A Survey on Design Pattern Detection Approaches

Mohammed Ghazi Al-Obeidallah *M.Al-Obeidallah@brighton.ac.uk*
Department of Computing
University of Brighton
Brighton, BN2 4GJ, United Kingdom

Miltos Petridis *M.Petridis@brighton.ac.uk*
Department of Computing
University of Brighton
Brighton, BN2 4GJ, United Kingdom

Stelios Kapetanakis *S.Kapetanakis@brighton.ac.uk*
Department of Computing
University of Brighton
Brighton, BN2 4GJ, United Kingdom

Abstract

Design patterns play a key role in software development process. The interest in extracting design pattern instances from object-oriented software has increased tremendously in the last two decades. Design patterns enhance program understanding, help to document the systems and capture design trade-offs.

This paper provides the current state of the art in design patterns detection. The selected approaches cover the whole spectrum of the research in design patterns detection. We noticed diverse accuracy values extracted by different detection approaches. The lessons learned are listed at the end of this paper, which can be used for future research directions and guidelines in the area of design patterns detection.

Keywords: Design Patterns, Detection, Reverse Engineering, Gang of Four, Patterns Recovery, Survey.

1. INTRODUCTION

Design pattern defined by Gamma *et al.* [1] as "a general reusable solution to commonly occurring problem in software design". A design pattern is a description or template for how to solve a problem that can be used in many different situations. It describes the core of the solution to the problem in such way this solution can be used many times over, without doing it in the same way twice [1].

The field of design patterns detection has attracted researchers from both academia and industry. Design patterns reflect the earliest set of design decisions that have been taken by the development team. Moreover, design patterns improve software documentations, speed up the development process, enable large-scale reuse of software architectures, capture expert knowledge, capture design trade-offs and help re-structuring systems. Many approaches have been used in the last two decades to recover design pattern instances from object-oriented source code. The main objective of design pattern detection approaches is to accurately extract the instances of design patterns. However, detection approaches differ in their input, extraction methodology, case studies, recovered patterns, system representation, accuracy and validation method.

In fact, the field of design pattern detection still faces a number of key challenges, such as the current detection approaches are working independently from each other, no standard benchmarks nor references to validate the recovered instances and the possible variants of the design pattern. In addition, the evaluation of design pattern detection approaches is somehow difficult since most of the current detection tools are not publicly available.

Design pattern detection approaches perform similar major steps for patterns recognition. These steps are related to information extraction from source code, archetype detection and results representation. This survey only focuses on Gang of Four [1] (GoF) design patterns. Of course, this does not suggest that GoF's patterns are better than other types of patterns (for example, architectural patterns and idiom patterns).

Some detection approaches applied the experiments on small size programs using a few patterns and they achieved high precision and recall rates. In addition, most of the detection approaches rely on code level for patterns detection, which uses the source code as input and represents it in one of the parsing formats (Abstract Syntax Tree (AST) or Abstract Syntax Graph (ASG)). The parsing and modeling techniques affect the accuracy of the detection process.

An empirical review and evaluation of existing detection approaches are important to guide the researcher through the weaknesses of the current detection approaches. This paper provides the current state of the art of design pattern detection approaches. To the best of our knowledge, there is no comprehensive study to classify design pattern detection approaches.

The rest of this paper is organized as follows: Section Two presents the sources of the state of the art selection. An overview of the most important detection approaches is presented in Section Three. Section Four presents analysis and discussion on the selected detection approaches. Section Five presents the lessons learned. Finally, Section Six presents the conclusion and the guidelines for future research directions.

2. STATE OF THE ART SELECTION

We reviewed 80 papers published in highly ranked conferences and journals from 1996 until 2015. However, 34 design pattern detection approaches have been selected and their recovery is part of our statistical analysis. The selection of these approaches was made based on their novelty, results confidently, the area of concern (GoF) and the publisher rank. Some approaches are excluded since they are not focusing on GoF patterns, they are extracting a few patterns which are easy to detect and their results are not validated. Table 1 shows the sources of selected papers.

Journals	IEEE Transactions on Software Engineering
	ACM Transaction on Software Engineering
	Empirical Software Engineering Journal
	Journal of Systems and Software
	Journal of Information and Software Technology
	The Journal of Object Technology
	Advances in Engineering Software
	Journal of Software Engineering and Applications
	The International Journal on Software Tools for Technology Transfer
Conferences	International Conference on Software Engineering (ICSE)
	Working Conference on Reverse Engineering (WCRE)
	International Conference on Automated Software Engineering
	Software Engineering and Knowledge Engineering
	Asia-Pacific Software Engineering Conference

TABLE 1. Sources of selected papers.

As Table 1 illustrates, all highly ranked journals and conferences are included in this survey. Most of the detection approaches are published in IEEE journal and conferences. In addition, two Ph.D. dissertations are included in this survey (MARPLE and HEDGEHOG).

3. OVERVIEW OF DETECTION APPROACHES

Design pattern detection approaches could be classified based on different criteria and aspects. This paper categorized the detection approaches based on their detection methodology and their analysis style.

3.1 Detection Methodology

Since the introduction of design patterns in 1995, different approaches have been developed to extract their instances from the source code. Most of these approaches use similar key steps which aim to match the source code representation to the GoF's catalog representation. The detection methodologies could be categorized based on their key recovery steps into four main groups: database query approaches, metrics-based approaches, UML structure, graph and matrix-based approaches and miscellaneous approaches.

3.1.1 Database Query Approaches

Database query approaches transform the source code into an intermediate representation, such as AST, ASG, UML structures, XMI, etc. SQL queries are used to extract pattern information from the generated representation. The database in use affects the performance of the queries. Unfortunately, database query approaches are not able to recover the instances of behavioral patterns.

The approach presented by Rasool *et al.* [2] used annotations, regular expressions and database queries to recover the instances of design patterns. The varying features of patterns are defined and rules are applied to match these features with the source code elements. The time and the search space are reduced by using appropriate semantics from large legacy systems. Rasool *et al.* approach only recovers specific patterns and its accuracy and efficiency are not reported.

Stencel and Wegrzynowicz [3] present a pattern recognition method to detect non-standard implementations of design patterns as well as the standard implementations. The Detection of Diverse Design Pattern Variants tool (D3) has been developed to implement the detection methodology. In addition, a simple program meta-model has been generated to hold program's core elements, such as attributes, operations and instances. D3 detected the creational instances of design patterns from Java source code using static analysis and SQL. The execution time is only reported, where D3 took 36 seconds to recover the creational instances from JHotDraw.

