Reverse-Engineering the Literature on Design Patterns and Reverse-Engineering

Simon Denier, Foutse Khomh, and Yann-Gaël Guéhéneuc
Ptidej Team
DIRO, Université de Montréal and
DGIGL, École Polytechnique de Montréal

October 16, 2008

Abstract

Since their inception in 1994, design patterns have been the subject of many papers. In the reverse-engineering community, several authors have proposed approaches to consider design patterns during reverse- and re-engineering. However, it has been recently put forward in the community that it is difficult to compare previous approaches due to the diversity of vocabulary and the lack of a general framework to map and relate these approaches. Consequently, we study 59 papers related to design patterns in the software engineering community at large (1) to identify and define common terms related to design patterns, (2) to identify recurring themes in the papers, and (3) to further characterise approaches for design pattern detection along several categories. Recurring themes allow us to provide the portrait of the "typical" paper on design patterns while catagories draw the portrait of the "typical" approach in design pattern detection. We propose to the community to use a fix vocabulary, to diversify the approaches, and to build a common benchmark to assess the reverse engineering of design patterns.

Chapter 1

Introduction

Since their introduction in the software engineering community in 1994, design patterns have been the subject of much discussions, research work, and conferences. Researchers studied design patterns with various purposes as shown in Section 0.4.

However, design patterns lie between programming-language features and abstract concepts to ease communication (teaching, documenting) as highlighted in 2000 by the debate among Chambers, Harrison, and Vlissides [13], as shown in the following definition:

> **Design patterns** systematically name, explain, and evaluate solutions to important and recurring object-oriented design problems. They capture design experience in a form that developers can use effectively. They express proven solutions and make them more accessible to developers of new programs and help developers to choose design alternatives. Thus, they capture experience in the community and create a common vocabulary by bringing forward reusable solutions. They also improve the documentation and maintenance of existing programs by making explicit the roles of classes and objects and the underlying intents of designs. Some of the most important design patterns are documented and presented in catalogs, for example this by the GoF [26]. The general process of use of design patterns during development and maintenance is as follows: during the (re)conception of the program, developers should identify a problem, find the related patterns, implement the chosen one, and document it so as to be able to reuse this knowledge in later stage. Knowing the design patterns used in programs can ease their comprehension and thus their maintenance. Each design pattern has many possible implementations; an implementation may differ from the one suggested in a catalog while still satisfying the intent of the pattern.

The definition of design patterns shows their extent but leaves much room for interpretation. This loose definition of patterns was desired by their early proponents [13] and is particularly relevant to the reverse-engineering community because of the many papers on design pattern identification.

One of the first of such papers has been published by Brown in 1996 [11] more than 10 years ago. Since, several other papers have presented new approaches and techniques with improved performance, precision, recall, and a larger variety of patterns.

Yet, these approaches are difficult to compare and to build upon due to the loose definition of design patterns and their related concepts. Consequently, there has been recent efforts in the literature [3, 28, 46] and dedicated workshops, such as DP4RE at WCRE 2006, to better choose and define the terms related to patterns and to organise the papers and approaches.

We build upon these previous efforts and propose three contributions to the body of knowledge on design patterns in the context of reverse engineering. First, we propose a study of the vocabulary pertaining to design patterns used in previous work and build a dictionary of terms to clearly name and define the concepts related to design patterns. Second, we classify previous work in different themes and sub-themes. Third, we detail several categories of the design pattern detection sub-theme to discuss the similarities and differences among previous approaches and highlight the difficulty of comparing them. This study follows previous categorisation efforts of the reverse engineering tools [29] and of the reverse engineering community [33].

We conclude our study by providing two portraits: one of the "typical" paper on design patterns using the identified themes and one of the "typical" approach in reverse engineering using the categories of the design pattern detection sub-theme. We also suggest research directions to the community on the use of the vocabulary related to design patterns, the diversity of approaches, and the comparison of the approaches with one another.

The rest of the paper is organised as follows. Section presents the process and the results of our analysis of the vocabulary. Section 0.4 introduces the different themes related to design patterns in the literature. Section 0.10 proposes categories to characterise approaches to detect design patterns. Section 0.12 discusses our study while Section 0.15 concludes with some suggestions.

