

Design patterns and code smells. Relationships and impact on selected software quality metrics

Tarek Alkhaeir
Faculty of Computing and Telecommunications
Poznań University Of Technology

Doctoral dissertation submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy

Supervisor
Andrzej Jaszkiewicz, Ph.D., Dr Habil, Assoc. Prof.

Supporting supervisor
Bartosz Walter, Ph.D.

Poznań, 2021

Contents

1 Abstract **4**

2 Abstract (PL) **4**

3 Introduction **5**

4 Design patterns **7**
4.1 Concepts, meaning and characteristics of design patterns in software development 7
4.2 Detection of design patterns 10

5 Code smells **11**
5.1 Concept, meaning and characteristics of code smells in software development 11
5.2 Detection of code smells 13

6 Literature overview **14**
6.1 Design patterns and code smells 14
6.2 Design patterns and changeability 15
6.3 Code smells and changeability 15
6.4 Design patterns and defects 16
6.5 Code smells and defects 16

7 Notation **17**

8 The relationship between design patterns and code smells **18**
8.1 Experimental design 18
8.1.1 Questions 18
8.1.2 Notation 19
8.1.3 Analyzed systems 19
8.1.4 Analyzed smells and patterns 20
8.1.5 Matching pattern and smell classes 20
8.2 Results 20
8.3 *EXP1-RQ1*- Are design pattern classes affected by fewer smells than other classes? 21
8.3.1 JFreeChart 21
8.3.2 Apache Maven 24
8.4 *EXP1-RQ2*- Does the relative number of smelly classes without design patterns to smelly classes with design patterns change during the evolution of a system? 27
8.4.1 JFreeChart 27
8.4.2 Apache Maven 28
8.5 *EXP1-RQ3*- Which code smell-design pattern pairs display significant relationships? 29
8.6 Discussion 35

8.7 *EXP1-RQ1*- Do design pattern classes display fewer smells than other classes? . . . 35
8.8 *EXP1-RQ2*- Does the relative number of smelly classes without design patterns to smelly classes with design patterns change during the evolution of a system? . . . 36
8.9 *EXP1-RQ3*- Which code smell-design pattern pairs display significant relationships . . . 37
8.10 Conclusion . . . 38

9 The effect of code smells on the relationship between design patterns and defects 39
9.1 Experimental design . . . 39
9.1.1 Questions . . . 39
9.1.2 Notation . . . 40
9.1.3 Analyzed systems . . . 40
9.1.4 Analyzed smells, patterns and defects . . . 40
9.1.5 Matching pattern, smell and defect classes . . . 41
9.2 Results . . . 41
9.2.1 *EXP2-RQ1* What is the impact of code smells on the presence/absence of defects in classes involved in design patterns? . . . 41
9.2.2 *EXP2-RQ2* What is the impact of code smells on the defect distribution (number of defects) in classes involved in design patterns? . . . 42
9.3 *EXP2-RQ3* What is the effect of code smells on the relationship between specific design patterns and defects? . . . 48
9.3.1 The binary relationship . . . 48
9.3.2 The distribution of defects . . . 48
9.4 Discussion . . . 50
9.4.1 *EXP2-RQ1* What is the impact of code smells on the presence/absence of defects in classes involved in design patterns? . . . 50
9.4.2 *EXP2-RQ2* What is the impact of code smells on the defect distribution (number of defects) in classes involved in design patterns? . . . 52
9.4.3 *EXP2-RQ3* What is the effect of code smells on the relationship between specific design patterns and defects? . . . 53
9.4.4 The binary relationship . . . 54
9.4.5 The distribution of defects . . . 55
9.5 Conclusion . . . 55

10 What is the impact of code smells on the relationship between design patterns and changeability 57
10.1 Experimental design . . . 57
10.1.1 Questions . . . 57
10.1.2 Notation . . . 58

10.1.3 Analyzed systems . . . 58
10.1.4 Analyzed smells, patterns and change-related metrics . . . 58
10.1.5 Matching patterns, smells and change metrics . . . 59
10.2 Results . . . 60
10.2.1 How the presence, absence and interaction between design patterns and code smells in a class affect the frequency of changes made to this class? . . . 60
10.2.2 How the presence, absence and interaction between design patterns and code smells in a class affect the change size? 63
10.2.3 How the presence, absence and interaction between specific design patterns and specific code smells in a class affect both change-related metrics (size and frequency)? . 66
10.3 Discussion . . . 69
10.3.1 **EXP3-RQ1** How the presence, absence and interaction between design patterns and code smells in a class affect the frequency of changes made to this class? . . . 70
10.3.2 **EXP3-RQ2** How the presence, absence and interaction between design patterns and code smells in a class affect the change size? . . . 71
10.3.3 **EXP3-RQ3** How the presence, absence and interaction between *specific* design patterns and *specific* code smells in a class affect both change-related metrics (size and frequency)? . . . 71
10.4 Conclusion . . . 74

11 Thesis conclusion **74**

12 Contributions **75**
12.1 Contributions for the research . . . 75
12.2 Contributions for the practice . . . 76

13 Limitations **77**

14 Future work **79**

1 Abstract

Design patterns are recommended generic solutions to common design problems. They have a complex relationship with various code quality characteristics. While several papers reported the positive impact of patterns on maintainability, changeability, and defects, other papers provided opposite conclusions, lessened the patterns effect or related their impact with various contextual factors. In this work, we investigate the relationship between design patterns and code smells and study the effect of code smells as a confounding variable in the patterns relationship with changeability and defects.

We start by analyzing two medium-size, mature Java systems with the aim of investigating if the existence of design patterns impacts the presence of code smells and examine how the association between the two phenomena evolve over time. After that, we used non-parametric statistical tests to explore the relationship between design patterns and changeability, and to measure the impact of code smells on this relationship with regards to 13 design patterns and 9 code smells in three medium-size, long-evolving open-source Java systems. Finally, we inspect the link between patterns and defects and capture the difference in the impact on defects between pattern classes with/without smells in 10 Java systems from the PROMISE dataset.

The results show that design pattern classes are more immune to code smells than other classes. However, the strength of this immunity varies between different patterns. Our results also suggest that the evolution of pattern classes through different releases of the same system slightly decrease their association with smells. Furthermore, our analysis concluded that code smells is a valid contextual factor that affects the relationship between design patterns with regards to both defects and changeability as in one hand, smelly patterns tend to receive smaller, but more frequent changes than other classes, and on the other hand, smelly pattern classes are positively associated with defects and also attract more defects than both non-pattern and non-smelly-pattern classes.

2 Abstract (PL)

Wzorce projektowe stanowią polecane ogólne rozwiązania typowych problemów związanych z projektowaniem oprogramowania. Użycie wzorców wpływa na różne właściwości i charakterystyki kodu źródłowego. O ile wiele prac wskazuje na pozytywny związek z wzorców z pielęgnowalnością, zmiennością oprogramowania oraz gęstością defektów, o tyle część wyników prowadzi do odmiennych, czasem nawet przeciwnych wniosków: wpływ wzorców okazuje się dużo słabszy lub powiązany z rozmaitymi czynnikami kontekstowymi. W tej pracy podjęto temat związku pomiędzy wzorcami projektowymi oraz tzw. przykrymi zapachami w kodzie programów, oraz zbadano wpływ tych zależności na zmienność kodu oraz jego gęstość defektów w nim zawartych.

Na początku pracy przedstawiono wyniki analizy dwóch średniej wielkości, dojrzałych systemów napisanych w języku Java pod kątem związków pomiędzy

klasami uczestniczącymi we wzorcach projektowych oraz obarczonych przykrymi zapachami, a także ewolucji tych związków w czasie. Następnie, za pomocą nieparametrycznych testów statystycznych, przeprowadzono badanie wzajemnego wpływu wzorców oraz przykrych zapachów na zmienność oprogramowania w trzech podobnych systemach. Wreszcie, badanie związku tych zjawisk z gęstością defektów, wykorzystując istniejące dane ze zbioru PROMISE.

Wyniki wskazują, że klasy uczestniczące na wzorce projektowe rzadziej są obciążone przykrymi zapachami niż pozostałe klasy, jednak związek ten zmienia się w zależności od konkretnego wzorca. Obserwacje dotyczące zmian tego związku w czasie pokazują także, że w toku ewolucji udział przykrych zapachów w klasach pełniących role we wzorcach nieznacznie spada.

Ponadto, wykonana analiza pozwala na stwierdzenie, że obecność przykrych zapachów jest czynnikiem wpływającym na zmienność kodu i gęstość defektów w we wzorcach projektowych. Klasy obarczone zapachami są zmieniane częściej, jednak same zmiany są mniejsze niż w przypadku innych klas, natomiast klasy wzorców posiadające przykre zapachy posiadają błędy częściej i w większej liczbie zarówno niż klasy nieuczestniczące we wzorcach, jak i klasy uczestniczące we wzorcach, ale pozbawione zapachów.

3 Introduction

Design patterns are reusable solutions to frequent design problems. They were first introduced to software engineering by the Gang of Four's book [42] and since their introduction they seized researchers' interest in a quest to explore and measure their impact on several code quality metrics. Researchers reported the benefit of using design patterns with regard to communication, implementation and documentation [95, 30]. They also documented the positive impact of patterns on maintainability [50], understandability [70], reusability [9] and reducing defect rate [105]. On the other hand, several other studies linked the effect of patterns on quality metrics with many contextual factors or even concluded that patterns have a negative impact on various code quality characteristics. For example, Wendorff et al. [108] reported that the usage of design patterns does not guarantee a better design and may lead to maintainability issues. A similar conclusion stating the negative impact of patterns on maintainability and code evaluation was reported by Khomh et al. [63]. The same author reported that Abstract Factory, Composite, and Flyweight patterns do not improve expandability [66]. Furthermore, Prechelt et al. [92] argued that different patterns have different effect on maintainability and that alternative simpler solutions may lead to a less error-prone code and decrease the cost of maintainability. The inconsistent impact of patterns on modularity, flexibility, and resusability based on their type was also reported by Wydaeghe [110].

The incompatible findings from those studies suggest that the relationship between patterns and code quality metrics is not decisively identified, and that there is a need to evaluate the effect of patterns on other code quality metrics. The inconsistent conclusions also drive us to think that undiscovered contextual

factors may have played a role in the patterns' relationship with those quality metrics. Those factors shaped our direction throughout this thesis. In this thesis, we will investigate the relationship between design patterns and code smells. Next, we study the effect of code smells as a confounding variable in the relationship between patterns on one hand and defects or changeability on the other.

Code smells [40] are surface indicators that usually correspond to deeper problems in the system. As in the cases of patterns, a number of studies evaluated the relationships between code smells and code quality metrics, such as: maintainability [111, 72, 10], understandability [41], security issues [5], defects [32, 73, 23, 22] and changeability [86, 77, 45, 58]. Those research papers often reported mixed or contradictory conclusions suggesting that the relationship between code smells and quality metrics is more complicated than it is initially perceived.

Design patterns and code smells represent two different approaches to assure code quality. Design patterns are perfective solutions, which positively impact some quality attributes which have been empirically validated. On the other hand, eliminating smells are defensive, concentrating on detecting and removing elements that could be harmful for a software system, or that could make it insufficiently effective. Moreover, the preventive methods also include mechanisms that can identify symptoms of anomalies before their negative impact on quality grows and could become destructive for the system.

Thus, patterns and smells not only represent contrasting concepts with regards to code quality, but also the way they are introduced to the code is different. While design patterns are intentionally implemented in the code to achieve specific design objectives, smells get introduced inadvertently. One more notable fact, that patterns and smells are not mutually exclusive, which means that a class can be part of a design pattern while at the same time affected by code smells. For example, the Subject object in the Observer design pattern is a potential suspect to have a God class code smell in it. As the evolution of the Subject could gradually increase its size, complexity, dependents and the number of the notification sent to the observer objects. This may lead to turning it to a God class. Those observations, along with the fact that the relationship between the two phenomena was not heavily investigated in the literature, drove us to investigate the link and possible interactions between the two phenomena, and to examine if this interaction could be considered as a potential confounding variable in the individual relationship between patterns on one hand on one hand and quality attributes such as defects and changeability on the other.

Defects are conditions in software products which do not meet the requirements or the customers' expectations. The process of detecting and removing software defects is an important step in the process of fulfilling the end user satisfaction [44] and reducing the economic liability associated with releasing flawed software products [56]. Furthermore, the effect of code defects on maintainability and maintainability effort was reported in many studies [1, 89]. Thus, the automatic detection of defects was heavily studied in the literature and many proposals were established, such as detecting defects based on metric-based

rules [81] or based on the deviation from good practices [61]. Additionally, the necessity of predicting the future defects based on the current ones, and identifying the defect-prone modules drove researchers to investigate the possibility of building an accurate and effective defect prediction model [68, 13, 38]. Because of the importance of eliminating defects on code quality and their link with maintainability, it is important to investigate the link between design patterns and defects and explore how the presence of code smells affects this relationship.

Beside defects, the other dependent variable which was chosen to be evaluated is changeability. The ISO 9126 model for software product quality considers software changeability as a subcharacteristic of maintainability [27], as it measures the ability of code to evolve and to be changed. Change requests in any system could be triggered by many conditions, such as a change of requirements, a shift in the technologies or even features enhancements. A modular design with a solid implementation should promise an effective fulfillment of those changes within strict limits of resources like time and budgets [51]. Over the years, many metrics were developed to measure code changeability [26, 25, 114]. In this work, we used change size and change frequency to measure it.

The remainder of this thesis is structured as follows. In Sec. 4 and Sec. 5, we describe the list of patterns and smells analyzed in this work, their characteristics and our motivation behind including them in the analysis. Next in Sec. 6 we list several research papers that study the relationship between patterns and smells. We also report the conclusions of studies which examine the effect of smells or patterns on both defects and changeability. Following that we allocate three sections to describe and report the findings of three experiments we conducted in this thesis. Those experiments investigate:

1. Sec. 8 The relationship between patterns and smells
2. Sec. 9 The effect of smells on the relationship between patterns and defects
3. Sec. 10 The effect of smells on the relationship between patterns and changeability

After that, in Sec. 12, we report our scientific and practical contributions. Finally, in Sec. 11 we conclude the findings and propose the directions of our future work.

4 Design patterns

4.1 Concepts, meaning and characteristics of design patterns in software development

Gamma et al. [42] defined design patterns as recommended generic solutions to frequently occurring design problems. They identified 23 design patterns and cataloged them into 3 different types; creational, structural and behavioral.

- *Creational patterns*: Handle the process of creating objects by encapsulating the creation logic.

- *Structural patterns*: Handle composition of classes and objects and organize the relationships between entities.
- *Behavioral patterns*: Organize the communication between objects and define their responsibilities.

In Table 1 we present the original list of design patterns identified in [42]. The analyses presented in the thesis embrace some of those patterns. The selection depends on the capabilities of the detecting tool at the time of each experiment. It is also important to point that throughout the thesis, State and Strategy patterns are reported as the same pattern, as they have the same structure and the detection tool was not able to differentiate between them. The same applies to Adapter and Command patterns.

Name (Category)	Description
Composite (S)	Composes objects in tree-like structures to represent part-whole hierarchies. It is used when the requirement is to treat a single object in a similar manner to a group of objects.
Prototype (C)	Clones an already existing object. It is used when creating a new object is a costly operation, so that cloning it is more affordable.
State-Strategy (B)	State: allows a class for changing its behavior in response to changing its state. Strategy: encapsulates a family of algorithms and make them interchangeable.
Factory Method (C)	Encapsulates an object creation logic behind a well defined common interface.
Template Method (B)	An abstract class which exposes base templates for executing its methods and postponing the implementation of those methods to its sub-classes.
Decorator (S)	Allows adding functionalities to an existing object without altering its structure. It achieves that by wrapping the decorated object and adding new features to it.
Singleton (C)	Creates a single instance of a class and ensures its uniqueness.
Proxy (S)	Provides a substitute object for another object. It controls access to the original object by intercepting requests sent to it and performs specific operations before or after the original object handles those requests.

Adapter (S)–Command (B)	Adapter: Works as a bridge between two incompatible interfaces. Command: Wraps the operations as object commands and sends them to the appropriate invoker objects.
Observer (B)	Defines a one-to-many relationship between a subject and its observers. When the subject changes its state, its observers are notified accordingly.
Visitor (B)	Changes the execution of an algorithm based on a visited object. The element object should accept and allow the visited object to operate on it.
Chain Of Responsibility (B)	Creates a chain of handling objects to serve a single request. If a handling object was able to handle the request, it answers with a response. If it is not, it forwards the request to the next object in the chain.
Bridge (S)	Decouples abstraction from its implementation, so that the two can be changed independently.
Abstract Factory (C)	Produces families of related objects without specifying their concrete classes.
Facade (S)	Hides the complexity of a system by providing a simplified interface to it with a limited set of functions.
Builder (C)	Allows the users to construct complex objects in a step-by-step manner. It also enables producing different types and representations of an object using the same construction code.
Flyweight(S)	Minimizes memory footprint by sharing data between similar objects and reusing those objects instead of creating new ones.
Mediator (B)	Provides a coordinator class which handles the communications between different classes.
Memento (B)	Stores the internal state of an object so it can be restored later.
Iterator (B)	Provides a standardized, uniform way for traversing a collection of objects.
Interpreter (B)	Defines a grammatical representation for a language and provides an interpreter to translate the grammar into a target form based on a specific context.

Table 1: List of design patterns presented in [42]. Category: C–Creational, B–Behavioral, S–Structural

Gamma et al. promoted design patterns as paradigm-specific best practices to solve design problems. They claimed that design patterns are built atop of two object oriented principles [42]:

- Program to an interface not an implementation,
- Favor object composition over inheritance.

The relationship between design patterns and various quality characteristics was heavily investigated in the literature. In Sec. 3 and Sec. 6 we referred to several studies that showed both the positive and the negative impact of patterns on several quality characteristics.

4.2 Detection of design patterns

With the increasing interest in design patterns we have been observing in the last two decades, methods of reliable pattern detection become a key issue for researchers from both academia and industry. As a result, several approaches have been proposed to automatically identify design pattern instances in object oriented source code. However, those approaches differ with respect to their input, methodology, detected types, accuracy and validation methods.

The detection approaches could be categorized into four main groups [4]: (i) database query approaches, (ii) metrics-based approaches, (iii) UML, Graph, and matrix-based approaches and finally, (iv) miscellaneous approaches. Following, we explain each of them:

- *Database query approaches*: In this approach, the source code is transformed into an intermediate representation, such as an AST or XMI. Next, SQL queries are used against the generated representation to retrieve information about the detected patterns. This approach can be used to detect structural or creational patterns. However, it can not detect behavioural patterns.

- *Metrics-based approaches*: This approach relies on source code metrics referring to aggregations, associations and dependencies. The calculated values are compared with values specific for a given pattern. When the similarity between the two metrics exceeds a certain threshold, a pattern is considered as detected.

- UML, Graph, and Matrix-Based approaches: These approaches represent the structural and behavioral information of the subject system as a UML structure, a graph or a matrix. The majority of those approaches have good precision and recall rates, but they usually cannot detect non-canonical implementations of the design patterns.

- Miscellaneous approaches: Those approaches could not be categorized under any of the other approaches. For example: Kraemer et al. [69] represented patterns as Prolog rules, which then used to query a repository of C++ codes. The tool implementing that approach was tested against several software systems and was able to detect only five structural design patterns: Adapter, Bridge, Composite, Decorator and Proxy, with a reported precision of 14-50 % based on the tested system.

Throughout the analyses presented in this thesis, we used a pattern-detection tool written by Tsantalis[1]. This tool uses the Similarity Scoring Approach

[1] `https://users.encs.concordia.ca/~nikolaos/pattern_detection.html`

(SSA), which belongs to the graph and matrix-based category. SSA calculates the similarity between the subject code and graphs representing canonical patterns. If the score exceeds a defined threshold value, the pattern is positively identified [103].

The tool was verified against several Java systems with a reported precision of 100% and a recall of 66.7-100% [4], which makes its performance comparable to other approaches that use *exact/inexact graph matching*, e.g., Discovery Matrix (DP-Miner) [35], the *sub-patterns approach* [112], or *metrics-based approaches*, e.g., MAISA [90] and FUJABA [85].

5 Code smells

5.1 Concept, meaning and characteristics of code smells in software development

Code smells [40] have been proposed by K. Beck as indicators of design issues that could hinder the future maintenance of a software system. Code smells may originate from sub-optimal design- or coding solutions, by making emergency fixes, by employing what is called anti-patterns [12] or as consequence of the technical debt [100]. Code smells should not be confused with defects as they refer to two distinct quality characteristics: defect affect reliability, while smells indicate maintainability-related flaws.

Code smells could be organized in different ways. Mäntylä [80] proposed a taxonomy that categorized code smells into five major groups:

- Bloaters: Classes or methods that have grown excessively, making them difficult to understand or maintain.
- Object-Orientation Abusers: Incorrect implementation of the object-oriented principles.
- Change Preventers: A change in one place in the code triggers sequences of changes in other places. This makes a potentially minor change an expensive operation.
- Dispensables: They represent code structures that are not needed. Removing those elements would make the code easier to understand and maintain.
- Couplers: They results from tight coupling between classes.

In Table 2 we present a list of well-known code smells. Like in the case of patterns, the experiments listed in this thesis analyze only some of those smells. The selection depends on the capability of the tool at the time of each experiment and the number of detected smells in the analyzed systems.

Name (Acronym, Category, Level)	Description
God Class (GC, B, C)	A complex class with too many responsibilities. This could be reflected by having too many methods, attributes or lines of code. It plays the role of a complex controller and it is usually tightly coupled with several other classes which creates maintainability issues.
Feature Envy (FE, C, M)	References members of other objects more frequently than the members of its own object.
Message Chains (MC, C, C)	Violates the Law of Demeter [75] by featuring dependency on a chain of calls that connect objects.
Data Clumps (DCl, B, M)	A set of variables that frequently appear together as method parameters or class attributes. Those attributes should be grouped together inside an entity on their own.
Data Class (DC, D, C)	A class has no responsibility but contains some data items and crude methods to handle them (setters and getters).
External Duplication (ED, D, M)	Duplication of code in different classes.
Schizophrenic Class (SC, CP, C)	A class with several unrelated responsibilities, which are used by several client classes in different contexts.
Tradition Breaker(TB, CP, C)	A subclass that breaks the inherited signatures by providing a new set of services which are not related to those provided by its base class.
Primitive Obsession (OP, B, C/M)	An excessive use of primitive types instead of small objects.
Internal Duplication (ID, D, M)	Duplication of code in a single class.
Sibling Duplication(SD, D, M)	Duplication of code in classes with the same super class.
Switch Statements (SS, OA, M)	A complex conditional statement with several branches.
Refused Bequest (RB, OA, C)	A subclass uses only some of the methods and properties inherited from its base class.
Divergent Change (DA, CP, C)	A small change to a class leads to a series of changes in unrelated methods.
Dead Code (DCo, D, C/M)	A variable, field, method or class is no longer used in the code.

Table 2: List of well-known code smells. Categories: B–Bloaters, C–Couplers, D–Dispensables, CP–Change Preventers, OA–Object-Orientation Abusers. Levels: C–Class, M–Method

5.2 Detection of code smells

In order to apply the metaphor of code smells in practice, accurate smell detection tools are needed. Many detection techniques have been developed and, according to [46], the majority of the currently known code smells could be detected automatically.

Code smells detection techniques could be classified into seven broad categories [62]:

- *Metric-based approach*: Code smells are defined by rules, based on a set of metrics and respective thresholds. The core challenge in this approach is to find optimal threshold values for each metric. This is a complicated issue and requires a significant standardization effort [62].

- *Search-based approaches*: This approach uses Search-Based Software Engineering (SBSE) [48] to solve engineering problems by applying optimization techniques. This approach requires significant knowledge and expertise, as most techniques in this approach apply ML algorithms to detect smells.

- *Symptom-based approaches*: In this approach, symptoms refer to certain notions, like class roles and structures. Those symptoms are translated into the detection rules. The accuracy of this approach is low due to different possible interpretations of the same symptoms. Moreover, the effort needed to translate the symptoms into detected rules is significant.