Marek Vokac constructed a tool to recover specific design patterns from C++ source code [4]. The tool relies on the descriptions of the structural signatures associated with the chosen design patterns. UNDERSTAND FOR C++ parser [5] has been used to generate a file that stores entities and references' data. The entities and references are transferred into an SQL database, which contains tables that correspond to the entities and references. In addition, the SQL table involves links to some files and metrics. The recognition of design patterns is done by a series of SQL statements designed to look for a given structure. The experiments are only conducted on Customer Relationship Management system.

SPOOL (Spreading Desirable Properties into the Design of Object-Oriented, Large Scale Software Systems) is a joint research project between the University of Montreal and Bell Canada. SPOOL environment comprises functionality for design composition, change effect analysis and detection of design patterns [6].

SPOOL extracted different source code information, such as classes, structures, attributes, parameters, return types, call actions, object instantiations and friendship relations. Design

patterns recovery aims to structure parts of class diagrams and resemble pattern diagrams. SPOOL support manual and automatic design patterns recovery. Moreover, SPOOL introduces the concept of the reference class (the most characteristics class that reflects the class behavior). SPOOL environment was applied to three industrial systems. For confident reasons, System A and System B are used to represent the first and second systems respectively. The third system is ET++ v3.0. The efficiency and accuracy of SPOOL are not reported.

3.1.2 Metrics-Based Approaches

Metrics-based approaches compute program related metrics, such as aggregations, associations and dependencies from different source code representations. In addition, different techniques are applied to compare pattern metric values with source code metrics. Metrics-based approaches reduce the search space through filtration.

MAISA (Metrics for Analysis and Improvement of Software Architecture) is a research tool developed at the University of Helsinki [7]. MAISA represents the detection of design patterns as a constraint satisfaction problem (CSP). In CSP, a large number of problems are represented as a set of constraints over variables in a particular domain. Metric prediction attributes are stored in a library and the user can select the pattern that he wants to search for. MAISA will search for the selected pattern and provides each match as a potential candidate. MAISA comprises a UML editor, pattern library, pattern miner, metric analyzer and reporting tool. MAISA was applied to Nokia's DX200 switching system and two instances of abstract factory design pattern are extracted.

FUJABA is a design patterns detection tool where design patterns are defined as sub-patterns [8]. FUJAPA applied transformation rules to capture structural and behavioral aspects of design patterns. Transformation rules are organized into multiple levels of hierarchies. For example, level one of the hierarchy holds the source code information. In addition, FUJABA used a combined bottom up and top down strategy to apply the transformation rules. The detection algorithm uses the assigned level numbers, which are associated with the transformation rules, to establish the orders of applying the rules on ASG. In addition, FUJABA uses fuzzy values to accept or reject the detected pattern elements (sub-patterns). The use of sub-patterns makes the detection process incremental. Hence, relevant information can be achieved in a short time. FUJABA is a semi-automatic tool, which needs the intervention of software engineer.

The approach presented by Antoniol *et al.* [9] generates an Abstract Object Language (AOL) representation for the source code and the design of the system under study. Class level metrics, such as a number of aggregations, associations and inheritances are computed as well. Specifically, a brute force approach to identify all possible pattern candidates was adopted. To identify all pattern candidates in a design containing N classes, all possible arrangements of the classes and their relationships are computed. The experiments have been performed on a public domain code and industrial code to assess the approach effectiveness. The reported precision was 55%.

The approach in [10] combines both clustering based and pattern based reverse engineering approaches. The approach shows that the occurrences of bad smells in the code of software system can falsify the results of a metric based clustering. Moreover, the approach applies pattern detection to an initial decomposition of the system to detect bad smells, which prevent the clustering algorithm to perform a further decomposition.

The technique presented by Uchiyama *et al.* (hereafter, Uchiyama technique) uses source code metrics and machine learning to detect design patterns [11]. By using Goal Question Metric method (GQM), some source code metrics are selected to judge roles. Pattern specialists define a set of questions to be evaluated and select some metrics to help to answer these questions. Moreover, Uchiyama technique uses a hierarchical neural network simulator in which the input is metric measurements of each role and the output is the expected role. The detection was done by matching the candidate roles, which are produced by the machine learning simulator, to the

pattern structure definitions. Searching is looking for all possible combinations of candidate roles that are in agreement with pattern structures. Uchiyama technique extracted inheritance, interface implementation and aggregation relationships. The reported precision and recall rates were 63% and 76% respectively.

3.1.3 UML Structure, Graph, and Matrix-Based Approaches

These approaches represent the structural and behavioral information of the targeted system as UML structure, graph or matrix. Most of these approaches have good precision and recall rates but they are not capable of handling the implementation variants of design patterns.

Seemann and Gudenberg in their work showed how to recover design information from Java source code [12]. A compiler collects the relationships information (method calls and inheritance hierarchies). The result of the compile phase is a graph. A filtering is made to the graph to detect design patterns. Seemann and Gruenberg's approach only detect Strategy, Bridge and Composite design patterns.

DEPAIC++ (DEsgin PAtterns Identification of C++ programs) is a design patterns detection tool developed by Espinoza *et al.* in 2002 [13]. DEPAIC++ is a canonical model formulated to analyze the structure of C++ classes. In addition, DEPAIC++ verifies whether the code being analyzed is using or not design patterns. DEPAIC++ composed by two modules that first transform C++ code into a canonical form and then, recognize design patterns. However, DEPAIC++ did not analyze the behavior of the source program. It detects design patterns starting from a structural analysis of source code whereas some design patterns implement different behaviors in their solutions.

Columbus is a reverse engineering framework developed at the University of Szeged to analyze C++ projects [14]. The extracted information is presented as Columbus schema for C++. The schema represents C++ elements at different levels of abstraction. Moreover, the schema description is represented using UML class diagram. The operation of Columbus is performed using three plug-ins:

- Extraction plug-in, which analyzes the C++ source file and creates a file to store the extracted information. Columbus reads the input files and passes them to the extractor, which will generate the appropriate internal representation. Furthermore, the C++ extractor uses a separate program called CAN (C++ ANalyzer) to parse the input source file.
- Linker plug-in: the main task of the linker plug-in is to build, in the memory, a complete internal representation of the project. Columbus applied different filtering methods, such as filtering using C++ elements categories, filtering by input source files and filtering by scopes.
- Exporter plug-in: the exporter plug-in exports the internal representation to a given output format (for example, HTML, Graphic Exchange Language (GXL) and MAISA).

Columbus extraction capabilities were applied on three C++ projects: IBM Jikes Complier, Leda Graph Library and Star Office Writer.

