Using Deep Learning to Identify Technical Debt

Levereging Self Admitted Technical Debt

Master's Thesis submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Master of Science in Informatics
Major in Artificial Intelligence

presented by
Simone Giacomelli

under the supervision of
Prof. Dr. Gabriele Bavota
co-supervised by
Dr. Csaba Nagy

January 2021

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Simone Giacomelli
Lugano, 1 January 2021

Dedication

Abstract

Composed of four parts:
a: context
b: problem
c: proposed solution
d: results

max 400 words. I will write it once the intro has been approved

Acknowledgements

Ack

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Technical Debt (TD) is a metaphor between the financial concept and software development. A TD is contracted when a workaround or shortcut is taken during code implementation. Choosing an easy and quick solution over a slower and more correct one gives the benefit to save time and deliver the artifact faster.
The downsides of incurring in TD are many, one of the most intuitive is that further work on the affected parts will be more expensive and time consuming than on clean and healthy code. There are also indirect effects to contemplate: the software could misbehave in the domain it is operating, causing costs and damages as the effect of unexpected output or wrong behavior [71].
TD do not only appear in software projects but can also be found in many layers of a technology stack: for example, delaying an hardware upgrade or a maintenance can give an immediate benefit of less downtime or financial savings, but an increased cost for future unexpected downtime or failures [2].
Developers or managers can choose to incur in TD because of strict deadline, limited resources available or just plain laziness [29, 2]. Cunningham, who coined the technical debt metaphor, writes in "The WyCash portfolio management system" [14]: "A little debt speeds development so long as it is paid back promptly with a rewrite". Cunningham implies that one could benefit from a small amount of TD but its should be paid back as soon as possible.
TD can arise from intentional and unintentional decision, e.g. an inexperienced person could contract it without being aware of it [29]; In both cases it's often done for saving the limited available resources and shortening time-to-market [70]. For example, startups are highly pressured to quickly test products and ideas in order to save capitals and be faster than competitors. Besker at al. studied how startups incur in TD, which are the factors, challenges and benefits of intentional acquisition of TD; two of the regulating factors found by the authors are the experience of the developers and the software knowledge of the founders [9].
In contrast to the beneficial viewpoint of TD, Ron Jeffries argues that the metaphor

could be "perhaps too gentle", because it highlights the wise aspect of the choice of contracting a debt; the problem is that people also takes debt unwisely. Technical debt benefits a software project as long as it is handled before the bigger long term cost is realized [24].
******
It is useful to list some aspects of TD that form the basis for a tractable model of the phenomenon; many of these aspects comes from the already well known attributes of financial debt. The *principal* is the amount that needs to be payed to 'make things right'. The *interest amount* is the added cost in addition to the principal;

In order to better analyze and create a tractable model of TD, Alves et al. identifies three variables [52]:

Identified variables (3) Tracking and managing is important Few proposed tracking and managing examples

there is a cost paying the debt and there are interest costs when efforts are wasted coping with non optimal code.

Why detecting it is important

the satd is there
it's undocumented
the team is not aware of it

Tom et al. in their systematic literature review studied, among others, the benefits and drawbacks of incurring in TD [70]; the part we are now interested in is the drawbacks:
Increasing costs over time, such as the amount of effort required to deliver a certain amount of functionality
Work estimation becomes difficult
Developer productivity is negatively impacted
Becomes increasingly difficult to repay as decisions are affected by existing debt
Increased risk involved in modifications to the system
Change becomes prohibitively expensive to the point of bankruptcy, and a complete rewrite and new platform may become necessary
Decreased quality in the end product

Ci sono team che risolvo as-you-go, altri hanno strategie per rilevare, identificare e fixare. It has been studied that TD [Investigating the Impact of Design Debt on Software Quality] E' importante rilevare e monitorare i TD perche' questi aumentano il costo di un progetto software, costo dovuto dall'interesse che si deve pagare in relazione al TD.

1.1 Objectives and Results

Once the context is clear, I will explain the goal of the thesis and summarize the achieved results The goal of the thesis is to exploit SATD to train a model that can acquire the capacity to distinguish between TD-free code and code affected by TD. Using the comments in open source projects I will identify class methods noted as SATD. Through the vcs commit history I will identify when this comments disappear; the assumption here is that when the comment is removed from the code the SATD has been fixed. Many cases are excluded to minimize the probability of keeping a false positive, i.e. the SATD is not fixed but the comment is removed. For example, the simplest case excluded is when the code is exactly the same and the only change is the SATD comment removal. Another reason of exclusion is when the code is changed too much; in such case it would not be prudent to keep the sample in the dataset.

The achieved results tell us that it is difficult to predict with high accuracy on methods with big bodies. As the train dataset is limited to shorter and shorter method size, the accuracy grows.

1.2 Structure of the Thesis

A simple bullet list saying "Chapter 2 presents the state of the art, bla... Chapter 3 ..."

- Chapter 2 presents the state of the art
- Chapter 3 explains how to leverage SATD to train a Deep Learning model to detect TD
- Chapter 4 Empirical Study design
- Chapter 5 Results discussion
- Chapter 6 Threats to validity
- Chapter 7 Conclusion

Chapter 2

State of the Art

The following sections describe existing approaches to detect different types of technical debt.