- *Visualization-based approaches*: It is a semiautomatic technique to detect smells, where data is visualized and enriched using the metric-based approach and then presented to the developer/observer to identify smells. However, the inevitable human involvement makes it error prone and effort-/time consuming.

- *Probabilistic approaches*: This approach evaluate the probability of a class to be affected by a smell. Most techniques in this approach considers the detection process as a fuzzy-logic problem.

- *Cooperative approaches*: This approach was proposed by Boussaa et al. [19] and it depends on the evolution of two populations in parallel. One population evolves a set of detection rules and the other one detects the other code smells which were not covered by the detection rules of the first population.

- *Manual approaches*: It depends on the human expertise in detecting the smells. The techniques in this approach are error prone and time consuming.

In this thesis we used the *InCode* tool to automatically detect smells.*InCode* is a proprietary Eclipse plugin that detects smells based on the static code analysis. The tool employs a technique called *detection strategies* [81], which

relies on Boolean expressions composed of selected code metrics and respective thresholds. As in other metrics-based approach, the chosen threshold values strongly affect the accuracy of the method. In this thesis, the default settings of inCode were used, following the recommendation by Lanza et al. [71].

inCode has several advantages. First, the approach it implements to detect smells is commonly used and accepted; additionally, the detection strategies are fairly accurate in detecting smells ($\approx 70\%$, according to [81]), and a comparative study by Arcelli Fontana et al. [39] found inFusion, a commercial version of inCode that employs the same detection rules, to report the lowest number of false positives among four analyzed smell detectors.

6 Literature overview

6.1 Design patterns and code smells

Investigating the relationship between design patterns and code smells is a relatively new topic, compared to the other studies that explore the link between patterns and many other code quality metrics. Sousa et al. [102] performed an exploratory study on five Java systems and concluded that the use of design patterns does not prevent the presence of code smells in them. The results also suggest that the association between patterns and smells varies between different patterns. For example, Composite, Factory method, and Singleton could be more useful in creating a smell-free code. On the the hand, Adapter-Command, Proxy, and State-Strategy tend to attract a high number of smells. Another exploratory study was presented by Cardoso et al. [24], who found two pattern-smell links; First, the co-occurrences between Command pattern and God Class and second, the co-occurrences of Duplicated Code inside Template pattern classes. The study also presented cases where the patterns were misused or overused, and provided recommendations of how to use those patterns more effectively.

Furthermore, a recent study by Alfadel et al. [6] found that design pattern classes are less smell-prone than other classes. However, classes participating in the Command pattern appeared to be associated with God Class, Blob and External Duplication smells.

Finally, Sousa et al. presented a systematic mapping study on the relationship between patterns and smells [101]. They identified 16 papers and concluded that inaccurate planning of a system together with the inappropriate application of certain patterns are the main causes of the presence of code smells in the patterns. The authors also found that the Command pattern is highly correlated with several smells and that other patterns, like Composite and Template Method, could be also linked with some smells.

6.2 Design patterns and changeability

Classes involved in design patterns tend to evolve in a different way than regular classes. Bieman et al. [18] studied five systems and found that in four out of five analyzed systems, the pattern classes were more change-prone than other classes. On the other hand, the study reported also that in the fifth system the pattern classes were less change-prone than other classes. Furthermore, Aversano et al. [15] performed an empirical study on three open source systems in an attempt to monitor the evolution of design pattern classes. They concluded that patterns which support the application purpose tend to change more frequently than other classes. A similar conclusion was also reported by Rossi et al. [96].

Ampatzoglou et al. [8] presented a study conducted on about 65,000 open source Java classes and concluded that the roles of design patterns classes can predict their stability. The results of the study also showed that classes playing exactly one role in a design pattern were more stable than classes playing more than one role or not involved in any pattern. Furthermore, the study found that some design patterns are less susceptible to changes coming from other classes than other patterns. Another empirical study performed by Di Penta et al. [34] investigated the relationship between design pattern roles and the class change proneness. The results also show that the role played by a class in a design pattern is a valid factor for predicting its changeability. For example, in the Adapter pattern, the class with the Adapter role changes more frequently than the Adaptee class. Also, in Composite pattern, classes that play the role of Composite tend to be more complex than expected and thus receive a higher number of changes.

Furthermore, Gatrell et al. [43] reported that design patterns classes are more change-prone than other classes and a similar conclusion is reported by Bieman et al. [17], who also reported that this conclusion hold up after adjusting the results for the class size.

6.3 Code smells and changeability

The relationship between code smells and change proneness was also investigated in the literature. For example, Palomba al. [91] conducted a large scale empirical study on 30 open source projects and found that smelly classes are more change-prone than smell-free classes. Another study conducted by Khomh et al. [65] on 9 releases of Azureus and in 13 releases of Eclipse found that smelly classes are more change-prone than other classes and that specific smells are more correlated with change than other smells.

Furthermore, Liu et al. [77] investigated the relationship between smells and fine-grained structural change-proneness. They conducted an experiment on 11 open source projects and concluded that smelly classes are more prone to structural changes than non-smelly classes, and classes infected by several smells tend to receive more extensive structural changes. However, after adjusting it for the class size, the effect of some smells on change-proneness decreased or even disappeared. A similar observation was made by Olbrich et al. [87]. They analyzed

data from three open source software systems and found that classes affected by smells, namely God and Brain Classes, change more frequently than other classes. However, after adjusting the result for the class size, they appeared less change-prone than any other classes.

Additionally, experiments using machine learning conducted by Kaur et al. [58, 59] indicated that code smells, in particular God Class and Long Method, are more accurate predictors of change-proneness than static metrics.

6.4 Design patterns and defects

Several studies exploring the relationship between design patterns and defects delivered mixed and sometimes contradictory conclusions. In order to compare the defect rates between classes participating in design patterns and other classes, Vokáč et al. [105] monitored the weekly maintenance and evolution of a large industrial product for three years and concluded that Observer and Singleton patterns tend to have a higher defect rates than other classes. On the other hand, Factory patterns had a lower number of defects and the results for the Template Method were inconclusive. Furthermore, Aversano et al. [14] presented an empirical study on three open source systems and concluded that the number of defects in pattern classes is higher if the implementation of those patterns include crosscutting concerns. The study also asserted that the nature of the pattern significantly affects its defect-proneness.

Moreover, Gatrell et al. [43] studied a large, proprietary, commercial system for two years and found that classes participating in design patterns are more fault-prone than the non-pattern classes. The authors also provided an explanation behind this observation that design pattern classes are more open to change than other classes and they introduced more defects during the evolution of the system. Additionally, the study asserted that some patterns, namely Adaptor, Template Method and Singleton, are more defect-prone than others.

On the other hand, other studies lessened the effect of design patterns on defects. For example, Elish and Mohammed [36] found no difference in the fault density between classes participating in the creational or behavioural patterns and classes without patterns. Nevertheless, structural patterns appeared to have a lower fault density than other classes. A detailed analysis for specific patterns reported also in this study showed that the relationship between patterns and defects varies between different patterns. Furthermore, Onarcan and Fu. [88] investigated the relationship between patterns and defects in a number of open source software projects and concluded that there is a little correlation between the number of pattern instances in those projects and the number of defects. They also concluded that individual design patterns may have either positive or negative impact on defect-proneness.

6.5 Code smells and defects

The connection between code smells and defects has also attracted the attention of researchers. Li et al. [74] investigated on the class level the relationship

between code smells and defects in an industrial, open source system and reported that the presence of some code smells, e.g., God Class, God method and Shotgun Surgery, is positively correlated with defect proneness, while there is no such correlation for other smells, e.g., Data Class and Feature Envy. The authors also suggested that identifying and refactoring classes with code smells during the development could be used systematically to decrease the number of reported defects. Furthermore, Jaafar et al. [53] conducted an empirical study on three open source systems: Azureus, Eclipse and JHotDraw, and reported that the majority of classes affected by code smells tend to be more fault-prone than other classes. Another large scale empirical investigation was performed by Palomba et al. [91]. The results show that smelly classes have a higher fault-proneness than non-smelly classes. Similar conclusions were reported by Nascimento et al. [84] and Bán and Ferenc [21].

On the other hand, Hall et al. [47] argued that the presence of code smells in some circumstances may indeed indicate a fault-prone code. However, the impact of those smells on the defects is rather minor. The authors also suggested that refactoring smelly classes is unlikely to reduce the number of defects in the effected code. A similar observation was reported by D'Ambros et al. [33], who concluded that none of the studied smells could be considered more harmful with respect to software defects.

Tufano et al. [104] investigated the reasons behind the introduction of smells, and to this end they studied the change history of 200 open source projects and concluded that in many cases the refactoring and bug-fixing activities lead to the introduction of smells.

Finally, Caior et al. [22] performed a systematic literature review on 18 studies in an attempt to analyze the impact of code smells on defects. They concluded that 16 studies showed the presence of code smells to affect the number of defects and that this impact could be positive or negative based on the study and the smell and only two studies concluded that code smells have no relationship with defects.

7 Notation

In the thesis, we use the following notation (unless stated otherwise) to denote specific sets in the data.

- ALL: All classes,
- DP: classes which participate in designs pattern(s),
- S: classes which contain code smell(s),
- SDP: classes with a design pattern(s) that are also affected by at least one code smell,
- $nSDP$: classes with a design pattern(s) that are not affected by any code smell(s),

- *SnDP*: classes which are not participating in a pattern, but are affected by at least one code smell,
- *nSnDP*: classes which do not participate in a design pattern and are not affected by code smell(s).

Relationships among S, nS, DP, nDP, SDP, $nSDP$, $SnDP$ and $nSnDP$ sets are presented in Figure 1.

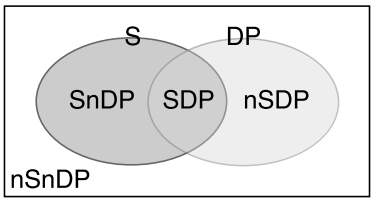


Figure 1: Relationships among the analyzed datasets

8 The relationship between design patterns and code smells

The relationship between patterns and smells was not heavily investigated in the literature. Although, we can intuitively expect that the presence of patterns is correlated with the absence of code smells. There is insufficient existing empirical evidence supporting this claim. To address that and to define this relationship, we conducted an experiment on two medium-size, open source Java systems aiming to investigate how design patterns impact the presence of code smells and how the link between the two phenomena evolve over time. This experiment was published in Information and Software Technology journal [107].

8.1 Experimental design

8.1.1 Questions

In this section, we present the list of specific questions which this experiment is dedicated to answer:

1. *EXP1-RQ1*- Are design pattern classes affected by fewer smells than other classes?
2. *EXP1-RQ2*- Does the relative number of smelly classes without design patterns to smelly classes with design patterns change during the evolution of a system?
3. *EXP1-RQ3*- Which code smell-design pattern pairs display significant relationships?

8.1.2 Notation

Adding to the the list of notations presented in Sec 7, we used the following notation:

- *SDPp*: the percentage of smelly classes participating in a design pattern: $SDPp = |SDP|/|DP|$, where $|m|$: is the number of classes in group m
- *SnDPp*: the percentage of smelly classes that do not participate in a design pattern: $SnDPp = |SnDP|/|nDP|$
- r : ratio of SnDPp to SDPp: $r = SnDPp/SDPp$.

8.1.3 Analyzed systems

The analysis in this experiment was conducted on two long-evolving, open source Java systems: Apache Maven[2] and JFreeChart[3]. They were selected for the study due to their relatively long evolution, a large number of releases, and comparatively high popularity among programmers. Table 3 summarizes basic statistics about those two systems.

System	lang	timespan	releases	classes	%SC	%PC	kLOC
Apache Maven	Java	57 months	32	290-838	10.0-9.4	33.4-39.9	53-57
JFreeChart	Java	155 months	55	101-629	20.8-20.0	18.8-45.2	9-162

Table 3: The Analyzed Systems in the first experiment. %SC is the percentage of smelly classes to all the system classes. %PC the percentage of pattern classes to all other classes

Maven is a software project management and comprehension tool. It automates building software projects by defining and resolving dependencies between different artifacts. Maven helps its users not only to manage, but also share the artifacts in public repositories, so that they can be automatically referenced and acquired by various projects. At he time of this experiment, Maven has had 48 releases in three different development lines (versions 1.x, 2.x and 3.x); two of them have already reached their End-of-Life statuses. The most recent version is 3.2.3, which have 717 classes and ca. 92 kLOC.

JFreeChart is a framework for creating, managing and processing various types of charts that visualize data of different kinds. It also supports numerous output formats for presenting the data. The project was founded in 2000, and is currently the most widely used chart-rendering and processing library for Java. At the time of this experiment, JFreeChart had 57 releases. Its latest version is 1.0.19 which counts 629 classes and ca. 226 kLOC.

[2] `https://maven.apache.org`
[3] `https://www.jfree.org/jfreechart/`

The analysis presented in this experiment have been conducted on selected releases of the subject systems. For Maven, we restricted the analyzed set to 32 subsequent releases between 2.2.0 (the earliest version compatible with Java 5.0) and 3.2.1 (the most recent at the time of writing this experiment). In case of JFreeChart, the analyzed set includes 55 releases, starting with the first publicly available release (0.5.6), and finishing with version 1.0.17, which was also the last one at the moment of conducting the study.

8.1.4 Analyzed smells and patterns

For this experiment, the analyzed design patterns are: Factory Method, Prototype, Singleton, Composite, Decorator, Proxy, Adapter-command, Observer, State-strategy and Template Method. Information about the chosen patterns can be found in Sec 4. The decision about including/excluding patterns was governed by the capability of the chosen tool at the time of the experiment and the total number of detected instances of the patterns.

The analyzed code smells are: Data class, External duplication, Data clumps, Feature envy, God class, Schizophrenic class, Message chains. Information about the smells in Sec 5. Similar to patterns, our choice of the analyzed smells also depended on the capability of the chosen detection tool.

8.1.5 Matching pattern and smell classes

We collected the classes in the *ALL* dataset, by identifying the fully qualified class names inside specific system releases. Then, based on the results of the design pattern and code smell detection, described in Sec 4 and Sec 5, we identified *DP* and *S* datasets.

The granularity for the detection of smells and patterns are not the same as the code smells can be attributed to classes or methods, while patterns are detected on the class level. To confront this issue, we had to adjust the granularity of the datasets to the class level by re-assigning the method-level smells to the enclosing classes.

In the next step we identified intersections of the sets to produce *SDP*, *SnDP*, *nSDP* and *nSnDP*. The resulting datasets were disjoint and complete, i.e., each class was reported exactly once in all datasets, and no class was omitted.

This procedure for generating the pattern-smell datasets would not only be used in this experiment but throughout all the experiments described in this thesis.

8.2 Results

In this section, we explain the procedure we used to answer every question from the questions defined in Sec 8.1.1. We also report the results together with a brief description about the findings. The results will be discussed in details in the discussion section.

8.3 *EXP1-RQ1-* Are design pattern classes affected by fewer smells than other classes?

We want to determine if design pattern classes are linked with fewer smells than other classess. For that we can formulate the following hypotheses:

- null hypothesis $H01 : SDPp = SnDPp$
- alternative hypothesis $Ha1 : SDPp! = SnDPp$
- alternative hypothesis $H11 : SDPp < SnDPp$
- alternative hypothesis $H21 : SDPp > SnDPp$

and to answer this question, we followed this next procedure for both systems.

1. Calculate *SDP* and *nSDP* for all releases in each system
2. Test the normality of *SDP* and *nSDP* distributions with the Shapiro-Wilk test
3. Apply t-test or Wilcoxon test (depending on the normality of the samples) to accept or reject reject the hypotheses defined above.

8.3.1 JFreeChart

Table 4 presents the values of the metrics defined in Sec. 7. In this table, we only present a *selective representative* of the analyzed releases of JFreeChart.

Table 5 reports some descriptive statistics for SDPp and SnDPp.

parameter	SDPp	SnDPp
mean value	0.174	0.205
median	0.148	0.209
std dev	0.088	0.047
variance	0.008	0.002

Table 5: Distribution parameters for SDPp and SnDPp for JFreeChart

Next, we test the normality of distribution for SDPp and SnDPp values. From the data presented in the QQ-plots in Fig. 2 and Fig. 3, the normality of the distribution for SDPp and SnDPp cannot be directly determined. Therefore, we conducted a Shapiro-Wilk normality test. We chose this test as it was found to demonstrate the highest statistical power for a given significance, outperforming other normality tests [94].

release	|classes|	|DP|	|S|	|SDP|	|nSDP|	SDPp	|nDP|	|SnDP|	SnDPp	r
0.5.6	101	19	21	2	17	0.105	82	19	0.232	2.210
0.6.0	89	18	25	4	14	0.222	71	21	0.296	1.333
0.7.0	111	24	29	4	20	0.167	87	25	0.287	1.719
0.7.1	133	28	34	5	23	0.179	105	29	0.276	1.542
0.7.2	134	28	12	2	26	0.071	106	10	0.094	1.324
0.7.3	135	33	39	14	19	0.424	102	25	0.245	0.578
0.7.4	139	35	40	15	20	0.429	104	25	0.240	0.559
0.8.0	148	29	37	8	21	0.276	119	29	0.244	0.884
0.8.1	174	31	51	12	19	0.387	143	39	0.273	0.705
0.9.0	210	26	64	10	16	0.385	184	54	0.293	0.761
0.9.1	233	26	61	10	16	0.385	207	51	0.246	0.639
0.9.2	244	28	63	11	17	0.393	216	52	0.241	0.613
0.9.3	349	57	76	13	44	0.228	292	63	0.216	0.947
0.9.4	373	57	75	15	42	0.263	316	60	0.190	0.722
0.9.5	476	62	75	9	53	0.145	414	66	0.159	1.097
0.9.6	479	62	76	9	53	0.145	417	67	0.161	1.110
0.9.7	587	74	84	10	64	0.135	513	74	0.144	1.067
0.9.8	594	74	89	10	64	0.135	520	79	0.152	1.126
0.9.9	617	122	81	12	110	0.098	495	69	0.139	1.418
0.9.10	602	125	88	14	111	0.112	477	74	0.155	1.384
0.9.11	628	129	93	14	115	0.109	499	79	0.158	1.450
0.9.12	656	140	101	14	126	0.100	516	87	0.169	1.690
0.9.13	675	143	106	15	128	0.105	532	91	0.171	1.629
0.9.14	706	145	127	20	125	0.138	561	107	0.191	1.384
0.9.15	726	147	132	20	127	0.136	579	112	0.193	1.419
0.9.16	739	152	140	21	131	0.138	587	119	0.203	1.471
0.9.17	794	163	147	22	141	0.135	631	125	0.198	1.467
0.9.18	816	171	147	22	149	0.129	645	125	0.194	1.504
0.9.19	855	176	147	22	154	0.125	679	125	0.184	1.472
0.9.20	868	176	151	23	153	0.131	692	128	0.185	1.412
0.9.21	650	180	109	23	157	0.128	470	86	0.183	1.430
1.0.0	773	212	201	16	196	0.075	561	185	0.330	1.397
1.0.1	775	212	116	24	188	0.113	563	92	0.163	1.347
1.0.2	831	213	106	29	184	0.136	618	77	0.125	1.156
1.0.3	508	209	97	31	178	0.148	299	66	0.221	1.070
1.0.4	518	215	102	33	182	0.153	303	69	0.228	1.087
1.0.5	523	217	101	34	183	0.157	306	67	0.219	4.400
1.0.6	534	218	102	35	183	0.161	316	67	0.212	1.442
1.0.7	561	226	106	36	190	0.159	335	70	0.209	0.919
1.0.8	561	226	106	36	190	0.159	335	70	0.209	1.493
1.0.8a	561	226	106	36	190	0.159	335	70	0.209	1.490
1.0.9	561	226	106	36	190	0.159	335	70	0.209	1.395
1.0.10	567	236	107	37	199	0.157	331	70	0.211	1.344
1.0.11	583	249	119	42	207	0.169	334	77	0.231	1.367
1.0.12	585	249	121	42	207	0.169	336	79	0.235	1.391
1.0.13	610	258	127	43	215	0.167	352	84	0.239	1.431
1.0.14	619	275	124	44 22	231	0.160	344	80	0.233	1.456
1.0.15	623	275	125	45	230	0.164	348	80	0.230	1.402
1.0.16	626	279	126	45	234	0.161	347	81	0.233	1.447
1.0.17	629	284	126	45	239	0.158	345	81	0.235	1.487

Table 4: Values of metrics for some the analyzed JFreeChart releases. Where |m|: is the number of classes in group m

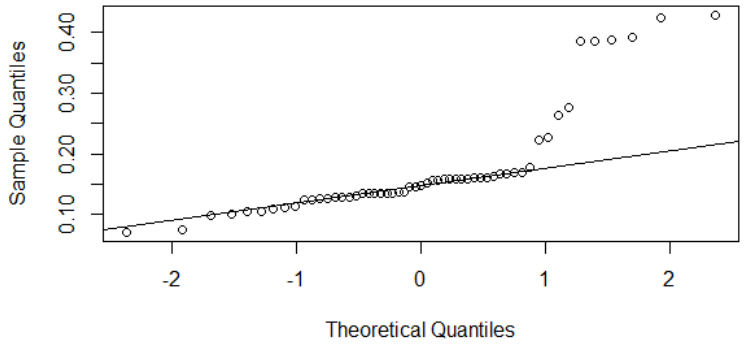


Figure 2: The normal QQ plot for JFreeChart SDPp

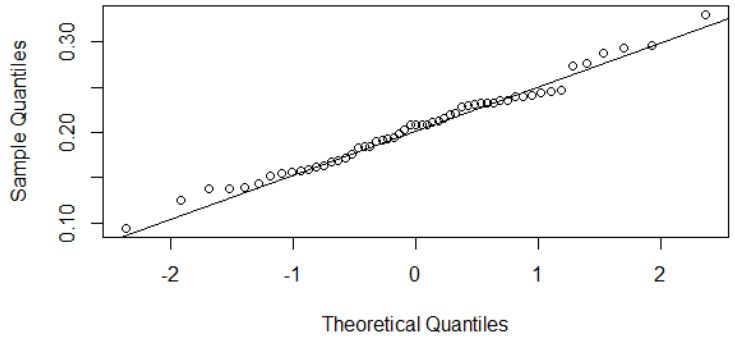


Figure 3: The normal QQ plot for JFreeChart SnDPp

Results are mixed: for SDPp (W=0.719, p-value=6.02e-09) they allow for rejecting the null hypothesis about the normality, but for SnDPp (W=0.986, p-value=0.787) they lead to the opposite conclusion. Effectively, the assumption that both variables have normal distribution is not valid.

After that, we verify H01 hypothesis concerning the difference between the variables. From the visual inspection of the respective mean and medians values of SDPp and SnDPp, and additionally based on the diagram presented in Fig. 4, we expect that for most of the analyzed releases of JFreeChart, the number of smelly classes among the classes which participate in design patterns is smaller than for the remaining classes. However, this assumption still requires strict verification.

One of the compared variables (namely SDPp) is not normally distributed, which prevents us from applying the t-test for paired samples. Therefore, we use a non-parametric one-tailed Wilcoxon signed-rank test [109], which is recommended as an effective replacement for a t-test, used for normal distributions.

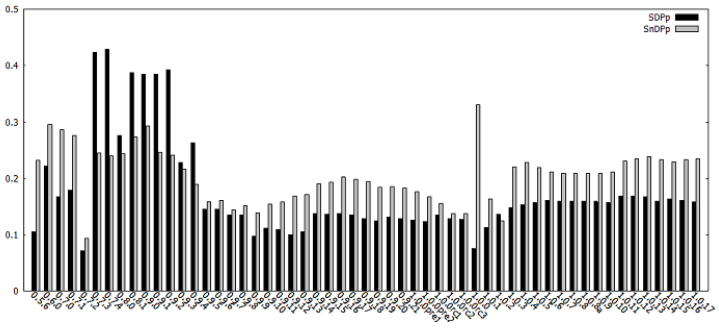


Figure 4: The values of SDPp and SnDPp for JFreeChart

The sample size n is 55 (it is assumed that for $n > 20$ the test statistic W can be approximated as a normal one with a mean calculates as $\mu_w = \frac{n(n+1)}{4}$ and a standard deviation of $\sigma_w = \sqrt{\frac{(n+1)(2n+1)}{24}}$). Based on that, both W-value and z-region were calculated.