Design patterns detection using Similarity Scoring Approach (SSA) is a research prototype developed in Java at the University of Macedonia to handle the problem of multiple variants of design patterns [15]. SSA describes the design patterns to be detected, as well as the system under study, as graphs. In addition, SSA represents all system static information as a set of matrices.

SSA uses a graph similarity algorithm to detect design patterns by calculating the similarity of vertices between the pattern and the system under study. To handle the system size problem, SSA divides the system into a number of subsystems and the similarity algorithm is applied to the subsystems instead of the whole system. SSA was applied to JHotDraw v5.1, JRefactory v2.6.24 and JUnit v3.7. Results were validated against the documentation of the systems.

Moreover, SSA uses matrices to represent the relationships between classes, which are directed graphs that can be mapped into a square matrix. To preserve the validity of the results, SSA similarity scores were bound within the range [0, 1]. In fact, SSA has a number of limitations. For example, SSA assumes that no more than one characteristic for a given design pattern instance is modified. To distinguish true positives from false positives, SSA uses a threshold value. For example, the pattern roles that have two characteristics, a threshold equals to 0.5 is assigned. In addition, SSA cannot detect the characteristics that are external to the subsystem boundaries such as chain of delegations and SSA does not employ any dynamic information.

However, SSA shows that the use of similarity algorithm produces more accurate results than the use of exact/inexact graph matching.

Design Patterns Discovery Matrix (DP-Miner) was developed at the University of Texas as a research prototype to detect design pattern instances [16]. DP-Miner represents design pattern structural characteristics as matrices and weights. Specifically, DP-Miner extraction methodology relies on calculating class weights and construction of relationship matrix. The weight of each class provides an indication of the number of attributes / operations in each class and its relationship with other classes. DP-Miner extracted Adapter, Bridge, Strategy and Composite instances from Java AWT package. Results validation is performed by manual tracing of Java AWT package and by referring to its documentation.

The approach presented by Dongjin *et al.* involves a sub-patterns representation for the 23 GoF design patterns – henceforth the sub-patterns approach - [17]. The source code and predefined GoF patterns are transformed into graphs with classes as nodes and relationships as edges. The instances of sub-patterns are identified by means of subgraph discovery. The joint classes have been used to merge the sub-pattern instances. Moreover, the behavioral characteristics of method invocations are compared with predefined method signature template of GoF patterns to obtain final instances. The sub-patterns approach introduces a structural feature model to represent GoF design patterns. The structural feature model extracted four main relationships: inheritance, aggregation, association and dependency. In fact, the sub-patterns approach defined 15 sub-patterns to represent GoF design patterns. A class-relationship directed graph has been used to represent the classes and their relationships.

The sub-patterns approach has been applied to nine open source systems and a Design Pattern Instances Detection tool has been developed as well. Precision, recall and F-measure metrics were used to assess the detection accuracy. Moreover, the execution time for the instances recovery, structural analysis and behavioral analysis is calculated. As it was reported by the authors, the sub-patterns approach spent longer time on method signature analysis than structural analysis. The validation of the results is performed manually and the repository of Perceron [18] has been used as a reference benchmark.

3.1.4 Miscellaneous Approaches

These approaches are not fit under any of the previous categories. Following is a brief description of each approach of this category.

One of the first approaches to detect design patterns was presented by Kraemer and Prechelt in 1996 [19]. They tried to improve the software maintainability through the detection of design patterns directly from C++ source code. Design patterns are represented as Prolog rules, which are used to query a repository of C++ codes. The detection process focused on five structural design patterns: Adapter, Bridge, Composite, Decorator and Proxy. Kraemer and Prechelt approach was applied to four real projects: NME, ACD, LEDA and zApp class library. The reported precision was 14-50%.

PTIDEJ (Pattern Traces Identification, Detection and Enhancement in Java) was developed at the University of Montreal using Java under the Eclipse platform and since then, PTIDEJ has evolved into a complete reverse engineering tool. PTIDEJ comprises several identification algorithms for

design patterns, micro patterns and idiom patterns [20][21]. PTIDEJ considers design pattern detection as a constraint satisfaction problem (CSP) in which decisions are made during the variable assignment phase. PTIDEJ used explanation-based constraint programming to identify micro-architectures similar to design motifs. Micro-architecture describes the organization (structure) of a set of classes of an object-oriented program.

CrocoPat is a tool for design pattern detection developed at the Technical University of Cottbus [22]. It represents the software metamodel in terms of relations. Design patterns are described by relational expressions. The main motivation of building CrocoPat is to handle the performance problem of the previous detection tools. The metamodel presented by CrocoPat divides the object-oriented program into packages, classes, methods and attributes. CrocoPat recovers design pattern instances using three main steps:

- Extraction of source code data using a program analysis tool (sotograp). The extracted data will be stored in a relation file.
- Creation of pattern definitions using pattern specification language. The CrocoPat's language uses relational algebra expressions to express the pattern definitions. The syntax and semantic of the expressions are also defined. Specifically, CrocoPat defines U (Universe) as a set of all values and X as a finite set of all attributes. A tuple t of X is a total function t: X→ U. Val (X) is the set of all tuples of X.
- Extraction of the call, inherit and contain relationships.

CrocoPat is only evaluated in terms of performance. The reported results only show the detection of the Composite and Mediator instances in Mozilla, JWAM and wxWindows.

System for Pattern Query and Recognition (SPQR) is a toolset for elemental design pattern detection in C++ source code developed at the University of Carolina [23]. SPQR uses a logical inference system to encode the rules, which will be combined later to form patterns using reliance operators, and to encode the structural/behavioral relationships between classes and objects using rho-calculus. SPQR is only applied to Killer Widget Application and the Decorator design pattern is extracted. SPQR results are only validated in terms of efficiency (CPU times and memory consumption).

Pattern Inference and Recovery Tool (PINOT) reclassifies the catalog of design patterns by intent [24]. PINOT was built from Jikes, open source java compiler, and focuses on the detection of common design patterns used in practice. To capture program intent, PINOT used static program analysis techniques to recover design pattern instances from four open source projects: Java AWT v1.3, JHotDraw v6.0, Java Swing v1.4 and Apache Ant v1.6. Structural driven patterns are detected using inter-class relationships. During the structural analysis, the virtual delegations, call dependencies, context interfaces, associations, aggregations, factory interfaces and singleton class structures are identified. Furthermore, PINOT used data flow analysis on Abstract Syntax Trees (ASTs) in terms of blocks to detect behavioral driven patterns. Method bodies are represented as a Control Flow Graph (CFG). The CFG is scanned later to determine method behaviors. The authors of PINOT only reported the required CPU times to detect the structural and behavioral driven patterns.