2.1 Automatic Identification of Software Bugs

This section deals with tools that are capable of detecting bugs automatically. We divide bugs in three different categories:

- Memory related bugs, e.g. null pointer violation, memory leak and buffer overflow.
- Concurrency related bugs, e.g. critical race conditions, deadlock and unsafe concurrent data access.
- Semantic related bugs, when the fault arise from a contradiction to the intention of the programmer or the original design.

Different tools use different methods to detect bugs; the first big distinction to be made is between dynamic and static analysis.

Static analysis is performed with no execution of the program and relies only on the source code or on some object code.

Dynamic analysis executes the program and inspects its behaviour at run-time. It can be implemented in many flavours, for example: with instrumentation of the executable, with a virtual processor or taking the form of a scheduler. Two are the main factors that are often taken under observation, one is the performance loss in the execution and the second is the extension of the run-time monitor (both in quality and quantity).

We can define another distinction on how bugs are identified; many tools use rules

to detect if some violation has occurred. The rules themselves are of two types: programming and statistical rules.

Programming rules are usually clear and squared, e.g. they derive from axioms, mathematical models or are manually defined.

Statistical rules are defined through a statistical analysis on multiple samples. Usually there is a training phase where observations are collected and correct rules (invariants) are refined.

Another means of detecting bugs is *model checking* where the tool verifies correctness in a usually finite state system. This verification is performed using formal methods on typically formally specified system.

The last category for bugs finding techniques is the *annotation-based* tools. Those requires the annotation of programs to extract semantics and verify consistency and correctness.

What follows is the description for each selected tool.

PREfix [12] A static analyzer for finding dynamic programming errors

PREFix is a source code analyzer that detects a broad range of errors in C and C++ code. Its goal is to detect many runtime issues on real world programs, without dynamic analysis and instrumentation; only the source code text is used. PREfix can detect defects efficiently through a model that abstracts functions and their interactions; the analyzer traces execution paths handling multiple language features: pointers, arrays, structs, bit field operations, control flows statements and so on.

The method used by the tool is based on the simulation of individual functions. It employs a virtual machine that simulates the actions of each operator and function call. With the detailed tracking it can report defects information to the user so to easily characterize the detected error. The tool can be applied both to a complete program source or only a subset. This bottom up approach is particularly useful when the source code is not fully available (e.g. in the case of a thirty part library).

RacerX [16] RacerX: effective, static detection of race conditions and deadlocks

RacerX deals with complex multithread systems. It detects race conditions and deadlock using static analysis. It can infer from the source code the object lock that is assigned to a particular code block. It detects code that is used in a multi-threaded context; it also detects when the code endeavours in dangerous shared access.

RaceX uses annotations only to mark the code that deals with lock acquisition; this requirement keeps the burden on the user to the minimum to increase ease of adoption. The authors of RaceX report their experiences on the biggest problem about race detection: in large codebase there are massive amounts of unprotected variable access; the key point is to report only those that can actually cause problems. There is

emphasis on two aspects:

- The first is to minimize the impact of reporting false positive in order to avoid the users to discard the use of the tool; to achieve this, RaceX employs specific techniques to lower the impact of analysis mistakes.
- The second is the speed of the tool: the authors keep the time of execution for the analysis under deep scrutiny; they claim that for a codebase of 1.8 million LOC the time required is between 2-14 minutes.

The tool has been found capable of finding severe problems in huge projects like Linux, FreeBSD and also in a large closed source commercial software.

Purify [27] Fast Detection of Memory Leaks and Access Errors. Winter 1992 USENIX Conference

Purify is a dynamic analysis tool for software testing and quality assurance . It instruments the object files generated by the compiler (the software dates back to 1992 and the supported platform is Sun Microsystem's SPARC); the process that acts on the object files include also third-party libraries. Purify detects multiple errors: memory leaks, access error, reading uninitialized memory. The injected instructions check every read and write memory operations; the slow down of the target is under three times in respect to the non-instrumented execution time. Enabling the use of Purify is as simple as adding a word in the makefile; the generated overhead is inside the limit of tolerance of developers and it allows to detect bugs early in the development cycle.

Valgrind [56] Valgrind: a framework for heavyweight dynamic binary instrumentation

Valgrind is a framework available for many Linux, Android and Darwin architecture. It is the foundation where many tools are built upon. All these tools, several are already included with the standard distribution, help to build more correct and faster program and are categorized as dynamic analysis tools.
The eight supplied tools can be divided in the following groups: memory error detector, cache profiler, thread error detector, heap profiler. There are also two additional tools that are provided to illustrate how to use the framework works and how to use the core low level infrastructure to implement instrumentation.
Basically, the core implements a synthetic CPU that asks the selected tool how to instrument the code and then continues and coordinates the execution. All the instructions are simulated and the memory access is sandboxed; this includes also the third party library linked into the executable. There are ways to manage and suppress every output generated to avoid clutter and unwanted error reports.
It is advised to enable the debug info into the executable; without them Valgrind is unable to determine which function is the owner of a specific instruction, as such it will produce almost useless error and profiling messages. It is also advised to use minimal compiler optimization to avoid incurring into false positive error reports.

