Review-based Comparison of Design Pattern Detection Tools

RODRIGO MOREIRA, Federal University of Minas Gerais (UFMG)
EDUARDO FERNANDES, Federal University of Minas Gerais (UFMG)
EDUARDO FIGUEIREDO, Federal University of Minas Gerais (UFMG)

Context: Design patterns are reusable solutions for recurring problems of software design. Although useful for software analysis, detecting design patterns is often challenging especially in large and complex software systems. In this context, several tools have been proposed for automating this process. *Objective:* Past attempts to summarize existing detection tools contain gaps in their scope, such as the lack of a comparison of the output provided by the tools in terms of precision and agreement. We address some of these gaps through a literature review and a comparison of design pattern detection tools. Our goal is to assist practitioners and researchers not only looking for useful tools, but also exploring opportunities for their improvements. *Method:* We present a systematic literature review of design pattern detection tools based on strict guidelines. We compare the performance of four tools in detecting six design patterns based on precision, recall, F-measure, and agreement. *Results:* From the 42 tools found, only ten are available for download. Altogether, the tools detect all 23 design patterns summarized by the Gang of Four's book. The comparison results suggest that some tools are more suitable for specific design patterns, e.g., the FINDER tool for Composite, Decorator and Visitor. We also observed a low agreement among tools. *Conclusions:* Despite the high number of tools published, design pattern detection tools are mostly ineffective and unavailable for use. Particularly, practitioners might struggle to find a tool that matches their expectations. The available tools provide inaccurate yet complementary detection results; thus, solutions for either improving or combining tools are needed. Researchers are encouraged to propose novel tools capable of filling this literature gap.

Categories and Subject Descriptors: [**Software and its engineering**] Abstraction, modeling and modularity; [**Software and its engineering**] Software maintenance tools; [**Software and its engineering**] Software design engineering; [**General and reference**] General literature; [**General and reference**] Experimentation

General Terms: Verification
Additional Key Words and Phrases: Software design, design pattern, automated development tool, systematic literature review, comparative study

ACM Reference Format:

Moreira, R., Fernandes, E. and Figueiredo, E. 2022. Review-based Comparison of Design Pattern Detection Tools. HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 29 (October 2022), 16 pages.

1. INTRODUCTION

Software design is the process of translating requirements into a detailed design representation of a software system [Yau and Tsai 1986]. This process includes making decisions on how to organize the source code [Yau and Tsai 1986]. Since these decisions may be difficult, design patterns have been proposed for driving them [Gamma et al. 1995]. A design pattern is a reusable solution for a recurring problem of software design [Gamma et al. 1995]. Examples of design patterns are Composite and Visitor. Composite helps to treat a group of objects as if they were a single object, while Visitor helps to dynamically introduce new operations to an object without changing its structure [Gamma et al. 1995].

Although useful, design patterns are not usually documented in the source code. In particular, analyzing source code and performing changes may require a deep understanding on which and how design patterns occur. In this context, design pattern detection (DPD) may facilitate the software analysis [Kim et al. 2016] and support design decisions [Ampatzoglou et al. 2012; Zanoni et al. 2015]. However, the literature suggests that DPD is challenging due to several reasons, such as the size and complexity of real-world systems [Dong et al. 2007; Wang and Tzerpos 2005]. Thus, software engineers need automated assistance for making DPD easier, faster, and more accurate. Several studies have proposed tools for automating DPD [Wang and Tzerpos 2005; De Lucia et al. 2010a; Dong et al. 2008; Fontana et al. 2012; Fulop et al. 2008]. Unfortunately, past attempts to summarize these tools [Riaz et al. 2015; Yarahmadi and Hasheminejad 2020] contain some gaps in their scope. For instance, they lack a comparison of the output provided by the tools in terms of precision and agreement.

This paper fills these literature gaps by presenting a systematic literature review (SLR) and a comparison of DPD tools. Our goal is to assist practitioners and researchers in choosing tools that meet their needs. We followed strict guidelines for SLRs [Kitchenham and Charters 2007]. We then analyzed the data to understand both nature and scope of existing tools. For instance, we identified the design patterns and programming languages supported by tools. We also compared the performance of four tools – FINDER [Dabain et al. 2015], GEML [Barbudo et al. 2021], MARPLE-DPD [Zanoni et al. 2015], and PTIDEJ [Moha and Guéhéneuc 2007] – in detecting six design patterns: Chain of Responsibility (CoR), Composite, Decorator, Prototype, Singleton, and Visitor. We computed precision, recall, F-measure, and agreement [Gwet 2014].

Our SLR revealed 42 tools although only ten are available online. Altogether, the tools claim to detect all 23 design patterns compiled by the Gang of Four's book [Gamma et al. 1995]. The frequency of published tools remained stable over the years, thereby suggesting this research topic has not yet reached saturation. This assumption is plausible if we consider the poor performance observed for the tools compared in our study. Our results suggest low precision, low recall, and low agreement. They also lack essential features, such as data filtering and results export.

We summarize our study contributions as follows. First, we present an extensive catalog of 42 DPD tools published in the last two decades, each characterized by the supported design patterns, programming languages, etc. Second, we quantitatively compare four state-of-the-art tools based on precision, recall, F-measure, and agreement. Third, we present a list of manually validated instances of six design patterns detected in two Java systems. Fourth, we provide the study replication package [1] containing information, such as the detected instances, our manual classification of true or false positives, the list of papers from the SLR, etc.

The results presented in this paper have multiple implications for both practitioners and researchers interested in DPD. Dozens of DPD tools have been proposed in the last two decades and, still, existing tools provide very limited support and considerably unreliable output. On one hand, practitioners must be aware of possible limitations of the existing tools while picking tools that match their needs. On the other hand, we strongly encourage researchers and tool builders to propose novel tools or improve existing ones.

2. LITERATURE REVIEW PROTOCOL

2.1 Goal and Research Questions

We defined our study goal based on the Goal-Question-Metric template [Basili and Rombach 1988] as follows: We aim to *analyze* the state of the art in DPD tools; *for the purpose of* summarizing the contributions made by previous studies; *with respect to* multiple aspects of the primary studies that either proposed or compared tools, as well as the main features of existing tools; *from the point of view of* researchers and practitioners interested in DPD; *in the context of* primary studies published over the past two decades. We introduce below our research questions (RQs).

RQ$_1$: *What is the publication landscape of DPD tools over the past two decades?*

RQ$_2$: *What DPD tools were published in the last two decades and what are their main features?*

RQ$_{2.1}$: *What are the existing DPD tools?*

[1] https://doi.org/10.5281/zenodo.5921504

RQ$_{2.2}$: *What are the main features of the existing DPD tools?*
RQ$_{2.3}$: *What detection approaches have been used by the existing DPD tools?*

2.2 Search for Primary Studies

We chose four Web engines to search for primary studies: ACM Digital Library (dl.acm.org), IEEE Xplore (ieeexplore.ieee.org), ScienceDirect (www.sciencedirect.com), and Springer (www.springer.com). All these engines provide a vast amount of studies published in Software Engineering [Dyba et al. 2007]. These engines have been used for conducting SLRs in various Software Engineering topics [Santos et al. 2018; Souza et al. 2018; Tavares et al. 2020; Cruz et al. 2019]. They have also been used in SLRs related to DPD, e.g., software pattern application [Riaz et al. 2015] and anti-pattern detection tools [Fernandes et al. 2016]. We run the following search string on each engine: *("design pattern detection") AND (tool OR "software solution") AND (automated OR automatic)*. This search string is a result of multiple pilot searches and followed by manual inspection of the obtained output.

