**An empirical investigation of the connection between design patterns and bad smells**

**Abstract:**

Software systems are frequently created in a way that violates the object-oriented paradigm's best practices, leading to the occurrence of particular inconsistencies that are commonly referred to as "code smells." Best practices for creating object-oriented software systems are compiled in design patterns. Despite the stark differences between code smells and design patterns, there may be a correlation between the two. This study aims to empirically determine whether the presence of design patterns at various granularity levels is associated with the presence of code smells.

We conducted an empirical analysis employing 30 small to medium-sized open-source Java-based systems, 20 design patterns, and 13 code smells. We used association rules and statistical analyses. Results confirm that classes participating in design patterns have less smell-proneness and smell frequency than classes not participating in design patterns.  We also observed that every design pattern category behaves uniformly in the subject systems with regard to smell susceptibility. However, we found that some patterns might occasionally be connected to particular smells based on the association rules learning and the suggested validation technique. For instance, God Class, Blob, and External Duplication smell can all co-occur with Command patterns.

**Introduction**

Design patterns (DP) are recurrent fixes for typical issues in software architecture. DPs seek to lessen coupling and increase reuse. DPs can facilitate teamwork by employing terminology rather than the conventional explanations. 23 design patterns that can be categorized as creational, structural, or behavioral were found by Gamma et al. Creational design patterns are ones that deal with creating objects that fulfil a function that is appropriate for the circumstance. By locating straightforward ways to realize relationships between various entities, structural design patterns are design methodologies that simplify the process of creating software. Communication patterns are behavioral design patterns..

Several classes of various roles are referred to as design patterns or design motifs. Every design motif has a role in the source code. Software quality characteristics including maintainability, reusability, and fault-proneness are impacted by DPs. Design patterns may have a good or negative impact on software quality qualities. When utilized improperly, design patterns can reduce system performance and make programming excessively complex. The results of many research on the influence of design patterns on software quality have varied. Voká discovered that classes using participant design patterns have fewer flaws. Prechelt et al. have demonstrated that design patterns can influence maintenance factors favorably. However, other research found a detrimental effect of design patterns on software quality.

Code smells are issues that arise in a small section of code that make software challenging to maintain and modify. Code smells were first identified by Fowler in 1999 as indicators of software maintenance issues. Code smells, in contrast to design patterns, are warning indications that developers might use to find a specific problem. Code smells can therefore indicate areas of the code that need closer inspection. Researchers have reported varying findings about the influence of code smells on various quality factors, much like design patterns. Li and Shatnawi discovered a connection between code smells and defect density.

However, research has shown that smells can help to identify maintenance problems. Code smells have been identified using a variety of parameters, including the software domain and the programmer's experience. Therefore, it is crucial for researchers to develop new systems to identify bad smells. Expanding on other aspects might therefore aid practitioners in better comprehending how bad smells relate to code.

Recent studies looked at design patterns and code smells in various ways. Most of them talked about the subject from refactoring points. For instance, research has suggested refactoring tools for fragments. In other investigations, the structural relationship between design patterns and code smells was examined. Additionally, Wendorff described instances in which design patterns may have a negative effect on qualities like maintainability, which may result in the introduction of bad smells. According to McNatt and Bieman, when certain design patterns—such as Command, Proxy, Bridge, and Observer—are not used correctly, the system's performance may suffer and the code may become unmanageable.

While bad smells signify a lack of design or coding problems, design patterns are connected to effective software design and code. Since DPs and bad smells have opposing structural relationships, they are rarely studied in the same research setting. Given the contradictory effects of design patterns on quality attributes and the hazy relationships between code smells and maintainability issues as a result of design pattern applications, the relationship between DPs and code smells, with a focus on their co-occurrences, merits more study. Additionally, very few research have looked into the connection between DPs and code smells directly. As a result, more research is required to thoroughly examine and compare a generalized proof of how design patterns affect code smells.

