UNIVERSIDADE FEDERAL DE ALAGOAS

INSTITUTO DE COMPUTAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

MASTER'S THESIS

EXPLORING THE RELATIONSHIP BETWEEN SMELLS, BUGS AND HARMFUL CODE THROUGH TRANSFER LEARNING

MASTER CANDIDATE

DURVAL PEREIRA CÉSAR NETO

ADVISOR

BALDOINO FONSECA DOS SANTOS NETO, DR.

MACEIÓ, AL

AUGUST - 2023

Catalogação na Fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico
Bibliotecário: Marcelino de Carvalho Freitas Neto – CRB-4 - 1767

A447u Cesar Neto, Durval Pereira.
Exploring the relationship between smells, bugs and harmful code through transfer learning / Durval Pereira Cesar Neto. – 2023.
57 f. : il.

Orientador: Baldoino Fonseca dos Santos Neto.
Dissertação (mestrado em Informática) - Universidade Federal de Alagoas. Instituto de Computação. Maceió, 2023.

Bibliografia: f. 49-57.

1. Qualidade de *software*. 2. Engenharia de *software*. 3. Transferência de aprendizagem. 4. Aprendizado de máquina. I. Título.

CDU: 004.81:159.953.5

UNIVERSIDADE FEDERAL DE ALAGOAS/UFAL
Programa de Pós-Graduação em Informática – PPGI
Instituto de Computação/UFAL
Campus A. C. Simões BR 104-Norte Km 14 BL 12 Tabuleiro do Martins
Maceió/AL - Brasil CEP: 57.072-970 | Telefone: (082) 3214-1401

Folha de Aprovação

DURVAL PEREIRA CESAR NETO

COMPREENDENDO CÓDIGO NOCIVO POR MEIO DE TRANSFER LEARNING

Dissertação submetida ao corpo docente do Programa de Pós-Graduação em Informática da Universidade Federal de Alagoas e aprovada em 28 de agosto de 2023.

Banca Examinadora:

gov.br
Documento assinado digitalmente
BALDOINO FONSECA DOS SANTOS NETO
Data: 29/08/2023 14:23:47-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. BALDOINO FONSECA DOS SANTOS NETO
UFAL – Instituto de Computação
Orientador

gov.br
Documento assinado digitalmente
MARCELO COSTA OLIVEIRA
Data: 30/08/2023 09:41:58-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. MARCELO COSTA OLIVEIRA
UFAL – Instituto de Computação
Examinador Interno

gov.br
Documento assinado digitalmente
LEOPOLDO MOTTA TEIXEIRA
Data: 29/08/2023 14:48:06-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. LEOPOLDO MOTTA TEIXEIRA
UFPE-Universidade Federal de Pernambuco
Examinador Externo

Resumo

A presença de *code smells* em um projeto de software é um forte indicativo da baixa qualidade do mesmo no contexto de sua implementação e em uma parcialidade de casos podem estes mesmos *code smells* serem os trechos de código nocivos para a aplicação, tornando-se os culpados na geração de *bugs*. Atualmente, existem diferentes abordagens para a detecção de *code smells* e recentemente houve um maior aprofundamento na análise da correlação entre estes *code smells* e a nocividade dos mesmos para o código, mas ainda há muito a ser pesquisado no contexto de como podemos melhorar a acurácia na detecção destes códigos potencialmente nocivos. Pensando nisto, este trabalho visa ampliar os métodos para detecção de códigos nocivos utilizando o aprendizado por transferência para construir um grande conjunto de dados para treinamento e validação dos modelos de aprendizagem de máquina.

Abstract

The presence of code smells in a software project is a strong indication of its low quality in the context of its implementation, and in many cases, these same code smells can be harmful code segments for the application, becoming the culprits in bug generation. Currently, there are different approaches for detecting code smells, and there has been a recent deeper analysis of the correlation between these code smells and their harmfulness to the code. However, there is still much to be researched in the context of how we can improve the accuracy of detecting these potentially harmful codes. With this in mind, this work aims to expand the methods for detecting harmful code by using transfer learning to build a large dataset for training and validating machine learning models.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

The software life cycle is a continuous cycle, always returning to its beginning after reaching its end, with stages like development and maintenance being repeatedly executed [MHD97, Rup10]. However, this cycle is not free from challenges [RPTU84]. The pursuit of error-free and easily maintainable code often encounters obstacles in the form of bugs - unexpected errors that can disrupt the proper functioning of a program [FBF+20, SZZ05]. Furthermore, code quality is also influenced by code smells, subtle hints that indicate potential issues with the structure or design of the source code [BSC+20, OGS+16, SOG+17, OSO+18, UBO+20, BUC+20].

Software defects, more commonly known as bugs, constitute an ongoing challenge in the software development industry [FBF+20, BCDL+12, GZNM10]. These undesired errors can arise due to a variety of reasons, from logic mistakes to implementation issues. The presence of bugs can result in data loss, system malfunctions, and even security risks. As the complexity of software systems grows, identifying and rectifying bugs becomes an increasingly crucial task, leading to innovative approaches to enhance code quality and system reliability. Various approaches have been employed to automatically identify bugs in systems, such as the utilization of the SZZ algorithm [NDCK19], machine learning approaches [Son19], and search-based methods [PA21].

Code smells refer to coding practices that indicate potential quality issues in the source code. These subtle hints may not directly result in errors, but they can compromise the readability, maintainability, and healthy evolution of the software [UBO+20, BUC+20, FBF+20]. Identifying and addressing code smells is essential to avoid the accumulation of technical debt [LBT+21], which can lead to significant challenges in the long term. Early detection and correction of these issues can contribute to clearer, sustainable, and easily maintainable code. Currently, there are different approaches for detecting code smells - using various techniques such as static code analysis [UBO+20], change metrics [CKK+18], and with the intention of proposing rules and patterns in detecting these smells as well as understanding the impact of their existence in code on the quality and maintainability of software.

Considering these significant factors, a contemporary trend has gained prominence: the integration of advanced Machine Learning (ML) techniques as a potential solution to address the challenge of detecting symptoms of poor-quality code, such as code smells and bugs. Different approaches, such the use of Decision Trees [SG13], Support Vector Machines [PANVA09], Random Forest [TLNH19], and Naive Bayes [BHA14] are the most used for this purpose. Nevertheless, there exist only a limited number of studies that have explored the application of transfer learning to detect code smells [SELS19a, SELS21] and bugs [ZLTW20, DZYX20]. It differs from traditional machine learning and semi-supervised algorithms in that it takes into account the possibility that the domains of the training and test data may differ. Conventional machine learning algorithms predict future data using mathematical models that are trained on previously collected labeled or unlabeled data, which is assumed to be similar to future data [FAMZM16]. On the other hand, transfer learning permits variations in domains, tasks, and distributions used for training and testing. Transfer learning also is inspired by the human ability to apply prior knowledge to solve new but similar problems with greater efficiency. In machine learning, the primary objective of transfer learning is to develop lifelong learning methods that can preserve and reuse previously acquired knowledge.

1.1 Motivation

In various software development scenarios, it is interesting to observe how the same indicative signs of code smells can evolve to become parts of the code that cause substantial issues in the application. This means that code segments that initially exhibited characteristics suggesting poor practices or design deficiencies can, under certain circumstances, transform into actual culprits for the occurrence of bugs.

Detecting code smells, bugs, and harmful code in diverse programming languages with minimal effort is a pressing necessity in contemporary software engineering [TSLHS18, PZAF+19, Kau20]. The software development landscape is highly polyglot, with projects often spanning multiple languages [NWG12, YLC22]. As a result, developers face the challenge of maintaining code quality and identifying potential issues across this linguistic diversity. The ability to automate the detection of code smells, and bugs in different languages streamlines the development process, reduces manual effort, and enhances software reliability [LLC22]. This not only saves time and resources but also helps ensure consistent code quality and security practices, irrespective of the programming language employed. In a world where multilingual software development is increasingly common, tools and techniques that facilitate cross-language code analysis become indispensable for developers and researchers alike.

However, an even more intriguing approach to understanding the complex relationship between code smells and bugs is the concept of **harmful code**. Following the perspective presented by Lima *et al.* [Rod20], the term harmful code refers to code segments that are associated with errors and also exhibit one or more characteristics of code smells, indicating that they still possess problematic aspects related to their structure, maintenance, or performance. In their work, Lima *et al.* [Rod20] conducts a thorough study of understanding and identifying harmful code. They aim to identify the presence of harmful code generated by code smells and bugs

When it comes to detecting multi-language issues in software development, leveraging machine learning techniques has emerged as a potent approach [NLWS20]. Recent endeavors in this field have demonstrated the effectiveness of employing machine learning, particularly transfer learning, to identify individual problems like code smells and bugs within isolated programming languages [dSR20]. In the context of this landscape, our work stands out by aiming to explore the unexplored territories of cross-domain transfer learning. We seek to assess how well these machine learning techniques can be applied to transition from the detection of bugs to code smells, code smells to bugs, and ultimately harmful code to harmful code across different programming languages. This holistic perspective not only sheds light on the transferability of knowledge across language barriers but also advances our understanding of the interconnected nature of code quality issues in the multifaceted realm of software development.

1.2 Objective

The aim of this study is to enhance the approaches for detecting code smells, bugs, and harmful code through the application of transfer learning. This approach will be employed in building a database that will serve as a reliable oracle, enabling a more precise and comprehensive analysis of the characteristics that can indicate quality issues in the source code.

1.2.1 Specific Objectives

Based on the established overall objective, we have formulated the following specific objectives as means to achieve it:

- Compile a database of *code smells*.
- Compile a database of *bugs*.
- Build a machine learning model trained to detect *code smells* from a *bugs* database.
- Build a machine learning model trained to detect *bugs* from a *code smells* database.
- Build a machine learning model trained to detect harmful code from a database trained with both *code smells* and *bugs*.

1.3 Work Structure

Finally, the work is organized as follows:

In Chapter 2, a theoretical framework will be presented, addressing the main computational topic of this proposal: i) code smells and bugs; ii) harmful codes; and iii) transfer learning. In Chapter 3, we provide the main study on transfer learning and harmful codes. In Chapter 4, we introduce the second study, which explores the ability of code segments exhibiting code smells to identify the occurrence of bugs, and vice versa. Lastly, Chapter 5 contains a conclusion on the material presented, as well as a discussion on potential directions for future research.

Chapter 2

Theoretical Framework

2.1 Bugs

2.2 Code smells

Code smells are symptoms of quality issues that can impact various aspects of a software system's quality [OGC+15, OGC+14, OGS+14]. When a software system contains excessive smells, it becomes challenging to maintain and enhance. Code smells are categorized into three main types: implementation smells, design smells, and architecture smells [OGS+16, OSO+18, OSO+19, SS18, SELS19b], based on their scope, granularity, and impact. Implementation smells are typically limited in scope and impact, often affecting individual methods. Examples include long method, complex method, long parameter list, and magic number [Mar99]. Design smells, on the other hand, occur at a higher level of granularity, often affecting abstractions like classes or sets of classes. Some common examples of design smells are God class, multifaceted abstraction, wide hierarchy, and insufficient modularization [Mar99].

While being a pervasive issue in software development, they also manifest across various programming languages, including Java, C#, and C++. In Java, common code smells often arise due to complex program structures, leading to decreased code maintainability [SOG+17]. For instance, Multifaceted Abstraction might be evident when a class combines disparate functionalities, such as handling both user interface and data access logic. On the other hand, Wide Hierarchy in Java can manifest as deep inheritance chains, making the codebase harder to navigate. Finally, Long Method smells are characterized by overly complex and lengthy functions, making code comprehension difficult. C#, being another object-oriented language, shares similarities with Java [Mok03]. Multifaceted Abstraction might occur when a class handles multiple concerns, like combining file I/O and network communication. Moreover, Insufficient Modularization might be seen in poor namespace organization, leading to a cluttered project structure. C++, on the other hand, the code smells like Multifaceted Abstraction could be found in classes that mix low-level memory management with high-level algorithmic operations. Insufficient Modularization might involve inadequate separation of header and implementation files. Wide Hierarchy might result from deep template hierarchies. Long Method code smells in C++ can be identified by excessively long and complex functions.