The slowdown of the execution depends on the specific instrumentation of the selected tool, and it is roughly between four to fifty times of the original speed.

CP-Miner [41] CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code

CP-Miner stands for Copy and Pasted code Miner. Zhenmin et al. found that a significant portion of source code in many widely used open source projects are duplications made with copy and paste. This practice introduces bugs, the main reason being that programmers leave identifier untouched instead of renaming them consistently to match the new code context. When the label do not exists in the new place it will be detected with a compilation error; on the other hand, if the identifier exists in the new context it will not be detected by the compiler and it will introduce a hidden bug very hard to detect.
CP-Miner tolerates modification to the pasted code. In order to be detected, the segments do not need to be identical; they can also contain insertions or modifications. The tool is capable of detecting duplicated code but not all detection are true positive; nonetheless it is able to report many significant duplicates with hidden issues. The authors detected 28 copy-paste related bugs in the Linux kernel code base and 23 in FreeBSD; these bugs were reported and most of them where previously unknown to the project team. The analysis is done infra-project and targeted the following large projects: Linux, FreeBSD, PostgreSQL and Apache HTTP server.
The main contributions of the paper are: scalability, bugs detection and statistical study of the copy-pasted code.

- *Scalability* is a strong point of CP-Miner because the technique used in it allow to quickly and efficiently scan large projects including operating system code. For example, it took 20 minutes to find 150,000-190,000 copy-pasted segments respectively in the FreeBSD and Linux kernel; such fragments account for roughly one fifth of the code base. At the time, both projects had more the three million lines of code.

- *Bugs detection* was found to be very effective because of the positive response from the open source project maintainers; most of the reported bugs were not found by static or dynamic analysis detection tools.

- The authors conducted a *statistical study* of the copy-pasted code to give an overview of the phenomenon; the majority of the copied code is between 5 and 16 statements. Around 50 percent of the code has only two copies but around 7 percent has more than eight duplicates. Roughly 12 percent of the copy-pasted code segments are whole functions. Kernel modules are affected with different concentration, depending on the module under analysis: drivers, arch and crypt modules have high copy and paste segments than other parts of the project.

D. Engler's [17] Bugs as Deviant Behaviour: A General Approach to Inferring Errors in Systems Code

The approach taken by Engler et al is to extract from source code beliefs and properties that must or could hold. Static analysis automatically detects two types of beliefs: MUST belief and MAY belief. The first one is something that must certainly hold, for example the dereference of a pointer holds the credence that the reference is not null and must be valid. The second one is statistical by nature; the code is observed and searched for patterns that suggests beliefs, for example a call to function "x" followed by a call to function "y". The probabilistic nature of the MAY belief comes when validating it: at first it is treated as a MUST belief then a search yields all uses in the source code of such belief (i.e. the use of "x" and "y"). If it turns out that such pattern is respected most of the times then the belief is probably valid, otherwise it is treated as a coincidence and, as such, discarded. MUST beliefs bear no doubt about their validity and represent internal consistency. All contradictions from both MUST beliefs and valid MAY beliefs are reported as errors (i.e. bugs). The authors leverage their prior work [18] where they used static analysis to fix manual defined rules for specific system; for example a call to spin_lock(l) must be paired with a following spin_unlock(l). Such patterns were previously specified by hand; with the current work they enable a system to infer the same (and more) rules automatically. They reported that the automatic system is able to detect all the manual rules plus a considerable additional amount that goes from ten to one hundred patterns more. The general idea underling this paper, is that the source code contains intrinsic information about what is correct; finding errors in real system means exploiting what is intended as "correct". Sadly, most of the times these rules are not documented, not formalized or if they are available, they are present in informal and unusable format. With this work the authors use static analysis to extract the beliefs that the programmers infused in the source code, without the need of a priori knowledge. Manually performing the task of extracting correct behaviours and rules from source code is usually a hard, difficult and daunting experience, particularly in view of multiple releases and big code bases. Engler et al show of being able to apply this techniques to complex systems as Linux and OpenBSD operating systems. The results are hundreds of detected contradictions (i.e. errors or bugs) reported; many of them have been assessed and resulted in kernel patches.

DIDUCE [26] Tracking Down Software Bugs Using Automatic Anomaly Detection

The paper introduces a tool written by the authors called DIDUCE: Dynamic Invariant Detection ∪ Checking Engine. It is based on instrumentation of Java programs so to observe their behaviour at run-time. During the lifetime of the target Java process, DIDUCE gather information and collects hypothesis of invariants; those are the rules that the program should obey. As violation to the invariants are encountered the tool relaxes the hypothesis allowing for different behaviour. Every invariant has a confidence level, the process previously described updates it; then the user can go through

all the anomalies reported ordered by their rank. The tool is intended to be used in the discovery of the root cause of bugs and as an aid in better understanding the program under analysis. The query of this ranking can be done, for example, just before a crash occurs: inspecting what DIDUCE detects as anomalies often lead to the discovery of the root cause of the problem.