Some patterns' descriptions may suggest correlations with smells that may exist. The Strategy pattern, for instance, produces a collection of objects in classes that represent algorithms. Data that the objects could require is, however, separated. Consequently, such an instance can leave a Feature Envy smell in the classes that carry data. The Strategy pattern's scenario is comparable to that of the Factory Method pattern. A collection of parameters are sent to set up an instance when a factory method creates objects from other classes. Therefore, the pattern can lead to a link between the smell of blobs and long parameter lists. These theories further encouraged us to carry out this investigation to determine whether design patterns and code smells might be connected.

**Related work**

In this section, we evaluate the associated literature with an emphasis on the relationship between DPs and code smells, the impact of DPs on code smells, and the impact of code smells on software quality attributes.

**The impact of design patterns on software quality attributes**

Researchers looked into how DPs affected various quality characteristics, including fault-proneness, maintainability, and change-proneness. Prechelt et al experiments contrasted software maintenance using various design patterns to other scenarios that involved more straightforward solutions. They discovered that design solutions that make use of GoF's patterns are frequently simpler to maintain than their matching basic alternatives.

However, the authors also discovered that there were instances where using design patterns made it more difficult to maintain the program. They also revealed that design patterns could necessitate more maintenance work than a simple solution. The Visitor pattern resulted in a significant cost in terms of development time and poor accuracy, according to Vokac et al., who conducted a study to replicate Prechelt et al. Additionally, they discovered that Decorator makes maintenance easier even while it makes it more difficult to grasp the program's control flow. This reduces understandability even while its maintainability is good. While some research concluded DPs having a detrimental effect on maintainability, others found no discernible trend of any influence and advised conducting a more thorough and useful analysis. Additionally, Garzas et al. discovered that pattern-based designs make it more difficult to understand and modify them. However, Hegedus et al. observed that DPs can have a good impact on maintainability and result in code improvements after evaluating the effects of specific DPs in JHotDraw systems.

A controlled experiment was carried out by Lutz Prechelt et al. to determine whether the maintainer could work more effectively and quickly if DPs in program code were explicitly documented using comments as opposed to having merely well-documented program without references to DPs. They claimed that maintenance tasks based on patterns were completed more quickly or encountered fewer issues than when done the other way. In a study to assess the connection between DPs and change-proneness, Aversano et al. The findings demonstrate that DPs are more vulnerable to modifications when DPs have a major impact on system functionality. Gatrell et al. discovered, however, that some participation classes in DPs are more susceptible to modifications than non-participant classes. The links between software change-proneness and design structure were looked at by Bieman et al.

The experiment was run on thirty subject program from Github. Design structure was identified using design patterns, class size, and participation in class inheritance. According to the study's findings, classes that use design patterns are more prone to change than other classes. This outcome might be explained by the fact that design patterns give the system's core functions and roles. This could help to explain why classes that use design patterns are, on average, more prone to change than other classes. Change-proneness in classes that use design patterns may be a clue that these patterns also use the code smells Divergent Change or Shotgun Surgery, as described by Fowler et al.

Additionally, they found that patterns are better suited for applications that typically change more frequently. The literature mentions a few DPs that support the existence of defects. This problem is covered in several studies. Using C++ systems, Vocak investigated the connection between DPs and the quantity of defects. The outcomes display various trends for various patterns. For instance, because of their extensive structural associations, Singleton and Observer were shown to have more flaws. Because Factory Method classes are loosely coupled to the system structure, Factory Methods, in contrast, were more resistant to errors. Additionally, the Template Method and Decorator patterns showed no discernible trend. Gatrell and Counsell looked into how DPs and fault-proneness are related.

In particular, the Adaptor, Singleton, and Template Method classes were found to be more fault-prone than classes that did not participate in DPs, according to the authors. They used C# systems to perform their investigation on eleven Gang of Four (GoF) patterns. Using Java systems, Ampatzoglou et al. assessed the correlation between eleven DPs and fault frequency. The findings indicate a link between the examined patterns and defect frequency that is both positive and negative. For instance, the Adaptor pattern has a positive correlation with defect frequency while the Observer pattern has a negative correlation with defect frequency.

From the aforementioned investigations, it can be shown that different studies produced varying conclusions about the impact of design patterns on software quality, leaving the question of the impact of design patterns on software quality unanswered.

