robustness.

Our survey showed that experts consider that design patterns do not always improve quality and that some design patterns have particularly negative impact on the attributes highlighted by Gamma et al. This paper received the Most Influential Paper award at the 25^{th} International Conference on Software Analysis, Evolution, and Reengineering (SANER) in 2018 "for its pioneering examination of the relation between design patterns and code quality". Since the publication of our survey,

978-1-5386-4969-5/18/$31.00 © 2018 IEEE

researchers have been intensively investigating the impact of design patterns on the quality of software processes and systems. For example Garzás et al. [14] investigated whether design patterns can ease understanding and changing UML designs while Hegedűs et al. [15] examined the impact of design patterns on maintainability through an analysis of more than 300 versions of JHotdraw—a framework whose design uses many design patterns—and reported correlations between numbers of design patterns instances and maintainability.

To stimulate both industry practitioners and researchers to reflect on the impact of patterns on software development activities, we co-founded the Patterns Promotion and Anti-patterns Prevention (PPAP) workshop series in 2013. The workshop ran for four years and served as a forum for researchers to discuss advances and problems in understanding the impact of patterns on quality. It brought together up to 20 participants and culminated in a card sorting exercise at PPAP'16, co-located with SANER'16, that highlighted three levels of problems met by the community:

1) The need for a strong(er) community, possibly with special issues in journals dedicated to patterns, (more) workshop and conference series focusing on patterns, and on-line vetted activities and resources.
2) The need to bridge research activities with industrial practice, in particular to promote scientifically-validated results to the practitioners and to study the impact of the practices concretely used by practitioners.
3) The need for support in teaching patterns, in particular to students with little experience and with stringer time constraints, through reviewing code with patterns, summer schools, and the sharing of teaching material.

In this retrospective paper, we reflect about these previous studies, events, and their results and take a step back from these studies on patterns and re-examine the impact of patterns on software development activities. This paper is not a systematic review of the literature on patterns but a discussion of key works that contribute to our understanding of patterns. After choosing, discussing, and summarising such works on design patterns in Section II and III, we introduce the many other types of patterns that exist in the literature in Section IV. We name and define these patterns and provide seminal references. The diversity of types of patterns supports the observations that developers are creatures of habits and that theories from other research fields apply to software engineers. It also highlights the importance of systematically and thoroughly studying these patterns to assess their impact on software development activities in general and software quality in particular. We discuss our lack of understanding of the unifying principles supporting patterns in Section V. Finally, we suggest that the research community should promote best practices in the study of patterns. We outline some avenues for future research on patterns in Sections VI and VII.

II. Methodology to Identify Works

To survey the literature on design patterns and identify the different themes addressed by the research community, we first must reconcile the terminology used by researchers when reporting studies about design patterns. We performed a domain analysis [16] as follows:

1) Identify collections of papers related to design patterns.
2) Choose one such collection and download all papers.
3) Read all papers and identify design pattern-related terms.
4) Compile all terms including synonyms and homonyms.
5) For each occurrence of a term, identify its synonyms and homonyms and retain the most significant term.
6) Define the retained terms.
7) Map all the terms with the retained terms.

This process is similar to that of reverse engineering programs written in different languages into a common meta-model. The programs are the papers in the literature and the programming languages are the terms related to design patterns used in each paper. The common meta-model is the set of retained terms. In contrast to the usual reverse engineering process, in which a "language-independent" meta-model is built and the different programs map into it, we do not define such a meta-model because it would only be a model of the English language[1], with which all terms are expressed and defined. We simply define a model (a set of terms) and map the set of all existing terms into this set.

The result of this process is a set of terms and a mapping. The mapping takes the form of a table with all existing terms as columns and retained terms as rows. Cells include references to papers that use the terms with the definitions of the retained terms. The terms and their mapping provide a useful background for the co-understanding of research papers on design patterns.

The remaining of this section provides more details about the different steps of our domain analysis.

Steps 1, 2, and 3: Identification of the Terms: The first three steps of the process are used to identify all terms related to design patterns. In the first step, we identify two main collections of papers on design patterns, available on-line[2,3]. These two collections mostly overlap and are representative of other such collections. One of the authors contributed to the second library with more than 60 papers. Therefore, we choose this second collection for the sake of simplicity.

The chosen collection contains 99 references. We download 59 papers in PDF or HTML format, excluding books because of their restricted availability and of their breadth and some papers because of the unsuitability of their encoding for search[4]. In these 59 papers[5], we identify all terms related to design patterns and store them with their associated references in a table. We convert all terms into singular nouns in British English for the sake of homogeneity. We leave out

[1]We only study papers written in English because it is the language used in the main conferences in software engineering.

[2]http://liinwww.ira.uka.de/bibliography/SE/patterns.html

[3]http://www.patternforge.net/wiki/index.php?title=Papers_on_Pattern_Repositories

[4]The PDF format allows papers to be encoded as images that are not suitable for search.

[5] The complete list of studied papers is available at http://www.ptidej.net/downloads/replications/saner18mipa/.

some terms that either are synonyms of others, for example "Definition" is used as a synonym of "Formalisation", or that have little impact on the mapping, for example "Detection" vs. "Identification", or that have agreed-upon definitions, such as "Role".

Steps 4, 5, and 6: Definitions of the Terms: The Steps 4 to 6 of the process concern the sorting and definitions of the retained terms. First, we study again all papers and associated each found term with each paper that uses the term disregarding its definition. Table I presents in its left column the 42 terms related and the references to their defining papers.

Then, we group terms that we understood as having similar definition. With this grouping, we retain 16 terms. Table I presents in bold the retained terms and their main synonyms. We do not show here homonyms because we study them in Step 7. We base our choice of the retained term both on the "popularity" of a term (the number of papers using it) and on its "trend" in recent literature. Therefore, this choice can be at times arbitrary when more than one term are popular or recent. Yet, Table I is useful because it can help the community in making an informed choice of the terms to be retained for each definition.

Step 7: Mapping Between All Terms and Retained Terms: The last step consists in mapping the terms in the literature with those retained in the previous steps. It brings a common background to understand the papers. Table II shows all terms of the literature in columns and retained terms in rows. Thus, a column shows the homonyms of a term and a row shows the synonyms of a retained term.

Table II shows some interesting facts. First, several terms are homonyms, being used repeatedly in the literature with different definitions. This is particularly true with the term "Design pattern", which has been used to mean "Design motif", "Design pattern", "Idiom", "Instance", "Occurrence", and "Solution". This wide use of the term "Design pattern" is not surprising given the ubiquitousness of this concept but its loose definition. Previous authors often implicitly use the term "Design pattern" with different definitions because the extent of their work was not fully understood at the time.

Second, different terms have been used for one definition by the same authors. For example, in Eden's paper [23], the terms "Design motif" and "Formalisation" are used to mean "Design motif". Although it is rarely the case in the studied papers, such a synonymy could lead to confusion in the understanding of the work if read without care.