The paper describes also the findings during the application of DIDUCE to four Java real-life programs, one of which is the JSSE Library (Java Secure Sockets Extension); the bug under analysis was found during the development of a proxy server to be applied to the JSSE Library. The problem was that using the proxy server triggered unexpected behaviour in the JSSE internals. Thanks to DIDUCE the programmer was able to find the issue in the core of the library: it was reported with high confidence that method read() returned a different value than usual. This method returns the number of bytes read from the stream and Java specs clearly define that the method can also return with a buffer not fully filled. A common Java programmers pitfall is to ignore this result and skip the loop on read() to obtain all expected data. DIDUCE reported a violated invariant with high confidence because the result from the method was always 74 and in one instance (just before the Exception) the number was smaller; this information made apparent that using the proxy triggered a bug present in JSSE Library. The authors' experience suggests that discovering bugs using their proposed methodology is simple across many different kind of programs, shown in the paper with four compelling cases.

AccMon [81] AccMon: Automatically Detecting Memory-related Bugs via Program Counter-based Invariants

The contributions of this paper are two innovative ideas: the first is a novel statistical approach to detect bugs in memory related issues, the second is a novel architectural extension to decrease the overhead of the monitoring process.

The first is called PC-based invariant detection (PC stands for program counter); it leverages the observation that most programs access memory location mostly from the same instructions. Being probabilistic in nature, it can detect memory access anomalies that deviate from the baseline; they usually are the causes for bugs, stack overflows and many other memory-related issues. We can see that there are two phases: one where the statistical data are gathered and the baseline rules (e.g. invariants) are formed; the other phase put to use the collected rules and check for violations that will be reported. These two phases are intended to be used in multiple runs but also in the same single long-running execution.

The second contribution is called Check Look-aside Buffer (CLB); it aims to lower the burden of the dynamic process monitor activities to decrease the overhead needed. The authors report their experiments on performance: in the worst case analyzed the loss of speed of the process is less than 3 times. In other tools, this slowdown can be of one order of magnitude greater.

The effectiveness of the authors' contribution are tested through AccMon, a tool they developed in order to implement the idea of a PC-based invariant detection; such experiments shows that the proposed novel ideas are sometime capable of finding more bugs in respect to other tools such as Purify or CCured. Then, in conjunction with previous work from the same authors (iWatcher) it is demonstrated the effectiveness of CLB in lowering the burden of the overhead.

The authors report many other advantages in using AccMon in respect to other tools:

- the analysis is not done on the values of the variables, thus It can detect also bugs that do not violate value-based invariants.
- in the current form, it uses source code to achieve compiler-based optimization but it can directly work with binaries without the need of compilation.
- it does not need type information; given it's statistical nature it can detect anomalies just using abstract memory pointers.
- it's possible to switch off the monitoring activities dynamically at runtime, with almost no overhead. The authors states that AccMon can be used in production runs.

Liblit's [42] Bug isolation via remote program sampling

The underling observation that pave the road for this paper is that often the user community of a program has more raw throughput of running and executing it than the developers. In other words, the number of executions that the team responsible for the program can apply for testing is dwarfed by the number of executions that the community can or will bring up to bear. The authors propose an infrastructure for gathering data from the user's execution to a central information store; then they propose a process, called automatic bug isolation, to analyze gathered data in order to provide information to the developers to help find and fix bugs. This infrastructure shows multiple benefits:

- Use a vast amount of data that is generated by the execution of the program by the user community; it is usually discarded and do not contribute to better the quality and the experience for the user itself.
- Enabling the collection of information helps to draw a clearer picture on the effective use of the program and drive better decisions about development roadmap
- Map and define feature usage statistics
- Avoid issues related to manual feedback generated from an user intervention: usually the user is unsophisticated and non technical; it's ability to have a positive impact on the bug reporting is limited. The benefits of automatically gather this information are many and varied.

Designing an infrastructure that was able to scale was a non-trivial process; there are two main issues to address.

One is to make the lowest impact on the performance on the program execution. It is very important to be respectful of the resources used in gathering debug information. To achieve this goal, the authors employ sampling; they also address a technique to conduct fair sampling.

The other one is a craftiness in gathering and periodically sending data to the central system: even collecting a small amount of information has an huge impact on scalability.

The authors then focus on the data analysis phase. They propose three different applications with increasing level of sophistication:

- They show how to share the burden of assertion across the use base so to inflict upon each user only a small fraction of the checks.
- They show how to start from a large set of predicates (predicate guessing) and shrinking it down over time to reveal the smallest set that can deterministically predict a bug.
- They show how to use linear regression to isolate non-determinstc bugs; in other words they shrink the set of predicates that has the highest correlation with the failure.

ESC/Java [19] Extended Static Checking for Java

This tool perform static analysis on Java source code looking for errors and warnings to report. It provides specific Java annotation to formally express design decisions. ESC handles and warns about multiple common programming errors, e.g. null dereference, index out of bounds, types errors; it also warns about concurrency errors, like race conditions and deadlock. Aided by the custom annotations, ESC employs an engine to decode the semantics of the program and apply techniques to automatically prove theorems; doing so it is able to report potential bugs that are not detected by the type system and that are detected at runtime.

One core requirement imposed on ESC is the modularity of checking; in other words, it can work on pieces of code (i.e. methods) in isolation to the rest of the code. This restriction was chosen for scalability reasons even if the downside is the need of custom annotations. The authors argue that the cost of using the annotations are not an hard overhead: when developers are engaged in manual code review they need information that usually come from unstructured sources (e.g. natural language comments) and they already sustains the burden of gathering additional knowledge not present in the raw code.