Chapter 2

Dictionary of Terms

The problem arising from the loose definition of design patterns and their first catalog is that of *interpretation*: the definition leaves space for interpretation and thus also for debate and ambiguities.

We propose and implement a process to retain and define the key terms related to design patterns. This process is similar to the process of domain analysis [49]. As such, the objective of the process is to provide a list of terms most discriminating the concepts related to design patterns to avoid ambiguities. An ontology of the concepts is future work.

2.1 Process

Our process divides in seven steps and results in a list of terms characterising design patterns, their definitions, and a mapping between the terms in previous papers and these.

1. Identify collections of references of papers related to design patterns.
2. Choose one such collection and download all referenced papers.
3. Read all the paper and identify terms related to design patterns.
4. Compile all terms including potential 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 in the literature 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 terms related to design patterns used in each paper. The common meta-model is the set of identified 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 exiting 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 term with the definition of the retained terms. The terms and their mapping provides a useful background for the co-understanding of previous papers.

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

2.2 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 libraries of references on design patterns, available at `http://liinwww.ira.uka.de/bibliography/SE/patterns.html` and `http://www.patternforge.net/wiki/index.php?title=Papers_on_Pattern_Repositories`. These two libraries mostly overlap and are representative of other such libraries. One of the authors contributed to the second library with more than 60 papers. Therefore, we choose this second library for the sake of simplicity.

The chosen library 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[2]. In these 59 papers[3], 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 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 understanding, for example "Detection" vs. "Identification", or that have agreed-upon definitions, such as "Role".

2.3 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 1 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 definitions. With this grouping, we retain 16 terms. Table 1 presents in bold the retained terms and their main synonyms. We do not show here homonyms because they will be studied 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 1 is useful because it can help the community in making an informed choice of the terms to be retained for each definition.

2.4 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 studied papers. Table 2 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 terms.

Table 2 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 relative novelty of this concept and 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 [24], 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 2 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. Out study also hints 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 the co-understand the studied papers using a common set of terms.

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

[3]The complete list of studied papers is available at `http://www.ptidej.net/downloads/Poster.pdf`.

Terms	Definitions
Abstract Design Pattern [47], Basic Form [6], **Design Motif** [28], Design Template [11], Essence [12], Expression [20], Formalisation [24], Implementation [36], Lattice [24], Leitmotiv [58], Logic [12], Realisation [8], Reification [36], Representation [40], Specification [24], and Structure [26]	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 [28].
Alternative [11], Modified Pattern [60], **Variant** [12], Version [11]	A design motif (see previous definition) manifesting variety, deviation of its canonical form as described in [26].
Canonical form [11]	Original design motif as described in [26]
Cliché [51], **Idiom** [62], Micropattern [23]	A standard algorithmic fragment [51].
Design Component [55], **Instance** [24], Microarchitecture [38]	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 [28].
Design Pattern [26]	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. [26]
Elemental Pattern [56], Fragments [24], Minipattern [44], **Subpattern** [18]	Abstract syntax graph structures [43] that are constituent parts of other patterns or subpatterns. An idiom could be a subpattern while subpatterns are not necessarily idioms.
Fundamental Pattern [1]	A core of design patterns that capture good object-oriented design and that can be used in various contexts [1].
Hybrid Pattern [50]	Pattern generated through hybridization and whose intent is to solve a high level design problem in a generic context [50].
Intent [26]	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? [26]
Language of Patterns [17]	A structured method of describing good design practices within a field of expertise [18].
Occurrence [11]	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 [24].
Protopattern [17]	"patterns in waiting" that are not yet known to recur [3].
Secondary Role [9]	Role that can be superimposed over the defining role of a class in a program [9].
Solution [26]	The solution of a design pattern is the description composed of the "Structure", "Participants", and "Collaborations" sections, as defined in the format in [26].
Trick [23]	A trick is an operator specifying the sequence of steps taken in the realization of a design motif [23].

Table 2.1. Retained Terms related to Design Patterns and their Definitions. Terms are grouped by sets of synonyms, see Table 2 for the precise mapping among terms.