We found the following numbers of primary studies by engine: 44 studies from ACM Digital Library, 18 from IEEE Xplore, 41 from ScienceDirect, and 76 from Springer. While running the search string on each engine, we set the interval from January 2000 to December 2021 as the range of publication year. The total number of primary studies, regardless of the existence of duplicates, equals 179. We automatically exported the metadata of all these studies to a Comma-Separated Values (CSV) file. For this purpose, we relied on either the native export feature provided by IEEE Xplore and Springer or the Data Miner[2]extension for the Google Chrome browser.

2.3 Filtering of Primary Studies

We relied on strict guidelines [Kitchenham and Charters 2007] for filtering the primary studies. First, we removed all duplicates found in the initial set of 179 primary studies. Second, we applied the following inclusion criteria: 1) the study is written in English, complete, and available online for download; 2) the study is either a conference/journal/symposium/ workshop paper; and 3) the study either proposes or compares DPD tools. With respect to the third criteria, some papers presented results of scripts or algorithms without an explicit mention of a tool implementation. Since the aim of this study is to evaluate ready-for-use tools, we discarded these studies and those that do not match either of the inclusion criteria. As a result, 25 valid primary studies remained for analysis.

We expanded our set of primary studies by performing backward snowballing [Wohlin 2014] on the 25 valid primary studies. First, we took notes of any DPD tools mentioned in the body of these studies and, then, collected the studies used as reference for these tools. Second, we applied the inclusion criteria mentioned above on the newly retrieved studies. We identified 17 additional studies, thereby resulting in a final set of 42 valid primary studies for analysis. It is important to highlight that this process was done in pairs.

2.4 Data Extraction from Primary Studies

The first two authors of this paper collaborated in the full-text read of all 42 selected primary studies, with the purpose of extracting and tabulating data. First, we extracted and tabulated the following metadata from each study: paper title, list of authors, release year, publication venue, and type of venue (conference, journal, etc.). Second, we extracted and tabulated the name of all DPD tools either mentioned in or proposed by each study. Third, we extracted and tabulated the following technical data of each DPD tool: tool name, tool type (plug-in or stand-alone), programming languages used to implement the tool, programming languages supported in DPD, availability to use for free, scope of the detectable design patterns, download availability, documentation availability, Graphical User Interface (GUI), detection techniques, and the way used to evaluate the DPD tools. Whenever technical data were not explicit in a study, we searched for data in the tool's website or the study's repository if

[2] `https://dataminer.io`

applicable. Regarding the tool availability for download, we restricted our search to the content of the paper and the available links.

3. LITERATURE REVIEW RESULTS

3.1 Publication Landscape (RQ_1)

Figure 1 presents the number of DPD tools published in the literature by year since 2000. The publication year refers to the year in which the primary study was published. We found out that new tools have been introduced every year with the exception of 2004 and 2013. This result suggests that introducing novel DPD tools is still a live research worldwide. Most studies appeared in conferences (55%), followed by journals (33%), symposiums (7%), and workshops (5%). Examples of prestigious conferences include International Conference on Software Engineering (ICSE) and International Conference on Automated Software Engineering (ASE). Examples of prestigious journals include IEEE Transactions on Software Engineering (TSE) and ACM Transactions on Software Engineering and Methodology (TOSEM).

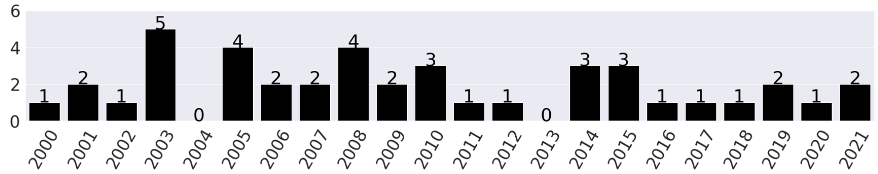


Fig. 1: Number of Tools Published by Year

3.2 Existing Tools ($RQ_{2.1}$)

Table I summarizes the 42 DPD tools revealed by our literature review. The tools listed above the thick line are available for download, while the ones listed below it are not. The first column lists the names of the tools and the respective reference, i.e., the primary study in which the tool was introduced. The second column informs whether the tool is either stand-alone or plug-in. Most tools (71%) are stand-alone, which suggests they are quite flexible to use since they do not depend on a particular Integrated Development Environment (IDE) to run. It is important to point out that some tools, such as DPRE [De Lucia et al. 2010b], are not clearly defined as plug-in or stand-alone tools in their respective papers. Since these tools were not available for download during the study execution, we assumed that they were stand-alone tools.

The third and fourth columns report on what programming languages were used for developing each tool and in which languages the tool can perform DPD. Java is a very popular language[3], so that our results are expected: most tools are written in Java (55%) and able to detect design patterns in Java (69%). Surprisingly, however, we could not find DPD tools for other popular programming languages, such as JavaScript and Python. The fifth column presents the scope of detectable design patterns. Section 3.3 discusses details on the data of this column. The sixth column reports on documentation availability for each tools. Only eight tools (19%) have some type of documentation (website, `README`, etc.) available online. As suggested in our tool comparison (Section 4.2), documentation may be a key to adopting a tool. Thus, we strongly encourage researchers to provide a minimum documentation to their tools. The seventh column informs whether the DPD tools provide any sort of Graphical User Interface (GUI). This knowledge is important for practitioners looking for easy-to-use interfaces, since GUIs aim at facilitating user interactions [Jansen 1998]. We detail the data summarized by this column in Section 3.3.

