Exploring Design Smells for Smell-Based Defect Prediction

Bruno Sotto-Mayor*, Amir Elmishali, Meir Kalech

Ben-Gurion University of the Negev, Beer-Sheva, 8410501, Israel

Rui Abreu

Faculty of Engineering of University of Porto, Porto, 4200-465, Portugal

Abstract

Defect prediction is commonly used to reduce the effort from the testing phase of software development. A promising strategy is to use machine learning techniques to predict which software components may be defective. Features are key factors to the prediction's success, and thus extracting significant features can improve the model's accuracy. In particular, **code smells** are a category of those features that have been shown to improve the prediction performance significantly. However, Design code smells, a state-of-the-art collection of code smells based on the violations of the object-oriented programming principles, have not been studied in the context of defect prediction. In this paper, we study the performance of defect prediction models by training multiple classifiers for 97 real projects. We compare using Design code smells as features and using other Traditional smells from the literature and both. Moreover, we cluster and analyze the models' performance based on the categories of Design code smells. We conclude that the models trained with both the Design code smells and the smells from the literature performed the best, with an improvement of 4.1% for the AUC score, compared to models trained with only Traditional smells. Consequently, Design smells are a good addition to the smells commonly studied in the literature for defect prediction.

Keywords: Defect Prediction, Code Smell, Mining Software Repositories, Software Quality, Software Engineering

*Corresponding author.

Email addresses: `mail@brunosottomayor.com` (Bruno Sotto-Mayor), `amirelm@post.bgu.ac.il` (Amir Elmishali), `kalech@bgu.ac.il` (Meir Kalech), `rui@computer.org` (Rui Abreu)

1. Introduction

There has been a higher demand for fast software delivery in recent years while maintaining good product quality. Therefore, the research has focused on finding approaches to allocate resources to the software development pipeline effectively. Since testing is one of the most time-consuming phases, the attention has been geared toward finding novel approaches to minimize the testing time without degrading the software quality. In particular, one of those approaches is based on the application of defect prediction tasks that use prediction models to predict which software components are defective, thus facilitating the scheduling of resources to those components.

Commonly studied in the literature, defect prediction algorithms use the historical information of software, such as its previous versions (through version control tools) and its reported bugs (through issue tracking tools), as the source of predictors for the classification of defects. In particular, they extract a set of features from each component of a software repository and assign a target label to whether the component has defects. Henceforth, it creates the training dataset, fed into a machine-learning algorithm to output a classification model. It is consequently used to predict which components in the following versions may be defective or not. In the end, these models have been shown to produce classifiers with a great predictive performance (Lessmann et al., 2008; Nagappan and Ball, 2005).

The features extracted from the software repositories are key to the defect prediction models' success. One of those features studied for defect prediction is the set of code smells. They are patterns in the source code that indicate deeper underlying issues in the system (Fowler and Beck, 1999). Due to their high correlation with the presence of defects, they are a good predictor candidate to build defect prediction classifiers (Piotrowski and Madeyski, 2020). Moreover, their static and relatively fast acquisition is a good motivator for their usage, and they also provide additional information for defect prediction when used in combination with other metrics.

Previous research on smell-based defect prediction studied the impact of different smells proposed in the literature. These include the works published by Fowler and Beck (1999) and Brown (1998), which introduce well known conceptual models for refactoring and code smells. In recent years, several papers have been published exploring the impact of those smells in defect prediction. Most of them proved the correlation between defects and smells and a positive impact on the use of smells in defect prediction (Piotrowski and Madeyski, 2020). In the scope of this study, we denominate the group of these smells as Traditional smells.

Although there is intensive research on the application of Traditional code smells in defect prediction, there is a collection of smells that have not been explored in this context. In particular, the Design code smells proposed by Ganesh et al. (2013), a comprehensive catalog of 31 structural Design smells, conceptualized with the four fundamental object-oriented design principles. This conceptualization of code smells, described as the violation of object-oriented design principles, is expected to introduce new defects into the system. In particular, since software defects are a causal effect of poor software quality driven by design flaws (D'Ambros et al., 2010), and such flaws are the product of the violation of rules derived from the object-oriented paradigm; therefore, there is a strong intuition as to why this group of smells is a strong predictor of defects. Consequently, the authors developed a tool called Designite that detects 17 of those design defects (Sharma, 2018). In this paper, we empirically study the impact of the 17 Design code smells in defect prediction, and we compare our results with Traditional smells used in the literature. All in all, our goal is to focus solely on the impact of Design smells over the smells traditionally used in defect prediction.

Given the research goals, we mined 97 Apache repositories and extracted the Design code smells and 20 other Traditional smells used in the literature. In addition, we created datasets for each category of features and their combinations. Thus we applied feature selection, trained several classifiers with those datasets, and selected and optimized the best models. Furthermore, we clustered the Design and Traditional smells based on the categories of Design code smells, conceptualized from the object-oriented design principles. In the end, the models trained with both Design and Traditional smells performed the best, with a 4.1% gain of the AUC compared with a classifier trained with only the Traditional smells with a significance level of $p < 0.05$.

The rest of the paper is structured as follows. In Section 2, we discuss the related work. In Section 3, we define the problem regarding defect prediction and the code smells. In Section 4 we describe the evaluation methodology for the defect prediction classifiers. Lastly, in Section 5, we present and discuss this study's results.

2. Related Work

Defect prediction is one of the most actively researched areas in software engineering. Its goal is to efficiently produce a list of defective-prone software components, which will allow developers to more effectively allocate their time and resources to those components (Paterson et al., 2019).

The application of code smells in defect prediction has been studied in the literature. Recently, Piotrowski and Madeyski (2020) performed a systematic literature review of 27 papers from 2006 to 2019 to analyze the relationship between smells and defects as well as, to evaluate the performance of code smells in defect prediction using machine learning techniques. They confirm that there is a positive correlation between code smells. Hence they are a good indicator of defects, and they positively influence the performance of defect prediction models. Moreover, they identify that the smells such as God Class, God Method, and Message Chains have a high impact on defect prediction. A few smells, such as the Middle Man and Speculative Generality, whose results have not been conclusive and should be further explored.

From those studies, Ma et al. (2016) research the possibility of improving defect prediction results using code smells. They also evaluate whether they could use the defect prediction's results to prioritize code smells refactoring. They consider eight code smells (i.e., Anti-singleton, Blob, Class Data Should Be Private, ComplexClass, LazyClass, LongParameterList, RefusedParentBequest, and SwissArmyKnife). They detect them from a single version from four projects using the DECOR smell detection tool (Moha et al., 2010). They conclude that the code smells detection results improve the recall of fault prediction in all projects by $9\% \sim 16\%$.

Taba et al. (2013) propose a set of antipattern metrics derived from the history of code smells within the software's version history. They evaluate defect prediction models trained with 13 smells taken from multiple versions of Eclipse and ArgoUML (i.e, AntiSingleton, Blob, ClassDataShouldBePrivate, ComplexClass, LargeClass LazyClass, LongParameterList, LongMethod, MessageChain, RefusedParentBequest, SpaghettiCode, SwissArmyKnife and SpeculativeGenerality). In the end, they found that those smells produce a higher density of bugs compared to traditional metrics. In addition, the antipattern metrics were shown to produce a higher exploratory power than those metrics.

Palomba et al. (2019, 2016) study the effectiveness of the measure of severity of code smells (i.e., code smell intensity) as a predictor in defect prediction. They evaluate the predictive power of the intensity index of 6 code smells (i.e., GodClass, DataClass, BrainMethod, ShotgunSurgery, DispersedCoupling, and MessageChains) by adding it to existing prediction models and comparing them against baseline metrics. They also applied an empirical comparison between their model and the antipattern metrics model suggested by Taba et al. (2013). They conclude that the intensity index always positively contributes to the state-of-the-art prediction models.

Although several studies have explored the impact of code smells in defect prediction, there is a collection of code smells developed by Ganesh et al. (2013), based on the design principles of object-oriented programming, which were not studied in the context of defect prediction. In particular, they re-frame and generalize a large scale of literature smells into a model based on the four design principles of object-oriented programming. We expect it to produce additional meaningful information to predict defects in code. Furthermore, Sharma (2018) developed a tool called *Designite* that implements a subset of 17 of the proposed smells. This paper aims to understand what contribution the Design smells introduce to the code smells already studied for defect prediction.

3. Problem Definition Methodology

This section defines the application of code smells in defect prediction. We start by formally defining defect prediction and its process, from the datasets' extraction and construction to the training and evaluation of the defect prediction classifiers (Section 3.1). Then, we describe the code smells and their contribution to the defect prediction (Section 3.2).

3.1. Defect Prediction

The defect prediction goal is to predict which components on the next version of the software have defects. Formally, a software repository is usually composed by a sequence of n versions $V = \{v_1, ..., v_n\}$, consequently composed by a discrete number of components. In our study, we assume each component is a file, where $v_i = \{f_1, ..., f_k\}$. As such, each instance of a mined dataset represents a particular file f_j for a specific version v_k of the software repository.

Moreover, given a software component, defect prediction's classification problem is to determine the state of the said component - defective or not. Therefore, machine learning techniques can be applied to train a classification model that predicts its state for a given component. For this task, a training set is required. The training set includes instances where components' features describe each instance - those are extracted from the source code and its label, which state whether the component is defective. One of the critical aspects to achieving a good performance while predicting the target state is to choose meaningful features (Moser et al., 2008). Product metrics and process metrics are the most widely explored categories, vastly studied in the earlier times of defect prediction, and they have generally shown positive results (Li et al., 2018). While the product metrics describe the design and behaviour of the current state of the software (e.g., CK (Chidamber and Kemerer, 1994) and McCabe's cyclomatic complexity (McCabe, 1976)), the process metrics extract the features from historical information stored in software repositories such as version control systems and issue tracking systems(e.g., churn (Nagappan and Ball, 2005) and entropy metrics (Hassan, 2009)).

The typical approach for classification in defect prediction is the application of supervised machine learning algorithms on the data. This process begins with the generation of the datasets. It extracts the set of predictors from the components of each version of the software repository and attaches the corresponding defective information.

Hereafter, the dataset is processed, accounting for missing values and scaling abnormalities in the data, and it is split into a training set and a test set. The training set is used as input to a learning algorithm, which outputs a classification model that predicts, for a new unlabeled instance, whether it is defective or not. After creating the classifier, the test set evaluates the model's performance by comparing the predicted classification against the actual classification. Overall, the classification model defines a mapping between the features and the target label.

To extract the target label from the software repositories, i.e., the defective information, most defect prediction approaches retrieve this information from version control systems and issue tracking systems. In particular, since issue tracking systems keep track of fixed reported defects, by matching those issues to the respective bug-fixing commits, they can map the code files that were involved in the defect; this is done since there is a conventional practice of writing the id of the defective issue to the message of the commit that is fixing it. Then, after filtering out the irrelevant files, these approaches label each of the files in that specific commit as defective or not (Borg et al., 2019).