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			[60]	**[2, 11]**						[60]											
Design Motif	[52]				[38]		**[15, 24]**	[12, 11]	[48, 61]	[4]	[58]	[20]	[41, 24]					[20, 10]			
Design Pattern							[20]	**[12, 11]**													
Fundamental Pattern															**[40, 11]**						
Hybrid Pattern																**[50]**					
Idiom			[54]		[38, 11]		[23]	[54]						[24]			**[6, 62]**				
Instance						[36]		[2, 10]										[12, 11]	**[61, 24]**		
Intent											[36]									**[12, 11]**	
Language of Patterns																					**[18, 17]**
Occurrence								[54, 55]										[52, 48]	[55, 38]		
Protopattern																					
Role																					
Secondary Role																					
Solution								[50, 21]													
Subpattern			[6, 62]			[55]	[23]			[56]				[24]			[17]				
Trick																					
Variant		[12, 11]																[2, 54]			

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	[24, 23]		[9]	[38, 11]	[23]	[14, 15]		[58]		[14]		[10, 17]		[55, 38]	[40, 24]	[12, 11]				
Design Pattern														[9, 1]						
Fundamental Pattern																				
Hybrid Pattern																				
Idiom					[24, 23]				[17]											
Instance								[58]			[11]			[27]						
Intent																				
Language of Patterns																				
Occurrence								**[24, 11]**		[60, 8]				[27]						
Protopattern									**[52, 17]**											
Role																				
Secondary Role													**[9]**							
Solution														**[12, 11]**		[1]				
Subpattern					[24, 23]				[36]								**[1, 18]**			
Trick																		**[24, 23]**		
Variant							[60]												**[62, 12]**	[41, 6]

Table 2.2. Mapping of the Terms related to Design Patterns. We only present a maximum of 2 references per cell for lack of space. The complete table is available at `http://www.ptidej.net/downloads/Poster.pdf`.

Chapter 3

Themes and Classification Spaces

Using the retained terms, we study the papers and identify several recurring themes; one paper possibly following more than one theme. These themes are useful to characterise the papers and to compare papers with one another.

We present 6 themes: purpose, process, formalisation, forward engineering, reverse engineering, and documentation. We defined each theme, and relate the studied papers with them. These themes are useful to categorise previous papers and thus relate them with one another.

3.1 Purpose Theme

Design patterns serve many purposes as shown in their definition. We divide this theme in two sub-themes: community purpose and project purpose.

Community purpose. The community purpose concerns the use of design patterns to spread information. Agerbo and Cornils [1] discuss the use of design patterns in programs and put forward the idea of a design pattern library to solve the problem of tracing design patterns in programs. They also described the risks of over-using design patterns in programs and of having too many patterns to choose from. They argue that if patterns grow in large numbers, it will be increasingly difficult for users to grasp them all and they will not form a common vocabulary anymore.

Project purpose. The project purpose concerns the use of design patterns as development tools. Their use can be described at different development stages:

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 development have been the subject of many papers. Beck [6] suggests that patterns generate architectures. Similarly, Guruprasad *et al.* [32] propose the *Pattern Oriented Technique* for developing programs by focusing on design patterns by viewing a program as a collaboration of design patterns. Lange and Nakamura [39] 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 at a certain point in the understanding process, they could help in "filling in the blanks" and in further exploring a program, improving thus the understandability of the program. Many other papers focused on the use of patterns during development: [10, 16, 57].

3.2 Process Theme

This theme focuses on the processes of using design patterns during the different stages of the development and maintenance. The general process is presented in the definition of design patterns in Section . During the maintenance of program, more particularly, a developer identifies code smells or a design problem, find appropriate

refactorings or design patterns, implement these, and document their use to be able to reuse this knowledge in the future [37].

Many studies focused on the selection of design patterns and attempted to draw rules to combine these patterns during the development. Guéhéneuc *et al.* [31] present a recommender system to help users in choosing among the 23 design patterns from the GoF. [50, 32] draw rules for the combination and application of design patterns. However, despite these studies, there is still no consistent program/language of design patterns. This theme on the process of using design patterns have also been the topic of the works of [10, 16, 57].

3.3 Formalisation Theme

The formalisation theme relates approaches to describe design motifs. Despite their importance, few specification languages are reported in the literature. Among these, LePUS [22] is a full-fledged logic language with well defined semantics. It has a compact vocabulary and can represent regularities and relations among classes, functions, and inheritance class hierarchies. Similarly, Hedin [34] presents an approach based on attribute grammars for formalising design motifs, which provides attributes extensions describing conventions by declarative semantic rules. On this same theme we have [56].