**The impact of code smells on software quality attributes**

Software systems are thought to have maintenance issues as a result of code smells. Additionally, according to some studies, classes with code smells are more likely to change often and contain flaws than classes without code smells.

God and Brain class smells have been linked to increased change size, change frequency, and flaws, according to research by Olbrich et al. Classes containing these code smells had higher change size, change frequency, and defects. More smells were used to further corroborate the study's findings.

Oddly, Olbrich et al. discovered that when their findings were normalized for size, or Line of Code (LOC), the conclusions no longer held under the premise that classes included in code smells, such as God Class and Brain Class, have a ratio of functionality similar to other classes, on average. They came to the conclusion that, if the smelly classes are created on purpose, code smells are generally not harmful and may even be a useful approach to organise code.

**The relationship between design patterns and code smells**

The most relevant work to ours is that of Walter and Alkhair. In their study, they conducted an empirical study to evaluate the relationship between DP and code smell at the class level. The authors explored this relationship using 10 GoF design patterns and 7 code smells. They used open source projects, JfreeChart and Maven, with many subsequent versions of each project. The results show that the presence of DP is not strongly associated with code smell. Any class that participates in design patterns is most likely not a stinky class. However, for certain patterns like Singleton, more supported, for others, etc.Composite, less supported. No clear trends were found for smell occurrence across the pattern-based systems analyzed. The data provided by the survey is limited to the class level and not the DP category and role level. This is due to the limitations of the design pattern detection tools used in the study.

In general, their results indicate that design pattern classes are associated with a lower number of smell Inspired by their work and supported by the fact that different design patterns can be associated with different code smells, this study focused on more systems, design patterns, and code smells. We also consider smell frequency factors when investigating possible co-occurrences of design patterns and code smell Additionally, we test the relationship between DP and smell from a category-level perspective. Finally, we manually examine key cases that indicate potential relationships to gain more insight and associate a particular code smell with a particular design pattern or set of design patterns. Determine if the In contrast to the Walter and Alkhair study, we used a publicly available, manually validated design pattern dataset to increase confidence in our results. Although our overall results are close to their study, we observed that some patterns were associated with specific smell For example, you can see that the Command Pattern is associated with the Smell Blob, External Duplication, and God Class. We conclude that there is a correlation between the presence of certain design patterns and smell in certain scenarios.

**Empirical study setup**

The main purpose of this research is to investigate whether there is a relationship between design patterns and code smells.

Define the following sub-goals:

Empirically assess the smell susceptibility and frequency of code in design and non-design pattern classes at the class level.

Empirical assessment of code smell vulnerability for design pattern classes and non-design pattern classes in the design pattern categories (Creative, Structural, and Behavior).

Empirical evaluation of the sensitivity of individual design motifs to code smells. This helps identify the role of important design patterns.

The following sections present empirical research details, including:

research question, design patterns and code smells analyzed, data collection, research methodology, and measurement testing.

**Research Question**

To achieve the goals of this research, we assess the impact of design patterns on code smell at different levels of granularity:

**Class Level:**

Classes assigned to related design patterns and classes not involved in design patterns i.e. (Smelly Design Pattern Classes (SDP) vs. Smelly Non-Design Pattern Classes (SnDP)).

**Category Level:**

This level empirically assesses differences in smell sensitivity between classes containing different categories of design patterns.

**Motif Level:**

This level empirically evaluates differences in the number of smell that occur between classes that individually correspond to a specific single design motif. Additionally, this level identifies the role of any important design motifs.

Based on the above levels, formulate the following research questions.

**RQ1:** Are design pattern classes more smell-prone and more frequent than non-design pattern classes?

**RQ2:** Do code smells have significant differences in terms of proneness when they are present in the different categories of design pattern classes?

**RQ3:** Are the participant classes in a specific individual design motif more smell-prone for a specific smell?

**Research methodology**

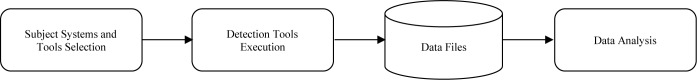
To achieve our research goals, we follow the methodology shown in Figure 1.

