UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Rodrigo André Ferreira Moreira

Design Pattern Detection Tools: Review-based Comparison and Survey Studies

Belo Horizonte
2023

Rodrigo André Ferreira Moreira

Design Pattern Detection Tools:
Review-based Comparison and Survey Studies

Final Version

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Eduardo Magno Lages Figueiredo
Co-Advisor: Eduardo Moreira Fernandes

Belo Horizonte
2023

© 2023, Rodrigo André Ferreira Moreira.
Todos os direitos reservados

Moreira, Rodrigo André Ferreira

M838d Design pattern detection tools: [recurso eletrônico]: review-based comparison and survey studies / Rodrigo André Ferreira Moreira— 2023.
1 recurso online (85f. il, color.)

Orientador: Eduardo Magno Lages Figueiredo.
Coorientador: Eduardo Moreira Fernandes.
Dissertação (mestrado) - Universidade Federal de Minas Gerais, Departamento de Ciência da Computação, Instituto de Ciências Exatas.
Referências: f. 77-85.

1. Computação – Teses. 2. Software - Desenvolvimento – Teses. 3. Software – Ferramentas de manutenção - Teses. 4. Administração de projetos - Revisão de literatura – Teses. I. Figueiredo, Eduardo Magno Lages II. Fernandes, Eduardo Moreira. III Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Computação. IV. Título.

CDU 519.6*32(043)

Ficha Ficha catalográfica elaborada pela bibliotecária Belkiz Inez Rezende Costa CRB 6/1510 Universidade Federal de Minas Gerais - ICEx

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Design Pattern Detection Tools: Review-based Comparasion and Survey Studies

RODRIGO ANDRÉ FERREIRA MOREIRA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores :

Prof. Eduardo Magno Lages Figueiredo - Orientador
Departamento de Ciência da Computação - UFMG

Dr. Eduardo Moreira Fernandes - Coorientador
Queen's University

Prof. Marco Túlio de Oliveira Valente
Departamento de Ciência da Computação - UFMG

Profa. Simone Diniz Junqueira Barbosa
Departamento de Informática - PUCRJ

Belo Horizonte, 24 de março de 2023.

Resumo

Contexto: Padrões de projeto são soluções para problemas recorrentes de design de software. Ferramentas de Detecção de Padrões de Projeto (DPP) existentes suportam padrões de projeto e linguagens de programação específicas. Diversas ferramentas foram propostas para automatizar o processo de DPP em sistemas. No entanto, escolher a ferramenta mais adequada pode ser difícil. *Objetivo*: Conduzimos um estudo sistemático para entender quais são as ferramentas existentes e suas características, como é o desempenho de DPP dessas ferramentas, por que as ferramentas foram propostas para contextos específicos e como potenciais usuários as percebem. *Método*: Conduzimos uma revisão sistemática da literatura sobre as ferramentas de DPP desenvolvidas nos últimos 20 anos e suas características. Em seguida, comparamos o desempenho de quatro ferramentas na detecção de seis padrões em relação a precisão, cobertura, F-measure e concordância. Por último, conduzimos duas enquetes: a primeira com objetivo de capturar os motivos para a escolha dos dos desenvolvedores por contextos específicos a serem atendidos pelas ferramentas; a segunda visando descrever o quão úteis são essas ferramentas na visão de potenciais usuários. *Resultados*: Identificamos 42 ferramentas de DPP e listamos suas características, tais como os padrões de projeto detectáveis e as linguagens de programação suportadas. Os resultados do estudo comparativo sugerem que algumas ferramentas são mais adequadas para detectar certos padrões com precisão e cobertura satisfatórios. Também observamos uma baixa concordância entre os resultados de detecção obtidos por ferramentas distintas. Em relação às enquetes, percebemos que os desenvolvedores de ferramentas geralmente escolhem apoiar a detecção de padrões específicos devido às estruturas internas, enquanto que linguagens de programação são geralmente escolhidas pela sua popularidade na indústria de software. Exemplos de benefícios esperados do uso de ferramentas de DPP incluem a compreensão de sistemas de software e o apoio à realização de tarefas como limpeza de código-fonte. No entanto, ferramentas que são difíceis de utilizar tendem a ser descartadas por potenciais usuários.

Palavras-chave: Padrões de projeto. Ferramenta automática de desenvolvimento. Revisão sistemática da literatura. Estudo comparativo quantitativo. Enquete. Análise qualitativa.

Abstract

Context: Design patterns are solutions for recurring problems of software design. Existing Design Pattern Detection (DPD) tools target specific design patterns and programming languages. Several tools have been proposed for automating the DPD process. However, choosing the most suited tool may be troublesome. *Objective*: We conducted a systematic study to find out which are the existing tools and their respective features, how they perform when detecting design patterns, why these tools were developed to address their specific contexts, and how potential users perceive them. *Method*: We first conducted a systematic literature review about DPD tools developed in the last 20 years and their main features. We then compared the performance of four tools in detecting six design patterns based on precision, recall, F-measure, and agreement. Lastly, we carried out two survey studies: the first aimed at capturing reasons why DPD tool designers target specific contexts, and the second survey for capturing the usefulness of DPD tools from the perspective of potential users. *Results*: We found 42 DPD tools and listed their features, such as availability, detectable design patterns, and supported programming languages. The comparison results suggest that some tools are more suitable for detecting specific design patterns with satisfactory precision and recall. We also observed a low agreement among the detection results obtained by different tools. Regarding the surveys, we perceived that tool designers often support the detection of specific design patterns due to their internal structure, while programming languages are often chosen by their popularity in the software industry. Expected benefits of using DPD tools include program comprehension and support to the conduction of quality improvement tasks, such as source code cleaning. However, tools that are difficult to use tend to be discarded by potential users.

Keywords: Design patterns. Automated development tool. Systematic literature review. Quantitative comparative study. Survey study. Qualitative analysis.

List of Figures

3.1 Number of DPD tools published by year . . . 29
3.2 Number of tools by GUI type . . . 34
3.3 Number of tools by problem modeling approach . . . 34

5.1 Most used programming language . . . 54
5.2 Years of experience as a programmer . . . 54
5.3 Current main role . . . 54
5.4 Familiarity with design patterns . . . 55
5.5 Data analysis procedures for closed survey questions . . . 56
5.6 Taxonomy of reasons to target specific design patterns . . . 57
5.7 Taxonomy of reasons to support the detection of more design patterns . . . 59
5.8 Taxonomy of reasons to target specific programming languages . . . 59
5.9 Taxonomy of reasons to support the detection in more languages . . . 61
5.10 Taxonomy of contexts in which tools have been used . . . 61
5.11 Contexts in which developers would consider using a DPD tool . . . 63
5.12 Expected benefits of using a DPD tool . . . 64
5.13 Taxonomy of barriers to the use of DPD tools . . . 66
5.14 Taxonomy of reasons why detecting specific patterns is worthwhile . . . 67

List of Tables

3.1 Web search engines used to search for primary studies 27
3.2 Overview of the tools available for download 30
3.3 Overview of the tools unavailable for download 31
3.4 Number of tools that detect each design pattern 33
3.5 Detection approaches of the tools 35

4.1 Number of patterns detected by tool 43
4.2 Patterns and their respective roles 43
4.3 Accuracy measures by tool 46

5.1 Structure of the survey with tool designers 51
5.2 Structure of the survey with potential tool users 52
5.3 Demographic information of each participant 55
5.4 Details of reasons to target specific design patterns 58
5.5 Details of reasons to target specific programming languages 60
5.6 Details of contexts in which tools have been used 62
5.7 Details of the expected benefits of using a DPD tool 65
5.8 Details of barriers to the use of DPD tools 66
5.9 Details of reasons why detecting specific patterns is worthwhile 68

Contents

1 Introduction **10**

1.1 Problem Statement . . . 10
1.2 Research Method . . . 12
1.3 Results and Contributions . . . 14
1.4 Publications . . . 15
1.5 Dissertation Outline . . . 16

2 Background and Related Work **17**

2.1 Design Patterns . . . 17
2.2 Design Pattern Detection Tools . . . 20
2.3 Accuracy Metrics and Agreement . . . 20
2.4 Existing Literature Reviews on DPD Tools . . . 22
2.5 Closely Related Survey Studies . . . 23
2.6 Chapter Summary . . . 24

3 Literature Review on DPD Tools **25**

3.1 Study Protocol . . . 25
3.1.1 Goal and Research Questions . . . 26
3.1.2 Search for Primary Studies . . . 27
3.1.3 Filtering of Primary Studies . . . 27
3.1.4 Data Extraction from Primary Studies . . . 28
3.2 Results and Discussions . . . 29
3.2.1 Publication Landscape (RQ_1) . . . 29
3.2.2 Existing DPD Tools ($RQ_{2.1}$) . . . 30
3.2.3 Main Features of DPD Tools ($RQ_{2.2}$) . . . 32
3.2.4 Detection Approaches behind the DPD Tools ($RQ_{2.3}$) . . . 35
3.3 Threats to Validity . . . 37
3.4 Chapter Summary . . . 37

4 Comparative Study of DPD Tools **39**

4.1 Study Protocol . . . 40
4.1.1 Goal and Research Questions . . . 40
4.1.2 Selection of Tools . . . 40
4.1.3 Selection of Design Patterns and Software Systems . . . 41

4.1.4 Validation of Design Pattern Instances 42
4.1.5 Computation of Accuracy Metrics 44
4.2 Tool Comparison Results 45
4.3 Threats to Validity 47
4.4 Chapter Summary 48

5 Survey Study on DPD Tools 49
5.1 Study Design 49
5.1.1 Goal and Research Questions 50
5.1.2 Survey Structure 51
5.1.3 Participant Characterization 52
5.1.4 Data Analysis Procedures 55
5.2 Results of the Survey with Tool Designers 57
5.2.1 Reasons to Detect Specific Design Patterns 57
5.2.2 Reasons to Support Specific Programming Languages 59
5.2.3 Reported Contexts of Tool Usage 61
5.3 Results of the Survey with Tool Users 62
5.3.1 Potential Tool Use Contexts 62
5.3.2 Expected Benefits of Using a DPD Tool 63
5.3.3 Barriers to the Use of DPD Tools 64
5.3.4 Design Patterns Worth Detecting 67
5.4 Study Implications 68
5.5 Threats to Validity 70
5.6 Chapter Summary 71

6 Conclusion 73
6.1 Work Overview 73
6.2 Main Contributions 75
6.3 Study Implications 76
6.4 Future Work 77

Bibliography 78

Chapter 1

Introduction

Software design is the process of translating requirements into a detailed design representation of a software system [85]. This process includes making decisions on how to organize the source code [85]. Since making these decisions is not a trivial task, design patterns have been proposed for driving them [37]. Design pattern is a reusable solution for a recurring problem of software design [37]. Examples of design patterns are Composite, Singleton, and Visitor. The Composite design pattern allows treating a group of objects and individual objects in the same manner, by composing objects into a tree structure [37]. Singleton controls the instantiation of an object, ensuring that there is only one instance and providing a global point of access to it [37]. Visitor helps to dynamically introduce new operations to an object without changing its structure [37].

Analyzing software and performing changes may require a deep understanding of which and how design patterns occur in the source code [79]. Unfortunately, the identification of design patterns implementations may be troublesome due to using combination of patterns or their poor identification [79]. The literature also suggests that detecting design patterns is challenging due to the size and complexity of real-world systems [25, 78]. Thus, software engineers need automated assistance for making DPD easier, faster, and more accurate. In this context, design pattern detection (DPD) may facilitate the software analysis [45] by providing the automated detection of design pattern instances, thereby assisting design decision making [4, 86].

1.1 Problem Statement

Several studies have proposed tools for automating DPD [19, 26, 34, 64, 78]. Each tool presents different features, such as different detecting strategies (the approach implemented by the tool to perform the DPD), different detectable design patterns, different supported languages, etc. For instance, the PINOT tool [64] performs a static analysis on the source code and uses Abstract Syntax Trees (AST) as the problem modeling strategy.

PINOT is a command line based tool that detects Singleton, Facade, Chain of Responsibility, and 14 other design patterns from systems written in Java. DPVK tool [78] also performs static analysis on the source code to detect the design patterns, but it uses facts as the problem modeling strategy. It is also a command line based tool and detects 18 different design patterns. Unlike the PINOT tool, it does not detect Singleton, Facade, and Chain of Responsibility.

As the DPD tools provide different functionalities, it is important to know them beforehand in order to choose the tool best suited for a specific context. For example, researchers that are developing a new DPD tool might be interested in comparing their tool with existing ones in order to evaluate their tool's precision. Thus, they would be interested in knowing which existing tools detect the same design patterns or support the same programming language as their tool. Furthermore, practitioners who are looking for a DPD tool to assist them with system comprehension will need to find a suitable tool for the system they are working on. They need a tool that supports the programming language of their system, for instance, and they may prefer a tool that provides a Graphical User Interface (GUI).

There have been past attempts to summarize these tools [60, 84]. Unfortunately, they contain some gaps in their scope. They are more focused on the technical details of the implementation of each approach instead of the tool itself and its attributes (type of GUI, supported programming languages, etc). These attributes may be pivotal in order to choose a DPD tool to be used in a specific context.

Research Problem 1: Lack of catalog of available tools containing their attributes.

There have been studies comparing the output of DPD tools in terms of precision and recall [18, 57], but none regarding the agreement of their output. Generally, these comparative studies are presented as a form of benchmarking the proposed tool against tools available in the literature. However, these studies contain some design problems, such as an arbitrary selection of tools for comparison, lack of manually validated instances, etc. Furthermore, most studies rely on the reported results of the selected tools instead of installing, executing and validating the results.

This lack of reproduction and validation of reported results can be considered a threat to the validity of these studies. Similarly to the tool's attributes, the tool's performance is also an important factor when choosing a DPD tool. Once one selects which tools support ones technical needs, one would most likely pick the best-performing tool when attempting to detect a specific subset of design patterns. Not only is this information important for practitioners, but researchers may also find it helpful in order to evaluate which kind of approaches provide better results when detecting certain design patterns and possibly produce new detection approaches.

Research Problem 1: Lack of systematic comparison of the tools in terms of precision, recall and agreement.

The DPD tools available in the literature [48, 74, 10] generally target very specific design patterns (such as Observer, Singleton and Visitor) and programming languages (Java). Unfortunately, we do not know why DPD tool designers have targeted specific design patterns and programming languages. We believe that this information can shed light on the types of technical support that tool designers need for targeting other patterns and languages of relevance in the industry.

Research Problem 3: Lack of evidence about the design rationale behind the proposal of the tools.

Furthermore, to the best of our knowledge, there are no studies that provide evidence of the expected usefulness of DPD tools from the perspective of potential tool users. Although there are many DPD tools available in the literature which claim to successfully detect design pattern instances from source code, are they actually useful and aligned with the software industry's needs? Due to the lack of empirical evidence from qualitative studies, it is not possible to confirm it. It could be that the DPD tools fall short of meeting users' expectations. Or maybe the perceived barriers to using DPD tools outweigh the perceived benefits. We believe that acquiring such understanding is essential to propose novel DPD tools (or refine existing tools) that are able to meet the current needs of potential users.

Research Problem 4: Lack of evidence regarding the expected usefulness from the point of view of practitioners.

1.2 Research Method

We proposed and executed three studies that complement each other in order to address the problems described in the previous section.

Study 1: This study fills the literature gaps discussed in *Research Problem 1* through a literature review 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 systematic literature reviews [46]. We relied on Web search engines that provide a vast amount of studies published in Software Engineering [29] such as ACM Digital Library[1], and Springer[2]. We then analyzed the data to understand both the nature and scope of the existing tools. Regarding the steps followed, we first listed the results obtained from the Web search engines. Then, we filtered them according to a set of inclusion and exclusion criteria by reading the metadata. After that, we conducted a full read of the papers and extracted information about the tools. For instance, we identified the design patterns and programming languages supported by the tools. Lastly, we expanded our results by conducting a backward snowballing on the selected studies.

Study 2: This study provides the information to address the *Research Problem 2* by comparing the performance of four state-of-the-art tools – FINDER [18], GEML [10], MARPLE-DPD [86], and PTIDEJ [48] – in detecting six design patterns: Chain of Responsibility, Composite, Decorator, Prototype, Singleton, and Visitor. We computed performance in terms of precision, recall, F-measure, and agreement [38] measures. We used a virtual environment to install and execute all four tools. We then extracted the instances of each design pattern detected by the tools and conducted a manual validation in pairs in order to classify them as true or false positives. After that, we computed the precision, recall, F-measure, and agreement metrics. It is important to highlight that we made use of the R package irrCAC [3] to obtain the agreement metrics.

Study 3: This study addresses the aforementioned knowledge gaps discussed in *Research Problem 3* and *Research Problem 4* by conducting two online surveys aimed at capturing the perceptions of tool designers and tool users regarding DPD tools, relying on strict guidelines [56]. For the first survey, we emailed all authors of the 42 DPD tools found in *Study 1*. We asked authors questions regarding the reasons for targeting specific design patterns and programming languages with their tools. Designers of nine out of the 42 DPD tools responded to our survey.

For the second survey, we recruited 21 junior or senior developers who are potential users of DPD tools. We ask them about the expected usefulness of DPD tools. We relied on Grounded Truth procedures [69] to manually extract topics from the open questions answered by the survey participants. Additionally, we followed strict guidelines [17] to generate taxonomies from the topics aimed at summarizing the empirical knowledge on reasons to target specific contexts in the DPD tool design as well as the expected usefulness of these tools.

[1] https://dl.acm.org
[2] https://www.springer.com
[3] https://cran.r-project.org/web/packages/irrCAC/vignettes/overview.html

1.3 Results and Contributions

We created a catalog of the DPD tools published in the last 20 years presenting their main features such as availability for download, scope of design pattern detection, supported programming languages, etc. This catalog will help both researchers and practitioners when choosing the DPD tool that best suits their context. Our SLR revealed 42 DPD tools published in the last 20 years. However, only 10 of them were available for download during our investigation. Altogether, the tools claim to detect all 23 design patterns compiled by the Gang of Four's book [37]. The frequency of published tools remained stable over the years, thereby suggesting this research topic has not yet reached saturation. From the 42 DPD tools found, 68% are stand-alone. In addition, 69% of the tools are compatible with Java systems. However, only 19% of the tools have some companion documentation. 69% are designed to perform DPD on Java systems, support the detection of complex design patterns such as Composite (67%) and Observer (64%), and provide users with some sort of GUI other than command line (50%).