The results (W=364.5, z=-3.398, p-value=0.00034, significant at <0.05) allow for rejecting the hypothesis H01. Next, by comparing the medians for both variables (SnDPp: 0.209, SDPp: 0.148), we can accept H11 (SDPp <SnDPp) instead. Therefore, it is valid to conclude that in case of JFreeChart the classes which participate in design patterns exhibit fewer code smells than the other classes.

8.3.2 Apache Maven

Table 7 provides descriptive statistics for SDPp and SnDPp for Maven. The complete data for all Maven's analyzed releases is available in Table 6.

Out of the 33 analyzed releases of the system, version 3.2.0 appeared significantly different in size and in the number of classes from both neighbour versions (the preceding and the following ones). These differences could not be explained by the changes made to the source code in this release, and probably resulted from faulty or incomplete files made available for download by programmers. Therefore, this release was excluded from further analysis.

parameter	SDPp	SnDPp
mean value	0.089	0.140
median	0.079	0.128
std dev	0.020	0.037
variance	0.001	0.001

Table 7: Descriptive statistics for SDPp and SnDPp in Apache Maven

release	|classes|	|DP|	|S|	|SDP|	|nSDP|	SDPp	|nDP|	|SnDP|	SnDPp	r
2.2.0	290	97	29	12	85	0.124	193	17	0.088	0.710
2.2.1rc1	293	98	29	12	86	0.122	195	17	0.087	0.713
2.2.1rc2	293	98	29	12	86	0.122	195	17	0.087	0.713
2.2.1	293	98	29	12	86	0.122	195	17	0.087	0.713
3.0.0	787	307	124	36	271	0.117	480	88	0.183	1.564
3.0.1rc1	694	309	74	24	285	0.078	385	50	0.13	1.667
3.0.1	694	309	88	23	286	0.074	385	65	0.169	2.284
3.0.2rc1	709	320	76	25	295	0.078	389	51	0.131	1.679
3.0.2	809	320	128	36	284	0.113	489	92	0.188	1.664
3.0.3rc1	712	321	76	24	297	0.075	391	52	0.133	1.773
3.0.3	812	322	128	34	288	0.106	490	94	0.192	1.811
3.0.4rc3	713	323	73	24	299	0.074	390	49	0.126	1.703
3.0.4rc4	713	323	73	24	299	0.074	390	49	0.126	1.703
3.0.4rc5	713	323	73	24	299	0.074	390	49	0.126	1.703
3.0.4	821	323	131	34	289	0.105	498	97	0.195	1.857
3.0.5	821	323	131	34	289	0.105	498	97	0.195	1.857
3.0a3	564	254	54	16	238	0.063	310	38	0.123	1.952
3.0a4	564	254	55	17	237	0.067	310	38	0.123	1.836
3.0a5	571	255	57	19	236	0.075	316	38	0.12	1.600
3.0a6	582	259	60	21	238	0.081	323	39	0.121	1.494
3.0a7	593	268	60	20	248	0.075	325	40	0.123	1.640
3.0b1	626	299	63	19	280	0.064	327	44	0.135	2.109
3.0b2	637	303	69	24	279	0.079	334	45	0.135	1.709
3.0b3	676	298	72	25	273	0.084	378	47	0.124	1.476
3.0rc1	686	304	73	24	280	0.079	382	49	0.128	1.620
3.0rc2	686	304	73	24	280	0.079	382	49	0.128	1.620
3.0rc3	686	304	73	24	280	0.079	382	49	0.128	1.620
3.1.0a1	730	331	75	25	306	0.076	399	50	0.125	1.645
3.1.0	838	331	150	36	295	0.109	507	114	0.225	2.064
3.1.1	838	331	148	36	295	0.109	507	112	0.221	2.028
3.2.0	740	111	79	6	105	0.054	629	73	0.116	2.148
3.2.1	740	334	79	25	309	0.075	406	54	0.133	1.773

Table 6: Values of metrics for the analyzed Apache Maven releases. Where |m|: is the number of classes in group m

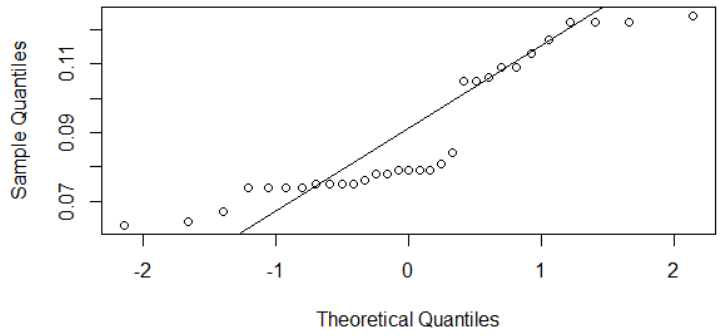


Figure 5: The normal QQ plot for Apache Maven SDPp

As follows from Fig. 5, the SDPp distribution is likely to be not normal, which is validated by the Shapiro-Wilk test of normality (W=0.837, p-value<0.001).

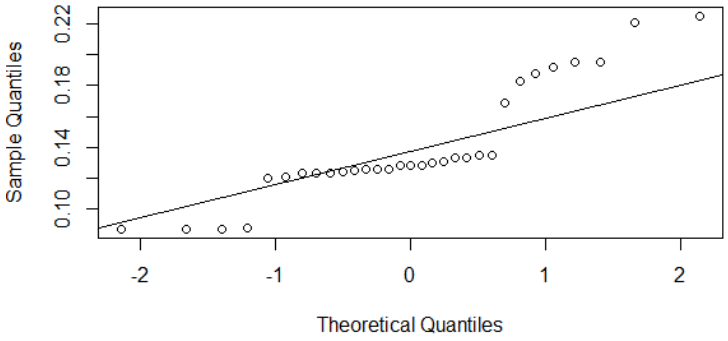


Figure 6: The normal QQ plot for Apache Maven SnDPp

Next, we examined the SnDPp distribution. In this case also, both the QQ-plot and the results of Shapiro-Wilk test (W=0.855, p-value <0.001) suggest that the values are not normally distributed for this variable.

Similar to jfreechart the assumption of the normal distribution of both compared variables is not met which lead us to use Wilcoxon signed-rank test. The sample size (N=32) allows us to approximate the obtained distribution with a normal one, and using the z-value instead. The z-value is -4.750, and since the p-value for the computed z-value <0.00001 (which is significant at $\alpha = 0.05$), the hypothesis concerning equality of variables for Apache Maven is rejected, and one of the alternative hypotheses can be accepted instead.

As follows from the diagram in Fig. 7 and from the comparison of medians for SnDPp (=0.128) and SDPp(=0.079), SnDPp >SDPp. Additionally, the W

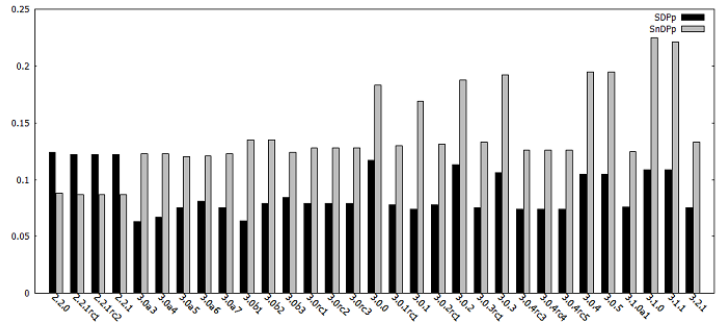


Figure 7: The values of SDPp and SnDPp for Apache Maven

statistic calculated for the one-tailed test corroborates the observation, as it just halves the obtained p-value of the two-tailed test presented above. We assert that for Apache Maven the number of smell-infected classes is lower among those which participate in design patterns than for the other classes.

The results for both systems (Maven and JFreechart) are consistent. As they suggest that design pattern classes tend to have fewer smells than other classes.

8.4 *EXP1-RQ2*- Does the relative number of smelly classes without design patterns to smelly classes with design patterns change during the evolution of a system?

8.4.1 JFreeChart

In this section, we are interested in how the r parameter, $r = SnDPp/SDPp$, is affected by the evolution of the analyzed systems. With this parameter we can observe how much the relation between SDPp and SnDPp changes in the subsequent releases of each system. Stating that the following hypotheses can be formulated:

- null hypothesis H02: r is approximately constant throughout the evolution of a system,
- alternative hypothesis Ha2: there is a trend (non-null) in the values of r for subsequent releases of the system,
- alternative hypothesis H12: the trend for r is positive,
- alternative hypothesis H22: the trend of r is negative.

and to answer this question, we tested those hypotheses by applying the following procedure for all releases of both systems:

1. calculate the value of r,
2. test if there a trend exists for the subsequent values of r, and determine its monotonicity.

The calculated values of r for all JFreeChart releases are presented in Table 4, and visualised in Fig. 8.

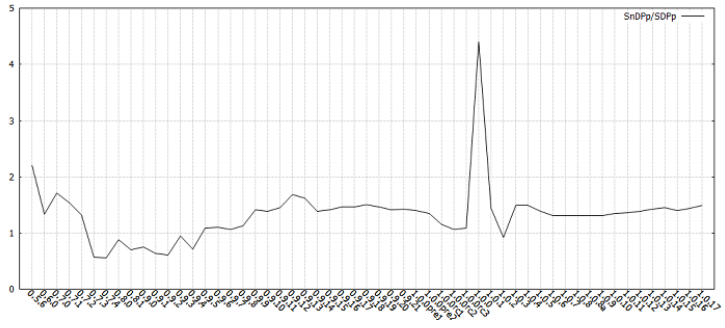


Figure 8: The values of r for JFreeChart in the subsequent releases

For the statistical verification of the hypothesis we applied a two-tailed Mann-Kendall non-parametric trend test [79]. The test was chosen because of its insensitivity to the magnitude of data and the missing data, which made it suitable in this case. The results are presented in Table 8.

parameter	value
Kendall's τ	0.192
stat. S	285.000
variance(S)	18966.333
p-value (two-tailed)	0.039

Table 8: Results of Mann-Kendall trend test for JFreeChart

As the p-value is lower than the significance level $\alpha = 0.05$, the null hypothesis, stating that no trend exists, can be rejected, but with a narrow margin. It should be noted, however, that the obtained p-value is relatively high and could exceed smaller α values, which makes the result for JFreeChart uncertain.

To identify the monotonicity of the trend, a one-tailed test was applied with an alternative hypothesis H12 stating that the trend is positive. The obtained approximation of p-value=0.02 is still lower than $\alpha = 0.05$, so the null hypothesis can be rejected and the alternative accepted instead. However, the above objections are still valid. Therefore, the conclusion that the trend is a *stable* or slightly *positive* for JFreechart.

8.4.2 Apache Maven

For Apache Maven we followed the same procedure as for JFreeChart. First, the r values for all 32 releases of the system were calculated (Table 9) and visualized

(Fig. 9).

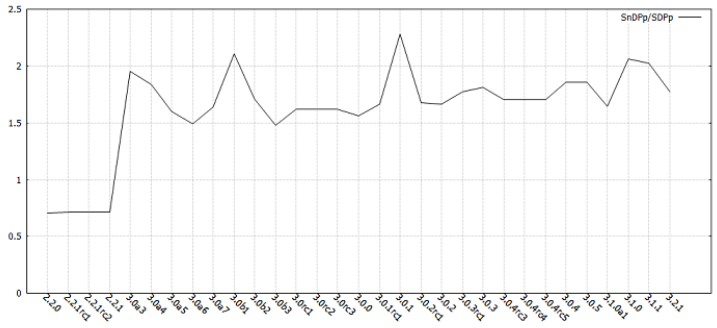


Figure 9: The values of r for Apache Maven

In order to verify the monotonicity of r, we conducted the Mann-Kendall two-tailed test again; the results are presented in Table 9.

parameter	value
Kendall's τ	0.457
stat. S	210.00
variance(S)	3448.667
p-value (two-tailed)	$<$0.001

Table 9: Results of the Mann-Kendall trend test for Apache Maven

The obtained p-value (0.001) is smaller than the assumed α=0.05, so we reject the null hypothesis H0, and one of the alternative hypotheses can be accepted instead. As a result, and additionally based on the Fig. 9, we conclude that for Apache Maven there exists a *positive* trend for subsequent r values.

The results from both analyzed systems are inconclusive: for Apache Maven the r parameter displays positive monotonicity, whereas in case of JFreeChart the trend of r is uncertain. We cannot provide a conclusive answer to EXP1-RQ2, but we can assert that the trend for r is stable or increasing.

8.5 *EXP1-RQ3*- Which code smell-design pattern pairs display significant relationships?

In this section, we are interested in discovering the association between individual design patterns and code smells. In particular, we are interested in extracting rules which display significant relationships between patterns and smells. In order to do that and possibly discover other new findings, we decided to mine the dataset.

The number of classes exhibiting both smells and patterns in each release is relatively small, which makes it not suitable for data mining. To mitigate that, we decided to extend the dataset by analyzing all identified instances in all releases, for both systems. That increased the size of the dataset, but also posed a risk that instances found in different releases could depend one each other. We attempted to reduce the hazard and avoid counting the same class in different releases several times, by removing duplicates from the dataset. In this context, a duplicate is a class with a set of the same code smells, existing in more than one release. It should be noticed that this procedure could still leave duplicates, if the class was simply renamed. However, the manual inspection we performed on a random sample of classes, did not reveal such a case. As a result, the extended data set included 2105 classes with at least one code smell and being a part of at least one design pattern.

By examining the description of the patterns, we can expect some of their links with smells. For example:

- The Strategy pattern defines a set of algorithms as objects which are separated from the data. This separation could result in introducing Feature Envy smell in the dependent classes that hold the data, as the decoupled algorithm objects still needs the data.

- The Observer pattern is founded on the relation between a subject class and observers that are notified if the subject changes its state. Gradual evolution of the subject may increase the number of the notifications, and eventually, making the subject a God Class.

- Proxy pattern is a smart facade to another class, which can also introduce a God Class smell to the proxy. Additionally, a stack of proxies that includes several intermediate layers could be also a manifestation of a Middle Man smell.

By scanning Table 10 which presents data about the identified pattern-smell pairs, some conclusions could be drawn directly:

- Composite and Singleton classes contain only 0 and 1 instance of a code smell, respectively. These two patterns have a simple construction and precisely define the responsibilities of the participating classes. This could provide a possible explanation for the observation. It is worth to say, that this is in contradiction with the common critique of the Singleton pattern, which is claimed to negatively affect the design.

- Classes participating in the Template Method pattern contain only God- and Schizophrenic Class smells.

[4]Pattern and smelly classes
[5]All classes which participate in design patterns
[6]Classes with smells and patterns
[7]All detected smell instances (both participating and not participating in patterns)

pattern ↓/code smell →	DC	GC	FE	SC	MC	DCl	ED	Sum[4]	Total[5]
Observer	0	68	0	30	0	0	0	98	451
Composite	0	0	0	0	0	0	0	0	54
Singleton	1	0	0	0	0	0	0	1	5364
Proxy	0	20	0	0	0	0	0	20	591
Prototype	95	215	7	27	0	0	1	345	3441
State-Strategy	26	390	78	69	2	0	0	565	7117
Adapter-Command	55	347	50	68	0	0	0	520	4928
Template Method	0	19	0	18	0	0	0	37	474
Decorator	0	0	0	20	0	0	0	20	935
Factory Method	0	0	0	16	0	0	0	16	836
Sum[6]	177	1059	135	248	2	0	1	1622	24191
Total[7]	1977	1359	353	409	14	6	683	4801	

Table 10: Relationships of individual code smells and design patterns. DC – Data Class, GC – God Class, FE – Feature Envy, SC – Schizophrenic Class, MC – Message Chains, DCl – Data Clumps, and ED – External Duplication.

- Classes with Decorator and Factory Method contain only instances of the Schizophrenic Class smell.
- Proxy and Observer patterns are collocated with God Class smell.
- Virtually no design pattern was found to co-exist with the Data Clumps and Message Chains smells. This result from an overall small number of classes with these smells.
- External Duplication was detected only in a single class participating in a Prototype
- Prototype, State-Strategy and Adapter-Command display numerous relations with some code smells, in particular Data Class, God Class, Feature Envy and Schizophrenic Class.

We are primarily interested in identifying relationships that strongly connect individual patterns and code smells. In order to extract them, we employed *association rules*, which express the dependencies between attributes in a data set. In our case, the attributes correspond to the presence of code smells and design patterns. Specifically, we used the Weka's [8] implementation of the Apriori algorithm [3].

Extracted association rules can be evaluated with different measures reflecting their practical properties. The most popular measures include *support* and *confidence* [3], which are an intuitive way reflect how important and accurate a given rule is. Support measures how *important* a rule is with respect to the entire dataset, whereas confidence reflects its *accuracy*. Both metrics have values from range $(0; 1)$, and higher values indicate more significant rules. However, these measures were found not to be suitable for rules which are extracted from relatively small, highly sparse datasets like in the case of our dataset. Additionally, our dataset contains only Boolean values that denote the presence or absence of a pattern or a smell. This prevent the support and confidence metrics to be convenient measures of the rules' significance [16]. Instead, we decided to use *conviction* [20] to evaluate the rules. Conviction combines support and confidence in a single measure, showing how often an analyzed rule would be incorrect if the analyzed association could be attributed to a random chance. Conviction takes values from range $(0.5; \infty)$: 1 indicates that antecedent and consequent are independent, values smaller than 1 – indicate negative dependency, whereas values greater than 1 – a positive dependency between both sides of the rule. It should be noted that the conviction is sensitive to changes of both confidence and support, which means that a given value could be interpreted in various ways.

The imbalanced distribution of data in our dataset (dominance of zeros over ones on all attributes) would produce several strong, but uninteresting rules of the form $(attr_n = 0) \Rightarrow (attr_m = 0)$, where 0 code smells and patterns could appear on both sides of the rule. As we are interested in finding associations

[8] http://www.cs.waikato.ac.nz/ml/weka/

between a design pattern as an antecedent and a smell instance as a consequent, we decided to analyze the associations for every pattern and code smell separately, and to ignore rules with code smells on the left-hand-side. The minimum support and confidence levels in Weka was set to very small values, so that even weak rules could be identified for further analysis.

Based on that, 261 association rules have been extracted. As some of them display very low conviction, which makes them not interesting, below we present 43 rules with conviction ≥ 2.0, ordered by the descending value of this measure. The threshold was chosen arbitrally as a value slightly greater than the border value of 1. As a result, the reduced set contains rules that exhibit at least moderately positive dependency between the antecedent and consequent.

rule	confidence	conviction
Singleton $\Longrightarrow \neg$ GC	1	346.62
Singleton $\Longrightarrow \neg$ DC	1	252.12
State-Strategy $\Longrightarrow \neg$ ED	1	231.13
Singleton $\Longrightarrow \neg$ ED	1	174.2
Adapter-Command $\Longrightarrow \neg$ ED	1	160.04
Singleton $\Longrightarrow \neg$ SC	1	104.32
Singleton $\Longrightarrow \neg$ FE	1	90.03
Decorator $\Longrightarrow \neg$ DC	1	87.89
FactoryMethod $\Longrightarrow \neg$ DC	1	78.59
Decorator $\Longrightarrow \neg$ GC	1	60.42
Prototype $\Longrightarrow \neg$ ED	1	55.87
Proxy $\Longrightarrow \neg$ DC	1	55.56
FactoryMethod $\Longrightarrow \neg$ GC	1	54.02
TemplateMethod $\Longrightarrow \neg$ DC	1	44.56
Observer $\Longrightarrow \neg$ DC	1	42.4
Decorator $\Longrightarrow \neg$ ED	1	30.36
FactoryMethod $\Longrightarrow \neg$ ED	1	27.15
State-Strategy $\Longrightarrow \neg$ DC	1	24.78
State-Strategy $\Longrightarrow \neg$ TB	1	21.66
Proxy $\Longrightarrow \neg$ ED	1	19.19
Singleton $\Longrightarrow \neg$ TB	1	16.32
Decorator $\Longrightarrow \neg$ FE	1	15.69
TemplateMethod $\Longrightarrow \neg$ ED	1	15.39
Adapter-Command $\Longrightarrow \neg$ TB	1	15
Observer $\Longrightarrow \neg$ ED	1	14.65
FactoryMethod $\Longrightarrow \neg$ FE	1	14.03
Proxy $\Longrightarrow \neg$ SC	1	11.49
Prototype $\Longrightarrow \neg$ TB	1	10.47
Proxy $\Longrightarrow \neg$ FE	1	9.92
Adapter-Command $\Longrightarrow \neg$ DC	0.99	8.27
TemplateMethod $\Longrightarrow \neg$ FE	1	7.96
Observer $\Longrightarrow \neg$ FE	1	7.57
Prototype $\Longrightarrow \neg$ FE	1	7.22
Composite $\Longrightarrow \neg$ DC	1	5.08
Singleton $\Longrightarrow \neg$ MC	1	3.57
Composite $\Longrightarrow \neg$ GC	1	3.49
Prototype $\Longrightarrow \neg$ DC	0.97	3.37
Adapter-Command $\Longrightarrow \neg$ MC	1	3.28
Decorator $\Longrightarrow \neg$ TB	1	2.85
FactoryMethod $\Longrightarrow \neg$ TB	1	2.54
Prototype $\Longrightarrow \neg$ SC	0.99	2.39
Prototype $\Longrightarrow \neg$ MC	1	2.29
State-Strategy $\Longrightarrow \neg$ DCl	1	2.03

Table 11: Extracted association rules combining design patterns and code smells

As expected, all extracted rules combine the presence of a design pattern and the absence of a code smell. The rules that involve the presence of a code smell, have a negligible support.

8.6 Discussion

In this section, we provide explanations to our results reported in the result section and we relate the findings to other studies.

8.7 *EXP1-RQ1*- Do design pattern classes display fewer smells than other classes?

The main finding of our experiment is that there exists a link between design patterns and code smells. Classes participating in design patterns display smells less frequently than other classes. This general observation was further corroborated in the answer to EXP1-RQ3, where we identified several *negative* relationships between individual patterns and code smells.

As follows from Tables 4 and 6, smelly classes are not prevalent: they affect only 9.0–30.5% of classes in JFreeChart and 9.6–17.9% for Maven. Moreover, classes that have smells and participate in patterns are also very infrequent: they make only ca. 1.5–7.2% of all classes in JFreeChart, and 0.8–4.6% in Maven.

The relation between SnDPp and SDPp for JFreeChart, presented in Fig. 4, varies in time. For majority of releases the observed relationship is in line with our final conclusion ($SDPp < SnDPp$), but for releases 0.7.3–0.9.4 it is reversed. Manual inspection of the code revealed that in release 0.7.3 a large number of classes with Data Clumps were introduced, and several of them participated also in Adapter-Command, State-Strategy and Template Method patterns. These smells were subsequently removed from the release 0.9.4 onwards. Release 1.0.0 is another special case, for which SnDPp peaks, and SDPp drops down. We do not observe these changes in further releases, which suggests that it was a one-time event resulting from sub-optimal design choices, and the smells were quickly eradicated in the next releases.

We observe a similar phenomenon in case of Maven. As follows from Fig. 7, the relationship between SDPp and SnDPp also changes in time: for releases 3.0.0a-3.2.1, SDPp is smaller than SnDPp, which supports our conclusion; however, for releases 2.2.0-2.2.1 the results are reversed and the initial order is restored in the next releases. Manual inspection showed that this behaviour resulted from the introduction of numerous instances of Data Clumps, God Class, Feature Envy and Data Class smells, located mainly in classes with Adapter-Command, State-Strategy and Factory Method patterns. It is important to notice that absolute values for SDP and SnDP (36 and 88, respectively) are higher in release 3.0.0 than for the first analyzed release. However, this effect is compensated by an increase in the total number of classes (from 293 in release 2.2.1 to 787 in 3.0.0), which results in reversed order for SDPp and SnDPp variables.

By analyzing the presence of individual smells in the code, we can identify the most frequent ones: Data Clumps and God Class for JFreeChart, which dominate over other smells from very early releases of the system, and are later accompanied by a few instances of Feature Envy and Schizophrenic Class. Similarly, the design patterns are also not uniformly distributed: Adapter-Command, State-Strategy, Prototype and Observer are among the most commonly used. Collocations of these patterns and smells make up 88% of all smelly classes that participate in patterns for JFreeChart, and 91% for Maven.