[3] TIOBE Index for October 2021: `https://www.tiobe.com/tiobe-index/`

Table I. : Overview of the Tools

Name	Type	Language		Scope of Detectable Design Patterns	Docs	GUI
		Devel.	Suppor.			
DesPaD [2016]	Stand -alone	Java	Java	Abstract Factory, Adapter, Bridge, +20	No	No
DPJF [2012]	Plug-in	Java	Java	CoR, Composite, Decorator, +2	Yes	Yes
FINDER [2015]	Stand -alone	Java	Java	Abstract Factory, Adapter, Bridge, +17	Yes	No
GEML [2021]	Stand -alone	Java	Java	Abstract Factory, Adapter, Builder, +12	Yes	Yes
MARPLE-DPD [2015]	Plug-in	Java	Java	Adapter, Composite, Decorator, +19	Yes	Yes
PatRoid [2019]	Stand -alone	Python	Java*	Abstract Factory, Adapter, Bridge, +20	Yes	No
PINOT [2006]	Stand -alone	C++	Java	Abstract Factory, Adapter, Bridge, +14	No	No
PTIDEJ [2007]	Stand -alone	Java	Java	Adapter, Builder, CoR, +14	Yes	Yes
Reclipse [2010]	Plug-in	Java	Java	CoR, Command, Composite, +2	Yes	Yes
SSA/DPD [2006]	Stand -alone	Java	Java	Adapter, Command, Composite, +9	No	Yes
Alnusair et al. [2009]	Stand-alone	N/A	Java	Composite, Observer, Singleton and 2 other	No	N/A
Antoniol et al. [2001]	Stand-alone	Java	C++	Adapter, Bridge, Composite and 2 other	No	Yes
APRT [2017]	Stand-alone	Java	Java	Singleton, Strategy	No	N/A
CrocoPat [2003]	Stand-alone	N/A	Java	Composite	No	N/A
D^3 (D-cubed) [2008]	Stand-alone	Java	Java	Abstract Factory, Builder, Factory Method, Singleton	No	No
DEPAIC++ [2002]	Stand-alone	Java	C++	Abstract Factory, Composite, Iterator	No	Yes
DP-Miner [2008]	Stand-alone	N/A	Java	Adapter, Composite, Decorator, State	No	Yes
DPDT [2021]	Stand-alone	C++	Java	Bridge, Composite, Flyweight and 4 other	No	N/A
DPF [2014]	Plug-in	Java	Java	Adapter, Command, Composite and 8 other	No	Yes
DPRE [2010b]	Stand-alone	Java	Java	Adapter, Bridge, Command and 8 other	No	Yes
DPVK [2005]	Stand-alone	N/A	Eiffel	Abstract Factory, Adapter, Bridge and 15 other	No	No
ePad [2018]	Plug-in	N/A	Java	Abstract Factory, Builder, Command and 9 other	No	Yes
Hedgehog [2001]	Stand-alone	N/A	Java	Bridge, Factory Method	No	N/A
Heuzeroth et al. [2003a]	Stand-alone	Java	Java	Chain of Responsibility, Mediator, Observer, Visitor	No	N/A
Heuzeroth et al. [2003b]	Stand-alone	N/A	Java	Composite, Decorator, Observer	No	N/A
JADEPT [2009]	Stand-alone	Java	Java	Chain of Responsibility, Observer, Visitor	No	Yes
Maisa [2000]	Stand-alone	Java	UML	Abstract Factory	No	N/A
nMarple [2010]	Plug-in	N/A	.NET	Abstract, Builder, Chain of Responsibility and 13 other	No	Yes
PatternFinder [2008]	Stand-alone	N/A	.NET	Abstract Factory, Adapter, Bridge and 20 other	No	Yes
Philippow et al. [2005]	N/A	N/A	C++	Abstract Factory, Adapter, Bridge and 20 other	No	N/A
PRAssistor [2005]	Stand-alone	Java	Java	Composite, Singleton, Visitor	No	No
Rasool & Mader [2011]	Stand-alone	C#	C++, Java	Abstract Factory, Adapter, Bridge and 20 other	No	Yes
Rasool & Mader [2014]	Plug-in	.NET, C#	C#, C++, Java	Abstract Factory, Adapter, Bridge and 20 other	No	Yes
Sartipo & Hu [2008]	Plug-in	Java	Java	Adapter, Bridge, Decorator and 4 other	No	Yes
SparT-ETA [2020]	Stand-alone	Java	Java	Abstract Factory, Adapter, Bridge and 19 other	No	N/A
SPQR [2003]	Stand-alone	N/A	Language Independent	Decorator	No	N/A
Thongrak et al. [2014]	Stand-alone	N/A	UML	Strategy	No	Yes
Van Doorn et al. [2019]	Stand-alone	Java	UML	Abstract Factory, Adapter, Bridge and 13 other	Yes	No
VokÃąc [2006]	Stand-alone	N/A	C++	Decorator, Factory Method, Observer and 2 other	No	N/A
VPML [2015]	Plug-in	Java	UML	Abstract Factory, Adapter, Bridge and 8 other	No	Yes
Web of Patterns [2007]	Plug-in	Java	Java	Abstract Factory, Adapter, Bridge and 7 other	No	Yes
Zhang & Li [2003]	N/A	N/A	C++	N/A	No	N/A

*Android apps written in Java.

3.3 Main Features of the Tools (RQ$_{2.2}$)

Supported Programming Languages: The TIOBE Index for October 2021 reports on Python, C, Java, C++, and C# as the five most popular languages worldwide. We found a considerable number of DPD tools compatible with at least Java (29; 69%), C++ (7; 17%), and C# (1; 2%). We also found tools compatible with UML (4; 10%), .Net platform (2; 5%), Eiffel (1; 2%), and Language Independent (1; 2%). This result suggests that practitioners could benefit from using existing tools. Curiously, four tools perform DPD on UML models [Thongrak and Vatanawood 2014; van Doorn et al. 2019; Elaasar et al. 2015; Paakki et al. 2000] rather than on source code. One tool is advertised as language independent [Smith and Stotts 2003]. This means that the tool runs on an abstract syntax tree, which can be derived from programs in different programming languages, such as C and Java.

Scope of Detectable Design Patterns: Table II shows the number of tools that detect each of the 23 design patterns cataloged by the Gang of Four (GoF) [Gamma et al. 1995]. The table is divided into three sets of three columns according to the pattern category: *behavioral patterns*, *structural patterns*, and *creational patterns*. Percentages are computed with respect to all 42 DPD tools and each tool may detect multiple design patterns. All 23 design patterns mentioned above are detectable by at least one tool. Nine out of 23 design patterns (35%) are detectable by at least 50% of the tools. We observed a balanced distribution of these nine design patterns across categories: four design patterns are Behavioral Patterns (i.e., Observer, State, Strategy, and Visitor); two are Creational (i.e., Factory Method and Singleton); and three are Structural (i.e., Adapter, Composite, and Decorator). We conclude that researchers focused on supporting the detection of not only easily detectable design patterns (e.g., Singleton because it involved very specific code elements), but also more complex ones, such as Composite and Visitor.

Table II. : Number of Tools that Detect Each Design Pattern

Behavioral Patterns			**Structural Patterns**			**Creational Patterns**		
Pattern	**Total**	**Percentage**	**Pattern**	**Total**	**Percentage**	**Pattern**	**Total**	**Percentage**
Observer	27	64%	Composite	28	67%	Singleton	22	52%
Strategy	24	57%	Decorator	25	60%	Factory Method	22	52%
State	21	50%	Adapter	21	50%	Abstract Factory	19	45%
Visitor	21	50%	Proxy	20	48%	Prototype	14	33%
Template Method	20	48%	Bridge	19	45%	Builder	13	31%
Command	18	43%	Flyweight	12	29%			
Chain of Responsibility	16	38%	Facade	8	19%			
Iterator	14	33%						
Mediator	13	31%						
Memento	12	29%						
Interpreter	11	26%						

Free-to-Use Availability: This information may be particularly important for small-sized teams with scarce resources available for developing software systems. We chose to report this information instead of the tool's license, since this information is usually not explicitly reported in papers. Out of the 42 DPD tools, we found that: 12 tools (29%) are explicitly reported by the authors as free to use; 1 tool (2%) is pay to use; and for the remaining 29 tools (69%) we could not find explicit reports on whether they are free or pay to use.

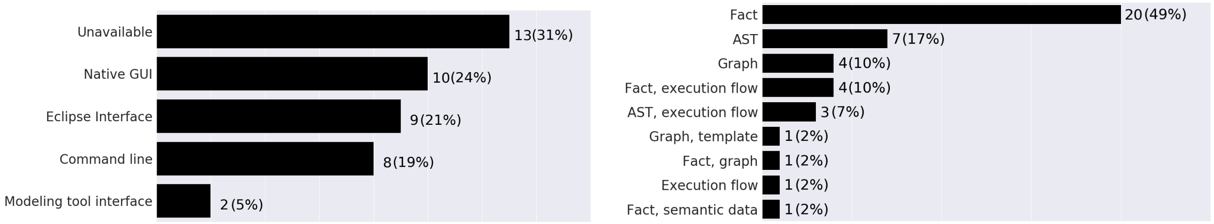
Download Availability: We assessed the download availability of DPD tools by searching for download links within the studies. We used these links for downloading the tools whenever they were available. Otherwise, we Google searched for the tool name. We found that ten out of the 42 DPD tools (24%) were downloadable during our study. This result is reasonable if we consider our wide range of publication year, i.e., two decades of research. Table I is organized in such a way the first ten tools are those available for download.

GUI Availability: Figure 2a presents the number of DPD tools by their Graphical User Interface (GUI). As we know, DPD may be challenging in practice (Section 1). Thus, the more a tool facilitates its use – e.g., through GUI – the better. Our results suggest that authors of the existing tools acknowledge the importance of GUI. Indeed, 10 tools (24%) provide their own native GUI; 9 tools (21%) are plug-ins for the Eclipse IDE[4] and, therefore, use the Eclipse GUI; 8 tools (19%) rely on command line; and 2 tools (5%) are embedded to software modeling tools and, thus, use the GUI provided by these modeling tools. We were unable to find out if there is a GUI provided by 13 out of the 42 tools (31%), due to poor documentation or unavailability online.