2.3 Harmful code

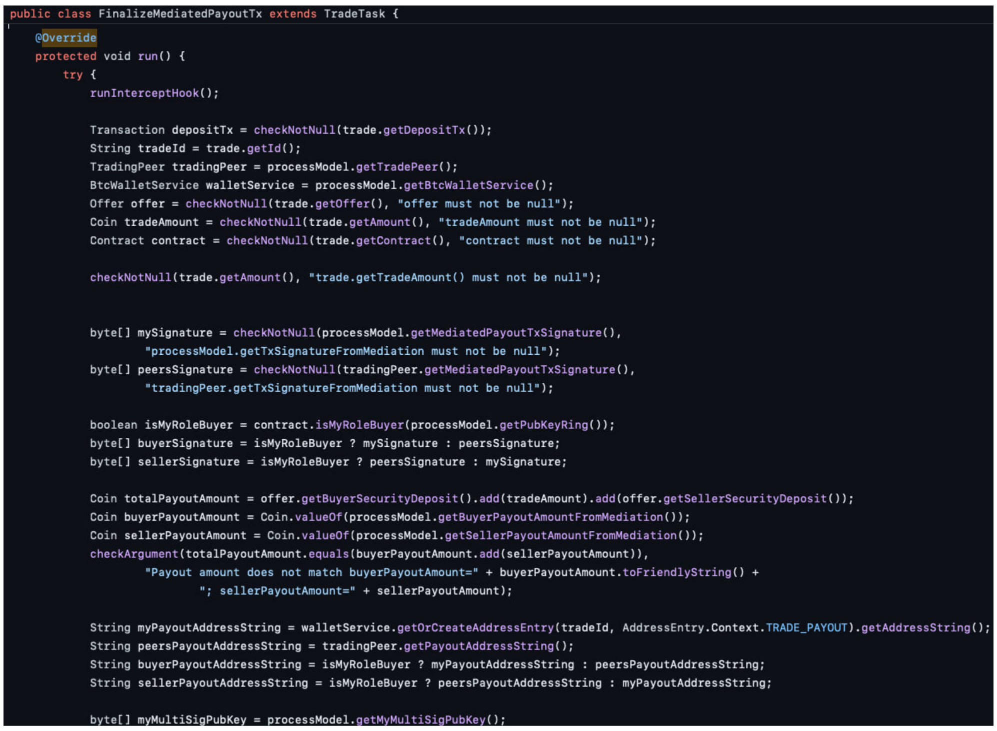
In their work, Lima *et al.* [Rod20] define the level of harmfulness of code segments in four categories, as follows:

2.3.1 Clean segment

A code segment that, at the present time, displays no evidence of any detected code smells or bugs within its source code. This implies that the segment has undergone analysis for potential issues or inconsistencies utilizing a specific methodology (see Section 3.1.3). In the context of bug detection, our methodology relies on bug reporting. Therefore, code is considered "bug-free" if, up to the present moment, no bugs have been reported or discovered within it.

2.3.2 Smelly segment

A code segment that has been flagged for having one or more code smells, suggests potential design or implementation issues. However, it's important to note that despite the presence of these code smells, this specific code segment has never been reported or identified as having one or more bugs. In other words, it has not been associated with actual



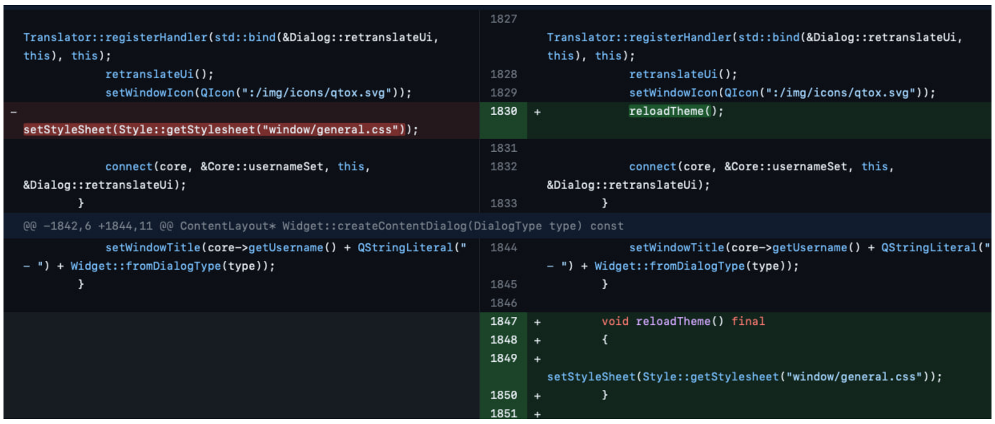
```java
public class FinalizeMediatedPayoutTx extends TradeTask {

    @Override
    protected void run() {
        try {
            runInterceptHook();

            Transaction depositTx = checkNotNull(trade.getDepositTx());
            String tradeId = trade.getId();
            TradingPeer tradingPeer = processModel.getTradePeer();
            BtcWalletService walletService = processModel.getBtcWalletService();
            Offer offer = checkNotNull(trade.getOffer(), "offer must not be null");
            Coin tradeAmount = checkNotNull(trade.getAmount(), "tradeAmount must not be null");
            Contract contract = checkNotNull(trade.getContract(), "contract must not be null");

            checkNotNull(trade.getAmount(), "trade.getTradeAmount() must not be null");


            byte[] mySignature = checkNotNull(processModel.getMediatedPayoutTxSignature(),
                    "processModel.getTxSignatureFromMediation must not be null");
            byte[] peersSignature = checkNotNull(tradingPeer.getMediatedPayoutTxSignature(),
                    "tradingPeer.getTxSignatureFromMediation must not be null");

            boolean isMyRoleBuyer = contract.isMyRoleBuyer(processModel.getPubKeyRing());
            byte[] buyerSignature = isMyRoleBuyer ? mySignature : peersSignature;
            byte[] sellerSignature = isMyRoleBuyer ? peersSignature : mySignature;

            Coin totalPayoutAmount = offer.getBuyerSecurityDeposit().add(tradeAmount).add(offer.getSellerSecurityDeposit());
            Coin buyerPayoutAmount = Coin.valueOf(processModel.getBuyerPayoutAmountFromMediation());
            Coin sellerPayoutAmount = Coin.valueOf(processModel.getSellerPayoutAmountFromMediation());
            checkArgument(totalPayoutAmount.equals(buyerPayoutAmount.add(sellerPayoutAmount)),
                    "Payout amount does not match buyerPayoutAmount=" + buyerPayoutAmount.toFriendlyString() +
                            "; sellerPayoutAmount=" + sellerPayoutAmount);

            String myPayoutAddressString = walletService.getOrCreateAddressEntry(tradeId, AddressEntry.Context.TRADE_PAYOUT).getAddressString();
            String peersPayoutAddressString = tradingPeer.getPayoutAddressString();
            String buyerPayoutAddressString = isMyRoleBuyer ? myPayoutAddressString : peersPayoutAddressString;
            String sellerPayoutAddressString = isMyRoleBuyer ? peersPayoutAddressString : myPayoutAddressString;

            byte[] myMultiSigPubKey = processModel.getMyMultiSigPubKey();
```

Figure 2.1: Long Method smelly code segment

software defects or issues reported by users or developers. This indicates a distinct pattern where code quality issues (smells) have been detected, but no related functional issues (bugs) have been reported, highlighting the potential disconnect between code quality and software reliability. We can see an example of the smelly segment in Figure 2.1.

2.3.3 Buggy segment

A code segment with bugs refers to a portion of code that was removed when a bug is fixed (bug-fixing commit [SZZ05]). It's important to note that, in this case, no code smells were detected at this point in the code's history. This observation underscores a unique scenario where issues related to the code's functionality (bugs) have been encountered, but no design or implementation problems (code smells) were identified. This lack of code smells may suggest that, while functional defects have been addressed, the code's overall quality has remained unaffected by typical code quality issues. We can see an example of the buggy segment in Figure 2.2.



```
Translator::registerHandler(std::bind(&Dialog::retranslateUi,
this), this);
            retranslateUi();
            setWindowIcon(QIcon(":/img/icons/qtox.svg"));
-
setStyleSheet(Style::getStylesheet("window/general.css"));

            connect(core, &Core::usernameSet, this,
&Dialog::retranslateUi);
        }
@@ -1842,6 +1844,11 @@ ContentLayout* Widget::createContentDialog(DialogType type) const
            setWindowTitle(core->getUsername() + QStringLiteral("
- ") + Widget::fromDialogType(type));
        }
```

```
Translator::registerHandler(std::bind(&Dialog::retranslateUi,
this), this);
            retranslateUi();
            setWindowIcon(QIcon(":/img/icons/qtox.svg"));
+           reloadTheme();

            connect(core, &Core::usernameSet, this,
&Dialog::retranslateUi);
        }
            setWindowTitle(core->getUsername() + QStringLiteral("
- ") + Widget::fromDialogType(type));
        }

+       void reloadTheme() final
+       {
+
setStyleSheet(Style::getStylesheet("window/general.css"));
+       }
+
```

Figure 2.2: Buggy code segment (left) - Bug Fixed (right)

2.3.4 Harmful segment

A smelly code segment, denoting that it has been associated with one or more code smells, could also be considered buggy. This means that it has encountered one or more reported bugs at this point in its history. This combination of code smells and historical bug reports characterizes the segment as both smelly and buggy, and this combination is what we call *HARMFUL* code.

In part of our work, we aim to detect the presence of *HARMFUL* code segments.

2.4 Transfer learning

Machine learning involves various techniques for sharing and adapting knowledge from one specific task in a domain to a broader task in the same domain. For instance, a healthcare provider employs predictive modeling to anticipate patient re-admissions, enabling early intervention and improving patient care outcomes. On the other hand, human beings demonstrate a unique capability, the ability to transfer knowledge across related domains to efficiently address novel challenges. This human-like approach becomes particularly advantageous when the new task shares fundamental similarities with the existing knowledge, enabling us to expedite problem-solving by leveraging our prior insights.

Transfer learning, at its core, involves transferring knowledge acquired in one source task to enhance learning in a related target task [LWQ17]. One of the major advantages of transfer learning is being valuable in scenarios characterized by a scarcity of training data [TS10]. When collecting ample training data for a target task proves challenging and resource-intensive, we can identify a source task with similar underlying characteristics and access to a vast training dataset. Subsequently, we train a machine learning model on this source task, utilizing the abundant dataset, and then fine-tune the model on the target task, leveraging the available yet limited training data. This strategic process empowers us to harness prior knowledge effectively, significantly improving performance on the target task. For instance, in traditional machine learning, typically, the same dataset is used for training and testing for each task [SJP09]. Figure 2.3 illustrates the difference between the two processes, and how using the transfer learning process, multiple pre-trained models for different tasks can be harnessed to achieve a model that solves the target task.

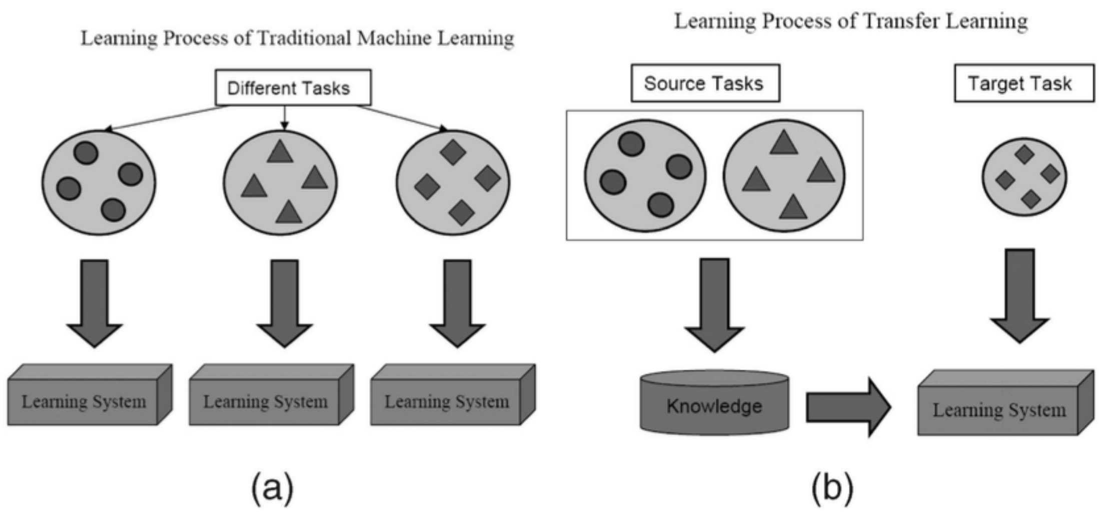


Figure 2.3: Difference between: (a) traditional machine learning and (b) transfer learning in machine learning [SJP09]

2.5 Related Work

2.5.1 Code smell to Bug

Takahashi *et al.* [TSLHS18] proposed a technique to improve bug localization by using code smells. The authors found that by using the content of a bug report, containing a summary and description as inputs for an existing bug localization approach and detecting the code smells of the source code with inFusion, it is possible to calculate the score of each module and with that improve the traditional bug localization approach by 142.25% and 30.50% on average for method and class levels, respectively.

In a related academic endeavor, Ubayawardana [UK18] directed attention towards utilizing code smells as a potential metric for constructing a bug prediction model. This involved considering both traditional source code metrics and metrics derived from code smells as proposed in existing literature. The research proceeded by training the model using diverse iterations of 13 distinct Java-based open-source projects and subsequently leveraging this trained model to forecast bugs within a specific version of a project. The findings of this study illuminated the insufficiency of relying solely on source code metrics to forecast project bugs. Instead, greater predictive accuracy was attainable by amalgamating code smell-based metrics with traditional source code metrics.