Finally, Table II shows that some more "specialised" terms, such as those characterising design patterns, "Fundamental patterns" or "Hybrid patterns", are less subject to having homonyms and synonyms because they have been more precisely defined from the start by their authors. From Table II, we can also observe that some terms have been used only during a short period of time, such as "Leitmotiv" while others have been more prevalent, such as "Formalisation" for example. This table highlights the problem of vocabulary faced by the community and allows for a better understanding of the literature about design patterns, using a common set of terms. Using the retained terms, we now examine the content of the papers to identify recurring usages and themes addressed by the research community with design patterns.

III. Design Patterns in Software Development

Overall, we identified seven main themes: knowledge sharing, development tools, formalisation, forward engineering, reverse engineering, documentation, and quality. We found many papers discussing more than one of these themes.

A. Sharing Knowledge with Design Patterns

Design patterns are powerful knowledge-sharing tools because they encapsulate developers' experience and provide a common vocabulary for communication across domains. However, an excessive use of design patterns is likely to result in designs in which it is difficult to recognise the structure of the participating design patterns. This problem is known as the tracing problem. Agerbo and Cornils [38] analyzed the use of design patterns in programs and proposed the idea of a design-pattern library to solve the problem of tracing design patterns in systems.

We concur that it is important to keep track of existing patterns and suggest to put in place a repository and a vetting process of patterns that have been shown experimentally to benefit software engineers and systems. Such a repository and vetting process would help educate software engineers in using appropriate patterns and also ensuring that patterns continue to form a consistent vocabulary. Such a repository could also alleviate the risks (also put forward by Agerbo and Cornils) of over-using design patterns in systems and of having too many patterns to choose from. In fact, Agerbo and Cornils [38] argued that if patterns grow in large numbers, it will be increasingly difficult for software engineers to grasp them all and they will not form a common vocabulary anymore. This difficulty is especially visible with the many different types of patterns now being used and studied, shown in Section IV.

B. Development with Design Patterns

Design patterns can be used as development tools in different software development activities:

1) Design: identify a problem and propose a solution.
2) Implementation: instantiate a design motif.
3) Maintenance: document and suggest refactorings.

The benefits of design patterns during software development have been the subject of many papers. Beck [18] suggests that patterns generate architectures. Similarly, Ram et al. [55] propose the *Pattern Oriented Technique* for developing systems via patterns as collaborations of design patterns. Also, during the maintenance of systems, software engineers often identify code and design smells and look for refactorings or patterns to remove them. These refactoring and pattern transformations are documented to be reused in the future [56]. Lange and Nakamura [57] demonstrate that patterns can serve as guide in program exploration and thus make the process of program understanding more efficient. Through a trail of pattern execution, they show that if patterns were recognized

TABLE I
RETAINED TERMS RELATED TO DESIGN PATTERNS AND THEIR DEFINITIONS. TERMS ARE GROUPED BY SETS OF SYNONYMS, SEE TABLE II FOR THE MAPPING AMONG TERMS.

Terms	Definitions
Abstract Design Pattern [17], Basic Form [18], **Design Motif** [19], Design Template [20], Essence [21], Expression [22], Formalisation [23], Implementation [24], Lattice [23], Leitmotiv [25], Logic [21], Realisation [26], Reification [24], Representation [27], Specification [23], and Structure [8]	A design motif is the prototype proposed by a design pattern to solve the related recurring design problem. Typically, a design motif is built from the "Structure" section of the design pattern as defined in the GoF, including elements from other relevant sections to characterise the participants and their responsibilities [19].
Alternative [20], Modified Pattern [28], **Variant** [21], Version [20]	A design motif (see previous definition) manifesting variety, deviation of its canonical form as described in [8].
Canonical form [20]	Original design motif as described in [8]
Clich [29], **Idiom** [30], Micropattern [31]	A standard algorithmic fragment [29].
Design Component [32], **Instance** [23], Microarchitecture [33]	An instance of a design motif is the concrete implementation of the solution of the pattern in a program. A micro-architecture is deemed an instance of a design motif if it can be asserted that the developers indeed chose conscientiously and appropriately to implement the solution of the design pattern to solve the corresponding design problem [19].
Design Pattern [8]	A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented designs. It identifies the participating classes and their instances, their roles and collaborations, and the distribution of responsibilities. Each design pattern focuses on a particular object-oriented recurring design problem. It describes when its solution applies, whether or not its solution can be applied in view of other design constraints, and the consequences and trade-offs of its use. It also provides sample code to illustrate an implementation. [8]
Elemental Pattern [34], Fragments [23], Minipattern [35], **Subpattern** [36]	Abstract syntax graph structures [37] that are constituent parts of other patterns or subpatterns. An idiom could be a subpattern while subpatterns are not necessarily idioms.
Fundamental Pattern [38]	A core of design patterns that capture good object-oriented design and that can be used in various contexts [38].
Hybrid Pattern [39]	Pattern generated through hybridization and whose intent is to solve a high level design problem in a generic context [39].
Intent [8]	A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address? [8]
Language of Patterns [40]	A structured method of describing good design practices within a field of expertise [36].
Occurrence [20]	An occurrence of a design motif is the concrete implementation of the solution of the pattern in a program. However, in the contrary to an instance, it cannot be asserted that the design motif was used with purpose by the developers, possibly because several micro-architectures could conform to the structure of the design motif but without conforming to the intent of the design pattern [23].
Protopattern [40]	"patterns in waiting" that are not yet known to recur [41].
Secondary Role [42]	Role that can be superimposed over the defining role of a class in a program [42].
Solution [8]	The solution of a design pattern is the description composed of the "Structure", "Participants", and "Collaborations" sections, as defined in the format in [8].
Trick [31]	A trick is an operator specifying the sequence of steps taken in the realization of a design motif [31].

TABLE II
MAPPING OF THE TERMS RELATED TO DESIGN PATTERNS. WE ONLY PRESENT A MAXIMUM OF 2 REFERENCES PER CELL BECAUSE OF A LACK OF SPACE. THE COMPLETE TABLE IS AVAILABLE ON-LINE[5].

Retained Terms	Abstract Design Pattern	Alternative	Basic Form	Canonical Form	Cliché	Design Component	Design Motif	Design Pattern	Design Template	Elemental Pattern	Essence	Expression	Formalisation	Fragment	Fundamental Pattern	Hybrid Pattern	Idiom	Implementation	Instance	Intent	Language of Patterns
Canonical form			[28]	**[43, 20]**						[28]											
Design Motif	[44]				[33]		**[45, 23]**	[21, 20]	[46, 47]	[48]	[25]	[22]	[49, 23]					[22, 50]			
Design Pattern							[22]	**[21, 20]**													
Fundamental Pattern															**[27, 20]**						
Hybrid Pattern																**[39]**					
Idiom			[51]		[33, 20]		[31]	[51]						[23]			**[18, 30]**				
Instance						[24]		[43, 50]										[21, 20]	**[47, 23]**		
Intent											[24]									**[21, 20]**	
Language of Patterns																					**[36, 40]**
Occurrence								[51, 32]										[44, 46]	[32, 33]		
Protopattern																					
Role																					
Secondary Role																					
Solution								[39, 52]													
Subpattern			[18, 30]			[32]	[31]			[34]				[23]			[40]				
Trick																					
Variant		[21, 20]																[43, 51]			