Our results from the comparative study suggest that each tool could be applied to detecting a particular subset of design patterns. FINDER showed sufficient accuracy for Composite, Decorator, Singleton, and Visitor. Still, future work is required to further support practitioners interested in automating DPD in practice. For the six design patterns evaluated, the four DPD tools tend to disagree on the instances detected, meaning that their output is not redundant. In other words, they detect distinct design pattern instances. Documentation on how to install, execute, and use the tools is often scarce, which may compromise the adoption of tools in practical settings.

Among our results from the third study, we found five different reasons for targeting specific design patterns in the proposal of DPD tools. Most tool designers reported that internal aspects of a design pattern (e.g., the way it is structured in the source code) favored its detection. Additionally, popularity and abstraction level are key reasons to decide for detecting design patterns in specific programming languages. Tool designers showed interest in proposing tools for a wider scope of design patterns and programming languages. Tool designers have knowledge on their DPD tools being used for program analysis tasks, especially program comprehension. This finding meets the expected benefits of using DPD tools reported by the potential tool users. According to the tool users, other benefits include software quality improvement (e.g., via code organization and cleaning) and more. Potential tool users claim that tool limitations (e.g., low accuracy and lack of documentation), as well as the inherent difficulty to use the tool, are key barriers to the DPD tool adoption.

Our main contributions from this work include:

1. an extensive catalog of 42 DPD tools published in the last two decades, each characterized by the supported design problems, programming languages, etc.;

2. a quantitative comparison of four state-of-the-art tools based on widely used measures, i.e. precision, recall, F-measure, and agreement;

3. a list with manually validated instances of six design patterns detected from two Java software systems often investigated in studies on DPD tools [26, 21, 57], i.e. JHotDraw and JRefactory; and

4. replication packages for studies 1 and 2 [4], and study 3 [5];

5. empirical evidence about the design rationale behind the proposal of the tools;

6. empirical evidence regarding the expected usefulness from the point of view of practitioners;

1.4 Publications

This section lists the scientific publications and submissions resulting from both this work and other topics.

- Rodrigo Moreira, Wesley K. G. Assunção, Jabier Martinez and Eduardo Figueiredo. Open-source software product line extraction processes: the ArgoUML-SPL and Phaser cases. In *Empirical Software Engineering (EMSE)*, 2022.

- Rodrigo Moreira, Eduardo Fernandes and Eduardo Figueiredo. Review-based Comparison of Design Pattern Detection Tools. In *Proceedings of Conference on Pattern Language of Programs (PLoP)*, 2022.

- Rodrigo Moreira, Eduardo Fernandes and Eduardo Figueiredo. Why are design pattern detection tools created and how do users perceive them. Submitted to *Software Quality Journal (SQJ)*.

[4] https://doi.org/10.5281/zenodo.5553470
[5] https://doi.org/10.5281/zenodo.7465098

1.5 Dissertation Outline

The remainder of this work is structured as follows.

Chapter 2 introduces important concepts that are used throughout this work, such as design patterns, design pattern detection tools, and accuracy metrics. We also discuss some related work regarding relevant the topics.

Chapter 3 introduces the protocol adopted for performing our systematic literature review, describing the steps followed, the digital databases and search string used, and so forth. It also presents the respective findings obtained from our investigation of the literature regarding the DPD tools found and their respective threats to validity.

Chapter 4 describes the protocol of our tool comparison study, for instance, the systems and design patterns evaluated, and the Design Pattern Detection tools used. The metrics obtained from our study are reported along with a discussion of the results and respective threats to validity.

Chapter 5 presents the design of our survey study: their structures, participants, and how we analyzed the quantitative and qualitative data. We present the taxonomies obtained from the responses obtained followed by a discussion and respective threats to validity.

Chapter 6 concludes this work by summarizing the contributions and results, and also suggesting future work.

Chapter 2

Background and Related Work

Knowing the design patterns instances in a system is useful for developers implementing new features or performing maintenance tasks on a system they are not very familiar with. As this is not a trivial task to be executed manually and becomes increasingly more complex for larger systems, automated assistance is required. Design pattern detection tools were created to address this issue.

In this chapter, we present some important concepts to understand the studies that will be presented in this work. The chapter is structured as follows: Section 2.1 presents the definition of software design patterns, and also describes the six design patterns that will be used in the comparative study; Section 2.2 introduces design pattern detection tools, the most common types of analysis and modeling strategies; Section 2.3 presents the metrics that will be used to compare the performance of the analyzed tools; Section 2.4 presents related works about DPD tools, while Section 2.5 presents related works about survey studies involving software development tools and tool users; and lastly, Section 2.6 summarizes this chapter and introduces the next one.

2.1 Design Patterns

Design patterns are reusable solutions for common and recurring problems of software design [37]. A design pattern is generally composed of a description of a problem that occurs frequently, followed by a general solution to the problem described and the respective outcomes [37]. Design patterns enable the reuse of knowledge from other developers of previously implemented solutions for certain problems. The best-known design patterns are the 26 described in the book "Design Patterns: Elements of Reusable Object-Oriented Software", whose authors are commonly known as the "Gang of Four" [37]. In their book, each design pattern is presented following a defined template containing various information:

1. the name of the pattern,

2. the design problem the pattern solves,
3. an abstract description of its components, their relationships and responsibilities,
4. the results and trade-offs from using the design pattern.

The 26 patterns are also classified into three groups according to their purpose: Creational, Structural, and Behavioral patterns.

The **creational patterns** [37] are the ones that deal with object creation mechanisms, reducing the complexity of creating objects and in a controlled manner. There are five creational design patterns: Abstract Factory, Builder, Factory Method, Prototype, and Singleton. For these patterns, there are two recurring themes: encapsulating knowledge about which concrete class the system uses, and hiding how the classes are created and composed. Among these five patterns, we explain in greater detail the Prototype and Singleton patterns as they will be discussed in the comparative study (Chapter 4).

The Prototype pattern [37] substitutes the direct instantiation of objects for a copying process of a prototypical instance. It is used when a system should be independent of how its products are created, composed and represented. The prototype defines two roles: the Prototype role and the Concrete Prototype role. The class that plays the Prototype role declares an interface for cloning itself, while the classes that play the Concrete Prototype role implement the cloning operation. The Prototype pattern effectively hides the concrete product classes from the client. However, implementing the clone operation may not be trivial in some cases, for example, if the class contains other objects that do not support copying or have circular references.

The Singleton pattern [37] is used to ensure that a specific class has only one instance while providing a global point of access to it. It is a simple design pattern and it contains only the Singleton role. The class that plays the Singleton role hides its constructor and provides a public operation that returns the sole instance of the class. By hiding the constructor, it ensures that the class can never be instantiated from outside the class. Moreover, the public operation enables a global access point to a single instance of the class. A liability of using this design pattern is that it forces the instantiation of the Singleton class even if it is not used.

The **structural patterns** [37] are the ones that deal with the composition of objects in order to create larger structures to realize new functionality. There are seven structural design patterns: Adapter, Bridge, Composite, Decorator, Face, Flyweight, and Proxy. Similarly to the creational design patterns, we explain in greater detail two patterns that will be used later on: Composite and Decorator.

The Composite pattern [37] allows the clients to treat both individual objects and compositions of these individual objects uniformly by composing said objects into tree structures. This design pattern defines three roles: the Component, the Leaf, and the Composite. The Component defines the interface of the objects that will be handled.

This interface is then implemented by the Leaf, which gives concrete implementations to the Component methods. Lastly, the Composite defines the complex objects which are composed of components that have children. It stores child components and implements child-related operations.

The Decorator pattern [37] defines the dynamic attachment of additional responsibilities to an object. Instead of implementing a subclass that contains the additional functionality at compile time, the Decorator pattern provides it dynamically at run-time. This design pattern is composed of four roles: Component, Concrete Component, Decorator, and Concrete Decorator. The Component defines the interface of the objects that will receive new responsibilities dynamically. The Concrete Component is the concrete implementation of the previous interface. The Decorator defines an interface of the object that will be decorated (the Component) and maintains a reference to a Component object. Lastly, the Concrete Decorator overrides the Component's method, providing additional functionality.

The **behavioral patterns** [37] focus on the communication between objects, providing flexibility for complex control flows. There are 11 behavioral design patterns: Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, and Visitor. Once again, for this study, we will provide details for the Chain of Responsibility and Visitor patterns.

The Chain of Responsibility pattern [37] allows a request to be passed down a chain of objects until one of them is able to handle the request. It allows for a flexible and decoupled way of handling requests, enabling complex flows for handling a request. This design pattern is composed of two roles: Handler and Concrete Handler. The Handler defines the interface for handling the requests, while the Concrete Handler implements the interface and handles the request. The Concrete Handler also contains information about the next Concrete Handler in case it is unable to process the request.

The Visitor pattern [37] allows the implementation of new operations for objects without changing the classes without modifying the objects themselves. It is composed of four roles: Visitor, Concrete Visitor, Element, and Concrete Element. The Visitor declares the "visit" operation for each Concrete Element class, which allows the visitor to access the element. The Concrete Visitor implements each "visit" operation. The Element defines an "accept visitor" operation that takes a visitor as an argument, while the Concrete Element implements the interface.

2.2 Design Pattern Detection Tools

Design pattern detection (DPD) tool is a software that analyzes the source code to extract instances of design patterns implemented. These tools implement algorithms to automate the detection of design patterns. This detection process is a task that becomes increasingly complex to be done manually due to the size of the code base that is being evaluated. Helping software developers understand a system in order to perform maintenance or evolution tasks is among the main benefits of using DPD tools [27].

The DPD tools implement algorithms that leverage different types of analysis and modeling strategies. Regarding the types of analysis, the three main types are static, dynamic, and hybrid [9]. The static analysis consists of the analysis of the static code, extracting information from classes, methods, and the architecture in general. Meanwhile, a dynamic analysis evaluates the execution flow of the system, such as method calls. These two analysis types are not mutually exclusive and one can complement the other, which is precisely what the hybrid analysis consists of: a combination of static and dynamic analysis.

Moreover, the DPD tools generally employ a modeling strategy to transform the code base information for the employed type of analysis. For example, tools that implement static analysis extract static information from the system, such as classes, methods, relationships between classes, variables, operations, etc [9]. Then, they may transform this information into graphs, abstract syntax trees, or custom templates. Eventually, they perform some isomorphism algorithm to match these structures with the predefined structures that represent the design patterns. Meanwhile, for tools that implement dynamic analysis, the algorithms usually evaluate the execution flow in terms of method calls, object instantiation, accessed or modified fields, etc [9].

2.3 Accuracy Metrics and Agreement

Accuracy metrics are statistical measures used to evaluate the performance of a predictive model [70]. These metrics are used to assess how well the model is able to make correct predictions on a given set of data. In the context of DPD tools, they are used to evaluate the instances obtained when executing the detection algorithm. The most common accuracy metrics are Precision, Recall, F-Measure.

Precision [70] measures the proportion of true positives (correctly predicted positive outcomes) out of all positive predictions made by the model. Regarding the output of DPD tools, it is the ratio between the number of true positive design patterns detected and the number of instances detected. The precision metric measures how often the instances detected by the DPD tool are correct.

$$Precision = \frac{TruePositives}{TotalInstances}$$

Recall [70] measures the proportion of true positives (correctly predicted positive outcomes) out of all actual positive outcomes in the dataset. In the DPD tools context, it is the ratio between the number of true positive design patterns detected and the number of existing design patterns in the system. The recall metric measure how complete, in terms of existing design patterns in the system, is the output of the tool.

$$Recall = \frac{TruePositives}{TotalTruePositives}$$

F-Measure [70] is a weighted average of precision and recall. It provides a single score that balances both precision and recall to give an overall measure of a model's performance. It can also be obtained by calculating the harmonic mean of the precision and recall.

$$F\text{-}Measure = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

Lastly, the agreement metric is a statistical measure that is used to evaluate the level of agreement between two or more raters or judges who are assessing a set of data [38]. It is an important tool for assessing the reliability and validity of data collected by multiple raters or judges. In the context of DPD tools, it measures the agreement between multiple tools when detecting design patterns instances from the same system.

There are several types of agreement metrics that can be used, depending on the nature of the data and the type of analysis being conducted. Some commonly used agreement metrics include Cohen's Kappa [38], Fleiss' Kappa [38] and Intraclass Correlation Coefficient (ICC) [38]. In this work, we employed Gwet's AC1 agreement coefficient [38] which is a statistical measure used to evaluate the level of agreement between two or more raters or judges who are evaluating a categorical dataset. It was proposed by Gwet [38] as an improvement over Cohen's Kappa, which can only be used for two raters.

The AC1 statistic ranges from -1 to 1, where 1 indicates perfect agreement, 0 indicates agreement due to chance, and negative values indicate disagreement. A value of 0.6 or above is generally considered to indicate substantial agreement, while values between 0.4 and 0.6 indicate moderate agreement, and values below 0.4 indicate poor agreement [38].

2.4 Existing Literature Reviews on DPD Tools

Since the Gang-of-Four (GoF) book [37] has highly influenced software engineering research and practice, many papers have been published in the literature investigating design patterns and their detection methods [19, 26, 34, 64, 78]. This section, therefore, focuses on the most related secondary and comparative studies that investigate design patterns.

Some secondary studies [60, 84] have investigated and summarized the existing literature on design patterns. For instance, Riaz et al. [60] 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 to empirical studies involving human participants. In fact, our work focuses on automated approaches to detect design patterns.

Yarahmadi and Hasheminejad [84] present a broad systematic literature review on design patterns, including their detection methods. Their secondary study reviews research papers published between 2008 and 2019 and aims to answer 13 research questions. One of these questions is related to what approaches exist to support DPD and which approaches are the most frequently used. Unlike Yarahmadi and Hasheminejad's work [84], we focus on DPD tools. Moreover, we also present a comparative study of three detection tools with respect to their precision, recall, and agreement.

Regarding comparative studies, Dong et al. [27] 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. They also discuss 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. [59]. 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.

2.5 Closely Related Survey Studies

To the best of our knowledge, this dissertation reports the first survey-based study designed for capturing the perceptions of tool designers and tool users on DPD tools. Thus, we could not cross our study data with data reported by past work. Still, we found a few survey- or interview-based studies [36, 44, 47, 80] on the expected usefulness of software development tools other than DPD tools.

The first study [36] reports on the findings of a survey with software developers on software documentation tools. These tools support a variety of documentation tasks, such as document generation and validation. A total of 32 participants were recruited from an academic course, similar to what we did for the survey with potential DPD tool users. The replacement of manual with automated documentation, as well as the ease to use a documentation tool, were pointed out as perceived benefits of using tools. Both process automation and ease to use are aligned with responses to our second survey with DPD tool users.

The second study [44] reports on interviews conducted with software developers about automated code inspection tools. These tools were designed for facilitating the detection of software defects or bugs. A total of 20 participants were recruited from industry contacts. The authors opted for semi-structured interviews, which implies pre-defining questions in the form of a flexible survey (questions can be skipped, customized, and rearranged) [41]. The authors found that developers often perceive inspection tools as useful; however, low accuracy and difficulty to use are clear barriers to the tool's adoption. This finding is a perfect match with the findings of our second survey.

The third study [47] is a multi-case study on the perceptions of students on software modeling tools. These modeling tools are mostly in the context of the Unified Modeling Language (UML) [72] and object-oriented programming. A total of 369 undergraduate students were recruited for the survey, which is similar to what we did in our work but at a larger scale. The authors concluded that, beyond the inherent complexity of using a tool, the type of software modeling supported affects the perceived usefulness of a tool. This finding is similar to what we found regarding the potential usefulness of DPD tools, which depends on the need for automating the detection of a specific design pattern.

The fourth study [80] is a survey with developers about security tools. Analysis tools that find and fix vulnerabilities belong to the scope of security tools targeted by the study. Authors conducted two survey iterations. The first iteration had 119 participants, who were recruited by convenience as they were industry contacts of the authors. The second iteration had 61 valid responses given by developers recruited from expert mailing lists. Such large samples were recruited via drawing or distribution of gift cards, a practice we cannot replicate as we are researchers with very scarce resources. Study results suggest

that users tend to discard security tools perceived as complex to use, which is aligned with the findings of our survey with DPD tool users. Thus, we reinforce our recommendation for well-documented tools as a means to increase tool adoption.

2.6 Chapter Summary

In this chapter, we presented some background information important for understanding the remaining of this work. We discussed design patterns and described the six patterns that will be used in our comparative study (Chapter 4 in terms of use, benefits, and roles. We also presented what are design pattern detection tools, their benefits and purpose, along with common types of analysis and modeling strategies used for implementing the detection approaches. The precision, recall, F-measure, and agreement metrics were also introduced, as they are also important results obtained from our comparative study. Lastly, we discussed some related work regarding both DPD tools and survey studies involving software development tools and tool users.

In the next chapter, we present our systematic literature review which is our first building block for conducting the remaining of our proposed studies. We describe our review goal, protocol and discuss our findings regarding the DPD tools developed in the last 20 years.

Chapter 3

Literature Review on DPD Tools

There are several design pattern detection (DPD) tools available in the literature [19, 26, 34, 64, 78]. Moreover, each tool presents different features such as different detecting strategies, different detectable design patterns, different supported languages, etc. As the DPD tools provide distinct functionalities, it is important to know them beforehand in order to choose the tool best suited for a specific situation. However, previous studies that intended to summarize these tools [60, 84] contained some gaps in their scope. For instance, they do not provide information about the DPD tools' main features and scopes. In order to solve this problem, we conducted a Systematic Literature Review (SLR). This SLR enables us to create a catalog of DPD tools that can be used by both researchers and practitioners interested in using or extending them. Once we obtained the final list of studies, we conducted a full read of the papers and extracted the information of interest (i.e. the tools' features).