8.8 *EXP1-RQ2*- Does the relative number of smelly classes without design patterns to smelly classes with design patterns change during the evolution of a system?

The results obtained in response to EXP1-RQ2 do not lead to a single conclusion. Diagrams in Fig. 8 and Fig. 9 present the values of r in all the analyzed releases. The parameter r is a ratio of SnDPp to SDPp, and it measures the relation between smelly classes that do and do not participate in design patterns. The data in diagrams exhibit significant variability of the analyzed parameter.

The chart in Fig. 8 presents a peak for release 1.0.0 of JFreeChart, which is the first major officially released version. This event cannot be easily explained based only on the aggregated data, so we manually inspected the code. The main finding is that in this release the number of smelly classes is substantially higher than in other releases, while the number of the smelly design pattern classes is lower. It is a consequence of the fact that some classes (e.g., `PolarPlot`, `PiePlot` and `ChartPanel`), which had smells and patterns in previous versions, have been removed from the codebase in this release. However, this change resulted in proliferation of smells (in particular Feature Envy) in other classes: the number of classes having this smell increased from 10 in release 0.9.21 to 117 in 1.0.0, and then decreased back to 10 in release 1.0.1. That could result from the eradicating effect of design patterns reported by [52], which are applied to remove a code smell, and which are then removed altogether.

In case of JFreeChart, a clear trend for r could not be determined due to a relatively high p-value obtained from the statistical test, which makes the result questionable for relatively higher α values. We can, however, make a conservative conclusion that the r value is approximately stable or slightly increasing, although the latter conclusion should be verified in larger experiments.

The analogous observation for Maven is more clear, as the positive trend has been statistically validated. The peaks in releases 3.0a3, 3.0b1 and 3.0.1 are much smaller, and do not alter significantly the general trend for the variable.

Data collected in this study are insufficient to provide a well-supported interpretation of the results for EXP1-RQ2. One possible explanation is that patterns precisely define roles and interactions of their participating classes, and provide some guidance for developers on the recommended design solutions. As a result, developers may want to prefer patterns that help removing a code smell than to apply alternative design choices, and do that throughout the evolution of the

design pattern	cum. conviction
Singleton	987.18
State-Strategy	279.60
Adapter-Command	186.59
Decorator	197.21
FactoryMethod	176.33
Prototype	81.61
Proxy	96.16
TemplateMethod	67.91
Observer	64.62
Composite	8.57

Table 12: Cumulative conviction of rules that include specific design patterns

software system, possibly even with increasing frequency.

8.9 *EXP1-RQ3*- Which code smell-design pattern pairs display significant relationships

Looking for an answer to EXP1-RQ3, we focused on extracting the relationships between individual smells and patterns, represented by association rules. Within a rule, the presence of a pattern is an antecedent, and a code smell serves as a consequent. Noticeably, all significant rules we found combine the *presence* of a design pattern, and the *absence* of a code smell, which indirectly supports the finding for EXP1-RQ1. Some rules, however, represent patterns connected with smells, but their significance (expressed by conviction) is almost negligible, and have not been reported in Table 11. As we see in Table 12, Singleton, State-Strategy, Adapter-Command, Factory Method and Decorator are among patterns that are usually not collocated with smells, whereas for Composite this relationship is considerably weaker.

In order to evaluate the impact of particular design patterns on the significance of the rules (measured by conviction), we calculated the cumulative conviction of all rules that include a given design pattern. The results are presented in Table 12. State-Strategy, Decorator, Adapter-Command and Factory Method are the strongest patterns that are present in the extracted rules, with cumulative conviction over 170, but they are far outperformed by Singleton, which is peaking with the cumulative conviction at 987.18. On the other hand, the cumulative conviction for Composite is just 8.57, which suggests that the presence of this pattern is not a strong factor that affects the presence of code smells.

In a similar manner, we can identify the smells that are usually not collocated with patterns (see Table. 13). In this case, External Duplication, Data Class and God Class exhibit highest cumulative conviction, whereas Data Clumps has the lowest value for this measure.

code smell	acronym	cum. conviction
External Duplication	ED	727.98
Data Class	DC	602.62
God Class	GC	464.55
Schizophrenic Class	SC	118.20
Tradition Breaker	TB	68.84
Feature Envy	FE	52.42
Message Chains	MC	9.14
Data Clumps	DCl	2.03

Table 13: Cumulative conviction of rules that include specific code smells

8.10 Conclusion

In this experiment we analyzed the relationships between design patterns and code smells. Based on the evidence collected from the analysis of two medium-size Java systems, our findings generally support the intuitive hypothesis that the presence of design patterns correlates with the absence of code smells in the same classes.

The experiment contributions are threefold:

- We found that the presence of design patterns is linked with the absence of code smells in the same classes. The systematic literature review on the effectiveness of patterns[113] concludes that knowledge about the presence of patterns could be used as a factor for building a framework that supports maintenance. As a result of our experiment, the framework could be extended to include also code smells. This fact could be exploited by developers in constructing more effective smell detectors, which utilize the knowledge of patterns to concentrate the analysis on the parts of code that deserve more thorough examination. The presence of patterns appears to be one of the contextual variables in this case.

- The significance of the relationships between design patterns and code smells varies with respect to the specific patterns and smells. What is noteworthy that all extracted significant pairs combine the presence of a design pattern as an antecedent, and the absence of a code smell as a consequent, which means that both phenomena are usually disjoint. Specifically, we identified patterns, which are more likely not to be related with code smells than others: State-Strategy, Adapter-Command, and Factory Method. To our surprise, these patterns are outperformed in this context by the Singleton, which is not in line with the findings reported in other studies. The conclusion concerning diversity of the patterns' impact on smells could also be used by tools vendors for enhancing capabilities of code smell detectors and tuning the analysis process with respect to individual patterns. Additionally, these differences between patterns could

provide recommendations for developers concerning the choice of a pattern in a given context.

- The ratio of smelly classes that don't and do participate in a design pattern, appears stable or slightly increasing in subsequent releases of both analyzed systems. The obtained results are mixed: in one of the system the ratio is increasing, whereas appears rather stable in the other system. It means that the number of smelly classes that are not a part of design patterns is either growing proportionally to the number of other smelly classes, or slightly faster. We could conclude that the frequency of smells within classes with patterns is lower or equal during the code evolution than for other classes.

The results confirm the intuitive hypothesis on the mutually exclusive nature of smells and patterns. They represent different approaches to assuring code quality, but they appear negatively correlated. These observations supplement our knowledge about code smells by introducing a new factor that can affect their presence, and confirm several previous conclusions about the context-sensitive nature of smells.

9 The effect of code smells on the relationship between design patterns and defects

The relationship between patterns and defects was investigated in the literature, but with mixed results. While the majority of studies found the presence of patterns to be positively correlated with defects, other works reported the opposite conclusions. This may suggest that contextual factors affect this relationship. One of those contextual factors could be code smells. In order to investigate the confounding effect of smells on code that contains design patterns, in terms of the resulting defects, we designed and performed an experiment on 10 medium size Java systems from the PROMISE dataset [2]. This experiment was published in IEEE Access journal [7].

9.1 Experimental design

9.1.1 Questions

This experiment considers three questions that examine the defect-proneness of pattern classes, depending on the presence/absence of code smells in them.

1. *EXP2-RQ1* What is the impact of code smells on the presence/absence of defects in classes involved in design patterns?
2. *EXP2-RQ2* What is the impact of code smells on the defect distribution (number of defects) in classes involved in design patterns?
3. *EXP2-RQ3* What is the effect of code smells on the relationship between specific design patterns and defects?

System	Description
Ant-1.7	Java library and command-line tool to compile, assemble, test and run Java applications.
JEdit-4.2	A modular and extensible text editor with hundreds of customizable plugins.
Lucene-2.4	Java library for performing advanced indexing and searching.
Camel-1.6	A message-oriented middleware and integration framework that provides an object-based implementation of the enterprise integration patterns.
Log4j-1.2	An extensible logging framework for Java applications.
Xalan-2.7	An XML processor for applying XSL transformations and XPath queries.
Poi-3.0	A library for manipulating MS Office documents.
Ivy-2.0	An extensible dependency manager.
Xerces-2.0	An XML parser.
Velocity-1.6	A template engine with a built-in expression language.

Table 14: List of subject systems analyzed in experiment 2

9.1.2 Notation

In addition to notation defined in Sec 7. We define the following notation:

- *DEF*: Classes with at least one defect;
- *DEF-DP*: Classes involved in design pattern(s) and with at least one defect;
- *DEF-nDP*: Classes not involved in design pattern(s) and with at least one defect;

9.1.3 Analyzed systems

We performed our analysis on 10 small- and medium-size Java systems from the PROMISE [2] dataset, one of the largest public repositories of empirical software data. We used one of the datasets that provides information about defects. The original dataset includes 14 open source java systems: Ant, Camel, Ckjm, Forrest, Ivy, JEdit, Log4J, Lucene, PBeans, Poi, Synapse, Velocity, Xalan and Xerces. We decided to exclude four systems: Ckjm, PBeans, Synapse and Forrest, due to the negligible number of patterns (< 5) in them.

Table 14 presents the list of the analyzed systems

9.1.4 Analyzed smells, patterns and defects

In this experiment, we studied 13 design patterns. The analyzed design patterns are: Factory Method, Prototype, Singleton, Composite, Decorator, Proxy,

Adapter-command, Observer, State-strategy, Chain Of Responsibility, Visitor and Template Method. Information about the chosen patterns can be found in Sec 4. The pattern detection strategy and tool are presented in Sec 4.2.

For smells, we analyzed 10 code smells:Data class, External duplication, Data clumps, Feature envy, Internal Duplication, Tradition Breaker, Sibling Duplication, God class, Schizophrenic class and Message chains. Information about the smells in Sec 5. The smells algorithm and tool are presented in Sec 5.2.

In this experiment defects were acquired from the PROMISE repository. PROMISE has used Buginfo tool to collected data about defects; Buginfo is a tool which evaluates every commit in the repository of the analyzed system. The tool labels the commit as a bug fix if it solves an issue reported as a bug in the bug tracking system. For each analyzed project, the bug fixes commenting guidelines were discovered and formalized as regular expressions. Buginfo compares the regular expressions with the comment associated with the commit. If the comment matches the regular expression, Buginfo reports detecting a defect and increases the defect count for every class modified in the commit [57].

The PROMISE dataset has been validated and used several times in different research papers, e.g., concerning bug prediction [38, 37].

9.1.5 Matching pattern, smell and defect classes

First, we collected the classes in the *ALL* dataset, by identifying the fully qualified class names inside specific system releases within the source PROMISE dataset. Then, based on the results of the design pattern and code smell detection, described in 4.2 and 5.2, we identified *DP* and *S* datasets.

Since the code smells can be attributed to classes or methods, while patterns involve classes, we had to adjust the granularity of the datasets to the class level by re-assigning the method-level smells to the enclosing classes.

In the next step we identified intersections of the sets to produce *SDP*, *SnDP*, *nSDP* and *nSnDP*. The detailed procedure is described in Sec 8.1.5. In the resulting datasets each class was reported exactly once in all datasets, and no class was omitted. However, defects in the PROMISE dataset are assigned to files, which can contain one or more classes. To address the issue of several classes being included in a single file, but assigned to different datasets, we manually verified and excluded such cases.

9.2 Results

9.2.1 *EXP2-RQ1* What is the impact of code smells on the presence/absence of defects in classes involved in design patterns?

First, we evaluate if design pattern classes are associated with the presence or absence (considered as a binary value) of defects in them. For this purpose, we used the Odds Ratio (OR) test [99] to find the associations between the presence of patterns as an exposure and the presence of defects as an outcome. The OR function is specified as follows: $OR = \frac{a/c}{b/d}$ where:

System	*ALL*	*DEF*	*DP*	*nDP*	*SDP*	*DEF-DP*	*DEF-nDP*
Ant-1.7	745	166	110	635	12	38	128
JEdit-4.2	366	47	66	300	11	15	32
Lucene-2.4	335	198	164	171	12	105	93
Camel-1.6	964	187	188	776	13	75	112
Log4j-1.2	205	189	44	161	5	43	146
Xalan-2.7	904	893	151	753	23	151	742
Poi-3.0	442	281	57	385	11	39	242
Ivy-2.0	350	38	101	249	13	18	20
Xerces-2.0	586	435	110	476	19	74	361
Velocity-1.6	229	78	89	140	8	40	38

Table 15: Numbers of classes that belong to respective datasets

- a = number of exposed cases
- b = number of exposed non-cases
- c = number of unexposed cases
- d = number of unexposed non-cases

The result of the OR test is interpreted as follows:

- $OR = 1$ The exposure does not affect the odds of the outcome,
- $OR > 1$ The exposure is associated with higher odds of the outcome,
- $OR < 1$ The exposure is associated with lower odds of the outcome.

Next, we use Fisher's exact test (FET) to determine if the OR results are significant [97]. The results are presented in Table 16 and the extracted association rules are summarized in Table 17.

The extracted rules indicate that patterns are positively associated with the presence of defects. This is in line with the results of other studies , e.g., [43], and in contradiction to the common understanding of patterns in terms of their positive impact on code quality [93]; on the other hand, they reinforce our conjectures concerning contextual factors that may play a role in this association. Our next step is to investigate the role of code smells as a confounding factor. Table 18 presents the results from the OR and FET tests and the extracted rules are listed in Table 19.

9.2.2 *EXP2-RQ2* What is the impact of code smells on the defect distribution (number of defects) in classes involved in design patterns?

For this question we are interested whether the presence of patterns in classes is associated with a higher/lower number of defects, and what is the effect of

	OR		$log(OR)$		
System	DP	nDP	DP	nDP	FET
Ant-1.7	2.09	0.478	0.737	-0.737	Significant ($p = 0.002$)
JEdit-4.2	2.463	0.406	0.901	-0.903	Significant ($p = 0.014$)
Lucene-2.4	1.664	0.669	0.509	-0.401	Not significant ($p = 0.076$)
Camel-1.6	3.935	0.254	1.370	-1.370	Significant ($p < 0.001$)
Log4j-1.2	4.417	0	1.485	$-\infty$	Not significant ($p = 0.202$)
Xalan-2.7	∞	0	-	$-\infty$	Not significant ($p = 0.227$)
Poi-3.0	1.280	0.781	0.247	-0.247	Not significant ($p = 0.463$)
Ivy-2.0	2.483	0.403	0.909	-0.909	Significant ($p = 0.013$)
Xerces-2.0	0.654	1.527	-0.424	0.423	Not significant ($p = 0.070$)
Velocity-1.6	2.192	0.456	0.785	-0.785	Significant ($p = 0.007$)

Table 16: Results of the OR test and FET

System	Extracted Rules
Ant-1.7	$DP \implies$ Defects, $nDP \implies \neg$ Defects
JEdit-4.2	$DP \implies$ Defects, $nDP \implies \neg$ Defects
Lucene-2.4	No significant rules
Camel-1.6	$DP \implies$ Defects, $nDP \implies \neg$ Defects
Log4j-1.2	No significant rules
Xalan-2.7	No significant rules
Poi-3.0	No significant rules
Ivy-2.0	$DP \implies$ Defects, $nDP \implies \neg$ Defects
Xerces-2.0	No significant rules
Velocity-1.6	$DP \implies$ Defects, $nDP \implies \neg$ Defects

Table 17: Summarized findings from the OR test and FET

System	*SDP*		*nSDP*	
	OR	*FET*	*OR*	*FET*
Ant-1.7	5.054	Significant; $p = 0.007$	1.755	Significant; $p = 0.026$
JEdit-4.2	6.210	Significant;$p = 0.007$	1.646	Not significant; $p = 0.195$
Lucene-2.4	1.4	Not significant;$p = 0.767$	1.431	Not significant; $p = 0.119$
Camel-1.6	3.646	Significant; $p = 0.025$	3.701	Significant; $p < 0.001$
Log4j-1.2	∞	Not significant; $p = 1$	3.775	Not significant; $p = 0.317$
Xalan-2.7	∞	Not significant; $p = 1$	∞	Not significant; $p = 0.383$
Poi-3.0	∞	Significant; $p = 0.009$	0.879	Not significant; $p = 0.747$
Ivy-2.0	11.516	Significant; $p < 0.001$	1.243	Not significant; $p = 0.557$
Xerces-2.0	∞	Significant; $p = 0.006$	0.462	Significant; $p = 0.002$
Velocity-1.6	17.788	Significant; $p = 0.002$	1.573	Not significant; $p = 0.144$

Table 18: The OR test and FET results, considering the effect of code smells

System	Extracted rules
Ant-1.7	$SDP \implies DEF$ $nSDP \implies DEF$
JEdit-4.2	$SDP \implies DEF$
Lucene-2.4	no significant rules
Camel-1.6	$SDP \implies DEF$ $nSDP \implies DEF$
Log4j-1.2	no significant rules
Xalan-2.7	no significant rules
Poi-3.0	$SDP \implies DEF$
Ivy-2.0	$SDP \implies DEF$
Xerces-2.0	$SDP \implies$ Defects $nSDP \implies \neg\, DEF$
Velocity-1.6	$SDP \implies DEF$

Table 19: The findings from the OR test and FET, considering the effect of code smells

System	*DP*		*nDP*	
	Mean	Median	Mean	Median
Ant-1.7	0.772	0	0.399	0
JEdit-4.2	0.772	0	0.18	0
Lucene-2.4	2.158	1	1.228	1
Camel-1.6	1.345	0	0.307	0
Log4j-1.2	2.909	2	2.298	2
Xalan-2.7	1.311	1	1.338	1
Poi-3.0	2.403	1	0.942	1
Ivy-2.0	0.277	0	0.096	0
Xerces-2.0	5.618	1.5	1.974	1
Velocity-1.6	1.146	0	0.628	0

Table 20: Descriptive statistics for the number of defects inside pattern and non-pattern classes

smells as a contextual factor in this relationship is. To answer this, we first tested the normality of defect distributions in the subject datasets. In Table 20 we present the descriptive statistics of the datasets and in Table 34 we report the results of the Shapiro-Wilk test [98]. They show that both pattern and non-pattern values are not normally distributed.
Since all values are not normally distributed, we used a nonparametric Wilcoxon-Mann test (WMW) [78] to verify if two populations have different medians of defect distributions. In this section we verify the hypothesis that pattern and non-pattern classes do not differ with respect to defects. Analogously, similar hypotheses are used when comparing any other two groups using WMW throughout this analysis.

1. H0: $DP = nDP$ w.r.t. defects
2. Ha: $DP \neq nDP$ w.r.t. defects
3. Ha1: $DP < nDP$ w.r.t. defects
4. Ha2: $DP > nDP$ w.r.t. defects

The results of the WMW test are presented in Table 22.

As follows from the results, $DP > nDP$ for 9 out of the 10 analyzed systems, and the only remaining case is inconclusive. We repeated the same steps to measure the effect of smells on these results. In Table 23 we summarize the WMW test results between the smelly- and non-smelly pattern classes, and between the classes in those two groups and the non-pattern classes. We also present the extracted rules from this analysis in Table 24.

To assess the significance of our extracted rules, we performed an effect size analysis to measure the mean difference between the different groups. We

System	*DP*				*nDP*			
	W_{crit}	σ	W	p	W_{crit}	σ	W	p
Ant-1.7	0.976	1.566	0.540	≈ 0	0.995	1.040	0.441	≈ 0
JEdit-4.2	0.963	2.044	0.431	≈ 0	0.990	0.634	0.312	≈ 0
Lucene-2.4	0.983	2.706	0.768	≈ 0	0.984	1.591	0.766	≈ 0
Camel-1.6	0.985	3.276	0.440	≈ 0	0.996	1.084	0.308	≈ 0
Log4j-1.2	0.947	1.697	0.622	≈ 0	0.983	1.435	0.847	≈ 0
Xalan-2.7	0.948	1.736	0.673	≈ 0	0.995	0.743	0.571	≈ 0
Poi-3.0	0.958	4.105	0.580	≈ 0	0.992	1.263	0.601	≈ 0
Ivy-2.0	0.974	0.680	0.463	≈ 0	0.988	0.346	0.298	≈ 0
Xerces-2.0	0.976	9.515	0.619	≈ 0	0.993	3.039	0.545	≈ 0
Velocity-1.6	0.971	2.086	0.584	≈ 0	0.981	1.485	0.473	≈ 0

Table 21: Results of the Shapiro-Wilk test

System	*DP* vs. *nDP*	Conclusion	Hedges' g
Ant-1.7	$z = -3.406, p < 0.001$	$DP > nDP$	0.330 (small)
JEdit-4.2	$z = -2.782, p = 0.005$	$DP > nDP$	0.570 (medium)
Lucene-2.4	$z = -3.091, p = 0.001$	$DP > nDP$	0.420 (small)
Camel-1.6	$z = -8.104, p < 0.001$	$DP > nDP$	0.595 (medium)
Log4j-1.2	$z = -2.202, p = 0.027$	$DP > nDP$	0.407 (small)
Xalan-2.7	$z = 0.599, p = 0.548$	none	
Poi-3.0	$z = -2.391, p = 0.016$	$DP > nDP$	0.775 (medium)
Ivy-2.0	$z = -2.745, p = 0.006$	$DP > nDP$	0.386 (small)
Xerces-2.0	$z = -2.106, p < 0.035$	$DP > nDP$	0.736 (medium)
Velocity-1.6	$z = -2.780, p = 0.005$	$DP > nDP$	0.295 (small)

Table 22: Results of WMW and Hedges' g tests

used the Hedges' g test [49] with a corresponding 95% confidence interval (CI). Hedges' g provides a measure of the effect size weighted by the relative size of each sample. The results are interpreted according to Cohen's d conventions [29]:

- Negligible effect < 0.2
- Small effect $= 0.2$
- Medium effect $= 0.5$
- Large effect $= 0.8$

The results and interpretation are reported in Tables 22 and 24.

System	*SDP* vs. *nSDP*	*SDP* vs. *nDP*	*nSDP* vs. *nDP*
Ant-1.7	$z = -2.557, p = 0.010$	$z = -3.764, p < 0.001$	$z = -2.460, p = 0.013$
JEdit-4.2	$z = -2.323, p = 0.020$	$z = -3.823, p < 0.001$	$z = -1.601, p = 0.109$
Lucene-2.4	$z = -0.724, p = 0.468$	$z = -1.702, p = 0.088$	$z = -2.916, p = 0.003$
Camel-1.6	$z = -0.059, p = 0.952$	$z = -3.049, p = 0.002$	$z = -2.232, p = 0.025$
Log4j-1.2	$z = 0.535, p = 0.592$	$z = -0.392, p = 0.695$	$z = -2.786, p = 0.005$
Xalan-2.7	$z = -4.991, p < 0.001$	$z = -3.945, p < 0.001$	$z = -2.289, p = 0.022$
Poi-3.0	$z = -2.839, p = 0.004$	$z = -3.801, p < 0.001$	$z = -0.923, p = 0.355$
Ivy-2.0	$z = -3.783, p < 0.001$	$z = -5.489, p < 0.001$	$z = -1.267, p = 0.204$
Xerces-2.0	$z = -4.769, p < 0.001$	$z = -6.246, p < 0.001$	$z = 0.305, p = 0.760$
Velocity-1.6	$z = -2.307, p = 0.021$	$z = -3.519, p < 0.001$	$z = -2.129, p = 0.033$

Table 23: Results of the WMW test for specific patterns, considering the effect of code smells

System	*SDP* vs. *nSDP*	*SDP* vs. *nDP*	*nSDP* vs. *nDP*
Ant-1.7	*SDP* > *nSDP*. H'g=1.327 (large)	*SDP* > *nDP*. H'g=1.878 (large)	*nSDP* > *nDP*. H'g=0.156 (negligible)
JEdit-4.2	*SDP* > *nSDP*. H'g=1.319 (large)	*SDP* > *nDP*. H'g=2.765 (large)	not significant
Lucene-2.4	not significant	not significant	*nSDP* > *nDP*.H'g=0.420 (small)
Camel-1.6	not significant	*SDP* > *nDP*. H'g=0.284 (small)	*nSDP* > *nDP*.H'g=0.624 (medium)
Log4j-1.2	not significant	not significant	*nSDP* > *nDP*. H'g=0.404 (small)
Xalan-2.7	*SDP* > *nSDP*.H'g=1.379 (large)	*SDP* > *nDP*. H'g=0.926 (large)	*nSDP* > *nDP*. H'g=-0.224 (small)
Poi-3.0	*SDP* > *nSDP*. H'g=1.328 (large)	*SDP* > *nDP*. H'g=3.245 (large)	not significant
Ivy-2.0	*SDP* > *nSDP*. H'g=1.321 (large)	*SDP* > *nDP*. H'g=2.149 (large)	not significant
Xerces-2.0	*SDP* > *nSDP*. H'g=0.976 (large)	*SDP* > *nDP*. H'g=2.992 (large)	not significant
Velocity-1.6	*SDP* > *nSDP*. H'g=1.229 (large)	*SDP* > *nDP*. H'g=1.559 (large)	*nSDP* > *nDP*. H'g=0.195 (negligible)

Table 24: The extracted rules from the WMW test results presented in Table 23 together with Hedges' g results

Pattern	*ALL*	*DEF*	*SDP*	*nSDP*	def-*SDP*	def-*nSDP*
(Object) Adapter	417	232	55	362	39	193
Bridge	18	12	3	15	3	9
Chain of Responsibility	3	3	1	2	1	2
Composite	16	10	1	15	1	9
Decorator	103	65	1	102	1	64
Factory Method	79	36	2	77	1	35
Observer	14	9	2	12	2	7
Prototype	14	10	2	12	2	8
Proxy	29	26	3	26	3	23
Singleton	80	39	7	73	6	33
State	202	116	37	165	30	86
Template Method	60	25	8	52	4	21
Visitor	38	12	5	33	5	7

Table 25: The total number of classes for each pattern in each group

9.3 *EXP2-RQ3* What is the effect of code smells on the relationship between specific design patterns and defects?