Retained Terms	Lattice	Leitmotiv	Logic	Microarchitecture	Micropattern	Minipattern	Modified Pattern	Occurrence	Protopattern	Realisation	Reification	Representation	Secondary Role	Solution	Specification	Structure	Subpattern	Trick	Variant	Version
Canonical form																				
Design Motif	[23, 31]		[42]	[33, 20]	[31]	[53, 45]		[25]		[53]		[50, 40]		[32, 33]	[27, 23]	[21, 20]				
Design Pattern														[42, 38]						
Fundamental Pattern																				
Hybrid Pattern																				
Idiom					[23, 31]				[40]											
Instance								[25]			[20]			[54]						
Intent																				
Language of Patterns																				
Occurrence								**[23, 20]**		[28, 26]				[54]						
Protopattern									**[44, 40]**											
Role																				
Secondary Role													**[42]**							
Solution														**[21, 20]**		[38]				
Subpattern					[23, 31]				[24]								**[38, 36]**			
Trick																		**[23, 31]**		
Variant							[28]												**[30, 21]**	[49, 18]

at a certain point in the understanding process, they could help in "filling in the blanks" and in further exploring a system, improving thus its understandability. Other papers focus on design patterns during development, e.g., [50, 58, 59].

Other papers focus on the selection of design patterns and attempt to draw rules to combine these patterns during development. Guéhéneuc et al. [60] present a recommender system to help software engineers in choosing among the 23 design patterns by Gamma et al. Ram et al. [39, 55] propose rules for the combination and application of design patterns. Despite these studies, there is no languages of design patterns, i.e., pattern languages and their supporting tooling. The processes of using design patterns during development is also the topic of some papers [50, 58, 59].

C. Formalisation of Design Patterns

The formalisation relates to approaches that propose a specification of and specify design patterns (or parts thereof) and verify their implementations. Such approaches include specification languages and tools for the selection and introduction of design patterns in systems. They prevent implementations inconsistent with the intents of the patterns, which would void the benefits of these patterns.

Despite their importance, few specification languages are reported in the literature. LePUS [61] is a full-fledged logic language with a well-defined semantics. It has a compact vocabulary and can represent regularities and relations among classes, functions, and inheritance hierarchies. Similarly, Hedin [62]

presents an approach based on attribute grammars for formalising design patterns, which provides attributes extensions describing conventions by declarative semantic rules. Smith et al. also proposed a formalisation of design patterns based on the ς-calculus and implemented in the SPQR tool to model and identify regularities in systems designs [34]. However, these specification languages are seldom used in research and practice and no tooling in popular integrated development environments supports these languages.

D. Forward Engineering with Design Patterns

Design patterns are useful for the design of systems. Many studies have focused on design patterns in the context of forward engineering. We divide this theme in two sub-themes: code generation and language.

a) Code generation: With the growing interest in Model Driven Engineering [63], design patterns could be the bridge between the designs of systems and the automatic generation of their source code. Many techniques of code generation exists: Mens [64] proposes a taxonomy for the classification of model-transformation techniques and discusses their applicability. Budinsky [21] describes the architecture and implementation of a tool that automates the implementation of design patterns. The user of the tool supplies application-specific information for a given pattern, from which the tool generates all the pattern-prescribed code automatically. Soukup [59] explores the problems and possibilities of automated pattern implementation. He identifies three basic problems: the loss of the patterns during implementation, large clusters of mutually dependent classes caused by the use of multiple patterns, and the lack of a library of concrete reusable patterns. Another paper on the topic is the work by Agerbo and Cornils [38].

b) Languages: Syntax extensions and extended language constructs have been explored to simplify the specification of design patterns and to improve the readability of systems. Tatsubori et al. [65] show that compile-time MOPs provide a general framework to implement design patterns. They show that software engineers can use a MOP to create a library of reusable patterns based on syntax extensions and extended language constructs. Similarly, Chambers [58] also proposes an interesting contribution on the topic. Bosch [50] identifies four major problems associated with the implementation of design patterns using conventional object-oriented programming languages. He solves these problems with a layered object model, LayOM. LayOM is an object-oriented language that provides explicit support for design patterns. It is extensible with abstractions for other patterns.

E. Reverse Engineering with Design Patterns

There are two major trends in this theme: design pattern detection and pattern-based components recovery, which we consider here as sub-themes. Since their inception, it has been suggested that design patterns could play a central role in reverse engineering, e.g., the detection of occurrences of design patterns in systems could improve their understanding [47]. Occurrences would also improve the documentation. During the reverse engineering of systems, design patterns are also use to refactor code smells and design defects "away from" the code [56]. Many papers have been published in this sub-theme, e.g., [19, 66, 34, 28]. On the sub-theme of pattern-based component recovery, we can cite the work by Keller et al. [24].

F. Documenting with Design Patterns

This theme includes papers studying the role of design patterns in the documentation of systems. Many papers promote the idea that documenting patterns helps in understanding a design and thus in easing maintenance because developers often attempt to recover non-documented design patterns in systems. They illustrate how a combination of patterns allows to simply convey the essence of designs, e.g., in JUnit. They put forward the impact of design patterns on documentation and in the recovery of design information for revere- and re-engineering. Again, Lange and Nakamura [57] show that design patterns can guide exploration and ease understanding. The approach presented by Hedin [62] also discusses the identification of design patterns for the documentation of systems. Soukup [59] proposes an approach to build a library of reusable common patterns. Other interesting approaches exist related to documenting with design patterns [38, 50].

G. Impact of Design Patterns on Quality

Ampatzoglou et al. [67] examine the stability of classes implementing some design patterns. They report that classes playing exactly one role in a design pattern are more stable than classes playing zero or more than one role. This result corroborates findings by Khomh et al. [68], which show that playing more than one role in design patterns makes classes more change-prone. Khomh et al. [68] also observed that classes playing more than one role in design patterns are less cohesive, more coupled, more complex, and more issue-prone than playing only one role in a design pattern.

Ampatzoglou and Chatzigeorgiou [69] also examined the impact of State, Strategy, and Bridge design pattern on code quality and found them to improve cohesion, and reduce coupling, and complexity. Ali and Elish [70] reviewed the literature on the impact of design patterns on software quality and report that design patterns negatively impact maintainability, evolution, and change-proneness. The surveyed studies disagreed on the impact of design patterns on fault-proneness.

Khomh [71] examine co-occurrences of code smells and design patterns in systems and observe that the negative effect of code smells on change-proneness can be mitigated (and even reversed) by some design patterns.

IV. Types of Patterns

Before the introduction of design patterns in software engineering, there existed already few different types of patterns, in particular idioms of programming [72], but patterns really became mainstream with the book of Gamma et al. [8]. Since then, many other different types of patterns were defined and studied by the research community. In the following we

attempt to exhaustively list and succinctly define these different types. We consider only programming-related patterns and, thus, exclude management [73], reengineering [74], etc. patterns, which relate more to software development processes.