3.4 Detection Approaches ($RQ_{2.3}$)

Program Analysis Approaches: We found that 32 tools (76%) rely on static analysis for performing DPD. This result is quite expected for the following reasons. Many design patterns – especially creational and structural patterns – are characterized by patterns affecting the source code and its structure. Furthermore, static analysis uses both code and its structure as information sources (Section 2.1). In other words, it makes sense that researchers often rely on static analysis to perform DPD. In fact, only one tool [Arcelli et al. 2009](2%) relies on

[4] `https://www.eclipse.org/ide/`



(a) By GUI Type

(b) By Problem Modeling Approach

Fig. 2: Number of Tools by GUI Type and Problem Modeling Approach

dynamic analysis, which depends on tracking and understanding the execution of a software system (Section 2.1). This result is particularly curious because certain design patterns, especially behavioral ones, depend on those information sources to be characterized. Further exploring dynamic analysis could improve the accuracy of DPD tools for some patterns. Finally, nine tools (22%) rely on hybrid analysis: five tools were published in the first ten years analyzed, i.e., from 2000 to 2009; the other four tools appeared after 2010. We hypothesize that, over the decades, researchers realized that combining static and dynamic analyses could leverage the DPD accuracy. We also hypothesize that the inherent difficulty of performing dynamic analysis discussed by recent studies [De Lucia et al. 2018; Devecsery et al. 2018] may be preventing researchers to advance in proposing tools based on either dynamic or hybrid analysis.

Problem Modeling Approaches: Figure 2b presents the number of DPD tools according to the approach used for modeling the problem of detecting design patterns. We discussed that the majority of existing DPD tools (76%) rely exclusively on static analysis to detect instances of design patterns. Thus, it is reasonable to expect that most of the modeling approaches being used depend on analyzing the source code and its structure. In fact, most tools (49%) rely exclusively on data collected from the internal program structure, e.g., from classes, methods, and their relationships [Dabain et al. 2015; Rasool and Mäder 2011; van Doorn et al. 2019]. This is followed by 15% of the tools relying only on Abstract Syntax Tree (AST), whose nodes represent variables, operations, and other program elements [Shi and Olsson 2006; Robinson and Bates 2017; Heuzeroth et al. 2003a]. In addition, 10% of the tools rely only on graphs in general, whose nodes represent methods, classes, etc., and graph isomorphism is typically explored by the tools [Dobiš and Majtás 2008]. Another tool combines graph with template. Similar to graphs, templates are used for representing code elements in an alternative way (not necessarily as a graph) with the purpose of comparing a program with predefined templates for a given design pattern [Dong et al. 2008].

We also found tools relying on modeling approaches that are more suitable to either dynamic analysis or hybrid analysis. One of these approaches is execution flow through which the program execution is observed in terms of method calls, object instantiation, accessed or modified fields, and so forth [Arcelli et al. 2009; Huang et al. 2005]. The other approach is called semantic data, which computes information both statically and dynamically. For instance, static semantic constraints refer to how the classes interrelate, while dynamic semantic constraints refer to how particular methods operate [Blewitt et al. 2001]. Finally, only ten tools (17%) combine modeling approaches for supporting DPD, mostly to support hybrid analysis since only three of these ten combine two types of static modeling approaches.

4. TOOL COMPARISON PROTOCOL

4.1 Goal and Research Questions

We define the goal of the second part of our study as follows. This study aims to *analyze* the accuracy and applicability of existing DPD tools; *for the purpose of* comparing their strengths and weaknesses; *with respect to* precision, recall, F-measure, agreement, and certain tool features that regular users may find essential in practice;

from the point of view of researchers and practitioners interested in design patterns; *in the context of* published papers in the past 20 years and tools available for download. We introduce below our research questions (RQs).

RQ$_3$**:** *How accurate are the existing DPD tools?*

RQ$_4$**:** *To what extent the existing tools agree regarding the detection of design pattern instances?*

4.2 Selection of Tools

Although our literature review revealed 42 DPD tools (Section 3), 32 of them are not available for download. Regarding the remaining ten tools, we attempted, to our best effort, to install and use them with the help of the documentation available at their respective websites and the tool's configuration files (e.g., `README`). We excluded the tool PatRoid [Rimawi and Zein 2019] since it was designed to identify design patterns only in Android applications. This limitation would not allow us to compare its results with the other tools. Out of the nine remaining DPD tools, we managed to successfully install and use only four tools: FINDER [Dabain et al. 2015], GEML [Barbudo et al. 2021], MARPLE-DPD [Zanoni et al. 2015], and PTIDEJ [Moha and Guéhéneuc 2007]. As this was an already small enough subset of tools, we used the four tools for the comparative study. We discarded DesPaD [Oruc et al. 2016] due to configuration issue (stuck on second step, probably due to a issue with Subdue); DPJF [Binun and Kniesel 2012] due to use error when attempting to generate the factbase; PINOT [Shi and Olsson 2006] due to installation error when attempting to compile with the `make` command; Reclipse [Von Detten et al. 2010] due to installation issue (dependencies could not be installed, i.e., Palladio and SoMoX); and SSA [Tsantalis et al. 2006] because we were unable to use it in one of our chosen systems (JRefactory). We highlight that, in these cases, we lacked sufficient documentation to assist us in setting up and using the tools.

4.3 Selection of Design Patterns and Systems

While reading the papers from our systematic literature review, we listed the systems that were used to evaluate the proposed tools and systems with manually validated design pattern instances. We initially chose JUnit 3.7 and JHotDraw 5.1, as both systems are written in Java and have some validated instances of design patterns. However, due to Java version requirements, we were not able to use JUnit with all four DPD tools. Thus, we decided to use JRefactory 2.6.24 instead. These systems have been used in previous studies on DPD [Binun and Kniesel 2012; De Lucia et al. 2010b; Tsantalis et al. 2006].

To choose which patterns to evaluate in this study, we defined three criteria: the design pattern must have less than 100 instances collectively for both systems; at least two tools must return design pattern instances for either of the systems; no more than two design patterns per design pattern category (behavioral, creational and structural). Due to this, we first executed the tools to measure the number of instances they detect of each design pattern and then applied our inclusion criteria. The first and third criterion were destined to make a manual validation feasible. The manual validation of design pattern instances is not a trivial task, since it requires not only an understanding of the business logic of the system, but also knowledge about the design patterns themselves, such as the components, their roles and general structure. Although this subset of design patterns may not represent all capability of the tools detection prowess, each category comes with a different difficulty level for the detection process which may be an indicative of the tool's capability of coping with these difficulties.

The second criteria was created in order to make the calculation of the agreement coefficient feasible, since we need at least two raters in order to compute this metric. Creational design patterns help make a system more flexible by making the process of instantiating classes abstract [Gamma et al. 1995]. The chosen creational patterns were Prototype and Singleton. Behavioral design patterns distribute behavior between classes by describing the communication between them [Gamma et al. 1995], the behavioral patterns chosen were Chain of Responsibility and Visitor. Structural design patterns add flexibility to the system by composing objects into new ones with new functionality [Gamma et al. 1995], the structural patterns selected were Composite and Decorator.