It is evident throughout all the paper the importance the authors put on the tradeoff between the cost of the annnotation process and the benefit of true errors feedback; this sentence taken from the paper's introduction sums the core of this tradeoff: "if the

checker finds enough errors to repay the cost of running it and studying its output, then the checker will be cost-effective, and a success". Infact, they enumerates two important features needed by an ideal static checker :

- soundness - if the program has errors, it will find some
- completeness - every error reported is a true positive

ESC does not seek to honor these two features for the very belief quoted above. The authors observe that the alternative processes used to achieve software quality (testing and code reviews) do not possess any of the features of the ideal static checker.

At paper time of writing, ESC was used for two years on multiple kind of programs and was proven effective in finding meaningful bugs. The performance was adequate for interactive use on most methods. Even if it was proven of real usefulness, the users' feedback suggest that the cost of annotating was high and the number of warnings was excessive. It must be said that the annotation were added after the development of the project and not during the evolution of it; this is commonly know by developers to be a dreaded task.
At this point it is unclear if the tool delivers a positive tradeoff between cost and benefit; the experience of the authors during internal use of the tool was encouraging. They believe that ESC is already a valid tool to be used in classrooms: it enforces good design, modularity and verification.

VeriSoft [23] Model checking without a model: An analysis of the heart-beat monitor of a telephone switch using VeriSoft

The goal of VeriSoft is to detect problems in a concurrent reactive system (CRS) through the exploration of its state space. The *reactive* word stands to describe the continuous interaction of the system with the environment. The issues detected are: deadlocks, assert violations, livelocks. A CRS is composed of two parts: a finite number of processes written in arbitrary code (e.g. C, C++, Java, tcl, and so on) and a finite number of communication objects (TCP connections, semaphores, shared memory, and so on). The need of such tool arise from the difficulty of writing a robust and reliable CRS; it is well accepted that concurrent systems are prone to unexpected issues, difficult to track, test and to reproduce.

What VeriSoft does is a systematic state space exploration; it defines the state space as a directed graph: the nodes are the global states and the edges are the transitions between states. It follows that each global state should be uniquely identified and this is one of the core issue that VeriSoft solves with an original combination of algorithms; the author calls it "an efficient state-less search".

The paper reports the analysis conducted with VeriSoft on a software owned by Lucent Technologies: "Heart-Beat Monitor" (HBM). Such software monitors the status of a telephone switch elements and determines their state based on the propagation delays of messages sent through those elements. HBM is an important piece of software

because it has a big impact on the switch performance due to its influence on the switch routing.

The experience of the authors is reported as successful: they were able to find errors in the documentation and in the software itself. Subsequently they modified the code to strengthen some properties and tested them again with VeriSoft; after another run, as desired, VeriSoft reported satisfactory results. The development team of HBM decided to integrate the code changes from the authors for the next commercial release.

VeriSoft acts as a scheduler and has complete control over non-determinism so it can reproduce any interesting scenario (i.e. those that during the automatic tests led to errors and issues). One other benefit is that there is no need to describe the model with specific languages: it relies on the exercise of the actual code. One downside is that it cannot detect cycles in the graph of the global spaces and, as such, it can only detect violation of safety properties.

JPFinder [28] Applying Model Checking in Java Verification

Java PathFinder (Jpf) is a prototype translator from the Java language to Promela (Process Meta Language). Promela runs on SPIN (Simple Promela Interpreter). SPIN is a general tool to find concurrency problems and verify the correctness of a system.

It is reported that Jpf is not the first attempt of Java-to-Promela translator; in addition to this other work, Jpf can handle a significant number of features of the Java language. At the same time, many other are missing. The paper describes the issues due the impedance between Java and Promela; they come in two flavors: performance issues and missing feature issues.

Jps provides the programmer with Java static methods to annotate the source code with assertions; those assertions will be checked with the SPIN model checker.

The authors used a Chess game server written in Java as the test subject for finding synchronization bugs. They did not use the original source code but wrote a simplified abstraction in Java composed with 16 classes and roughly 1400 lines of code; it is reported that this Java program was non trivial and the development was done without thinking about formal verification. Then the authors fed the Java semplification to Jpf and were able to find a bug that was later confirmed.

CMC [55] CMC: A pragmatic approach to model checking real code

Model checking is very hard in practice; it usually involves the use of a specific domain language to describe the model and then a model checker. This common approach to model checking is very hard to endure in practice: it exposes the age-old dualism of having two parallel systems. On one hand we have the actual implementation, on the other hand we have the abstraction that represents the model of the implementation. Having two distinct bodies opens the following issues:

- The model could exhibit issues that are not present in the actual implementation.

- The implementation could show bugs that are not present or detected in the model.

- The need to maintain both systems coping with the impedance of two different ways of expressing meaning, behaviour and intent.

Complex systems often hides rare but nasty bugs that arise only after many weeks of continuous run; this is a major issue with such systems. Explicit checkers can help in this scenario; they search a huge state space without wasting the resources for repeated parts of usual testing.