In this chapter, we present our SLR, structured as follows: Section 3.1 introduces the SLR study protocol, presenting our goal, research questions and study design. Section 3.2 presents the results of the study, answering the research questions, presenting our result artifacts and discussions. Section 3.3 discusses the different threats to the validity of this study. Lastly, Section 3.4 summarizes this chapter and introduces the next one.

3.1 Study Protocol

This section presents the design decisions and rationale for conducting this SLR.

3.1.1 Goal and Research Questions

We defined our study goal based on the Goal-Question-Metric template [11] as follows: This work aims 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 the tools, as well as the main features of the 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?* – The Gang of Four's book [37] was first published in 1995 as an extensive catalog of design patterns. This catalog relied on decades of industry experience to assist researchers and practitioners concerned with poor software design. Several studies aimed to automate DPD as a means to facilitate software comprehension and design decision making [4, 21, 64, 74, 86]. Through RQ_1, we assess how the publication of DPD tools has evolved over time, thereby revealing how active and mature is the existing research on this topic.

RQ_2: *What DPD tools were published in the last two decades and what are their main features?* – Our major study goal is to assist practitioners and researchers while looking for useful tools, as well as to evaluate how the existing tools perform. We split RQ_2 into three questions to help us achieve that goal.

$RQ_{2.1}$: *What are the existing DPD tools?* – Our study covers two decades of publications, so that: 1) many existing DPD tools may be unavailable; 2) these tools may depend on either obsolete or unavailable dependencies; and 3) the tools may be incompatible with the existing Integrated Development Environments (IDEs). Thus, we have to list these tools and identify those that are available for download and use before we perform the tool comparison.

$RQ_{2.2}$: *What are the main features of the existing DPD Tools?* – We also have to identify the main features by tool, e.g., the scope of detectable design patterns and programming languages supported. By investigating this question, we expect to derive an extensive catalog of DPD tools that could be useful for whoever is looking for tools suitable to their needs.

$RQ_{2.3}$: *What detection approaches have been used by the existing DPD tools?* – Program analysis approaches can rely on static analysis, dynamic analysis or hybrid analisis [22]. *Static analysis* targets source code and its structure; *dynamic analysis* targets software execution; and *hybrid analysis* combines the two other approaches. Analysis can rely on different problem modeling approaches [33], such as Abstract Syntax Tree (AST) and tokens. By investigating the variety of approaches being used, we expect to reveal opportunities to further explore certain approaches, for instance.

3.1.2 Search for Primary Studies

Table 3.1 lists the four Web search engines we selected to support our search for primary studies. All these engines provide a vast amount of studies published in Software Engineering [29]. These engines have been used for conducting SLRs in various Software Engineering topics [16, 28, 67, 71]. They have also been used in SLRs related to DPD, e.g., software pattern application [60] and anti-pattern detection tools [33].

Table 3.1: Web search engines used to search for primary studies

Engine	Available at	Number of Studies
ACM Digital Library	https://dl.acm.org	44
IEEE Xplore	https://ieeexplore.ieee.org	18
ScienceDirect	https://www.sciencedirect.com	41
Springer	https://www.springer.com	76

We defined the following search string to be run in 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 output obtained.

The third column of Table 3.1 shows the number of primary studies returned by each engine. 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[1] extension for the Google Chrome browser.

3.1.3 Filtering of Primary Studies

We relied on strict guidelines [46] 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

[1] https://dataminer.io

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 [81] 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. It is important to highlight that in the case of multiple publications for the same tool, due to a new or enhanced version of the same tool, we discarded the previous version and listed only the most recent one. For example, the MARPLE-DPD tool [86] is a more recent version of the MARPLE tool [35], however, they are different from the nMARPLE [6] which supports DPD on a different programming language.

3.1.4 Data Extraction from Primary Studies

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

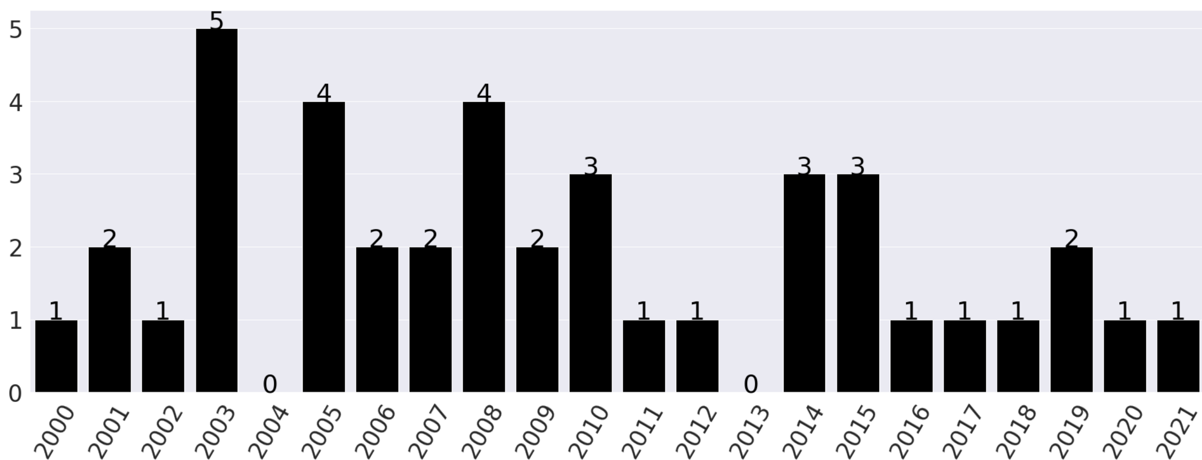
3.2 Results and Discussions

This section presents the findings of the SLR and their discussion, and the responses to our research questions.

3.2.1 Publication Landscape (RQ_1)

Figure 3.1 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 that presents each DPD tool was published. We found out that new tools have been introduced every year with the exception of 2004 and 2013. This result suggest that introducing novel DPD tools is still an interest of researchers worldwide.

Figure 3.1: Number of DPD tools published by year



Source: Elaborated by the author.

Regarding the publication venues, we found that most studies appeared in conferences (56%), followed by journals (32%), symposiums (7%), and workshops (5%). Several studies were published in prestigious venues. Examples of conferences include International Conference on Software Engineering (ICSE), International Conference on Automated Software Engineering (ASE), and International Conference on Software Analysis, Evolution and Reengineering (SANER). Examples of journals include IEEE Transactions on Software Engineering (TSE), ACM Transactions on Software Engineering and Methodology (TOSEM), and Journal of Systems and Software (JSS).

Summary of RQ_1: Introducing DPD tools remains an interest of Software Engineering researchers. The average of two valid primary studies published by year in the last two decades suggests that DPD has open challenges to be addressed. DPD research seems to be maturing since only 32% of the studies appeared in prestigious journals.

3.2.2 Existing DPD Tools ($RQ_{2.1}$)

Tables 3.2 and 3.3 collectively summarize the 42 DPD tools revealed by our literature review. Table 3.2 presents the tools available for download while Table 3.3 presents the ones unavailable. 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 [20], 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.

Table 3.2: Overview of the tools available for download

Name	Type	Language		Scope of Detectable Design Patterns	Docs	GUI
		Devel.	Suppor.			
DesPaD [51]	Stand -alone	Java	Java	Abstract Factory, Adapter, Bridge, +20	No	No
DPJF [14]	Plug-in	Java	Java	CoR, Composite, Decorator, +2	Yes	Yes
FINDER [18]	Stand -alone	Java	Java	Abstract Factory, Adapter, Bridge, +17	Yes	No
GEML [10]	Stand -alone	Java	Java	Abstract Factory, Adapter, Builder, +12	Yes	Yes
MARPLE-DPD [86]	Plug-in	Java	Java	Adapter, Composite, Decorator, +19	Yes	Yes
PatRoid [61]	Stand -alone	Python	Java*	Abstract Factory, Adapter, Bridge, +20	Yes	No
PINOT [64]	Stand -alone	C++	Java	Abstract Factory, Adapter, Bridge, +14	No	No
PTIDEJ [48]	Stand -alone	Java	Java	Adapter, Builder, CoR, +14	Yes	Yes
Reclipse [77]	Plug-in	Java	Java	CoR, Command, Composite, +2	Yes	Yes
SSA/DPD [74]	Stand -alone	Java	Java	Adapter, Command, Composite, +9	No	Yes

*Android apps written in Java.

The third and fourth columns report what programming languages were used for developing each tool and in which languages the tool can perform DPD. Java is a very

Table 3.3: Overview of the tools unavailable for download

Name	Type	Language		Scope of Detectable Design Patterns	Docs	GUI
		Devel.	Suppor.			
Alnusair et al. [3]	Stand-alone	N/A	Java	Composite, Observer, Singleton and 2 other	No	N/A
Antoniol et al. [5]	Stand-alone	Java	C++	Adapter, Bridge, Composite and 2 other	No	Yes
APRT [62]	Stand-alone	Java	Java	Singleton, Strategy	No	N/A
CrocoPat [13]	Stand-alone	N/A	Java	Composite	No	N/A
D^3 (D-cubed) [68]	Stand-alone	Java	Java	Abstract Factory, Builder, Factory Method, Singleton	No	No
DEPAIC++ [31]	Stand-alone	Java	C++	Abstract Factory, Composite, Iterator	No	Yes
DP-Miner [26]	Stand-alone	N/A	Java	Adapter, Composite, Decorator, State	No	Yes
DPDT [65]	Stand-alone	C++	Java	Bridge, Composite, Flyweight and 4 other	No	N/A
DPF [12]	Plug-in	Java	Java	Adapter, Command, Composite and 8 other	No	Yes
DPRE [20]	Stand-alone	Java	Java	Adapter, Bridge, Command and 8 other	No	Yes
DPVK [78]	Stand-alone	N/A	Eiffel	Abstract Factory, Adapter, Bridge and 15 other	No	No
ePad [21]	Plug-in	N/A	Java	Abstract Factory, Builder, Command and 9 other	No	Yes
Hedgehog [15]	Stand-alone	N/A	Java	Bridge, Factory Method	No	N/A
Heuzeroth et al. [39]	Stand-alone	Java	Java	Chain of Responsibility, Mediator, Observer, Visitor	No	N/A
Heuzeroth et al. [40]	Stand-alone	N/A	Java	Composite, Decorator, Observer	No	N/A
JADEPT [7]	Stand-alone	Java	Java	Chain of Responsibility, Observer, Visitor	No	Yes
Maisa [52]	Stand-alone	Java	UML	Abstract Factory	No	N/A
nMarple [6]	Plug-in	N/A	.NET	Abstract, Builder, Chain of Responsibility and 13 other	No	Yes
PatternFinder [24]	Stand-alone	N/A	.NET	Abstract Factory, Adapter, Bridge and 20 other	No	Yes
Philippow et al. [55]	N/A	N/A	C++	Abstract Factory, Adapter, Bridge and 20 other	No	N/A
PRAssistor [42]	Stand-alone	Java	Java	Composite, Singleton, Visitor	No	No
Rasool & Mader [57]	Stand-alone	C#	C++, Java	Abstract Factory, Adapter, Bridge and 20 other	No	Yes
Rasool & Mader [58]	Plug-in	.NET, C#	C#, C++, Java	Abstract Factory, Adapter, Bridge and 20 other	No	Yes
Sartipo & Hu [63]	Plug-in	Java	Java	Adapter, Bridge, Decorator and 4 other	No	Yes
SparT-ETA [83]	Stand-alone	Java	Java	Abstract Factory, Adapter, Bridge and 19 other	No	N/A
SPQR [66]	Stand-alone	N/A	Language Independent	Decorator	No	N/A
Thongrak et al. [73]	Stand-alone	N/A	UML	Strategy	No	Yes
Van Doorn et al. [75]	Stand-alone	Java	UML	Abstract Factory, Adapter, Bridge and 13 other	Yes	No
Vokác [76]	Stand-alone	N/A	C++	Decorator, Factory Method, Observer and 2 other	No	N/A
VPML [30]	Plug-in	Java	UML	Abstract Factory, Adapter, Bridge and 8 other	No	Yes
Web of Patterns [23]	Plug-in	Java	Java	Abstract Factory, Adapter, Bridge and 7 other	No	Yes
Zhang & Li [87]	N/A	N/A	C++	N/A	No	N/A

popular language[2], 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 3.2.3 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 discussed in our tool comparison (Section 4.1.2), 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 feature is important for practitioners looking for easy-to-use interfaces, since GUIs aim at facilitating user interactions [43]. We detail the data summarized by this column in Section 3.2.3.

Summary of $RQ_{2.1}$: From the 42 DPD tools found, 71% are stand-alone. In addition, 69% of the tools are compatible with Java systems. However, only 19% of the tools have some companion documentation. These numbers suggest a certain niche of applicability of the tools.

3.2.3 Main Features of DPD 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 [30, 52, 73, 75] rather than on source code. One tool is advertised as language independent [66]. 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 3.4 shows the number of tools that detect each of the 23 design patterns cataloged by the Gang of Four (GoF) [37]. 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

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

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 3.4: 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 M.	22	52%
State	21	50%	Adapter	21	50%	Abstract F.	19	45%
Visitor	21	50%	Proxy	20	48%	Prototype	14	33%
Template M.	20	48%	Bridge	19	45%	Builder	13	31%
Command	18	43%	Flyweight	12	29%			
CoR	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.

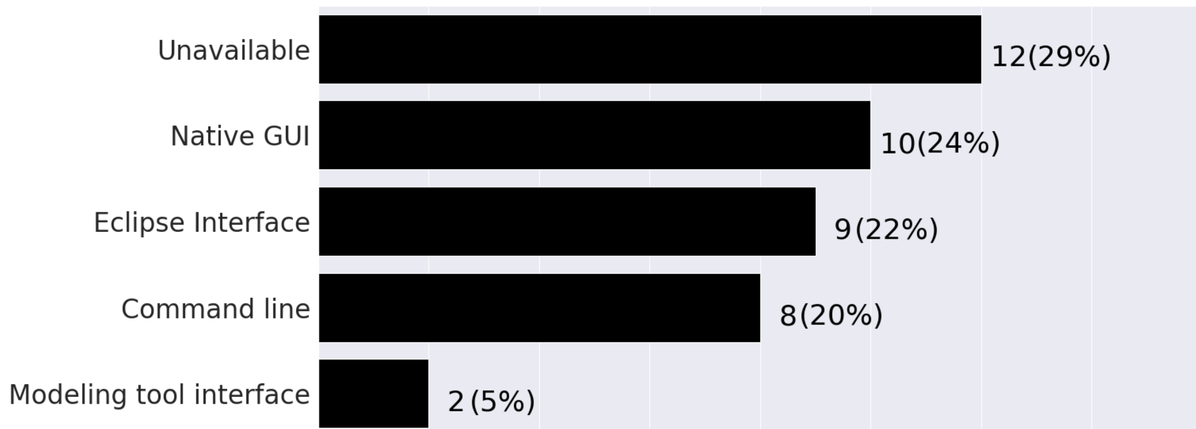
Download Availability: We assessed the download availability of DPD tools by searching for download links within the studies. We used these links for downloading the tools whenever they were available. Otherwise, we Google searched for the tool name. We found that ten out of the 42 DPD tools (24%) were downloadable during our study. This result is reasonable if we consider our wide range of publication year, i.e., two decades of research. Table 3.3 lists the tools that are were available for download during our search.

GUI Availability: Figure 3.2 presents the number of DPD tools by their Graphical User Interface (GUI). As we know, DPD may be challenging in practice. 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[3] and, therefore, use the Eclipse GUI; 8 tools (19%) rely on command line; and 2 tools (5%) are

[3]https://www.eclipse.org/ide/

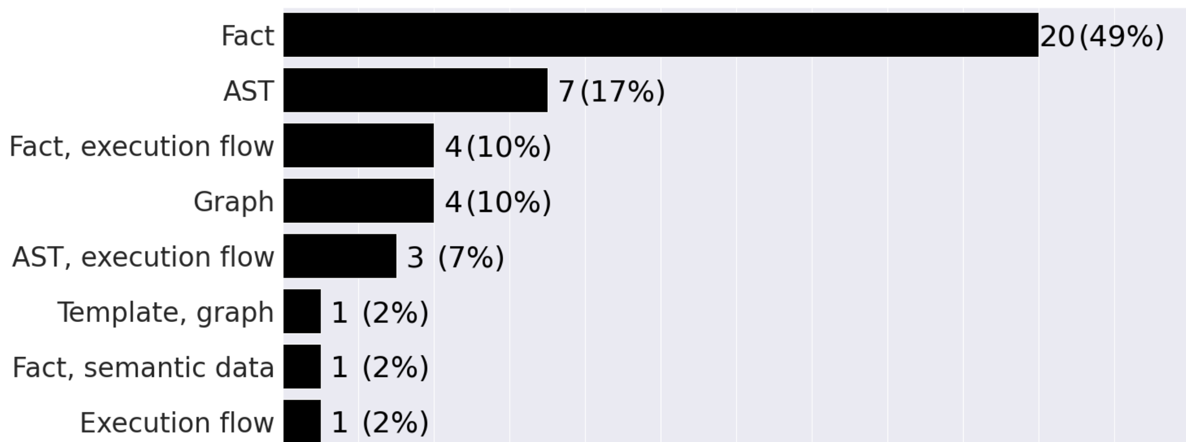
embedded to software modeling tools and, thus, use the GUI provided by these modeling tools. We were unable to find out if there is a GUI provided by 13 out of the 42 tools (31%), due to poor documentation or their unavailability online.

Figure 3.2: Number of tools by GUI type



Source: Elaborated by the author.

Figure 3.3: Number of tools by problem modeling approach



Source: Elaborated by the author.

Summary of $\mathbf{RQ_{2.2}}$: From a total of 41 tools, 68% are designed to perform DPD on Java systems, support the detection of complex design patterns such as Composite (66%) and Observer (66%), and provide users with some sort of GUI other than command line (51%). On the other hand, only a few tools (29%) are free to use and downloadable (24%).