3.4 Forward Engineering Theme

Design patterns are useful for the design of object-oriented programs. 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.

Code generation. With the growing interest in Model Driven Engineering [53], design patterns could be the bridge between the designs of programs and the automatic generation of their source code. Many techniques of code generation exists: Czarnecki and Helsen [19] propose a taxonomy for the classification of model transformation techniques and discuss their applicability. Budinsky [12] describe the architecture and implementation of a tool that automates the implementation of design motifs. 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 [57] explores the problems and possibilities of automated motif implementation. He identifies three basic problems: the loss of the motifs during implementation, large clusters of mutually dependent classes caused by the use of multiple motifs, and the lack for a library of concrete reusable motifs. Another paper on the topic is the Agerbo and Cornils' work [1].

Languages. Syntax extensions and extended language constructs have been explored to simplify the specification of design motifs and to improve the readability of programs. Tatsubori *et al.* [59] showed that compile-time MOPs provide a general framework to implement design motifs. They show that developers can use a MOP to create a library of reusable motifs based on syntax extensions and extended language constructs. Similarly, Chambers [16] also proposes an interesting contribution on the topic. Bosch [10] identifies four major problems associated with the implementation of design motifs using conventional object-oriented languages. He solves these problems with a layered object model, LayOM. LayOM is an extended object-oriented language that provides explicit support for design motifs. It is extensible with abstractions for other motifs.

3.5 Reverse Engineering Theme

The theme of reverse engineering concerns all the papers on design patterns in the context of reverse engineering. There are two major trends in this theme: design pattern detection and pattern based component 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 for example because the detection of occurrences in programs could improve their comprehension [61]. Occurrences would also improve the documentation of programs. During the reverse engineering of a program, design patterns are also use to refactor code smells and design defects "away from" the code. Many papers have been published in this sub-theme, for example: [28, 42, 56, 60]. On the sub-theme of pattern based component recovery, we can quote the work of [36]. We detail the sub-theme of design pattern detection in the following section.

3.6 Documentation Theme

This theme includes papers on the role of design patterns in the documentation of programs. Many papers promote the idea that documenting patterns in the code helps in understanding a design and thus in easing maintenance because developers often attempt to recover non-documented design patterns in programs. They have illustrated how a combination of pattern allows to simply convey the essence of design (for example, in JUnit). They have put forward the impact of design patterns on documentation and in the recovery of design information for revere- and re-engineering. Lange and Nakamura [39] shows that design patterns can serve as guide in program exploration and thus make the process of program understanding more efficient. The approach presented by [34] also discusses the identification of design motifs for the documentation of programs. Soukup [57] proposes an approach to build a library of reusable common motifs. Other interesting approaches are proposed by [1] and [10].

Chapter 4

Design Pattern Detection Sub-theme

We now further study the papers in the design pattern detection sub-theme and propose seven categories characterising the existing approaches.

The categories describe the characteristics of the approaches with nominal values, highlighted in italic in the following. Two nominal values in a given category are not necessarily exclusive to each other. We only include categories that help in discriminating previous approaches; unique characteristics of approaches are not taken into account because they are not useful in comparing approaches.

We do not look at characteristics such as usability, coverage of pattern catalogs, performance, or scalability. These characteristics are interesting for the evaluation of approaches but not for understanding the directions followed (or not) by the community.

4.1 Conceptual Categories

Conceptual categories describe the data and computations involved in the detection of occurrences.

Theory. The theory category describes the theory behind the computations performed by an approach to detect occurrences of motifs. Most of the existing approaches use *graph* theory: [42] uses graph transformation rules; [28] relies on constraint-based programming, which combines graph with logic. *Logic* programming is used to infer information based on the graph [61]. Smith and Scotts [56] uses a logic language called rho-calculus. *Heuristics* allow to deal with the large search space: Antoniol *et al.* [2] use metrics on the target model. Niere *et al.* [42] use fuzzy logic on the abstract syntax graph. The theory category is related with the motif model category, which represents the input to the computation model supported by the theory.