The first contribution reported by the authors is CMC: C and C++ model checker. It works directly on the implementation without the need to create a separate model to be checked. CMC needs some adaptations on the code, some are just good programming practices (e.g. asserts, specifying the environment), other are changes required specifically by CMC: one is for handling the non-determinism and another one is to handle the initialization functions and event handlers. The tool works by directly executing and scheduling the system under analysis. It needs to store and load the whole state space and as such it handles techniques to cope with the *state explosion problem*: simple heuristics help to prune a huge amount of states.

The second contribution is the application of CMC to three implementations of AODV routing protocol. The actual goal of CMC is to check network code implementations, but the ultimate goal is to check a broader range of programs. During the experimentation the authors, through CMC, were able to find 34 errors many of whom were meaningful errors. Actually, their work exposed also a bug with the AODV specification itself (that was later acknowledged in the RFC 3561 citing the first author of this paper).

Network code is of core importance for the stability of a system; it is prone to many issues that undermine correctness, e.g. packet loss, hardware errors, security attacks.

CMC proved to work well, given the results on three different implementations of a routing protocol. For a wider acquisition of CMC it essential for the authors, to lessen the burden of the code adaptation and automate it as much as possible.

2.2 Code Smells and Anti-patterns

This section explains briefly what code smells and anti-patterns are. The following sections contain the most interesting contribution from the literature on the subject.

An Anti-pattern is a common poor solution to a design problem. It presents itself in

object-oriented based systems where a developer applies a solution that is usually ineffective or worse, harmful. The term stems from its correct and desirable counter part: design pattern. Design pattern is defined as a reusable solution to a commonly occurring problem within a given context in software design.

Anti-patterns are well known to have negative effects on software projects: they hinder code comprehension, and increase maintenance costs.

A code smell is any aspect of the source code that hint to a deeper problem; in other words they are symptoms of a potential bigger issue. It does not always indicate that a real problem exists but it suggests to look closer and inspect if something more profound is present.

It must be said that neither anti-patterns nor code smells are strictly bugs: in fact, they do not imply incorrect results and they do not stop the program from functioning. Nevertheless, for the risks specified above, automatic detection of code smells and anti-patterns received a lot of attention.

In the following sections, we will report several contributions from the literature that start to be available in earnest from roughly 1995.

2.2.1 Four inspirational books description

There are four books that inspired many automatic detection techniques. The first editions of these books span from year 1995 to 1999. The following paragraphs describe them briefly.

Webster [77] Pitfalls of object-oriented development

This book analyzes the cycle of object oriented programming shedding light on its weaknesses and shortcomings. It compares and explains OOP and previous programming techniques. It provides insight and counsel on how to avoid OOP risks.

Riel [63] Object-oriented Design Heuristics

The author provide the reader with metrics to understand the quality of the object-oriented software. The book explains guidelines to help make better decisions on the design of the OO system. The sixty recommendations in this book are language-independent and help the reader to evaluate the quality of a software design.

Fowler [22] Refactoring: improving the design of existing code

The objective of this book is to give practical refactoring strategies to apply on projects so to improve the design of existing systems. It describes many code smells and for each it explains the appropriate actions to take to fix the problem. The first edition dates back to 1999.

Brown et al. [11] AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis

The reader finds the detailed description of 40 anti-patterns divided in three categories: managerial, architectural and developmental. Each anti-pattern entry is a card with many key aspects, e.g. name, description, root causes, symptoms and an optional anecdotal evidence.

2.2.2 Last decade proposed approaches

Many proposal for design flaws identification have been done in the last decade; many of them have some of their roots in previous mentioned books.

Travassos et al. [72] Detecting defects in object-oriented designs: using reading techniques to increase software quality

Travassos et al. created a set of techniques to identify manually defects in order to improve software quality. These practices help individuals to read object oriented code and assess, through a predefined taxonomy. The authors conducted an empirical study on these techniques and reported their feasibility.

jCOSMO [74] Java quality assurance by detecting code smells

This paper presents an approach to automate the detection of code smells. The authors assess that code smells are not precise and formal and need human intuition to appreciate them; the outcome of this observation is that a tool to automatically detect code smells needs to be user configurable. They developed a tool based on these ideas, jCOSMO, that comes already configured to detect two specific code smells (instanceof and typecast). The customization deals with three aspects on code smells: inclusion of new ones, exclusion and fine tuning for more precise definitions. The users of jCOSMO can benefit of automatic detection and graphical visualization of the results.

Simon et al. [66] Metrics based refactoring

The authors agree that the developer is the last authority with the power to decide where to apply refactoring techniques. One of their contributions is providing them with metrics to support subjective perception regarding code smells. They believe that a key issue is helping developers with tools that support human intuition. The authors demonstrate that metrics are effective in finding and pointing to places where code anomalies are detected. These are the presented refactorings: move method, move attribute, extract class and inline class.

Marinescu [51] Detection strategies: Metrics-based rules for detecting design flaws

Marinescu observed that there are multiple problems using quality metrics to improve software quality. Often the definition of the metrics are imprecise, confusing or incomplete. Another issue is the interpretation of the metrics; they seldom provide a

model that help to correctly understand and apply them to a concrete situation. Using the metrics in isolation leads to excessive detail and it becomes difficult to use this information to investigate design flaws. In other words, an isolate measure can be helpful to identify the presence of an anomaly but it does not point to the cause; this leaves the developer without a meaningful insight on how to handle the refactoring. The author calls this 'bottom-up' approach. Marinescu proposes a novel method called *detection strategy* to overcome this problem; it's a 'top-down' approach that starts from an abstract high level goal and drives the investigation of design trait that conforms to the strategy. This technique was applied to ten detection strategies and used on industrial case studies; the result of the experiment proved that the method is applicable and usable in practice.