- Design patterns introduced by Gamma et al. [8] describe and name common design problems and their solutions in object-oriented programming.
- Meta-patterns defined by Pree [75] are reusable object-oriented designs using a domain-independent terminology and notation.
- JNI idioms [76] capture common interactions between Java and C/C++ code.
- Exception-handling idioms [77] report good practices in handling exception, in particular in Java.
- Evolution patterns (commits, mutations) introduced by Kpodjedo et al. [78] describe patterns in the evolution of systems.
- Developers patterns made popular by Mylyn [79] describe developers' behaviour captured in their IDEs.
- File editing patterns [80] capture developers' file edits during source-code changes.
- Programming languages features are a form of pattern, idioms really, that solve particular programming problems, e.g., the try-with-resources idiom.
- Patterns of API usages [81] put together and make explicit systematically sets of method invocations and their parameters values for given APIs.
- EJB patterns and other such specialised patterns are complementary to patterns of API usages and describe design and architectural choices with specific frameworks.
- Patterns of logging [82] describe where, when, and what to log in systems to provide useful information.
- Micro-patterns introduced by Gil and Maman [83] are design decision easily identified automatically in systems.
- Patterns of inheritance were described by Denier et al. [84] to capture common practices in object-oriented programming related to inheritance and identify violations.
- Patterns of rules introduced by Wuyts et al. [85] are design constraints embodied and enforced using SOUL to enforce design "invariants".
- Cloud patterns [86, 87] describe good practices in implementing Cloud-based systems.
- Architectural patterns [88] are solutions to recurring problems when designing architectures.
- Debugging patterns formalised by Petrillo et al. [89] describe developers' behaviour when debugging systems.
- UML artifacts can be partly recovered from source code when defined and specified as patterns [90].
- Test patterns as first described by Firesmith [91] are good practices when testing systems.

We argue that the research community should follow examples set in other research fields and systematically define, classify, categorise, and relate patterns. We need software-pattern taxonomists[6] to create a unified taxonomy of patterns. Also, these taxonomists should identify pattern languages, if any, that would help developers "discourse" about their systems using patterns and, thus, make the best choices in their usages of patterns and their impacts on quality.

[6]https://en.wikipedia.org/wiki/Taxonomy_(biology)

V. Unifying Principles and Characteristics

We now claim that most design motifs put forward as solutions to design problems by design patterns share the same underlying principles: they introduce new classes and–or methods to add one or more levels of indirection and, through these levels of indirection, provide more flexibility. We describe two examples now succinctly to support our claim.

The Visitor design pattern intends to "[r]epresent an operation to be performed on elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates" [8, page 331]. The solution advocated by the Visitor design pattern is to introduce a hierarchy of visitor classes, which conform to a well-defined interface and whose methods are called by new methods declared into the hierarchy of original objects. Hence, the solution suggests adding one level of indirection from the original methods performing the operations, declared in the original objects, to the methods performing the operations declared in the visitors classes and called by novel methods in the original objects.

Similarly, the Observer design pattern intends to define "a one-to-many dependency between objects" [8, page 293] by describing the behaviour of observers in dedicated classes whose notification methods are called by a subject rather than implementing this behaviour directly into the subject. Hence, the solution also suggests adding one level of indirection from the original subject and behaviour of the observers.

However, we also claim that not all design patterns are based on this one principle of adding one level of indirection to increase flexibility. For example, the Composite design patterns has for design principle a cycle between a set inheritance relationships and a composition relationship. These examples show that the underlying principles of design patterns are currently ill-defined even though their definitions would have two major benefits:

1) The systematic formalisation and categorisation of current design patterns based on their underlying principles, e.g., to help teaching.
2) The systematic combinations of these underlying principles to identify novel design patterns from first principles rather than through experiences.

In addition, other types of patterns also do not have clear underlying principles and this lack of principles prevents a thorough discussions of these patterns and studies of their impacts on software quality. Moreover, although object-oriented programming languages are well-established, popular, and have been successful in practice, this lack of principles prevents the cross-fertilisation of patterns in different programming paradigms, e.g., functional programming, different designs, e.g., design by dependency injection, and different architecture styles, e.g., service-oriented architecture. Besides for-

malisation, categorisation, and cross-fertilisation, studying the principles underlying patterns could allow unifying patterns withing and across abstraction levels (implementation, design, architecture) and paradigms (object-oriented, functional).

Also, having unifying, underlying principles could help enforcing patterns implemented in systems. Although tools exist to enforce patterns, e.g., Wuyts' SOUL, these tools require the explicit definitions of the patterns being enforced. Similarly, unifying, underlying principles would help devising tools that generate implementation of patterns and–or refactor code to implement patterns.

Finally, studying the principles of patterns could also provide evidence for or against certain practices being "patterns". As suggested by Alexander and taken by Gamma et al. a problem and its solution must have been encountered in three different contexts, at least, before being afforded the name of "pattern". Thus, some current practices called patterns are not "real" patterns. Conversely, solutions to common problems, even if recurring in different contexts, are not "patterns" per se. For example, the programming idiom of iterating with an Iterator through a list is common in many programming languages and, as such, is a quintessential part of the languages rather than a pattern. Moreover, some patterns exist only to overcome limitations in programming languages, e.g., the try-with-resources feature in Java vs. destructors.

VI. Suggested Agenda

The previous sections summarised the tremendous advances that the research community has made in the past years in its study and understanding of design patterns in particular and different types of patterns in general. These sections also highlighted problems that the community is now facing.

The many excellent research studies on all types of patterns make it difficult—because of their sheer numbers and differing methodologies, subjects, objects, and objectives—for researchers, students, and practitioners to understand the differences and commonalities between patterns, to assess whether some results for some types of patterns are sound while similar results for others may be less so, and to identify and reproduce best practices and studies from some types of patterns to others. This diverse range of experimental designs yielded interesting results about patterns but makes difficult comparisons among experimental designs and among patterns and elusive understanding globally their impacts.

Therefore, we make the following suggestions to the research community interested in patterns.

a) Patterns from Developers' Behaviour: As developers are creatures of habits, patterns are most useful to developers during the development and evolution (comprehension) of their systems. We suggest that the community pursues research works on the direct impacts of patterns on developers, e.g., through the use of eye-trackers in experimental designs. Eye-trackers are now affordable and more and more popular in various research fields and could be used systematically to study patterns and their impacts on developers.

b) Patterns of Developers' Behaviour: Patterns may be used as building blocks of systems and also as templates against which to assess the code written by developers and their software development activities in their IDEs and with which to recommend patterns to follow. Hence, we suggest that the research community pursues research works on the analyses of developers' logs collected through, e.g., Mylyn [79] or FeedBaG [92], and on the usages of patterns to identify inefficient activities and help developers.

c) Patterns for Building Systems: Rich and Waters [29] pioneered the idea of patterns (idioms in their research works) as building blocks for systems. Few works followed their steps and this line of research seemed to have been mostly abandoned. We suggest to raise the levels of abstraction of programming languages with patterns to promote appropriate patterns as programming-language building blocks. For example, Oracle introduced in Java 7 the concept of Automatic Resource Management via try-with-resources blocks to implement and, thus, replace a common idiom.

d) Theories of Software Patterns: Based on existing works and possible future work, we suggest that the research community seizes the opportunities given by its research results to propose a theory—or competing theories—of patterns. Such theories should explain why certain types of patterns impact positively quality while others do not and also explain why certain types of patterns exist for certain software artifacts. Such a theory could then be used to frame experimental designs and identify novel patterns and their impacts.