4.4 Validation of Design Pattern Instances

Detection of Design Pattern Instances: We executed each selected tool on the chosen systems to obtain the list of pattern instances detected by tool. To do so, we create a virtual machine based on the Linux Mint 20 operating system with 8 GB of RAM, eight processors, and 3.59 GHz of clock speed. Table III presents the number of instances detected of each design pattern by each tools. Cells with a dash ("-") indicate that the tool could not detect the design pattern. For instance, GEML does not support Chain of Responsibility and Prototype detection, and due to an execution error it could not detect the Decorator pattern for JRefactory. The design patterns names in bold are the ones that were chosen for the comparative study. It is important to highlight that PTIDEJ output indicates the confidence level of each returned instance. We opted to discard the instances with a confidence value below 100% since we did not want to collect instances that the tool might classify as false positives.

Table III. : Number of Patterns Detected by Tool

Type	JHotDraw				JRefactory			
	FINDER	**GEML**	**MARPLE**	**PTIDEJ**	**FINDER**	**GEML**	**MARPLE**	**PTIDEJ**
Bridge	21	-	-	-	0	-	-	-
Builder	0	-	98	0	0	-	355	0
CoR	3	-	0	11	1	-	0	30
Command	25	-	25	0	0	-	39	0
Composite	4	3	5	0	2	2	2	0
Decorator	2	2	14	0	0	-	12	0
Facade	-	-	499	23	-	-	138	24
Factory Method	15	4	62	0	6	1	158	0
Flyweight	26	-	-	0	15	-	-	0
Mediator	0	-	0	0	0	-	0	0
Memento	2	-	0	0	24	-	0	0
Observer	13	12	61	0	8	1	90	0
Prototype	0	-	5	1	3	-	4	0
Proxy	0	0	0	51	0	13	0	104
Singleton	0	2	1	6	2	8	10	55
State	2	-	187	18	15	-	420	27
Strategy	7	-	95	-	24	-	197	-
Template Method	74	2	0	31	48	22	0	49
Visitor	0	6	9	1	4	4	19	1

Validation of Design Pattern Instances: Most design pattern instances consist of a combination of classes. For instance, a Composite design pattern instance contains one class playing the *Component* role, one or more classes playing the *Composite* role, and one or more classes playing the *Leaf* role. Roles can be of two types: anchor role, i.e., the Component role, and non-anchor role, i.e., the Leaf and Composite roles. Classes that play anchor roles differentiate one design pattern instance from another. Among the results obtained for a specific design pattern from a single DPD tool, there is no pair of instances with the same class playing the anchor role. On the other hand, there may be multiple instances with a class in common playing a non-anchor role.

Table IV presents the roles of each design pattern evaluated in this study. For illustration, let us consider the Composite design pattern. One of the true instances of the Composite pattern for the JHotDraw system is composed of: the class `CH.ifa.draw.framework.Figure` playing the *Component* role (anchor), the class `CH.ifa.draw.standard.CompositeFigure` playing the *Composite* role (non-anchor) and the class `CH.ifa.draw.figures.EllipseFigure` playing the *Leaf* role (non-anchor). Evaluating the results of the Finder tool, only one instance listed the `CH.ifa.draw.framework.Figure` class as the *Composite*, while two instances listed the `CH.ifa.draw.figures.EllipseFigure` class as the *Leaf*.

Table IV. : Patterns and Respective Roles

Pattern	**Anchor Roles**	**Non-anchor Roles**
Chain of Responsibility	Handler	Concrete Handler
Composite	Component	Composite, Leaf
Decorator	Decorator, Component	Concrete Component, Concrete Decorator
Prototype	Prototype	Concrete Prototype
Singleton	Singleton	N/A
Visitor	Element, Visitor	Concrete Element, Concrete Visitor

We validated the instances as true or false positive by manually inspecting all detected instances. For each design pattern, the first two authors discussed its definition and mechanisms in order to build a common ground on the pattern, its composing elements and roles. Thus, we were capable to understand what to look for while inspecting the instances – especially in terms of characteristics that turn an instance into a false positive instance. After that, those two authors inspected each instance of that particular design pattern individually. Each instance was discussed in pairs so we could reach a consensus on the instance classification. Both authors are familiar with the concept of design patterns and Java development. This validation process was conducted for all 234 instances detected by the four tools under comparison.

Computation of Equivalent Instances: When different tools detect a certain pattern, they may detect the same class for the anchor role but different classes for non-anchor roles. This poses a challenge for obtaining the true positive instances and the agreement between the tools. For example, when comparing two instances from different tools but with the same class playing the anchor role, it is very likely that they will not have the exact same classes playing the non-anchor roles. Thus, we defined two pattern instances as equivalent if they have the same class (or pair of classes for the Decorator and Visitor patterns) for the anchor roles and at least one common class for each of the non-anchor roles. With this definition, we aggregated the new true positive instances to our oracle database for each system.

Obtaining the Ground Truth: Although documentation is extremely important and helpful, it is often neglected [Aghajani et al. 2020]. In a similar manner, documentation of the design patterns implemented by a system is also not a common practice. Thus, obtaining the ground truth for a system is a very demanding task. To alleviate this issue, we created a proxy for the ground truth from the collective results of the P-MARt[5], which is a repository with previously identified and validated design pattern instances, and the instances that were manually validated in this study. A ground truth is required to quantify the recall metric for each tool. Our ground truth proxy is available in our replication package in the “Patterns Found” folder.

4.5 Computation of Accuracy Metrics

Regarding the accuracy of the tools, we chose three metrics to measure it: precision, recall and F-measure. The precision evaluates the correctness of the results returned by the tool [Baeza-Yates and Ribeiro-Neto 1999]. It is calculated from the ratio between the number of true positive instances detected and the total number of instances returned. The recall measures the ability to detect all known correct design pattern instances in a given system [Baeza-Yates and Ribeiro-Neto 1999]. It is obtained by the ratio of true positive instances detected and the total number of true positive instances in a system. In our case, we rely on both the P-Mart repository and our manual validation as a proxy of the ground truth. Lastly, we calculated the F-measure, which is the harmonic mean of the precision and recall, according to this equation: F = $\frac{2 \times precision \times recall}{precision + recall}$. This metric weights precision and recall equally, giving an overall score for each tool’s accuracy. We calculated all metrics for each pattern in each of the systems for each tool.

The agreement metric evaluated the redundancy of the results from the four different tools. A high agreement indicates that the tools detect the same instances, both true positives and false positives, whereas a low agreement means that the different pattern detection approaches obtain different instances. In order to calculate this metric, we listed all unique instances collectively detected by the tools and created an agreement table for each design pattern. After that, we used these tables as input for the calculation of Gwet’s AC1 coefficient [Gwet 2014], Overall Agreement and Confidence Interval over 95% using Gwet’s R package, i.e., `irrCAC`[6].

[5] `http://www.ptidej.net/tools/designpatterns/index_html#2`

[6] `https://cran.r-project.org/web/packages/irrCAC/index.html`

5. TOOL COMPARISON RESULTS

5.1 Precision, Recall, and F-measure (RQ$_3$)

Table V presents the Precision, Recall and F-measure of the tools for JHotDraw and JRefactory. Precision and recall cells with “N/A” mean the tool could not detect the pattern. F-measure cells with “N/A” mean there is neither a value of precision nor recall to calculate their harmonic mean. We used **bold font** in values greater than or equal to 50% to facilitate the discussions below. According to the results, FINDER managed to find instances of Composite, Decorator, Singleton, and Visitor. For these four design patterns, the tool presented precision and recall $\geq 50\%$ for at least one of the two systems under analysis i.e., JHotDraw and JRefactory – except in the case of Singleton. Overall, our results suggest that FINDER is applicable for detecting these design patterns. GEML successfully detected only Singleton instances but, compared to the other tools, it displayed the highest precision and recall for both systems. Although it did not detect a single instance of the other design patterns, practitioners may use it to detect Singleton.