The methodology phases are

(1) selection of relevant systems and detection tools,

(2) execution of tools to obtain relevant results of design pattern and code smell data, and

(3) data analysis and data mining.



In the first phase, we selected the systems used for analysis based on the github repository. Additionally, I chose design patterns and code smell tools. The inFusion tool was chosen for code smell detection and github repository was used for design pattern data. Due to possible duplication, we kept the default values ​​for the code smell detector configuration parameters.

In the second phase, we collected code smell and design pattern data for each system. For design patterns, I took the XML files available in each system's github repository and analyzed them to get each design pattern class and corresponding design pattern. For code smells, we run the tool on each system and collected the smell classes and the number of smell in each class for each type of smell. This procedure also created 10 files per system. Then, for each system, both files were matched to design patterns and code smells for later comparison. So I created one file per project to store the design pattern output and corresponding code smells. This makes the necessary design pattern and code smell information available for each class of each file in the system. Each file has 25 columns. The first column is the class name, followed by the corresponding design pattern for that class, if any. Column number 3 indicates the code's smell occurrence (0 or 1), followed by a column indicating the number of smell in that class. The fifth column indicates what kind of smell the class can have. The remaining 20 columns are related to the corresponding design pattern type and are grouped according to the design pattern category (ie creation, structure, and behavior).

**The used design patterns and code smells**

In this work, we used 20 design patterns out of the 23 proposed by Gamma et al. that are available in the P-Mart repository . Table 1 lists the design patterns used in this study.

Table 1

**The used design patterns.**

| **Name** | **Category** |
| --- | --- |
| Abstract Factory, Builder, Factory Method, Prototype, Singleton | Creational |
| Adapter, Bridge, Composite, Facade, Decorator, Proxy | Structural |
| Command, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor | Behavioral |
|  |  |

Several classifications have been associated with smells, including: Code Smell vs. Design Smell . Antipatterns are another concept related to code smells. Jafar et al. found that code smells are related to internal regions of classes, whereas anti-patterns are related to relationships between classes. The code smell detection tool used in this work was able to find 13 code smell types in the data set used. So let's analyze these 13 code smells.

are seven smells defined by Fowler and six odors defined by Lanza and Marinescu . Table 2 lists these odors and their abbreviations.

Table 2

**The used code smells.**

| **Name** | **Acronym** |
| --- | --- |
| Data Class | DC |
| Data Clumps | DCl |
| Refused Parent Bequest Class | RPB |
| Schizophrenic Class | SC |
| Blob Methods | BL |
| Intensive Coupling | IC |
| Sibling Duplication | SD |
| Internal Duplication | ID |
| External Duplication | ED |
| God Class | GC |
| Feature Envy | FE |
| Tradition Breaker | TB |
| Message Chains | MC |

**Detecting Code Smells:**

To detect code smells in selected systems, we followed the approach of Walter and Alkhair using the inCode tool. However, for this article, I have chosen to use an advanced version of the inCode tool, the 'inFusion' tool . inFusion was able to detect 22 code smells, 10 of which were identified by Fowler . iPlasma is an older version of it. Additionally, inFusion provides well-documented definitions of the detection rules and techniques used and associated metrics with references. Supports visualization and refactoring.

Additionally, due to the high false positive rate for most detectors, we ran the inCode tool on data samples from the affected systems to validate the detected instances. We use the inCode tool, which has been used in many studies on design patterns and code smells . In this example, we assumed that an smell is present in the system if both tools can detect the same instance of the smell. In this way, the results of smell detection are made more reliable. Considering 30% as a random sample of smell instances (~30 instances), we found an error rate of 4.1% (1 instances). Most importantly, the inFusion and inCode tools outperform other tools in their ability to detect smell independent of compilation problems.