VII. Conclusion

Ten years ago, we surveyed the research community to understand experts' perception on design patterns and their impact on different quality attributes: the four attributes put forward by Gamma et al., flexibility (expandability), modularity, reusability, and understandability, and six other attributes: simplicity, learnability, generality, modularity at runtime, scalability, and robustness. We showed that experts consider that design patterns do not always improve quality and that some design patterns have particularly negative impacts on the attributes highlighted by Gamma et al.

After this survey, other studies appeared that, essentially, followed three (non-mutually exclusive) research directions: (1) further studies of the impacts of design patterns on quality, e.g., the article by Zhang and Budgen "What do we know about the effectiveness of software design patterns?" [93], (2) studies of the definition and impact of other types of patterns, e.g., the article by Di Penta et al. "When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away)" [94], and (3) studies to enhance the identification of occurrences of patterns, e.g., "Code Smell Severity Classification Using Machine Learning Techniques" [95].

We argued that these studies tremendously improved our understanding in breadth and in depth of patterns. We then recommended that the research community builds upon these studies to study further patterns and their impact on quality, in particular to build a theory of patterns and explore research

directions that have been less explored, e.g., using patterns as building blocks of systems.

ACKNOWLEDGMENT

The authors would like to thank again the participants to the original survey, in 2007. We would like also to thank the research community for its appreciation of our study and the Most Influential Paper Award committee at SANER 2018 for the honour of choosing our paper.

REFERENCES

[1] G. Godin, A. Bélanger-Gravel, M. Eccles, and J. Grimshaw, "Healthcare professionals' intentions and behaviours: A systematic review of studies based on social cognitive theories," *Implementation Science*, vol. 3, no. 1, p. 36, 2008.

[2] E. Kilpinen, *Human Beings as Creatures of Habit*. The name of the publisher, 2012, vol. 12, ch. 5, pp. 45–69, cOLLeGIUM: Studies across Disciplines in the Humanities and Social Sciences 12.

[3] M. Fishbein and I. Ajzen, *Belief, Attitude, Intention, and Behavior: An Introduction to Theory and Research.* Reading, MA: Addison-Wesley, 1975. [Online]. Available: http://people.umass.edu/aizen/f&a1975.html

[4] A. Bandura, "Self-efficacy: Toward a unifying theory of behavioral change," *Advances in Behaviour Research and Therapy*, vol. 1, no. 4, pp. 139 – 161, 1978, perceived Self-Efficacy: Analyses of Bandura's Theory of Behavioural Change. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0146640278900024

[5] H. C. Triandis, "Values, attitudes, and interpersonal behavior." in *Nebraska symposium on motivation*. University of Nebraska Press, 1979.

[6] I. Ajzen and T. J. Madden, "Prediction of goal-directed behavior-attitudes, intentions, and perceived behavioral-control," *Journal of experimental social psychology*, vol. 22, no. 5, pp. 453–474, 1986.

[7] C. Alexander, "The origins of pattern theory: The future of the theory, and the generation of a living world," *IEEE Software*, vol. 16, no. 5, pp. 71–82, September/October 1999.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, 1^{st} ed. Addison-Wesley, 1994.

[9] Y.-G. Guéhéneuc, "A theory of program comprehension—joining vision science and program comprehension," *International Journal of Software Science and Computational Intelligence (IJSSCI)*, vol. 1, no. 2, pp. 54–72, April-June 2009, 18 pages. [Online]. Available: http://www.ptidej.net/publications/documents/JSSCI09.doc.pdf

[10] T. H. Ng, S. C. Cheung, W. K. Chan, and Y. T. Yu, "Do maintainers utilize deployed design patterns effectively?" in *29th International Conference on Software Engineering (ICSE'07)*, May 2007, pp. 168–177.

[11] P. C. Lotlikar and R. Wagh, "Using pogil to teach and learn design patterns — a constructionist based incremental, collaborative approach," in *2016 IEEE Eighth International Conference on Technology for Education (T4E)*, Dec 2016, pp. 46–49.

[12] Foutse Khomh and Y.-G. Guéhéneuc, "Do design patterns impact software quality positively?" in *Proceedings of the* 12^{th} *Conference on Software Maintenance and Reengineering (CSMR)*, C. Tjortjis and A. Winter, Eds. IEEE CS Press, April 2008, pp. 274–278, short Paper. 5 pages. [Online]. Available: http://www.ptidej.net/publications/documents/CSMR08.doc.pdf

[13] P. Wendorff, "Assessment of design patterns during software reengineering: Lessons learned from a large commercial project," in *Proceedings of* 5^{th} *Conference on Software Maintenance and Reengineering*, P. Sousa and J. Ebert, Eds. IEEE Computer Society Press, March 2001, pp. 77–84. [Online]. Available: http://www.computer.org/proceedings/csmr/1028/10280077abs.htm

[14] J. Garzás, F. García, and M. Piattini, "Do rules and patterns affect design maintainability?" *Journal of Computer Science and Technology*, vol. 24, no. 2, pp. 262–272, Mar 2009. [Online]. Available: https://doi.org/10.1007/s11390-009-9222-7

[15] P. Hegedűs, D. Bán, R. Ferenc, and T. Gyimóthy, "Myth or reality? analyzing the effect of design patterns on software maintainability," in *Computer Applications for Software Engineering, Disaster Recovery, and Business Continuity*, T.-h. Kim, C. Ramos, H.-k. Kim, A. Kiumi, S. Mohammed, and D. Ślezak, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 138–145.

[16] R. Prieto-Díaz, "Domain analysis: An introduction," *Software Engineering Notes*, vol. 15, no. 2, pp. 47–54, April 1990. [Online]. Available: http://portal.acm.org/citation.cfm?id=382296.382703

[17] I. Philippow, D. Streitferdt, M. Riebisch, and S. Naumann, "An approach for reverse engineering of design patterns," *Software and System Modeling*, vol. 4, no. 1, pp. 55–70, February 2005. [Online]. Available: http://www.springerlink.com/content/0dn4pmqh5uhnbk69/

[18] K. Beck and R. E. Johnson, "Patterns generate architectures," in *Proceedings of* 8^{th} *European Conference for Object-Oriented Programming*, M. Tokoro and R. Pareschi, Eds. Springer-Verlag, July 1994, pp. 139–149. [Online]. Available: http://citeseer.nj.nec.com/27318.html

[19] Y.-G. Guéhéneuc and G. Antoniol, "DeMIMA: A multi-layered framework for design pattern identification," *Transactions on Software Engineering (TSE)*, vol. 34, no. 5, pp. 667–684, September 2008, 18 pages. [Online]. Available: http://www.ptidej.net/publications/documents/TSE08.doc.pdf

[20] K. Brown, "Design reverse-engineering and automated design pattern detection in Smalltalk," Department of Computer Science, University of Illinois at Urbana-Champaign, Tech. Rep. TR-96-07, July 1996. [Online]. Available: http://citeseer.nj.nec.com/context/734211/0