Palomba *et al.* [PZAF+19] built a specialized bug prediction model for smelly classes. Specifically, the authors evaluated the contribution of a measure of the severity of code smells by adding it to existing bug prediction models based on both product and process metrics and comparing the results of the new model against the baseline models, creating a smell-aware prediction model which combines product, process and smell-related information. They observed that the addition of an intensity index (*i.e.,* a metric that quantifies the severity of code smells) as a predictor of buggy components of the software generally increases the performance of baseline bug prediction models.

In Kaur *et al.*'s [Kau20] literature review, a substantial body of prior research has been examined, focusing on code smells and quality attributes. The findings from these studies primarily revolve around the exploration of fault-proneness and defect-proneness as external quality attributes. However, it's important to note that the analysis in these studies primarily centers on the influence of code smells on these quality attributes rather than considering them collectively as a single entity, often referred to as "harmful code."

2.5.2 Bug to Smell

On the other hand, previous work [PZAF+19, PZF+16] assesses bug prediction models specifically tailored for classes exhibiting code smells. The authors embarked on evaluating the impact of incorporating a measure quantifying the severity of code smells into existing bug prediction models, which relied on both product and process metrics. By juxtaposing the outcomes of this new model with baseline models, they formulated a smell-aware prediction model that seamlessly amalgamates product, process, and smell-related information. Significantly, their observations unveiled that the integration of an intensity index, a metric gauging the severity of code smells, as a predictive factor for buggy software components, consistently elevates the performance of baseline bug prediction models.

While preceding research has approached the subject from distinct angles, examining: i) the application of transfer learning to detect code smells, and ii) the utilization of various machine learning techniques to identify problematic code instances, our study uniquely integrates both facets. Our approach not only evaluates the efficacy of identifying bugs using code smell instances but also evaluates the efficacy of using a dataset of bugs to detect the presence of code smells. This allows us to explore the practicality of employing transfer learning in detecting poor-quality structures of code and points of possible defects.

Furthermore, where previous work in transfer learning has primarily focused on different levels of granularity, such as source files [SELS19b, KSV+22, UK18], our investigation advances at the code expressions level. This perspective enables us to enhance the comprehensiveness and contextual understanding of our analysis.

2.5.3 Harmful Code

It was Lima *et al.* [Rod20] that later presented the term *harmful code* to determine a code snippet that already had one or more bug reported, *i.e.,* that is both *smelly* and *buggy*. In this work, the authors evaluated machine learning techniques to classify code harmfulness in 13 different Java projects, concluding that while the Random Forest is effective in classifying both *smelly* and *harmful code*, the Gaussian Naive Bayes is the less effective technique.

2.5.4 Transfer Learning

Earlier investigations have concentrated on exploring the viability of transfer learning within the scope of code smell detection. Sharma et al. [SELS19b] delved into the utilization of deep learning models for identifying code smells and examined the potential of applying transfer learning to this area. They further evaluated the performance of deep learning models within the transfer learning context. This research introduced a novel paradigm that employs transfer learning to detect code smells, particularly beneficial for programming languages where comprehensive code smell detection tools are scarce.

In a separate endeavor, Kovacevic *et al.* [KSV^{+}22] conducted an experiment focused on the automated detection of specific code smells, namely the Long Method and God Class, given their frequent occurrence during development. Distinguishing themselves from Sharma *et al.* [SELS19b], their study diverges on two key fronts, instead of training a deep-learning model for the same smell detection task across different programming languages, they opted to transfer the knowledge captured by code understanding models. Moreover, they employed a fully manually labeled code smell dataset as opposed to automatic labeling, thus sidestepping the learning of imperfect automatic code smell detectors.

Also, Ardimento *et al.* [AAB^{+}21] investigated whether the adoption of a transfer learning approach can be effective for just-in-time design smell prediction. The approach used a variant of Temporal Convolutional Networks to predict design smells and carefully selected fine-grained process and product metrics. The empirical results show that when the class is well-balanced the prediction model is effective for direct learning and is usable as an alternative with comparable results. Moreover, its results also demonstrated that transfer learning provides F1 scores very close to the ones obtained by direct learning.

While prior works consider, separately: i) the application of transfer learning for code smell detection and ii) the use of other machine learning techniques to identify harmful code, we combine both factors into our study. We not only evaluate the effectiveness of code harmfulness identification but also use the generated dataset to train a model, allowing us to understand the feasibility of transfer learning in harmful code detection, which could be beneficial, especially for programming languages where there aren't tools available for this process. Finally, while prior work in transfer learning conducts their analyses at different granularity levels, such as source file [SELS19b, KSV^{+}22], we conduct our analyses at the commit level, taking into account the historical aspects of software development.

Previous efforts [SELS19b,KSV^{+}22,AAB^{+}21,CSCT15] have primarily centered around assessing the application and practicality of transfer learning in the domain of code smell detection. In contrast, our study distinguishes by leveraging transfer learning for smell detection using a labeled dataset of bugs, as well as a trained dataset with labeled smells to detect bugs. By using this data, we are also able to compound a dataset of harmful code to detect other instances of harmful code in different programming languages.

Chapter 3

Is your code harmful too? Understanding harmful code through transfer learning

One of the main points of our work in understanding the relation between bugs and code smells through transfer learning, as pointed out in 2.3, is to investigate if is feasible to identify harmful code using transfer learning techniques. Thus, in this chapter, we present our main study: *"Is your code harmful too? Understanding harmful code through transfer learning"*.

This work reports a retrospective study aimed at using transfer learning to assess code harmfulness across various software projects and programming languages. It was built on a previous approach [Rod20] and [dSR20], categorizing code into clean, smelly, buggy, and harmful types based on code smells and reported bugs. The study gathered source code from 23 open-source projects, analyzing the historical development of more than 250k versions mined from these projects. It also examined five code smells, including design-type and implementation-type issues, using class-level and method-level granularity.

In conclusion, our study explored using transfer learning to detect harmful code across multiple programming languages, focusing on Java, C#, and C++. We found that knowledge transfer between Java and C# for various code smells was promising, while C++ posed more challenges. Despite limited data, a sample size of 32 showed positive outcomes for most code smells, though some situations required larger samples, like with Multifaceted Abstraction. Importantly, our research addressed a gap in the field by assessing transfer learning's application for harmful code detection using insights from different software systems. This insight guides software engineering researchers, stressing suitable sample sizes and understanding transfer learning intricacies for more accurate detection across diverse languages. Leveraging transfer learning's potential can boost software quality and development practices.

3.1 Main Study Design

In this section, we describe our research questions, the statistical approach and data collection procedures (Sections 3.1.2, 3.1.3 and 3.1.4), and the quantitative approach (Section 3.1.6.

Recent studies [SELS19b, KSV+22, KM19, AAB+21, DSPPDL21] have showcased the potential of transfer learning in software engineering, particularly concerning code smells detection and prediction. While various studies have proposed automatic approaches for detecting code smells and some have explored understanding harmful code [Rod20], there remains a significant research gap as none of these works have focused on assessing how knowledge from other software systems can be utilized for detecting harmful code.

To this goal, we collected source code from 23 open-source projects to create a dataset in which each project has historical granularity since all the history of software development is being taken into account (code segments for each commit version). We also evaluated the presence of five code smells: *Multifaceted Abstraction*, *Insufficient Modularization*, *Wide Hierarchy*, *Long Method*, and *Complex Method*, with the first three being a design-type with class-level granularity and the former two being implementation-type with method-level granularity.

Finally, to address this gap, our study aims to explore the application of transfer learning for detecting harmful code using five different types of code smells across three programming languages. By doing so, we seek to contribute new insights and advancements in the realm of code analysis and software quality assurance. Our research questions are designed to shed light on the effectiveness and adaptability of transfer learning techniques in this domain, offering valuable guidance for improving code analysis tools and promoting more efficient and accurate harmful code detection practices. We present them as follows:

RQ$_1$: How effective is transfer learning in detecting harmful code?

This research question seeks to evaluate the effectiveness of transfer learning (measured by the f1-score) of harmful code in open-source projects from the three different programming languages. Through this analysis, we expect to ascertain the extent to which transfer learning can be employed to generalize and identify harmful code in a more language-agnostic manner.

The insights garnered from this research have significant implications for the software development community. As harmful code poses serious security and reliability risks, the ability to efficiently detect it across different programming languages can lead to more robust and secure software systems. Moreover, our findings will provide valuable guidance to practitioners and researchers on optimizing transfer learning techniques for code analysis tasks, ultimately fostering safer and more reliable software development practices.

RQ$_2$: How efficient is transfer learning to detect code smells?

To address the RQ2, we delve into a comprehensive analysis of the effort required for transfer learning to effectively detect harmful code. We comprehend efficiency as the number of instances (sample size) comprising the training dataset utilized for training a deep learning model.

Our investigation encompasses an exploration of varying sample sizes, ranging from small to large (2 and 630, respectively), to ascertain how transfer learning's performance is impacted. We meticulously evaluate the model's effectiveness in detecting code smells across different programming languages as the sample size changes. Additionally, we evaluate potential trade-offs associated with different sample sizes. Larger datasets may improve model generalization and capture more diverse patterns, but they also require increased computation and time. Conversely, smaller datasets may lead to overfitting or suboptimal performance on specific programming languages.

Our findings will not only shed light on the ideal sample size for effective transfer learning but also provide insights into the generalizability and adaptability of the approach across diverse programming languages. As a result, our research contributes to the development of best practices in utilizing transfer learning for code smell detection, facilitating more efficient and scalable approaches for ensuring code quality and maintainability in software projects.

3.1.1 Programming Languages

In this study, we selected three prominent programming languages, namely Java, C#, and C++, for our comprehensive analysis. The rationale behind selecting this specific trio of programming languages lies in their prevalence and significance in various domains of software development. Java, known for its platform independence and widespread use in enterprise applications, holds a dominant position in the industry. C# is particularly relevant in the Microsoft ecosystem and game development, while C++ is widely employed in-system programming and performance-critical applications.

By including these diverse languages in our analysis, we aim to gain a comprehensive understanding of how transfer learning techniques perform across different programming paradigms and use cases. Each language comes with its own set of unique features, syntax, and programming conventions, which can significantly impact the effectiveness of transfer learning for code analysis.

The insights gained from this study will be invaluable for developers, researchers, and practitioners working on code analysis and software quality assurance. Understanding the strengths and limitations of transfer learning in different programming languages can lead to the development of more sophisticated and adaptable software analysis tools. Ultimately, our research contributes to the advancement of more robust and versatile approaches for ensuring code quality, improving software maintainability, and enhancing overall software development practices across a wide range of programming languages.

3.1.2 Project Selection

In order to avoid well-known mining perils [KGB+16], we applied the following methodology to select the projects for this study: (i) systems that have at least 500 commits; (ii) systems that are at least 3 years old, and are currently active; and (iii) Java, C++, and C# based systems, as previously mentioned in 3.1.1. Moreover, the project selection was based on related work [Rod20, dSR20, UBO+20, BUC+20, BUC+23], and we can see the project list in Tables 3.1, 3.2 3.3. Each table represents the projects selected to be analyzed in our study for the programming languages Java, C++ and C#, respectively. The tables not only list the projects' names but also describe the domain it - if it is a library, a framework, or

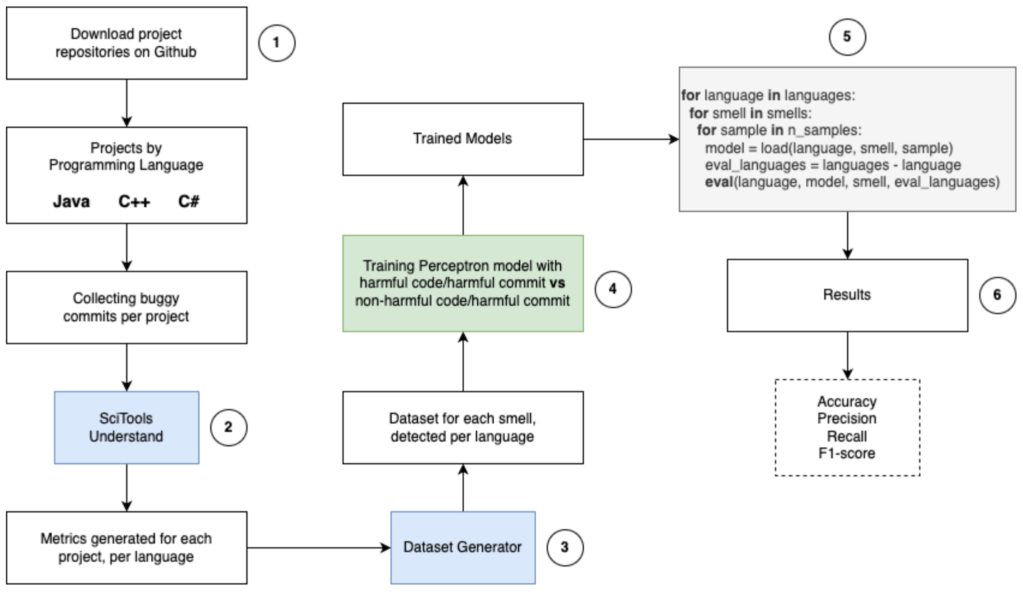

Download project repositories on Github
1
Projects by Programming Language
Java C++ C#
Collecting buggy commits per project
SciTools Understand
2
Metrics generated for each project, per language
Dataset Generator
3
Dataset for each smell, detected per language
Training Perceptron model with harmful code/harmful commit vs non-harmful code/harmful commit
4
Trained Models
5
for language in languages:
for smell in smells:
for sample in n_samples:
model = load(language, smell, sample)
eval_languages = languages - language
eval(language, model, smell, eval_languages)
Results
6
Accuracy
Precision
Recall
F1-score

Figure 3.1: Study design steps for creating the dataset and evaluating transfer learning

an app - as well as the number of: i) commits; ii) smells, and iii) bugs for each one of the projects.