Motif Model. The motif model category describes the characteristics used by the approaches to detect the motif. These characteristics are used to detect occurrences in the target programs. Therefore, a model of each motif must be provided as well as a mean to map constituents of the motif model against the target model (cf. target model category).

Many papers have focused on the *static structural* characterisation of the occurrences. *Behavioral* models (such as the call graph) are sometimes used to refine the structural model. Few work, such as [35], have focused on *dynamic* analysis as a primary mean for detection. To our knowledge, the *vocabulary* of design patterns (such as roles in classnames) has never been used for detection purpose.

Target Model. The category of the target model describes the characteristics of the model used to represent the target programs, in which occurrences of motifs are searched. Most previous approaches use an *abstract model* of the program, based on its source code or binaries. This abstract model has two advantages: first it is often described as language-independent; second it can be enriched with data helpful to the detection. The model can be built in *incremental* steps, each step enriching the model. Such information includes binary class relationships [28] or delegation calls [42]. The model can be *specific* to the detection of occurrences or *general*: Wuyts [61] and Smith and Scotts [56] show how they infer information useful for other tasks, such as programming style checking or elemental design patterns detection.

Weighting. The weighting category refines the motif model in two directions. First, some approaches use only a *partial* model of the motif: they deliberately ignore some constituents in their motif models, considered as optional or too common to be useful during the detection. Such constituents appear as if their weight was null. [35] characterize this approach as searching for "minimal key structures". The idea of "principal" roles in [5, 45] also relates to this category.

Second, some approaches give more weight to some constituents in the models, *i.e.*, they deem some constituents more important than the others for the detection. Such models are *selective* while others not giving importance to special constituents are *uniform*. Tsantalis *et al.* [60] demonstrates an example of *partial uniform* model using matrices. Matrices coefficients could also be used to make the approach *selective*.

In some approaches, weighting can also be negative: a negative weight indicates that a constituent is forbidden in the model, such as a relationship between two concrete roles [47].

4.2 Occurrence Categories

Occurrence categories describe the characteristics of the detected occurrences, *i.e.*, the results of the detection process.

Granularity. The granularity category describes the level of details attained by the approaches in the detected occurrences. Approaches could simply point to a set of classes implementing a *motif*. However, most approaches are more detailed and provide a mapping between *roles* and classes. Others could map every *constituents* of the motifs, including behaviors and relationships, with constituents of the target model, such as methods and associations through instance variables, for example Niere [42] attempts .

Cardinality. The cardinality category describes whether constituents in an occurrence can be *optional*, *single* (only one), or *multiple* (at least one). The cardinality category complements the granularity category. Each constituent in a motif model can be assigned a particular cardinality. For example, a role can be assigned to a single class, but another in the same motif can be assigned to multiple classes. This category is also linked to the weighting category: constituent with a null weight are typically optional.

Variation. The variation category describes how an approach accommodates variants from the *canonical* motif. An *isotopic* variant [56] exhibits a deviation from the canonical form which does not affect its structure or behavior. A common case is the existence of an intermediate class between two roles, related by inheritance or association. The occurrence still appears and behaves like the canonical motif. An *approximate* variant, on the contrary, does not exhibit all the characteristics of its motif [28]. Either a role is missing or its behavior differs from the intended behavior. As such, approximate occurrences are interesting to detect because they can be primary target for refactoring.

Chapter 5

Discussions

We now conclude our study with discussions and suggestions for future work in the community.

5.1 Instance, occurrence, precision, and recall

During our study of the literature and the design pattern detection approaches, we noticed an important difference between "instances" and "occurrences", related to the concept of precision and recall [25].

An instance is the concrete implementation of a design motif in a program. As such, only developers can decide whether some micro-architectures in the program are instances of some design motifs: the instance must respect the intent and other properties of the design pattern that are currently impossible to check automatically. On the contrary, an occurrence of a design motif is a micro-architecture that is similar to the motif: it conforms to a reasonable extent to the motif but does not necessarily conforms to the intent of the pattern. Consequently, current design pattern detection approaches can only detect occurrences, not instances, because only developers can vet whether an occurrence respect the intent of the related pattern and, thus, is an instance.