[21] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu, "Automatic code generation from design patterns," *IBM Systems Journal*, vol. 35, no. 2, pp. 151–171, February 1996. [Online]. Available: http://www.research.ibm.com/journal/sj35-2.html

[22] S. Denier, H. Albin-Amiot, and P. Cointe, "Expression and composition of design patterns with aspects," in *actes de la* 2^{e} *Journée Francophone sur le Développement de Logiciels Par Aspects*, L. Seinturier, Ed. Hermès, Septembre 2005. [Online]. Available: http://www.lifl.fr/jfdlpa05/denier.pdf

[23] A. H. Eden, A. Yehudai, and J. Gil, "Precise specification and automatic application of design patterns," in *Proceedings of the* 12^{th} *Conference on Automated Software Engineering*, M. Lowry and Y. Ledru, Eds. IEEE Computer Society Press, November 1997, pp. 143–152. [Online]. Available: http://www.eden-study.org/publications.html

[24] R. K. Keller, R. Schauer, S. Robitaille, and P. Pagé, "Pattern-based reverse-engineering of design components," in *Proceedings of the* 21^{st} *International Conference on Software Engineering*, D. Garlan and J. Kramer, Eds. ACM Press, May 1999, pp. 226–235. [Online]. Available: http://www.iro.umontreal.ca/~{}schauer/Private/Publications/icse1999/icse1999.html

[25] G. Sunyé, A. Le Guennec, and J.-M. Jézéquel, "Design patterns application in UML," in *Proceedings of the* 14^{th} *European Conference for Object-Oriented Programming*, E. Bertino, Ed. Springer-Verlag, June 2000, pp. 44–62. [Online]. Available: http://gerson.sunye.free.fr/publications.html

[26] F. Bergenti and A. Poggi, "IDEA: A design assistant based on automatic design pattern detection," in *Proceedings of the* 12^{th} *international conference on Software Engineering and Knowledge Engineering*, D. Cooke and J. Urban, Eds. Springer-Verlag, July 2000, pp. 336–343. [Online]. Available: http://www.ce.unipr.it/people/poggi/publications/index.shtml

[27] A. Lauder and S. Kent, "Precise visual specification of design patterns," in *Proceedings of* 12^{th} *European Conference for Object-Oriented Programming*, S. Demeyer and J. Bosch, Eds. Springer-Verlag, July 1998, pp. 114–134. [Online]. Available: http://www.cs.ukc.ac.uk/people/staff/sjhk/pubs.html

[28] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis, "Design pattern detection using similarity scoring," *Transactions on Software Engineering*, vol. 32, no. 11, November 2006.

[29] C. Rich and R. C. Waters, *The Programmer's Apprentice*, 1^{st} ed. ACM Press Frontier Series and Addison-Wesley, January 1990.

[30] W. Zimmer, "Relationships between design patterns," in *Pattern Languages of Program Design*, J. O. Coplien and D. C. Schmidt, Eds. Addison-Wesley, 1995, ch. 18, pp. 345–364. [Online]. Available: http://citeseer.nj.nec.com/53928.html

[31] A. H. Eden and A. Yehudai, "Tricks generate patterns," Department of Computer Science, University of Tel Aviv, Tech. Rep. 324, 1997.

[32] F. Shull, W. Melo, and V. R. Basili, "An inductive method for discovering design patterns from object-oriented software systems," Computer Science Department, University of Maryland, Tech. Rep. CS-TR-3597, January 1996. [Online]. Available: http://www.cs.umd.edu/projects/SoftEng/ESEG/papers/OODP_VAL.DOC.pdf

[33] C. Krämer and L. Prechelt, "Design recovery by automated search for structural design patterns in object-oriented software," in *Proceedings of the 3^{rd} Working Conference on Reverse Engineering*, L. M. Wills and I. Baxter, Eds. IEEE Computer Society Press, November 1996, pp. 208–215. [Online]. Available: http://www.computer.org/proceedings/wcre/7674/76740208abs.htm

[34] J. M. Smith and D. Stotts, "Elemental design patterns – a link between architecture and object semantics," Department of Computer Science, University of North Carolina, Tech. Rep. TR02-011, March 2002. [Online]. Available: http://rockfish.cs.unc.edu/pubs/TR02-011.pdf

[35] M. O'Cinnéide and P. Nixon, "A methodology for the automated introduction of design patterns," in *Proceedings of the 6^{th} International Conference on Software Maintenance*, T. M. Khoshgoftaar and K. Bennett, Eds., 1998.

[36] J. O. Coplien, "Software design patterns: Common questions and answers," in *The Patterns Handbook: Techniques, Strategies, and Applications*, L. Rising, Ed. Cambridge University Press, January 1998, pp. 311–320. [Online]. Available: http://citeseer.nj.nec.com/53146.html

[37] J. Niere, J. P. Wadsack, and A. Zündorf, "Recovering UML diagrams from Java code using patterns," in *Proceedings of the 2^{nd} workshop on Soft Computing Applied to Software Engineering*, J. H. Jahnke and C. Ryan, Eds. Springer-Verlag, February 2001, pp. 89–97. [Online]. Available: http://trese.cs.utwente.nl/scase/scase-2/Proceedings.pdf

[38] E. Agerbo and A. Cornils, "How to preserve the benefits of design patterns," in *Proceedings of the 13^{th} Conference on Object-Oriented Programming, Systems, Languages, and Applications*, C. Chambers, Ed. ACM Press, October 1998, pp. 134–143. [Online]. Available: http://citeseer.nj.nec.com/31381.html

[39] D. J. Ram, P. J. Kumar, and M. S. Rajasree, "Pattern hybridization: breeding new designs out of pattern interactions," *Software Engineering Notes*, vol. 29, no. 3, pp. 1–10, May 2004. [Online]. Available: http://portal.acm.org/citation.cfm?id=986729

[40] J. O. Coplien, "Idioms and patterns as architectural literature," *IEEE Software Special Issue on Objects, Patterns, and Architectures*, vol. 14, no. 1, pp. 36–42, January 1997.

[41] B. Appleton, "Patterns and software: Essential concepts and terminology," February 2000. [Online]. Available: http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html

[42] M. L. Bernardi and G. A. Di Lucca, "Improving design patterns quality using aspect orientation," in *Proceedings of the 3^{rd} Software Technology and Engineering Practice workshop series*, M. di Penta and Y. Zou, Eds. IEEE Computer Society Press, September 2005.