Table V. : Accuracy Measures by Tool

System	Metric	Tool	CoR	Composite	Decorator	Prototype	Singleton	Visitor
JHotDraw	Precision	FINDER	0%	**75%**	**50%**	0%	0%	0%
		GEML	N/A	0%	0%	N/A	**100%**	0%
		MARPLE	0%	**60%**	14%	20%	**100%**	0%
		PTIDEJ	0%	0%	0%	0%	0%	0%
	Recall	FINDER	0%	**75%**	**50%**	0%	0%	0%
		GEML	N/A	0%	0%	N/A	**100%**	0%
		MARPLE	0%	**75%**	**100%**	**50%**	**50%**	0%
		PTIDEJ	0%	0%	0%	0%	0%	0%
	F-measure	FINDER	0%	**75%**	**50%**	0%	N/A	N/A
		GEML	N/A	N/A	N/A	N/A	**100%**	N/A
		MARPLE	N/A	**67%**	25%	29%	**67%**	N/A
		PTIDEJ	N/A	N/A	N/A	0%	N/A	N/A
JRefactory	Precision	FINDER	0%	**100%**	0%	0%	**100%**	**50%**
		GEML	N/A	0%	N/A	N/A	**88%**	0%
		MARPLE	0%	0%	0%	0%	**50%**	11%
		PTIDEJ	0%	0%	0%	0%	13%	**100%**
	Recall	FINDER	0%	**100%**	0%	0%	20%	**100%**
		GEML	N/A	0%	N/A	N/A	**70%**	0%
		MARPLE	0%	0%	0%	0%	**50%**	**100%**
		PTIDEJ	0%	0%	0%	0%	**70%**	**50%**
	F-measure	FINDER	N/A	**100%**	N/A	N/A	33%	**67%**
		GEML	N/A	N/A	N/A	N/A	**78%**	N/A
		MARPLE	N/A	N/A	N/A	N/A	**50%**	19%
		PTIDEJ	N/A	N/A	N/A	N/A	22%	**67%**
Agreement			33%	48%	49%	33%	49%	49%
AC1 coefficient			-0.20	0.12	0.18	-0.22	0.11	0.14
Confidence Interval ($\geq 95\%$)			[-0.200,-0.200]	[0.016,0.235]	[0.138,0.221]	[-0.316,-0.160]	[0.038,0.175]	[0.075,0.198]

Additionally, MARPLE-DPD was able of detecting instances of all design patterns but Chain of Responsibility on either system. Moreover, it was the only tool that managed to successfully detect instances of Prototype. However, the tool presented precision and recall $\geq 50\%$ simultaneously only in the cases of Singleton (for both systems) and Composite (for JHotDraw). Therefore, we encourage developers to use MARPLE-DPD for detecting instances of Singleton and Composite with an acceptable accuracy. Lastly, PTIDEJ managed to detect only Visitor and Singleton patterns, and failed to detect the others. Moreover, the tool detected instances with precision and recall $\geq 50\%$ exclusively for Visitor, which may encourage developers in using PTIDEJ for this purpose.

5.2 Agreement of the Tools (RQ$_4$)

The last three rows from Table V present the overall agreement of the tools, the Gwet’s AC1 agreement coefficient and the confidence interval. The overall agreement relates the number of times the tools agree with the total number of instances rated. This metric does not account for the random chance of agreement between raters unlike the AC1 [Gwet 2014], yet it displays a moderate agreement between the tools. The overall agreement for all six design patterns ranges from 33% to 49%. However, this metric does not take into account the fact that the tools may agree by chance. The AC1 coefficient is ≤ 0.18 for all the design patterns, indicating a poor agreement between the tools. This means that the output from the tools is not redundant, and different tools return different instances. Although one could argue that combining the results of the four tools would be recommended due to the increase of distinct instances detected, the results for precision, recall and F-measure indicate that most of the results are false positives.

6. STUDY IMPLICATIONS

6.1 Researchers and Tool Designers

Some simple issues prevent the DPD tools from having better precision. When conducting the manual validation of the design pattern instances returned by the tools, we noticed some issues regarding the general structure of the design patterns detected. These issues are relatively simple to describe, and by applying filters to remove them, for example, they could increase the precision of the tools. We describe some examples below. Regarding *Composite*, some instances claimed that a class was a Composite, which meant that they should contain an attribute that is a collection of Components. However, when inspecting such class, it did not contain any sort of collection that suffices the aforementioned requirements. Moreover, some instances listed Leaf and Composite classes that were supposed to inherit from the Component class, but after checking the inheritance hierarchy this requirement was not met. Regarding *Singleton*, we observed the lack of a verification of the visibility of the constructor. Some instances classified classes with a public construct as a Singleton, which goes against the concept of restricting the instantiation of a class to a single instance that the Singleton design pattern defines. With respect to *Visitor*, similarly to Composite, this design pattern suffers from inconsistent inheritance trees. In some of the instances, after checking the inheritance hierarchy of the concrete classes, we noted that they did not implement their corresponding interfaces, but extended completely different classes. In addition, some of the concrete classes were not actually concrete classes, but abstract ones, which ultimately invalidates the design pattern instance since it is supposed to be instantiated.

We need tools targeting different programming languages. As we discussed in Section 3.3, most of the tools target Java systems and only a few target certain design patterns such as Facade, Mediator, Prototype, etc. Java is one the most popular languages but not the only one that makes use of object oriented principles and design patterns [Ali 2016; Ayeva and Kasampalis 2018]. Furthermore, there are other emerging languages and environments which also require proposing, detecting and recommending design patterns.

6.2 Practical Use of the Tools

Certain tools are more suitable for specific design patterns. Different tools are able to detect the same design patterns, such as Composite discussed in Section 4.4. However, we observed that they achieve a low coefficient agreement. This low value indicates that they tend to detect different instances. Moreover, from the data presented in Section 5.1, some tools have a higher precision than others when detecting the same pattern. For example, GEML succeed when detecting the Singleton pattern. This information would help practitioners choose the tool best suited for their needs.

7. THREATS TO VALIDITY

7.1 Regarding the Literature Review

Construct and Internal Validity: We carefully planned our literature review protocol through multiple meetings and incremental refinements of the study artifacts. Thus, we expected to mitigate threats regarding the study execution. We carefully built our search string based of multiple synonyms to key terms. We expected to collect as many relevant studies as possible. Moreover, we performed snowballing to avoid overlooking important studies.

We relied on existing guidelines [Kitchenham and Charters 2007] to define our inclusion and exclusion criteria of primary studies, thereby filtering out irrelevant studies. Finally, two authors of this paper collaborated in both reading the primary studies for performing the data collection. They carefully discussed any divergences in order to reach a consensus on the data we found for each study, as well as to discard out-of-scope studies. Thus, we expected to mitigate human biases. With respect to the availability of the tools, it is possible that some tools may be available for download since we did not conduct any investigation with search engines. We assumed that the tool authors are responsible for providing links within their respective papers.

Conclusion and External Validity: We performed descriptive analysis in order to interpret the literature review data similarly to a previous work [Fernandes et al. 2016]. Thus, we expected to summarize our study results in an effective way for communication purposes. Finally, we opted for four Web search engines to support the collection of primary studies. We relied on well-known engines used in our previous work [Fernandes et al. 2016], some of them specific to the Computing domain such as ACM Digital Library. Thus, we expected to support the generality of our findings.

7.2 Regarding the Comparative Study

Construct and Internal Validity: We carefully installed and configured the tools available online for the comparative study (Section 4). We selected both design patterns and systems before executing the tool comparison, thereby mitigating biases that could favor the accuracy results of a particular tool. We have no conflicts of interest with the authors of the DPD tools under analysis. Still, we mitigate possible threats by counting on a second author to inspect their execution and help with problem solving. In addition, we attempted to conduct a pilot study with two volunteers for manually validating a number of DPD instances. Unfortunately, this task was very complex and time consuming, even for experienced developers. We ended up discarding the pilot study results and relying on the manual validation performed by two of the paper authors. By doing this as a pair, we expected to mitigate biases in the validation process.