Precision and recall are two measures that can be used to assess design pattern detection approaches by comparing the identified occurrences with the known instances of motifs in a set of programs. Unfortunately, there is neither a common agreement on what *is* an instance in a program, nor a widely accepted benchmark with known instances of motif, such as P-MARt [30].

5.2 Portraits

Using our study, we draw the portraits of the typical paper on design patterns and the typical approach on design pattern detection.

A Typical Thematic Portrait. While studying the papers and identifying their themes, we notice that the majority of papers to date relate to the themes of reverse engineering to contribute to the documentation of programs for a project purpose. In comparison, they are very few papers focusing on the formalisation of design patterns. In the forward engineering theme many studies focus on language extension rather than code generation.

A Typical Design Pattern Detection Portrait. The review of approaches in the design pattern detection sub-theme outlines the current typical approach: it relies on *graph* theory, using *abstract* and *incremental* models of the target programs. The motif model is *static* with *structural* and basic *behavioral* information. The approach identifies *roles* in the detected occurrences and accommodates *isotopic* variants of the motif.

5.3 Future Work on Design Patterns

This typical portrait highlights some gaps in the literature on design motif detection. Graph theory has been refined with logic and heuristics, but few approaches have tried a different theory (for example, vocabulary). This predominance can be explained by the mostly static and structural motif models. The weighting category is sometimes used but it is difficult to judge its impact. The cardinality category is often overlooked when discussing the results.

Therefore our study highlights the need for the development of new hybrid approaches, combining static and dynamic information as well as structural and behavioral information. Current and future approaches also need to be evaluated in terms usability and coverage of the design pattern catalogs. Therefore, we suggest that future approaches attempt to fill these gaps.

Our study also suggests the need to use a uniform vocabulary when writing papers as well as a uniform classification in themes to ease the identification of relevant papers and the comparison among approaches. Being able to compare approaches is of high importance if the field of design pattern detection wants to consolidate and build on previous work, as it has been done in the field of clone detection [7].

Another gap in the literature on design pattern detection concerns the description of the motifs. Most approaches use a meta-model and a dedicated language to model motifs, yet none is quite satisfying yet with respect to coverage of the catalogs and usability for code generation or detection. Efforts must be pursued to devise a formalism satisfying the needs and build a representative library of reusable motifs.

The composition of motifs has also been little studied. Composition can refer to the cohabitation in implementation of motif constituents, the development of a composite form of motifs, the declaration of a new motif as a composite of motifs, or the implementation/generation of idioms/subpatterns in the code. There is a lack of studies about the impact of the composition of motifs on the code.

Finally, studies must be perform to concretely assess the impact of design pattern detection on the comprehension process as well as on the evolution of programs [5]. Such studies are important to justify—even if a posteriori—the work on detection and the use of design patterns during maintenance.

Chapter 6

Conclusion and Future Work

We presented an extensive study of the literature on design patterns triggered by the loose definitions of design patterns, which prevent comparison among approaches, in particular in the reverse-engineering community.

First, we performed a domain analysis of the literature on design pattern to propose a consistent vocabulary. The retained terms allowed us to cast previous papers in a common framework and to compare their approaches from the linguistic point of view. Second, using the previous vocabulary, we identified six themes categorising the papers on design patterns. The themes allow us to classify approaches and to help readers focus on approach of interest. Third, we refined the design pattern detection sub-theme into seven categories to further classify related approaches.

Finally, we discussed the difficulty of measuring the precision and recall of the detection approaches due to the lack of common vocabulary and of agreed-upon benchmarks. We drew the portraits of the typical paper and approach related to design patterns and their detection. We also suggested research directions for the community to strengthen the field of design pattern detection in reverse-engineering.

Future work includes, in addition to the concerns highlighted in Section 0.12, including more papers and books in our study, performing an historical study of the papers, lobbying the community to build a common repository of instances of design motifs, and comparing in the form of a competition the current approaches to design pattern detection.

Bibliography

[1] 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*, pages 134–143. ACM Press, October 1998.

[2] 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*, pages 153–160. IEEE Computer Society Press, June 1998.

[3] B. Appleton. Patterns and software: Essential concepts and terminology, February 2000.

[4] 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*, pages 262–269. IEEE Computer Society Press, March–April 2005.