[43] G. Antoniol, R. Fiutem, and L. Cristoforetti, "Design pattern recovery in object-oriented software," in *Proceedings of the 6^{th} International Workshop on Program Comprehension*, S. Tilley and G. Visaggio, Eds. IEEE Computer Society Press, June 1998, pp. 153–160. [Online]. Available: http://citeseer.nj.nec.com/antoniol98design.html

[44] R. Schauer and R. Keller, "Pattern visualization for software comprehension," in *Proceedings of the 6^{th} International Workshop on Program Comprehension*, S. Tilley and G. Visaggio, Eds. IEEE Computer Society Press, June 1998, pp. 4–12. [Online]. Available: http://ieeexplore.ieee.org/search/wrapper.jsp?arnumber=693273

[45] M. Ó. Cinnéide and P. Nixon, "Automated application of design patterns to legacy code," in *Proceedings of the 1^{st} Workshop on Object-Oriented Technology*, A. M. D. Moreira and S. Demeyer, Eds. Springer-Verlag, June 1999, pp. 176–120. [Online]. Available: http://portal.acm.org/toc.cfm?id=646779

[46] L. Prechelt and C. Krämer, "Functionality versus practicality: Employing existing tools for recovering structural design patterns," *Journal of Universal Computer Science*, vol. 4, no. 12, pp. 866–883, December 1998. [Online]. Available: http://www.jucs.org/jucs_4_12/functionality_versus_practicality_employing

[47] R. Wuyts, "Declarative reasoning about the structure of object-oriented systems," in *Proceedings of the 26^{th} Conference on the Technology of Object-Oriented Languages and Systems*, J. Gil, Ed. IEEE Computer Society Press, August 1998, pp. 112–124. [Online]. Available: http://www.iam.unibe.ch/~{}wuyts/publications.html

[48] F. Arcelli, S. Masiero, C. Raibulet, and F. Tisato, "A comparison of reverse engineering tools based on design pattern decomposition," in *Proceedings of the 16^{th} Australian Software Engineering Conference*, P. Strooper, Ed. IEEE Computer Society Press, March–April 2005, pp. 262–269. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/ASWEC.2005.5

[49] T. Mikkonen, "Formalizing design patterns," in *Proceedings of the 20^{th} International Conference on Software Engineering*, T. Katayama and D. Notkin, Eds. IEEE Computer Society Press, April 1998, pp. 115–124. [Online]. Available: http://babel.ls.fi.upm.es/services/babylon/mikkonen-fdp98.pdf

[50] J. Bosch, "Design patterns as language constructs," *Journal of Object-Oriented Programming*, vol. 11, no. 2, pp. 18–32, February 1998. [Online]. Available: http://citeseer.nj.nec.com/bosch98design.html

[51] J. Seemann and J. W. von Gudenberg, "Pattern-based design recovery of Java software," in *Proceedings of 5^{th} international symposium on Foundations of Software Engineering*, B. Scherlis, Ed. ACM Press, November 1998, pp. 10–16. [Online]. Available: http://www.informatik.uni-trier.de/~{}ley/db/indices/a-tree/s/Seemann:Jochen.html

[52] J. Dong, "UML extensions for design pattern compositions," *Journal of Object Technology*, vol. 1, no. 5, pp. 149–161, November 2002. [Online]. Available: http://www.jot.fm/jot/issues/issue_2002_11/article3/index_html

[53] M. Ó. Cinnéide, "Automated refactoring to introduce design patterns," in *Proceedings of the ICSE Doctoral Workshop*, J. Magee and M. Pezzè, Eds., June 2000. [Online]. Available: http://swt.cs.tu-berlin.de/lehre/seminar/ss02/

[54] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa, "Modularizing design patterns with aspects: A quantitative study," in *Proceedings of the 4^{th} international conference on Aspect-Oriented Software Development*, P. Tarr, Ed. ACM Press, March 2005, pp. 3–14. [Online]. Available: http://portal.acm.org/citation.cfm?id=1052898.1052899

[55] D. J. Ram, K. N. A. Raman, and K. N. Guruprasad, "A pattern oriented technique for software design," *SIGSOFT Softw. Eng. Notes*, vol. 22, no. 4, pp. 70–73, Jul. 1997. [Online]. Available: http://doi.acm.org/10.1145/263244.263265

[56] J. Kerievsky, *Refactoring to Patterns*, 1^{st} ed. Addison Wesley, August 2004. [Online]. Available: www.industriallogic.com/xp/refactoring/

[57] D. B. Lange and Y. Nakamura, "Interactive visualization of design patterns can help in framework understanding," in *Proceedings of the Tenth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '95. New York, NY, USA: ACM, 1995, pp. 342–357. [Online]. Available: http://doi.acm.org/10.1145/217838.217874

[58] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein, "MultiJava: Modular open classes and symmetric multiple dispatch for Java," in *Proceedings of the 15^{th} Conference on Object-Oriented Programming, Systems, Languages, and Applications*, D. Lea, Ed. ACM Press, October 2000, pp. 130–145. [Online]. Available: citeseer.nj.nec.com/clifton00multijava.html

[59] J. Soukup, "Implementing patterns," in *Pattern Languages of Program Design*, 1^{st} ed., J. O. Coplien and D. C. Schmidt, Eds. Addison-Wesley, May 1995, ch. 20, pp. 395–412. [Online]. Available: http://www.codefarms.com/publications/papers/patterns.html

[60] Y.-G. Guéhéneuc and Rabih Mustapha, "A simple recommender system for design patterns," in *Proceedings of the 1^{st} EuroPLoP Focus Group on Pattern Repositories (EPFPR)*, M. Weiss, A. Birukou, and P. Giorgini, Eds. N/A, July 2007, p. N/A, 2 pages. [Online]. Available: http://www.ptidej.net/publications/documents/EuroPLoP07PRb.doc.pdf

[61] A. H. Eden, Y. Hirshfeld, and A. Yehudai, "LePUS – A declarative pattern specification language," Department of Computer Science, University of Tel Aviv, Tech. Rep. 326/98, June 1998. [Online]. Available: citeseer.nj.nec.com/112816.html

[62] G. Hedin, "Language support for design patterns using attribute extension," in *Proceedings of the 1^{st} ECOOP workshop on Language Support for Design Patterns and Frameworks*, J. Bosch and S. Mitchell, Eds. Springer-Verlag, June 1997, pp. 137–140. [Online]. Available: http://www.cs.lth.se/Research/ProgEnv/LSDF.html

[63] D. C. Schmidt, "Model-driven engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, February 2006, guest Editor's Introduction. [Online]. Available: http://www.cs.wustl.edu/~schmidt/GEI.pdf

[64] T. Mens and P. V. Gorp, "A taxonomy of model transformation," *Electronic Notes in Theoretical Computer Science*, vol. 152, pp. 125 – 142, 2006, proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066106001435

[65] M. Tatsubori and S. Chiba, "Programming support of design patterns with compile-time reflection," in *Proceedings of the 1^{st} OOPSLA workshop on Reflective Programming in C++ and Java*, J.-C. Fabre and S. Chiba, Eds. Center for Computational Physics, University of Tsukuba, October 1998, pp. 56–60, uTCCP Report 98-4. [Online].

Available: http://www.csg.is.titech.ac.jp/~{}chiba/oopsla98/proc/index.html

[66] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh, "Towards pattern-based design recovery," in *Proceedings of the* 24^{th} *International Conference on Software Engineering*, M. Young and J. Magee, Eds. ACM Press, May 2002, pp. 338–348. [Online]. Available: http://portal.acm.org/citation.cfm?id=581382

[67] A. Ampatzoglou, A. Chatzigeorgiou, S. Charalampidou, and P. Avgeriou, "The effect of gof design patterns on stability: A case study," *IEEE Transactions on Software Engineering*, vol. 41, no. 8, pp. 781–802, Aug 2015.