Lanza and Marinescu [39] Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems

The ideal reader of this book, as defined by the book itself, is a fluent programmer who concretely deals with the maintenance and evolution of a complex large application. This text uses metrics in a practical way to improve software. It uses them in three phases:

- Characterizing the design. The goal is to picture a panorama of the design of the software system. It shows how to use tho metrics-based visualization techniques, *Overview Pyramid* and *Polymetric Views*: Overview Pyramid gives in one shot an overview of the complexity, coupling and inheritance; Polymetric Views shows entities and their relationships supplied with metrics.

- Evaluating the design. This phase serves to better understand and assess the design of the application. *Detection strategies* [51] supply a tool to detect flawed design. *Class Blueprint* is a powerful tool to visualize information about classes, i.e. control-flow and access structure; the goal of this tool is to closely inspect the design flaws detected.

- Design disharmonies. There are three categories of disharmonies: Identity, Collaboration and Classification.

The authors, after the identification of disharmonies, propose also insight on how to improve the design through refactoring.

Munro [54] Product metrics for automatic identification of "bad smell" design problems in java source-code

Munro's focus is on automation of bad smell detection in Java source code. The underling motivation of the paper is to enhanced the process for finding places where to apply refactoring. He begins with a precise definition of code smell, building up on the informal definition from Fowler and Beck. The core idea is to analyze the description

of the code smell and translate it in measurable attributes that are quantifiable. This process is divided in three phases: informal definition analysis, extract possible quantifiable measures and establishment of rules that are used on the metrics to identify the smells. The author applies this process to two specific smells: Lazy Class and Temporary Field. For example, he extracts quality measures as NOM (number of methods), WMC (weighted methods per class), LOC (line of code) and CBO (coupling between objects).

Moha et al. [53] DECOR: A method for the specification and detection of code and design smells
The authors split their contributions in three parts.

The first is DECOR (DEtection and CORrection), it's a method that systematically and formally describes the process to detect code and design smells. It is based on previous work in the field and it leverages the experience and fill the gap of missing features, for example: explicitness on how to specify the detection algorithms, opacity of the technique used, completeness of the analysis on smells description and others.

The process is defined through five well defined steps: description analysis, specification, processing, detection and validation. The correction part of DECOR, as stated by the authors, is for future work and it is not present in the paper.

The second contribution is DETEX (DETection EXpert); the authors revisit their previous detection technique through the lenses of DECOR and name it DETEX. It uses a DSL to specify smells with high level abstraction and it automatically generates the algorithms for the actual search process.

The third contribution is an empirical evaluation of DETEX; consistently with the fifth step of DECOR, Moha et al. provides evidence of the application of DETEX on four anti-patterns relating the results in the form of precision and recall. The target of these experiments are eleven open source projects.

Tsantalis and Chatzigeorgiou [73] Identification of move method refactoring opportunities

This paper has a narrow but highly focused contribution: define a process to identify Feature Envy code smells to enable Move Method refactoring.

The authors observe that high coupling and low cohesion are well known indicators of low quality software design; those features are often linked to unwanted outcomes: low maintainability, low productivity and high bug density.

We can split the proposed process in two parts: identification and ranking.

The identification phase serves to locate candidate methods that could benefit from a move method refactoring. For each method it is evaluated which class could be the destination; this part is driven with distance measures between entities (class, methods and attributes).

The ranking phase serves to avoid overloading the developer with suggestions. This

part uses cohesion and coupling as measures to drive the sorting; the metric employed (called Entity Placement) works evaluating the effect of the refactoring without actually modifying the source code. It is well specified that this is not a fully automated approach because the designer is ultimate responsible on which refactoring should be applied. As one could expect, there are cases where a move method is not a good choice (e.g. moving unit test methods into the target class).

The researchers evaluate their approach with quantitative analysis through the application on two open source-projects. They also track the evolution of the metrics between multiple iteration of refactoring. They then ask a third party to assess the conceptual validity of the proposed refactoring. Lastly they also report on the computational cost of their approach.

Ligu et al. [43] Identification of refused bequest code smells

The authors propose a method to identify the Refused Bequest anti-pattern. In synthesis such design smell happens when polymorphism is badly used, for example when a subclass overrides all the superclass methods. The detection of the smell is achieved with both static and dynamic analyses.

- Static analysis is used to identify those class hierarchies that are candidates to potential anomalies.

- Dynamic analysis is employed through the exercise of unit testing. The methods of the descendant classes are injected with instructions that intentionally raise an exception; the execution of tests will verify which methods are called or not called, and the result will determine a score towards or away good design. The merge of tests result, from the dynamic analysis, with other structural data generates an output that represents the smell strength.

The authors developed an Eclipse plug-in to incorporate the process described above.