In our study, we extend the practical field of defect prediction and explore the impact of code smells as predictors for the classification. In particular, we study the prediction power of a novel catalog of smells called design smells, derived from the generalization of traditional code smells defined in the literature but adapted to conform to the violations of design principles in the object-oriented programming paradigm. Since code smells have been shown to be both positively correlated with software defects and to positively influence defect prediction models' performance when used as features (Piotrowski and Madeyski, 2020), we conjecture that the added theoretical framework to the formulation of Design smells increases the knowledge of the defect prediction models. In particular, since it was observed that software defects are a causal effect of poor software quality, driven by design flaws (D'Ambros et al., 2010); therefore, as design smells are rooted in the violation of rules specified from the theoretical principles of the object-oriented paradigm, there is a strong intuition as to why design smells are strong predictors of defects in software.

As such, in the next section, we introduce the topic of code smells; we describe the most commonly studied code smells in the literature, to which we characterize as traditional code smells and to which we set for comparison in our study, then we describe the Design smells, which are the features to be evaluated in this research.

3.2. Code Smells

Code smells are defined as patterns in the source code that imply deeper problems in the system (Fowler and Beck, 1999). Their detection method is based on the violations of fundamental design principles that negatively impact design quality. They imply weaknesses in the design, which, although not technically incorrect, may lead to slower development and increase the risk of defect production. Consequently, they can contribute to the accumulation of technical debt, which can lead to a technical bankruptcy, rendering the project unmaintainable, thus having to be abandoned in the end (Suryanarayana et al., 2015; Tufano et al., 2015).

Traditional Code Smells: In the literature, the most common smells are those proposed by Fowler and Beck (1999) and Brown (1998). Fowler and Beck (1999) was the one to introduce the notion of code smells. Their primary purpose is to determine when the developer should do a specific code refactoring. They define 22 representations of code smells, each associated with a set of refactoring methods. For example, the *Shotgun Surgery* smell is detected when the developer divides a single responsibility among several classes; therefore, upon a change, he needs to update the code in several locations. When this type of pattern is not handled properly, it can lead to code regression bugs where a previously working feature stops functioning. Furthermore, Brown (1998) propose a list of negative patterns that cause development roadblocks and categorize them for the different software development roles: management, architectural, and development. Brown's primary motivation is to accurately describe commonly occurring situations, consequences, and solutions related to the three perspectives. For example, the Swiss Army Knife is a management perspective anti-pattern that describes the over-design of interfaces. It results in objects with numerous methods that attempt to anticipate every possible need, thus leading to designs that are harder to comprehend, use, and debug.

However, since the smells defined by Fowler and Brown are only conceptual descriptions of the patterns, other works made an effort to define formal methods of detection based on their description (Moha et al., 2010; Marinescu et al., 2005; Tsantalis et al., 2008; Danphitsanuphan and Suwantada, 2012). The main idea is to define rules using code metrics, define associations between them, and set thresholds. Moreover, because the thresholds are qualitatively defined (e.g., VERY-LOW), a common approach is to determine a meaningful value as the threshold mapped to a particular qualitative threshold. Thereupon, these values are derived from the statistical distribution of metrics overall projects (Arcelli Fontana et al., 2015).

As a consequence of the definition of rules, several tools were developed to extract smells. In our study, we used the Organic Project[1] which is based on Bavota et al. (2015) rules. We also used DECOR rules to detect smells and adapted them to work with Java files (Moha et al., 2010). In the end, we extract 20 smells to represent the Traditional smells used in the literature. [2]

Design Code Smells: The smells we are studying were introduced by Ganesh et al. (2013). They propose a comprehensive catalog of 31 Design smells, classified based on the violation of one of the four fundamental Object-Oriented design principles. Their primary motivation is the lack of common ground for the definition of smells. While some treat a smell as a problem itself, others consider it indicative of a deeper problem. Therefore, their goal is to create a framework that defines a taxonomy for all documented smells. In particular, to organize them under the perspective of the violation of a design principle. They base those violations on the fundamental design principles from the four major elements of the "object model" defined by Booch and Booch (2007): abstraction, encapsulation, modularity, and hierarchy. Hence, they generalized and renamed known code smells in the literature to conform to object-oriented design principles. Moreover, based on the proposed catalog of Design smells, Sharma (2018) developed Designite[3], a code assessment tool written in Java that detects 17 code smells.

One example of those smells that were proposed and implemented is the **Unnecessary Abstraction**. It is a Design smell that is detected when an abstraction is introduced in a software design, even though it is not needed (Ganesh et al., 2013). The following are the corresponding aliases to other smells in the literature, to which this Design smell is generalized.

- The **Irrelevant class**, which is detected when a class does not have any meaningful behavior in the design (Llano and Pooley, 2009).

[1]https://www.github.com/opus-research/organic.git
[2]https://www.github.com/Bruno81930/DesigniteJava
[3]https://www.designite-tools.com/designitejava/

- The **Lazy class"/"Freeloader**, which is detected when a class does "too little" (Fowler and Beck, 1999; Khomh et al., 2012).
- The **Small class**, which is detected when a class has no (or too few) variables or no (or too few) methods in it (Choinzon and Ueda, 2006; Johnson and Rees, 1992).
- The **Mini-class**, which is detected when a public, non-nested class defines less than three methods and less than three attributes (including constants) in it (Simon et al., 2006).
- The **No responsibility**, which is detected when a class has no responsibility associated with it (Budd, 2008).
- The **Agent classes**, which is detected when a class serve as an "agent" (i.e., they only pass messages from one class to another), indicating that the class may be unnecessary (Llano and Pooley, 2009).

Next, to illustrate the motivation to use Design smells for defect prediction, we show an example of how the presence of the *Unnecessary Abstraction* may induce defects after a change. The rationale for the inducing defect stems from the quality attributes that are impacted by the smell. In this instance, the Unnecessary Abstraction impacts the understandability of the system, as needless abstractions unnecessarily increase the design complexity and affect the overall design understandability. Furthermore, the smell impacts the system's reusability since a reusable abstraction requires unique and we-defined responsibilities to be appropriately used, and without it, it is less likely to be reused in a diverse context. Indifference towards these attributes leads to an increasing technical debt over new software releases. Consequently, it lowers the maintainability of the project and increases the probability of new defects being introduced (Suryanarayana et al., 2015).

This example is a concrete example from the *javax.swing*[4] package and was referenced by Suryanarayana et al. (2015) when they introduced the Unnecessary Abstraction. Listing 1 displays its code. It shows the interface *javax.swing.WindowConstants*; it defines four constants used to manage window-closing operations.

Before introducing enumerations in Java 1.5, developers used other features to define and share constants. Using inheritance had the advantage of providing the classes that implement them convenient access to all the constants without explicitly qualifying the constant values with the interface's name. Moreover, they did not constrain the implementing class of extending from another class since Java does not support multiple class inheritance. Therefore, *WindowConstants* was implemented as a "constant interface" that defines the constants to control the window-closing operations, which, was used to inherit other classes such as *JFrame*, *JInternalFrame*, and *JDialog*.

[4]https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html

Although using an interface to define and hold constants seemed like a beneficial alternative to the past non-existing enumerators, it is, in fact, a misuse of the abstraction mechanism. First, the derived classes are "polluted" with constants that may not be pertinent to the derived class. Then, every constant contains implementation details, thus exposing them through an interface violates encapsulation. Last, since the constants are part of an interface, any change to the constants can break existing implementations. We can reason how these issues may lead to defects. For instance, unnecessary constants that do not fit the interface implementing class context add extra responsibility that is not used and raise understandability and reusability issues that lead to a higher chance of development errors. Furthermore, not having well-encapsulated interfaces leads to ill-defined public interface definitions that prevent developers from easily understanding the code, thus being prone to more defects. Also, any change to a constant applied to the interface will propagate to all the implementations, which will require extra care and maintenance efforts, thus increasing the probabilities of inducing more defects.

```
 public interface WindowConstants {
  /** The do-nothing default window close operation.
     */
  public static final int DO_NOTHING_ON_CLOSE = 0;
  /** The hide-window default window close operation
     */
  public static final int HIDE_ON_CLOSE = 1;
  /** The dispose-window default window close
     operation. */
  public static final int DISPOSE_ON_CLOSE = 2;
  /** The exit application default window close
     operation. */
  public static final int EXIT_ON_CLOSE = 3;
}
```

Listing 1: WindowConstants interface taken from javax.swing package

Overlapping code smells. There is an issue concerning the smells' formularization when considering Design smells since they derive from the renaming and generalization of code smells present in the literature (also called alias). In particular, since we are comparing the effectiveness of design code smells in defect prediction compared to the traditional smells already studied for this effect, we need to clarify the distinction between the two classes of smells and specify the rationale for why Design smells qualify to predict defects. Therefore, we listed all the alias code smells used to derive each Design smells and compared their descriptions. We observed a small set of alias code smells included in the Traditional smells; however, we also observed slight distinctions between the definitions of the Design smells and the respective alias smells. Hence, inducing that given a set of alias smells, the derived code smell is a generalization of those smells that fits the violation of a particular object-oriented design principle.

One example is the insufficient modularization design smell. It is detected when an abstraction has not been completely decomposed; thus, further decomposition could reduce its size, implementation complexity, or both. Therefore, it is defined by two forms: A bloated interface, where the abstraction contains a large number of members in its public interface, and a bloated implementation, where the abstraction includes a large number of methods in its implementation or contains one or more methods with excessive implementation complexity. On the other hand, the god class, the matching overlapping alias smell, is detected when a class has 50 or more methods or attributes. Henceforth, we perceive the design smell as a generalization for the god class smell, where the definition of the alias smell is generalized to fit into the framework of violations of object-oriented principles.

All in all, we observed five design code smells whose aliases overlapped with the traditional smells. Nevertheless, we analyzed the data set with the Design and Traditional smells to access the number of occurrences where both overlapping smells were detected simultaneously. We observed that, on average, 1.32% of instances detected both overlapping smells simultaneously, compared to the 13.89% of instances with alternated detection, i.e., where either one of the smells is exclusively detected. Therefore, it emphasizes the distinction between the two smells, considering they follow an equivalent principle but are not the same. Moreover, we reran the experiment in this study without considering the overlapping smells in each class of predictions, and we verified that difference in the performance of the models is not significant.

In summary, we used code smells as features for defect prediction. We consider 20 smells commonly used in the literature as our baseline, extracted using the Organic and DECOR rules. We also consider 17 Design smells to study whether they increase the defect prediction performance. In Tables 1 and 2, we provide the description of each smell extracted using, respectively, Designite and the Traditional tools. We, respectively, categorize the collection of smells as the Design and Traditional smells. For short, we define the sets of smells as **D** for the set of Designite smells, **T** for the Traditional smells, and **D+T** for the combination of both Designite and Traditional smells. Finally, we ended up with 37 boolean features for each instance of the training set, each describing whether it was detected or not in the particular file.

4. Evaluation Methodology

Our research goal is to study the impact of Design smells on defect prediction. Therefore, we designed our study to empirically compare defect prediction classifiers' performance trained with the Design code smells against the smells traditionally used in the literature. As such, we evaluated the different models trained with both categories of smells individually and their combination – Design smells and Traditional smells – and we studied the performance of each

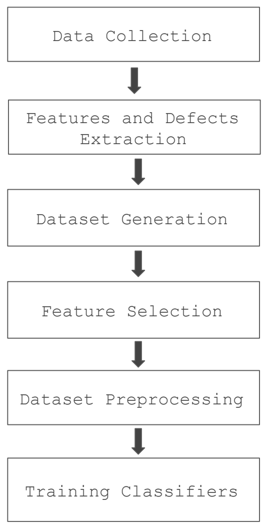


Figure 1: Overview of the methodology used in this study to build the code smells' classifiers.

category of those smells based on the Design smells taxonomy. With this in mind, we defined the following research questions.
RQ1. *Do Design code smells contribute to the performance of defect prediction models trained with Traditional code smells?*

> Defect prediction trained with smells has been thoroughly studied in the literature. However, Design smells, which are based on an object-oriented principled unifying framework, have not been studied in this scope. **RQ1** asks whether there is an improvement in the performance of defect prediction classification when based on both Design smells and the Traditional code smells from the literature.