[68] F. Khomh, Y. G. Gueheneuc, and G. Antoniol, "Playing roles in design patterns: An empirical descriptive and analytic study," in *2009 IEEE International Conference on Software Maintenance*, Sept 2009, pp. 83–92.

[69] A. Ampatzoglou and A. Chatzigeorgiou, "Evaluation of object-oriented design patterns in game development," *Information and Software Technology*, vol. 49, no. 5, pp. 445 – 454, 2007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584906000929

[70] M. Ali and M. O. Elish, "A comparative literature survey of design patterns impact on software quality," in *2013 International Conference on Information Science and Applications (ICISA)*, vol. 00, 06 2013, pp. 1–7. [Online]. Available: doi.ieeecomputersociety.org/10.1109/ICISA.2013.6579460

[71] F. Khomh, "Patterns and quality of object-oriented software systems," Ph.D. dissertation, Université de Montréal, 2010. [Online]. Available: http://hdl.handle.net/1866/4601

[72] J. O. Coplien, *Advanced C++ Programming Styles and Idioms*, 1^{st} ed. Addison-Wesley, August 1991. [Online]. Available: www.awprofessional.com/catalog/product.asp?product_id=%7BF983A2EA-89B7-4F25-B82B-6CC86496C735%7D

[73] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray, *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*, 1^{st} ed. John Wiley and Sons, March 1998. [Online]. Available: www.amazon.com/exec/obidos/tg/detail/-/0471197130/ref=ase_theantipatterngr/103-4749445-6141457

[74] S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object Oriented Reengineering Patterns*. Morgan Kaufmann Publishers Inc., 2002.

[75] W. Pree, "Meta patterns - a means for capturing the essentials of reusable object-oriented design," in *Proceedings of the 8th European Conference on Object-Oriented Programming*, ser. ECOOP '94. London, UK, UK: Springer-Verlag, 1994, pp. 150–162. [Online]. Available: http://dl.acm.org/citation.cfm?id=646152.679381

[76] M. Dawson, G. Johnson, and A. Low, "Best practices for using the java native interface," July 2009. [Online]. Available: https://www.ibm.com/developerworks/library/j-jni/index.html

[77] G. B. de Pádua and W. Shang, "Studying the prevalence of exception handling anti-patterns," in *Proceedings of the 25th International Conference on Program Comprehension*, ser. ICPC '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 328–331. [Online]. Available: https://doi.org/10.1109/ICPC.2017.1

[78] Ségla Kpodjedo, F. Ricca, P. Galinier, Y.-G. Guéhéneuc, and G. Antoniol, "Design evolution metrics for defect prediction in object oriented systems," *Empirical Software Engineering (EMSE)*, vol. 16, no. 1, pp. 141–175, February 2011, 34 pages. [Online]. Available: http://www.ptidej.net/publications/documents/EMSE11a.doc.pdf

[79] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '06/FSE-14. New York, NY, USA: ACM, 2006, pp. 1–11. [Online]. Available: http://doi.acm.org/10.1145/1181775.1181777

[80] F. Zhang, F. Khomh, Y. Zou, and A. E. Hassan, "An empirical study of the effect of file editing patterns on software quality," *J. Softw. Evol. Process*, vol. 26, no. 11, pp. 996–1029, Nov. 2014. [Online]. Available: http://dx.doi.org/10.1002/smr.1659

[81] Wei Wu, F. Khomh, , B. Adams, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of API changes and usages based on Apache and Eclipse ecosystems," *Journal of Empirical Software Engineering (EMSE)*, vol. 21, no. 6, pp. 2366–2412, December 2016, 47 pages. [Online]. Available: http://www.ptidej.net/publications/documents/EMSE16a.doc.pdf

[82] W. Shang, M. Nagappan, A. E. Hassan, and Z. M. Jiang, "Understanding log lines using development knowledge," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, 2014, pp. 21–30. [Online]. Available: https://doi.org/10.1109/ICSME.2014.24

[83] Y. Gil and I. Maman, "Micro patterns in java code," in *Proceedings of the* 20^{th} *Conference on Object-Oriented Programming Systems Languages and Applications*, R. P. Gabriel, Ed. ACM Press, October 2005, pp. 97–116. [Online]. Available: portal.acm.org/citation.cfm?id=1094811.1094819

[84] Simon Denier and Y.-G. Guéhéneuc, "Mendel: A model, metrics, and rules to understand class hierarchies," in *Proceedings of the* 16^{th} *International Conference on Program Comprehension (ICPC)*, R. Krikhaar and R. Lämmel, Eds. IEEE CS Press, June 2008, pp. 143–152, 10 pages. [Online]. Available: http://www.ptidej.net/publications/documents/ICPC08a.doc.pdf

[85] R. Wuyts, K. Mens, and T. D'Hondt, "Explicit support for software development styles throughout the complete life cycle," Programming Technology Lab, Vrije Universiteit Brussel, Tech. Rep. Vub-Prog-TR-99-07, April 1999. [Online]. Available: http://progwww.vub.ac.be/Research/ResearchPublicationsDetail2.asp?paperID=72

[86] A. Homer, J. Sharp, L. Brader, M. Narumoto, and T. Swanson, *Cloud Design Patterns: Prescriptive Architecture Guidance for Cloud Applications*. Microsoft patterns & practices, 2014.

[87] C. Leymann, F. Fehling, R. Retter, W. Schupeck, and P. Arbitter, *Cloud computing patterns*. Springer, 2014.

[88] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, 1^{st} ed. John Wiley and Sons, August 1996. [Online]. Available: http://www.amazon.com/exec/obidos/tg/detail/-/0471958697/104-1238236-1419115

[89] F. Petrillo, Z. Soh, F. Khomh, M. Pimenta, C. Freitas, and Y. G. Guhneuc, "Understanding interactive debugging with swarm debug infrastructure," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, May 2016, pp. 1–4.

[90] Y.-G. Guéhéneuc, "A systematic study of uml class diagram constituents for their abstract and precise recovery," in *Proceedings of the* 11^{th} *Asia-Pacific Software Engineering Conference (APSEC)*, D.-H. Bae and W. C. Chu, Eds. IEEE CS Press, November-December 2004, pp. 265–274, 10 pages. [Online]. Available: http://www.ptidej.net/publications/documents/APSEC04.doc.pdf

[91] D. G. Firesmith, "Pattern language for testing object-oriented software," *Object Magazine*, vol. 5, no. 8, p. 5, 1996.

[92] S. Amann, S. Proksch, S. Nadi, and M. Mezini, "A study of visual studio usage in practice," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, March 2016, pp. 124–134.

[93] C. Zhang and D. Budgen, "What do we know about the effectiveness of software design patterns?" *Transactions on Software Engineering*, vol. 38, no. 5, pp. 1213–1231, September–October 2012. [Online]. Available: http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=5975176

[94] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 403–414. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818805

[95] F. Arcelli Fontana and M. Zanoni, "Code smell severity classification using machine learning techniques," *Know.-Based Syst.*, vol. 128, no. C, pp. 43–58, Jul. 2017. [Online]. Available: https://doi.org/10.1016/j.knosys.2017.04.014