3.2.4 Detection Approaches behind the DPD Tools ($RQ_{2.3}$)

Table 3.5: Detection approaches of the tools

Name	Analysis Type	Modeling Type
DesPaD [51]	Static analysis	AST
DPJF [14]	Static analysis	Fact
FINDER [18]	Static analysis	Fact
GEML [10]	Static analysis	Fact
MARPLE-DPD [86]	Static analysis	AST
PatRoid [61]	Static analysis	Fact
PINOT [64]	Static analysis	AST
PTIDEJ [48]	Static analysis	Fact
Reclipse [77]	Hybrid analysis	AST and Execution Flow
SSA/DPD [74]	Static analysis	Graph
Alnusair et al. [3]	Static analysis	Fact
Antoniol et al. [5]	Static analysis	Fact
APRT [62]	Static analysis	AST
CrocoPat [13]	Static analysis	Fact
D^3 (D-cubed) [68]	Static analysis	AST
DEPAIC++ [31]	Static analysis	Fact
DP-Miner [26]	Static analysis	Template and Graph
DPDT [65]	Static analysis	Graph and Fact
DPF [12]	Static analysis	AST
DPRE [20]	Hybrid analysis	Fact and Execution Flow
DPVK [78]	Static analysis	Fact
ePad [21]	Hybrid analysis	Fact and Execution Flow
Hedgehog [15]	Static analysis	Fact and Semantic Data
Heuzeroth et al. [39]	Hybrid analysis	AST and Execution Flow
Heuzeroth et al. [40]	Hybrid analysis	AST and Execution Flow
JADEPT [7]	Dynamic analysis	Execution Flow
Maisa [52]	Static analysis	Fact
nMarple [6]	Static analysis	Graph
PatternFinder [24]	Hybrid analysis	Graph
Philippow et al. [55]	Static analysis	Fact
PRAssistor [42]	Hybrid analysis	Fact and Execution Flow
Rasool & Mader [57]	Static analysis	Fact
Rasool & Mader [58]	Static analysis	Fact
Sartipo & Hu [63]	Hybrid analysis	Fact and Execution Flow
SparT-ETA [83]	Hybrid analysis	Graph
SPQR [66]	Static analysis	AST
Thongrak et al. [73]	Static analysis	Fact
Van Doorn et al. [75]	Static analysis	Fact
Vokác [76]	Static analysis	Fact
VPML [30]	Static analysis	Fact
Web of Patterns [23]	Static analysis	Fact
Zhang & Li [87]	Static analysis	Fact

Program Analysis Approaches: Table 3.5 presents the information about both the type of program analysis and problem modeling approaches implemented by each tool. 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. In other words, it makes sense that researchers often rely on static analysis to perform DPD. In fact, only one tool [7](2%) relies on dynamic analysis, which depends on tracking and understanding the execution of a software system (Section 3.1.1). 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 [21, 22] may be preventing researchers to advance in proposing tools based on either dynamic or hybrid analysis.

Problem Modeling Approaches: Figure 3.3 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 [18, 57, 75]. This is followed by 15% of the tools relying only on Abstract Syntax Tree (AST), whose nodes represent variables, operations, and other program elements [39, 62, 64]. 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 [24]. 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 [26].

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 [7, 42]. 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 [15]. 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.

Summary of $\mathbf{RQ_{2.3}}$: Regarding the program analysis approaches, most of the tools (76%) rely on static analysis to support DPD. Additionally, 76% of the tools rely on a single modeling approach, mostly depending on information of source code and its structure.

3.3 Threats to Validity

This section discusses the mains threats to the validity of our SLR study.

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 primary studies as possible. Moreover, we performed snowballing to avoid overlooking important studies. We relied on existing guidelines [46] to define our inclusion and exclusion criteria of primary studies, thereby filtering out irrelevant studies. Finally, two researchers collaborated in both reading the primary studies for performing the data collection. They carefully discussed any divergences until reaching 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.

Conclusion and External Validity: We performed descriptive analysis in order to interpret the literature review data, similarly to other similar work [16, 33, 28, 71]. 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 [33], some of them specific to the Computing domain such as ACM Digital Library. Thus, we expected to support the amplitude of our study results.

3.4 Chapter Summary

In this chapter we presented the design and results of our systematic literature review about design pattern detection tools published in the last 20 years. From the 42 DPD tools found, 68% are stand-alone and only 19% of the tools have some companion documentation. In addition, 69% of the tools are designed to perform DPD on Java systems, support the detection of complex design patterns such as Composite (67%) and Observer (64%), and provide users with some sort of GUI other than command line (50%). On the other hand, only a few tools (29%) are explicitly free to use and downloadable (24%). Regarding the program analysis approaches, most of the tools (76%) rely on static analysis to support DPD. Additionally, 76% of the tools rely on a single modeling approach, mostly depending on information of source code and its structure.

In the next chapter, we use our catalog of tools and other information from our SLR to conduct a comparative study about the performance of DPD tools in terms of precision, recall and agreement. The information about the DPD tools' performance complements the information regarding the technical aspects, such as design patterns support and programming language supported. The tool comparison allows not only practitioners to find the best performing tools according to their needs, but also researchers to evaluate which kind of approaches yield better results when detecting specific design patterns and also to come up with new detection approaches.

Chapter 4

Comparative Study of DPD Tools

Design pattern detection (DPD) tools rely on different approaches in order to extract the instances from evaluated systems. Each type of approach may provide different outputs for the same system, yielding a varying number detected instances with different levels of precision. Thus, it is important to know how reliable the DPD tools are. However, studies available in the literature [60, 84] do not present a comparison of the instances detected from installing and executing different tools and the resulting accuracy metrics. These studies simply reuse the reported performance results from the original studies. This can be considered a threat to the validity to these studies, since it can be a form of propagation of possible invalid results from the original studies. Furthermore, they also do not present information about the agreement between the tools regarding the detection of design pattern instances. Understanding the level of agreement between different tools may allow the proposal of novel approaches that combine previous ones for example. Information about the accuracy and agreement of the tools complements the technical details of the tools, such as supported programming languages and design patterns.

In this chapter, we tackle this issue by conducting a comparative study of the design pattern instances detected by different DPD tools in terms of precision, recall and agreement. From the list of DPD tools obtained in the previous chapter, we used four tools that were available for download and that we were able to properly use. We used the tools to detect six design patterns from two different systems. We manually classified the output instances as true or false positives and computed precision, recall and agreement metrics.

The chapter is structured as follows: Section 4.1 introduces the comparative study protocol, presenting the goal, research questions and study design. Section 4.2 presents the results of the study, answering the research questions, presenting the results artifacts and discussions. Section 4.3 discusses the different threats to the validity of this study. Lastly, Section 4.4 summarizes this chapter and introduces the next one.

4.1 Study Protocol

This section presents the design decisions and rationale for conducting this comparative study.

4.1.1 Goal and Research Questions

We define the goal of this 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?* – To investigate the accuracy of tools, we computed three metrics regarding the detection of different design patterns in two software systems (see Section 4.1.3). The *precision* metric captures how often the tool identifies design patterns correctly. *Recall* helps understand whether the tool manages to detect all known correct instances of design patterns or only a subset of instances. Finally, *F-measure* corresponds to the harmonic mean of precision and recall.

RQ$_4$: *To what extent do the existing tools agree regarding the detection of design pattern instances?* – Section 3.2.4 discusses that each DPD tool relies on different detection approaches. As a result, different tools may be able to detect different instances of design patterns in the same system. Through RQ$_4$, we investigate the agreement degree among existing tools. We aim to understand whether the tools provide us with redundant or complementary detection results.

4.1.2 Selection of Tools

Although our literature review revealed 42 DPD tools (Section 3.2), 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 [61] 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 [18], GEML [10], MARPLE-DPD [86], and PTIDEJ [48]. As this was an already small enough subset of tools, we used the four tools for the comparative study. We discarded DesPaD [51] due to configuration issue (stuck on the second step, probably due to an issue with Subdue); DPJF [14] due to use error when attempting to generate the factbase; PINOT [64] due to installation error when attempting to compile with the `make` command; Reclipse [77] due to installation issue (dependencies could not be installed, i.e., Palladio and SoMoX); and SSA [74] 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.

4.1.3 Selection of Design Patterns and Software 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 [14, 20, 74].

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 criteria were destined to make 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 capabilities of the tools detection prowess, each category comes with a different difficulty level for the detection process which may be an indicator of the tool's capability of coping with these difficulties.

The second criterion was created to make the calculation of the agreement coefficient feasible, since we need at least two raters to compute this metric. Creational design patterns help make a system more flexible by making the process of instantiating classes abstract [37]. The chosen creational patterns were Prototype and Singleton. Behavioral design patterns distribute behavior between classes by describing the communication between them [37], 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 [37], the structural patterns selected were Composite and Decorator.

4.1.4 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 the 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 4.1 presents the number of instances detected of each design pattern by each tool. 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.

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 4.2 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.frame-`

Table 4.1: 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

`work.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 4.2: Patterns and their 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 positives by manually inspecting all detected instances. For each design pattern, two researchers 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 researchers 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 researchers 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 [1]. 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[1], 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.

4.1.5 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 [8]. 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 [8]. 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. 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,

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

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 [38], Overall Agreement and Confidence Interval over 95% using Gwet's R package, i.e., `irrCAC`[2].

4.2 Tool Comparison Results

Precision, Recall, and F-measure (RQ_3): Table 4.3 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.

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 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. We attempted to compare the metrics obtained in this study with the results presented in the studies that presented each of these four tools. However, the studies either did not present any performance metrics [48] or presented the overall metrics for a collection of systems [86, 10] or different systems [18], thus, preventing us from conducting this comparison.

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

Table 4.3: Accuracy measures by tool

System	Metric	Tool	CoR	Comp.	Deco.	Prot.	Sing.	Visi.
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]

Summary of RQ$_3$: The comparison results suggest that each tool could be applied for detecting a particular subset of design patterns. FINDER showed sufficient accuracy for Composite, Decorator, Singleton and Visitor. Still, future work is required to further support practitioners interested in automating DPD in practice.

Agreement of the Tools (RQ$_4$): The last three rows from Table 4.3 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 [38], 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.

Summary of RQ_4: For the six design patterns evaluated, the four DPD tools tend to disagree on the instances detected, meaning that their output is not redundant. In other words, they detect distinct design pattern instances.

4.3 Threats to Validity

Construct and Internal Validity: We carefully installed and configured the tools available online for the comparative study (Section 4.1). 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 researcher 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 researchers. 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 [33]. 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 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 [37]. We selected pattern of different categories, i.e., Behavioral, Creational, and Structural, to support some variety in our findings. Lastly, similarly to the previous threats, although we analyzed only two systems both are functional systems that contain implementations of design patterns and are commonly used for benchmarking DPD tools in the literature.

4.4 Chapter Summary

In this chapter, we presented the design and results of our comparative study of four different design pattern detection tools. We computed the precision, recall and agreement of each tool when detecting six different design patterns from two systems. The comparison results suggest that each tool could be applied to detecting a particular subset of design patterns. FINDER showed sufficient accuracy for Composite, Decorator, Singleton and Visitor. Still, future work is required to further support practitioners interested in automating DPD in practice. For the six design patterns evaluated, the four DPD tools tend to disagree on the instances detected, meaning that their output is not redundant. In other words, they detect distinct design pattern instances.

In the next chapter, we present two survey studies: the first to obtain evidence about the design rationale behind the proposal of the tools, and the second to obtain evidence regarding the expected usefulness from the point of view of practitioners. We expect that the results of the first survey allow us to understand the technical difficulties that tool designers may face when developing DPD tools, while the results of the second survey would enable tool designers to propose new tools (or refine existing ones) that match the needs of potential users.

Chapter 5

Survey Study on DPD Tools

There are several design pattern detection (DPD) tools available in the literature (3). However, their scope of detection and supported programming languages are not very diverse. For example, out of 42 tools published in the last 20 years, 28 tools (67%) are able to detect Composite while only 8 can detect Facade. Regarding the supported programming languages, 29 tools are able to extract instances from Java systems, but no tool can detect design patterns in JavaScript or Python systems. Furthermore, it is not clear why the DPD tool designers opted for these specific scopes. In fact, there is no evidence on the expected usefulness of DPD tools from the perspective of potential tool users. Eventually, we cannot conclude that the available DPD tools match the users' expectations.

In this chapter, we present two survey studies to answer the issues presented: one aimed at capturing the design rationale of tool designers and the other to understand the perception of tool users regarding DPD tools. For the first survey, we elaborated a five question survey and emailed them to the tool developers, eventually obtaining nine responses. Meanwhile, for the second survey we developed an eight question survey using Google Forms and shared it with potential tool users, which yielded a total of 21 responses.

The chapter is structured as follows. Section 5.1 introduces the study protocol, presenting the goal, research questions and design of both surveys. Section 5.2 presents the results of the study, answering the research questions, presenting the results artifacts and discussions. Section 5.5 discusses the different threats to the validity of this study. Section 5.4 discusses the implications of our findings. Lastly, Section 5.6 summarizes this chapter and introduces the next one.

5.1 Study Design

This section provides details regarding our study design as follows. Section 5.1.1 introduces the study goal and research questions (RQs). Section 5.1.2 presents the struc-

ture of our surveys: the survey with tool designers and the survey with potential tool users. Section 5.1.3 describes our study participants for each survey. Lastly, Section 5.1.4 describes our data analysis procedures.

5.1.1 Goal and Research Questions

We relied on the Goal-Question-Metric template [11] to define our study goal as follows: *analyze* the perceptions of tool designers and potential tool users about DPD tools; *for the purpose of* understanding the extent in which existing DPD tools meet the needs of tool users, as well as assisting tool designers in proposing novel tools; *with respect to* i) what reasons have led tool designers to target specific design patterns and programming languages and ii) the expected usefulness of DPD tools from the perspective of potential tool users; *from the point of view of* tool designers who have contributed to academic publications and junior or senior developers who are potential tool users; *in the context of* 42 tools summarized in our systematic literature review (Chapter 3).

We defined the two RQs below to drive our research.

RQ_1: *What is the design rationale behind the proposal of the existing DPD tools?* – Our literature review presented in Chapter 3 catalogs 42 DPD tools published over the last two decades. We found a certain bias in the design patterns these tools detect: at least 64% of the tools detect Composite or Observer, while only a few tools detect other relevant patterns such as Mediator (31%) and Facade (19%). We also found biased the choice for target languages: 69% of the tools target Java, while no tools target trending languages such as e.g., JavaScript and Python. The lack of documentation regarding the reasons why tool designers have targeted specific patterns and languages motivates our RQ. We believe that RQ_1 can shed light on the types of technical support that tool designers need for targeting other patterns and languages of relevance in industry.

RQ_2: *How useful potential tool users expect DPD tools to be?* – A few previous studies [36, 44, 47, 80] rely on surveys or interviews to investigate the expected usefulness of different software development tools. These studies highlight the importance of understanding an eventual mismatch between the tool designers' effort and the practical needs of developers. To the best of our knowledge, there is no survey-based empirical study aimed at understanding the expected usefulness of DPD tools. We advocate that acquiring such understanding is essential to propose novel DPD tools (or refine existing tools) able to meet the current needs of potential users. With RQ_2, we capture contexts in which junior or senior developers expect DPD tools to be useful, as well as benefits and barriers to the use of these tools.

5.1.2 Survey Structure

Survey is aimed at collecting information from people as a means to characterize their opinion, knowledge, or behavior [82]. Surveys help researchers derive conclusions from a sample of participants that does not necessarily reflect an entire population. Thus, survey replication is highly recommended. We opted for a survey-based study due to our positive recent experiences [49, 50]. We relied on strict guidelines for conducting online surveys [56] to propose two complementary surveys: one targeting tool designers and one targeting potential tool users. We opted for short surveys to prevent candidate participants from ignoring our survey. We describe below the structure of each survey.

Survey with tool designers: Table 5.1 presents the five open questions of the survey with tool designers (D1 to D5). We opted not to ask demographic information directly to the tool designers as this information was publicly available on their websites. We customized the survey according to the specific information of each tool (see text between brackets). D1 captures the reasons why the tool designers decided to detect specific design patterns. D2 captures the designers' intent to detect other design patterns they believe to be relevant. D3 and D4 are similar to D1 and D2 but target programming languages rather than design patterns. D5 captures the designers' knowledge of the usage of their tools.

Table 5.1: Structure of the survey with tool designers

ID	Question
D1	Your tool detects [list of design patterns]. Why did you choose them?
D2	Would you implement detection tools for other design patterns? If so, which design patterns would you support? Why?
D3	Your tool detects design patterns in systems implemented in [list of programming languages]. Why did you choose them?
D4	Would you implement detection tools for other languages? If so, what languages would you support? Why?
D5	In which contexts of software development has your tool been used (e.g. adding a new feature to the system, enhancing an existing feature or code, removing an obsolete feature, fixing a bug, etc.)?

Survey with tool users: Table 5.2 lists the eight questions of our survey with tool users: U1 to U5 are closed questions, while U6 to U8 are open questions. U1 to U4 capture demographic information aimed at helping us understand the possible generalization of our findings. These questions cover the programming language most used by the user (U1), years of experience as a programmer (U2), current main role as a programmer (U3), and familiarity with design patterns (U4). U5 to U8 capture the expected usefulness of DPD tools: contexts in which tool users would consider using DPD tools (U5), expected

benefits of using a DPD tool (U6), reasons for not using a DPD tool (U7), and design patterns that are worth detecting via tool (U8).

Table 5.2: Structure of the survey with potential tool users

ID	Question	Possible answers
U1	Which programming language do you use mostly on a daily basis?	JavaScript; Python; Java; C#; PHP; other
U2	How many years of experience do you have as a programmer?	Up to 3 years; 3 - 5 years; 5 - 10 years; more than 10 years
U3	What is your current main role?	Back-end developer; front-end developer; full stack developer; tester; other
U4	How familiar are you with design patterns?	Not familiar; somewhat familiar; fairly familiar; very familiar
U5	In which contexts of software development would you consider using a design pattern detection tool?	Adding a new feature to the system; enhancing an existing feature or code; removing an obsolete feature; fixing a bug; other
U6	What benefits do you expect from using a design pattern detection tool?	N/A (open question)
U7	What would prevent you from using a design pattern detection tool?	N/A (open question)
U8	What design patterns do you believe are worth detecting through a tool? Why?	N/A (open question)

5.1.3 Participant Characterization

As discussed in Section 5.1.2, this chapter introduces an empirical study based on two online surveys, each targeting a specific group of participants. We describe below our participant recruitment procedures and characterize each group.