DeMIMA (Design Motif Identification: Multilayered Approach) is a semi-automatic tool, developed at the University of Montreal, that identifies microarchitectures similar to design motifs in the source code [25]. DeMIMA involves three layers: two layers to generate source code abstract model and class relationships and one layer to recognize design patterns from the generated abstract model. DeMIMA was implemented in Java on top the of PTIDEJ framework [20]. In addition, DeMIMA uses explanation-based constraint programming to handle the constraint satisfaction problem. DeMIMA identifies micro-architectures similar to the design motifs by transforming them into constraints that reflect the relationships between pattern's participant classes. The used constraints are inheritance constraint, strict transitive inheritance constraint, transitive inheritance constraint, use constraint, ignorance constraint and creation constraint.

Design Patterns Recovery Environment (DPRE) is a design pattern recovery prototype developed at the University of Salerno [26]. DPRE uses a two-phase approach to recover structural design patterns from object-oriented code. Figure 1 shows DPRE recovery process. DPRE phase one provides a coarse-grained level where design pattern candidates are identified by analyzing class diagram information extracted during the preliminary analysis. In the second phase, codes of the classes that participate in design pattern identification are examined to check their compliance with the corresponding GoF patterns' source code. The effectiveness of DPRE is characterized by precision, which is ranging from 62% to 97%.

Zanoni introduced an Eclipse plug-in called MARPLE (Metrics and Architecture Reconstruction Plug-in for Eclipse), which supports both the detection of design pattern instances and software architecture reconstruction activities [27]. MARPLE tries to handle the variant problems of design patterns detection through the detection of sub-components called "basic elements". The architecture of MARPLE involves five main modules that interact with each other through XML data transfer.

The approach presented by Alnusair *et al* [28] - henceforth Sempatrec - uses ontology formalism to represent the conceptual knowledge of the source code and semantic rules to capture the structure and behavior of design patterns.

A tool named Sempatrec (SEMantic PATtern RECovery) has been developed as a plug-in for the Eclipse IDE to implement the approach. Sempatrec processes the Java bytecode of the targeted software, generates an RDF (Resource Description Framework) ontology and stores the ontology locally in a pool.

Specifically, Sempatrec generates a source code representation ontology (SCRO) to provide an explicit representation of the conceptual knowledge structure found in the source code. However, the developed SCRO serves as a basis for design pattern recovery where a design pattern ontology sub-model will be created. This sub-model extends the SCRO's vocabularies and involves an upper design pattern ontology that is further extended with a specific ontology for each design pattern.

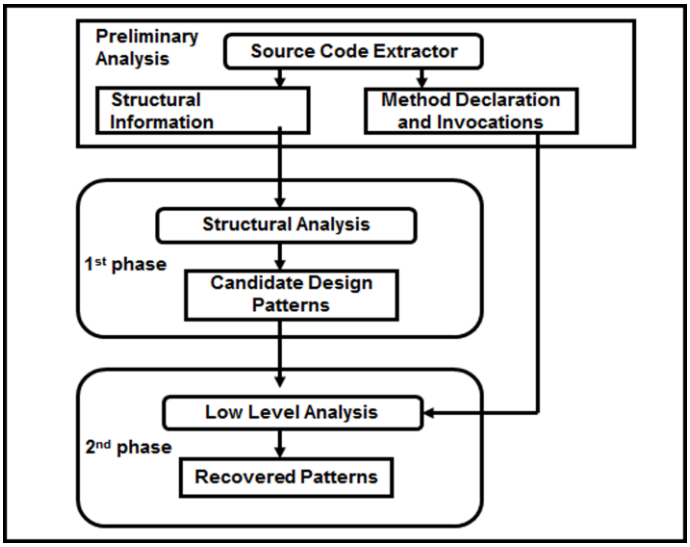

FIGURE 1: DPRE Recovery Process.

Table 2 summarizes the whole spectrum of design pattern detection approaches. Some of the miscellaneous approaches are listed in table and do not appear in this section (Pat [19], KT [29], DP++ [30], Kim and Boldyreff [31], Heuzeroth *et al.* [32], Philippow *et al.* [33], HEDGEHOG [34],

and Kaczor *et al.* [35]). However, ALL table approaches are involved in the statistical analysis of this survey.

3.2 Analysis Style

Based on the analysis style performed, pattern detection approaches could be classified into structural analysis approaches, behavioral analysis approaches and semantic analysis approaches.

Structural analysis approaches detect the instances of design patterns based on the static parts of the system under study. They explore inter-class relationships, method invocations and data types.

Behavioral analysis approaches consider the execution behavior of the program. The behavioral aspects of the program are extracted by using static and dynamic analysis techniques. The behavioral analysis is useful since the structure of the patterns is not enough to provide a fingerprint inside the source code. For example, State and Strategy patterns have similar structures. Similarly, Chain of Responsibility, Proxy and Decorator patterns have identical structures.

However, the possible variants of the same implemented behavior increase the number of false positive instances.

Semantic analysis complements the structural and behavioral aspects to reduce the number of false positive instances. Naming conventions and annotations were used to retrieve the role information. Semantic information is important to distinguish between design patterns that have identical structural and behavioral aspects, such as State, Strategy and Bridge. Table 2 shows the analysis style used by each detection approach.

Detection Methodology	Tool/ Author	Year	Analysis Style	R
Database Query Approaches	Rasool *et al.*	2010	ST, SE	[2]
	D3	2008	ST, BE	[3]
	Marek Vokac	2006	ST	[4]
	SPOOL	1999	ST	[6]
Metrics-Based Approaches	MAISA	2000	ST	[7]
	FUJAPA	2002	ST,BE	[8]
	Antoniol *et al.*	1998	ST,BE	[9]
	Detten and Becker	2011	ST	[10]
	Uchiyama *et al.*	2014	ST,BE	[11]
UML Structure, Graph and Matrix Based Approaches	Seemann and Gudenberg	1998	ST,SE	[12]
	DEPAIC++	2002	ST	[13]
	Columbus	2002	ST	[14]
	SSA	2006	ST	[15]
	DP-Miner	2007	ST,BE,SE	[16]
	Dongjin *et al.*	2015	ST,BE	[17]
Miscellaneous Approaches	Pat	1996	ST	[19]
	PTIDEJ	2001 2004	ST	[20][21]
	CrocoPat	2003	ST	[22]
	SPQR	2003	ST	[23]
	PINOT	2006	ST,BE	[24]
	DeMIMA	2008	ST	[25]
	DPRE	2009	ST	[26]
	MARPLE	2012	ST,BE	[27]
	Sempatrec	2014	ST,SE	[28]
	KT	1996	ST	[29]
	DP++	1998	ST	[30]
	Kim and Boldyreff	2000	ST	[31]
	Heuzeroth *et al.*	2003	ST,BE	[32]
	Philippow *et al.*	2005	ST	[33]
	HEDGEHOG	2005	ST,BE,SE	[34]
	Kaczor *et al.*	2006	ST	[35]

Note: **ST:** Structural Analysis **BE:** Behavioral Analysis
SE: Semantic Analysis **R:** Reference

TABLE 2: Summary of detection approaches based on their detection methodology and analysis style.