9.3.1 The binary relationship

In this section we investigate the binary relationship between individual patterns and defects and how the presence of smells impacts this relationship. As the number of specific patterns is too low in each system, we merged the datasets. In Table 25 we present the descriptive data on the resultant dataset.

Next, we performed again the OR test for each pattern and applied FET to measure the significance of the results (see Table 26).

From the results we can conclude that the classes involved in Adapter, Decorator, Proxy and State are more defect-prone than non-pattern classes. The Visitor pattern appears to be associated with the absence of defects, but due to a large p-value we consider the result to be uncertain.

We followed these steps by measuring the effect of smells on these extracted associations and for that we repeated the same tests taking into consideration the effect of smells (see Table 27). The results suggest that the positive association between Adapter, State, and Visitor patterns with defects exists only if they are affected by smells.

9.3.2 The distribution of defects

In this section we investigate whether specific patterns attract more or fewer defects than non-pattern classes, and how the introduction of smells affects this distribution.

Pattern	OR	$log(OR)$	FET
(Object)Adapter	1.336	0.289	p=0.004
Bridge	2.086	0.735	p=0.159
Chain of Responsibility	∞	∞	p=0.117
Composite	1.737	0.552	p=0.323
Decorator	1.800	0.587	p=0.003
Factory Method	0.883	−0.124	p=0.571
Observer	1.897	0.640	p=0.292
Prototype	2.607	0.958	p=0.111
Proxy	9.102	2.208	p<0.001
Singleton	0.989	−0.011	p=1.000
State	1.423	0.352	p=0.014
Template Method	0.740	−0.301	p=0.298
Visitor	0.477	−0.740	p=0.034

Table 26: Results of the OR test and FET for the specific patterns

Pattern	*SDP*			*nSDP*		
	OR	$Log(OR)$	FET	OR	$Log(OR)$	FET
Adapter	2.634	0.968	$p = 0.001 < 0.05$	1.204	0.185	$p = 0.091$
Decorator	∞	∞	$p = 0.490$	1.772	0.572	$p = 0.006 < 0.05$
Proxy	∞	∞	$p = 0.117$	8.042	2.084	$p = 0 < 0.05$
State	4.501	1.504	$p = 0.0001 < 0.05$	1.137	0.128	$p = 0.429$
Visitor	∞	∞	$p = 0.028 < 0.05$	0.278	-1.280	$p = 0.001 < 0.05$

Table 27: Results of OR test and FET for specific patterns, considering the effect of code smells

Pattern	*DP* vs. *nDP*	Conclusion	Hedges' g
(Object)Adapter	$z = -5.284, p < 0.001$	$DP > nDP$	0.574 (medium)
Bridge	$z = -1.598, p = 0.109$	H0 cannot be rejected	
Chain Of Resp	$z = -1.758, p = 0.078$	H0 cannot be rejected	
Composite	$z = -1.839, p = 0.065$	H0 cannot be rejected	
Decorator	$z = -2.907, p = 0.003$	$DP > nDP$	0.146 (negligible)
Factory Method	$z = -0.308, p = 0.757$	H0 cannot be rejected	
Observer	$z = -1.925, p = 0.054$	H0 cannot be rejected	
Prototype	$z = -2.331, p = 0.019$	$DP > nDP$	0.351 (small)
Proxy	$z = -4.761, p < 0.001$	$DP > nDP$	0.517 (medium)
Singleton	$z = -1.921, p = 0.054$	H0 cannot be rejected	
State	$z = -4.385, p < 0.001$	$DP > nDP$	0.629 (medium)
Template Method	$z = 0.117, p = 0.906$	H0 cannot be rejected	
Visitor	$z = 1.857, p = 0.063$	H0 cannot be rejected	

Table 28: Results of the WMW test for specific patterns, together with Hedges 'g results

In Table 28, we present the results of the WMW test that compare the distribution of defects for specific patterns with the distribution of defects in non-pattern classes.

According to the results, the Adapter, Decorator, Prototype, Proxy and State patterns are more defect-prone than non-pattern classes, while the results for the other patterns are inconclusive. We also performed a Hedges' g test and the results show that the extracted rules have a different significance depending on the pattern type. They report that none of the extracted rules has a large effect size and that the significance varies between medium, small or negligible, depending on the type of the pattern.

To investigate the effect of smells on the previous rules, we performed a similar analysis for the classes with code smells. In Table 29 we present the results of the WMW test that compares the defects distribution in the smelly and non-smelly classes involved in patterns, with a distribution of defects in non-pattern classes. Table 30 reports the extracted rules from the WMW test and the significance of those extracted rules based on the results of the Hedges' g test.

9.4 Discussion

9.4.1 *EXP2-RQ1* What is the impact of code smells on the presence/absence of defects in classes involved in design patterns?

If we consider the binary relationship between patterns and defects (i.e., defective and defect-free classes) the results show that in five of the analyzed systems (Ant-1.7, JEdit-4.2, Camel-1.6, Ivy-2.0 and Velocity-1.6), the presence of pat-

Pattern	*SDP* vs. *nSDP*	*SDP* vs. *nDP*	*nSDP* vs. *nDP*
(Object) Adapter	$z = -3.35, p < 0.001$	$z = -5.323, p < 0.001$	$z = -3.716, p < 0.001$
Bridge	$z = -2.594, p = 0.009$	$z = -2.991, p = 0.002$	$z = -0.416, p = 0.677$
Chain Of Resp	insufficient data	insufficient data	$z = -1.702, p = 0.088$
Composite	insufficient data	insufficient data	$z = -1.468, p = 0.141$
Decorator	insufficient data	insufficient data	$z = -2.786, p = 0.005$
Factory Method	$z = 0.171, p = 0.863$	$z = 0.182, p = 0.855$	$z = -0.341, p = 0.732$
Observer	$z = -2.167, p = 0.030$	$z = -2.657, p = 0.007$	$z = -0.996, p = 0.318$
Prototype	$z = -1.613, p = 0.106$	$z = -2.235, p = 0.025$	$z = -1.607p = 0.107$
Proxy	$z = -1.779, p = 0.075$	$z = -2.658, p = 0.007$	$z = -4.131, p < 0.001$
Singleton	$z = -2.183, p = 0.028$	$z = -2.858, p = 0.004$	$z = -1.135, p = 0.256$
State	$z = -3.987, p < 0.001$	$z = -5.656, p < 0.001$	$z = -2.199, p - 0.027$
Template Method	$z = -0.536, p = 0.591$	$z = -0.566, p = 0.570$	$z = 0.347, p = 0.728$
Visitor	$z = -3.269, p = 0.001$	$z = -1.987, p = 0.046$	$z = 2.767, p = 0.005$

Table 29: Results of WMW test for specific patterns, considering the effect of code smells

terns is positively associated with the presence of defects. For the other five systems we could not find any significant rules. It is also important to point that no rule that contradicts the extracted rules was identified, so in no system could we relate patterns to the absence of defects. After the introduction of the effect of smells to the analysis, we found that three out of five systems that were found to have a positive relationship between patterns and defects exhibit this relationship only if the patterns are affected by smells (JEdit-4.2, Camel-1.6, Ivy-2.0 and Velocity-1.6), and for the other two (Ant-1.7 and Camel-1.6) the relationship exists regardless of the presence of smells in the patterns.

In the remaining five systems, for which we initially could not reject the null hypothesis, patterns in Poi-3.0 were found to be positively related with defects only if they were smelly; in Xerces-2.0 the effect was even more evident: smelly patterns have been positively associated with defects, while the association for non-smelly patterns was negative.

By including code smells we did not only extract more rules, but we also found that in the majority of systems, patterns are positively associated with defects only when they are smelly. On the other hand, the results for non-smelly patterns are mixed, so while they are positively associated with defects in a couple of systems, they also have negative or no relationship with defects in other systems. This observation provides a possible explanation for the mixed [106] or small relationship [88] reported in the literature between design patterns and defect-proneness. The presence of code smells appears to be a factor that interacts with design patterns and has a decisive impact on defects in the subject code by amplifying the previously existing defect-proneness. Consequently, it has a practical consequence for software developers. The intense use of patterns

Pattern	*SDP* vs. *nSDP*	*SDP* vs. *nDP*	*nSDP* vs. *nDP*
(Object) Adapter	*SDP* > *nSDP*. H'g=0.429(small)	*SDP* > *nDP*. H'g= 1.730(large)	*nSDP* > *nDP*. H'g=0.475(small)
Bridge	*SDP* > *nSDP*. H'g= 2.120(large)	*SDP* > *nDP*. H'g= 9.72(large)	not significant
Chain Of Resp	not significant	not significant	not significant
Composite	not significant	not significant	not significant
Decorator	not significant	not significant	*nSDP* > *nDP*. H'g=0.14(negligible)
Factory Method	not significant	not significant	not significant
Observer	*SDP* > *nSDP*. H'g=4.294(large)	*SDP* > *nDP*. H'g=5.163(large)	not significant
Prototype	not significant	*SDP* > *nDP*. H'g=1.349(large)	not significant
Proxy	not significant	*SDP* > *nDP*. H'g=1.349(large)	*nSDP* > *nDP*. H'g=0.421(small)
Singleton	*SDP* > *nSDP*. H'g=0.6(medium)	*SDP* > *nDP*. H'g=1.708(large)	not significant
State	*SDP* > *nSDP*. H'g=0.813(large)	*SDP* > *nDP*. H'g= 2.135(large)	*nSDP* > *nDP*. H'g=0.334(small)
Template Method	not significant	not significant	not significant
Visitor	*SDP* > *nSDP*. H'g=1.476(large)	*SDP* > *nDP*. H'g=1.469(large)	*nSDP* < *nDP*.H'g=-0.308(small)

Table 30: The conclusions from the WMW test results presented in Table 29, together with the Hedges' g effect size test results

can lead to their interactions and the proliferation of cross-cutting effects [14], resulting in some types of code smells. That, in turn, could effectively diminish or revert the expected advantages of applying design patterns, even if the pattern classes attract fewer smells than the non-pattern ones [107].

9.4.2 *EXP2-RQ2* What is the impact of code smells on the defect distribution (number of defects) in classes involved in design patterns?

The results for nine out of ten of the analyzed systems indicate that design pattern classes are linked with a higher number of defects than the non-pattern classes. Only in the case of Xalan-2.7 no significant rules were identified. The effect size analysis reported that the mean difference between the smelly and non-smelly patterns in terms of defects is between [0.2-0.5] of standard deviation, which entails that the significance of those extracted rules are either small or medium, depending on the system.

By introducing information about code smells into the analysis, we obtained new insights into those results. First, in the majority of systems (seven out of ten), smelly patterns have a higher number of defects than the non-smelly patterns, and no system produced contradictory results. The effect size for the majority of those extracted rules is large, indicating that the difference between the two groups is of a large significance.

With regard to the effect of smells on the relationship between pattern vs. non-pattern classes with defects, the extracted rules were difficult to interpret, because while smelly patterns are associated with more defects than non-pattern classes in eight systems, the non-smelly patterns also have more defects than non-pattern classes in six systems. Those results initially suggested that the smelliness of a pattern is not a valid contextual factor for analyzing defect-proneness. A thorough analysis of the extracted rules shows that the extracted rules for the relationship between smelly patterns and defects are stronger than those which show the relationship between non-smelly patterns and defects. All the rules extracted for the smelly patterns are significant at $\alpha = 0.01$, while

only two rules are significant at the same level for the non-smelly patterns. The effect size analysis also strengthens this conclusion, since for the smelly pattern rules the mean difference is large enough to be of a practical significance, while the effect size for the non-smelly pattern rules is either small or even negligible.

To have a more comprehensive understanding of the results and to isolate the effect of smells, we again performed a WMW test to compare the smelly vs. non-smelly patterns with smelly vs. non-smelly non-pattern classes. The results are summarized in Table 31.

System	*SDP* vs. *SnDP*	*SDP* vs. *nSnDP*	*nSDP* vs. *SnDP*	*nSDP* vs. *nSnDP*
Ant	not significant	$SDP > nSnDP$(large)	not significant	$nSDP > nSnDP$(small)
JEdit	not significant	$SDP > nSnDP$(large)	not significant	not significant
Lucene	not significant	not significant	not significant	$nSDP > nSnDP$(small)
Camel	not significant	not significant	$nSDP > SnDP$(small)	$nSDP > nSnDP$(medium)
Log4j	not significant	not significant	not significant	$nSDP > nSnDP$(small)
Xalan	$SDP > SnDP$(small)	$SDP > nSnDP$(large)	$SnDP > nSDP$(small)	not significant
Poi	$SDP > SnDP$(large)	$SDP > nSnDP$(large)	not significant	not significant
Ivy	$SDP > SnDP$(large)	$SDP > nSnDP$(large)	not significant	not significant
Xerces	$SDP > SnDP$(large)	$SDP > nSnDP$(large)	$SnDP > nSDP$(negligible)	not significant
Velocity	$SDP > SnDP$(large)	$SDP > nSnDP$(large)	not significant	not significant

Table 31: The extracted rules from the WMW test, together with the effect size interpretation of the Hedges' g test

The results confirm our initial observations that smelly patterns, in the majority of systems, have a higher defect distribution than the non-pattern classes, regardless of whether those classes are smelly or not. On the other hand, the comparison of non-smelly patterns with smelly and non-pattern classes showed that they have similar defect distribution in the majority of the systems and that non-smelly patterns have a higher defect distribution only in case of Camel, while they have a lower number of defects in case of Xalan and Xerces. This may suggest that the effect of smells and patterns on defects is cumulative and the results of comparing data that belong to only one group (smells or patterns) could be attributed to other contextual factors. It is also worth mentioning that the effect size analysis of the relationship between $nSDPvs.SnDP$ shows that the significance of the mean difference between those two groups is small or even negligible. The results also suggest that classes which are not participating in a pattern and are not affected by smells tend to attract fewer defects than pattern classes.

Finally, the results of the effect size analysis strengthen our conclusion, as they show the large significance of the extracted rules related to smelly design patterns, while they demonstrate the small significance of the rules related to the non-smelly patterns.

9.4.3 *EXP2-RQ3* What is the effect of code smells on the relationship between specific design patterns and defects?

In the subsequent sections we discuss the binary and the cumulative relationships between specific patterns and defects, and the effect of smells on these relationships.

9.4.4 The binary relationship

We are interested in analyzing the binary relationship between specific patterns and defects, and in describing how the presence of smells affects this relationship. However, the dataset has a very small number of instances for some patterns, e.g., Chain of Responsibility (3 instances, all of them are smelly), Bridge (18 instances, only three of them are smelly), Observer and Prototype (14 instances each, only 2 instances in each case are smelly). Because of the small sample size, FET reported the insignificance of the extracted rules even if they were supported in 100% of cases. Nevertheless, the detailed analysis reported some significant associations such as the Adapter, Decorator, Proxy and State patterns are positively related with the presence of defects, while the presence of the Visitor pattern is associated with the absence of defects. The Visitor case contradicts our findings reported in Sec. 9.2.1 and contradicts our findings for all other patterns in this section.

The case of Visitor is unique: among 38 instances, 12 of them are defective and 26 are defect-free. All of them come from a single system, velocity-1.6, and are located in a single package `org.apache.velocity.runtime.parser.node`. All instances represent objects that visit and parse a specific type of a node. As the amount of the code inside the Visitor pattern is minimal, and its logic is clear and simple, no defects were reported for the majority of those instances. In the remaining classes, defects were cosmetic or related to special cases which have not been covered.

After introducing the effect of smells to our analysis, the results showed that in the case of Adapter, State and Visitor, smelly patterns are positively associated with the presence of defects. However, the Decorator and Proxy patterns require the investigation to be replicated on a larger dataset: although the presence of smells in the pattern classes was associated with defects in 100% of the cases, the small number of smelly classes (1 for Decorator and 3 for Proxy) invalidated the FET results. Furthermore, we found that the non-smelly Visitor classes are associated with the absence of smells. These results are consistent with the findings reported in Sec. 9.2.1.

On the other hand, for both the Decorator and Proxy patterns, the results suggest that smell-free pattern classes are also associated with the presence of defects, which compels us to conduct further investigation. For the other patterns, the small number of detected instances prevented us from extracting any rules.

For the Decorator pattern, the majority of defective non-smelly instances belong to two systems, Xalan-2.7 and Camel-1.6. In both systems the evolution of the Decorator pattern scattered its functionalities into many small objects representing crosscutting concerns, which in turn became hard to comprehend and maintain, and as a consequence, produced defects.

With regards to the Proxy pattern, there are 23 defective non-smelly instances, and 16 of them belong to Xalan-2.7. Almost all Visitor instances in this system belong to a single package `org.apache.xalan.xsltc.compiler`. Those instances parse specific types of instructions before passing the parsed

segments to a converter object. Those instances extend a single parent class, `Instruction`, and their tight coupling with this shared parent causes them to also share the same defects.

9.4.5 The distribution of defects

The results of the detailed analysis are consistent with our findings reported in 9.2.2. They indicate that the Adapter, Decorator, Prototype, Proxy and State patterns are linked with more defects than non-pattern classes. While we could not extract any rules for other patterns, no extracted rules in any of the patterns contradicted our findings. The introduction of smells into our analysis resulted in the observation that, in case of the Adapter, Bridge, Observer, Singleton, State and Visitor patterns, smelly patterns attract more defects than non-smelly patterns. For all other patterns, we could not extract any rules that contradict our findings.

With respect to the effect of smells on the relationship between patterns and defects we found that for the majority of pattern types (Adapter, Bridge, Observer, Prototype, Proxy, Singleton, State and Visitor), the smelly design patterns attracted more defects than non-pattern classes. The effect size analysis reported that the mean difference between those smelly patterns and non-pattern classes is greater than the 0.8 of standard deviation, which indicates the large significance of those extracted rules. For the non-smelly patterns, we found that only in the case of Adapter, Decorator, Proxy and State did the non-smelly patterns attract more smells than the non-pattern classes and those extracted rules have a small significance. Moreover, in case of the Visitor pattern, we concluded that non-smelly patterns have fewer defects than non-pattern classes.

The defect proneness of specific smells and patterns has also been studied in the literature. Our findings partially confirm the results reported by Aversano et al. [14], Vokac [106] and [88] with respect to the Singleton and Observer patterns and their positive association with defects. Our results show that for both of them the presence of smells additionally amplified the defect proneness of the affected classes. For other patterns, the results differ, which may also indicate the confounding role of code smells addressed in our work.

9.5 Conclusion

In this experiment we investigated the links between design patterns and defects, and how the presence/absence of smells affects these relationships. Our analysis included 10 small- and medium-size Java systems. The findings suggest that pattern classes are associated with more defects than non-pattern classes, and that smells could be considered as a contextual factor in this relationship since smelly pattern classes attract more defects than both non-smelly pattern and non-pattern classes.

The experiment findings are three-fold:

- Investigating the binary relationship between patterns and defects showed that patterns are positively associated with the presence of defects, thus validating the results reported in previous studies. However, by including the presence of smells as a confounding variable in this relationship, our results indicate that only smelly patterns have a unanimously positive association with defects, while non-smelly patterns delivered mixed results.

- Our results show that pattern classes have a greater number of defects than non-pattern classes. Introducing the effect of smells into the analysis reveals that smelly classes attract more defects than non-smelly classes, and that both smelly- and non-smelly pattern classes have, in a different rate, a higher defect distribution than non-pattern classes. The findings also suggest that the relationship between smelly patterns and defects is more significant than the relationship between non-smelly patterns and defects.

- The relationship between specific patterns and defects varies, both in terms of the binary and quantitative relationships. This variation still holds true if smells are introduced into this relationship.

  Nevertheless, there are some common findings between all the patterns. For example, our analysis did not reveal a pattern that attracts a lower number of defects than non-pattern classes. In contrast, the Adapter, Decorator, Prototype, Proxy and State patterns tend to have a higher defect distribution than non-pattern classes. The introduction of smells into the analysis showed that the majority of smelly pattern classes attract more defects than non-pattern classes, and that non-smelly patterns attract more or fewer defects, depending on their type. For example, the non-smelly Adapter, Decorator, Proxy and State classes attract more defects than the non-pattern classes. On the other hand, smelly Visitor classes are linked with a lower defect distribution than non-pattern classes.

  What is also noticeable in our results is that no non-smelly pattern attracts a higher defect number than a smelly pattern. On the contrary, smelly Adapter, Bridge, Observer, Singleton, State and Visitor classes tend to have more defects than non-smelly pattern classes.

  Furthermore, the binary association between different patterns and defects also varies between patterns. The Adapter, Decorator, Proxy and State patterns are associated with the presence of defects, while the Visitor pattern is associated with the absence of defects. Taking into consideration the effect of smells on the previous findings showed that the majority of smelly patterns tend to have positive associations with defects, but with a different confidence and significance.

The results, albeit preliminary, can inspire and foster further research on the contextual factors that affect defect-proneness, changeability and other import-

ant software properties. Understanding their role may help in isolating their individual impact and the interactions they play a role in.

The findings can have an impact on the development of practice. Design patterns promote good practices. However, if pattern classes are affected by code smells, the advantages of patterns could be challenged by defects resulting from their interaction with smells. Therefore, we conclude that preventing and removing code smells may reduce the defect-proneness of the code, so we advise programmers to take this possibility into account.

10 What is the impact of code smells on the relationship between design patterns and changeability

The relationships between design patterns or code smells on one hand and changeability on the other hand were studied in the literature. However, the interaction effect between patterns and smells on changeability was not investigated. As design patterns and code smells represent different design concepts, our hypothesis that the presence, absence or interaction between the two phenomena can affect the code changeability. To study that, we conducted an experiment to analyze these properties and their impact on two change-related metrics: frequency and change size. The experiment was performed on three medium size, long evolving Java systems with regard to 13 design patterns and 9 code smells.

10.1 Experimental design

10.1.1 Questions

The experiment reports the individual impact of patterns or smells on two change-related metrics frequency and change size and how the interaction between the two studied phenomena affects the change related metrics. Specifically, the experiment answers the following questions:

1. **EXP3-RQ1** How the presence, absence and interaction between design patterns and code smells in a class affect the frequency of changes made to this class?

2. **EXP3-RQ2** How the presence, absence and interaction between design patterns and code smells in a class affect the change size?

3. **EXP3-RQ3** How the presence, absence and interaction between *specific* design patterns and *specific* code smells in a class affects both change-related metrics (size and frequency)?

10.1.2 Notation

In addition to the notation defined in Sec 7. We define the following notation:

- (*rel*) of the file (F) : is a sequence of revisions from rev_{N+1} to rev_{rel}, where N is a tagged revision included in the previous release. Thus, *rel* is the tag attached to the last revision in the release.