RQ2. *How do the different categories of Design smells impact the performance of the defect prediction models?*

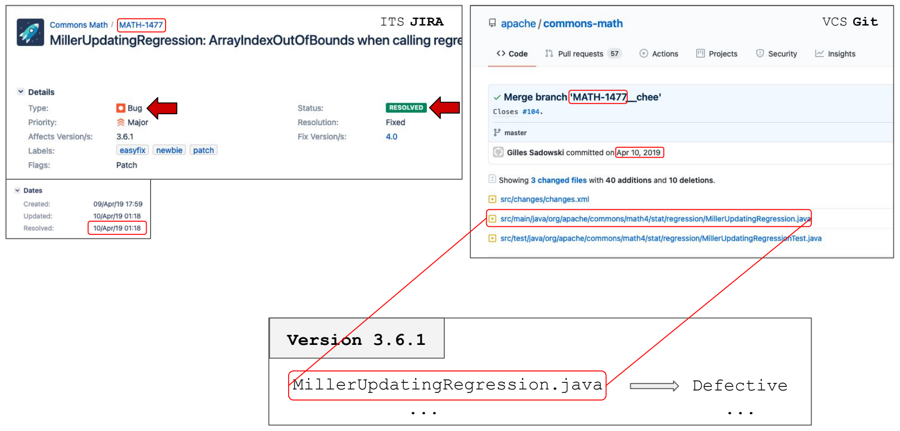
> The Design smells collection is based on the four object-oriented principles, i.e., the categories of abstraction, modularization, encapsulation, and hierarchy. Their definition already categorizes the Design smells; however, the Traditional smells must be clustered based on those categories. **RQ2** asks how do the smells in each category impact the performance of defect prediction models for the models trained with Design and Traditional smells individually and combined.

We divide this section by following the same approach used in defect prediction studies. We start by collecting the data from repositories, including smells, metrics, and defects information. Then, we apply feature selection, whose purpose is to examine which features influence the best defect prediction. Next, we train classification models to predict defects based on several algorithms and optimize them with hyper-parameter optimization. Last, we cross-validate the models and evaluate them using different classification metrics. Figure 1 outlines this methodology, described by the following subsections.

4.1. Dataset Construction

Our approach's first step is to collect the data and generate the datasets required for the classifiers training and testing. Therefore, we started by iterating each project's versions and extracting the defective information, i.e., whether each file has defects or not, and the target features.

Then, we preprocessed the datasets for training. In particular, we handled the missing information, standardized the data, and dealt with data imbalance. In the end, we split the data into training and testing datasets.



MillerUpdatingRegression: ArrayIndexOutOfBounds when calling regre

Details

Type: Bug
Priority: Major
Affects Version/s: 3.6.1
Labels: easyfix newbie patch
Flags: Patch
Status: RESOLVED
Resolution: Fixed
Fix Version/s: 4.0

Dates

Created: 09/Apr/19 17:59
Updated: 10/Apr/19 01:18
Resolved: 10/Apr/19 01:18

Merge branch 'MATH-1477__chee'

```
Closes #104.
```

master

Gilles Sadowski committed on Apr 10, 2019

Showing 3 changed files with 40 additions and 10 deletions.

src/changes/changes.xml

src/main/java/org/apache/commons/math4/stat/regression/MillerUpdatingRegression.java

src/test/java/org/apache/commons/math4/stat/regression/MillerUpdatingRegressionTest.java

```
MillerUpdatingRegression.java   ⟹   Defective
             ...                       ...
```

Figure 2: Example of the application of the defect extraction methodology.

Data Collection: We collected the data from 97 projects. In particular, we selected five versions from each project and applied the feature extraction approach on a file granularity. The criteria to select the versions was based on the ratio of defects in each file. We opted for versions with ratios between 10% and 30% of defects since they compose an adequate representation of defects that reduce the class imbalance. The low number of defects produces a class imbalance, and it is not an outlier, for instance, a version created to fix issues. The versions were selected from 97 Apache projects written in Java, and they represent different populations concerning authors, number of contributors, number of commits, and release times. In Table 5, we describe each project's respective information about the project's age, the number of contributors, and for the training and testing sets, the average number of files, number of commits, and ratio of defects. In the end, we collected features from a high number of versions,

Table 1: Description of the Design code smells extracted using Designite - **Design**. It includes a description of each smell and the category of the type of smell.

Name	Description (It is detected when...)	Category
Imperative Abstraction	An operation is turned into a class.	Abstraction
Multifaceted Abstraction	An abstraction has more than one responsibility assigned to it.	Abstraction
Unnecessary Abstraction	An abstraction which is currently not needed is introduced in a software design.	Abstraction
Unutilized Abstraction	An abstraction is left unused (either not directly used or not reachable).	Abstraction
Deficient Encapsulation	The accessibility of one or more members of an abstraction is more permissive than actually required.	Encapsulation
Unexploited Encapsulation	The client code uses explicit type checks (through chained if-elses or switch) instead of exploiting the variation in types encapsulated within a hierarchy.	Encapsulation
Missing Hierarchy	A code segment uses conditional logic to explicitly manage variation in behavior where a hierarchy could have been created and used to encapsulate those variations.	Hierarchy
Wide Hierarchy	The inheritance hierarchy is "too" wide indicating that intermediate types may be missing.	Hierarchy
Deep Hierarchy	The inheritance hierarchy is "excessively" deep.	Hierarchy
Rebellious Hierarchy	A subtype rejects the methods provided by its supertype.	Hierarchy
Broken Hierarchy	A supertype and its subtype conceptually do not share an "IS-A" relationship resulting in broken sustitutability.	Hierarchy
Multipath Hierarchy	A subtype inherits both directly as well as indireclty from a supertype leading to unnecessary inheritance paths in the hierarchy.	Hierarchy
Cyclic Hierarchy	A supertype in a hierarchy depends on any of its subtypes.	Hierarchy
Broken Modularization	The data and/or methods that ideally should have been localized into a single abstraction are separated and spread across multiple abstractions.	Modularization
Insufficient Modularization	An abstraction exists but has not been completely decomposed and a further decomposition could reduce its size, implementation complexity, or both.	Modularization
Cyclically-dependent Modularization	Two or more abstractions depend on each other directly or indirectly, thus creating a tight coupling between the abstractions.	Modularization
Hub-like Modularization	An abstraction has dependencies (both incoming and outgoing) with large number of other abstractions.	Modularization

Table 2: Description of the smells traditionally used in the literature – **Traditional**. It includes a description of each smell and the category of the type of smell.

Name	Description (It is detected when...)	Category
God Class	An object controls too many other objects in the system and has grown beyond all logic, thus becoming the class that does everything.	Modularization
Class Data Should Be Private	A class exposes its fields, thus violating the principle of encapsulation.	Encapsulation
Complex Class	A class has at least one large and complex method, in terms of cyclomatic complexity and lines of code.	Abstraction
Lazy Class	A class has few fields and methods.	Abstraction
Refused Bequest	A class is defined as abstract but has very few children, which do not make use of its methods.	Hierarchy
Spaghetti Code	A class declares long methods with no parameters and uses global variables.	Abstraction
Speculative Generality	A class is defined as abstract but has very few children which do not make use of its methods.	Abstraction
Data Class	A class only contain fields, getters and setters. A strict example when it contains only public fields.	Modularization
Brain Class	A class tends to be complex and centralize the functionality of the system.	Modularization
Large Class	A class contains many fields, methods and lines of code.	Abstraction
Swiss Army Knife	A class has an excessive number of method definitions, thus providing many different unrelated functionality.	Abstraction
Anti Singleton	A class provides mutable class variables, which consequently can be used as global variables.	Modularization
Feature Envy	An object accesses fields of another object to execute some operation.	Modularization
Long Method	A class has at least one method that is very long, in term of LOCs.	Abstraction
Long Parameter List	A class has at least one method with a list of parameters that is too long in comparison to the average number of parameters per methods in the system.	Abstraction
Message Chain	A class uses a long chain of method invocations to realise at least one of its functionality.	Modularization
Dispersed Coupling	The methods of a class access (depend on) many operations across an excessive number of classes.	Modularization
Intensive Coupling	The methods of a class excessively access operations of one or a few other classes.	Modularization
Shotgun Surgery	A single change is made to multiple classes simultaneously.	Abstraction
Brain Method	There is at least one method that centralizes the intelligence of a class.	Modularization

representing a considerable diversity of projects and a good representation of defects for each version.

Extracting Defects: After selecting the versions, the following step was to analyze the source code and collect the features under study and the target variable. First, we identified which files were defective and which were not. The applied methodology to extract the defects is a variant of the SZZ (Borg et al., 2019) algorithm. Compared to SZZ, our approach only requires locating the bug-fixing commits. All the implementation regarding the bug-inducing commits does not apply to this study's goal; hence it was skipped in the implementation. Moreover, SZZ was proposed for Bugzilla as the issue tracking system; therefore, the matching between issues and commits was based on a numerical id that identified the issue; specifically, SZZ would identify which commits fixed a specific issue if the issue id, a single number, would be matched in the commit title or message. However, Herbold et al. (2020) observed that there is a high probability of mismatching the id with another number in the commit. Therefore, we used the JIRA issue tracker instead, whose issue id follows the format <PROJECT>–<NUMBER>. Furthermore, after identifying the bug-fixing commit, another issue presented by Herbold et al. (2020) was the criterion used to select the files that were fixed. They observed that not all of the files in the change were directly related to the fix, particularly test cases, refactorings, and changes targeting comments in a file. Therefore, we filtered out test cases and files whose changes were refactorings and comments beyond selecting files written in Java.

In Figure 2, we show an example of the application of the defects extraction methodology. It contains a screen capture of the Version Control System (VCS) (i.e., Git) and the Issue Tracking System (ITS) (i.e., Jira) for a particular issue of the project *commons-math.* This example describes the methodology used to extract defects on that project by focusing on the different aspects that define the methodology. First, we highlight the *Type* and the *Status* of the ITS screen capture issue. Our methodology only selected issues referred to as a *Bug* and that was *Resolved.* Therefore, for every version of *commons-math*, we only process those particular issues. Then, in both screen captures of the tools, we highlight the id of the issue, in this example *MATH-1427*, and the date the issue was resolved, i.e., *April 10, 2019.* The goal is to match the ITS issue to the VCS, thus identifying the files involved in the fix. We first track the commits that were submitted on the resolved date. Then we match the commit that mentions the issue id from the title and the commit message. In our example, the issue id was found in the commit title. Consequently, we observe the commit changed files, and since we are only interested in the Java files that are not tests, we assign the *MillerUpdatingRegression.java* as a defective file. Last, when mapping the defective files to a version, we use the *Affects version/s* parameter in the ITS since we are interested in the files before the fix when the defect is reported.