3.1.3 Metrics and Code Smells

We utilized *only* the **detection rules** from the DesigniteJava [1] tool [SS18] to identify code smells in our projects, along with their corresponding thresholds, which were collected using the Understand [2] to collect the metrics in all systems. Moreover, we needed to make a pair relation between the DesigniteJava metrics names and the Understand metrics, as they do not have the same name. Finally, the code smell list and their respective thresholds can be seen in Table 4.2, and the metrics name pair relation can be seen in Table 4.1.

3.1.4 Finding Bugs

We rely on previous work [Rod20] methodology to collect the bugs used in our dataset. This methodology utilizes a GitHub macro present in commit messages that fix bugs. These macros typically include keywords such as "Fixes", "Fixed", "Fix", "Closes", "Closed" or "Close", followed by a # and the issue/pull request number, *e.g.,* "Fixes #12345". This

[1]https://www.designite-tools.com/designitejava/
[2]https://scitools.com/

Table 3.1: Java Software projects analyzed in this work

System	Domain	# Commits	# Smells	# Bugs
exoplayer	Library	15057	134	71
baritone	Library	3036	2025	2863
junit5	Framework	7753	0	2
lombok	Library	3540	1530	1346
mindustry	Game	16469	20	39
mockito	Framework	5929	14	73
okhttp	Http Client	5400	128	62
termux-app	App	1431	28	10
bisq	App	17481	661	781
l2jorg	Library	2228	238	260

macro automatically closes/merges the issue/pull request, and by examining their label list, we can check if that issue/pull request contains the label ’bug’ or ’defect’, showing that the commit was a bug-fixing commit. Finally, to collect the commit that contains the bug, we got the parent commit of the bug-fixing commit.

3.1.5 Discovering Harmful Code

Harmful code is a term introduced by Lima *et al.* [Rod20] to determine a code snippet has two characteristics: (i) *smelly*, when the code contains a code smell; and (ii) *buggy* when the code contains a bug. When containing both characteristics, we say that a code snippet is *harmful*.

Table 3.2: C++ Software projects analyzed in this work

System	Domain	# Commits	# Smells	# Bugs
gdal	Library	51643	3446	312
keepassxc	Software	4476	113	83
osquery	Framework	6313	17	12
tdesktop	Software	14819	181	59
px4-autopilot	Framework	42390	187	26
qtox	Software	8298	115	64

3.1.6 Application of Transfer Learning

We trained our transfer learning models with the generated dataset from previous steps, allowing us to execute the transfer learning, according to the pseudo-code of Fig. 3.1 step 5. The result of this step is a file with all necessary metrics to perform the model evaluation. Our transfer learning model was built using a perceptron [Kan03], which utilizes a set of crucial parameters that heavily influence its performance. We will discuss them as follows:

Embedding layer. It is the embedding layer added to the model. Embedding layers are often used for text data. In our code, it is configured to accept input sequences of integers (representing words or tokens) with a vocabulary size of 20,000 words. Each input word will be embedded into a vector of size 8. The input_length parameter specifies the length of input sequences.

Flatten layer. After the embedding layer, a flattened layer is added. This layer is used to convert the 2D output from the previous layer into a 1D array.

Dense layer. A dense (fully connected) layer is added with a single neuron. The 'sigmoid' activation function is applied to this neuron, which is typical for binary classification problems. It is used to output a probability that a given input belongs to a particular class.

Compilation parameters. We used the 'adam' optimizer, which is a commonly used optimization algorithm, and the loss function is set to 'binary_crossentropy,' indicating that

Table 3.3: C# Software projects analyzed in this work

System	Domain	# Commits	# Smells	# Bugs
efcore	Library	14104	1923	1484
humanizer	Library	2319	12	10
jellyfin	Software	24154	518	292
omnisharp-roslyn	Http Server	6113	0	2
quicklook	Software	838	12	15
neo	Blockchain	1404	159	277
ryujinx	Software	2840	45	38

Table 3.4: Understand Software Metrics

Name	Abbrev.	*SciTools Understand*	Granularity
Lack of Cohesion in Methods	LCOM	PercentLackOfCohesion	Class
Number of Fields	NOF	CountDecClassVariable + CountDeclInstanceVariable	Class
Number of Methods	NOM	CountDeclMethod	Class
Number of Public Methods	NOPM	CountDeclMethodPublic	Class
Weighted Methods per Class	WMPC	SumCyclomaticModified	Class
Number of Children	NC	CountClassDerived	Class
Lines of Code	LOC	CountLine	Class/Method
Cyclomatic Complexity	CC	Cyclomatic	Method

this model is likely used for binary classification tasks. Hence, the whole algorithm can be seen in the snippet below:

.

Table 3.5: Detection Rules for the Code Smells

Name	Granularity	Type	Metric	Logical Op.
Multifaceted Abstraction	Class	Design	LCOM >= 0.8 NOF >= 7 NOM >= 7	AND
Insufficient Modularization	Class	Design	NOPM >= 20 NOM >= 30 WMC >= 100	OR
Wide Hierarchy	Class	Design	NC >= 10	N/A
Long Method	Method	Implementation	LOC >= 100	N/A
Complex Method	Method	Implementation	CC >= 8	N/A

```
for language in languages:
    for smell in smells:
        for sample in n_sample:
            model = load(language, smell, sample)
            model = Sequential()
            model.add(Embedding(20000, 8,
                input_length=padding))
            model.add(Flatten())
            model.add(Dense(1, activation='sigmoid'))
            model.compile(optimizer='adam',
                loss='binary_crossentropy')
            eval_languages = languages - language
            eval(language, model, smell, eval_languages)
```

Evaluation

To evaluate the models, we first need to compute the results of TP, TN, FP, and FN, that are described as follows:

- **TP**: True Positive, when the model correctly predicts the "YES" target class.
- **TN**: True Negative, when the model correctly predicts the "NO" target class.
- **FP**: False Positive, when the model incorrectly predicts the target class as "YES" when it should be "NO".

- **FN**: False Negative, when the model incorrectly predicts the target class as "NO" when it should be "YES".

Then, we are able to calculate the metrics that are the output of our model:

- **Accuracy**: The rate of correct classification by the model (as either a smell or not).

$$A = (TP + TN)/(TP + TN + FP + FN)$$

- **Precision**: Of the samples classified by the model as smells, how many were actually smells.

$$P = TP/(TP + FP)$$

- **Recall**: The proportion of correctly classified smells out of the total number of samples that were actually smells.

$$R = TP/(TP + FN)$$

- **F1-score**: The harmonic mean of precision and recall.

$$F = 2 * [(P * R)/(P + R)]$$

3.2 Main Study Results

In this section, we present the key findings and main results derived from our study.

3.2.1 Effective Transfer Learning to Detect Harmful Code

In this RQ, our primary focus is to assess the effectiveness of transfer learning using the Perceptron [Kan03] model for detecting harmful code. Our approach involves training individual models for each programming language with buggy commits and distinct code smell types using dedicated training datasets that consist of relevant code snippets and corresponding smells in buggy commits. Subsequently, we check the performance of each trained model is meticulously evaluated on testing datasets containing code snippets from all programming languages studied to understand if harmful code from one language can be detected in the others.

Table 3.6: Transfer learning of harmful code trained in C++, C#, and Java and tested in C++, C#, and Java.

		Java	C#	C++
Complex Method	Java	88%	59%	53%
	C#	79%	84%	14%
	C++	74%	54%	97%
Long Method	Java	77%	58%	4%
	C#	70%	66%	-
	C++	37%	45%	92%
Wide Hierarchy	Java	12%	13%	1%
	C#	1%	7%	1%
	C++	0%	1%	100%
Insufficient Modularization	Java	80%	74%	13%
	C#	71%	84%	27%
	C++	34%	63%	99%
Multifaceted Abstraction	Java	83%	41%	7%
	C#	50%	50%	29%
	C++	42%	34%	98%

The study enables us to identify the model's strengths and limitations in different contexts, offering a nuanced understanding of its performance in detecting harmful code beyond the initial training dataset. The resulting insights are presented in Table 3.6, where the first and second columns highlight the smell types involved in the buggy snippet and programming languages used for training, while the horizontal arrangement corresponds to the programming languages in the testing datasets.

Figure 3.2 presents the results of our transfer learning model for harmful code detection. The confusion matrices provide a detailed breakdown of the model's performance across different programming languages and all code smells analyzed combined. Moreover, we also have a confusion matrix for each smell type for each language type, those can be seen

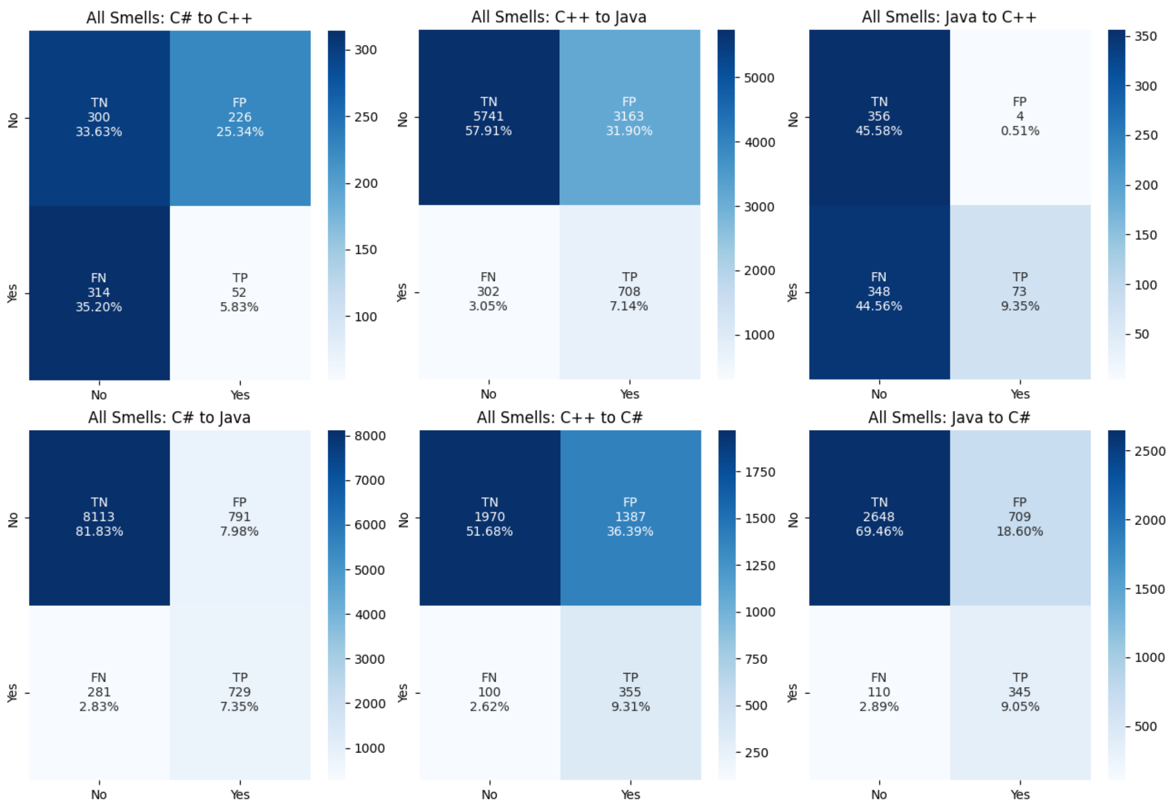


Figure 3.2: Harmful Code results for the transfer learning combined for all code smells and divided by each language.

in our replication package [3].

By analyzing the confusion matrices, we gained valuable insights into the model's strengths and limitations, identifying areas where the model excelled and areas that could benefit from further refinement. These results contribute to a better understanding of the transfer learning model's effectiveness in detecting code smells and lay the groundwork for future research and improvements in code analysis and software quality assurance.

Harmful Code between Java and C#

In Table 3.6, the analysis highlights interesting observations concerning transfer learning between Java and C# programming languages. We notice a noteworthy variation in effectiveness, ranging from 1% in the Wide Hierarchy smell to a maximum of 79% in the Complex Method. This variability indicates that different types of smells exhibit distinct levels of transferability between these languages. Specifically, the Complex Method demonstrates promising results, with a respectable F1 score of 0.88 when evaluated within Java itself and a commendable 0.59 when applied to C#. This suggests that the rule set for this smell translates effectively between the two languages.