10.1.3 Analyzed systems

We analyze three small- and medium-size open source Java systems; JHotDraw (JHD), ArtOfIllussion (AOI) and JEdit (JE). Those systems have evolved for a long time and had several public releases. Additionally, they are also curated under the Qualitas Corpus[9] umbrella.

JHotDraw[10] is a framework for developing structured editors of 2D graphics. It started in 1996 as a playground project for implementation of design patterns. However, it underwent a major rewrite since release 7.0. The most recent public release is 7.6, which includes 679 classes and 80 kLOC. Data has been collected for 9 releases of the system, which span over 648 revisions.

ArtOfIllusion[11] is a tool for modeling, processing and rendering images from scene-describing files. We analyze 16 releases of the system, from 2.4.1 to 3.0.2, which include 426 revisions; the 3.0.2 release includes 500 classes and 118 kLOC.

JEdit[12] is a highly-customizable text editor, with more than 150 specialized plug-ins. In this experiment we analyze 11 releases of the system, from 4.0 to 5.4, that span over 21207 revisions.

10.1.4 Analyzed smells, patterns and change-related metrics

We studied 13 design patterns; Factory Method, Prototype, Singleton, Composite, Decorator, Proxy, Adapter-command, Observer, State-strategy, Chain Of Responsibility, Visitor, Bridge and Template Method. Information about the chosen patterns can be found in Sec 4. The pattern detection strategy and tool are presented in Sec 4.2.

For smells, we analyzed 9 code smells:Data class, External duplication, Data clumps, Feature envy, Internal Duplication, Tradition Breaker, God class, Schizophrenic class and Message chains. Information about the smells in Sec 5. The smells algorithm and tool are presented in Sec 5.2.

With regards to changeability, we consider two metrics of changeability: *change frequency* (change-proneness) and *change size*.

As a proxy construct for measuring change size, we used code churn [83] (*CHURN*), defined as a sum of lines added and lines deleted in all revisions *rev* of the file f that belong to a given *release*.

[9] `http://qualitascorpus.com`
[10] `http://jhotdraw.sf.net`
[11] `http://aoi.sf.net`
[12] `http://www.jedit.org`

$$CHURN(f_{release(f)}) = \sum_{rev \in f_{release}} lines_{added}(rev) + lines_{deleted}(rev)$$

Churn is a cumulative measure, and may be biased by the size of the subject class and the number of revisions in the analyzed release. To address this, we adjusted the code churn values for both class size and the number of revisions in the respective release. Our metric reflects the size of an average change made to a single line of code in a single revision of the subject class.

The other metric, *FREQ*, counts the revisions in the repository log that affected the given class. This metric is commonly used as a primary measure of the change-proneness of the source code [58, 64]. In response to the issues mentioned for the change size, we adjusted the metrics for the number of revisions in the subject release.

Both metrics capture different dimensions of changeability and are language-agnostic, which can help in replicating this experiment in different settings. It is also worth mentioning that in this analysis we consider all changes to be equal, ignoring their cause and the maintenance activity that triggered them: corrective, perfective, adaptive or preventive [76].

To improve the readability of the results, the presented values of *CHURN* and *FREQ* have been multiplied by a factor of 100.

10.1.5 Matching patterns, smells and change metrics

First we identified the intersections between the S and DP sets. Similar to the previous experiments, there was a mismatch in granularity as patterns usually involve a number of classes, while smells affect a method or a class. To address that, we adjusted the granularity of both sets to the class level: method-level smells have been assigned to the classes that include the subject methods, and design patterns are consistently assigned to one of the classes playing one of the roles. Then, the fully-qualified class names (i.e., including their package names) in sets ALL, S and DP are textually matched to produce the following respective datasets SDP, $nSDP$, $SnDP$, and $nSnDP$.

After identifying those datasets, we matched their classes to the change related metrics defined in sec 10.1.4. It is important to address that we do not analyze classes as single datapoints, as they could change their statuses (S/nS) or (DP/nDP) in time. Instead, we consider releases as sequences of revisions, in which the subject classes have not changed their (S/nS) or (DP/nDP) statuses. In order to validate this approach, we manually analyzed a stratified sample of the classes that changed their status and conducted statistical analysis on the entire dataset. Results indicate that releases can be effectively used as an approximation for such sequences

10.2 Results

For every question, the results of the tests together with the direct findings from those results are presented. A detailed justification behind the results is presented in sec 10.3 To answer the first and the second questions, we applied the same following procedure:

(1) Identify all public releases (*Rel*) of the system.

(2) For each release (*Rel*), identify sets S and DP by using the respective detectors for code smells and design patterns.

(3) For each release (*Rel*), identify datasets *SDP*, *nSDP*, *SnDP*, and *nSnDP* as respective intersections of S and DP.

(4) For each class (C) in each *Rel*, collect *CHURN* and *FREQ* values.

(5) For each analyzed dataset, test if its distribution is normal, using Shapiro-Wilk test [98].

(6) Test if the datasets have the same distribution (with respect to the subject metric *CHURN* or *FREQ*).

(7) If the distributions are different, perform post-hoc tests to identify the pair-wise relationships between the datasets.

10.2.1 How the presence, absence and interaction between design patterns and code smells in a class affect the frequency of changes made to this class?

Following the procedure presented in 10.2, first we identify the public releases *Rel* of all the systems; the following releases of the subject systems were identified. For JHD: 5.4b1, 6.0b1, 7.0.9, 7.2, 7.3, 7.3.1, 7.4.1, 7.5.1, 7.6. For AOI: 2.4, 2.4.1, 2.5, 2.5.1, 2.6, 2.6.1, 2.7, 2.7.1, 2.7.2, 2.8, 2.9, 2.9.1, 2.9.2, 3.0, 3.0.1,and 3.0. For JE: 4.0, 4.1, 4.2, 4.3, 4.4, 4.5, 5.0, 5.1, 5.2

The choice of releases was conditioned by the availability of proper revision tags in the repository.

System	Number of releases	Number of classes
JHD	9	5137
AOI	16	1305
JE	9	4044

Table 32: Statistics regarding the size of the experiment

Next, in Table 33 we present descriptive statistics of *S*, *nS*, *DP*, *nDP*, *SDP*, *nSDP*, *SnDP* and *nSnDP* datasets for *FREQ* metric of all the the systems.

	JHD			AOI			JE		
Dataset	N	μ	$\widetilde{\mu}$	N	μ	$\widetilde{\mu}$	N	μ	$\widetilde{\mu}$
S	838	3.979	3.279	328	4.856	2.500	552	0.687	0.188
nS	4299	3.653	2.703	977	3.406	2.222	3492	0.556	0.188
DP	808	4.119	3.279	391	4.081	2.469	754	0.717	0.188
nDP	4329	3.629	2.703	914	3.638	2.326	3290	0.542	0.188
SDP	167	4.424	3.289	193	4.931	3.704	175	0.976	0.188
SnDP	671	3.868	2.703	135	4.748	2.326	377	0.552	0.188
nSDP	641	4.039	3.226	198	3.251	2.326	579	0.639	0.188
nSnDP	3658	3.585	2.703	779	3.445	2.222	2913	0.540	0.188

Table 33: Descriptive statistics for *FREQ* in all the systems. N is the number of classes. $\widetilde{\mu}$ is the median and μ is the mean value

In Table 34, the results of the Shapiro-Wilk test of normality are presented for the values of *FREQ* metric. Based on them, we conclude that none of the datasets in all the systems is normally distributed, which affects the subsequent analyses.

	JHT				AOI				JE			
Dataset	W_{crit}	σ	W	p	W_{crit}	σ	W	p	W_{crit}	σ	W	p
S	0.996	2.921	0.835	≈ 0	0.991	4.677	0.693	≈ 0	0.995	2.339	0.243	≈ 0
nS	0.999	2.871	0.796	≈ 0	0.996	2.871	0.796	≈ 0	0.999	1.589	0.295	≈ 0
DP	0.996	3.330	0.803	≈ 0	0.992	3.731	0.717	≈ 0	0.996	2.305	0.265	≈ 0
nDP	0.996	2.783	0.806	≈ 0	0.996	4.020	0.597	≈ 0	0.999	1.542	0.293	≈ 0
SDP	0.983	3.298	0.830	≈ 0	0.985	4.238	0.727	≈ 0	0.984	3.669	0.231	≈ 0
SnDP	0.995	2.811	0.835	≈ 0	0.980	5.256	0.644	≈ 0	0.992	1.316	0.378	≈ 0
nSDP	0.995	3.337	0.791	≈ 0	0.986	2.941	0.704	≈ 0	0.995	1.687	0.336	≈ 0
nSnDP	0.999	2.776	0.799	≈ 0	0.996	3.735	0.591	≈ 03	0.999	1.569	0.285	≈ 0

Table 34: Results of Shapiro-Wilk test of normality for *FREQ* values for all the systems

Next, we tested if the datasets have the same distribution. Since they are not normally distributed, for each system we used Kruskal-Wallis non-parametric test with the following hypotheses:

- *H0*: values in *SDP*, *nSDP*, *SnDP* and *nSnDP* datasets (for *FREQ* metric) have the same distribution,
- *H1*: values in the datasets have different distributions.

Datasets for *FREQ* metric	H_{crit}	H	p-value
JHD	7.815	24.147	$< .050$
AOI	7.815	63.798	$< .050$
JE	7.815	9.388	$< .050$

Table 35: Results of the Kruskal-Wallis test for *FREQ* values in all the systems

Results presented in Table 35 show that in all systems the H0 is rejected, which indicates that distributions of datasets are different with respect to *FREQ* metric.

Following that and to identify the specific pair-wise relationships between datasets, we applied the post-hoc, non-parametric Wilcoxon-Mann-Whitney (WMW) test to all pairs of datasets. As an example, below we present the hypotheses, reasoning process and conclusions based on the comparison of *SDP* and *nSnDP* with respect to the *FREQ* metric in JHD system.

We formulate the following hypotheses:

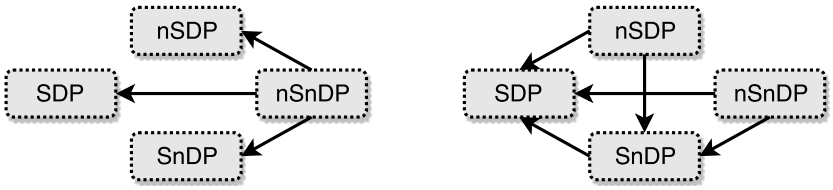
1. H0: $nSnDP = SDP$ w.r.t. *FREQ*
2. Ha: $nSnDP$!= SDP w.r.t. *FREQ*
3. Ha1: $nSnDP < SDP$ w.r.t. *FREQ*
4. Ha2: $nSnDP > SDP$ w.r.t. *FREQ*

The compared datasets are large enough (N=3658 for *nSnDP* and N=167 for *SDP*) to use Z-value instead of the W statistic. The result of a two-tailed non-parametric Wilcoxon test at $\alpha = 0.05$ ($z = -3.63, p < 0.00$) indicates that H0 should be rejected and based on the result of the one-tail test, and by comparing the medians (2.703 for *nSnDP* and 3.289 for *SDP*), we conclude that Ha1 should be accepted instead.

We applied this procedure to test all pairs of datasets in all the systems. Results of WMW tests and conclusions are presented in Tables 36 and 37.

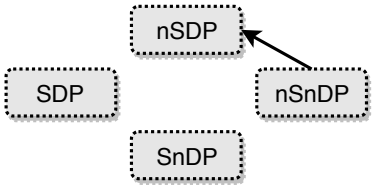
	JHD	AOI
Datasets	Result and conclusion	Result and conclusion
DP/nDP	$z = -3.40, p = .007 \Rightarrow DP > nDP$	$z = -3.90, p = .001 \Rightarrow DP > nDP$
S/nS	$z = -3.68, p < .001 \Rightarrow S > nS$	$z = -7.47, p < .001 \Rightarrow S > nS$
nSDP/SDP	$z = -1.88, p = .060 \Rightarrow$ H0 not rejected	$z = -6.00, p < .001 \Rightarrow SDP > nSDP$
nSDP/nSnDP	$z = -2.64, p = .004 \Rightarrow nSDP > nSnDP$	$z = -2.86, p = .004 \Rightarrow SnDP > nSDP$
nSDP/SnDP	$z = 0.15, p = .880 \Rightarrow$ H0 not rejected	$z = -0.52, p = .603 \Rightarrow$ H0 not rejected
SnDP/SDP	$z = -1.89, p = .058 \Rightarrow$ H0 not rejected	$z = -2.673, p = .003 \Rightarrow SDP > SnDP$
SnDP/nSnDP	$z = -2.95, p = .003 \Rightarrow SnDP > nSnDP$	$z = -7.56, p < .001 \Rightarrow SDP > nSnDP$
SDP/nSnDP	$z = -3.63, p < 0.001 \Rightarrow SDP > nSnDP$	$z = -3.01, p = .002 \Rightarrow SnDP > nSnDP$

Table 36: Results of pair-wise WMW tests for *FREQ* metric in JHD and AOI



(a) JHD

(b) AOI



(c) JE

Figure 10: Relationships identified among datasets for *FREQ* metric in all the systems. The arrows are pointing to the groups with the biggest *FREQ*

	JE
Datasets	Result and conclusion
DP/nDP	$z = -2.95; p = .003 \Rightarrow DP > nDP$
S/nS	$z = -1.17; p = .242 \Rightarrow$ H0 not rejected.
nSDP/SDP	$z = -0.22, p = .826 \Rightarrow$ H0 not rejected.
nSDP/nSnDP	$z = 1.22, p = .222 \Rightarrow$ H0 not rejected.
nSDP/SnDP	$z = -2.62, p = .008 \Rightarrow nSDP > nSnDP$
SnDP/SDP	$z = -1.11, p = .267 \Rightarrow$ H0 not rejected.
SnDP/nSnDP	$z = -1.77, p = .077 \Rightarrow$ H0 not rejected.
SDP/nSnDP	$z = -0.81, p = .418 \Rightarrow$ H0 not rejected.

Table 37: Results of pair-wise WMW tests for *FREQ* metric in JE

All identified relationships (with regard to *FREQ* metric) among the four intersected datasets are depicted as arrows in Fig. 10. Based on Fig. 10, we conclude that classes in *nSnDP* dataset have the lowest values of *FREQ*, compared to all other datasets.

10.2.2 How the presence, absence and interaction between design patterns and code smells in a class affect the change size?

Following the procedure presented in 10.2, in Table 38 we present descriptive statistics of *S*, *nS*, *DP*, *nDP*, *SDP*, *nSDP*, *SnDP* and *nSnDP* datasets for *CHURN* metric of all the systems.

	JHD			AOI			JE		
Dataset	N	μ	$\widetilde{\mu}$	N	μ	$\widetilde{\mu}$	N	μ	$\widetilde{\mu}$
S	838	0.868	0.302	328	0.611	0.170	552	0.124	0.021
nS	4299	1.624	0.575	977	1.487	1.485	3492	0.144	0.038
DP	808	2.020	0.793	391	1.249	0.549	754	0.152	0.032
nDP	4329	1.404	0.491	914	1.274	0.841	3290	0.139	0.035
SDP	167	1.113	0.561	193	0.532	0.156	175	0.113	0.018
SnDP	671	0.807	0.283	135	0.724	0.194	377	0.129	0.024
nSDP	641	2.257	0.872	198	1.947	1.612	579	0.163	0.040
nSnDP	3658	1.513	0.547	779	1.370	1.359	2913	0.140	0.037

Table 38: Descriptive statistics for *CHURN* in all the systems. N is the number of classes. $\widetilde{\mu}$ is the median and μ is the mean value

Next, in Table 39 we present the results of Shapiro-Wilk test of normality for the values of *CHURN* metric. Based on them, we also conclude that none of the datasets in all the systems is normally distributed.

	JHT				AOI				JE			
Dataset	W_{crit}	σ	W	p	W_{crit}	σ	W	p	W_{crit}	σ	W	p
S	0.996	1.224	0.679	≈ 0	0.991	1.323	0.363	≈ 0	0.995	0.288	0.422	≈ 0
nS	0.999	3.755	0.332	≈ 0	0.996	2.454	0.441	≈ 0	0.999	0.348	0.354	≈ 0
DP	0.996	4.954	0.309	≈ 0	0.992	2.473	0.380	≈ 0	0.996	0.441	0.275	≈ 0
nDP	0.996	3.122	0.349	≈ 0	0.996	2.158	0.455	≈ 0	0.999	0.313	0.401	≈ 0
SDP	0.983	1.371	0.773	≈ 0	0.985	0.760	0.691	≈ 0	0.984	0.289	0.359	≈ 0
SnDP	0.995	1.178	0.649	≈ 0	0.980	1.849	0.292	≈ 0	0.992	0.287	0.445	≈ 0
nSDP	0.995	5.495	0.315	≈ 0	0.986	3.248	0.391	≈ 0	0.995	0.477	0.266	≈ 0
nSnDP	0.999	3.347	0.350	≈ 0	0.996	2.194	0.468	≈ 0	0.999	0.316	0.395	≈ 0

Table 39: Results of Shapiro-Wilk test of normality for *CHURN* values for all the systems

Following that, we tested if the analyzed datasets have the same distribution with regards to *CHURN* metric. Since they are not normally distributed, we also used Kruskal-Wallis non-parametric test for every system with the following hypotheses:

- *H0*: values in *SDP*, *nSDP*, *SnDP* and *nSnDP* datasets (for *CHURN* metric) have the same distribution,
- *H1*: values in the datasets have different distributions.

Datasets for *FREQ* metric	H_{crit}	H	p-value
JHD	7.815	105.656	≈ 0
AOI	7.815	128.846	$< .050$
JE	7.815	11.751	$< .050$

Table 40: Results of the Kruskal-Wallis test for *CHURN* values in all the systems

Results presented in Table 40 show that H0 is rejected in all the systems, which indicates that the distributions of datasets are different with regards to *CHURN* metric. To identify the specific pair-wise relationships between datasets, we applied the post-hoc, non-parametric Wilcoxon-Mann-Whitney (WMW) test and the results are presented in Table 41 and Table 42

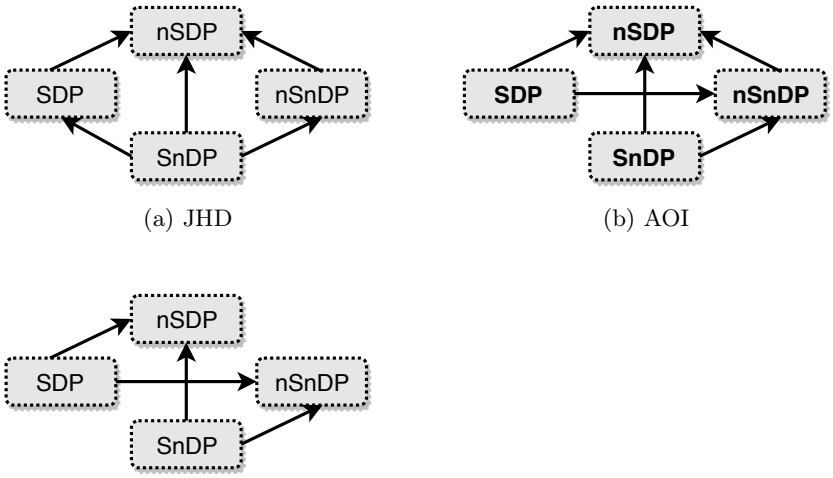
	JHD	AOI
Datasets	Result and conclusion	Result and conclusion
DP/nDP	$z = -4.65, p < .001 \Rightarrow DP > nDP$	$z = 1.18, p = .238 \Rightarrow$ H0 not rejected
S/nS	$z = 8.91, p < .001 \Rightarrow nS > S$	$z = 10.46, p < .001 \Rightarrow nS > S$
nSDP/SDP	$z = 3.68, p < .001 \Rightarrow nSDP > SDP$	$z = 9.61, p < .001 \Rightarrow nSDP > SDP$
nSDP/nSnDP	$z = -4.57, p < .001 \Rightarrow nSDP > nSnDP$	$z = -4.34, p < .001 \Rightarrow nSDP > nSnDP$
nSDP/SnDP	$z = -9.52, p < .001 \Rightarrow nSDP > SnDP$	$z = 7.764, p < .001 \Rightarrow nSDP > SnDP$
SnDP/SDP	$z = -2.17, p = .030 \Rightarrow SDP > SnDP$	$z = -1.28, p = .200 \Rightarrow$ H0 not rejected
SnDP/nSnDP	$z = 8.44, p < .001 \Rightarrow nSnDP > SnDP$	$z = 5.70, p < .001 \Rightarrow nSnDP > SnDP$
SDP/nSnDP	$z = 1.7, p = .089 \Rightarrow$ H0 not rejected	$z = 8.08, p < .001 \Rightarrow nSnDP > SDP$

Table 41: Results of pair-wise WMW tests for *CHURN* metric in JHD and AOI

	JE
Datasets	Result and conclusion
DP/nDP	$z = -0.16; p = .873 \Rightarrow$ H0 cannot be rejected.
S/nS	$z = 3.18; p < .002 \Rightarrow nS > S$
nSDP/SDP	$z = 2.69, p = .007 \Rightarrow nSDP > SDP$
nSDP/nSnDP	$z = -1.03, p = .300 \Rightarrow$ H0 not rejected.
nSDP/SnDP	$z = 2.45, p = .014 \Rightarrow nSDP > SnDP$
SnDP/SDP	$z = 0.75, p = .450 \Rightarrow$ H0 not rejected.
SnDP/nSnDP	$z = 2.13, p = .033 \Rightarrow nSnDP > SnDP$
SDP/nSnDP	$z = 2.37, p = .018 \Rightarrow nSnDP > SDP$

Table 42: Results of pair-wise WMW tests for *CHURN* metric in JE

All identified relationships (with regard to *CHURN* metric) among the four intersected datasets are depicted as arrows in Fig. 11 and based on the Fig. 11, we conclude that churn is the highest in *nSDP*.



(a) JHD (b) AOI

(c) JE

Figure 11: Relationships identified among datasets for *CHURN* metric in all the systems. The arrows are pointing to the groups with the biggest *CHURN*.

10.2.3 How the presence, absence and interaction between specific design patterns and specific code smells in a class affect both change-related metrics (size and frequency)?

In this section we present results for specific smells and patterns. Due to a different data collection and analysis scheme, we followed a different procedure. The procedure is described in the following steps:

(1) Identify all public releases *Rel* of the system.

(2) For each specific code smell (S_{smell}) in each release (*Rel*), identify classes that are simultaneously involved in design patterns (set SDP_{smell}) or not involved (set $SnDP_{smell}$) in design patterns.

(3) For each specific design pattern DP_m, in each public release identify classes that are simultaneously affected (set $SDP_{pattern}$) or not affected by any code smells (set $nSDP_{pattern}$).

(4) Repeat steps 5 and 7 from the procedure described in sec 10.2.

10.2.3.1 Specific patterns

In Table 43 we present descriptive statistics of *CHURN* and *FREQ* metrics for classes involved in specific patterns and affected/not affected by code smells. Numbers in parentheses indicate smelly/non-smelly instances. There were no instances of Visitor and Chain Responsibility patterns, so they have been removed.

Pattern DP_i	*FREQ*				*CHURN*			
	$nSDP_i$		SDP_i		$nSDP_i$		SDP_i	
	$\widetilde{\mu}$	μ	$\widetilde{\mu}$	μ	$\widetilde{\mu}$	μ	$\widetilde{\mu}$	μ
Composite (17/34)	3.704	5.195	3.636	4.066	0.831	2.188	0.207	0.757
Prototype (9/100)	4.301	4.970	3.289	3.819	1.197	2.451	0.381	0.778
Singleton (21/121)	0.595	1.318	0.281	4.338	0.144	0.547	0.154	0.493
State-Strategy (246/672)	1.974	3.088	2.524	3.688	0.500	1.960	0.199	0.724
Template Method (58/144)	1.276	2.257	0.564	2.361	0.162	0.985	0.069	0.410
Decorator (11/84)	2.225	3.545	2.222	3.649	1.574	2.531	0.163	1.049
Proxy (37/26)	1.235	2.139	2.469	4.842	0.133	0.639	0.138	0.428
Adapter-Command (252/499)	1.235	2.770	2.326	3.720	0.198	1.334	0.098	0.458
Observer (42/87)	1.235	2.962	2.274	4.162	0.383	1.233	0.118	0.422
Visitor (0/4)	1.724	2.019	n/a	n/a	0.537	1.206	n/a	n/a
Bridge (21/163)	2.703	4.020	4.651	5.897	0.818	2.309	0.163	0.706
Chain of Responsibility (0/4)	1.659	1.486	n/a	n/a	0.274	0.825	n/a	n/a
Factory Method (14/59)	1.149	1.999	4.234	3.619	0.421	1.921	0.177	0.860

Table 43: Descriptive statistics for *CHURN* and *FREQ* metrics in SDP_i and $nSDP_i$ datasets. Numbers next to pattern name indicate smelly/non-smelly instances. $\widetilde{\mu}$ is the median and μ is the mean value.