Our study used the Jira tracker from Apache to collect the defects information and mapped a boolean value for each file asserting whether it is defective or not. Moreover, to select the five versions for each project (mentioned in the previous section), we used the Jira tracker to calculate the defective ratios of all the versions, which were used to select the versions that fall under the rate criteria 10% and 30%.

We always extract the defects from the same versions as the smells are extracted, considering the versions' history. For instance, if we consider versions $v.1$ to $v.2$, to which we extract the defects from the commits until release $v.2$, we extract the smells from version $v.1$; since the defective code is from the previous version and not the release one.

Extracting Design Smells: We extracted the main feature under study by running Designite on each version's source code. It produced a mapping for each class where, for every smell, it described whether it was detected or not. Therefore, we abstracted the class to file granularity and assumed it was detected in the file if it was detected in a class.

Extracting Traditional Smells: We extracted the smells commonly used in the literature for smell-based defect prediction using Organic[5] to analyze the existence of code smells in the source code. This tool is an eclipse plugin that detects 11 types of smells based on the rules implemented by (Bavota et al., 2015). Consequently, we adapted and extended it to detect all the smells mentioned in the previous section. In the end, we were able to detect 20 Traditional smells both from class and method granularity. Therefore, we assume that if we detect a smell in a method or a class, the whole file is marked as defective.

Feature Selection: In addition to comparing the defect prediction classifiers trained by each feature category, we aim to explore whether selecting subgroups of smells, including both Design and Traditional smells, improves the prediction performance. Therefore, wefiltered out the highly correlated set of smells, generated a subset of smells based on multiple feature selection methods, and re-applied the previously defined process to generate the datasets for each subset of features. Hence, we trained classifiers with both Design smells and Traditional smells and applied feature selection on the training set to study whether it improves the defect prediction performance. Furthermore, we study which feature selection methods select the subset of features that produce the most accurate defect prediction classifiers. We applied the following three feature selection methods based on univariate statistical tests while selecting both 20 and 50 percentile of the features: the chi-square test, the ANOVA F-value, and the mutual information. In addition, we also applied recursive feature elimination considering the F1-Score.

Dataset Preprocessing: We applied preprocessing operations from the generated datasets to prepare and split the data to create the defect prediction classifiers. One of the first required operations is to handle missing values; thus, we dropped rows with missing values. Then, in Table 3, we describe the relative frequency (in percentage) of instances, based on whether at least one smell was detected or no smell was detected (not detected), and whether the instance-mapped file is defective or not. We observed that there is data imbalance on both groups of instances where no smell was detected (with 12% instances with

[5]https://github.com/opus-research/organic.git

no smell detected) and where defects occurred (with 13% defective instances). Therefore, to address the first issue, we dropped the instances where no smell was detected since it represents a non-significant percentage of instances in our dataset, and evaluating instances with no smells does not fit this study's goal. Moreover, after splitting the data into training and test sets, by allocating the first four project versions for training and the last version for testing, we address the second issue of data imbalance for the defective files. We applied the SMOTE oversampling technique, which synthetically increases the ratio of defective instances in the training set (Chawla et al., 2002).

Table 3: Frequency distribution for the number of smells that were detected for each smell group and whether they are defective.

	Detected	Not Detected
Defective	12%	1%
Not Defective	76%	11%

Training Classifiers: We trained the defect prediction classifiers with the datasets generated from the collection step. Moreover, since we are looking to assess the Design smells' feature sensitivity, we experimented with a broad set of classifiers. We applied hyper-parameter optimization with cross-validation to validate and select the classifiers and configurations with the best performances. Then we selected the ten highest performance classifier configurations for each dataset, based on the mean accuracy of each project, and trained them, thus creating defect prediction classifiers for each dataset. We experimented with three statistical classifiers: Linear discriminant analysis, quadratic discriminant analysis, and logistic regression; a Bernoulli naive Bayes classifier; k-nearest neighbors classifier; decision tree and random forest classifiers; support vector machine; and neural network: a multilayer perceptron. For the practical application of these classifiers, we used the scikit-learn (Pedregosa et al., 2011) tool, which supports each one of them. Table 4 presents the classifiers and configurations used in this study.

In particular, we first applied a grid search with 10-fold cross-validation on all the classifiers to find the best classifiers for each target dataset, i.e., Design (D), Traditional (T), and Designite + Traditional (D+T), and to optimize the parameters for higher performance. We used the mean accuracy as the optimization score. To do so, we used scikit-learn's GridSearchCV tool, which does an exhaustive search over the specified parameters for each classifier we analyzed. After obtaining the score of all combinations of parameters and classifiers, we selected the ten configurations with the highest accuracy for each dataset. Then, we compared the performance of the D+T smells by applying two experiments. First, we trained each model for the three datasets using the classifier configuration with the highest score for each project and metric (e.g., AUC-ROC). In other words, we train the most optimal classifier for every dataset, project, and metric, thus providing an optimal comparison of the three datasets' perfor-

Table 4: Classifier's configurations considered in this study to train the defect prediction models.

Classifier	Configurations
Linear Discriminant Analysis	
Quadratic Discriminant Analysis	
Logistic Regression	**C:** {0.0001, 1, 10000}
Bernoulli Naive Bayes	
K-Nearest Neighbor	
DecisionTree	**Criterion:** {gini, entropy}
Random Forest	**Number of Estimators:** {10, 100}
Support Vector Machine	**C:** {0.1, 100}
Multilayer Perceptron	**Hidden Layer Sizes:** {(17, 8, 17)}; **Activation:** {tanh, relu}

mance, regardless of the machine learning algorithm. Second, we trained all the models using the same classifier configuration regardless of project and metric, decided from the configuration with the overall highest performance for the F1-Score. With this, we analyzed the performance of the three datasets using a consistent classifier, thus setting a baseline algorithm for all models, rendering a more accurate comparison.

4.2. Data Analysis and Metrics:

We calculated different metrics commonly studied in defect prediction and compared them to analyze the features under study to evaluate the trained models. The evaluation metrics are precision, recall, F1-Score, AUC-ROC, and the Brier Score. We discuss their rationale in the rest of the section.

Precision and recall are two widely used metrics in defect prediction. They measure the relationships between specific parameters in the confusion matrix:

$$precision = \frac{TP}{TP + FP} \qquad recall = \frac{TP}{TP + TN} \tag{1}$$

where,

- TP describes the number of classes containing defects that were correctly predicted as defective;
- TN describes the number of non-defective classes that were correctly predicted as non-defective;
- FP describes the number of non-defective classes that were incorrectly predicted as defective.

In addition, we calculated the F1-Score, i.e., the harmonic mean of both precision and recall, defined as follows:

$$F1 = 2 \times \frac{precision \times recall}{precision + recall} \tag{2}$$

Table 5: Information from the 97 projects used in this study describing each project **duration** in months (Age), **number of contributors** (#Contrb) and the **average numbers of files** (AVG#Files), **number of commits** (#Commits) and **percentage of defects** (%Defects) for the training and testing datasets used in the classification.

				AVG#Files		#Commits		%Defects	
	Project	Age(Mth)	#Cntrb	Training	Testing	Training	Testing	Training	Testing
1	camel	137	638	2396	4951	390	688	7.20	5.0
2	hadoop	127	280	4268	4710	221	324	6.75	15.3
3	flink	77	718	2549	4988	491	755	10.43	8.8
4	kafka	89	691	1588	2381	266	274	18.00	17.1
5	openmeetings	49	11	552	591	175	116	15.85	11.0
6	karaf	114	136	880	1509	200	175	7.43	6.7
7	hbase	120	301	1067	3828	225	232	19.60	11.8
8	uima-ruta	77	4	1206	1249	123	62	5.65	3.8
9	lucene-solr	226	194	1074	4973	687	260	16.85	15.6
10	deltaspike	96	53	565	1188	180	44	4.30	1.8
11	jackrabbit-oak	99	22	653	1112	708	737	19.88	15.1
12	pulsar	42	286	1212	1989	191	345	0.78	0.5
13	ofbiz	80	19	1164	1163	1002	1163	3.35	12.9
14	cayenne	149	32	3303	3740	224	147	9.00	3.7
15	commons-codec	146	27	91	124	195	91	20.68	16.1
16	parquet-mr	71	136	406	571	61	105	18.25	17.7
17	kylin	65	172	674	1391	512	94	16.93	16.0
18	hive	115	230	1856	3231	569	1199	21.55	27.2
19	commons-validator	116	27	136	141	59	57	7.55	5.7
20	maven-surefire	172	91	367	862	89	222	20.08	10.1
21	syncope	83	32	706	1945	146	167	17.15	23.8
22	commons-math	103	35	914	1467	625	168	8.10	4.3
23	tomcat	123	48	1487	1775	171	618	0.25	0.1
24	atlas	42	93	636	841	215	248	20.73	12.0
25	struts	159	43	781	1035	109	164	10.00	10.0
26	tika	120	87	364	835	185	388	18.23	10.7
27	servicecomb-java-chassis	34	94	1136	1593	153	292	16.32	5.1
28	ranger	48	77	881	949	250	135	16.62	17.1
29	cassandra	121	308	374	523	129	271	12.75	12.2
30	cxf	143	145	3334	5159	199	360	3.72	3.5
31	avro	119	166	494	650	65	141	3.05	10.6
32	nifi	62	307	2562	4835	255	377	8.12	7.3
33	bookkeeper	84	85	413	613	123	188	25.18	10.9
34	systemml	26	85	1454	1631	284	366	10.35	8.0
35	asterixdb	101	52	1630	4666	283	222	4.40	3.6
36	maven	123	129	642	961	115	32	3.35	9.4
37	zeppelin	53	323	314	443	271	329	18.03	13.8
38	commons-collections	139	51	482	536	134	64	13.93	10.4
39	jena	94	64	5939	6393	157	215	14.07	11.6
40	calcite	83	232	1528	1934	133	383	20.93	19.3
41	tez	49	27	734	1089	123	115	9.40	1.3
42	commons-lang	93	141	201	274	314	230	16.65	20.1
43	activemq	156	93	1947	3420	308	667	7.55	9.8
44	curator	105	94	405	658	141	39	16.98	10.8
45	phoenix	70	93	1171	1434	181	180	19.65	14.4
46	samza	69	119	847	1257	185	172	14.70	17.3
47	nutch	114	40	427	602	128	205	14.18	24.8
48	qpid-jms	61	18	521	535	50	101	6.05	6.5