4. ANALYSIS AND DISCUSSION

Design patterns are flexible design templates that may have several implementations. However, design patterns are described informally, which may cause misunderstanding. New approaches

and tools are continuously proposed with the new trend of applying new technologies. This section aims to provide a comprehensive comparison between all design pattern detection approaches in terms of system representation, case studies, recovered design patterns and evaluation criteria.

4.1 Intermediate Representation of the Source Code

To the best of our knowledge, all the detection approaches in the literature are targeting the source code of the system under study and avoid targeting the system's design model to extract the instances of design patterns. The design model does not provide any runtime data, which are necessary for the extraction of design patterns (for example, the association relationships). Normally, the design documents are inconsistent with the source code. Furthermore, most of the design models are not publicly available. All these reasons made the source code better choice than the design model to extract the instances of design patterns.

Most design pattern detection approaches use Abstract Syntax Tree (AST) representation to generate the source code model. The source code model holds all the required information to recover design pattern instances. Table 3 lists the intermediate representation used by different detection approaches.

Some approaches used their own defined representation, such as [20], [25] and [35]. These approaches defined PADL, Pattern and Abstract Level Description Language, to extract the source code information. Two approaches did not generate an intermediate representation of the source code, [11] and [31], and they used software metrics to gather source code information. However, each detection approach may use the same representation in a different format. For example, DPRE [26] uses AST representation to generate a graph that represents class diagrams of the systems. On the other hand, Heuzeroth *et al.* [32] use AST representation to define the static aspects of the patterns and the Temporal Logic Actions (TLA) to represent their dynamic aspects.

4.2 Targeted Systems (Case Studies)

The majority of the detection approaches targeted open source codes that have been programmed using Java or C++. Two approaches, MAISA [7] and DP-Miner [16], targeted UML and XML open source systems. KT [29] applied its detection methodology on Smalltalk programs. Only one approach, CrocoPat [22], conducted its experiments on both Java and C++ open source systems. Figure 2 shows the programming languages used to program the targeted systems.

In fact, the majority of the detection approaches that have been developed after 2008 applied their experiments on Java open source programs. This could facilitate the comparison between detection approaches.

Furthermore, the detection approaches used different open source systems to evaluate their methodologies. The most commonly used open source systems are JHotDraw v5.1, JRefactory v2.6.24, JUnit v3.7, Java AWT package and QuickUML 2001. The selection of these approaches were made by the detection approaches due to the following reasons:

- They used some well-known design patterns.
- The authors and the relevant literature indicate explicitly the implemented design patterns in the documentation.
- They are open source and their codes are publicly available.
- They vary in size.

Table 4 lists the case studies used by different detection approaches to evaluate the detection methodology. It appears clearly that there is no common agreement in the literature on the suitable case studies to evaluate any new detection approach. In addition, the number of required case studies is not clear. For example, some approaches use more than 5 case studies while

other approaches only use two case studies. DeMIMA [25] applied its methodology to 33 industrial components, but there is no information about them.

System Representation	Author(s)/Tool
AST (Abstract Syntax Tree)	Antoniol *et al.* [9], Detten and Becker [10], PINOT [24], DPRE [26], MARLPE [27], KT [29], Heuzeroth *et al.* [32], HEDGEHOG [34]
ASG (Abstract Syntax Graph)	FUJABA [8] Columbus [14]
UML, Graph	SPOOL [6], Seemann and Gudenberg [12], Dongjin *et al.* [15], DP++ [30], Philippow *et al.* [33].
Matrix	SSA [15] DP-Miner [16]
Prolog	MAISA [7] Pat [19]
PADL	PTIDEJ [20], DeMIMA [25], Kaczor *et al.* [35].
Metadata	D3 [3] Marek Vokac [4]
Other representations	Canonical form (DEPAIC++ [13]) Annotations (Rasool *et al.* [2]) BDDs (CrocoPat [22]) OTTER (SPQR [23]) SCRO (Sempatrec [28])
No representation	Uchiyama *et al.* [11], Kim and Boldyreff [31]

TABLE 3: The Intermediate representation used by existing approaches.

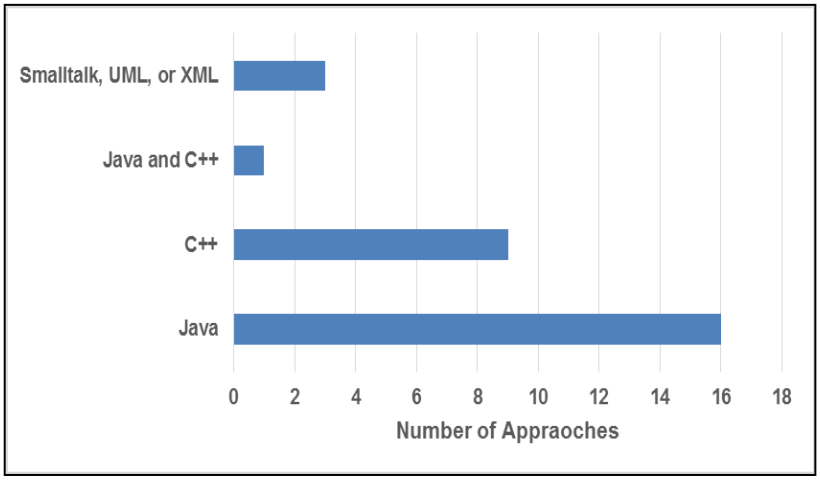

FIGURE 2: Programming languages used to program the targeted systems.