Group A: *Tool designers* – Our literature review summarizes 42 DPD tools. We used the papers that introduced each tool to extract contact information of the respective tool designers. We extracted name and email address of all authors credited in each paper. The 42 tools were published over a span of 20 years, so that certain email addresses were likely to be invalid. We prevented this issue by searching for the latest papers published by each author on Google Scholar[1]. We then extracted the email addresses provided in these papers; we also extracted the email addresses informed in the author's website whenever its link was provided by the author's profile on Scholar.

[1] https://scholar.google.com

We emailed our survey to all authors (i.e., tool designers). We tried all email addresses extracted per author and, if all addresses were invalid and our emails returned, we simply discarded the author from Group A. We extracted 95 email addresses from a total of 108 tool designers. We waited two months for responses to the survey, so that tool designers (especially designers of older tools) had enough time to gather information about the design decisions of their previous work. Tool designers of nine out of the 42 tools responded to our survey, one designer by tool, thereby covering 21.4% of the tools.

Demographic information for Group A: Out of the nine survey participants, one is a PhD working in the industry, one is a PhD student, one is a PhD working in the industry and also a professor at a university, and six are PhD professors at a university. Regarding their positions in the coauthors' list of their respective papers, four of them are first authors, four are second authors and one is third author. Thus, the survey participants are qualified to provide information about the decisions behind the proposal of the DPD tools.

Group B: *Tool users* – We had no reference list of software developers who are potential users of DPD tools, and the universe of possible survey participants in very broad in this case. Aimed at achieving a diverse group of participants in terms of background, we opted for recruiting tool users based on opportunistic broadcasting. We sent the survey link to three groups of software developers on WhatsApp[2], which is an instant messaging platform largely used in Brazil. We encouraged all developers to forward our survey link to other developers and similar WhatsApp groups. Because we could not track all developers reached by our survey link, it is impossible to compute response rate. Still, we obtained responses from four senior developers.

We also forwarded our survey link to undergraduate students enrolled in the Software Engineering course at the Federal University of Minas Gerais (UFMG), i.e., where the author was enrolled as a Master's student. We gave each student a month to respond the survey, enough time for an opinion-based survey. A total of 17 junior developers responded to our survey.

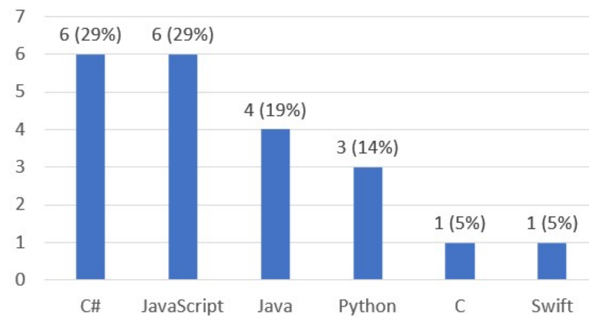
Demographic information for Group B: Figures 5.1, 5.2, 5.3 and 5.4 summarizes demographic information of the potential tool users collected from survey questions U1 to U4 (Table 5.2). Table 5.3 presents the demographic information of each participant. We discuss below our general observations on this matter.

Figure 5.1 shows that the programming languages more often used by the potential tool users on a daily basis are C# (29% of the users), JavaScript (29%), and Java (19%). With the exception of Swift, all programming languages reported by the tool users are top-seven most popular languages according to the TIOBE Index for December 2022[3]. Figure 5.2 suggests that 38% of the tool users are senior developers with more than three

[2] https://www.whatsapp.com/

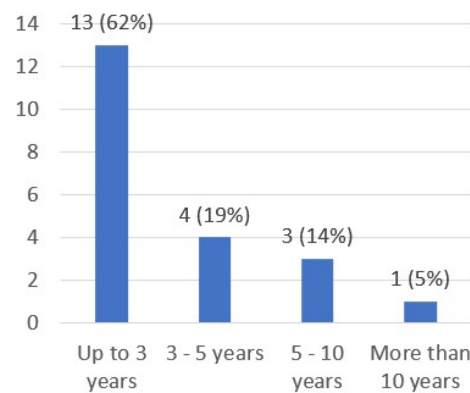
[3] https://www.tiobe.com/tiobe-index/

Figure 5.1: Most used programming language



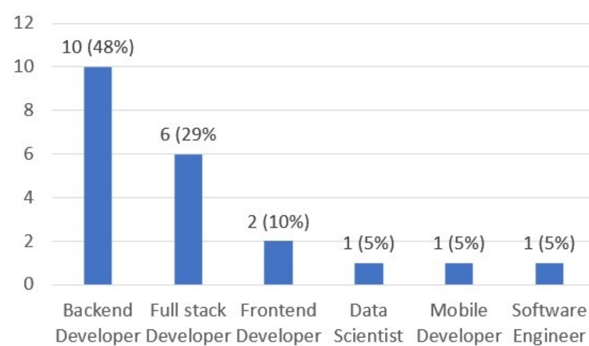
Source: Elaborated by the author.

Figure 5.2: Years of experience as a programmer



Source: Elaborated by the author.

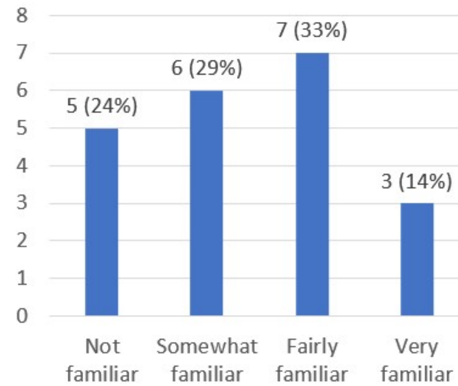
Figure 5.3: Current main role



Source: Elaborated by the author.

years of experience as a programmer, while most users (62%) are junior developers with up to three years of experience. Figure 5.3 shows that 76% of the tool users currently work as either back-end developers or full stack developers. Coincidentally, Figure 5.4 suggests that 76% of the tool users are at least minimally familiar with design patterns; 48% are either fairly or very familiar.

Figure 5.4: Familiarity with design patterns



Source: Elaborated by the author.

Table 5.3: Demographic information of each participant

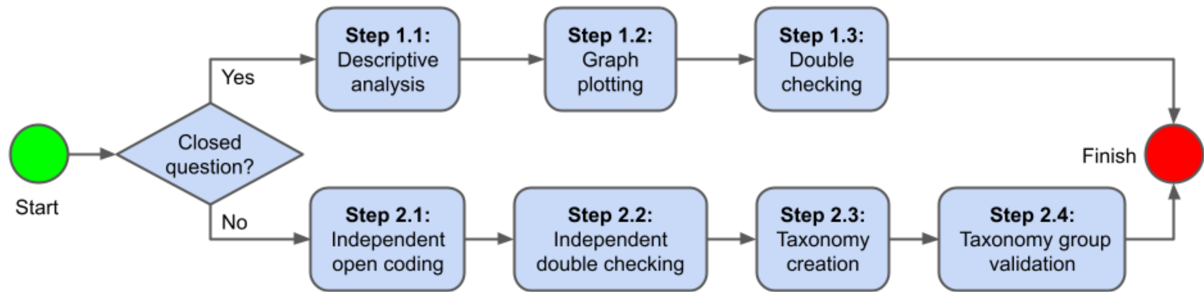
ID	Programming Language	Experience	Main Role	Familiarity
SD1	Java	More than 10 years	Full stack Developer	3
SD2	C#	Up to 3 years	Backend Developer	3
SD3	C#	Up to 3 years	Backend Developer	3
SD4	C#	Up to 3 years	Backend Developer	2
JD1	Java	5 - 10 years	Software Engineer	4
JD2	Java	Up to 3 years	Full stack Developer	2
JD3	Java	Up to 3 years	Full stack Developer	4
JD4	JavaScript	Up to 3 years	Frontend Developer	2
JD5	Python	Up to 3 years	Backend Developer	1
JD6	Python	Up to 3 years	Backend Developer	2
JD7	JavaScript	Up to 3 years	Frontend Developer	1
JD8	JavaScript	3 - 5 years	Full stack Developer	2
JD9	JavaScript	5 - 10 years	Backend Developer	3
JD10	C	5 - 10 years	Backend Developer	1
JD11	JavaScript	Up to 3 years	Full stack Developer	2
JD12	Swift	Up to 3 years	Mobile Developer	4
JD13	C#	3 - 5 years	Backend Developer	3
JD14	Python	Up to 3 years	Data Scientist	1
JD15	C#	Up to 3 years	Backend Developer	1
JD16	C#	3 - 5 years	Backend Developer	3
JD17	JavaScript	3 - 5 years	Full stack Developer	3

5.1.4 Data Analysis Procedures

Figure 5.5 depicts our data analysis procedures. We defined a total of seven steps, which range from quantitative to qualitative analysis of the survey data. We relied on strict guidelines for conducting empirical research in software engineering [17, 56, 69, 82]

to support the definition of our procedures.

Figure 5.5: Data analysis procedures for closed survey questions



Source: Elaborated by the author.

Only a few of our survey questions (U1 to U5) are closed questions, whose data are typically more straightforward to analyze. One researcher computed frequency of survey answers (e.g., programming languages reported on survey question U1) for the descriptive analysis in Step 1.1. The same researcher plotted the bar charts summarized in Figures 5.1, 5.2, 5.3 and 5.4 in Step 1.2. Three researchers contributed to performing double checking in Step 1.3.

The remaining survey questions are open questions and we opted for manual analysis to achieve accuracy in the data analysis. One researcher performed Step 2.1 to extract textual topics (i.e., codes) from each open question response via open coding [69]. Another researcher double checked the codes and suggested fixes in Step 2.2, thereby mitigating subjective biases. Both researchers discussed to reach consensus in case of disagreement. The two researchers created together in Step 2.3 all taxonomies based on the codes extracted for each survey question. We followed an iterative process as suggested by past work [17]. Three researchers helped validate the taxonomies created for each survey question in Step 2.4.

We provide additional comments regarding open coding (Step 2.1) and taxonomy creation (Step 2.3) as follows. Not all survey responses provided a valid code during the open coding; some survey responses provided more than one valid code. Thus, the number of codes grouped by taxonomy is not necessarily equal to the number of survey participants. We decided to shorten the taxonomies presented in this chapter for the sake of simplicity. Our study replication package [4] includes the full taxonomies.

[4] https://doi.org/10.5281/zenodo.7465098

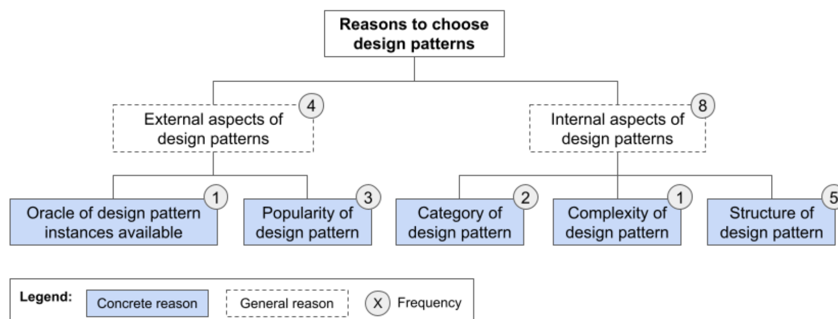
5.2 Results of the Survey with Tool Designers

This section reports the results of the survey with tool designers as follows. Section 5.2.1 presents the reasons why tool designers targeted specific design patterns. Section 5.2.2 discusses the reasons why tool designers targeted specific programming languages. Section 5.2.3 presents the contexts in which tool designers know that their tools have been used.

5.2.1 Reasons to Detect Specific Design Patterns

Survey question D1: Figure 5.6 summarizes the reasons reported by tool designers to target specific design patterns. We found five concrete reasons (blue rectangles with continuous borders), which we grouped into two general reasons (white rectangles with dashed borders): **External aspects of design patterns** and **Internal aspects of design patterns**. We annotated each reason with its frequency; such frequency corresponds to the number of participants (out of the nine tool designers) who mentioned a reason.

Figure 5.6: Taxonomy of reasons to target specific design patterns



Source: Elaborated by the author.

Table 5.4 describes the reason founds and illustrates each reason with quotes from the survey responses. Four responses refer to **External aspects of design patterns**, whose reasons include: **Oracle of design pattern instances available** (R1) and **Popularity of design pattern** (R2), which is the the most frequent one. The multiple mentions to R2 suggest that tool designers have tried to align their work with current industry trends. On the other hand, eight responses refer to **Internal aspects of design patterns**, whose reasons include: **Category of design pattern** (R3), **Complexity of design pattern** (R4), and **Structure of design pattern** (R5), which is the most

frequent one. We infer from the high frequency of mentions to R5 that tool designers prioritize the detection of design patterns whose internal structure is easier to characterize.

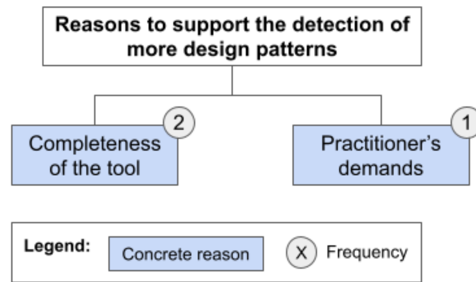
Table 5.4: Details of reasons to target specific design patterns

ID	Description	Sample quote
R1	Reference list of (typically validated) instances of design pattern instances detected on a software system	*"we selected patterns that were documented in the selected subject systems"* – Tool designer TD6
R2	Frequency in which a design pattern is discussed (by either researchers or practitioners) or implemented in industry	*"GoF patterns are well known and widely used in software design"* – Tool designer TD4
R3	Category assigned to a design pattern (e.g., based on its purpose of internal structure)	*"Because [the design patterns] are representative of the main categories/families of design patterns"* – Tool designer TD1
R4	Inherent difficulty in understanding the concept or implementation of a design pattern	*"[The design patterns] are of varying complexities"* – Tool designer TD1
R5	Internal composition of a design pattern implemented in the source code of a system	*"because [these design patterns] are very similar with regard to the structural properties (classes, relations between them)"* – Tool designer TD5

Survey question D2: We asked tool designers whether they would implement tools for detecting design patterns beyond the current scope of their DPD tools. Out of the nine tool designers surveyed, four responded "Yes", two responded "No", and three did not respond the question. Examples of design patterns mentioned by the tool designers who responded affirmatively include Flyweight, Interpreter, and Iterator. We discuss below the reasons reported by tool designers to target these specific design patterns in an hypothetical future.

Figure 5.7 summarizes the reasons reported by the tool designers on that matter. We found only two different reasons (blue rectangles with continuous borders). We labeled the first reason as **Completeness of the tool** (R1) by inferring that tool designers aim at completing the detection scope of the tool given a literature reference. R1 can be illustrated by the tool designer TD9 who reported that their *"original goal was to support all 23 [design patterns of the GoF's book]"*. Such interest may be due to the prestige that the mentioned book [37] has in the context of object-oriented programming. We labeled the second reason as **Practitioner's demands** (R2) as suggested by the interest of designer TD5 in implementing tools for detecting *"any pattern that is relevant to a developer"*.

Figure 5.7: Taxonomy of reasons to support the detection of more design patterns

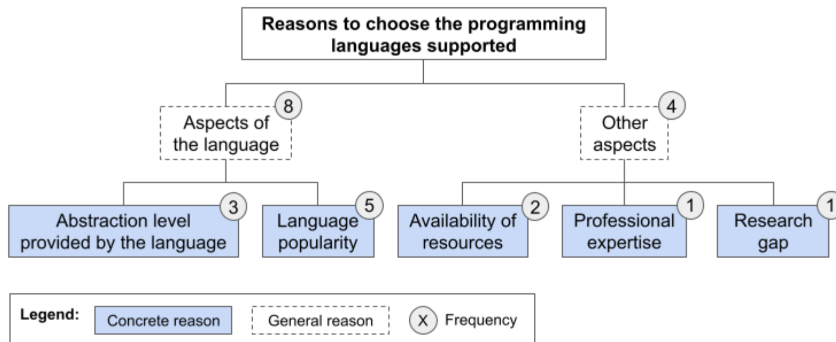


Source: Elaborated by the author.

5.2.2 Reasons to Support Specific Programming Languages

Survey question D3: Figure 5.8 depicts the reasons reported by tool designers to target specific programming languages. We found five concrete reasons (blue rectangles with continuous borders), which we grouped into two general reasons (white rectangles with dashed borders): **Aspects of the language** and **Other aspects**. We annotated each reason with its frequency, i.e., the number of participants (out of the nine tool designers) who mentioned a reason.

Figure 5.8: Taxonomy of reasons to target specific programming languages



Source: Elaborated by the author.

Table 5.5 describes the reasons found and illustrates each reason with quotes from the survey responses. Eight participants refer to **Aspects of the language**, whose two reasons are: **Abstraction level provided by the language** (R1) and **Language popularity** (R2). The high frequency of mentions to R2 suggests a certain effort from the tool designers' side to align their work with industry trends. Still, R1 plays an important role in deciding for performing DPD on a specific language. On the other hand, four participants refer to **Other aspects**, which include the following reasons: **Availability of resources** (R3), **Professional expertise** (R4), and **Research gap** (R5). These reasons suggest that convenience plays an essential role in the decision of tool designers.