	Project	Age(Mth)	#Cntrb	AVG#Files		#Commits		%Defects	
				Training	Testing	Training	Testing	Training	Testing
49	juneau	40	14	1134	1691	86	48	7.48	11.4
50	myfaces-tobago	168	11	581	693	287	298	7.30	3.3
51	isis	90	39	4364	3646	191	225	2.02	0.5
52	wicket	155	77	2609	3153	203	77	5.83	7.4
53	santuario-java	104	4	622	664	32	55	3.28	6.5
54	helix	89	35	650	761	111	42	16.70	8.3
55	storm	68	337	824	1879	501	232	15.77	14.2
56	airavata	46	43	1334	1389	579	2452	3.10	25.4
57	myfaces	166	26	344	446	159	37	3.80	4.0
58	commons-dbcp	141	33	83	120	147	66	24.17	14.2
59	commons-vfs	151	31	368	510	290	17	23.00	5.9
60	opennlp	83	31	712	787	175	112	9.08	6.6
61	tomee	89	95	5228	6046	429	60	13.00	27.0
62	tinkerpop	67	120	1213	1547	270	266	8.18	6.3
63	directory-server	163	18	1079	1868	1284	98	6.22	7.0
64	commons-compress	132	54	143	362	122	91	21.73	12.7
65	accumulo	85	114	1482	2007	566	505	16.25	18.3
66	giraph	53	26	555	1009	147	147	17.57	4.2
67	johnzon	66	19	169	299	32	107	9.75	21.7
68	jclouds	109	234	6714	7131	198	48	15.40	29.9
69	manifoldcf	96	11	806	824	88	20	0.40	0.2
70	shiro	115	37	599	703	57	111	3.60	2.4
71	knox	86	47	805	1191	144	376	17.25	20.9
72	drill	70	147	1470	2500	293	152	20.38	20.9
73	crunch	54	37	386	516	59	78	19.50	11.6
74	commons-io	124	59	155	227	193	142	18.25	24.7
75	commons-cli	96	34	44	48	67	32	15.18	14.6
76	jackrabbit	171	22	2001	2399	185	196	9.48	6.5
77	openwebbeans	124	17	1047	1129	173	374	12.30	9.8
78	xmlgraphics-fop	243	5	1228	2004	535	54	15.28	6.6
79	tajo	31	30	1056	1727	278	22	20.05	10.1
80	commons-email	122	22	37	47	51	44	26.18	14.9
81	directory-studio	134	10	1319	1808	546	117	6.98	2.2
82	tapestry-5	151	25	1262	1650	95	101	15.80	14.1
83	archiva	111	22	722	747	553	527	7.27	12.6
84	olingo-odata4	67	21	1906	1771	128	54	3.50	4.1
85	openjpa	135	10	1507	4481	386	88	7.78	7.4
86	commons-jexl	142	22	116	211	166	25	18.05	8.1
87	roller	149	11	611	613	196	76	6.50	19.6
88	reef	31	67	1595	1938	135	205	6.03	1.7
89	activemq-artemis	56	164	3697	4077	421	359	14.60	8.6
90	beam	66	637	2420	3530	756	731	16.70	12.6
91	metron	39	62	754	1121	101	144	14.93	10.2
92	cocoon	120	18	1911	2593	692	266	0.33	0.2
93	carbondata	46	166	893	1103	225	47	21.95	12.4
94	commons-csv	67	28	28	32	67	68	17.50	18.8
95	commons-beanutils	112	24	227	257	80	104	8.20	30.0
96	commons-net	116	19	220	254	145	267	11.75	37.4
97	continuum	76	12	494	610	177	99	11.90	12.8

Moreover, we computed the Area Under the Curve (AUC) of the Receiver Operating Characteristic curve. AUC summarises the ability of the classifier to discriminate between defective and non-defective classes. As such, the closer it is to 1, the higher the classifier's skill to discern the classes affected or not by the defect. Nevertheless, a score closer to 0.5 describes a classifier with lower accuracy, thus having a classification ability closer to a random classifier.

Furthermore, we evaluated the classifiers using the Brier Score. This score measures the distance between the probabilities predicted by a model and the actual outcome. Accordingly, the Brier Score is defined as follows:

$$\frac{1}{N}\sum_{i=1}^{N}(p_c - o_c) \tag{3}$$

Where p_c is the predicted probability by the classifier, and o_c is the actual outcome for the class c. Therefore the distance will be 0 if the class is not defective and 1 otherwise. N is the total number of classes in the dataset. A low Brier Score represents a good classifier performance, while a high score represents a low performance.

5. Results

This section discusses the obtained results, focusing on the research questions we initially defined.

To address **RQ.1**, we studied whether the performance of the models trained with both smells sets would perform better than any of the single sets of smells. We compared the D+T smells' performance by the following two approaches: (1) we used the classifier configuration with the highest score from the ten configurations for each project and metric (e.g., AUC ROC). Thus we got the highest possible score; (2) we used the same classifier configuration for **all** projects and metrics. We selected this classifier based on the configuration with the highest average F1-Score on all projects for the D+T smells. In this way, we got the results from a consistent classifier on all projects and metrics. Moreover, we evaluated different subsets of smells using feature selection to analyze whether subsets of smells trained models with better performance than the complete set. Last, we applied a robustness test to analyze each project's average and maximum error for the three sets of smells.

Experiment with the highest score from ten classifiers based on project and score:. Figure 3 compares the performance for each of the evaluated metrics on the three sets of smells: Design, Traditional, and Design + Traditional. These models were trained using the classifier configuration with the highest value for each project and evaluation metric from the ten top configurations selected in the hyperparameter optimization stage. Therefore, we sought to understand how different are the average performances for each set of smells. Overall, we can verify that there is an increase in the performance of D+T compared to the T smells. In particular, it is more prominent in the AUC

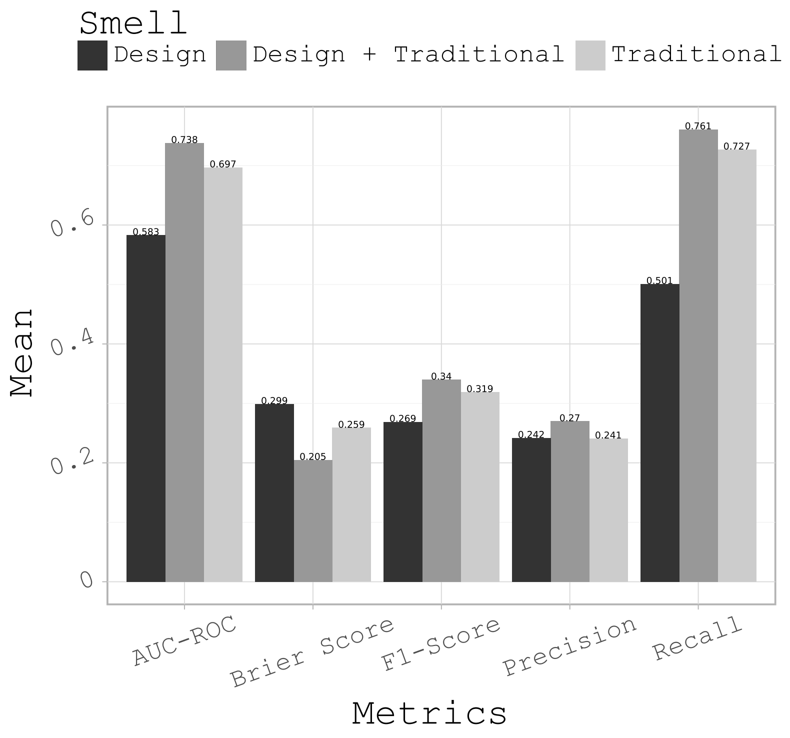


Figure 3: Score comparison between Design Models, Traditional Models and their combination for the best classifier of each project and metric.

metric, where we see an improvement of 4.1%. Moreover, the precision and the recall also show better performances for the D+T. We verified that the recall is significantly larger than precision for both variables, with results close to 76% for the D+T.

This difference is better represented in the distribution presentation in Figure 4. This figure illustrates a violin plot and a box plot to compare the different scores' distributions: AUC ROC, the Brier Score, and the F1-Score, between the models trained with D+T, smells, and those only trained with T. The violin plot displays the rotated probability density of the score on all projects, and the box plot describes the standardized summary of all projects' scores. We observe a significant improvement in the distribution (AUC) when the Design smells are added to the Traditional. Moreover, although there is an improvement in the F1-Score distribution with higher performance for the D+T smells, this improvement is not considerable. Lastly, the Brier Score is 4.6% lower for the D+T, which means it has a higher performance than the T smells separate. Furthermore, we observe a bimodal distribution on the T smells in the Brier

Score. It shows a concentration of projects whose performance is deficient; however, the higher concentration of projects shows more increased performance.

In general, for the D+T dataset, all categories perform similarly, while in the individual datasets, there are considerable differences between abstraction and modularization against encapsulation and hierarchy.

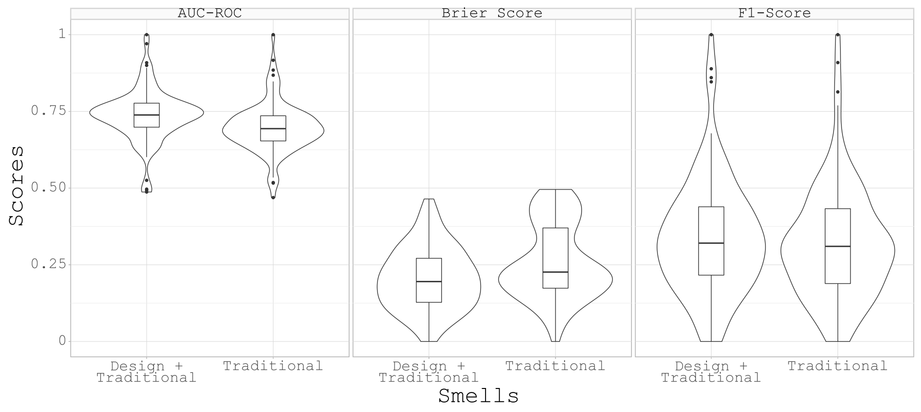


Figure 4: Distribution of the models trained with both **Design and Traditional Smells** compared with only **Traditional Smells** for three scores: AUC ROC, Brier Score and F1-Score.

The plots are relatively normalized for both the AUC score and the F1-Score, despite having small clusters of outlier projects on both distribution edges in both D+T and T smells of the AUC score and in the D+T distribution of the F1-Score. The distribution peak for the D+T smells compared to the T smells is visibly higher for the AUC score, thus showing better performance for the most significant concentration of projects. Conversely, for the F1-Score, the peaks for both D+T smells and T smells are on the same level. In the end, based on both the AUC and Brier score, we verify an overall performance improvement when both Design smells and Traditional smells are trained together.

In addition to this analysis, we checked the statistical significance with the paired t-test and the Mann-Whitney test and calculated the effect size using the Pearson r correlation and Cohen's d (Cohen, 2013). We confirmed that the models trained with D+T smells have higher performances for all the scores than those trained with only the T smells, at a significance level of $p < 0.05$.

Moreover, we measured Pearson r correlation of 0.4 for the AUC, defined as a medium-strength association, while the F1-Score measured a large-strength association of 0.8, and the Brier Score, a small-strength association of 0.2. Next in order, we measured Cohen's d of 1.4 for the AUC, thus determining a large effect size between both variables and a small Cohen's effect size of 0.4 for both F1-Score and Brier Score.