Furthermore, examining the Long Method and Insufficient Modularization, we observe a similar pattern, with minor differences in effectiveness, ranging from 3% to 12%. These findings imply that certain code smells have relatively consistent transferability between Java and C#, while others may necessitate more targeted adjustments for optimal cross-language detection. These insights elucidate the intricate relationship between different smells and their transferability across programming languages. Understanding such nuances is crucial for devising more effective and versatile transfer learning approaches in code analysis, ultimately improving software quality and maintainability across diverse language ecosystems.

Finally, the outcomes for the Wide Hierarchy and Multifaceted Abstraction smells were less promising. We obtained an F1-score of 0.01 as the result for Wide Hierarchy detection between the two languages and encountered a difference of 9% in transferring the knowledge of how to detect the Multifaceted Abstraction smell to the other language. These results lead us to the conclusion that not all code smells are equally effective in detecting harmful code between these two languages.

[3]https://opus-research.github.io/sbqs2023_harmful_code_transfer_learning/

Finding 1: The smells of Complex Method, Long Method, and Insufficient Modularization demonstrate a high level of effectiveness in transferring knowledge between C# and Java.

The varying degrees of transferability indicate that some smells may not generalize well across different language contexts, underscoring the importance of carefully considering the choice of code smells and their applicability when employing transfer learning techniques for code analysis in such scenarios.

Table 3.7: F-Measure of Transfer Learning trained with multiple sample sizes

		Samples = 4			**Samples = 8**			**Samples = 16**			**Samples = 32**		
		Java	C#	C++	Java	C#	C++	Java	C#	C++	Java	C#	C++
Complex Method	Java	46%	46%	1%	67%	52%	6%	77%	59%	10%	78%	56%	20%
	C#	36%	46%	-	71%	58%	3%	79%	59%	14%	75%	60%	97%
	C++	48%	41%	31%	74%	54%	59%	16%	16%	35%	64%	46%	82%
Long Method	Java	39%	56%	-	27%	33%	-	45%	57%	-	71%	58%	-
	C#	40%	57%	-	48%	61%	-	63%	60%	-	65%	56%	-
	C++	13%	19%	67%	31%	45%	32%	29%	34%	71%	28%	34%	75%
Wide Hierarchy	Java	0%	1%	1%	0%	3%	0%	5%	2%	0%	12%	-	-
	C#	0%	-	1%	0%	1%	1%	0%	0%	1%	0%	7%	-
	C++	0%	1%	1%	-	-	100%	-	-	100%	-	-	-
Insufficient Modularization	Java	18%	-	-	42%	31%	-	73%	67%	68%	63%	72%	13%
	C#	30%	21%	-	63%	62%	-	71%	67%	-	59%	73%	10%
	C++	34%	29%	16%	30%	53%	85%	22%	46%	93%	23%	47%	92%
Multifaceted Abstraction	Java	32%	2%	-	56%	23%	-	62%	37%	7%	61%	40%	-
	C#	19%	14%	20%	25%	26%	23%	42%	34%	29%	46%	40%	25%
	C++	19%	34%	7%	32%	23%	83%	31%	25%	77%	33%	22%	88%

C++ and a low knowledge transferability

In the context of the C++ language, the observed poor transferability across all smell types suggests that the application of transfer learning techniques for harmful code detection faces significant challenges in this language. The relatively low F1 scores highlight the difficulty in effectively adapting harmful code detection rules between C++ and other programming languages.

For the Complex Method, the model achieved good results when training with C++ and applying to Java (0.74), but not so good when applied to C# (0.54), when compared with the 0.97 when applied to itself. Moreover smell types Long Method, Multifaceted Abstraction, and Insufficient Modularization, showed F1 scores lower than 0.5, which emphasizes the complexity of these smells in C++ code. These findings indicate that the underlying structures and coding practices in C++ present unique nuances that hinder the straightforward transfer of knowledge learned from other languages.

The results underscore the importance of considering language-specific characteristics when applying transfer learning techniques in code analysis. As C++ is a language known for its intricacies and versatility, it may require tailored approaches and specialized models to achieve more accurate and effective harmful code detection.

Finding 2: C++ shows low knowledge transferability across languages, with Complex Method and Insufficient Modularization achieving an F1 score of 0.5 with C# language.

Further research and exploration of domain-specific features and transfer learning strategies can aid in improving the transferability of knowledge across programming languages, ultimately enhancing the overall performance of harmful code transfer learning approaches in the context of C++.

3.2.2 Efficient Transfer Learning to Detect Harmful Code

Java high transferability with small samples

The observed behaviors from Tables 3.7 and 3.8 indicate that, for the majority of smells, reaching a sample size of 32 or 64 is sufficient, as the gains from increasing to 128, 256, or 512 are not significantly substantial.

For the Complex Method, when comparing Java to C#, the behavior reaches its best

Table 3.8: F-Measure of Transfer Learning trained with multiple sample sizes

		Samples = 64			**Samples = 128**			**Samples = 256**			**Samples = 512**		
		Java	C#	C++	Java	C#	C++	Java	C#	C++	Java	C#	C++
Complex Method	Java	75%	54%	33%	79%	56%	53%	85%	56%	43%	88%	59%	16%
	C#	75%	60%	12%	75%	67%	3%	78%	72%	8%	78%	84%	7%
	C++	66%	47%	90%	67%	45%	94%	23%	29%	95%	47%	42%	97%
Long Method	Java	73%	57%	3%	69%	57%	3%	73%	56%	-	77%	57%	-
	C#	70%	66%	-	64%	60%	-	59%	63%	-	-	-	-
	C++	35%	39%	84%	35%	38%	92%	36%	41%	90%	-	-	-
Wide Hierarchy	Java	-	-	-	-	-	-	-	-	-	-	-	-
	C#	-	-	-	-	-	-	-	-	-	-	-	-
	C++	-	-	-	-	-	-	-	-	-	-	-	-
Insufficient Modularization	Java	69%	68%	10%	70%	71%	7%	76%	70%	3%	80%	74%	3%
	C#	53%	74%	22%	60%	76%	27%	65%	81%	26%	62%	84%	24%
	C++	25%	48%	95%	29%	52%	95%	34%	63%	96%	34%	61%	97%
Multifaceted Abstraction	Java	64%	40%	2%	70%	40%	2%	76%	40%	2%	83%	41%	7%
	C#	50%	44%	4%	49%	44%	8%	50%	50%	20%	42%	29%	98%
	C++	30%	23%	88%	36%	24%	92%	41%	29%	95%	2%	-	-

score with just 16 samples. However, when comparing Java to C++, the best performance is achieved with 128 samples. In the case of Long Method, when comparing Java to C#, the optimal score is achieved with 32 samples, while in the comparison between Java and C++, 64 samples yield the best result. Regarding Wide Hierarchy, the comparison of Java to C# shows that 8 samples are sufficient for the best performance. In the context of Insufficient Modularization, when comparing Java to C#, a sub-optimal performance is reached with 32 samples (72% compared to the optimal 74%). In the comparison between Java and C++, the best case is achieved with 16 samples. Lastly, for Multifaceted Abstraction, the best case is observed when comparing Java to C# with 64 samples. However, when training with C++, a sub-optimal case is encountered with only 8 samples.

Overall, these findings provide valuable insights into the optimal sample sizes for each smell type when applying transfer learning to detect harmful codes in the Java language. Properly selecting sample sizes can lead to efficient and effective harmful code detection, ultimately enhancing software quality and maintainability.

Finding 3: For most harmful code trained in the Java language, sample sizes of 32 or 64 are sufficient for effective transfer learning, with minimal gains observed beyond 128 samples.

This finding underscores the efficiency of transfer learning for detecting harmful code trained in the Java language. It demonstrates that developers and researchers can achieve effective results with relatively modest sample sizes, particularly at 32 or 64 samples. Furthermore, the observation of diminishing returns beyond 128 samples suggests that investing in substantially larger datasets may not yield substantial improvements in detection accuracy. This insight offers practical guidance for optimizing resource allocation in code smell detection efforts, emphasizing the potential benefits of focused data collection and model training with moderate-sized datasets. Thus, our recommendation is to not go over 64 samples.

Harmful code between C# and C++

The Tables 3.7 and 3.8 results present compelling evidence of successful knowledge transfer between the C++ and C# languages for most code smells, demonstrating the potential for cross-language applicability in code smell detection.

An intriguing observation from the results is that a sample size of 32 already yields promising outcomes for most of the analyzed smells. This highlights the efficiency of transfer learning even with a relatively modest amount of data, which can be advantageous when dealing with limited resources or large-scale software projects. Nevertheless, to achieve optimal performance, larger sample sizes are necessary, as evidenced by the Multifaceted Abstraction smell, where a sample size of 512 was required to achieve favorable results. This discrepancy in sample size requirements underlines the importance of tailoring the transfer learning approach based on the specific code smell and language combination, ensuring more accurate and efficient detection.

Finding 4: Effective knowledge transfer between most code smells in C++ and C# languages using transfer learning, and optimal performance often necessitates larger sample sizes, particularly observed in the case of Multifaceted Abstraction.

These findings have significant implications for practitioners and researchers in software engineering. Understanding the transferability of knowledge between programming languages can guide the development of more effective code analysis tools that can be applied across diverse language ecosystems. Moreover, the insights on sample size requirements shed light on the trade-offs between computational resources and detection accuracy, aiding in the optimization of transfer learning techniques for code smell detection in real-world software projects. Overall, this study contributes valuable knowledge towards advancing transfer learning applications in software engineering and promoting better code quality and maintainability.

3.3 Limitations and Threats to Validity

3.3.1 Construct and Internal Validity.

The influence of accurate identification, definition, and the choice of code smell detection techniques and thresholds on the results must be considered. To mitigate this influence, we based our detection techniques and thresholds on well-established previous work [PBD+18, ACA+16, KSV+22]. Additionally, potential issues in the study, such as data collection, bug detection, the selection of transfer algorithms, and the chosen programming languages, were addressed by aligning our choices with relevant related research [FAMZM16,SELS19b, AAB+21, DSPPDL21, FAZ17, BUC+20, BUC+23, UBO+20, FBF+20, SZZ05, Rod20]. By building upon established methodologies and basing our decisions on existing literature, we aimed to enhance the robustness and validity of our study findings.

Our methodology encountered limitations in executing the ablation study for our perceptron model. Our initial intention was to perform a comprehensive analysis by selectively modifying and evaluating different model components. However, practical constraints, including having only one labeled variable and tokens as training input, along with resource limitations, posed challenges to implementing this aspect of the study. To address this limitation, we opted for an ablation approach related to the sample size, as detailed in RQ2. Although we maintain that our research findings hold value within this context, it's important to acknowledge the absence of a complete ablation study as a limitation when interpreting the broader implications of our results.

3.3.2 Conclusion and External Validity.

In addition to the challenges related to the ablation study (internal), it is important to acknowledge that our research also lacks a baseline comparison (external), which could have provided a valuable reference point for evaluating the effectiveness of our transfer learning approach. Furthermore, we acknowledge that the absence of similar studies for comparison is due to the relatively recent emergence of the harmful code detection domain and the novelty of applying transfer learning to this field. We addressed this limitation by making the model between the same languages (*e.g.,* C++ vs C++, Java vs Java) as a way for comparing the cross-language models.

It is crucial to ensure that the observed performance differences between languages and code smells selected are not due to chance or random variations. We used known approaches for the buggy and smelly data collected in our study [OSO+19, OSO+18, SOG+17, FBF+20, Rod20].

The specific selection of programming languages, code smells, and datasets containing buggy snippets used in this work may limit the generalizability of the results to other software systems or environments. To mitigate this concern, future research can explore a broader range of programming languages, code smells, and datasets, ensuring a more comprehensive evaluation of transfer learning's applicability across different scenarios. Additionally, conducting comparative studies with varying datasets and real-world software projects can help confirm the broader relevance and utility of our findings beyond the specific choices made in this study.

3.4 Main Study Conclusion

In conclusion, our study delved into the application of transfer learning for harmful code detection across multiple programming languages, with a focus on Java, C#, and C++. The findings revealed promising transferability of knowledge between Java and C# in the presence of various code smell types, while C++ exhibited more challenging transferability. Notably, a sample size of 32 demonstrated favorable outcomes for most smells, underscoring the efficiency of transfer learning even with limited data. However, for certain situations, achieving optimal performance necessitated larger sample sizes, as evident in the case of Multifaceted Abstraction.

In addition to the findings related to transfer learning's effectiveness and sample size requirements, our study contributes to the broader field of software engineering by addressing a significant research gap. While prior research has explored automatic approaches for detecting code smells and understanding harmful code, our study is among the first to assess the application of transfer learning for detecting harmful code using knowledge from other software systems.

These insights provide valuable guidance for practitioners and researchers in software engineering, emphasizing the importance of selecting appropriate sample sizes and understanding the nuances of transfer learning for more accurate and effective harmful code detection across diverse language ecosystems. By harnessing transfer learning's potential and optimizing its application, we can enhance software quality and maintainability, contributing to the advancement of code analysis techniques and fostering better software development practices.