**Statistics on smelly and design pattern classes.**

| **Projects** | **# Classes** | **# & Percentage of DP Classes** | **# of Smelly DP Classes(SDP)** | **# of Smelly non-DP Classes(SnDP)** | **# &Percentage of Smelly Classes** | **# of Non-Smelly DP Classes(nSDP)** | **#of Non-Smelly non-DP Classes(nSnDP)** | **# & Percentage of Non-Smelly Classes** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| P1 | 267 | 85 (31.8%) | 9 | 25 | 34 (12.7%) | 76 | 157 | 233(87.3%) |
| P2 | 155 | 103 (66.5%) | 5 | 3 | 8 (5.5%) | 98 | 49 | 147(94.8%) |
| P3 | 215 | 41 (19.1%) | 5 | 20 | 25 (11.6%) | 36 | 154 | 190(88.4%) |
| P3 | 238 | 55 (23.1%) | 5 | 23 | 28 (11.8%) | 50 | 160 | 210(88.2%) |
| P4 | 217 | 48 (22.1%) | 6 | 21 | 27 (12.4%) | 42 | 148 | 190(87.6%) |
| P5 | 165 | 41 (24.8%) | 13 | 26 | 39 (23.6%) | 28 | 98 | 126(76.4%) |
| P6 | 446 | 43 (9.6%) | 6 | 29 | 35 (7.8%) | 37 | 374 | 411(92.2%) |
| P7 | 78 | 52 (67.7%) | 2 | 2 | 4 (5.1%) | 50 | 24 | 74(94.9%) |
| P8 | 156 | 41 (26.3%) | 1 | 6 | 7 (4.5%) | 40 | 109 | 149(95.5%) |
| P9 | 24 | 7 (29.2%) | 2 | 10 | 12 (50%) | 5 | 7 | 12(50%) |
| **Total All Projects** | **1961** | **516(26.3%)** | **54** | **165** | **219(11.2%)** | **462** | **1280** | **1742(88.8%)** |

Table 6

**Design pattern instances in the subject systems.**

| **Category** | **Patterns** | **P1** | **P2** | **P3** | **P4** | **P5** | **P6** | **P7** | **P8** | **P9** | **P10** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Creational Patterns** | Abs Factory |  |  |  |  | 1 |  |  |  | 1 |  |
| Builder |  |  |  | 1 |  |  | 2 |  | 1 | 1 |
| Factory Method | 1 | 3 |  | 1 | 1 |  | 3 |  |  |  |
| Prototype |  | 2 |  |  |  |  |  |  |  |  |
| Singleton | 8 | 2 | 8 |  | 3 | 1 |  | 2 | 1 | 2 |
| **Structural patterns** | Adapter | 2 | 1 | 2 | 1 | 2 | 2 | 1 |  |  |  |
| Bridge | 1 |  |  |  |  | 2 |  |  |  |  |
| Composite |  | 1 |  |  | 1 |  | 2 | 1 | 1 |  |
| Decorator |  | 1 |  |  |  |  |  | 1 |  |  |
| Façade |  |  |  |  | 1 |  |  |  |  |  |
| Flyweight |  |  |  |  |  |  |  |  |  |  |
| Proxy | 1 |  | 1 | 1 |  |  | 1 |  |  |  |
| **Behavioral patterns** | Chain of Responsibility |  |  |  |  |  |  |  |  |  |  |
| Command | 2 | 1 | 2 |  |  | 2 |  |  | 1 |  |
| Interpreter |  |  |  |  |  |  |  |  |  |  |
| Iterator | 1 |  | 1 |  |  | 1 | 1 | 1 |  |  |
| Mediator | 1 |  |  |  |  |  |  |  |  |  |
| Memento | 1 |  | 1 |  |  | 2 |  |  |  |  |
| Observer |  | 2 |  |  | 1 |  | 2 | 3 | 1 | 2 |
| State | 3 | 2 |  | 3 |  |  |  |  |  |  |
| Strategy | 3 | 4 | 1 | 1 | 1 | 2 |  |  |  |  |
| Template | 9 | 2 |  | 8 | 4 | 3 | 1 |  |  |  |
| Visitor | 1 |  | 1 |  |  |  | 1 |  |  |  |
| **SUM** |  | **34** | **21** | **17** | **16** | **15** | **15** | **14** | **8** | **6** | **5** |

**Conclusion:**

I investigated the coexistence of design patterns and smell. We started our investigation by collecting design samples and code smell data. For design patterns he used the Github repository. For code smell detection, we used inFusion tools through 10 open source projects available on github.