Conclusion and External Validity: We computed precision, recall, and agreement similar to a previous work [Fernandes et al. 2016]. We opted for the Gwet's AC1 agreement coefficient due to its applicability to multiple raters and the fact that raters may agree by chance. Thus, we expected to avoid the misuse of statistics. We relied on our manually-validated instances of design patterns, plus the instances validated by previous studies, to compute recall. Thus, we expected to obtain as many true positive instances as possible by analyzed systems. Finally, we applied the inclusion criteria to filter out DPD tools for the comparison. One could argue that we compared only four out of the 42 existing tools and, therefore, our results may not represent the state-of-the-accuracy. Still, we highlight that our criteria were meant to assure that all tools are usable and fairly comparable. Similar reasoning applies to the analysis of six out of the 23 GoF design patterns [Gamma et al. 1995]. We selected pattern of different categories, i.e., Behavioral, Creational, and Structural, to support some variety in our findings. Lastly, likewise the previous threats, although we analysed only two systems both are functional systems that contain implementations of design patterns and are commonly used for benchmarking DPD tools in the literature.

8. RELATED WORK

Secondary studies [Riaz et al. 2015; Yarahmadi and Hasheminejad 2020] summarized the existing literature on design patterns. For instance, Riaz et al. [Riaz et al. 2015] conducted a systematic mapping study to characterize existing empirical studies involving software patterns and human participants. Their analysis was based on 30 primary empirical studies, being 24 original studies and 6 replications. They then classified the primary studies in terms of measures used for evaluation and considered threats to validity. Unlike this study, we do not restrict our analysis on empirical studies involving human participants. In fact, our paper focuses on automated approach to detect design patterns.

Yarahmadi and Hasheminejad [Yarahmadi and Hasheminejad 2020] present a broad systematic literature reviews on design patterns, including their detection methods. Their secondary study reviews research papers published between 2008 until 2019 and aims to answer 13 research questions. One of these question is related to what approaches exist to support DPD and which approaches are the most frequently used. Unlike Yarahmadi and Hasheminejad's work [Yarahmadi and Hasheminejad 2020], we focus on DPD tools. Moreover, we also present a comparative study of four detection tools with respect to their precision, recall, F-measure and agreement.

Regarding comparative studies, Dong et al. [Dong et al. 2009] present a comparative study of pattern mining techniques and tools about their pattern aspects checked, intermediate representations, exact or approximate matches, visualization, automated or human interactive support. It is also discussed the instances obtained and why different techniques and tools detect different instances of the same pattern for the same system. However, they did not conduct an experiment to install and run the tools in order to compare their output and calculate precision and recall metrics. Instead, they relied on the results reported by the authors.

Another comparative study was conducted by Rasool et al. [Rasool et al. 2011]. In this study, the authors selected a subset of 6 DPD tools with a major focus on evaluating their precision and recall. Although their subset of tools is larger than ours, there is only one DPD tool in common (PTIDEJ). They also present the overall precision and recall of each tool instead of individual values for each design pattern. In addition, we calculate the F-measure and agreement of the tools for each individual design pattern.

9. CONCLUSION AND FUTURE WORK

Detecting design patterns is an important yet challenging task. This paper presented a systematic literature review followed by a comparative study of DPD tools. We found 42 DPD tools but only ten are available online. Considering the 23 Gang of Four's patterns, Composite and Observer are the most commonly detected design patterns. By comparing four state-of-the-art tools, we found low accuracy results. Besides a considerable number of false positives, we figured out that the tools have a weak detection agreement. As future work, we plan to perform a user study aimed at investigating the developer's perception of the existing DPD tools.

ACKNOWLEDGMENTS

We thank our shepherds Alfredo Goldman and João Daniel and our writers' workshop colleagues for their valuable feedback. This research was partially supported by Brazilian funding agencies: CNPq (grant 132820/2020-0), CAPES, and FAPEMIG (grant BPD-00460-22).

REFERENCES

E. Aghajani, C. Nagy, M. Linares-Vásquez, L. Moreno, G. Bavota, M. Lanza, and D. C. Shepherd. 2020. Software documentation: The practitioners' perspective. In 42nd ICSE. 590–601.

J. Ali. 2016. Mastering PHP Design Patterns (1st ed.). Packt Publishing.

A. Alnusair and T. Zhao. 2009. Towards a Model-driven Approach for Reverse Engineering Design Patterns.. In 2nd TWOMDE, co-located with the 12nd MoDELS. 1–15.

A. Ampatzoglou, G. Frantzeskou, and I. Stamelos. 2012. A methodology to assess the impact of design patterns on software quality. Inf. Softw. Technol. 54, 4 (2012), 331–346.

G. Antoniol, G. Casazza, M. Di Penta, and R. Fiutem. 2001. Object-oriented design patterns recovery. J. Syst. Softw. 59, 2 (2001), 181–196.

F. Arcelli, D. Franzosi, and C. Raibulet. 2010. .NET reverse engineering with MARPLE. In 5th ICSEA. 227–231.

F. Arcelli, F. Perin, C. Raibulet, and S. Ravani. 2009. JADEPT: Dynamic analysis for behavioral design pattern detection. In 4th ENASE. 95–106.

K. Ayeva and S. Kasampalis. 2018. Mastering Python Design Patterns (2nd ed.). Packt Publishing.

R. Baeza-Yates and B. Ribeiro-Neto. 1999. Modern Information Retrieval (1st ed.). ACM Press.

R. Barbudo, A. Ramírez, F. Servant, and J. R. Romero. 2021. GEML: A grammar-based evolutionary machine learning approach for design-pattern detection. J. Syst. Softw. 175 (2021), 110919.

V. Basili and D. Rombach. 1988. The TAME project: Towards improvement-oriented software environments. IEEE Trans. Softw. Eng. 14, 6 (1988), 758–773.

M. L. Bernardi, M. Cimitile, and G. Di Lucca. 2014. Design pattern detection using a DSL-driven graph matching approach. J. Softw.: Evol. Process 26, 12 (2014), 1233–1266.

D. Beyer, A. Noack, and C. Lewerentz. 2003. Simple and efficient relational querying of software structures. In 10th WCRE. 216–225.

A. Binun and G. Kniesel. 2012. DPJF: Design pattern detection with high accuracy. In 16th CSMR. 245–254.

A. Blewitt, A. Bundy, and I. Stark. 2001. Automatic verification of Java design patterns. In 16th ASE. 324–327.

D. Cruz, E. Figueiredo, and J. Martinez. 2019. A literature review and comparison of three feature location techniques using ArgoUML-SPL. In 13th VaMos. 1–10.

H. Dabain, A. Manzer, and V. Tzerpos. 2015. Design pattern detection using FINDER. In 30th SAC. 1586–1593.

A. De Lucia, V. Deufemia, C. Gravino, and M. Risi. 2010a. An Eclipse plug-in for the detection of design pattern instances through static and dynamic analysis. In 26th ICSM. 1–6.

A. De Lucia, V. Deufemia, C. Gravino, and M. Risi. 2010b. Improving behavioral design pattern detection through model checking. In 14th CSMR. 176–185.

A. De Lucia, V. Deufemia, C. Gravino, and M. Risi. 2018. Detecting the behavior of design patterns through model checking and dynamic analysis. ACM Trans. Softw. Eng. Methodol. 26, 4 (2018), 1–41.

D. Devecsery, P. Chen, J. Flinn, and S. Narayanasamy. 2018. Optimistic hybrid analysis: Accelerating dynamic analysis through predicated static analysis. In 23rd ASPLOS. 348–362.