Chapter 4

Do Bugs follow the bad smells?

Another important investigation of our work is whether bugs are directly involved with code smell snippets and if this relation can be detected. To add more depth to our study, in this chapter, we presented the paper "Bugs follow the bad smells? The relationship between bugs and code smells, a transfer learning approach.".

This research is centered on creating a model that was trained using an extensive dataset of code smells in three distinct programming languages. Subsequently, we assessed its performance using data that contained software bugs. Simultaneously, we conducted a parallel investigation where we trained a model using bug-related data and subjected it to testing involving code smells. This exploration had the potential to advance our understanding of software vulnerabilities and improve our strategies for developing more resilient and maintainable software systems. Moreover, we aimed to grasp the transferability of knowledge across various programming languages.

The findings of our study indicated that models trained using code smells were not well-suited for the detection of bugs. Conversely, models trained with bug-related data exhibited satisfactory results in identifying code smells, although there was room for improvement. Considering the potentially challenging and time-consuming nature of training models to identify bugs using code smells, our study offered valuable guidance for researchers in terms of resource allocation and focus. These results shed light on the optimal utilization of time and resources within the research community.

4.1 Research Study Design

Previous studies have suggested using certain quality attributes to predict if code parts might have bugs [GZNM10, BCDL$^+$12]. Nevertheless, this methodology, as highlighted by Palomba *et al.* [PZF$^+$16], overlooks the consideration of code smells. Intriguingly, these code smells can be assessed using the same attributes, leading to a noteworthy overlap between code smells and bugs. This intersection sheds light on their inherent similarity and interrelationship. Furthermore, within the same investigation, the researchers devised a model based on code smells to detect bugs. Their findings underscore that incorporating the severity of code smells can notably enhance the accuracy of bug prediction. Given this insight, our objective is to employ code snippets linked to code smells in training a transfer learning model for bug detection. Simultaneously, we aim to utilize code snippets containing bugs to train a model for the detection of code smells.

In addressing this, we must initially establish the methodology that will guide our process of detecting the presence of code smells and bugs.

4.1.1 Metrics and Code smells

First, for the purpose of detecting code smells, we employ a methodology used by many previous studies [UBO$^+$20, UBC$^+$21, BUC$^+$20]. This methodology involves the utilization of software metrics, which are applied across a range of detection strategies. The culmination of this process leads to determining the presence or absence of a code smell. These software metrics are collected using the Understand [1] framework, which is applicable to the C++, C#, and Java programming languages, the use of this framework is well established in the community [R$^+$21, GSKJ21, BSC$^+$20]. With these metrics in hand, we proceed to leverage the well-known code smell detection strategies outlined by Sharma [SS18] within the DesignateJava tool, which are reported in Table 4.1. This methodology involves the utilization of software metrics, which can be seen in Table 4.2, which are applied across a range of detection strategies. The culmination of this process leads to a conclusive determination regarding the presence (smelly) or absence (non-smelly) of a code smell. In our investigation, we address the following code smells:

[1] https://scitools.com/

Table 4.1: Understand Software Metrics

Name	**Abbrev.**	***SciTools Understand***	**Granularity**
Lack of Cohesion in Methods	LCOM	PercentLackOfCohesion	Class
Number of Fields	NOF	CountDecClassVariable + CountDeclInstanceVariable	Class
Number of Methods	NOM	CountDeclMethod	Class
Number of Public Methods	NOPM	CountDeclMethodPublic	Class
Weighted Methods per Class	WMPC	SumCyclomaticModified	Class
Number of Children	NC	CountClassDerived	Class
Lines of Code	LOC	CountLine	Class/Method
Cyclomatic Complexity	CC	Cyclomatic	Method

- **Multifaceted Abstraction**: Class or module attempts to handle multiple responsibilities, violating the principle of single responsibility. It results in tangled and convoluted code, making maintenance and understanding challenging.
- **Insufficient Modularization**: Code lacks proper organization into separate modules or functions. It can lead to tangled dependencies, reduced reusability, and difficulty in tracking down issues.
- **Wide Hierarchy**: Excessive depth or breadth of class inheritance hierarchies. It can lead to intricate relationships and increased complexity, making the code harder to modify and understand.
- **Long Method**: When a method becomes excessively lengthy, it becomes a breeding ground for confusion and error. This code smell can hinder code readability, maintenance, and lead to difficulties in debugging.
- **Complex Method**: When a method contains intricate logic and a multitude of branching conditions. Such complexity can make the code challenging to understand, test,

and modify.

Table 4.2: Detection Rules for the Code Smells

Name	**Granularity**	**Type**	**Metric**	**Logical Op.**
Multifaceted Abstraction	Class	Design	LCOM >= 0.8 NOF >= 7 NOM >= 7	AND
Insufficient Modularization	Class	Design	NOPM >= 20 NOM >= 30 WMC >= 100	OR
Wide Hierarchy	Class	Design	NC >= 10	N/A
Long Method	Method	Implementation	LOC >= 100	N/A
Complex Method	Method	Implementation	CC >= 8	N/A

4.1.2 Bug detection

In order to detect the presence of a bug, we applied the methodology that has been highly used by recent studies [Rod20, FBF+20, KVGS11, PBDP+18]. This approach utilizes a GitHub macro found within commit messages specifically addressing bug fixes. These macros typically encompass keywords such as "Fixes", "Fixed", "Fix", "Closes", "Closed", or "Close", followed by a # sign and the associated issue or pull request number, as exemplified by "Fixes #2023". The macro is designed to automatically close or merge the relevant issue or pull request. By analyzing the list of labels associated with these issues or pull requests, we can verify if they are marked with the 'bug' or 'defect' label, signifying their classification as bug-fixing instances. Moreover, to collect the code snippets related to the bug, we analyze the diff of the bug-fixing commit with its parent. By doing that, we are able to collect the code removed or modified, which we call a buggy code. This systematic methodology ensures a comprehensive and reliable compilation of bug-related data for our study.

4.1.3 Model evaluation

We relied on the Goal Question Metric template [WRH+12] to describe our study goal as follows: *analyze* a transfer learning model of smelly and buggy code snippets; *for the purpose of* detect buggy and smelly code; *concerning* software metrics and detection rules; *from the viewpoint of* software developers when performing code changes; *in the context of* twenty-three distinct open-source systems, spanning across three diverse programming languages. We introduce our research questions (RQs) as follows.

RQ$_1$: How effective is transfer learning to detect bugs from code smells?

RQ$_2$: How effective is transfer learning to detect code smells from bugs?

Our first research question seeks to explore the effectiveness of utilizing transfer learning as a means to identify software bugs that originate from code smells. Our goal here is to observe how known code smells can be useful to find new bugs in the system. In order to answer this research question, we need to establish an oracle capable of identifying the existence of code smells within the source code of the analyzed software projects. Moreover, we applied the methodology of Section 4.1.1 to generate this oracle, which is composed by open-source projects from the three different programming languages. Finally, by employing performance evaluation metrics such as precision, recall, and F-measure, our study aims to quantify the capacity of transfer learning techniques to detect bugs that emerge due to the existing code smells from our database.

Conversely, our second research question aims to understand the inverse relation, to observe how effective a model is, when trained with bug-related data from three different programming languages, to detect the presence of code smells. To address this research question, we applied the methodology of Section 4.1.2 to generate our database of buggy data.

We will analyze the results for this RQ using the same method as the first research question. We think that the identification of code smells (or bugs) that stem from bugs (or code smells) is important for enhancing code quality and maintaining a robust software ecosystem. A transfer learning approach that effectively tackles this relationship could revolutionize the bug detection process, leading to more comprehensive and nuanced resolution strategies. Furthermore, our findings will serve as a valuable compass for practitioners and researchers, guiding them in the exploration of transfer learning methodologies for the task of identifying quality-related attributes across languages and domains.

4.2 Research Results

Within this section, we outline the results and principal outcomes of our study.

4.2.1 Effective Transfer Learning for Detecting Bugs

In this research question, our primary objective is to evaluate the effectiveness of transfer learning, for that purpose, we utilized the Perceptron model to detect the presence of bugs. Our approach centers on training separate models for each programming language, utilizing code snippets associated with code smells from training datasets. These code snippets are selected from commits that contain code smells. This examination aims to determine the model's capability to identify bugs from one language in the code of another, enhancing bug detection by leveraging insights from code smell analysis. Thus, Table 4.4 shows the Accuracy, Precision, Recall, and F-Measure for each programming language evaluated. From the table, we can infer that all models have shown poor results. However, within those results, we can see that the C++ model has the highest accuracy, precision, recall, and F1 score among the three programming languages. It demonstrates strong performance in correctly classifying and predicting outcomes in comparison to the models trained on Java and C#. The C# model, on the other hand, has the lowest performance in terms of these metrics. This suggests that the C++ model might be better suited for this classification task when compared to the Java and C# models.

Concerning the effectiveness of the model, Table 4.3 illustrates F-measure percentages across the following programming languages: Java, C#, and C++. This metric serves as an indicator of effectiveness in the classification tasks. Notably, the F-measure results exhibit significant variations among the languages. The highest F-measure of 44%, indicated by a blue background, is achieved by C++, reflecting its superior performance. On the other hand, C# showcases the lowest F-measure of 8%, emphasized with a pink background, signifying the weakest outcome. Meanwhile, Java falls in between with an F-measure of 9%. The color-coded highlighting effectively draws attention to the best and worst performances for quick visual assessment. This table sheds light on the effectiveness of the F-measure across distinct programming languages, showcasing C++ as the standout performer and C# as the one with comparatively weaker results. However, the results are not nearly optimal, demonstrating that code smells are effective in detecting the presence of bugs. Finally, the '-' means the model was not able to run for that model because of a correlation higher than 0.75 in the data.

Table 4.3: Results for Model Trained with Code Smells and Tested with Bugs

	Java	C#	C++
Java	9%	26%	-
C#	8%	21%	-
C++	26%	44%	-

Table 4.4: Accuracy, precision, recall, and F-measure for the bug correctly classified by the prediction model trained with smells

Model Language	Accuracy	Precision	Recall	F1
Java	0.195	0.241	0.283	0.260
C#	0.187	0.204	0.216	0.210
C++	0.291	0.364	0.558	0.440

Table 4.5: Accuracy, precision, recall, and F-measure for the smell correctly classified by the prediction model trained with bugs

Model Language	Accuracy	Precision	Recall	F1
Java	0.500	0.500	1.000	0.666
C#	0.500	0.500	1.000	0.666
C++	0.500	0.500	1.000	0.666

Finding 5: Code smells are not suited, in the context of transfer learning models, for the detection of bugs.

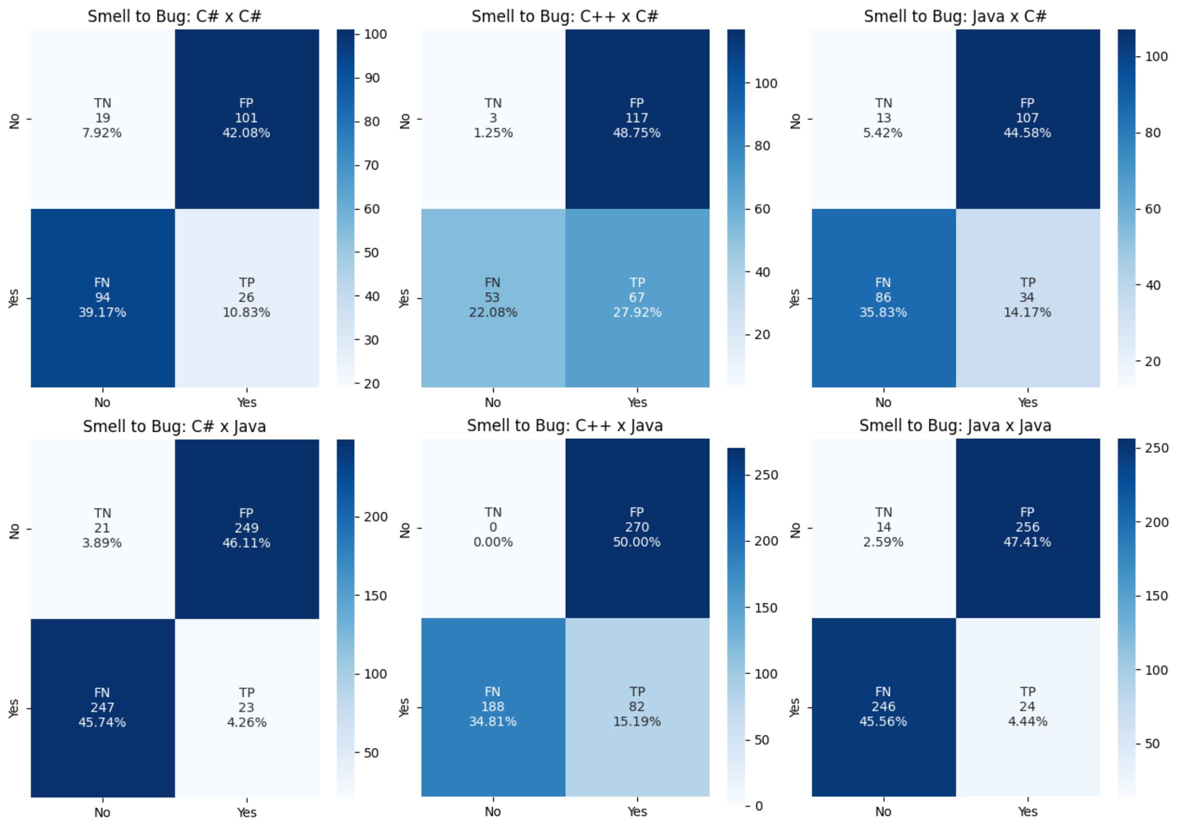


Figure 4.1: Confusion Matrix of the models trained with smells and tested with bugs

Figure 4.1 presents the results of our transfer learning model for bug detection. The confusion matrices provide a detailed breakdown of the model's performance across different programming languages. In general, we can see in the matrices that the number of FP and FN are bigger except for the C++ vs C#. In other words, the model fails to identify actual positive instances (higher FN) while also incorrectly labeling negative instances as positive (higher FP). Finally, Tables 4.4 and 4.5 show the results for the accuracy, precision, recall, and f-measure for the results of our trained model. Consistently uniform values were noticed across all languages. In an effort to decipher this phenomenon, a closer examination of our dataset revealed a crucial insight. The application of a correlation threshold, removing columns with coefficients exceeding 0.75, inadvertently resulted in a reduced dataset size. As a consequence, the models, regardless of the programming language, were operating with a substantially similar set of data.