[5] L. Aversano, G. Canfora, L. Cerulo, C. D. Grosso, and M. di Penta. An empirical study on the evolution of design patterns. In *Proceedings of the* 6^{th} *joint meeting of the European Software Engineering Conference and the* 14^{th} *Symposium on Foundations of Software Engineering*, pages 385–394. ACM Press, September 2007.

[6] K. Beck and R. E. Johnson. Patterns generate architectures. In *Proceedings of* 8^{th} *European Conference for Object-Oriented Programming*, pages 139–149. Springer-Verlag, July 1994.

[7] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. In *IEEE Transactions on Software Engineering*, 33(9):577–591, September 2007.

[8] 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*, pages 336–343. Springer-Verlag, July 2000.

[9] 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*. IEEE Computer Society Press, September 2005.

[10] J. Bosch. Design patterns as language constructs. In *Journal of Object-Oriented Programming*, 11(2):18–32, February 1998.

[11] K. Brown. Design reverse-engineering and automated design pattern detection in Smalltalk. Technical Report TR-96-07, Department of Computer Science, University of Illinois at Urbana-Champaign, July 1996.

[12] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from design patterns. In *IBM Systems Journal*, 35(2):151–171, February 1996.

[13] C. Chambers, B. Harrison, and J. Vlissides. A debate on language and tool support for design patterns. In *proceeding of the* 27^{th} *Conference on Principles of Programming Languages*, pages 277–289. ACM Press, January 2000.

[14] M. Ó. Cinnéide. Automated refactoring to introduce design patterns. In *Proceedings of the ICSE Doctoral Workshop*, June 2000.

[15] 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*, pages 176–120. Springer-Verlag, June 1999.

[16] 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*, pages 130–145. ACM Press, October 2000.

[17] J. O. Coplien. Idioms and patterns as architectural literature. In *IEEE Software Special Issue on Objects, Patterns, and Architectures*, 14(1):36–42, January 1997.

[18] J. O. Coplien. Software design patterns: Common questions and answers. In *The Patterns Handbook: Techniques, Strategies, and Applications*, pages 311–320. Cambridge University Press, January 1998.

[19] K. Czarnecki and S. Helsen. Classification of model transformation approaches. 2003.

[20] 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*. Hermès, Septembre 2005.

[21] J. Dong. UML extensions for design pattern compositions. In *Journal of Object Technology*, 1(5):149–161, November 2002.

[22] A. H. Eden, Y. Hirshfeld, and A. Yehudai. LePUS – A declarative pattern specification language. Technical Report 326/98, Department of Computer Science, University of Tel Aviv, June 1998.

[23] A. H. Eden and A. Yehudai. Tricks generate patterns. Technical Report 324, Department of Computer Science, University of Tel Aviv, 1997.

[24] 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*, pages 143–152. IEEE Computer Society Press, November 1997.

[25] W. B. Frakes and R. Baeza-Yates. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, 1992.

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

[27] 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*, pages 3–14. ACM Press, March 2005.

[28] Y.-G. Guéhéneuc and G. Antoniol. Demima: A multi-layered framework for design pattern identification. In *Transactions on Software Engineering*, September 2008. *Accepted for publication.*

[29] Y.-G. Guéhéneuc, K. Mens, and R. Wuyts. A comparative framework for design recovery tools. In *Proceedings of the* 10^{th} *Conference on Software Maintenance and Reengineering*, pages 121–130. IEEE Computer Society Press, March 2006.

[30] Y.-G. Guéhéneuc, H. Sahraoui, and Farouk Zaidi. Fingerprinting design patterns. In *Proceedings of the* 11^{th} *Working Conference on Reverse Engineering*, pages 172–181. IEEE Computer Society Press, November 2004.

[31] 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*, July 2007.

[32] D. J. R. Guruprasad, M. K. N. A. R. Guruprasad, and M. K. N. Guruprasad. A pattern oriented technique for software design. In *ACM SIGSOFT Software Engineering Notes*, 22(4):70–73, July 1997.

[33] A. E. Hassan and R. C. Holt. The small world of software reverse engineering. In *Proceedings of the* 11^{th} *Working Conference on Reverse Engineering*, pages 278–283. IEEE Computer Society Press, November 2004.