First, we performed WMW test to identify the pair-wise relationships between specific patterns and the non-pattern classes (Table 44).

Design pattern DP_i	CHURN (nDP/DP_i)	FREQ (nDP/DP_i)
State/Strategy	$nDP < DP_i$ (p<.001)	$nDP < DP_i$ (p<.001)
Visitor	H0 not rejected (p=.360)	H0 not rejected (p=.737)
Adapter/Command	$nDP > DP_i$ (p=.028)	$nDP < DP_i$ (p=.006)
Singleton	$nDP > DP_i$ (p=.008)	$nDP > DP_i$ (p<.001)
Prototype	$nDP < DP_i$ (p<.001)	$nDP < DP_i$(p=.001)
Observer	H0 not rejected (p=.468)	H0 not rejected (p=.115)
Decorator	$nDP < DP_i$ (p=.001)	$nDP < DP_i$ (p<.001)
Factory Method	$nDP < DP_i$ (p=.001)	H0 not rejected (p=.914)
Bridge	$nDP < DP_i$ (p<.001)	$nDP < DP_i$ (p<.001)
Chain of Responsibility	H0 not rejected (p=.678)	H0 not rejected (p=.894)
Template Method	$nDP > DP_i$ (p=.003)	H0 not rejected (p=.263)
Proxy	H0 not rejected (p=.453)	$nDP < DP_i$ (p=.005)
Composite	$nDP < DP_i$ (p=.003)	$nDP < DP_i$ (p<.001)

Table 44: Results of WMW tests for CHURN and FREQ metrics for specific design patterns

Next, we performed WMW test to identify the pair-wise relationships between smelly and non-smelly patterns (Table 45).

Pattern DP_i	*CHURN* ($nSDP_i/SDP_i$)	*FREQ* ($nSDP_i/SDP_i$)
Composite	$nSDP_i > SDP_i$ (p=.025)	H0 not rejected (p=.270)
Prototype	H0 not rejected (p=.128)	H0 not rejected (p=.262)
Singleton	H0 not rejected (p=.711)	H0 not rejected (p=1)
State-Strategy	$nSDP_i > SDP_i$ (p<.001)	$nSDP_i < SDP_i$ (p=.026)
Template Method	$nSDP_i > SDP_i$ (p=.047)	H0 not rejected (p=.271)
Decorator	$nSDP_i > SDP_i$ (p=.031)	H0 not rejected (p=.810)
Proxy	H0 not rejected (p=.496)	$nSDP_i < SDP_i$ (p=.001)
Adapter-Command	$nSDP_i > SDP_i$ (p<.001)	$nSDP_i < SDP_i$ (p<.001)
Observer	$nSDP_i > SDP_i$ (p=.003)	H0 not rejected (p=.066)
Visitor	(not found)	(not found)
Bridge	$nSDP_i > SDP_i$ (p=.018)	$nSDP_i < SDP_i$ (p=.012)
Chain of Responsibility	(not found)	(not found)
Factory Method	H0 not rejected (p=.140)	$nSDP_i < SDP_i$ (p<.001)

Table 45: Results of WMW tests of *CHURN* and *FREQ* metrics for SDP_i and $nSDP_i$ datasets

10.2.3.2 Specific smells

In Table 46 we reverse the settings and report results for specific smells in classes involved/not involved in design patterns. Like for patterns, we provide the number of classes involved/not involved in a pattern for each specific smell.

Code smell S_i	*FREQ*				*CHURN*			
	S_inDP		S_iDP		S_inDP		S_iDP	
	$\widetilde{\mu}$	μ	$\widetilde{\mu}$	μ	$\widetilde{\mu}$	μ	$\widetilde{\mu}$	μ
Data Clumps (12/68)	0.595	1.911	0.489	2.135	0.117	0.484	0.097	0.383
Schizophrenic Class (134/327)	2.703	3.518	2.703	3.914	0.159	0.464	0.162	0.740
Sibling Duplication (180/249)	4.255	4.733	3.636	4.781	0.238	0.714	0.257	0.830
Tradition Breaker (13/33)	4.255	4.582	3.279	4.314	0.427	0.788	0.588	1.074
External Duplication (36/252)	1.818	1.877	0.394	1.384	0.157	0.550	0.066	0.359
Internal Duplication (62/86)	3.279	4.604	4.545	6.493	0.225	0.894	0.118	0.379
Feature Envy (65/79)	2.368	3.959	3.704	5.996	0.133	0.616	0.099	0.424
God Class (223/152)	2.374	4.401	2.500	4.277	0.124	0.406	0.093	0.429
Data Class (12/68)	1.235	2.129	2.398	2.583	0.494	1.093	0.761	1.205

Table 46: Descriptive statistics for *CHURN* and *FREQ* metrics in S_iDP and S_inDP datasets. Numbers next to smell name indicate pattern/non-pattern instances. $\widetilde{\mu}$ is the median and μ is the mean value.

In Table 47 we present the results of WMW test to identify the pair-wise relationships between specific smells and the non-smelly classes.

Code smell S_i	CHURN (nS/S_i)	FREQ (nS/S_i)
Feature Envy	$nS > S_i$ (p<.001)	$nS < S_i$ (p<.001)
Schizophrenic Class	$nS > S_i$ (p=.003)	$nS < S_i$ (p<.001)
God Class	$nS > S_i$ (p<.001)	$nS < S_i$ (p<.001)
Tradition Breaker	$nS < S_i$ (p=.017)	$nS < S_i$ (p<.001)
External Duplication	$nS > S_i$ (p<.001)	H0 not rejected (p=.668)
Internal Duplication	H0 not rejected (p=.369)	$nS < S_i$ (p=0)
Sibling Duplication	$nS < S_i$ (p<.004)	$nS < S_i$ (p<.001)
Data Clumps	$nS > S_i$ (p<.001)	$nS > S_i$ (p<.001)
Data Class	$nS < S_i$ (p<.001)	H0 not rejected (p=.652)

Table 47: Results of WMW tests for CHURN and FREQ metrics for specific code smells

Next, in Table 48, we present the results of WMW test, performed to determine the pair-wise relationship between specific smells in the context of participating in any pattern.

Smell S_i	*CHURN* (S_inDP/S_iDP)	*FREQ* (S_inDP/S_iDP)
Data Clumps	H0 not rejected (p=.471)	H0 not rejected (p=.912)
Schizophrenic Class	H0 not rejected (p=.689)	H0 not rejected (p=.960)
Sibling Duplication	H0 not rejected (p=.675)	H0 not rejected (p=.653)
Tradition Breaker	H0 not rejected (p=.575)	H0 not rejected (p=.880)
External Duplication	$S_inDP > S_iDP$ (p=.018)	$S_inDP > S_iDP$ (p=.021)
Internal Duplication	$S_inDP > S_iDP$ (p=.008)	H0 not rejected (p=.065)
Feature Envy	H0 not rejected (p=.638)	$S_inDP < S_iDP$ (p=.012)
God Class	H0 not rejected (p=.327)	H0 not rejected (p=.787)
Data Class	H0 not rejected (p=.718)	H0 not rejected (p=.073)

Table 48: Results of WMW tests of *CHURN* and *FREQ* metrics for S_inDP and S_iDP datasets

10.3 Discussion

The first observation concerns the low number of classes with smells and patterns. As follows from Table 33 only 16.31% of classes in JHD, 25.13% in AOI and 13.65% in JE are affected by smells, and 15.73% in JHD, 29.92% in AOI and 18.65% in JE involve in a pattern. Moreover, there are only 3.27% in JHD, 1.48% in AOI and 4.33% in JE of classes with both smells and patterns. These values show that interactions between smells and patterns cannot be easily observed.

10.3.1 EXP3-RQ1 How the presence, absence and interaction between design patterns and code smells in a class affect the frequency of changes made to this class?

According to the literature, classes with code smells [64, 67] or involved in design patterns [18] are more change-prone than other classes, and our analysis confirms those results. If we consider only datasets DP/nDP, and S/nS, then the following relationships are true:

1. $FREQ(S) > FREQ(nS)$ (for JHD and AOI; for JE the H0 is not rejected)
2. $FREQ(DP) > FREQ(nDP)$ (for all the subject systems)

However, In this experiment we are particularly interested in observing results of interactions between code smells and patterns, and their impact on the changeability of the affected code.

Results concerning change frequency are presented in Fig. 10. Although not identical, they are consistent for all systems: a significant difference between datasets identified in one system is confirmed or, at least, not rejected in remaining systems.

As follows from the post-hoc tests, the *nSnDP* dataset either has the lowest change frequency among all datasets (in JHD), or is one of such datasets (in JE and AOI). In AOI, in turn, the *SDP* classes have the highest change frequency, followed by *SnDP* classes, and finally by *nSDP* or *nSnDP*. In other systems, *nSDP* classes change more frequently than in *nSnDP*. Additionally, the *nSnDP* dataset changes less frequently in JE than *nSDP*, which is the only significant relationship identified in this system. These observations indicate that change frequency is lowest in *nSnDP* classes and highest in *SDP*. The relationship between datasets of classes with either a smell or a pattern, i.e., *SnDP* and *nSDP* is unclear.

As the subject datasets are not normally distributed, we could not apply ANOVA method to quantitatively identify the effects of interaction between smells and patterns. However, we can observe that presence of smells and patterns has an additive effect on the change frequency of the affected classes: changes in *nSnDP* classes are less frequent than in classes that have either smells or patterns, but not both. Consequently, the *SDP* classes also appear to change more frequently than *nSDP* and *SnDP* classes.

Classes in the datasets with smells, i.e., *SDP* and *SnDP*, change more often than in the non-smelly datasets, regardless of the presence of design patterns (in JHD: *SDP* >*nSnDP* and *SnDP* >*nSnDP*; in AOI: *SDP* >*nSDP* *nSnDP* *SnDP*; in JE *nSDP* >*nSnDP*). The presence of design patterns becomes significant only if both subject datasets include smelly classes (in AOI: *SDP* >*SnDP*; in JHD and JE: *nSDP* >*nSnDP*). These observations suggest that the presence of code smells has a stronger impact on change frequency than the presence of design patterns, but this observation needs to be verified on a larger code sample.

10.3.2 EXP3-RQ2 How the presence, absence and interaction between design patterns and code smells in a class affect the change size?

Like for change frequency, the results for change size, aggregated for all smells and patterns vary across the subject systems (see Fig. 11). However, they are consistent: if a significant difference between subject datasets is identified in one system, it is confirmed or, at least, not rejected in remaining systems.

Changes for smelly classes S are smaller than for non-smelly classes nS (WMW test; for JHD: $z = 8.91, p < 0.001$; for AOI: $z = 10.46, p < 0.001$; and for JEdit: $z = 3.18; p < 0.002$;). A similar effect for smelly classes was previously reported by Counsell et al. [31]. They found that developers prefer to make only superficial changes to smelly classes, instead of applying a complex refactoring aimed at eradicating a root cause of the smell. The authors explained this effect by development economics: smaller changes require less effort, even though they usually need to be applied several times and bring only short-term gains. Our results also show that classes involved in *any* design pattern receive larger changes than non-pattern classes (for JHD: WMW test: $Z = -4.65, p < 0.001$). This effect could be attributed, e.g., to improved flexibility resulting from pattern implementation, which allows for applying more extensive updates in one go [28, 55, 60, 54]. However, this conclusion is not directly supported in AOI and JE, as the null hypothesis could not be rejected for them.

Investigating the interaction effect between the two code properties on the change size *CHURN*, we found that in AOI, *nSDP* classes received the largest changes, followed by *nSnDP* then by *SnDP* and *SDP*. For JHD, the *nSDP* received the largest changes, followed by *SDP* or *nSnDP*; on the other hand, changes made to *SnDP* classes were the smallest. In JE, classes in *nSDP* and *nSnDP* received larger changes than the other datasets. The aggregated results for all systems indicate that *nSDP* classes receive the largest changes, while changes in the *SnDP* dataset are the smallest (in JHD) or are among the smallest (in AOI and JE).

The presence of code smells also appears a stronger predictor of change size than the presence of patterns: classes in non-smelly datasets (i.e., *nSDP* and *nSnDP*) receive larger changes (in JHD and AOI: (*nSDP* >*SDP*, *nSDP* and *SnDP*) and in JE: (*nSDP* >*SDP* and *SnDP*)), and the presence of patterns is meaningful only if the compared datasets do not differ with respect to smells (in JHD: *nSDP* >*nSnDP* and *SDP* >*SnDP*; in AOI: *nSDP* >*nSnDP*).

10.3.3 EXP3-RQ3 How the presence, absence and interaction between *specific* design patterns and *specific* code smells in a class affect both change-related metrics (size and frequency)?

In Tables 44 and 47, we present results for specific patterns and smells. Specifically, we test if classes with a a pattern DP_i and a smell S_i, respectively, change more frequently or have larger changes than classes without any smell

(dataset nS) or any pattern (dataset nDP).

With regard to *FREQ*, if we consider specific patterns, we find that only Singleton classes appear to be less change-prone than classes that are not involved in any pattern. Classes participating in State-Strategy, Adapter-Command, Prototype, Decorator, Bridge, Proxy and Composite are in line with the aggregated results and change more frequently than non-pattern classes, while there is no difference in frequency of changes made to Visitor, Observer, Factory Method, Chain of Responsibility and Template Method classes, compared to non-pattern classes. For specific smells, only Data Clumps classes change less frequently than non-smelly classes, and there is no difference for External Duplication and Data Classes. Classes with other smells consistently exhibit increased change-proneness.

With regard to *CHURN*, the detailed analysis also identified three patterns to exhibit a different behaviour with respect to change size: Adapter-Command, Singleton and Template Method receive smaller changes than non-pattern classes. In our case, Singleton instances appear to be rather stable classes, each providing an instance of a global variable that is unlikely to be changed. For example, a JHD Singleton class `org.jhotdraw.gui.plaf.paletter.PaletterButtonUI` represents a ButtonUI for palette components. Throughout the release 7.6 it received only 2 changes comprising 6 lines, and the changes were merely cosmetic. Also classes participating in Template Method pattern are not intensively changed. A class `org.gjt.sp.jedit.gui.EditAction` represents a menu-installable command. It was modified only once, and only two lines were changed. These two cases could be explained by programming economics: rather than modifying the classes, programmers prefer to extend the system by adding new classes. This is in line with the Open-Closed principle [**?**], and is a recommended solution for adding new features to code. The Detailed results presented in Table 47 show that classes affected by Tradition Breaker, Sibling Duplication and Data Class received larger changes than non-smelly classes, which is also in conflict with the aggregated results for all smells. Specifically, Data Classes have not been found to be related with increased maintenance effort [**?**] or correlated with increased change proneness [64]. In fact, the `net.n3.nanoxml.XMLValidationException` class from JHD stores only a set of properties and performs no processing on it. As such, it has been modified only once, and most of its functionality was added in one revision. Additionally, the changes were larger than the average.

On the other hand, Tradition Breaker was found to increase change-proneness [64]. However, no clear pattern was identified for the change frequency of duplicates: they could be more stable, indifferent and more change-prone than other classes [**?**]. For example, an affected JHD class `org.jhotdraw.samples.svg.figures.SVGImageFigure` has one superclass and implements two interfaces. It underwent changes in 24 revisions, comprising 461 lines, which is well above the average for all classes. The changes affected both the overridden methods and other methods, usually by extending their functionality or adapting them to API variations. Also some changes made in the superclass had to be overridden in the subject class; this triggered additional updates, which could explain the

observed variations in the inherited contracts.

Next, we discuss the results reported when investigating the intersection effect between smells and patterns on change metrics; Results for the specific patterns presented in Table 45 show that classes participating in State-Strategy, Proxy, Adapter-Command, Bridge and Factory Method tend to change more frequently when they are affected by smells, while there are no major differences for other patterns.

As an example of Adapter/Command pattern, we can consider two classes from JHD: `org.jhotdraw.draw.DefaultDrawingEditor` (non-smelly) and `org.jhotdraw.draw.DefaultDrawingView` (affected by a God Class). Functionally they are similar (both are Commands to be run within a framework), and they play the role of a Subject (Receiver) in the Command pattern. The `DefaultDrawingView` class received 15 changes in releases 7.0.x, while the `DefaultDrawingEditor` was changed 7 times. Manual analysis revealed that actual responsibility of `DefaultDrawingView` is scattered and includes managing the view, read/write capabilities and handling incoming events. Changes made to this class throughout its life affected various functional parts of the class. However, even if the changes do not directly affect the pattern-related parts, they trigger subsequent fixes to all parts, which could explain the increased change frequency of the class. On the other hand, changes made to `DefaultDrawingEditor` that provides only the editing functions, were more systematic and focused, which is reflected in lower change proneness of that class.

Regarding the size of changes, *CHURN* values for smelly instances of Composite, State/Strategy, Template Method, Decorator, Adapter/Command, Observer and Bridge patterns are lower than for the non-smelly classes involved in the same patterns. To discuss that, we consider two classes participating in State-Strategy pattern: `org.jedit.gjt.sp.jedit.syntax.TokenMarker` representing a line-splitting element, affected by a God Class smell, and a smell-free `org.jedit.gjt.sp.jedit.ActionListHandler.TokenMarker` received smaller, but more frequent changes concerning its text-splitting capabilities implemented in two methods, `markTokens()` and `handleRuleStart()`. The changes were linked: extensions in one of them frequently resulted in subsequent fixes made to the other one, which affected the frequency and the size of changes. On the other hand, in `ActionListHandler` the main functionality has been included in a single method that was changed more cohesively and only once per extension. Based on these examples, we may conjecture that design patterns affected by some smells, e.g., ones related to dispersed responsibility assignment, display a different change behaviour than properly implemented patterns.

If we reverse the setting discussed above, i.e., analyze changes in smelly classes, depending if they are involved in a pattern or not (Table 48), no significant difference in change proneness exists. Only classes affected by External- and Internal Duplication smells receive smaller changes if they are also parts of design patterns. Additionally, classes with External Duplication also change less frequently. This observation corroborates findings reported by Mondal et al. [82], who noticed that clones are generally more unstable than other code,

subject also to other factors. Our results indicate that the presence of design patterns, which decreases both frequency and the change size of duplicated code, can be such a factor.

10.4 Conclusion

To answer the questions presented in this section, we conducted an exploratory study on the impact of code smells and design patterns and their interactions on two change-related metrics.

Both smells and patterns were found to affect change frequency and change size of the affected classes. Specifically, our results corroborate previous findings concerning the change proneness of smells and patterns. In general, classes with either code smells or design patterns were found to receive more changes than other classes. There are few exceptions: classes with Singleton pattern or Data Clumps code smell change less frequently than other classes. Moreover, the observed effect for all smells and patterns is additive, i.e., the effect is the largest for classes with both smells and patterns, and the smallest for classes without any of the elements. If both factors are considered, the impact of smells on change frequency is stronger than the one for design patterns.

With respect to change size, we observed the largest changes in classes with patterns, but without smells, while smelly classes not involved in patterns are subject to the smallest changes. Again, it is not unanimous, and some smells and patterns display a different behaviour: classes with Tradition Breaker, Sibling Duplication and Data Class smells have larger changes, while classes with Adapter-Command, Singleton and Template Method receive smaller changes.

Interactions among collocated smells and patterns may additionally affect their change frequency and change size. In particular, classes involved in some patterns are more sensitive to code smells with respect to changeability. That could provide some hints for programmers on how to prioritize classes for refactoring, based on the predicted changeability.

The results, albeit preliminary, foster the discussion concerning contextual factors that affect practical properties of software systems. Additionally, the identified impact of smells and patterns on changeability could be also practically exploited: it helps to improve predictions of change-proneness and to focus attention of developers on specific categories of classes, because they could change in a different way than other classes.

11 Thesis conclusion

The objectives of this thesis have been reached. We performed three experiments and concluded their outcome that helped us to answer the research questions. In the first experiment we investigated the relationship between design patterns and code smells; we found that pattern classes tend to be affected by fewer smells than other classes. We also found that the throughout the different releases of the analyzed systems the patterns classes are affected less frequently by smells

than other classes. Finally, we concluded that none of the analyzed patterns could be linked with the presence of any smells. On the other hand, we were able to extract significant pairwise relationships between specific patterns as antecedent and the absence of specific code smells as consequent. For example, Singleton pattern could be linked with the absence of God class and Data class smells. The list of the extracted rules could be found in Table 11 and a full discussion of the experiment's results can be found in Sec 8.6.

In the second experiment, we investigated the effect of code smells as a confounding factor in the relationship between design patterns and defects. Our results suggest that code smells could be considered as a contextual factor: smelly design pattern classes tend to attract a higher defect rates than other classes, while non-smelly pattern classes have no or slightly negative correlation with defects. Our analysis also shows which specific design patterns have positive relationship with defects or are affected more frequently by defects, and how the smelliness of those patterns affects those relationships. For example, the Adapter, Visitor and State patterns are positively related with the presence of defects only when they are affected by smells. Our results are discussed in detail in Sec 9.4.

The third experiment reported in this thesis investigates how the presence, absence and mutual interactions between patterns and smells affect the size and the frequency of the changes in the code. Our findings suggest that the frequency and the size of changes for pattern classes are bigger than for the non-pattern classed. On the other hand, the frequency of changes for smelly classes is higher than the frequency of changes for the non-smelly classes, while the size of changes is smaller for the smell-effected classes. The detailed analysis showed that the majority of both specific patterns and specific smells exhibit the same results. Regarding the churn (size of the changes), the churn for smelly classes is lower than the churn for the non-smelly classes.

When studying the interaction between patterns and smells, we found that classes with both smells and patterns receive smaller, but more frequent changes than other classes. Detailed analysis for specific patterns showed that the majority of patterns exhibit the same results, while specific code smells presented mixed results. The individual patterns and smells relationships with change metrics are listed and discussed in Sec 10.3.

12 Contributions

In this section, we summarize the contributions of this thesis for the practice and research.

12.1 Contributions for the research

- Our thesis investigated the relationship of design patterns with the presence of code smells and found that pattern classes tend to be affected by fewer smells than other classes. This finding is particularly important for

understanding the nature of both phenomena and their associations with other properties and structures in the code.

- Understanding the difference in evolution between smell-free and smelly pattern classes gives us another insight into how pattern classes change over time and how confounding variables, smells in this case, affect the default behaviour.
- Our findings support what several studies have already shown: pattern classes tend to have more defects than other classes. However, our findings assert that only smelly pattern classes have a positive relationship with defects, while non-smelly patterns have no or slightly negative relationship with defects. This is important to understand the circumstances which make the patterns more defect prone than other classes.
- The frequency and the size of changes which pattern classes receive through the evolution of a system is affected by confounding variables; in our case it is the presence of code smells in the subject classes. Similar to patterns, the size and the frequency of changes in classes which are affected by smells differ when those classes are part of a pattern and when they are not.
- The studied change-related metrics *CHURN* and *FREQ* are mostly independent. We conjecture that the two measures identify different aspects of change, and should be considered as complementary. By focusing solely on the change frequency, we may ignore an important aspect, describing the ability of a class to accept large chunks of code in a single commit.
- In two of our experiments, we studied code smells as a confounding variable in the pattern relationships with defects and with change-related metrics. In both cases we were able to identify such an effect. This suggest that confounding variable analysis should be considered when studying the relationship between code properties.