For a fine-grained understanding of the classifiers' improvement, we calculated the AUC difference between D+T and T for each project, thus getting a perspective on each project's particular improvements. We observed an im-

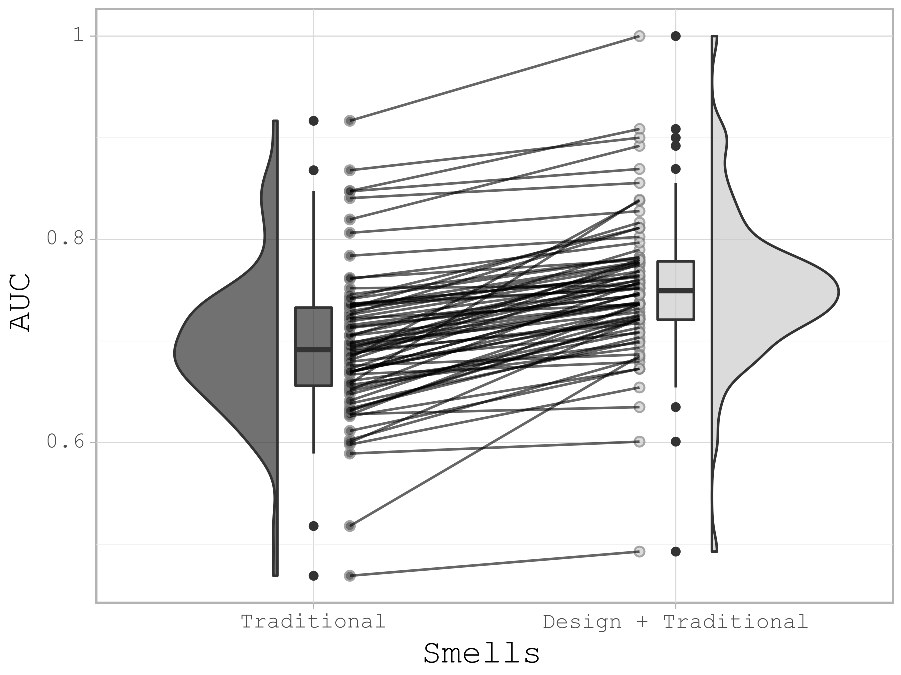


Figure 5: Improvement for each project between Design and Design + Traditional smells.

provement on 76 out of 97 projects (78%), as we introduced the Design smells to the Traditional. In particular, out of the improved projects, there was an average improvement of 5.65% in the AUC, with the highest improvement being 18.0% and the lowest 0.24%.

In Figure 5, we display the distribution of the AUC of the projects to which we observed an improvement (76 projects) by combining the Design smells (D) to the Traditional smells (D+T). For each set, this figure contains a violin plot representing each project's distribution and a box plot describing the distribution's limits, percentiles, and mean values. Moreover, it contains several points representing the AUC for each specific project; the line connecting each set's points describes the improvement of the combination of D to T. We observe that it was a significant improvement for most projects where there was an improvement. It enforces the conclusion that adding the Design smells to the Traditional smells leads to better defect prediction performances. Moreover, it gives us a finer-grained perspective on the improvement of each project.

Experiment with a consistent classifier for all projects, based on the configuration with the highest overall F1-Score from all projects:. Next, we present the performance based on a consistent classifier across all projects and metrics rather than the experiments presented above with different classifier configurations. We searched for the classifier configuration with the best overall F1-Scores for the majority of the projects. We compared the performance of the models trained with the three sets of smells under study. We found that the Support Vector Machine with a regularization parameter of $C = 0.1$ had the configuration that performs the best for most projects for the F1-Score metric and the AUC. In Figure 6, we show the comparison of the arithmetic averages for all projects of the different scores for the three smells sets: D, T, and D+T. We can verify that the different models' trends are very similar to the ones on the best classifiers (Figure 3). Although, they show smaller performances, including an improvement of 3.4%, rather than 4.1%, in the AUC when the Design smells are used in conjunction with the Traditional smells (D+T).

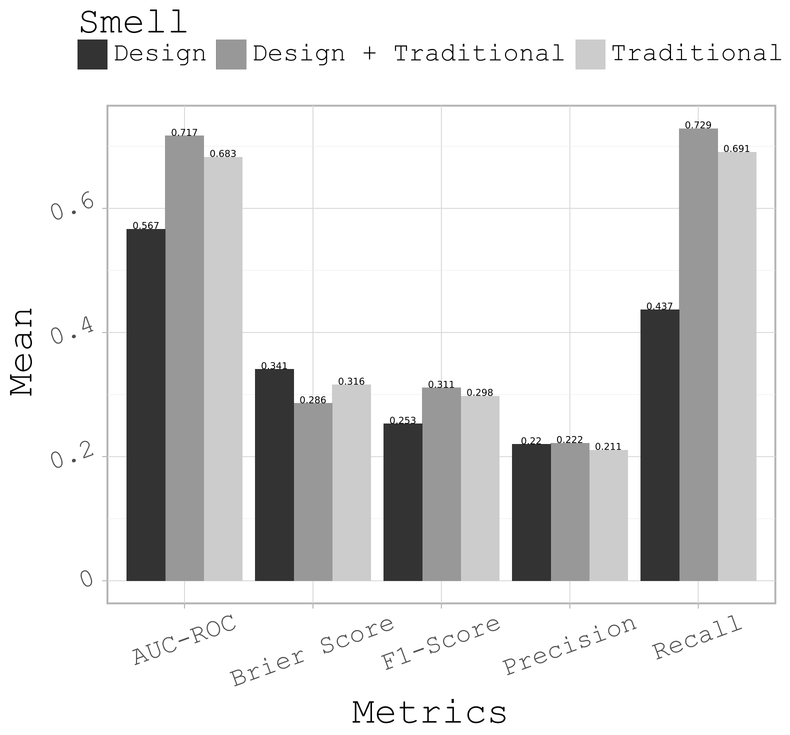
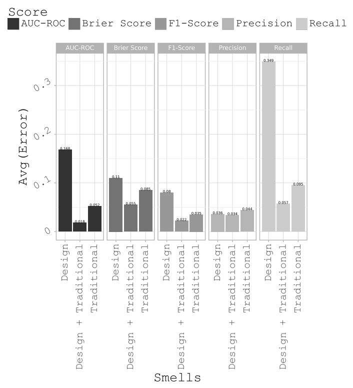
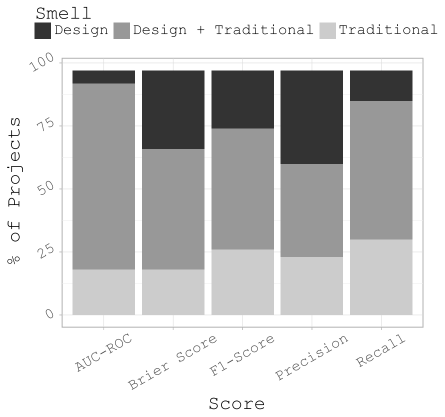


Figure 6: Score comparison between Design Models, Traditional Models and their combination for overall best classifier configuration – **Support Vector Machine** $C = 0.1$.

Experiment with the robustness of the model's performance for each project:. Moreover, we used the results from the models trained with the SVM to test our results' robustness. The goal is to understand better how consistent the different sets of smells are on average for each project. In particular, for each project, we analyze how each smell set's score competes against an oracle, which is defined by the highest score among the three sets of smells (D, T, and D+T). As such, we calculate the difference between each set of smells and the oracle for each particular project, thus producing the error that describes how far each set is from achieving the best result possible for that project. For instance, assume in project Camel that the F1-Score for D is 0.16, T is 0.27, and D+T is 0.28. Consequently, the oracle is the set with the highest score, i.e., the combination of both D+T smells – 0.28. The errors are 0.12, 0.01, and 0, thus showing how distant each set from the best possible score for this project is. In Figure 7a we illustrate the average error for all projects. We can see that the set D+T consistently has the lowest error across all metrics. Moreover, it is further explained in Figure 7b, which presents the distribution of the oracles for the projects, i.e., the percentage of projects for each class of the smells with the highest performance, for each metric. We verify significantly higher percentages of oracles corresponding to the Design + Traditional smells. This result's significance is that if we consistently use the Design + Traditional smells, we can achieve the best prediction on average.



(a) Average error for the difference between the each dataset of smells and the correspondent oracle for each project.



(b) Distribution of the oracles by the projects, for each metric. It represents the sets of smells with highest performances in each project.

Figure 7: Robustness Test.

Experiment with the feature selection methods over the different datasets:. Beyond studying the performance contribution of Design smells to the smells used traditionally in the literature, we also explore whether subsets from the combination of smells would perform better than the original set of D+T smells. To this end, we applied seven different methods of feature selection to identify the subsets of features with the best performances. Nevertheless, we first removed the highly correlated smells from the data since their inclusion would have impacted the experiment's validity. In particular, we measured the Pearson coefficient and characterized them as highly correlated code smells if the coefficient surpassed 0.5. In table 5, we describe the pair of smells with high

correlation and their respective coefficient. In addition, to validate the exclusion of the correlated metrics, we built the defect prediction model for the D and T smells without each column of the highly correlated smells and compared the models' performance with the original D+T code smells' trained model. Henceforth, the scores difference between the model trained with all the smells and those trained without one column of the highly correlated smells showed an improvement of 0.08 for one of the models and a regression of 0.01 for the other.
Given the data set without the correlated features, Figure 8 shows the distribution of the difference in AUC between each feature selection method and the combination of all smells. In gray, we show the distributions representing the improvements over using all the smells, and in black, we show the distributions where using all smells performs better than the selection. We observed that feature selection for the AUC ROC slightly improves some models' performances compared against the original models, with all features of D+T code smells. All in all, between 35% and 64% of projects showed an improvement over all the selection techniques, with an average improvement for those projects compared to using all the smells ranging between 2.3% and 5.6%. The mutual information with 20 and 50 percentile showed an improvement on 65% and 56% projects, thus showing an average improvement of 5.6% and 4.2%. The chi-squared test with 20 and 50 percentile had 63% and 46% of projects showing an improvement with an average improvement of 5.0% and 2.4%. The ANOVA F-value with 20 and 50 percentile showed an improvement on 61% and 45% projects with an average improvement of 4.7% and 2.3%. The recursive elimination showed the lowest improvement rate with 35% of improvement projects and an average of 2.9% improvement compared to using all the smells. Moreover, the feature-selection results for the F1-Score and the Brier Score showed a consistent pattern with the AUC ROC scores. Between 39% and 55% of projects showed a better F1-Score after applied feature selection, with an average difference between 1.7% and 6.3%, and between 20% and 57% of projects showed a better Brier Score with an average difference between 4.0% and 10.7%.

Smell 1	**Smell 2**	**Pearson's r**
Missing Hierarchy	Unexploited Encapsulation	0.78
Class Data Should Be Private	Deficient Encapsulation	0.68
Large Class	Insufficient Modularization	0.58
Long Method	Complex Class	0.54
Lazy Class	Unutilized Abstraction	0.52
Deep Hierarchy	Cyclic Hierarchy	0.51

Table 5: Pearson correlation coefficient for pair of code smells with high correlation ($r > 0.5$)

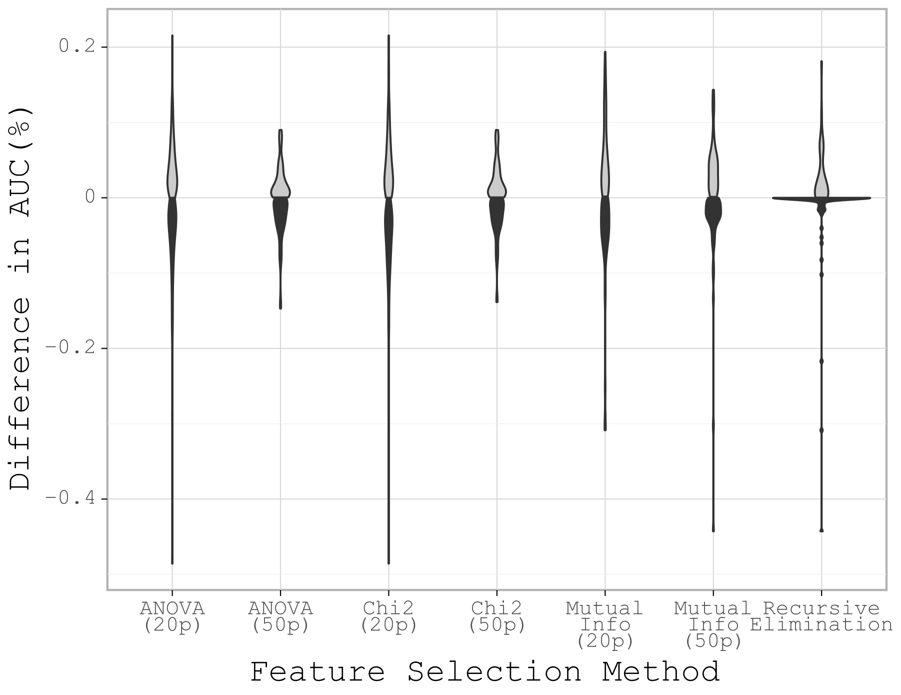


Figure 8: Difference in AUC between each feature selection method compared with all smells, for the set of Design + Traditional smells.