[34] 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*, pages 137–140. Springer-Verlag, June 1997.
[35] D. Heuzeroth, T. Holl, and W. Löwe. Combining static and dynamic analyses to detect interaction patterns. In *Proceedings the* 6^{th} *world conference on Integrated Design and Process Technology*. Society for Design and Process Science, June 2002.
[36] 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*, pages 226–235. ACM Press, May 1999.
[37] J. Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 1st edition, August 2004.
[38] 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*, pages 208–215. IEEE Computer Society Press, November 1996.
[39] D. B. Lange and Y. Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Proceedings of the* 10^{th} *annual conference on Object-oriented programming systems, languages, and applications*, pages 342 – 357. ACM Press, 1995.
[40] A. Lauder and S. Kent. Precise visual specification of design patterns. In *Proceedings of* 12^{th} *European Conference for Object-Oriented Programming*, pages 114–134. Springer-Verlag, July 1998.
[41] T. Mikkonen. Formalizing design patterns. In *Proceedings of the* 20^{th} *International Conference on Software Engineering*, pages 115–124. IEEE Computer Society Press, April 1998.
[42] 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*, pages 338–348. ACM Press, May 2002.
[43] 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*, pages 89–97. Springer-Verlag, February 2001.
[44] 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*, 1998.
[45] M. D. Penta, L. Cerulo, Y.-G. Guéhéneuc, and G. Antoniol. An empirical study of the relationships between design pattern roles and class change proneness. In *Proceedings of the* 24^{th} *International Conference on Software Maintenance*. IEEE Computer Society Press, September–October 2008.
[46] N. Pettersson and W. Löwe. Efficient and accurate software pattern detection. In *Proceedings of the* 13^{th} *Asia Pacific Software Engineering Conference*, pages 317–326. IEEE Computer Society Press, December 2006.
[47] I. Philippow, D. Streitferdt, M. Riebisch, and S. Naumann. An approach for reverse engineering of design patterns. In *Software and System Modeling*, 4(1):55–70, February 2005.
[48] L. Prechelt and C. Krämer. Functionality versus practicality: Employing existing tools for recovering structural design patterns. In *Journal of Universal Computer Science*, 4(12):866–883, December 1998.
[49] R. Prieto-Díaz. Domain analysis: An introduction. In *Software Engineering Notes*, 15(2):47–54, April 1990.
[50] D. J. Ram, P. J. Kumar, and M. S. Rajasree. Pattern hybridization: breeding new designs out of pattern interactions. In *Software Engineering Notes*, 29(3):1–10, May 2004.
[51] C. Rich and R. C. Waters. *The Programmer's Apprentice*. ACM Press Frontier Series and Addison-Wesley, 1^{st} edition, January 1990.
[52] R. Schauer and R. Keller. Pattern visualization for software comprehension. In *Proceedings of the* 6^{th} *International Workshop on Program Comprehension*, pages 4–12. IEEE Computer Society Press, June 1998.
[53] D. C. Schmidt. Model-driven engineering. In *IEEE Computer*, 39(2):25–31, February 2006. Guest Editor's Introduction.
[54] 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*, pages 10–16. ACM Press, November 1998.
[55] F. Shull, W. Melo, and V. R. Basili. An inductive method for discovering design patterns from object-oriented software systems. Technical Report CS-TR-3597, Computer Science Department, University of Maryland, January 1996.
[56] J. M. Smith and D. Stotts. Elemental design patterns – a link between architecture and object semantics. Technical Report TR02-011, Department of Computer Science, University of North Carolina, March 2002.
[57] J. Soukup. Implementing patterns. In *Pattern Languages of Program Design*, chapter 20, pages 395–412. Addison-Wesley, 1^{st} edition, May 1995.
[58] 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*, pages 44–62. Springer-Verlag, June 2000.
[59] 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*, pages 56–60. Center for Computational Physics, University of Tsukuba, October 1998. UTCCP Report 98-4.
[60] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis. Design pattern detection using similarity scoring. In *Transactions on Software Engineering*, 32(11), November 2006.
[61] 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*, pages 112–124. IEEE Computer Society Press, August 1998.
[62] W. Zimmer. Relationships between design patterns. In *Pattern Languages of Program Design*, chapter 18, pages 345–364. Addison-Wesley, 1995.