4.2.2 Effective Transfer Learning for Detecting Code Smells

Table 4.6: Results for Model Trained with Bugs and Tested with Code Smells

	Java	C#	C++
Java	65%	64%	67%
C#	67%	63%	67%
C++	67%	64%	67%

In this research question, the outcomes stemming from training a transfer model with bug-related data to identify code smells offer intriguing insights. Our methodology involves utilizing bug-related code snippets from various programming languages to train a transfer learning model using the Perceptron algorithm. The trained model is subsequently evaluated for its ability to detect code smells within these code snippets. In Table 4.5, we can observe that all three models, trained on different programming languages (Java, C#, and C++), exhibit the exact same performance across all metrics. The accuracy, precision, recall, and F1 score are consistent and identical for all models. This suggests that the models are making predictions with complete accuracy and precision, achieving a recall of 100%, and an F1 score of 0.6.

Regarding the effectiveness, the results, presented in Table 4.6, offer a comparative view of performance metrics, likely indicating success rates, across three programming languages: Java, C#, and C++. Highlighted cells with cyan backgrounds suggest the highest success rates, representing the best outcomes. Notably, the 67% success rate is recurrent, observed in multiple scenarios: where Java and C# intersect, as well as where C# and C++ intersect. Conversely, the magenta cell, indicating a 63% success rate, could signify the weakest performance among the considered cases. The remaining values fall within the mid-range of around 64-65%. In summary, we can observe that the models presented good results when compared to the baseline (*e.g.,* Java vs Java, C++ vs C++).

Finding 6: Code snippets with bugs are good predictors, in the context of transfer learning models, for the detection of code smells.

The outcomes of our transfer learning model for code smell detection are displayed in Figure 4.2. Through confusion matrices, a comprehensive breakdown of the model's performance across various programming languages is depicted. In the matrices, we can see that most cases involved with C++ language had a small sample because the dataset with bugs of C++ only had 18 cases. Moreover, we can see that in this model the results are better, since we can see a higher number of True positive cases, however, the number of false positives is still high.

General Finding: Transfer learning models trained with code smells are poorly suited to detect bugs, while the ones trained with bugs are better suited to detect code smells. However, both models still need improvements.

4.3 Limitations and Threats to Validity

4.3.1 Construct and Internal Validity.

Considering the significant impact of accurately identifying, defining, and selecting code smell detection techniques and thresholds on the research outcomes, it becomes imperative to

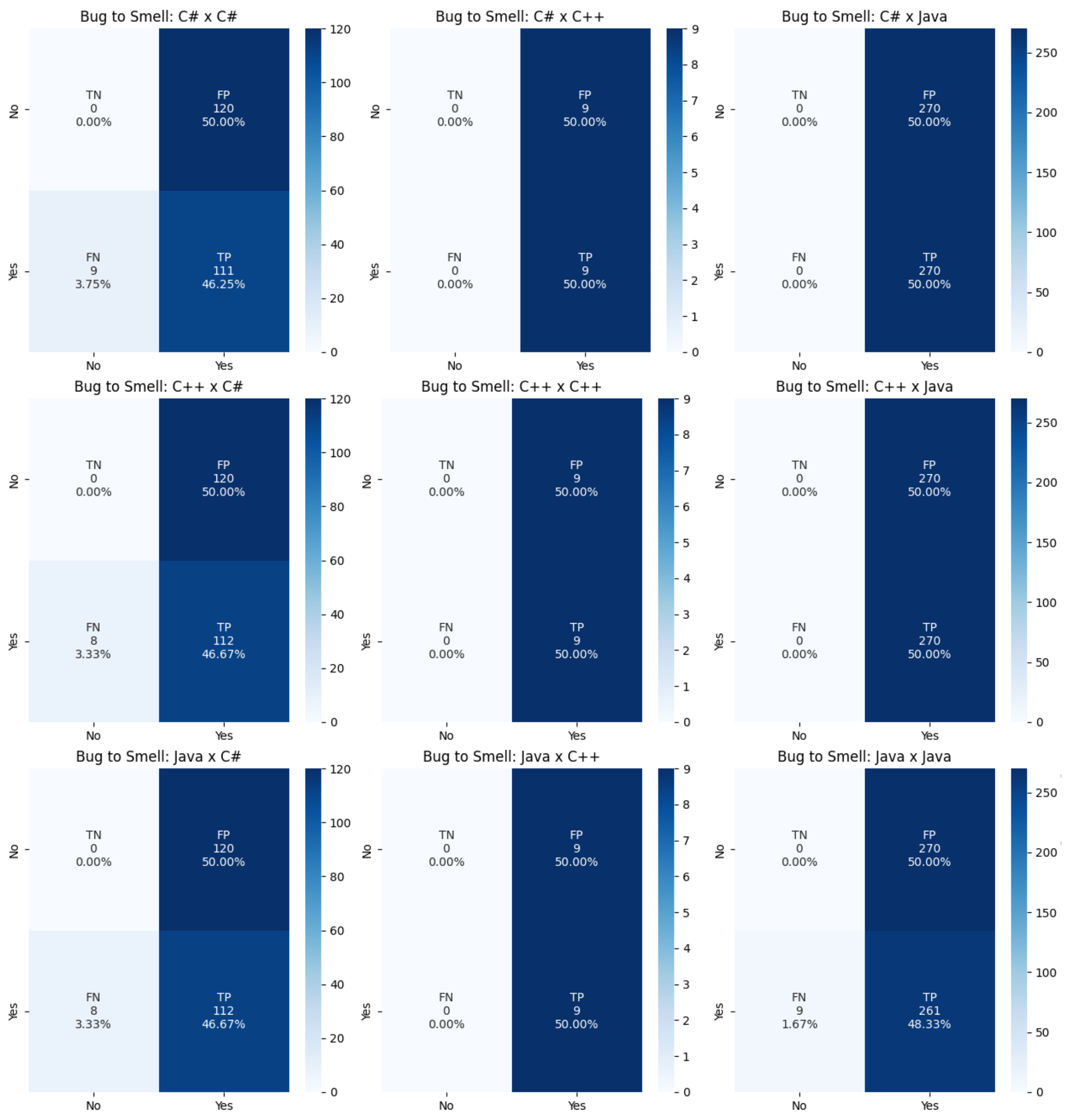


Figure 4.2: Confusion Matrix of the models trained with bugs and tested with code smells

address this issue. Thus, to mitigate this potential influence, we based our detection methods and thresholds on well-established prior studies [PBD+18, BUC+20, UBO+20, UBC+21]. Furthermore, we proactively tackled potential study-related challenges encompassing data collection, the choice of transfer algorithms, and programming language selection by aligning our decisions with pertinent existing research [FAMZM16, SELS19b, AAB+21, DSPPDL21, FAZ17]. Through the utilization of established methodologies and leveraging insights from the broader literature, we aimed to fortify the reliability and validity of our study's outcomes.

4.3.2 Conclusion and External Validity.

Mitigating the potential impact of chance or random fluctuations is essential in assessing the performance disparities across chosen programming languages However, the specific languages, code smell types, and datasets featuring buggy code snippets utilized in this study might restrict the broader applicability of findings to different software contexts. Furthermore, the efficacy of large datasets without a comprehensive understanding of project attributes in constructing accurate oracles is debatable, as noted by existing research [HGA+17, HGFC18, dMOU+22], concerning the potential limitations of solely increasing dataset sizes to enhance effectiveness.

4.4 Research Conclusion

In conclusion, the exploration of transfer learning between bugs and code smells represents a not very effective avenue within the realm of software engineering. The interplay between these two crucial aspects of software quality and maintainability offers valuable insights into the evolution of software systems. First, smells are not very suited to train models to detect bugs, and second, bugs work better to train models to detect code smells.

Moreover, our study has highlighted the ability of code smells to detect bugs in software development. Code smells serve as precursors to potential issues, and addressing them can mitigate the emergence of bugs, enhancing overall software quality. Conversely, analyzing bugs provides insights into the real-world implications of poor code quality. By understanding and leveraging this connection, developers can create more reliable and efficient software systems, ultimately elevating the user experience and long-term maintainability.

Chapter 5

Conclusion & Future Work

In this dissertation, we study how transfer learning models behave when trained with *bugs* and *code smells*, which were used to train a harmful code model. With that said, we present the contributions of our work:

Contribution 1: A transfer learning model trained with a dataset of five *code smells* in three different programming languages. This model can be employed in various types of knowledge transfer studies from code smells to other quality attributes of systems.

Contribution 2: A transfer learning model trained with a dataset of *bugs* in three different programming languages. Similar to the first model, this one can also be utilized in other studies in the realm of software quality or defects.

Contribution 3: A transfer learning model trained with a dataset of harmful code in three different programming languages. This model opens doors for new studies in the area of harmful code, allowing researchers to explore this field without exerting significant effort in training models.

Contribution 4: A set of findings related to harmful code. Within this contribution, we have two smaller contributions:

- Contribution 4.1: How efficient are the harmful code models across three programming languages;
- Contribution 4.2: How effective are the harmful code models, and what is the optimal sample size to train a harmful code model.

Lastly, we have **Contribution 5:** A set of findings related to the knowledge transfer between *bugs* and *code smells*. Here, we uncover that a model trained with *code smells* is not very efficient at detecting *bugs*, neither within the same language nor across different languages. Conversely, models trained with *bugs* yield better results, illuminating the path for future work.

Envisioning potential advancements of these studies, our objective is to qualitatively investigate cases where harmful code was successfully transferred across languages, aiming to validate with developers whether they can make such inferences manually and confirm the instances identified by the model.

Bibliography

[AAB+21] Pasquale Ardimento, Lerina Aversano, Mario Luca Bernardi, Marta Cimitile, and Martina Iammarino. Transfer learning for just-in-time design smells prediction using temporal convolutional networks. *ICSOFT 2021*, pages 310–317, 2021.

[ACA+16] Lucas Amorim, Evandro Costa, Nuno Antunes, Baldoino Fonseca, and Márcio Ribeiro. Experience report: Evaluating the effectiveness of decision trees for detecting code smells. *ISSRE 2015*, 2016.

[BCDL+12] Mario Luca Bernardi, Gerardo Canfora, Giuseppe A Di Lucca, Massimiliano Di Penta, and Damiano Distante. Do developers introduce bugs when they do not communicate? the case of eclipse and mozilla. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 139–148. IEEE, 2012.

[BHA14] Diksha Behl, Sahil Handa, and Anuja Arora. A bug mining tool to identify and analyze security bugs using naive bayes and tf-idf. In *2014 International Conference on Reliability Optimization and Information Technology (ICROIT)*, pages 294–299, 2014.

[BSC+20] Ana Carla Bibiano, Vinicius Soares, Daniel Coutinho, Eduardo Fernandes, João Lucas Correia, Kleber Santos, Anderson Oliveira, Alessandro Garcia, Rohit Gheyi, Baldoino Fonseca, et al. How does incomplete composite refactoring affect internal quality attributes? In *Proceedings of the 28th International Conference on Program Comprehension*, pages 149–159, 2020.

[BUC+20] Caio Barbosa, Anderson Uchôa, Daniel Coutinho, Filipe Falcão, Hyago Brito, Guilherme Amaral, Vinicius Soares, Alessandro Garcia, Baldoino Fonseca, Marcio Ribeiro, et al. Revealing the social aspects of design decay: A retrospective study of pull requests. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*, pages 364–373, 2020.