Tool/ Author	Case Studies
Rasool *et al.* [2]	JHotDraw v6.1.2 and Apache Ant v1.6.2.
D3 [3]	Applied Java Patterns and JHotDraw v6.0.b1
Marek Vokac [4]	Customer Relationship Management system
SPOOL [6]	ET++ and two telecommunication systems
MAISA [7]	Nokia DX200 switching system
FUJAPA [8]	Java AWT
Antoniol *et al.* [9]	LEDA, Libg++, Galib, Mec, Socket and 8 small size industrial systems.
Detten and Becker [10]	Common Component Modeling Example
Uchiyama *et al.* [11]	Java library v1.6.0, JUnit v4.5 and Spring v2.5
Seemann and Gudenberg [12], DEPAIC++ [13]	Not mentioned
Columbus [14]	IBM Jikes complier, Leda graph library and Star office writer
SSA [15]	JHotDraw v5.1, JRefactory v2.6.24 and JUnit v3.7.
DP-Miner [16]	Java AWT
Dongjin *et al.* [17]	Java AWT v5.0, JHotDraw v5.1, JUnit v3.8, Dom4J v1.6.1, Lizzy v1.1.1, Hodoku v2.1.1, Barcode4j v2.1.0, RstpProxy v3.0 and Teamcenter
Pat [19]	NME, LEDA and zApp
PTIDEJ [20][21]	Java AWT, Java.net packages, JHotDraw v5.1, JRefactory v2.6.24, JUnit v3.7, Lexi v0.0.1α, Netbeans v1.0.x and QuickUML 2001
CrocoPat [22]	Mozilla, JWAM and wxWindows
SPQR [23]	Killer Widget Application
PINOT [24]	Java AWT v1.3, JHotDraw v6.0, Java Swing v1.4 and Apache Ant v1.6
DeMIMA [25]	JHotDraw v5.1, JRefactory v2.6.34, JUnit v3.7, MapperXML v1.9.7, QuickUML 2001 and 33 industrial components
DPRE [26]	JHotDraw v5.1, Apache Ant v1.6.2, JHotDraw v6.0b1, QuickUML 2001, Swing and Eclipse JDT components (Core v3.3.3 and User Interface v3.3.2).
MARPLE [27]	30 open source projects
Sempatrec [28]	JHotDraw v5.1, JRefactory v2.6.24 and JUnit v3.7
KT [29]	KT and three Smalltalk programs
DP++ [30]	DTK library
Kim and Boldyreff [31]	Three systems (no information about them)
Heuzeroth *et al.* [32]	Java swing
Philippow *et al.* [33]	Students projects
HEDGEHOG [34]	AJP code example, pattern box and java language (v1.1 and v1.2)
Kaczor *et al.* [35]	JHotDraw v5.1, QuickUML 2001 and Juzzle

TABLE 4: Summary of the case studies conducted by detection approaches.

4.3 Recovered Design Patterns

Table 5 shows a summary of the recovered design patterns extracted by different detection approaches. Most of the approaches successfully detect the Composite design pattern. This is because its structure is easy to detect. On the other hand, the Memento and Interpreter design patterns are only detected by three approaches, since they require dynamic analysis capabilities for the extraction process. However, most of the detection approaches focused on a specific set of design patterns.

Moreover, as Table 5 illustrates, only three approaches successfully detect all GoF design patterns. Specifically, Kim and Boldyreff [31] extracted all GoF design patterns from three systems which are programmed using C++. Unfortunately, there is no information about these systems. In addition, Philippow *et al.* [33] extracted all GoF design patterns from student projects, which are also programmed using C++. The main disadvantage of the previous two approaches is their results validation. In fact, it is not clear how the extracted pattern instances are validated.
Dongjin *et al.* [17] extracted all GoF design patterns from Java open source projects by using sub-patterns and method signatures. Dongjin *et al.* used the repository of Perceron [18] as a benchmark to validate the extracted instances. However, the experimental results presented by Dongjin *et al.* seem to be not accurate. For example, the reported experimental results involve Java AWT and JHotDraw open source systems, which are not listed by Perceron. In addition, Dongjin *et al.* approach recovered two Factory method instances (after behavioral analysis) from JUnit and Percerons only reported one true positive Factory instance. Dongjin *et al.*'s approach reported the precision and recall for the Factory method detection as 100%.

4.4 Evaluation Criteria

Precision and recall metrics have been used by most of the approaches to evaluate the accuracy of the detection process. A few approaches reported the F-measure, which provides the harmonic means of recall and precision. Accuracy differs from one approach to another since some approaches extracted a few patterns and achieved high precision. The validation method, pattern definitions and pattern variants could also affect the detection accuracy. The precision, recall and F-measure are calculated as follows [36]:

Precision = [True Positives / (True Positives + False Positives)] %
Recall = [True Positives / (True Positives + False Negatives)] %
F-measure = 2 × [(Precision×Recall) / (Precision+Recall)] %

Where:
True positives: the number of instances, which are correctly detected.
False positives: the number of instances, which are incorrectly detected.
False negatives: the number of instances, which are incorrectly rejected.

The reported accuracy for the majority of detection approaches in the literature is presented in Table 6. As Table 6 illustrates, the reported accuracy for most of the detection approaches is not balanced (i.e. high precision and low recall or vice versa). The main reason of this would be the high differences between the number of correctly detected instances and the number of missed instances. Specifically, the unbalanced accuracy suggests that there is no trade-off between the number of correctly detected instances and the number of rejected instances (missed instances).
Some approaches only reported the number of true positives and true negatives, such as D3 [3], Marek Vokac [4] and Heuzeroth *et al.* [32]. On the other hand, some approaches used CPU times, such as Rasool *et al.* [2], DP-Miner [16], CrocoPat [22], SPQR [23] and PINOT [24] to evaluate their detection efficiency. For example, PINOT spent 66.79 seconds, 8.98 seconds, 10.68 seconds, and 12.58 seconds to detect design pattern instances from Swing, JHotDraw, Java AWT and Ant respectively.

Furthermore, most detection approaches validated their results based on manual tracing of the source code and they achieved high accuracy. On the other hand, only two approaches, Dongjin *et al.* [17] and Sempatrec [28], validated their results based on design pattern repositories, such as the repository of Perceron [18] and the design pattern detection tools benchmark platform [37]. Consequently, different accuracy values achieved by different approaches, since there is no standard benchmark to validate the extracted design pattern instances.