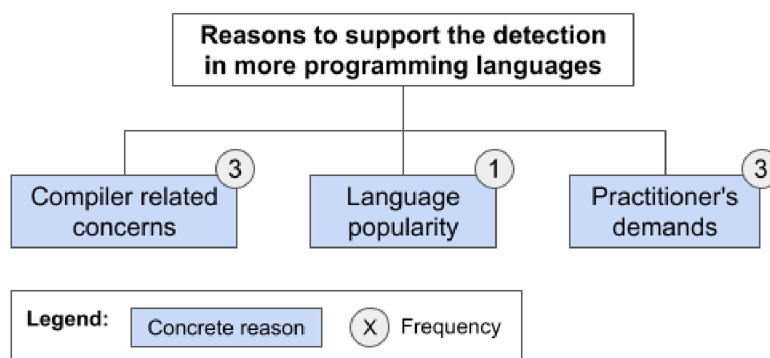
Table 5.5: Details of reasons to target specific programming languages

ID	Description	Sample quote
R1	Abstraction level provided by a programming language to support the implementation of software features	*"[This language] provided the right level of abstraction"* – Tool designer TD1
R2	Frequency in which a programming language is adopted by either researchers or software developers	*"[Language X] was the most popular object-oriented programming language at that time"* – Tool designer TD2
R3	Availability of technologies compatible with or designed for manipulating a programming language	*"because we had created [auxiliary tool name], the tool we used as backend for [our DPD tool]"* – Tool designer TD5
R4	Practical experience with the use of a programming language	*"Because of my experience with [Language X] use and [Language X] compilers"* – Tool designer TD5
R5	Research field yet to be addressed by scientific research	*"we were the first model to address this research gap"* – Tool designer TD4

Survey question D4: We asked tool designers whether they would implement tools for detecting design patterns on systems implemented in programming languages other than the languages already supported by their DPD tools. Out of the nine tool designers surveyed, seven responded "Yes", one responded "No", and one did not respond the question. Examples of programming languages mentioned by the tool designers who responded affirmatively are C, C++, C#, and Python. These four examples are among the five most popular languages (see TIOBE Index for December 2022).

Figure 5.9 summarizes the reasons reported by tool designers to target the aforementioned specific languages in an hypothetical future. We found three concrete reasons in total (blue rectangles with continuous borders). We labeled the first reason as **Compiler related concerns** (R1), which refers to the compilation environment available for a specific language. This reason is illustrated by the mention to *"any [...] language that has an open source, reliable, well documented and understandable compiler"* made by tool designer TD5. We labeled the second reason as **Language popularity** (R2), which can be illustrated by the mention of *"any widely used language"* also made by tool designer TD5. The third reason was labeled as **Practitioner's demands** (R3), which is illustrated by a mention of *"whatever language [that] is being used by practitioners"* from the response of tool designer TD1. R2 and R3 reinforce our previous assumption of designers being aware of industry trends.

Figure 5.9: Taxonomy of reasons to support the detection in more languages

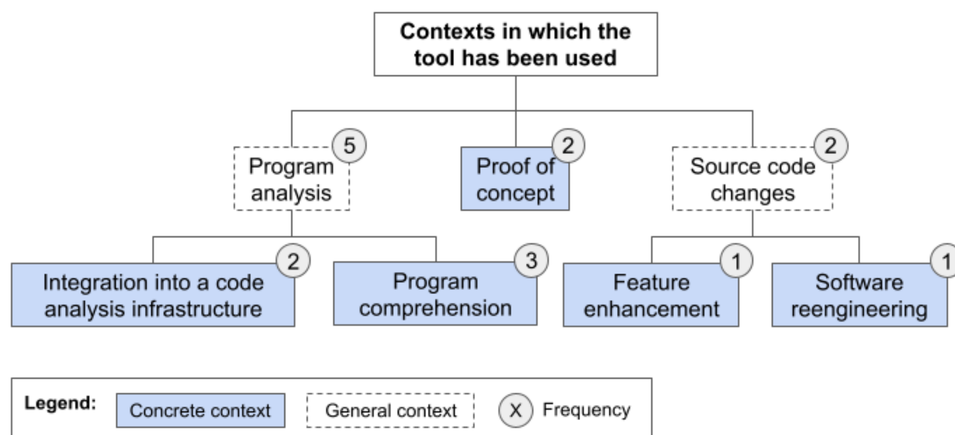


Source: Elaborated by the author.

5.2.3 Reported Contexts of Tool Usage

Survey question D5: Figure 5.10 summarizes the contexts reported by tool designers in which their tools have been used. We found five concrete contexts (blue rectangles with continuous borders). Four out of the five contexts were grouped into two general contexts (white rectangles with dashed borders): **Program analysis and Source code changes**. We annotated each context with its frequency; such frequency is the number of participants (out of the nine tool designers) who reported a context. We discuss below each context with sample quotes extracted from the survey responses.

Figure 5.10: Taxonomy of contexts in which tools have been used



Source: Elaborated by the author.

Table 5.6 describes and illustrates each context reported by tool designers. Five responses refer to **Program analysis**, whose contexts were labeled as: **Integration into a code analysis infrastructure** (C1) and **Program comprehension** (C2), the latter being the most frequent context. Two responses refer to **Source code changes**, and contexts include: **Feature enhancement** (C3) and **Software reengineering** (C4). C3

and C4 suggest that DPD tools have been used to assist developers in making design decisions on their systems. Finally, two responses refer to **Proof of concept** (C5).

Table 5.6: Details of contexts in which tools have been used

ID	Description	Sample quote
C1	Integration of the DPD tool into a larger code analysis infrastructure	*"[A company] wanted to implement a search engine [...] and the search engine would fetch design pattern instances already implemented in the company's codebase [based on the reports of our DPD tool]"* – Tool designer TD6
C2	Process of understanding the internal code structure and execution of a software system	*"[Our tool] is used to provide concrete information for developers and technical managers about the usage of design patterns in the implemented code"* – Tool designer TD4
C3	Structural or functional improvement of existing software features	*"[Our tool has been used for] enhancing an existing feature or code"* – Tool designer TD3
C4	Reconstruction of an entire software system aimed at improved quality	*"The tool has been used to re-engineer systems and applications"* – Tool designer TD2
C5	Prototype designed to demonstrate the feasibility of a concept	*"We wanted to demonstrate the capability [of our DPD tool to detect design patterns]"* – Tool designer TD1

5.3 Results of the Survey with Tool Users

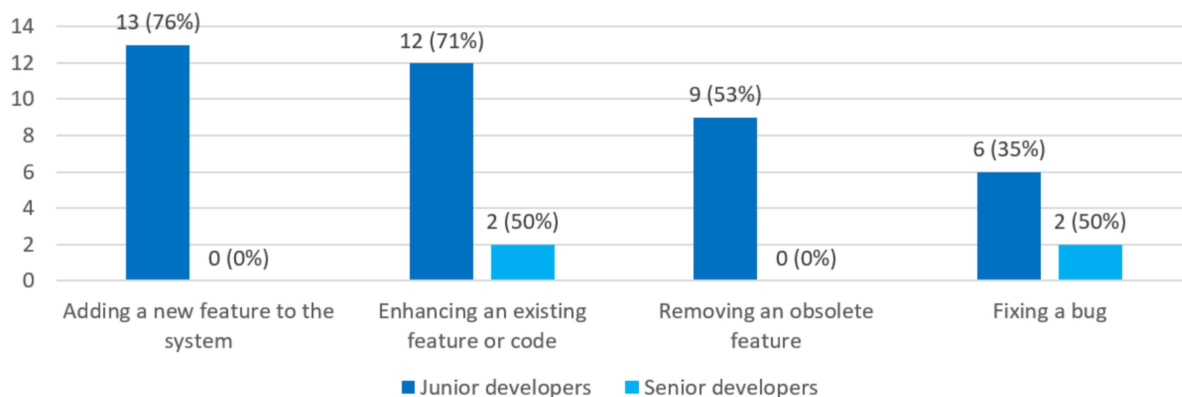
This section reports the results of our survey with tool users. Section 5.3.1 presents the contexts in which tool users would use a DPD tool. Section 5.3.2 discusses expected benefits of using a tool. Section 5.3.3 discusses barriers that would prevent tool users from using a tool. Lastly, Section 5.3.4 discusses why tool users believe that detecting certain design pattern is worthwhile.

5.3.1 Potential Tool Use Contexts

Survey question U5: Figure 5.11 summarizes the responses of tool users regarding contexts in which they would consider using a DPD tool. U5 is a closed question and each tool user could report more than one context. Tool users were also allowed to

mention additional contexts in the "Other" field, but none of our survey participants did that. Our 21 survey participants provided us with 44 responses in total. We discuss below our main observations.

Figure 5.11: Contexts in which developers would consider using a DPD tool



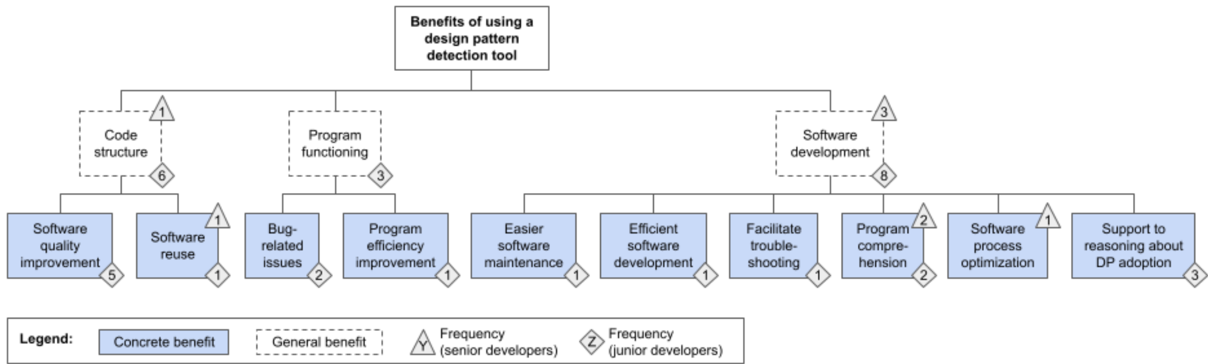
Source: Elaborated by the author.

Regarding responses of the 17 junior developers, more than 70% of the participants reported that DPD tools may be useful while either adding new software features (13 responses) or enhancing existing features (12 responses). Additionally, 53% of the participants mentioned that DPD tools could help remove obsolete features, which is aligned to the daily software development practices. Curiously, 35% of the participants mentioned bug fixing as a context in which using a DPD could be useful. These results are expected if we consider the common wisdom that well-structured code can facilitate feature addition or enhancement [32, 53] and reduce bug-proneness [54]. Senior developers also mentioned feature enhancement and bug fixing in their responses, which is in line with the opinion of junior developers.

5.3.2 Expected Benefits of Using a DPD Tool

Survey question U6: Figure 5.12 summarizes the benefits expected by the tool users from using a DPD tool. We found a total of ten concrete benefits (blue rectangles with continuous borders), which are grouped into three general benefits (white rectangles with dashed borders): **Code structure**, **Program functioning**, and **Software development**. We annotated the benefits with two different frequencies: triangles represent the frequency of senior developers who reported a specific benefit, while diamonds represent the frequency of junior developers who reported that benefit. We discuss below our major findings followed by quotes from the survey responses.

Figure 5.12: Expected benefits of using a DPD tool



Source: Elaborated by the author.

Table 5.7 describes and exemplifies with quotes each expected benefit. Quotes marked with an asterisk (*) refer to quotes that have been translated from Portuguese to English. Six junior developers and one senior developer mentioned **Code structure**, which is related to improving the internal source code structure. The concrete benefits in this category are: **Source code quality improvement** (B1) and **Software reuse** (B2). Additionally, three junior developers mentioned **Program functioning**, a category of benefits associated with program functioning improvements. The concrete benefits include: **Bug-related issues** (B3) and **Program efficiency improvement** (B4). Finally, eight junior developers and 3 senior developers mentioned benefits related to **Software development**: **Easier software maintenance** (B5), **Efficient software development** (B6), **Facilitate troubleshooting** (B7), **Program comprehension** (B8), **Software process optimization** (B9), and **Support to reasoning about DP adoption** (B10).

5.3.3 Barriers to the Use of DPD Tools

Survey question U7: Figure 5.13 summarizes the barriers mentioned by potential tool users as possible barriers to the adoption of DPD tools. We found a total of ten concrete barriers (blue rectangles with continuous borders), which we grouped into three general barriers: **Development limitations**, **Tool limitations**, and **Tool usage limitations**. We annotated the barriers with two different frequencies: triangles represent the frequency of senior developers who reported a specific barrier, while diamonds represent the frequency of junior developers who reported that barrier. We discuss below our main findings.

Table 5.8 provides descriptions and sample quotes for each barrier. Three junior

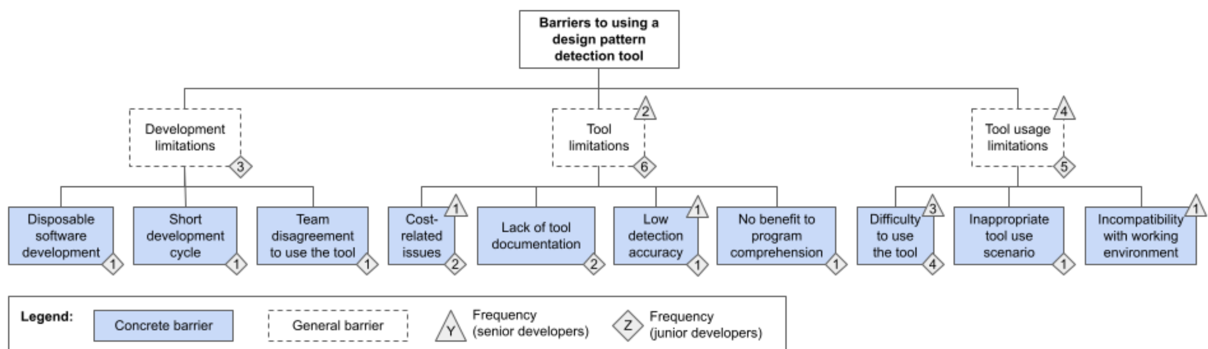
Table 5.7: Details of the expected benefits of using a DPD tool

ID	Description	Sample quote
B1	Internal quality improvement, e.g., via source code cleaning	*"[DPD tools may help] making the code base cleaner for new features"* – Junior developer JD11
B2	Knowledge or source code reuse	*"[Using a DPD tool may help the] development of similar applications"** – Senior developer SD4
B3	Dealing with software bugs, e.g., via prevention or fixing	*"[Using a DPD tool may help] prevent bugs"* – Junior developer JD13
B4	Improved efficiency in terms of program execution time	*"[Using a DPD tool may lead to] increased code velocity"* – Junior developer JD15
B5	Facilitation of software maintenance tasks	*"A benefit would be an] easy-to-maintain source code that other people can use"** – Junior developer JD3
B6	Improved efficiency in terms of time spent to develop software	*"[Using a DPD tool] saves time for the developer"* – Junior developer JD4
B7	Facilitation of the monitoring and fixing of failures, faults or defects	*"[Using a DPD tool allows for] better and more precise troubleshooting"* – Junior developer JD6
B8	Process of understanding the internal structure and execution of a system	*"[Using a DPD tool] saves time for code understanding"* – Junior developer JD1
B9	Adjust and improvement of the software development processes	*"[Using a DPD tool may assist] process optimization"** – Senior developer SD2
B10	Process of reasoning about benefits, drawbacks, and challenges in adopting design patterns	*"[Using a DPD tool may help to] identify [...] which design pattern to use or if the current one used is the most appropriate"* – Junior developer JD4

developers mentioned a general barrier we labeled as **Development limitations**, which refers to barriers faced over the life cycle of a software system. The respective concrete barriers are: **Disposable software development** (B1), **Short development cycle** (B2), and **Team disagreement to use the tool** (B3). Six junior developers and 2 senior developers mentioned **Tool limitations** as a general barrier associated with either technical or functional limitations of the DPD tool. The respective concrete barriers include: **Cost-related issues** (B4), **Lack of tool documentation** (B5), **Low detection accuracy** (B6), and **No benefit to program comprehension** (B7). The occurrences of B5 and B6 are red flags because, as our SLR and comparative study suggests (Chapters 3 and 4), i) existing DPD tools are poorly documented and ii) their accuracy is often low.

Finally, five junior developers and four senior developers mentioned **Tool usage limitations** as a general barrier. This barrier refers to limitations that emerge while using a DPD tool, and it includes: **Difficulty to use the tool** (B8), **Inappropriate**

Figure 5.13: Taxonomy of barriers to the use of DPD tools



Source: Elaborated by the author.

Table 5.8: Details of barriers to the use of DPD tools

ID	Description	Sample quote
B1	Development of disposable software, e.g., proof of concept or prototype	*"[I would not use a DPD tool in] a [software] project that [...] is disposable, e.g., a Minimum Viable Product (MVP)"** – Junior developer JD3
B2	Tight time to deliver software products	*"[I would not use a DPD tool in] a [software] project that must be quickly completed [...], e.g., a Minimum Viable Product (MVP)"** – Junior developer JD3
B3	Conflicting perceptions on the usefulness of a tool	*"[Using a DPD would depend on the] agreement of the people involved in the [software] project"** – Junior developer JD12
B4	Costs derived from the tool adoption	*"[I would not use DPD] tools that are heavy [to run]"** – Senior developer SD2
B5	Poorly documented tool	*"[I would not use a DPD tool due to] lack of documentation"** – Junior developer JD14
B6	Low precision and recall in the detection of design pattern instances	*"[I would not use DPD tools in case I was] not sure if the tools are good (i.e., high precision) for design pattern detection"* – Junior developer JD1
B7	Little or no perceived support to understanding the internal structure or execution of a software system	*"If this tool prevents me from learning somehow, then I would not use it"* – Junior developer JD11
B8	Inherent difficulty of adopting a tool	*"[I would not use a DPD tool due to its] learning curve"* – Junior developer JD16
B9	Development scenario in which using the tool is inadequate or unnecessary	*"not being the proper scenario [would prevent me from using a DPD tool]"* – Junior developer JD13
B10	Working environment does not accommodate the use of a DPD tool	*"incompatibility [of using a tool] with [the] working environment [would prevent me from using a DPD tool"* – Senior developer SD3

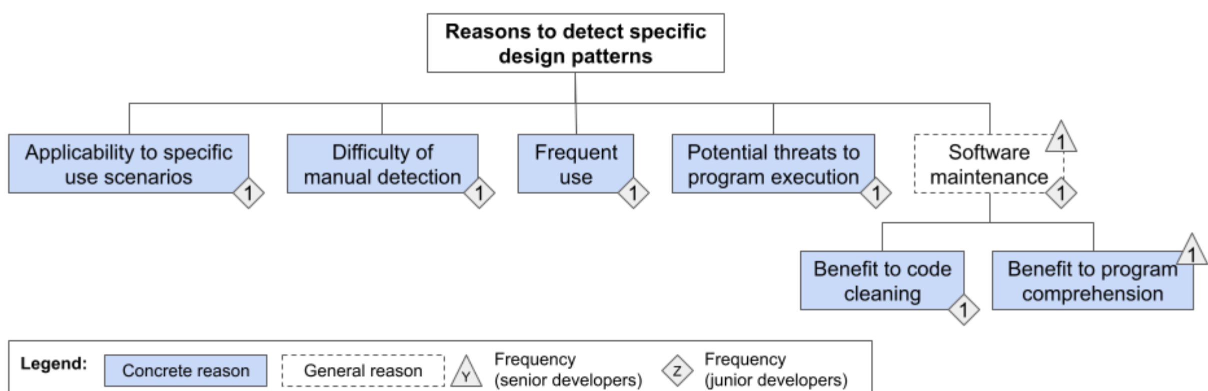
tool use scenario (B9), and **Incompatibility with working environment** (B10). The high concentration of mentions to B8 must be stressed because, in Chapter 4.1.2, we report our own difficulty with configuring and running existing DPD tools. Such difficulty should be definitely addressed by tool designers to promote the adoption of their tools.