RQ2. explores the impact of each of the categories of Design code smells (i.e., abstraction, encapsulation, modularization, and hierarchy) on the performance of smell-based defect prediction. As such, we clustered the combination of both the Design and Traditional smells based on those categories. We considered the mapping provided by Ganesh et al. (2013) for the categorization of Design smells. In addition, we clustered Traditional smells based on the description of each smell, which is described in Table 2. Tables 1 and 2 describe the mapping of each smell to the particular cluster, respectively, for the D+T smells. These categories are well known in the literature, from the principles of object-oriented programming, and were previously used to classify the smells in Table 1 (Suryanarayana et al., 2015). Therefore, we used the description from the smells and the taxonomies to associate each Traditional smell to the respective category in Table 2. We classified the models using the Support Vector Machine with a regularization parameter of $C = 0.1$. Because we previously verified that using the best classifier for each project and score, and using the best classifier consistently across projects and scores, produces a similar pattern of results.

In Figure 9, we represent the arithmetic mean of the scores of all projects, based on the four Design code smells categories. We observe that for the datasets

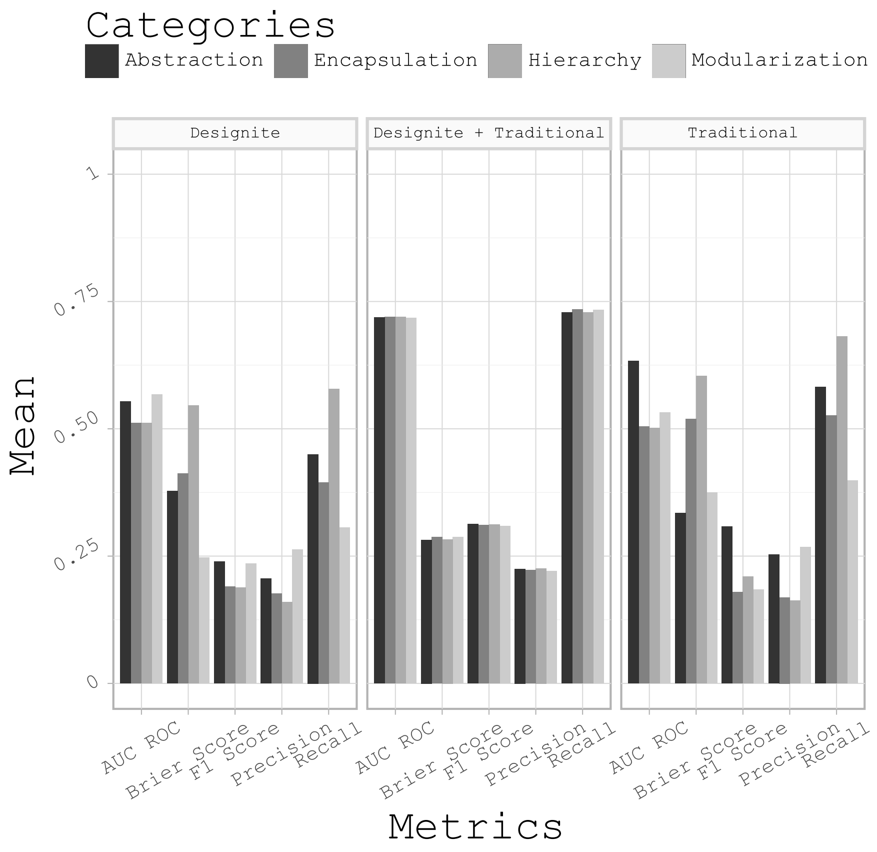


Figure 9: Performances for each category of the Design code smells, for models using the three datasets: Design, Traditional and the combination of both.

with only Design and only Traditional smells, the encapsulation and hierarchy categories scores are significantly lower than the abstraction and modularization categories. We reason that the encapsulation and hierarchy smells for the individual datasets do not provide meaningful information. However, as we observe the scores for the D+T smells, we verify that all categories show results in approximately the same range, with values superior to the individual sets of Design smells and Traditional smells. Hence, we reason that the information from the encapsulation and hierarchy categories for Design smells and Traditional smells complement each other, thus providing meaningful information when used together.

Furthermore, beyond evaluating the different categories within the smells, we analyzed the importance of each code smell. To obtain a clearer understandability of the impact of each smell on the overall improvement in each category.

Therefore, we calculated the importance of each code smell using linear principal component analysis over all the features with 20 principal components, which captures 95% of the explained variance. In particular, PCA builds relevant features by linearly transforming correlated variables into smaller uncorrelated variables, where the projected data describes the linear combinations of the original data that capture most of its variance. To measure the importance using the linear PCA, we tracked the loading of each feature, which are the weights of the computer eigenvectors' that define the principal components' orientation relative to the original variables. Henceforth, we used the tool published by Taskesen (2020) to compute the linear PCA and measure the loadings. In Table 6, we enumerate the most important features ordered by their loading over the principal components with the highest variance. Furthermore, we do not include the features that did not achieve absolute maximum loading (i.e., weak loading).

	Code Smell	**Group**	**Category**
1.	Lazy Class	Traditional Smells	Abstraction
2.	Unutilized Abstraction	Design Smells	Abstraction
3.	Feature Envy	Traditional Smells	Modularization
4.	Long Parameter List	Traditional Smells	Abstraction
5.	Message Chain	Traditional Smells	Modularization
6.	Insufficient Modularization	Design Smells	Modularization
7.	Broken Hierarchy	Design Smells	Hierarchy
8.	Cyclic Dependent Modularization	Design Smells	Modularization
9.	Refused Bequest	Traditional Smells	Hierarchy
10.	Dispersed Coupling	Traditional Smells	Modularization
11.	Unnecessary Abstraction	Design Smells	Abstraction
12.	Speculative Generality	Design Smells	Abstraction
13.	Intensive Coupling	Traditional Smells	Modularization
14.	Data Class	Traditional Smells	Modularization
15.	Class Data Should Be Private	Traditional Smells	Encapsulation
16.	Spaghetti Code	Traditional Smells	Abstraction

Table 6: List of code smells ordered by their importance and respective smell group - Design or Traditional, and smell category - Abstraction, Modularization, Hierarchy, and Encapsulation.

In summary, the results from the study of the impact of Design smells in defect prediction are the following:

- The models trained with D+T smells outperform those trained with only T smells on all scores, with a significant increase of the AUC of 4.1% ($p < 0.05$).
- We observed an improvement on 78% of the projects when adding the Design smells to the Traditional smells. In particular, we observed an average improvement of 5.65% in the AUC, ranging from 0.24% to 18%.
- Applying feature selection to the D+T smells produces models with slightly higher accuracy than models trained with all the smells. For instance, the Brier Score had the most significant increase of 1.6% $\sim$ 3.6%.
- In general, for the D+T dataset all categories perform similarly, while in the individual datasets there are considerable differences between abstraction and modularization against encapsulation and hierarchy.

6. Threats To Validity

For our study, we identified the following threats to validity.

Our evaluation is constrained to the *comparison of Design code smells with the Traditional smells* discussed in the literature. We do not compare the performance of defect prediction models trained with other metrics found in the literature, like code metrics or process metrics. However, our study's scope is to assess the performance of the Design smells compared to the smells already evaluated for defect prediction. Comparing the Design smells with other metrics should fall under future work.

The projects we used to extract the smells were limited to *open-source projects from Apache written in Java*, which is a threat to the generalization of our results. However, several defect prediction studies have used projects from Apache as the software archive (Hosseini et al., 2019) and, in addition, projects from Apache have been integrated into the Promise data set (Jureczko and Madeyski, 2010). The use of the *Jira as the issue tracking system* is also a threat to validity towards the result's generalization. However, their use is coupled with the issue tracking system of the Apache projects.

We used the *tool provided by the authors that proposed the Design smells*, which could be a threat to validity since we are assuming that the tool is reliable. The same applies for the *tools used to extract the Traditional smells*. However, the authors adapted and validated them, which could also be a threat considering the possibility of experimental or validation errors. Nevertheless, we provide all of the source code used in this study and the extracted data sets, allowing for external reproducibility and validation.

The algorithm we used to label the defects raises the possibility for non-defective code to be labeled as defective due to post-release commits being the bug introducing commits. For instance, given versions v:1 to v:2, since we want to extract the defects from the commits until release v:2, we extract the smells from version v:1. Therefore, it could lead to a defect being raised in the commits after the release of the version. In turn, it would label the non-defective code, to which we extracted the smells, as defective. However, we have strong reasons to assume that such cases are not significant and would not invalidate the validity of our results. When bug issues are open in most software development practices, they are assigned to releases and not commits. In particular, since reported bugs are mainly observed in major releases and not specific commits. For instance, when a user downloads and uses specific software, he does it from a released version. Nevertheless, even if their development practices may assign defects to commits after releasing a version, among all the evaluated projects and considered versions and bug issues, the produced noise is not significant enough to impact the validity of the observed results.

7. Conclusion and Future Work

This study evaluated the Design code smells as a predictor for defect prediction. We compared the models trained with smells taken from previous studies (Traditional smells). We extracted the smells from 97 projects and used a broad range of classifier algorithms to train models with the Design smells, the Traditional smells and, Design + Traditional smells. In the end, we evaluated and retrieved the scores from the best classifiers and compared the performance of the different smells.

The results showed that Design code smells are not good enough to independently be used for defect prediction. However, when used in combination with other Traditional smells used in the literature (Design + Traditional), they improve the defect prediction performance compared against each category. Moreover, we explored the impact of the smells when clustered by Design code smells categories and observed a significant improvement of both encapsulation and hierarchy code smells when considered for the combination of both Design and Traditional smells, compared against when used individually.

The current results suggest that for future work, we should further explore the factors that influenced the improved results for the combination of Design and Traditional smells, for instance, by looking into the encapsulation and hierarchy smells. We should also extend this study to cover cross-project learning, thus evaluating how effectively we can capture the learning from these smells by one project to apply to another project.

CRediT authorship contribution statement

Bruno Sotto-Mayor: Conceptualization, Methodology, Programming on software, Data handling, Investigation, Writing - original draft, Reviewing and editing. **Amir Elmishali**: Methodology, Programming on software, Reviewing and editing. **Meir Kalech**: Funding acquisition, Supervision, Reviewing and editing. **Rui Abreu**: Supervision, Reviewing and editing.