R / DP	[2]	[3]	[4]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]	[17][31][33]	[19]	[20][21]	[22]	[23]	[24]	[25]	[26]	[27]	[28]	[29]	[30]	[32]	[34]	[35]
SI	✓	✓	✓						✓				✓		✓		✓			✓	✓		✓	✓				✓	
FM	✓	✓	✓	✓								✓	✓		✓		✓			✓	✓		✓	✓				✓	
AF		✓			✓						✓				✓					✓			✓	✓				✓	✓
BU		✓										✓			✓								✓						
PR												✓	✓		✓		✓				✓		✓					✓	
AD	✓						✓	✓	✓				✓	✓	✓	✓	✓			✓	✓	✓	✓	✓		✓		✓	
BR				✓		✓	✓	✓		✓		✓		✓	✓	✓				✓		✓						✓	
CO	✓					✓		✓		✓	✓		✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓		✓	✓
DE			✓					✓					✓		✓	✓	✓		✓	✓	✓	✓		✓	✓		✓	✓	
FA								✓							✓					✓		✓							
FL								✓							✓					✓						✓		✓	
PR	✓							✓				✓			✓	✓				✓		✓	✓				✓	
CoR															✓					✓							✓		
CM													✓		✓		✓				✓								
IT											✓				✓													✓	
IN															✓														
ME															✓			✓		✓							✓		
MN															✓														
OB	✓		✓										✓		✓		✓			✓	✓			✓			✓	✓	
ST									✓				✓		✓		✓			✓	✓			✓				✓	
SR						✓			✓	✓		✓	✓	✓	✓		✓			✓	✓			✓				✓	
TM			✓	✓					✓			✓	✓		✓		✓			✓	✓			✓	✓			✓	
VI	✓												✓		✓		✓			✓	✓			✓			✓	✓	
Total	7	4	5	3	1	3	2	7	5	3	3	7	12	4	23	5	12	2	1	17	12	6	8	11	3	3	5	16	2

Note:
PR: Prototype **FM:** Factory Method **BU:** Builder **AD:** Adapter **SI:** Singleton
BR: Bridge **DE:** Decorator **PX:** Proxy **CO:** Composite **FA:** Façade
FL: Flyweight **VI:** Visitor **TM:** Template Method **CM:** Command
OB: Observer **ST:** State **SR:** Strategy **IT:** Iterator **AF:** Abstract Factory
CoR: Chain of Responsibility **ME:** Mediator **MN:** Memento **IN:** Interpreter
R: Reference **DP:** Design Pattern

TABLE 5: Summary of design patterns recovered by detection approaches.

Tool/ Author	Precision %	Recall %
Rasool *et al.* [2]	94	92
Antoniol *et al.* [9]	30	Not Mentioned
Uchiyama *et al.* [11]	63	76
SSA [15]	100	66.7-100
Dongjin *et al.* [17]	68-100	73-100
Pat [19]	14-50	Not Mentioned
DeMIMA [25]	34	100
DPRE [26]	62-67	Not Mentioned
MARPLE [27]	76	63
Sempatrec [28]	61-82	88-90
Kim and Boldyreff [31]	43	Not Mentioned
HEDGEHOG [34]	100	85
D3 [3], Marek Vokac [4], SPOOL [6], MAISA [7], FUJAPA [8], Detten and Becker [10], Seemann and Gudenberg [12], DEPAIC++ [13], Columbus [14], DP-Miner [16], PTIDEJ [20][21], CrocoPat [22], SPQR [23], PINOT [24], KT [29], DP++ [30], Heuzeroth *et al.* [32], Philippow *et al.* [33], Kaczor *et al.* [35]	Not Mentioned	Not Mentioned

TABLE 6: Summary of reported accuracy by detection approaches.

5. LESSONS LEARNED

This paper presented a comprehensive comparison between different design pattern detection approaches. The lessons learned can be summarized as follows:

- Design patterns are described from different perspectives by different approaches, such as structural aspects, behavioral aspects and semantic aspects.
- Current detection approaches use different tools to get the intermediate representation of the targeted source code. This will directly affect the recovery process.
- The discovery tool of each approach only supports the discovery of specific patterns. Only a few approaches successfully detected all GoF design patterns.
- Different approaches conduct experiments on different open source systems.
- Recall and precision were used to evaluate the accuracy of the detection process. Only a few approaches reported F-measure, such as Dongjin *et al.* [17] and Sempatrec [28]. In addition, some approaches measure the CPU times and memory consumptions to evaluate their detection efficiency.
- No standard benchmark to validate the extracted design pattern instances. The available benchmarks, to the best of our knowledge, are the repository of Perceron [18], the Design Pattern Detection tools benchmark platform [37], P-MARt [38] and BEFRIEND [39].

6. CONCLUSION

Design patterns detection can help maintainers to understand the design of a program and help to document the systems.

This paper presented the current state of the art of design pattern detection approaches. Specifically, we presented a comparative study on design pattern detection approaches in terms of detection methodology, analysis style, system representation, case studies, recovered design patterns and evaluation criteria. The major contribution of this paper is the necessity to address

all detection approaches and tools. This will guide the researchers in the future to develop more accurate detection tools. In addition, this survey will facilitate the comparison between different detection approaches and any new detection approach, since there is no trusted benchmark to evaluate the recovered design pattern instances. Most design pattern detection approaches target open source systems which do not have proper documentation. It could be worthwhile to conduct the experiments on industrial and commercial applications. In addition, disparity among the results is noticed. The main reason could be the missing roles and the implementation variants of design patterns. Precision and recall were used to evaluate the accuracy of the detection process. However, the reported accuracy is not balanced (i.e. high precision and low recall or vice versa). One possible solution is to use common formalized definition of GoF patterns.

Finally, all detection approaches are working independently without any ability to integrate them together. The research community should put efforts to build new approaches which may be integrated with other existing approaches.

7. REFERENCES

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[2] G. Rasool, I. Philippow, P. Mader, “Design Pattern Recovery Based on Annotations”. International Journal of advances in Engineering Software, Vol 41, Issue 4, 2010, pp. 519-526.

[3] K. Stencel, and P. Wegrzynowicz, “Detection of Diverse Design Pattern Variants”, 15th Asia-Pacific Software Engineering Conference, 2008, pp. 25-32.

[4] M. Vokac, “ An efficient tool for recovering design patterns from C++ code”, Journal of Object Technology, Volume 5, No. 1, 2006, pp. 139–157.

[5] Scientific Tool works Inc. Understand for C++, 2003.

[6] Rudolf K. Keller, Reinhard Schauer, Sebastien Robitaille, and Patrick Page; Pattern based reverse-engineering of design components. In ICSE 99: Proceedings of the 21st International Conference on Software Engineering, pages 226–235, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[7] Paakki J., Karhinen A., Gustafsson J., Nenonen L. and Verkamo A.I., Software metrics by architectural pattern mining, Proceedings of the International Conference on Software: Theory and Practice (16th IFIP World Computer Congress), 2000, 325–332.

[8] Niere, J., Shafer, W., Wadsack, J.P., Wendehals, L., Walsh, J., 2002. Towards pattern design recovery. In: Proceedings of International Conference on Software Engineering (ICSE’02), Orlando, FL, USA, pp. 338–348.