5.3.4 Design Patterns Worth Detecting

Survey question U8: We asked potential tool users to list design patterns they considering to be worth detecting via a DPD tool. Out of the nine survey participants, only five participants reported one or more specific design patterns. Contrary to our expectation, all design patterns pointed by the tool users are summarized in the GoF's book [37]: Observer was mentioned by three tool users; Composite, Decorator, Singleton, Strategy, and Visitor were mentioned by two tool users each. This result surprised us because we expected at least one mention of specific design patterns proposed for trending/more modern programming languages, e.g., JavaScript and Swift – which are among the most used by our survey participants (Figure 5.1).

Figure 5.14 summarizes the reasons that potential tool users mentioned in their responses for automating the detection (via DPD tool) of specific design patterns. We found a total of six concrete reasons, which we depict with blue rectangles with continuous borders. The white rectangle with dashed borders represents a general reason. We annotated the reasons with two different frequencies: triangles represent the frequency of senior developers who reported a specific reason, while diamonds represent the frequency of junior developers who reported that reason. We discuss below our major findings.

Figure 5.14: Taxonomy of reasons why detecting specific patterns is worthwhile



Source: Elaborated by the author.

Table 5.9 describes and exemplifies with quotes each reported reason. Four concrete reasons address different concerns and, therefore, were not grouped into general reasons: **Applicability to specific use scenarios** (R1), **Difficulty of manual detection** (R2), **Frequent use** (R3), and **Potential threats to program execution** (R4). Only one survey participant reported each of these reasons. Additionally, **Software maintenance** is the only general reason and it was mentioned by one junior developers and one senior developer. This general reason covers two concrete reasons: **Benefit to code cleaning** (R5) and **Benefit to program comprehension** (R6).

Table 5.9: Details of reasons why detecting specific patterns is worthwhile

ID	Description	Sample quote
R1	Implementing a design pattern fits a specific development context	*"if it's meaningful in the situation and the context that I am [working on], then it's worth [detecting a design pattern through a tool]"* – Junior developer JD11
R2	Manually detecting a design pattern is challenging if not unfeasible	*"[These design patterns] are useful but may be hard to [manually] identify"* – Senior developer SD3
R3	A design pattern is frequently implemented in a given system	*"because I use [these design patterns] more often"** – Junior developer JD3
R4	Implementing a design pattern may harm the program execution	*"[This design pattern] may cause problems in larger programs due to the use of threads"** – Junior developer JD12
R5	Implementing a design pattern favors code cleaning	*"[These design patterns] contribute to a cleaner code"** – Junior developer JD3
R6	Implementing a design pattern helps understand the code structure and software execution	*"[This design pattern] would facilitate the comprehension of other developers' code"** – Senior developer SD2

5.4 Study Implications

Implication 1: *Technical support is required to expand the coverage of existing DPD tools* – Our first survey revealed that tool designers strive to propose DPD tools that can be perceived as useful in industry settings. Mentions to covering multiple categories of design patterns (Figure 5.6), as well as the focus on popular programming languages (Figure 5.8), support this claim. Still, survey responses also suggest an opportunistic choice for design patterns to detect based on technical limitations. Examples of limitations are the inherent complexity of detecting certain patterns and the availability of technologies to manipulate source code implemented in a given language. It could be that

such lack of technical support is preventing tool designers from covering more patterns and languages with their tools. Thus, we encourage that researchers and practitioners join forces to address these technical limitations in future work.

Implication 2: *Detection accuracy must be a priority while proposing DPD tools, not the variety of detectable design patterns* – Many responses to our first survey (Figure 5.6) explicitly mention the GoF's book [37] as a reference to design patterns implementation. Thus, it was expected that tool designers would justify their interest in completing their tools to detect as many GoF patterns as possible (Figure 5.7). However, responses to the second survey stress that DPD is worthwhile, e.g., when a specific design pattern is frequently implemented in their systems and difficult to manually detect (Figure 5.14). Moreover, potential tool users mentioned that low detection accuracy may prevent them from using a tool (Figure 5.13). This result is quite interesting because, as the comparative study presented in Chapter 4 revealed, the accuracy of existing DPD tools is often low or insufficient. We advocate that detection accuracy should be a priority, not the variety of detectable design patterns, in the proposal of novel tools.

Implication 3: *There may be a demand for DPD tools compatible with popular programming languages to be addressed* – Our first survey revealed that tool designers have interest in supporting DPD on systems implemented in very popular programming languages such as C# and Python (Section 5.2.2). Curiously, the literature review presented in Chapter 3 catalogs only three tools compatible with at least one of these languages, e.g., Philippow et al. [55] for C++ systems. More critically, none of these tools were publicly available for download during the execution of Chapter 4's comparative study. Our results in this chapter suggest that there may be an unmet demand for tools targeting popular languages. We advocate for the proposal of novel tools aimed at addressing such demand in industry.

Implication 4: *The proposal of DPD tools should consider that DPD is only part of a broader software development life cycle* – Our first survey revealed that tool designers are aware of their tools being used in program comprehension and integrated with code analysis infrastructures (Figure 5.10). This finding meets the users' perceptions of tools as facilitators of program comprehension (Figure 5.12). However, our second survey reveals other expected benefits of using a tool, e.g., facilitate bug fixes and the addition or enhancement of software features (Table 5.7). This finding suggests that DPD tools can be used with more complex intents in mind when compared to the pure program comprehension. We encourage tool designers to propose their tools with this observation in mind, so that DPD tools could be integrated with other tools, e.g., for bug fixing or feature enhancement. This could improve the expected usefulness of DPD tools from the viewpoint of tool users.

Implication 5: *Documentation of DPD tools should be significantly improved for the sake of tool usefulness* – Our second survey revealed that the difficulty to use the

tool and its incompatibility with the working environment may prevent tool users from adopting a DPD tool (Figure 5.13). This observation is critical if we consider that we as experts had difficulty to configure and run some of the tools empirically assessed in Chapter 4. Additionally, potential tool users mentioned that the lack of tool documentation plays an important role in discarding a DPD tool. We also reported in our recent work that existing DPD tools are poorly documented, which makes the experience of using a DPD tool even more negative. Thus, we encourage tool designers to be more attentive to the documentation of their DPD tools.

5.5 Threats to Validity

We discuss below threats to the validity of our study. For this purpose, we rely on strict guidelines for experimentation in software engineering [82].

Construct validity is associated with the study planning and, especially, the preparation of artifacts necessary to conduct the study. We carefully defined our study goal and RQs (Section 5.1.1) based on multiple iterations of meetings with three researchers. We then expected to avoid defining a weak goal, one that could not lead to insightful findings on the design and usefulness of DPD tools. Additionally, we relied on strict literature guidelines for conducting online surveys in software engineering [56]. We did our best to propose short and simple questions, thereby avoiding misunderstanding in responding the surveys. We polished our survey structure (Section 5.1.2) in multiple discussion sessions with three researchers. None of the survey questions has changed once we forwarded the surveys to participants.

Internal validity is related to the execution of our experimental procedures. To recruit as many participants as possible to our first survey (Section 5.2), we used the full list of authors of the 42 DPD tools found in our literature review from Chapter 3. Thus, we expected to avoid selection biases regarding the participation of DPD tool designers with specific viewpoints. We waited two months for responses so that tool designers could comfortably answer all survey questions. Regarding our second survey (Section 5.3), we opted to recruit tool users via broadcasting on *WhatsApp* groups, which are popular in Brazil, and one Software Engineering class. We waited one month for responses, which was long enough at least for the junior developers who are potential tool users.

Conclusion validity is associated with the data analysis procedures. As discussed in Section 5.1.4, we relied on strict guidelines to analyze our open survey questions [17, 69]. Combined with our previous experience in analyzing quantitative and qualitative data [49, 50], the literature support was essential to guide the survey data analysis. Two researchers

had multiple meeting sessions to discuss disagreement and reach consensus. Thus, we expected to reduce human biases in both code extraction and taxonomy creation. For the closed questions, we simply computed descriptive statistics [82]. Another threat is related to poor translation of some of the answers done by the researchers (from Portuguese to English), and also possible incorrect use of the English language by the participants since they are not native English speakers. We carefully translated and double-checked all of the answers and translations where applicable while also taking the context of the answer into consideration where applicable.

External validity is related to the potential generalization of our study findings. Regarding our first survey, the tool designers of only nine out of the 42 DPD tools (21.4%) responded to the survey (Section 5.1.3). Such response rate is reasonable if we consider that: 26 tools were published more than ten years ago; several email addresses were invalid; and some tool designers may have left academia, thereby losing interest in responding surveys on their past academic work. Still, we acknowledge that such rate may prevent us from generalizing our findings to all DPD tool designers.

Regarding the second survey, we managed to recruit 17 junior developers, but only a few senior developers responded to our request via WhatsApp. It is worth mentioning that, prior to recruiting survey participants via WhatsApp, we tried to email active contributors of GitHub[5] software projects. We emailed a total of 200 contributors and waited 2 months for survey responses. Despite our effort, none of the GitHub contributors participated, which frustrated our goal of hearing them to assist the proposal of useful DPD tools. We expect that other researchers can replicate our survey with a larger number of participants for a richer, more comprehensive empirical knowledge.

5.6 Chapter Summary

In this chapter, we presented an empirical study aimed at capturing the perceptions of tool designers and potential tool users on DPD tools. We carefully conducted two online surveys based on strict literature guidelines [56]. Our first survey (Section 5.2) targeted the designers of 42 DPD tools. It revealed different aspects of the rationale behind the proposal of DPD tools to detect specific design patterns in specific programming languages. Our second survey (Section 5.3) targeted both junior and senior developers who are potential tool users. We computed descriptive statistics and relied on strict guidelines [17, 69] for performing open coding and the generation of taxonomies.

[5] https://github.com/

The survey results are complementary: both tool designers and tool users perceived DPD tools as useful to program comprehension and assistance to software changes. Despite the best effort of tool designers, the existing DPD tools have some critical limitations, e.g., lack of documentation and low detection accuracy, that tool users see as barriers to the tool adoption (Section 5.3.3).

In the next chapter, we conclude this work by presenting the final considerations, main contributions, study implications and future work.

Chapter 6

Conclusion

Design patterns are reusable solutions to common and recurring problems of software design [37]. Understanding which and how design patterns occur in a system may assist developers during maintenance and evolution activities [45]. However, detecting design patterns is an important but challenging task in software development due to several reasons, such as high complexity and the size of real-world systems. Thus, several tools [19, 26, 34, 64, 78] have been developed throughout the years to automate the design pattern detection process. This dissertation presented three complementary studies aimed to understand which are these design pattern detection tools (DPD), how they perform when detecting design patterns, why these tools were developed for their specific contexts, and how potential users perceive them.

In this chapter, we summarize our studies results in Section 6.1. We then present our main contributions in Section 6.2. Section 6.3 presents the multiple implications of each study presented. Lastly, Section 6.4 discusses future work.

6.1 Work Overview

The first study presented in this work (Chapter 3 was the systematic literature review (SLR). Its purpose was to provide both researchers and practitioners a catalog of the DPD tools published in the last 20 years presenting their main features, such as availability for download, scope of design pattern detection, supported programming languages, etc. This catalog might help developers when choosing the DPD tool that best suits their context. Our SLR revealed 42 DPD tools published in the last 20 years. However, only 10 of them were available for download during our investigation. Altogether, the tools claim to detect all 23 design patterns compiled by the Gang of Four's book [37]. The frequency of published tools remained stable over the years, thereby suggesting this research topic has not yet reached saturation.

From the 42 DPD tools found, 68% are stand-alone. In addition, 69% of the tools are compatible with Java systems. However, only 19% of the tools have some companion documentation. Our results also showed that 69% are designed to perform DPD on Java systems, support the detection of complex design patterns, such as Composite (67%) and Observer (64%), and provide users with some sort of GUI other than command line (50%). On the other hand, only a few tools (29%) are explicitly free to use and downloadable (24%). Regarding the program analysis approaches, most of the tools (76%) rely on static analysis to support DPD. Additionally, 76% of the tools rely on a single modeling approach, mostly depending on the information of source code and its structure.

The second study was the comparison of four DPD tools in terms of precision, recall, and agreement (Chapter 4. This information complemented the technical data provided by our SLR when choosing a DPD tool, as the performance of the tool is also a very important metric. Our results suggested that each tool could be applied to detecting a particular subset of design patterns. FINDER showed sufficient accuracy for Composite, Decorator, Singleton, and Visitor. Still, future work is required to further support practitioners interested in automating DPD in practice. For the six design patterns evaluated, the four DPD tools tend to disagree on the instances detected, meaning that their output is not redundant. In other words, they detect distinct design pattern instances. Documentation on how to install, execute, and use the tools is often scarce, which may compromise the adoption of tools in practical settings.

Lastly, our third study presented two survey studies aimed at understanding the design rationale behind the proposal of the tools and the expected usefulness of DPD tools from the point of view of practitioners. We believe that the empirical results may enable future tool designers to propose tools that are more aligned with the needs of the practitioners and also instigate solutions for the technical difficulties that the designers may encounter when developing their approaches.

Among our results, we found five different reasons for targeting specific design patterns in the proposal of DPD tools. Most tool designers reported that internal aspects of a design pattern (e.g., the way it is structured in the source code) favored its detection. Additionally, popularity and abstraction level are key reasons to decide for detecting design patterns in specific programming languages. Tool designers showed interest in proposing tools for a wider scope of design patterns and programming languages. However, this would depend on the availability of enabling technologies (e.g., friendly compilers) besides the practitioner's demands. This finding could inspire engineers in filling gaps in terms of these technologies. Tool designers have knowledge on their DPD tools being used for program analysis tasks, especially program comprehension. This finding meets the expected benefits of using DPD tools reported by the potential tool users.

According to the tool users, other benefits include software quality improvement (e.g., via code organization and cleaning) and more. Potential tool users claim that tool

limitations (e.g., low accuracy and lack of documentation), as well as the inherent difficulty to use the tool, are key barriers to the DPD tool adoption. This finding is aligned to the barriers to adopting other types of automated tools discussed in previous work [2, 44].

6.2 Main Contributions

Our main contributions from this work include:

1. an extensive catalog of 42 DPD tools published in the last two decades, each characterized by the supported design problems, programming languages, etc.;
2. a quantitative comparison of four state-of-the-art tools based on widely used measures, i.e. precision, recall, F-measure, and agreement;
3. a list with manually validated instances of six design patterns detected from two Java software systems often investigated in studies on DPD tools [26, 21, 57], i.e. JHotDraw and JRefactory; and
4. replication packages for studies 1 and 2 [1], and study 3 [2];
5. empirical evidence about the design rationale behind the proposal of the tools;
6. empirical evidence regarding the expected usefulness from the point of view of practitioners;

Contribution 1 is our solution for *Research Problem 1.* By providing a catalog with the DPD tools published in the last two decades with their respective attributes, both researchers and practitioners can make use of it in order to find a tool that suits their needs or to address gaps in the literature.

Contribution 2 addresses the *Research Problem 2.* It provides information related to the performance of the tools in terms of precision, recall, F-measure and agreement which can also be used by practitioners in order to choose one of the available tools. Meanwhile, researchers may use this information as evidence that more precise approaches are needed.

Contributions 3 and 4 can be used for extending this study in the future. By providing the manually validated instances of design patterns, researchers can make use of this data set for benchmarking their tools for example. Furthermore, the first replication

[1] https://doi.org/10.5281/zenodo.5553470
[2] https://doi.org/10.5281/zenodo.7465098

package, other than allowing replicability itself, also enables the extension of our study for tools that will be developed and also for other design patterns or systems.

Contribution 5 provides the lacking information introduced in *Research Problem 3*, which is the output of the first survey of Study 3. Similarly, contribution 6 provides the lacking information introduced in *Research Problem 3*, which was obtained from the second survey of Study 3. Moreover, the second replication package described in contribution 4 enables the extension of our study by aggregating new responses from both tool designers and potential tool users.

6.3 Study Implications

The results presented in this work 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, the existing tools provide very limited support and considerably unreliable output. On the 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 the existing ones.

Furthermore, with the survey study we noticed that technical support is required to expand the coverage of existing DPD tools. Although tool designers strive to propose DPD tools that can be perceived as useful in industry settings, survey responses also suggest an opportunistic choice for design patterns to detect based on technical limitations. It could be that such lack of technical support is preventing tool designers from covering more patterns and languages with their tools.

Lastly, Documentation of DPD tools should be significantly improved for the sake of tool usefulness. Potential tool users mentioned that the lack of tool documentation plays an important role in discarding a DPD tool. We also reported in Study 1 that existing DPD tools are poorly documented, which makes the experience of using a DPD tool even more negative. Thus, we encourage tool designers to be more attentive to the documentation of their DPD tools.