Declaration of competing interest

The authors declare that they have no known competing interest or personal relationships that could have appeared to influence the work reported in this paper.

Data Availability

The datasets generated during and/or analysed during the current study are available in the public data repository, https://zenodo.org/record/4103861.

Code Availability

The software developed during the current study is available from the public data repository at the website of https://zenodo.org/record/4110652.

Acknowledgements

This work was supported by the Cyber Security Research Center at Ben-Gurion University of the Negev and the ministry of science grant No. 102784.

References

Arcelli Fontana, F., Ferme, V., Zanoni, M., Yamashita, A., 2015. Automatic metric thresholds derivation for code smell detection, in: 2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics, pp. 44–53.

Bavota, G., De Lucia, A., Di Penta, M., Oliveto, R., Palomba, F., 2015. An experimental investigation on the innate relationship between quality and refactoring. Journal of Systems and Software 107, 1–14. URL: `https://linkinghub.elsevier.com/retrieve/pii/S0164121215001053`, doi:10.1016/j.jss.2015.05.024.

Booch, G., Booch, G., 2007. Object-Oriented Analysis and Design with Applications. The Addison-Wesley Object Technology Series. 3rd ed ed., Addison-Wesley.

Borg, M., Svensson, O., Berg, K., Hansson, D., 2019. Szz unleashed: An open implementation of the szz algorithm - featuring example usage in a study of just-in-time bug prediction for the jenkins project, in: Proceedings of the 3rd ACM SIGSOFT international workshop on machine learning techniques for software quality evaluation, Association for Computing Machinery. p. 7–12. URL: `https://doi.org/10.1145/3340482.3342742`, doi:10.1145/3340482.3342742. publisher-place: Tallinn, Estonia.

Brown, W.J., 1998. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. Wiley.

Budd, T., 2008. Introduction to object-oriented programming. Pearson Education India.

Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P., 2002. Smote: Synthetic minority over-sampling technique. Journal of Artificial Intelligence Research 16, 321–357. doi:10.1613/jair.953.

Chidamber, S., Kemerer, C., 1994. A metrics suite for object oriented design. IEEE Transactions on Software Engineering 20, 476–493. URL: `http://ieeexplore.ieee.org/document/295895/`, doi:10.1109/32.295895.

Choinzon, M., Ueda, Y., 2006. Detecting defects in object oriented designs using design metrics. 7th Joint Conference on Knowledge-Based Software Engineering , 61–72URL: `www.scopus.com`. cited By :12.

Cohen, J., 2013. Statistical power analysis for the behavioral sciences. Academic press.

D'Ambros, M., Bacchelli, A., Lanza, M., 2010. On the impact of design flaws on software defects, in: 2010 10th International Conference on Quality Software, pp. 23–31. doi:10.1109/QSIC.2010.58.

Danphitsanuphan, P., Suwantada, T., 2012. Code smell detecting tool and code smell-structure bug relationship, in: 2012 Spring Congress on Engineering and Technology, pp. 1–5. doi:10.1109/SCET.2012.6342082.

Fowler, M., Beck, K., 1999. Refactoring: improving the design of existing code. The Addison-Wesley object technology series, Addison-Wesley.

Ganesh, S., Sharma, T., Suryanarayana, G., 2013. Towards a Principle-based Classification of Structural Design Smells. The Journal of Object Technology 12, 1:1. doi:10.5381/jot.2013.12.2.a1.

Hassan, A.E., 2009. Predicting faults using the complexity of code changes, in: 2009 IEEE 31st International Conference on Software Engineering, IEEE. pp. 78–88. URL: `http://ieeexplore.ieee.org/document/5070510/`, doi:10.1109/ICSE.2009.5070510.

Herbold, S., Trautsch, A., Trautsch, F., Ledel, B., 2020. Issues with SZZ: An empirical assessment of the state of practice of defect prediction data collection. arXiv:1911.08938 [cs] URL: `http://arxiv.org/abs/1911.08938`. arXiv: 1911.08938.

Hosseini, S., Turhan, B., Gunarathna, D., 2019. A systematic literature review and meta-analysis on cross project defect prediction. IEEE Transactions on Software Engineering 45, 111–147. doi:10.1109/TSE.2017.2770124.

Johnson, P., Rees, C., 1992. Reusability through fine-grain inheritance. Software: Practice and Experience 22, 1049–1068. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380221203, doi:https://doi.org/10.1002/spe.4380221203.

Jureczko, M., Madeyski, L., 2010. Towards identifying software project clusters with regard to defect prediction, in: Proceedings of the 6th International Conference on Predictive Models in Software Engineering - PROMISE '10, ACM Press. p. 1. URL: http://portal.acm.org/citation.cfm?doid=1868328.1868342, doi:10.1145/1868328.1868342.

Khomh, F., Di Penta, M., Guéhéneuc, Y.G., Antoniol, G., 2012. An exploratory study of the impact of antipatterns on class change-and fault-proneness. Empirical Software Engineering 17, 243–275.

Lessmann, S., Baesens, B., Mues, C., Pietsch, S., 2008. Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings. IEEE Transactions on Software Engineering 34, 485–496. URL: http://ieeexplore.ieee.org/document/4527256/, doi:10.1109/TSE.2008.35.

Li, Z., Jing, X.Y., Zhu, X., 2018. Progress on approaches to software defect prediction. IET Software 12, 161–175. URL: https://digital-library.theiet.org/content/journals/10.1049/iet-sen.2017.0148, doi:10.1049/iet-sen.2017.0148.

Llano, M., Pooley, R., 2009. Uml specification and correction of object-oriented anti-patterns, in: 4th International Conference on Software Engineering Advances, ICSEA 2009, Includes SEDES 2009, pp. 39–44. URL: https://ieeexplore.ieee.org/xpl/conhome/5298192/proceeding, doi:10.1109/ICSEA.2009.15. international Conference on Software Engineering Advances 2009, ICSEA 2009 ; Conference date: 20-09-2009 Through 25-09-2009.

Ma, W., Chen, L., Zhou, Y., Xu, B., 2016. Do We Have a Chance to Fix Bugs When Refactoring Code Smells?, in: 2016 International Conference on Software Analysis, Testing and Evolution (SATE), IEEE. pp. 24–29. URL: http://ieeexplore.ieee.org/document/7780189/, doi:10.1109/SATE.2016.11.

Marinescu, C., Marinescu, R., Mihancea, P., Ratiu, D., Wettel, R., 2005. iplasma: An integrated platform for quality assessment of object-oriented design., in: Proceedings of the 21st IEEE International Conference on Software Maintenance - Industrial and Tool volume, ICSM 2005, 25-30 September 2005, Budapest, Hungary, pp. 77–80.

McCabe, T., 1976. A Complexity Measure. IEEE Transactions on Software Engineering SE-2, 308–320.

URL: http://ieeexplore.ieee.org/document/1702388/, doi:10.1109/TSE.1976.233837.

Moha, N., Gueheneuc, Y.G., Duchien, L., Le Meur, A.F., 2010. DECOR: A Method for the Specification and Detection of Code and Design Smells. IEEE Transactions on Software Engineering 36, 20–36. URL: http://ieeexplore.ieee.org/document/5196681/, doi:10.1109/TSE.2009.50.

Moser, R., Pedrycz, W., Succi, G., 2008. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction, in: Proceedings of the 13th International Conference on Software Engineering - ICSE '08, ACM Press. p. 181. URL: http://portal.acm.org/citation.cfm?doid=1368088.1368114, doi:10.1145/1368088.1368114.

Nagappan, N., Ball, T., 2005. Use of relative code churn measures to predict system defect density, in: Proceedings of the 27th International Conference on Software Engineering - ICSE '05, ACM Press. p. 284. URL: http://portal.acm.org/citation.cfm?doid=1062455.1062514, doi:10.1145/1062455.1062514.

Palomba, F., Zanoni, M., Fontana, F.A., De Lucia, A., Oliveto, R., 2016. Smells Like Teen Spirit: Improving Bug Prediction Performance Using the Intensity of Code Smells, in: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE. pp. 244–255. URL: https://ieeexplore.ieee.org/document/7816471/, doi:10.1109/ICSME.2016.27.

Palomba, F., Zanoni, M., Fontana, F.A., De Lucia, A., Oliveto, R., 2019. Toward a Smell-Aware Bug Prediction Model. IEEE Transactions on Software Engineering 45, 194–218. URL: https://ieeexplore.ieee.org/document/8097044/, doi:10.1109/TSE.2017.2770122.

Paterson, D., Campos, J., Abreu, R., Kapfhammer, G.M., Fraser, G., McMinn, P., 2019. An Empirical Study on the Use of Defect Prediction for Test Case Prioritization, in: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), IEEE. pp. 346–357. URL: https://ieeexplore.ieee.org/document/8730206/, doi:10.1109/ICST.2019.00041.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E., 2011. Scikit-learn: Machine learning in Python. Journal of Machine Learning Research 12, 2825–2830.

Piotrowski, P., Madeyski, L., 2020. Software Defect Prediction Using Bad Code Smells: A Systematic Literature Review, in: Poniszewska-Marańda, A., Kryvinska, N., Jarząbek, S., Madeyski, L. (Eds.), Data-Centric Business and Applications. Springer International Publishing. volume 40 of *Lecture Notes on Data Engineering and Communications Technologies*, pp. 77–99. URL: `http://link.springer.com/10.1007/978-3-030-34706-2_5`, doi:10.1007/978-3-030-34706-2.

Sharma, T., 2018. DesigniteJava. URL: `https://zenodo.org/record/2566861`, doi:10.5281/ZENODO.2566861.

Simon, F., Seng, O., Mohaupt, T., 2006. Code-Quality-Management: technische Qualität industrieller Softwaresysteme transparent und vergleichbar gemacht. dpunkt-Verlag.

Suryanarayana, G., Samarthyam, G., Sharma, T., 2015. Refactoring for Software Design Smells: Managing Technical Debt. Elsevier, Morgan Kaufmann, Morgan Kaufmann is an imprint of Elsevier.

Taba, S.E.S., Khomh, F., Zou, Y., Hassan, A.E., Nagappan, M., 2013. Predicting Bugs Using Antipatterns, in: 2013 IEEE International Conference on Software Maintenance, IEEE. pp. 270–279. URL: `http://ieeexplore.ieee.org/document/6676898/`, doi:10.1109/ICSM.2013.38.

Taskesen, E., 2020. pca is a python package that performs the principal component analysis and makes insightful plots. URL: `https://doi.org/10.5281/zenodo.5745418`, doi:10.5281/zenodo.5745418. If you use this software, please cite it using these metadata.

Tsantalis, N., Chaikalis, T., Chatzigeorgiou, A., 2008. Jdeodorant: Identification and removal of type-checking bad smells, in: 2008 12th European Conference on Software Maintenance and Reengineering, IEEE Computer Society, USA. p. 329–331. URL: `https://doi.org/10.1109/CSMR.2008.4493342`, doi:10.1109/CSMR.2008.4493342.

Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., Poshyvanyk, D., 2015. When and Why Your Code Starts to Smell Bad, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, IEEE. pp. 403–414. URL: `http://ieeexplore.ieee.org/document/7194592/`, doi:10.1109/ICSE.2015.59.