[9] G. Antoniol, R. Fiutem, and L. Cristoforetti, “Design pattern recovery in object-oriented software”, In Proceedings of the 6th international workshop on program comprehension, 1998, pp. 153–160.

[10] M. V. Detten, and S. Becker, “Combining Clustering and Pattern Detection for the Reengineering of Component-based Software Systems”, In Proceedings of the 7th International Conference on the Quality of Software Architectures, QoSA, pp. 23-32, 2011.

[11] Uchiyama, S., Kubo, A., Washizaki, H., and Fukazawa, Y. (2014). Detecting Design Patterns in Object-Oriented Program Source Code by Using Metrics and Machine Learning. Journal of Software Engineering and Applications, 7, 983-998. doi: 10.4236/jsea.2014.712086.

[12] Jochen Seemann and Juergen Wolff von Gudenberg. Pattern-based design recovery of java software. In SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering, pages 10–16, New York, NY, USA, 1998. ACM Press.

[13] Felix Agustin Castro Espinoza, Gustavo Nuez Esquer, and Joel Surez Cansino. Automatic design patterns identification of C++ programs. In EurAsia-ICT 02: Proceedings of the First EurAsian Conference on Information and Communication Technology, pages 816–823, London, UK, 2002. Springer-Verlag.

[14] Ferenc, R., Beszedes, A., Tarkiainen, M., Gyimothy, T.: Columbus—reverse engineering tool a schema for C++. 18th IEEE international conference on software maintenance (ICSM'02), pp. 172–181, October 2002.

[15] Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., Halkidis, S.: Design Pattern Detection Using Similarity Scoring. IEEE Transaction on Software Engineering 32(11) (2006).

[16] Dong, J., Lad, D.S., Zhao, Y.: Dp-miner: Design pattern discovery using matrix. In: Proc. 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, ECBS 2007, pp. 371–380 (2007).

[17] Dongjin Yu, Yanyan Zhang, and Zhenli Chen: A comprehensive approach to the recovery of design pattern instances based on sub-patterns and method signatures. The Journal of Systems and Software 103 (2015) 1–16.

[18] Ampatzoglou, A.,Michou, O.,Stamelos, I.,2013b. Building and mining a repository of design pattern instances: practical and research benefits. EntertainmentComput.4, 131–142.

[19] Christian Kramer and Lutz Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In Working Conference on Reverse Engineering, pages 208–1996.

[20] Y.-G. Guéhéneuc and N. Jussien, "Using Explanations for Design Patterns Identification," Proc. First IJCAI Workshop Modelling and Solving Problems with Constraints, C. Bessie`re, ed., pp. 57-64, Aug. 2001.

[21] Y.-G. Guéhéneuc, H. Sahraoui, and F. Zaidi, "Fingerprinting Design Patterns," Proc. 11th Working Conf. Reverse Eng. (WCRE'04), Nov. 2004.

[22] Beyer, D., Lewerentz, C. CrocoPat: efficient pattern analysis in object-oriented programs. In: Proceedings of the International Workshop on Program Comprehension (IWPC'03), Portland, OR, USA, pp. 294–295 (2003).

[23] J. McC. Smith, and D. Stotts. SPQR: Flexible Automated Design Pattern Extraction from Source Code. In Proceedings of the 2003 IEEE International Conference on Automated Software Engineering, Montreal QC, Canada, October, 2003, pp. 215-224.

[24] Nija Shi and Ronald A. Olsson. Reverse engineering of design patterns from java source code. In ASE '06: Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06), pages 123–134, Washington, DC, USA, 2006. IEEE Computer Society.

[25] Y.-G. Guéhéneuc and G. Antoniol, "DeMIMA: A Multi-Layered Framework for Design Pattern Identification," IEEE Trans. Software Eng., vol. 34, no. 5, pp. 667-684, Sept./Oct. 2008.

[26] Lucia, A.D., Deufemia, V., Gravino, C., and Risi, M., Design pattern recovery through visual language parsing and source code analysis, The Journal of Systems and Software, Vol 82, pp. 1177–1193, 2009.

[27] M. Zanoni, “Data mining techniques for design pattern detection,” Ph.D. dissertation, Universita degli Studi di Milano-Bicocca, 2012.

[28] Alnusair, A., Zhao, T., Yan, G., 2014. Rule based detection of design patterns in program code. Int.J.Softw.ToolsTechnol.Trans.16 (3), 315–334.

[29] Kyle Brown. Design reverse-engineering and automated design pattern detection in Smalltalk, Master’s thesis, North Carolina State University, 1996.

[30] Bansiya Jagdish: Automating Design-Pattern Identification. Dr. Dobb’s Journal. June 1998.

[31] Kim, H. and Boldyreff, C. (2000) A Method to Recover Design Patterns Using Software Product Metrics. In: Proceedings of the 6th International Conference on Software Reuse: Advances in Software Reusability, Vienna, 27-29 June 2000, 318-335.

[32] Heuzeroth, D., Holl, T., Hogstrom, G., Lowe, W., 2003. Automatic design pattern detection. In: Proceedings of International Workshop on Program Comprehension (IWPC’03), Portland, OR, USA, pp. 94–103.

[33] Philippow, I., Streitferdt, D., Riebish, M., Naumann, S., 2005. An approach for reverse engineering of design patterns. Software System Modeling 4 (1), 55–79.

[34] A. Blewitt. Hedgehog: Automatic Verification of Design Patterns in Java. PhD thesis, School of Informatics, University of Edinburgh, 2005. http://www.bandlem.com/Alex/Papers/PhDThesis.pdf.

[35] Kaczor O. Guéhéneuc Y-G, Hamel S. Efficient identification of design patterns with bit-vector algorithm. In: Proceedings of the 10th European conference on software maintenance and reengineering, Bari, Italy; 22–24 March 2006. p. 184–93.

[36] W.B Frakes and R.Baeza, Yates, Information Retrieval: Data Structure and Algorithms, Prentice Hall, 1992.

[37] Arcelli Fontana, F., Caracciolo, A., Zanoni, M., 2012. DPB: A benchmark for design pattern detection tools. In: Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR’12). IEEE Computer Society, Szeged, Hungary, pp. 235–244. doi:10.1109/C.

[38] Y.-G. Guéhéneuc, “P-MARt: Pattern-like micro architecture repository,” Proceedings of the 1st EuroPLoP Focus Group on Pattern Repositories, 2007.

[39] Fülöp, L. J., Hegedus, P., & Ferenc, R. (2008). BEFRIEND - A benchmark for evaluating reverse engineering tools. Periodica Polytechnica, Electrical Engineering, 52(3-4), 153-162. DOI: 10.3311/pp.ee.2008-3-4.04.