6.4 Future Work

As future work, we propose to extend our comparison study protocol to more design patterns giving a more complete landscape of their detection prowess. By obtaining the performance metrics for other design patterns, we would be able to provide more precise recommendations on which DPD tool performs better when detecting each design pattern. We would also like to propose extending our comparison protocol to more tools. Since we successfully managed to contact other tool designers during our survey study, we believe it is also possible to obtain tools that were unavailable at the time of our study. We also want to extend the results of our survey with potential tool users by evaluating the correlation between each demographic attribute extracted and the taxonomies obtained.

Moreover, another topic to be explored as future work would be to conduct an experiment to evaluate the perception of users when utilizing DPD tools. Our second survey investigated the possible advantages and drawbacks of using DPD tools according to the participants. However, the information we obtained was more abstract and higher level. We believe that displaying a concrete tool during a practical session may provide more meaningful insights into what functionality the tool should provide. Furthermore, we would conduct interviews instead of surveys in order to obtain more concrete responses.

Bibliography

[1] Emad Aghajani, Csaba Nagy, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, Michele Lanza, and David C Shepherd. Software documentation: The practitioners' perspective. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*, pages 590–601, 2020.

[2] Khalid Alkharabsheh, Yania Crespo, Esperanza Manso, and José Taboada. Software design smell detection: A systematic mapping study. *Software Quality Journal (SQJ)*, 27(3):1069–1148, 2019.

[3] Awny Alnusair and Tian Zhao. Towards a model-driven approach for reverse engineering design patterns. In *Proceedings of the 2nd Workshop on Transforming and Weaving Ontologies in Model Driven Engineering (TWOMDE), co-located with the 12nd International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, pages 1–15, 2009.

[4] Apostolos Ampatzoglou, Georgia Frantzeskou, and Ioannis Stamelos. A methodology to assess the impact of design patterns on software quality. *Information and Software Technology (IST)*, 54(4):331–346, 2012.

[5] Giuliano Antoniol, Gerardo Casazza, Massimiliano Di Penta, and Roberto Fiutem. Object-oriented design patterns recovery. *Journal of Systems and Software (JSS)*, 59(2):181–196, 2001.

[6] Francesca Arcelli, Davide Franzosi, and Claudia Raibulet. .NET reverse engineering with MARPLE. In *Proceedings of the 5th International Conference on Software Engineering Advances (ICSEA)*, pages 227–231, 2010.

[7] Francesca Arcelli, Fabrizio Perin, Claudia Raibulet, and Stefano Ravani. JADEPT: Dynamic analysis for behavioral design pattern detection. In *Proceedings of the 4th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, pages 95–106, 2009.

[8] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Pearson, 1st edition, 1999.

[9] Thoms Ball. The concept of dynamic analysis. *ACM SIGSOFT Software Engineering Notes*, 24(6):216–234, 1999.

[10] Rafael Barbudo, Aurora Ramírez, Francisco Servant, and José Raúl Romero. GEML: A grammar-based evolutionary machine learning approach for design-pattern detection. *Journal of Systems and Software (JSS)*, 175:110919, 2021.

[11] Victor Basili and Dieter Rombach. The TAME project: Towards improvement-oriented software environments. *IEEE Transactions on Software Engineering (TSE)*, 14(6):758–773, 1988.

[12] Mario Luca Bernardi, Marta Cimitile, and Giuseppe Di Lucca. Design pattern detection using a DSL-driven graph matching approach. *Journal of Software: Evolution and Process (S:E&P)*, 26(12):1233–1266, 2014.

[13] Dirk Beyer, Andreas Noack, and Claus Lewerentz. Simple and efficient relational querying of software structures. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE)*, pages 216–225, 2003.

[14] Alexander Binun and Günter Kniesel. DPJF: Design pattern detection with high accuracy. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 245–254, 2012.

[15] Alex Blewitt, Alan Bundy, and Ian Stark. Automatic verification of Java design patterns. In *Proceedings of the 16th International Conference on Automated Software Engineering (ASE)*, pages 324–327, 2001.

[16] Daniel Cruz, Eduardo Figueiredo, and Jabier Martinez. A literature review and comparison of three feature location techniques using argouml-spl. In *13th VaMos*, pages 1–10, 2019.

[17] Daniela Cruzes and Tore Dyba. Recommended steps for thematic synthesis in software engineering. In *Proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 275–284, 2011.

[18] Haneen Dabain, Ayesha Manzer, and Vassilios Tzerpos. Design pattern detection using FINDER. In *Proceedings of the 30th Symposium on Applied Computing (SAC)*, pages 1586–1593, 2015.

[19] Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. An Eclipse plug-in for the detection of design pattern instances through static and dynamic analysis. In *Proceedings of the 26th International Conference on Software Maintenance (ICSM)*, pages 1–6, 2010.

[20] Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. Improving behavioral design pattern detection through model checking. In *Proceedings of the*

14th European Conference on Software Maintenance and Reengineering (CSMR), pages 176–185, 2010.

[21] Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. Detecting the behavior of design patterns through model checking and dynamic analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 26(4):1–41, 2018.

[22] David Devecsery, Peter Chen, Jason Flinn, and Satish Narayanasamy. Optimistic hybrid analysis: Accelerating dynamic analysis through predicated static analysis. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 348–362, 2018.

[23] Jens Dietrich and Chris Elgar. Towards a web of patterns. *Journal of Web Semantics (JoWS)*, 5(2):108–116, 2007.

[24] Michal Dobiš and L'ubomír Majtás. Mining design patterns from existing projects using static and run-time analysis. In *Proceedings of the 3rd Central and East European Conference on Software Engineering Techniques (CEE-SET)*, pages 62–75, 2008.

[25] Jing Dong, Dushyant Lad, and Yajing Zhao. DP-Miner: Design pattern discovery using matrix. In *Proceedings of the 14th International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS)*, pages 371–380, 2007.

[26] Jing Dong, Yongtao Sun, and Yajing Zhao. Design pattern detection by template matching. In *Proceedings of the 23rd Symposium on Applied Computing (SAC)*, pages 765–769, 2008.

[27] Jing Dong, Yajing Zhao, and Tu Peng. A review of design pattern mining techniques. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 19(6):823–855, 2009.

[28] Adriano Lages dos Santos, Maurício RA Souza, Marcela Dayrell, and Eduardo Figueiredo. A systematic mapping study on game elements and serious games for learning programming. In *10th CSEDU*, pages 328–356, 2018.

[29] Tore Dyba, Torgeir Dingsoyr, and Geir Hanssen. Applying systematic reviews to diverse study types: An experience report. In *Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 225–234, 2007.

[30] Maged Elaasar, Lionel Briand, and Yvan Labiche. VPML: An approach to detect design patterns of MOF-based modeling languages. *Software and Systems Modeling (SoSyM)*, 14(2):735–764, 2015.

[31] Félix Agustín Espinoza, Gustavo Esquer, and Joel Cansino. Automatic design patterns identification of C++ programs. In *Proceedings of the 1st Eurasian Conference on Information and Communication Technology (EurAsia ICT)*, pages 816–823, 2002.

[32] Eduardo Fernandes. *On the relation between refactoring and critical internal attributes when evolving software features*. PhD thesis, Informatics Department, Pontifical Catholic University of Rio de Janeiro, 2021. DOI: https://doi.org/10.17771/PUCRio.acad.53129.

[33] Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, and Eduardo Figueiredo. A review-based comparative study of bad smell detection tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 18:1–18:12, 2016.

[34] Francesca Fontana, Andrea Caracciolo, and Marco Zanoni. DPB: A benchmark for design pattern detection tools. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (ICSMR)*, pages 235–244, 2012.

[35] Francesca Arcelli Fontana and Marco Zanoni. A tool for design pattern detection and software architecture reconstruction. *Information sciences*, 181(7):1306–1324, 2011.

[36] Andrew Forward and Timothy Lethbridge. The relevance of software documentation, tools and technologies: A survey. In *Proceedings of the 2nd Symposium on Document Engineering (DocEng)*, pages 26–33, 2002.

[37] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Professional, 1st edition, 1995.

[38] Kilem Gwet. *Handbook of inter-rater reliability: The definitive guide to measuring the extent of agreement among raters*. Advanced Analytics, 4th edition, 2014.

[39] Dirk Heuzeroth, Thomas Holl, Gustav Hogstrom, and Welf Lowe. Automatic design pattern detection. In *Proceedings of the 11th International Workshop on Program Comprehension (IWPC)*, pages 94–103, 2003.

[40] Dirk Heuzeroth, Stefan Mandel, and Welf Lowe. Generating design pattern detectors from pattern specifications. In *Proceedings of the 18th International Conference on Automated Software Engineering (ASE)*, pages 245–248, 2003.

[41] Siw Hove and Bente Anda. Experiences from conducting semi-structured interviews in empirical software engineering research. In *Proceedings of the 11th International Software Metrics Symposium (METRICS)*, pages 1–10, 2005.

[42] Heyuan Huang, Shensheng Zhang, Jian Cao, and Yonghong Duan. A practical pattern recovery approach based on both structural and behavioral analysis. *Journal of Systems and Software (JSS)*, 75(1-2):69–87, 2005.

[43] Bernard Jansen. The graphical user interface. *ACM SIGCHI Bulletin*, 30(2):22–26, 1998.

[44] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pages 672–681, 2013.

[45] Jongwook Kim, Don Batory, Danny Dig, and Maider Azanza. Improving refactoring speed by 10x. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 1145–1156, 2016.

[46] Barbara Kitchenham and Stuart Charters. Guidelines for performing systematic literature reviews in software engineering. Technical report, Version 2.3. EBSE Technical Report. EBSE-2007-01, 2007.

[47] Grischa Liebel, Omar Badreddin, and Rogardt Heldal. Model driven software engineering in education: A multi-case study on perception of tools and UML. In *Proceedings of the 30th Conference on Software Engineering Education and Training (CSEE&T)*, pages 124–133, 2017.

[48] Naouel Moha and Yann-Gaël Guéhéneuc. Ptidej and DECOR: Identification of design patterns and design defects. In *Companion Proceedings of the 22nd Conference on Object-Oriented Programming Systems and Applications (OOPSLA)*, pages 868–869, 2007.

[49] Romulo Nascimento, Eduardo Figueiredo, and Andre Hora. JavaScript API deprecation landscape: A survey and mining study. *IEEE Software*, 39(3):96–105, 2021.

[50] Edson Oliveira, Eduardo Fernandes, Igor Steinmacher, Marco Cristo, Tayana Conte, and Alessandro Garcia. Code and commit metrics of developer productivity: A study on team leaders perceptions. *Empirical Software Engineering (EMSE)*, 25(4):2519–2549, 2020.

[51] Murat Oruc, Fuat Akal, and Hayri Sever. Detecting design patterns in object-oriented design models by using a graph mining approach. In *Proceedings of the 4th International Conference in Software Engineering Research and Innovation (CONISOFT)*, pages 115–121, 2016.

[52] Jukka Paakki, Anssi Karhinen, Juha Gustafsson, Lilli Nenonen, and A Inkeri Verkamo. Software metrics by architectural pattern mining. In *Proceedings of the International Conference on Software: Theory and Practice (ICS), co-located with the 16th IFIP World Computer Congress (WCC)*, pages 325–332, 2000.

[53] Matheus Paixao, Jens Krinke, DongGyun Han, Chaiyong Ragkhitwetsagul, and Mark Harman. The impact of code review on architectural changes. *IEEE Transactions on Software Engineering (TSE)*, 47(5):1041–1059, 2019.

[54] Fabio Palomba, Marco Zanoni, Francesca Fontana, Andrea De Lucia, and Rocco Oliveto. Smells like teen spirit: Improving bug prediction performance using the intensity of code smells. In *Proceedings of the 32nd International Conference on Software Maintenance and Evolution (ICSME)*, pages 244–255, 2016.

[55] Ilka Philippow, Detlef Streitferdt, Matthias Riebisch, and Sebastian Naumann. An approach for reverse engineering of design patterns. *Software and Systems Modeling (SoSyM)*, 4(1):55–70, 2005.

[56] Teade Punter, Marcus Ciolkowski, Bernd Freimut, and Isabel John. Conducting on-line surveys in software engineering. In *Proceedings of the 2nd International Symposium on Empirical Software Engineering (ISESE)*, pages 80–88, 2003.

[57] Ghulam Rasool and Patrick Mäder. Flexible design pattern detection based on feature types. In *Proceedings of the 26th International Conference on Automated Software Engineering (ASE)*, pages 243–252, 2011.

[58] Ghulam Rasool and Patrick Mäder. A customizable approach to design patterns recognition based on feature types. *Arabian Journal for Science and Engineering (AJSE)*, 39(12):8851–8873, 2014.

[59] Ghulam Rasool, Patrick Maeder, and Ilka Philippow. Evaluation of design pattern recovery tools. *Procedia Computer Science*, 3:813–819, 2011.

[60] Maria Riaz, Travis Breaux, and Laurie Williams. How have we evaluated software pattern application? A systematic mapping study of research design practices. *Information and Software Technology (IST)*, 65:14–38, 2015.

[61] Diaeddin Rimawi and Samer Zein. A model based approach for android design patterns detection. In *Proceedings of the 3rd International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*, pages 1–10, 2019.

[62] Ashley Robinson and Christopher Bates. APRT: Another Pattern Recognition Tool. *Journal on Computing (JoC)*, 5(2):46–52, 2017.

[63] Kamran Sartipi and Lei Hu. Behavior-driven design pattern recovery. In *Proceedings of the 12th International Conference on Software Engineering and Applications (SEA)*, pages 179–185, 2008.

[64] Nija Shi and Ronald Olsson. Reverse engineering of design patterns from Java source code. In *Proceedings of the 21st International Conference on Automated Software Engineering (ASE)*, pages 123–134, 2006.

[65] Jyoti Singh, Sripriya Roy Chowdhuri, Gosala Bethany, and Manjari Gupta. Detecting design patterns: a hybrid approach based on graph matching and static analysis. *Inf. Technol. Manag.*, pages 1–12, 2021.

[66] Jason Smith and David Stotts. SPQR: Flexible automated design pattern extraction from source code. In *Proceedings of the 18th International Conference on Automated Software Engineering (ASE)*, pages 215–224, 2003.

[67] Maurício RA Souza, Lucas Veado, Renata Teles Moreira, Eduardo Figueiredo, and Heitor Costa. A systematic mapping study on game-related methods for software engineering education. *Inf. Softw. Technol.*, 95:201–218, 2018.

[68] Krzysztof Stencel and Patrycja Wegrzynowicz. Detection of diverse design pattern variants. In *Proceedings of the 15th Asia-Pacific Software Engineering Conference (APSEC)*, pages 25–32, 2008.

[69] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. Grounded theory in software engineering research: A critical review and guidelines. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 120–131, 2016.

[70] Abdel Aziz Taha and Allan Hanbury. Metrics for evaluating 3d medical image segmentation: analysis, selection, and tool. *BMC medical imaging*, 15(1):1–28, 2015.

[71] Cleiton Silva Tavares, Amanda Santana, Eduardo Figueiredo, and Mariza A. S. Bigonha. Revisiting the bad smell and refactoring relationship: A systematic literature review. In *CIbSE*, pages 434–447, 2020.

[72] Dave Thomas. UML – Unified or Universal Modeling Language? *Journal of Object Technology (JOT)*, 2(1):7–12, 2003.

[73] Malisa Thongrak and Wiwat Vatanawood. Detection of design pattern in class diagram using ontology. In *Proceedings of the 18th International Computer Science and Engineering Conference (ICSEC)*, pages 97–102, 2014.

[74] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros Halkidis. Design pattern detection using similarity scoring. *IEEE Transactions on Software Engineering (TSE)*, 32(11):896–909, 2006.

[75] Ed van Doorn, Sylvia Stuurman, and Marko van Eekelen. Static detection of design patterns in class diagrams. In *Proceedings of the 8th Computer Science Education Research Conference (CSERC)*, pages 79–88, 2019.

[76] Marek Vokác. An efficient tool for recovering design patterns from C++ code. *Journal of Object Technology (JOT)*, 5(1):139–157, 2006.

[77] Markus Von Detten, Matthias Meyer, and Dietrich Travkin. Reverse engineering with the Reclipse tool suite. In *Companion Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, pages 299–300, 2010.

[78] Wei Wang and Vassilios Tzerpos. Design pattern detection in Eiffel systems. In *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE)*, pages 1–10, 2005.

[79] Peter Wendorff. Assessment of design patterns during software reengineering: Lessons learned from a large commercial project. In *Proceedings Fifth European Conference on Software Maintenance and Reengineering*, pages 77–84. IEEE, 2001.

[80] Jim Witschey, Olga Zielinska, Allaire Welk, Emerson Murphy-Hill, Chris Mayhorn, and Thomas Zimmermann. Quantifying developers' adoption of security tools. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 260–271, 2015.

[81] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 1–10, 2014.

[82] Claes Wohlin, Per Runeson, Martin Höst, Magnus Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer Science & Business Media, USA, 1st edition, 2012.

[83] Renhao Xiong, David Lo, and Bixin Li. Distinguishing similar design pattern instances through temporal behavior analysis. In *Proceedings of the 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 296–307, 2020.

[84] Hadis Yarahmadi and Seyed Hasheminejad. Design pattern detection approaches: A systematic review of the literature. *Artificial Intelligence Review (AIR)*, 53(8):5789–5846, 2020.

[85] Stephen Yau and Jeffery Tsai. A survey of software design techniques. *IEEE Transactions on Software Engineering (TSE)*, SE-12(6):713–721, 1986.

[86] Marco Zanoni, Francesca Fontana, and Fabio Stella. On applying machine learning techniques for design pattern detection. *Journal of Systems and Software (JSS)*, 103:102–117, 2015.

[87] Zhixiang Zhang and Qinghua Li. Automated detection of design patterns. In *Proceedings of the 2nd International Workshop on Grid and Cooperative Computing (GCC)*, pages 694–697, 2003.