[BUC+23] Caio Barbosa, Anderson Uchôa, Daniel Coutinho, Wesley KG Assunçao, Anderson Oliveira, Alessandro Garcia, Baldoino Fonseca, Matheus Rabelo, José Eric Coelho, Eryka Carvalho, et al. Beyond the code: Investigating the effects of pull request conversations on design decay. 2023.

[CKK+18] Garvit Rajesh Choudhary, Sandeep Kumar, Kuldeep Kumar, Alok Mishra, and Cagatay Catal. Empirical analysis of change metrics for software fault prediction. *Computers & Electrical Engineering*, 67:15–24, 2018.

[CSCT15] Qimeng Cao, Qing Sun, Qinghua Cao, and Huobin Tan. Software defect prediction via transfer learning based neural network. In *2015 First International Conference on Reliability Systems Engineering (ICRSE)*, pages 1–10, 2015.

[dMOU+22] Rafael de Mello, Roberto Oliveira, Anderson Uchôa, Willian Oizumi, Alessandro Garcia, Baldoino Fonseca, and Fernanda de Mello. Recommendations for developers identifying code smells. *IEEE Software*, 40(2):90–98, 2022.

[DSPPDL21] Manuel De Stefano, Fabiano Pecorelli, Fabio Palomba, and Andrea De Lucia. Comparing within- and cross-project machine learning algorithms for code smell detection. *MaLTESQuE '21*, 2021.

[dSR20] André Moabson da Silva Ramos. Uma aplicação do aprendizado por transferência na detecção de code smells. 2020.

[DZYX20] Xiaoting Du, Zenghui Zhou, Beibei Yin, and Guanping Xiao. Cross-project bug type prediction based on transfer learning. *Software Quality Journal*, 28:39–57, 2020.

[FAMZM16] Francesca Fontana Arcelli, Mika V. Mäntylä, Marco Zanoni, and Alessandro Marino. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering 21*, 2016.

[FAZ17] Francesca Fontana Arcelli and Marco Zanoni. Code smell severity classification using machine learning techniques. *Knowledge-Based Systems*, 2017.

[FBF$^+$20] Filipe Falcão, Caio Barbosa, Baldoino Fonseca, Alessandro Garcia, Márcio Ribeiro, and Rohit Gheyi. On relating technical, social factors, and the introduction of bugs. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 378–388. IEEE, 2020.

[GSKJ21] Aakanshi Gupta, Bharti Suri, Vijay Kumar, and Pragyashree Jain. Extracting rules for vulnerabilities detection with static metrics using machine learning. *International Journal of System Assurance Engineering and Management*, 12:65–76, 2021.

[GZNM10] Philip J Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 495–504, 2010.

[HGA$^+$17] Mario Hozano, Alessandro Garcia, Nuno Antunes, Baldoino Fonseca, and Evandro Costa. Smells are sensitive to developers! on the efficiency of (un) guided customized detection. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 110–120. IEEE, 2017.

[HGFC18] Mário Hozano, Alessandro Garcia, Baldoino Fonseca, and Evandro Costa. Are you smelling it? investigating how similar developers detect code smells. *Information and Software Technology*, 93:130–146, 2018.

[Kan03] Laveen N Kanal. Perceptron. In *Encyclopedia of Computer Science*, pages 1383–1385. 2003.

[Kau20] Amandeep Kaur. A systematic literature review on empirical analysis of the relationship between code smells and software quality attributes. *Archives of Computational Methods in Engineering*, 27:1267–1296, 2020.

[KGB$^+$16] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. An in-depth study of the promises and perils of mining github. *Empirical Software Engineering*, 21:2035–2071, 2016.

[KM19] Rahul Krishna and Tim Menzies. Bellwethers: A baseline method for transfer learning. *IEEE Transactions on Software Engineering*, 45:1081–1105, 2019.

[KSV$^+$22] Aleksandar Kovacevic, Jelena Slivka, Dragan Vidakovic, Katarina-Glorija Grujic, Nikola Luburic, Simona Prokic, and Goran Sladic. Automatic detection of long method and god class code smells through neural source code embeddings. *Expert Systems With Applications 204*, 2022.

[KVGS11] Foutse Khomh, Stephane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. Bdtex: A gqm-based bayesian approach for the detection of antipatterns. *Journal of Systems and Software*, 84(4):559–572, 2011.

[LBT$^+$21] Valentina Lenarduzzi, Terese Besker, Davide Taibi, Antonio Martini, and Francesca Arcelli Fontana. A systematic literature review on technical debt prioritization: Strategies, processes, factors, and tools. *Journal of Systems and Software*, 171:110827, 2021.

[LLC22] Wen Li, Li Li, and Haipeng Cai. On the vulnerability proneness of multilingual code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 847–859, 2022.

[LWQ17] Jiaming Liu, Yali Wang, and Yu Qiao. Sparse deep transfer learning for convolutional neural network. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

[Mar99] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[MHD97] Michael J Muller, Jean Hallewell Haslwanter, and Tom Dayton. Participatory practices in the software lifecycle. In *Handbook of human-computer interaction*, pages 255–297. Elsevier, 1997.

[Mok03] Heng Ngee Mok. *From Java to C# A Java Developer's Guide*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[NDCK19] Edmilson Campos Neto, Daniel Alencar Da Costa, and Uirá Kulesza. Revisiting and improving szz implementations. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12. IEEE, 2019.

[NLWS20] Shuteng Niu, Yongxin Liu, Jian Wang, and Houbing Song. A decade survey of transfer learning (2010–2020). *IEEE Transactions on Artificial Intelligence*, 1(2):151–166, 2020.

[NWG12] Andrew Neitsch, Kenny Wong, and Michael W. Godfrey. Build system issues in multilanguage software. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 140–149, 2012.

[OGC$^+$14] W. Oizumi, A. Garcia, T. Colanzi, M. Ferreira, and A. Staa. When code-anomaly agglomerations represent architectural problems? An exploratory study. In *Proceedings of the 2014 Brazilian Symposium on Software Engineering (SBES); Maceio, Brazil*, pages 91–100, 2014.

[OGC$^+$15] W Oizumi, A Garcia, T Colanzi, A Staa, and M Ferreira. On the relationship of code-anomaly agglomerations and architectural problems. *Journal of Software Engineering Research and Development*, 3(1):1–22, 2015.

[OGS$^+$14] W. Oizumi, A. Garcia, L. Sousa, D. Albuquerque, and D. Cedrim. Towards the synthesis of architecturally-relevant code anomalies. In *Proceedings of the 11th Workshop on Software Modularity; Maceio, Brazil*, pages 39–52, 2014.

[OGS$^+$16] W Oizumi, A Garcia, L Sousa, B Cafeo, and Y Zhao. Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems. In *The 38th International Conference on Software Engineering; USA*, 2016.

[OSO$^+$18] W. Oizumi, L. Sousa, A. Oliveira, A. Garcia, O. Agbachi, R. Oliveira, and C. Lucena. On the identification of design problems in stinky code: experiences and tool support. *J. Braz. Comp. Soc.*, 24(1):13:1–13:30, 2018.

[OSO$^+$19] W. Oizumi, L. Sousa, A. Oliveira, L. Carvalho, A. Garcia, T. Colanzi, and R. Oliveira. On the density and diversity of degradation symptoms in refactored classes: A multi-case study. In *IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 346–357, 2019.

[PA21] Mrutyunjaya Panda and Ahmad Taher Azar. Hybrid multi-objective grey wolf search optimizer and machine learning approach for software bug prediction. In *Handbook of Research on Modeling, Analysis, and Control of Complex Systems*, pages 314–337. IGI Global, 2021.

[PANVA09] Saeed Parsa, Somaye Arabi Nare, and Mojtaba Vahidi-Asl. Early bug detection in deployed software using support vector machine. In Hamid Sarbazi-Azad, Behrooz Parhami, Seyed-Ghassem Miremadi, and Shaahin Hessabi, editors, *Advances in Computer Science and Engineering*, pages 518–525, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[PBD$^+$18] Fabio Palomba, Gabriele Bavota, Massimiliano DiPenta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *EMSE 2018*, 2018.

[PBDP$^+$18] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. In *Proceedings of the 40th International Conference on Software Engineering*, pages 482–482, 2018.

[PZAF^{+}19] Fabio Palomba, Marco Zanoni, Francesca Arcelli Fontana, Andrea De Lucia, and Rocco Oliveto. Toward a smell-aware bug prediction model. *IEEE Transactions on Software Engineering*, 45(2):194–218, 2019.

[PZF^{+}16] Fabio Palomba, Marco Zanoni, Francesca Arcelli Fontana, Andrea De Lucia, and Rocco Oliveto. 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)*, pages 244–255. IEEE, 2016.

[R^{+}21] André Moabson da Silva Ramos et al. Uma aplicação do aprendizado por transferência na detecção de code smells. 2021.

[Rod20] Rodrigo Lima, Jairo Souza, Baldoino Fonseca, Leopoldo Teixeira, Rohit Gheyi, Márcio Ribeiro, Alessandro Gracia, Rafael de Mello. Understanding and detecting harmful code. *SBES '20*, 2020.

[RPTU84] Chittoor V Ramamoorthy, Atul Prakash, Wei-Tek Tsai, and Yutaka Usuda. Software engineering: problems and perspectives. *Computer*, 17(10):191–209, 1984.

[Rup10] Nayan B Ruparelia. Software development lifecycle models. *ACM SIGSOFT Software Engineering Notes*, 35(3):8–13, 2010.

[SELS19a] Tushar Sharma, Vasiliki Efstathiou, Panos Louridas, and Diomidis Spinellis. On the feasibility of transfer-learning code smells using deep learning. *arXiv preprint arXiv:1904.03031*, 2019.

[SELS19b] Tushar Sharma, Vasiliki Efstathiou, Panos Louridas, and Diomidis Spinellis. On the feasibility of transfer-learning code smells using deep learning. *ACM Transactions on Software Engineering and Methodology*, 1, 2019.

[SELS21] Tushar Sharma, Vasiliki Efstathiou, Panos Louridas, and Diomidis Spinellis. Code smell detection by deep direct-learning and transfer-learning. *Journal of Systems and Software*, 176:110936, 2021.

[SG13] Daniela Steidl and Nils Göde. Feature-based detection of bugs in clones. In *2013 7th International Workshop on Software Clones (IWSC)*, pages 76–82, 2013.

[SJP09] Qiang Yang Sinno Jialin Pan. A survey on transfer learning. 2009.

[SOG$^+$17] Leonardo Sousa, Roberto Oliveira, Alessandro Garcia, Jaejoon Lee, Tayana Conte, Willian Oizumi, Rafael de Mello, Adriana Lopes, Natasha Valentim, Edson Oliveira, and Carlos Lucena. How do software developers identify design problems?: A qualitative analysis. In *Proceedings of 31st Brazilian Symposium on Software Engineering*, SBES'17, 2017.

[Son19] Tim Sonnekalb. Machine-learning supported vulnerability detection in source code. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1180–1183, 2019.

[SS18] Tushar Sharma and Diomidis Spinellis. A survey on software smells. *J. Syst. Softw. (JSS)*, 138:158–173, 2018.

[SZZ05] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *ACM SIGSOFT Software Engineering Notes*, 30:1–5, 2005.

[TLNH19] Ha Manh Tran, Son Thanh Le, Sinh Van Nguyen, and Phong Thanh Ho. An analysis of software bug reports using machine learning techniques. *SN Computer Science*, 1(1):4, Jun 2019.

[TS10] Lisa Torrey and Jude Shavlik. Transfer learning. In *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*, pages 242–264. IGI global, 2010.

[TSLHS18] Aoi Takahasi, Natthawute Sae-Lim, Shinpei Hayashi, and Motoshi Saeki. A preliminary study on using code smells to improve bug localization. *ICPC '18*, 2018.

[UBC$^+$21] Anderson Uchôa, Caio Barbosa, Daniel Coutinho, Willian Oizumi, Wesley KG Assunçao, Silvia Regina Vergilio, Juliana Alves Pereira, Anderson Oliveira, and Alessandro Garcia. Predicting design impactful changes in modern code review: A large-scale empirical study. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 471–482. IEEE, 2021.

[UBO$^+$20] Anderson Uchôa, Caio Barbosa, Willian Oizumi, Publio Blenílio, Rafael Lima, Alessandro Garcia, and Carla Bezerra. How does modern code review impact software design degradation? an in-depth empirical study. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 511–522. IEEE, 2020.

[UK18] Gihan M. Ubayawardana and Damith D. Karunaratna. *Bug Prediction Model using Code Smells*. IEEE, 2018.

[WRH$^+$12] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[YLC22] Haoran Yang, Wen Li, and Haipeng Cai. Language-agnostic dynamic analysis of multilingual code: promises, pitfalls, and prospects. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1621–1626, 2022.

[ZLTW20] Ziye Zhu, Yun Li, Hanghang Tong, and Yu Wang. Cooba: Cross-project bug localization via adversarial transfer learning. In *IJCAI*, 2020.