J. Dietrich and C. Elgar. 2007. Towards a web of patterns. J. Web Semant. 5, 2 (2007), 108–116.

M. Dobiš and L. Majtás. 2008. Mining design patterns from existing projects using static and run-time analysis. In 3rd CEE-SET. 62–75.

J. Dong, D. Lad, and Y. Zhao. 2007. DP-Miner: Design pattern discovery using matrix. In 14th ECBS. 371–380.

J. Dong, Y. Sun, and Y. Zhao. 2008. Design pattern detection by template matching. In 23rd SAC. 765–769.

J. Dong, Y. Zhao, and T. Peng. 2009. A review of design pattern mining techniques. Int. J. Softw. Eng. Knowl. Eng. 19, 6 (2009), 823–855.

T. Dyba, T. Dingsoyr, and G. Hanssen. 2007. Applying systematic reviews to diverse study types: An experience report. In 1st ESEM. 225–234.

M. Elaasar, L. Briand, and Y. Labiche. 2015. VPML: An approach to detect design patterns of MOF-based modeling languages. Softw. Syst. Model. 14, 2 (2015), 735–764.

F. A. Espinoza, G. Esquer, and J. Cansino. 2002. Automatic design patterns identification of C++ programs. In 1st EurAsia ICT. 816–823.

E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo. 2016. A review-based comparative study of bad smell detection tools. In 20th EASE. 18:1–18:12.

F. Fontana, A. Caracciolo, and M. Zanoni. 2012. DPB: A benchmark for design pattern detection tools. In 16th CSMR. 235–244.

L. Fulop, R. Ferenc, and T. Gyimóthy. 2008. Towards a benchmark for evaluating design pattern miner tools. In 12th CSMR. 143–152.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1995. Design Patterns (1st ed.). Addison-Wesley Professional.

K. Gwet. 2014. HandBOOK of Inter-Rater Reliability (4th ed.). Advanced Analytics.

D. Heuzeroth, T. Holl, G. Hogstrom, and W. Lowe. 2003a. Automatic design pattern detection. In 11th IWPC. 94–103.

D. Heuzeroth, S. Mandel, and W. Lowe. 2003b. Generating design pattern detectors from pattern specifications. In 18th ASE. 245–248.

H. Huang, S. Zhang, J. Cao, and Y. Duan. 2005. A practical pattern recovery approach based on both structural and behavioral analysis. J. Syst. Softw. 75, 1-2 (2005), 69–87.

B. Jansen. 1998. The graphical user interface. ACM SIGCHI Bulletin 30, 2 (1998), 22–26.

J. Kim, D. Batory, D. Dig, and M. Azanza. 2016. Improving refactoring speed by 10x. In 38th ICSE. 1145–1156.

B. Kitchenham and S. Charters. 2007. Guidelines for performing systematic literature reviews in software engineering. Technical Report. Version 2.3. EBSE Technical Report. EBSE-2007-01.

N. Moha and Y. G. Guéhéneuc. 2007. Ptidej and DECOR: Identification of design patterns and design defects. In 22nd OOPSLA. 868–869.

M. Oruc, F. Akal, and H. Sever. 2016. Detecting design patterns in object-oriented design models by using a graph mining approach. In 4th CONISOFT. 115–121.

J. Paakki, A. Karhinen, J. Gustafsson, L. Nenonen, and A. I. Verkamo. 2000. Software metrics by architectural pattern mining. In ICS, co-located with the 16th WCC. 325–332.

I. Philippow, D. Streitferdt, M. Riebisch, and S. Naumann. 2005. An approach for reverse engineering of design patterns. Softw. Syst. Model. 4, 1 (2005), 55–70.

G. Rasool and P. Mäder. 2011. Flexible design pattern detection based on feature types. In 26th ASE. 243–252.

G. Rasool and P. Mäder. 2014. A customizable approach to design patterns recognition based on feature types. Arab. J. Sci. Eng. 39, 12 (2014), 8851–8873.

G. Rasool, P. Maeder, and I. Philippow. 2011. Evaluation of design pattern recovery tools. Procedia Comput. Sci. 3 (2011), 813–819.

M. Riaz, T. Breaux, and L. Williams. 2015. How have we evaluated software pattern application? A systematic mapping study of research design practices. Inf. Softw. Technol. 65 (2015), 14–38.

D. Rimawi and S. Zein. 2019. A model based approach for android design patterns detection. In 3rd ISMSIT. 1–10.

A. Robinson and C. Bates. 2017. APRT: Another Pattern Recognition Tool. GSTF J. Comput. 5, 2 (2017), 46–52.

A. L. dos Santos, M. R. A. Souza, M. Dayrell, and E. Figueiredo. 2018. A systematic mapping study on game elements and serious games for learning programming. In 10th CSEDU. 328–356.

K. Sartipi and L. Hu. 2008. Behavior-driven design pattern recovery. In 12th SEA. 179–185.

N. Shi and R. Olsson. 2006. Reverse engineering of design patterns from Java source code. In 21st ASE. 123–134.

J. Singh, S. Roy Chowdhuri, G. Bethany, and M. Gupta. 2021. Detecting design patterns: a hybrid approach based on graph matching and static analysis. Inf. Technol. Manag. (2021), 1–12.

J. Smith and D. Stotts. 2003. SPQR: Flexible automated design pattern extraction from source code. In 18th ASE. 215–224.

M. R. A. Souza, L. Veado, R. T. Moreira, E. Figueiredo, and H. Costa. 2018. A systematic mapping study on game-related methods for software engineering education. Inf. Softw. Technol. 95 (2018), 201–218.

K. Stencel and P. Wegrzynowicz. 2008. Detection of diverse design pattern variants. In 15th APSEC. 25–32.

C. S. Tavares, A. Santana, E. Figueiredo, and M. A. S. Bigonha. 2020. Revisiting the Bad Smell and Refactoring Relationship: A Systematic Literature Review.. In CIbSE. 434–447.

M. Thongrak and W. Vatanawood. 2014. Detection of design pattern in class diagram using ontology. In 18th ICSEC. 97–102.

N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis. 2006. Design pattern detection using similarity scoring. IEEE Trans. Softw. Eng. 32, 11 (2006), 896–909.

E. van Doorn, S. Stuurman, and M. van Eekelen. 2019. Static detection of design patterns in class diagrams. In 8th CSERC. 79–88.

M. Vokác. 2006. An efficient tool for recovering design patterns from C++ code. J. Object Technol. 5, 1 (2006), 139–157.

M. Von Detten, M. Meyer, and D. Travkin. 2010. Reverse engineering with the Reclipse tool suite. In 32nd ICSE. 299–300.

W. Wang and V. Tzerpos. 2005. Design pattern detection in Eiffel systems. In 12th WCRE. 1–10.

C. Wohlin. 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In 18th EASE. 1–10.

R. Xiong, D. Lo, and B. Li. 2020. Distinguishing similar design pattern instances through temporal behavior analysis. In 27th SANER. 296–307.

H. Yarahmadi and S. Hasheminejad. 2020. Design pattern detection approaches: A systematic review of the literature. Artif. Intell. Rev. 53, 8 (2020), 5789–5846.

S. Yau and J. Tsai. 1986. A survey of software design techniques. IEEE Trans. Softw. Eng. SE-12, 6 (1986), 713–721.

M. Zanoni, F. Fontana, and F. Stella. 2015. On applying machine learning techniques for design pattern detection. J. Syst. Softw. 103 (2015), 102–117.

Z. Zhang and Q. Li. 2003. Automated detection of design patterns. In 2nd GCC. 694–697.

Received July 2022; revised September 2022; accepted January 2023

PLoP'22, October 17–24, Virtual Online. Copyright 2022 is held by the author(s). HILLSIDE 978-1-941652-18-3