Studies were evaluated at three levels: class, category, and individual pattern level. At the class level, we found that classes that participate in design patterns smell less than classes that do not participate in design patterns. Both smell sensitivity and smell frequency were considered at the class level and both showed consistent results.

At the category level, the results show that there is little significant difference between the categories regarding the sensitivity of the target system to smell. However, at the level of individual design motifs, specific examples of associations between design patterns and smell were discovered using association rule metrics, support, and trust. Although most of the rules showed weak associations between the presence of design patterns and the absence of code smells, we observed associations that could be conducive to the development of bad smells. The results show that the most notable cases are the stinky blob and god class command patterns.

**References:**

1. Prechelt L., Unger B., Tichy W. F., Brossler P., and Votta L. G., "A controlled experiment in maintenance: comparing design patterns to simpler solutions," *IEEE transactions on Software Engineering*, vol. 27, no. 12, pp. 1134–1144, 2001. [[Google Scholar](https://scholar.google.com/scholar_lookup?journal=IEEE+transactions+on+Software+Engineering&title=A+controlled+experiment+in+maintenance:+comparing+design+patterns+to+simpler+solutions&author=L.+Prechelt&author=B.+Unger&author=W.+F.+Tichy&author=P.+Brossler&author=L.+G.+Votta&volume=27&issue=12&publication_year=2001&pages=1134-1144&)]

2. K. Beck et al., "Industrial experience with design patterns," in Proceedings of the 18th international conference on Software engineering, 1996, pp. 103–114.

3. Gamma E., Helm R., Johnson R., and Vlissides J., Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1994. [[Google Scholar](https://scholar.google.com/scholar?q=Gamma+E.+,++Helm+R.+,++Johnson+R.+,+and++Vlissides+J.+,++Design+Patterns:+Elements+of+Reusable+Object-Oriented+Software+.++Addison+Wesley+,++1994+.+)]

4. F. Khomh, Y. G. Gueheneuc, and G. Antoniol, "Playing roles in design patterns: An empirical descriptive and analytic study," in 2009 IEEE International Conference on Software Maintenance, 2009, pp. 83–92.

5. Coplien J. O., "Software design patterns: Common questions and answers," *The Patterns Handbook: Techniques, Strategies, and Applications*, pp. 311–320, 1998. [[Google Scholar](https://scholar.google.com/scholar_lookup?journal=The+Patterns+Handbook:+Techniques,+Strategies,+and+Applications&title=Software+design+patterns:+Common+questions+and+answers&author=J.+O.+Coplien&publication_year=1998&pages=311-320&)]

6. Vokac M., "Defect frequency and design patterns: An empirical study of industrial code," *IEEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 904–917, 2004. [[Google Scholar](https://scholar.google.com/scholar_lookup?journal=IEEE+Transactions+on+Software+Engineering&title=Defect+frequency+and+design+patterns:+An+empirical+study+of+industrial+code&author=M.+Vokac&volume=30&issue=12&publication_year=2004&pages=904-917&)]

7. B. Wydaeghe, K. Verschaeve, B. Michiels, I. V. Bamme, E. Arckens, and V. Jonckers, "Building an OMT-editor using design patterns: an experience report," in Proceedings. Technology of Object-Oriented Languages. TOOLS 26 (Cat. No.98EX176), 1998, pp. 20–32.

8. L. Prechelt and B. Unger, "A Series of Controlled Experiments on Design Patterns: Methodology and Results " in Proceedings of Software Technik, Software Technik Trends, 1998.

9. P. Wendorff, "Assessment of design patterns during software reengineering: lessons learned from a large commercial project," in Proceedings Fifth European Conference on Software Maintenance and Reengineering, 2001, pp. 77–84.

10. F. Khomh and Y.-G. Gueheneuce, "Do Design Patterns Impact Software Quality Positively?," presented at the Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering, 2008.