12.2 Contributions for the practice

- Automated smell detection: The presence of patterns could become another factor that automated detection tools consider as the link between patterns and smells can serve as a hint concerning the likely distribution of smells in a software system.
- When identifying the classes that participate in patterns, the code reviewers could focus their efforts on finding smells in the remaining parts of code, which could improve their productivity and effectiveness of the review.
- Defect prediction tools: Our findings suggest that mainly smelly design patterns have a positive relationship with defects, while non-smelly patterns have no or slightly negative relationship. Data about the distribution

of both patterns and smells and their intersections could become another input to incorporate.

- Programmers could be advised that preventing pattern classes from becoming smelly may reduce the defect-proneness of those classes.
- The identified impact of both patterns and smells and their interaction on changeability may help the programmers to improve the prediction of the change-proneness and to focus on specific categories of classes as they change differently than other classes.
- Classes involved in some patterns are more susceptible to code smells than others, and this interaction affects both the size and the frequency of their changes. This finding my help software developers in prioritizing classes for refactoring, based on their predicted changeability.

13 Limitations

In this section, we identify and discuss threats that affect the validity of results.

Construct validity is concerned with the definitions of the measured quantities and their relationships to the actual constructs. In our case, these threats apply to both independent and dependent variables.

- For the first experiment, the independent variables (IVs) indicate if individual classes are part of any pattern, while dependent variables (DVs) refer to classes with smells. The main issue is related to the improper assignment of the classes to the respective categories. Although the detectors we applied for identification of smells and patterns have high precision and recall, the reliability of the results directly depends on the accuracy of the detection process. While a manual verification of a small data sample did not reveal false positives, using other tools to identify the subject smells and patterns could alter the results. Additionally, we assumed that classes are affected by at most one smell and a single pattern. This simplification is not true in some cases and could have affected the results, e.g., collocated smells have been found to have even more detrimental impact on quality than individual smells [11]. Furthermore, the size of classes was not evaluated as the results were not normalized against the LOC of each class. Finally, for EXP1-RQ1 and EXP1-RQ2 we assumed that the different releases of the same system are independent. This assumption may not be accurate and could have affected the results.
- For the second experiment, together with the issue mentioned above related to the improper assignment of pattern and smell classes, and the lack of normalization against classes' LOC, we could also mention that the defects reported in the PROMISE dataset was adopted without any validation from our side.

- In the third experiment, the same threat related to the possible improper assignment of both pattern and smell classes is still valid. However, two more issues refer to definitions of the dependent variables should also be mentioned. We extracted the values of *CHURN* and *FREQ* using own-developed scripts, which have not been extensively tested and could be subject to defects. Moreover, the definitions of metrics are adjusted for the class size and the number of revisions within releases. That allows for comparing different classes, but may also bias the results. Furthermore, In our analysis, we ignore the cause of the changes and treat all changes equally. We are aware that the purpose of change could affect some of its properties, e.g., new features result in adding large portions of code, while bug fixes usually address and change only little fragments. This simplification limits our ability to explain why the code is modified.

Internal validity refers to the causal relationship between IVs and DVs.

Throughout the thesis, the IVs have been aggregated into disjoint datasets (*SDP*, *nSDP*, *DP* and *nDP*). We examined the relationships between those datasets and DVs. However, the collected evidence concerning these relationships is not unequivocal, as the extracted differences between subject datasets are insignificant in some systems. As a result, the observed variation in DVs could be also attributed to some unknown latent factors.

External validity is the extent, to which results could be extrapolated beyond the experimental setting.

The size and the number of the studied systems in all the experiments need to be highlighted here. In the first experiment we analyzed only two medium-size Java systems. In the second experiment, we collected data for 10 Java systems from the PROMISE repository [2] and in the third experiment we analyze three non-industrial Java systems of similar size, complexity, history and several other characteristics.

Additionally, we analyze only a subset of the known patterns and smells, and the total number of detected instances for some patterns is relatively small. Therefore, our conclusions should be interpreted carefully.

Conclusion validity is concerned with drawing invalid conclusions based on the collected evidence.

In the three experiments presented in this thesis, we analyzed the data using non-parametric tests due to the non-normal distributions of the data samples. That affects the results in two ways: first, the power of the tests is lower than for respective parametric tests and secondly, we could not determine the effect size of the identified differences among the subject datasets.

One more threat refers to the EXP1-RQ2, which is related to evaluating the trend of the ratio r. During the evolution of one of the subject systems the ratio r displayed positive monotonicity, whereas for the other system the trend could not be definitely determined, due to relatively high p-value. Based on that, evaluating the ratio r deserves further examination and the conclusion should be interpreted cautiously.

14 Future work

The presented results indicate that interactions between patterns and smells reveal interesting behaviour and can explain some of the contradictory results reported in the literature. However, we are aware that our findings need replicating and extending in several directions. We are planning to continue our current work with an objective to mitigate the limitations reported in Sec 13. In order to achieve that, we plan to look for answers to the following questions:

- Does the role played by a class in a design pattern affect its relationship with smells and, subsequently, its impact on both changeability and defects?
- What is the impact of *collocated* code smells affecting a single design pattern, with respect to the changeability and defect-proneness?
- Can we effectively use the simultaneous presence of smells and patterns as an effective predictor of changeability?

References

[1] Prioritizing maintainability defects based on refactoring recommendations. *22nd International Conference on Program Comprehension, ICPC 2014 - Proceedings*, 06 2014.

[2] The promise repository of empirical software engineering data, 2015.

[3] Rakesh Agrawal, Heikki Mannila, Ramakrishnan Srikant, Hannu Toivonen, A Inkeri Verkamo, et al. Fast discovery of association rules. *Advances in knowledge discovery and data mining*, 12(1):307–328, 1996.

[4] Mohammed Al-Obeidallah, Miltos Petridis, and Stelios Kapetanakis. A survey on design pattern detection approaches. 7:41–59, 12 2016.

[5] Mamdouh Alenezi and Mohammed Akour. Exploring the connection between design smells and security vulnerabilities. *International Journal of Innovative Technology and Exploring Engineering*, 9:449–452, 06 2020.

[6] Mahmood Alfadel, Khalid Aljasser, and Mohammad Alshayeb. Empirical study of the relationship between design patterns and code smells. *PLOS ONE*, 15:e0231731, 04 2020.

[7] T. Alkhaeir and B. Walter. The effect of code smells on the relationship between design patterns and defects. *IEEE Access*, 9:3360–3373, 2021.

[8] Apostolos Ampatzoglou, Alexander Chatzigeorgiou, Sofia Charalampidou, and Paris Avgeriou. The effect of gof design patterns on stability: A case study. *IEEE Transactions on Software Engineering*, 41, 08 2015.

[9] Apostolos Ampatzoglou, Apostolos Kritikos, George Kakarontzas, and Ioannis Stamelos. An empirical investigation on the reusability of design patterns and software packages. *Journal of Systems and Software*, 84:2265–2283, 12 2011.

[10] Bente Anda. Assessing software system maintainability using structural measures and expert assessments. pages 204 – 213, 11 2007.

[11] Francesca Arcelli Fontana, Alessandro Marino, and Vincenzo Ferme. Is it a Real Code Smell to be Removed or not? In *RefTest2013 – Int'l Ws. Refactoring & Testing (co-located with XP)*, 2013.

[12] Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. Linguistic antipatterns: what they are and how developers perceive them. *Empirical Software Engineering*, 21, 01 2015.

[13] Ishani Arora, Vivek Tetarwal, and Anju Saha. Open issues in software defect prediction. *Procedia Computer Science*, 46:906–912, 12 2015.

[14] L. Aversano, Luigi Cerulo, and Massimiliano Di Penta. Relationship between design patterns defects and crosscutting concern scattering degree: An empirical study. *Software, IET*, 3:395 – 409, 11 2009.

[15] Lerina Aversano, Gerardo Canfora, Luigi Cerulo, Concettina Del Grosso, and Massimiliano Di Penta. An empirical study on the evolution of design patterns. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 385–394, New York, NY, USA, 2007. ACM.

[16] Paulo J. Azevedo and Alípio M. Jorge. Comparing rule measures for predictive association rules. In Joost N. Kok, Jacek Koronacki, Raomon Lopezde Mantaras, Stan Matwin, Dunja Mladenič, and Andrzej Skowron, editors, *Machine Learning: ECML 2007*, volume 4701 of *Lecture Notes in Computer Science*, pages 510–517. Springer Berlin Heidelberg, 2007.

[17] James Bieman, Dolly Jain, and Helen Yang. Oo design patterns, design structure, and program changes: An industrial case study. pages 580–589, 02 2001.

[18] James M. Bieman, Greg Straw, Huxia Wang, P. Willard Munger, and Roger T. Alexander. Design patterns and change proneness: An examination of five evolving systems. In *Proceedings of the 9th International Symposium on Software Metrics*, METRICS '03, pages 40–, Washington, DC, USA, 2003. IEEE Computer Society.

[19] Mohamed Boussaa, Wael Kessentini, Marouane Kessentini, Slim Bechikh, and Soukeina Ben Chikha. Competitive coevolutionary code-smells detection. In Günther Ruhe and Yuanyuan Zhang, editors, *Search Based Software Engineering*, pages 50–65, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[20] Sergey Brin, Rajeevand Motwani, Jeffrey D. Ullman, and Shalom Tsur. Dynamic itemset counting and implication rules for market basket data. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data*, pages 255–264, Tucson, Arizona, USA, May 1997.

[21] Dénes Bán and Rudolf Ferenc. Recognizing antipatterns and analyzing their effects on software maintainability. pages 337–352, 06 2014.

[22] Aloisio Cairo, Glauco Carneiro, and Miguel Monteiro. The impact of code smells on software bugs: A systematic literature review. *Information*, 9:273, 11 2018.

[23] Aloisio Cairo, Glauco Carneiro, Antonio Resende, and Fernando Abreu. The influence of god class and long method in the occurrence of bugs in two open source software projects: An exploratory study (s). pages 199–204, 07 2019.

[24] Bruno Cardoso and Eduardo Figueiredo. Co-occurrence of design patterns and bad smells in software systems: An exploratory study. In *Proceedings of the annual conference on Brazilian symposium on information systems: Information systems: A computer socio-technical perspective*, pages 347–354, 2015.

[25] Gemma Catolino, Fabio Palomba, Andrea Lucia, Filomena Ferrucci, and Andy Zaidman. Developer-related factors in change prediction: An empirical assessment. 05 2017.

[26] M. Chaumun, Hind Kabaili, Rudolf Keller, and F. Lustman. A change impact model for changeability assessment in object-oriented software systems. volume 45, pages 130–138, 01 1999.

[27] Bee Bee Chua and Laurel Evelyn Dyson. Applying the iso 9126 model to the evaluation of an e-learning system. In *Proc. of ASCILITE*, pages 5–8, 2004.

[28] Mel O Cinnéide. *Automated application of design patterns: a refactoring approach*. PhD thesis, Trinity College Dublin, 2001.

[29] Jacob Cohen. Statistical power analysis for the behavioral sciences. *SERBIULA (sistema Librum 2.0)*, 2nd, 01 1988.

[30] James Coplien, Ron Crocker, Motorola Inc, Lutz Dominick, Gerard Meszaros, Frances Paulisch, and Kent Beck. Industrial experience with design patterns. 11 1997.

[31] Steve Counsell, R. M Hierons, H Hamza, Sue Black, and M Durrand. Is a strategy for code smell assessment long overdue? In *Ws. Emerging Trends in Softw. Metrics*, 2010.

[32] Marco D'Ambros, Alberto Bacchelli, and Michele Lanza. On the impact of design flaws on software defects. pages 23–31, 07 2010.

[33] Marco D'Ambros, Alberto Bacchelli, and Michele Lanza. On the Impact of Design Flaws on Software Defects. In *Int'l Conf. Quality Softw.*, pages 23–31, 2010.

[34] Massimiliano Di Penta, Luigi Cerulo, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An empirical study of the relationships between design pattern roles and class change proneness. pages 217 – 226, 11 2008.

[35] Jing Dong, Dushyant Lad, and Yajing Zhao. Dp-miner: Design pattern discovery using matrix. pages 371–380, 03 2007.

[36] Mahmoud Elish and Mawal Mohammed. Quantitative analysis of fault density in design patterns: An empirical study. *Information and Software Technology*, 66, 10 2015.

[37] Ezgi Erturk and Ebru Sezer. Iterative software fault prediction with a hybrid approach. *Applied Soft Computing*, 49, 08 2016.

[38] Rudolf Ferenc, Zoltan Toth, Gergely Ladányi, István Siket, and Tibor Gyimóthy. A public unified bug dataset for java. pages 12–21, 10 2018.

[39] Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology*, 11(2):5: 1–38, 2012.

[40] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[41] Martin Fowler, Kent Beck, J Brant, and William Opdyke. Refactoring: improving the design of existing code. 1999. *Google Scholar Google Scholar Digital Library Digital Library.*

[42] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1994.

[43] Matt Gatrell and Steve Counsell. Design patterns and fault-proneness a study of commercial software. pages 1–8, 05 2011.

[44] Robert B. Grady. *Practical Software Metrics for Project Management and Process Improvement.* Prentice-Hall, Inc., USA, 1992.

[45] Latifa Guerrouj, Zeinab Kermansaravi, Venera Arnaoudova, Foutse Khomh, Giuliano Antoniol, Yann-Gaël Guéhéneuc, and Benjamin Fung. Investigating the relation between lexical smells and change-and fault-proneness: An empirical study. *Software Quality Journal*, 05 2016.

[46] Aakanshi Gupta, Bharti Suri, and Sanjay Misra. A systematic literature review: Code bad smells in java source code. pages 665–682, 07 2017.

[47] Tracy Hall, Min Zhang, David Bowes, and Yi Sun. Some code smells have a significant but small effect on faults. *ACM Transactions on Software Engineering and Methodology*, 23:1–39, 09 2014.

[48] M. Harman. The current state and future of search based software engineering. In *Future of Software Engineering (FOSE '07)*, pages 342–357, 2007.

[49] Larry Hedges. Estimation of effect size from a series of independent experiments. *Psychological Bulletin*, 92:490–499, 09 1982.

[50] Péter Hegedüs, Dénes Bán, Rudolf Ferenc, and Tibor Gyimóthy. Myth or reality? analyzing the effect of design patterns on software maintainability. volume 340, pages 138–145, 01 2012.

[51] ISO Iso. Iec 9126-1: Software engineering-product quality-part 1: Quality model. *Geneva, Switzerland: International Organization for Standardization*, 21, 2001.

[52] Fehmi Jaafar, Yann-Gaël Guéhéneuc, Sylvie Hamel, and Foutse Khomh. Analysing anti-patterns static relationships with design patterns. *ECEASST*, 59, 2013.

[53] Fehmi Jaafar, Angela Lozano, Yann-Gaël Guéhéneuc, and Kim Mens. On the analysis of co-occurrence of anti-patterns and clones. 07 2017.

[54] Adam C. Jensen and Betty H.C. Cheng. On the use of genetic programming for automated refactoring and the introduction of design patterns. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, GECCO '10, pages 1341–1348, New York, NY, USA, 2010. ACM.

[55] Sang-Uk Jeon, Joon-Sang Lee, and Doo-Hwan Bae. An automated refactoring approach to design pattern-based program transformations in java programs. In *Ninth Asia-Pacific Software Engineering Conference, 2002.*, pages 337–345, 2002.

[56] Capers Capers Jones. *Software Quality: Analysis and Guidelines for Success*. Thomson Learning, 1st edition, 1997.

[57] Marian Jureczko and Lech Madeyski. Towards identifying software project clusters with regard to defect prediction. volume 9, page 9, 12 2010.

[58] Arvinder Kaur, Kamaldeep Kaur, and Shilpi Jain. Predicting software change-proneness with code smells and class imbalance learning. pages 746–754, 09 2016.

[59] Kamaldeep Kaur and Shilpi Jain. Evaluation of machine learning approaches for change-proneness prediction using code smells. In *Proceedings of the 5th International Conference on Frontiers in Intelligent Computing: Theory and Applications - FICTA 2016, Volume 1*, pages 561–572, 2016.

[60] Joshua Kerievsky. *Refactoring to Patterns.* Pearson Higher Education, 2004.

[61] Marouane Kessentini, Stephane Vaucher, and Houari Sahraoui. Deviance from perfection is a better criterion than closeness to evil when identifying risky code. pages 113–122, 01 2010.

[62] Wael Kessentini, Marouane Kessentini, Houari Sahraoui, Slim Bechikh, and Ali Ouni. A cooperative parallel search-based software engineering approach for code-smells detection. *Software Engineering, IEEE Transactions on*, 40:841–861, 09 2014.

[63] F. Khomh and Y. Gueheneuce. Do design patterns impact software quality positively? In *2008 12th European Conference on Software Maintenance and Reengineering*, pages 274–278, April 2008.

[64] Foutse Khomh, Massimiliano Di Penta, and Yann-Gaël Guéhéneuc. An Exploratory Study of the Impact of Code Smells on Software Change-proneness. In *Working Conf. Reverse Eng.*, pages 75–84. IEEE, 2009.

[65] Foutse Khomh, Massimiliano Di Penta, and Yann-Gaël Guéhéneuc. An exploratory study of the impact of code smells on software change-proneness. pages 75–84, 01 2009.

[66] Foutse Khomh and Yann-Gaël Guéhéneuc. Perception and reality: What are design patterns good for? 01 2007.

[67] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Softw. Engg.*, 17(3):243–275, June 2012.

[68] Akif Koru and Hongfang Liu. An investigation of the effect of module size on defect prediction using static measures. *ACM SIGSOFT Software Engineering Notes*, 30:1–5, 07 2005.

[69] C. Kramer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Proceedings of WCRE '96: 4rd Working Conference on Reverse Engineering*, pages 208–215, 1996.

[70] Danny Lange and Yuichi Nakamura. Interactive visualization of design patterns can help in framework understanding. volume 30, pages 342–357, 10 1995.

[71] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer Publishing Company, Inc., 2005.

[72] Michele Lanza and Radu Marinescu. Object-oriented metrics in practice. 01 2006.

[73] Wei Li and Raed Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of Systems and Software*, 80:1120–1128, 07 2007.

[74] Wei Li and Raed Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of Systems and Software*, 80:1120–1128, 07 2007.

[75] K. Lieberherr, I. Holland, and A. Riel. Object-oriented programming: An objective sense of style. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '88, page 323–334, New York, NY, USA, 1988. Association for Computing Machinery.

[76] Bennett P. Lientz and E. Burton Swanson. *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980.

[77] Huihui Liu, Bixin Li, Yibiao Yang, Wanwangying Ma, and Ru Jia. Exploring the impact of code smells on fine-grained structural change-proneness. *International Journal of Software Engineering and Knowledge Engineering*, 28:1487–1516, 10 2018.

[78] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *Ann. Math. Statist.*, 18(1):50–60, 03 1947.

[79] Henry B. Mann. Nonparametric Tests Against Trend. *Econometrica*, 13(3):245–259, July 1945.

[80] Mika V Mäntylä. An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement. In *Int'l Conf. Softw. Eng.*, pages 277–286, 2005.

[81] Radu Marinescu. Detection startegies: Metrics-based rules for detecting design flaws. pages 350– 359, 10 2004.

[82] Manishankar Mondal, Md Saidur Rahman, Chanchal Roy, and Kevin A. Schneider. Is cloned code really stable? *Empirical Software Engineering*, 23, 07 2017.

[83] J. C. Munson and S. G. Elbaum. Code churn: A measure for estimating the impact of code change. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 24–, Washington, DC, USA, 1998. IEEE Computer Society.

[84] Rogeres Nascimento and Claudio Sant'Anna. Investigating the relationship between bad smells and bugs in software systems. pages 1–10, 09 2017.

[85] J. Niere, W. Schafer, J. P. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 338–348, 2002.

[86] Steffen Olbrich, Daniela Cruzes, Victor Basili, and Nico Zazworka. The evolution and impact of code smells: A case study of two open source systems. pages 390–400, 10 2009.

[87] Steffen M Olbrich, Daniela S Cruzes, and Dag I K Sjøberg. Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. In *IEEE Int'l Conf. Softw. Maintenance*, pages 1–10, 2010.

[88] Ozan Onarcan and Yongjian Fu. A case study on design patterns and software defects in open source software. *Journal of Software Engineering and Applications*, 11:249–273, 01 2018.

[89] Ali Ouni, Marouane Kessentini, Houari Sahraoui, and Mounir Boukadoum. Maintainability defects detection and correction: A multi-objective approach. *Automated Software Engineering*, 20, 03 2012.

[90] Jukka Paakki, Anssi Karhinen, Juha Gustafsson, Lilli Nenonen, and A. Inkeri Verkamo. Software metrics by architectural pattern mining. 2000.

[91] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea Lucia. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*, pages 1–34, 08 2017.

[92] L. Prechelt, B. Unger, W.F. Tichy, P. Brossler, and L.G. Votta. A controlled experiment in maintenance: comparing design patterns to simpler solutions. *Software Engineering, IEEE Transactions on*, 27(12):1134–1144, Dec 2001.

[93] Lutz Prechelt, Barbara Unger-Lamprecht, Michael Philippsen, and Walter Tichy. Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. *Software Engineering, IEEE Transactions on*, 28:595–606, 07 2002.

[94] Nornadiah Razali and Yap B. Wah. Power comparisons of Shapiro-Wilk, Kolmogorov-Smirnov, Lilliefors and Anderson-Darling tests. *Journal of Statistical Modeling and Analytics*, 2(1), June 2011.

[95] Dirk Riehle. Lessons learned from using design patterns in industry projects. *Transactions on Pattern Languages of Programming*, 2:1–15, 01 2011.

[96] Bruno Rossi and Barbara Russo. Evolution of design patterns: A replication study. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, pages 38:1–38:4, New York, NY, USA, 2014. ACM.

[97] Guogen Shan and Shawn Gerstenberger. Fisher's exact approach for post hoc analysis of a chi-squared test. *PLOS ONE*, 12(12):1–12, 12 2017.

[98] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3-4):591–611, 1965.

[99] David J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman Hall/CRC, 4 edition, 2007.

[100] F. Shull. Perfectionists in a world of finite resources. *IEEE Software*, 28(2):4–6, 2011.

[101] Bruno Sousa, Mariza Bigonha, and Kecia A. M. Ferreira. A systematic literature mapping on the relationship between design patterns and bad smells. pages 1528–1535, 04 2018.

[102] Bruno Sousa, Mariza Bigonha, and Kecia A. M. Ferreira. An exploratory study on cooccurrence of design patterns and bad smells using software metrics. *Software: Practice and Experience*, 05 2019.

[103] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis. Design pattern detection using similarity scoring. *IEEE Transactions on Software Engineering*, 32(11):896–909, Nov 2006.

[104] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad. 05 2015.

[105] M. Vokac. Defect frequency and design patterns: An empirical study of industrial code. *Software Engineering, IEEE Transactions on*, 30:904 – 917, 01 2005.

[106] Marek Vokáč. Defect frequency and design patterns: an empirical study of industrial code. *Software Engineering, IEEE Transactions on*, 30(12):904–917, Dec 2004.

[107] Bartosz Walter and Tarek Alkhaeir. The relationship between design patterns and code smells: An exploratory study. *Information & Software Technology*, 74:127–142, 2016.

[108] Peter Wendorff. Assessment of design patterns during software reengineering: lessons learned from a large commercial project. In *Software Maintenance and Reengineering, 2001. 5th European Conference on*, pages 77–84, 2001.

[109] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 12 1945.

[110] B. Wydaeghe, K. Verschaeve, B. Michiels, I. Van Bamme, E. Arckens, and V. Jonckers. Building an omt-editor using design patterns: an experience report. In *Technology of Object-Oriented Languages, 1998. TOOLS 26. Proceedings*, pages 20–32, Aug 1998.

[111] Aiko Yamashita and Leon Moonen. Do code smells reflect important maintainability aspects? pages 306–315, 09 2012.

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

[113] Cheng Zhang and David Budgen. What do we know about the effectiveness of software design patterns? *IEEE Transactions on Software Engineering*, 38(5):1213–1231, 2012.

[114] Xiaoyan Zhu, Yueyang He, Long Cheng, Xiaolin Jia, and Lei Zhu. Software change-proneness prediction through combination of bagging and resampling methods. *Journal of Software: Evolution and Process*, 30, 10 2018.