2.2.3 Code smell detection formulated as an optimization problem

Kessentini et al. [31] Deviance from perfection is a better criterion than closeness to evil when identifying risky code

This paper proposes a novel method in detecting bad smells. The idea that fueled this work is inspired by artificial immune systems; such systems behave in the following way: the more something is detected as different the more it is considered extraneous. Based on this assumptions, the authors generate a set of detectors that are able to measure various manifestations of anomalies (smells). Thanks to this measurements, they can evaluate how far the system under inspection is from normalcy.

To be able to create the detectors they need to elect some model to be representative of what normality is; this elected model was taken from the project JHotDraw (by Erich Gamma) that represents, ideally, good design and good programming practices.

The authors report that the results outperform the state of the art and their developed tool is able to detect a good mix of bad smells.

Kessentini et al. [32] A cooperative parallel search-based software engineering approach for code-smells detection

The authors' proposed approach use P-EA (Parallel Evolutionary Algorithms) where multiple algorithms cooperate to find a consensus on the common goal of finding the bad smells. Those algorithms use different adaptations: fitness functions, change operator and solution representations. The cooperation between algorithms happens during the parallel execution in multiple iterations and it is not just a result of one final consensus.

To test the effectiveness of the implementation of the approach, the authors make an empiric comparative evaluation with: random search and two other methods not based on meta heuristics. The experiment is based on a benchmark of nine large open-source systems; the reported results shows that the approach is better of the state of the art.

Boussaa et al. [10] Competitive coevolutionary code-smells detection

The authors propose a novel approach to finding bad smells: they use two populations with their CCEA (Competitive Co-evolutionary Algorithm) search and they employ the use of a code-base sample that contains bad smells.

The first population goal is to maximize the detection of bad smells thanks to the generation of rules based on quality metrics.

The second population goal is to maximize the number of synthetic bad smells that the first populations is missing.

The two populations behave similarly to a machine learning GAN framework (Generative Adversarial Network).

The evaluation of the ideas is conducted on four systems through an existing benchmark. The authors reports the statistical analysis: CCEA shows great promise in its performance compared to random and single population approaches.

Sahin et al. [64] Code-smell detection as a bilevel problem

The proposed idea and implementation is based on a bilevel optimization. In such formulation there is an outer problem (upper-level) and an inner problem (lower-level).

The upper-level optimization goal is to maximize the detection of bad smells in a sample dataset through the generation of rules based on quality metrics.

The lower-level maximizes the generation of new artificial bad smell samples that

are not detected by the counter part.

The evaluation of the system was performed with 31 runs on nine open source-projects; seven bad smells were detected with an average of more than 86% in terms of precision and recall.

2.2.4 Non binary classification

The previous section identified code flaws in a binary class classification (smell/clean), while the following paragraphs focuses on those analyses that take care of borderline classes.

Khomh et al. [33] A bayesian approach for the detection of code and design smells

The nature of bad smells contains a measure of uncertainty due to its natural language definition. The authors propose an approach to handle this uncertainty with BBNs (Bayesian Belief Networks).

Through a systematic process of their own making, the authors convert classic detection rules to a BBNs probabilistic model, using the Blob anti-pattern as their test bench.

They use two open-source projects as targets for the evaluation of the model: GanttProject and Xerces. The authors make a comparison between their model and DECOR and show that it returns the same defective classes plus ordering them by importance.

The last contribution is about exploiting bad smells historical information (in the sense of alternative pre-made dataset) in order to train a machine learning model using Weka; the authors show that this calibration increases the quality of the detection.

Oliveto et al. [57] Numerical signatures of antipatterns: An approach based on B-Splines

This paper proposes to overcome two limitations of previous detection technique: the first limitation is the binary classification, there is no in-between or continuous classification of bad smells (e.g. DECOR by Moha et al. [53]). The second limitation is the need of expert knowledge to fuel the detection model (e.g. BBNs by Khom el al. [33]). To overcome these limitations, the authors propose ABS (Antipattern identification using B-Splines). It creates signature of anti-patterns using quality metrics; then it uses the B-spline to create an abstraction of such metrics. This process is applied both to known codes containing bad smells and to unseen unclassified source code; the distance with the B-splines of known anti-patterns measure the similarity to known anti-patterns. In reference to the second limitation, the authors observe that their technique needs only a dataset but no human intervention and tuning.

2.2.5 Usage of historical data for code smells

These two last papers of the section exploit the use of source version control as the basis for their contribution.

Ratiu et al. [61] Using history information to improve design flaws detection

The authors propose an idea to make use of historical data on source code to increment performance on the detection of bad smells. The underling concept is that the evolution of a system can give useful feedback to better determine and analyze the last state of the system. This paper uses the foundation of *design strategies* (Marinescu [51]) adding the concept of a system that evolves through time. The *history* is defined as a sequence of states of the same entity (e.g. system, class and method). The history is used to calculate and evaluate the entity measures *persistence* and *stability*.

The contributions of this paper are: (1) definition of a measure to show how persistent a smell is and how much maintenance effort it absorbed (2) show the improvement in accuracy detecting two class smells (3) describe the valuable information extracted from the history of the anomalies.

Palomba et al. [58] Mining version histories for detecting code smells

This paper shows how to exploit changes on source code to achieve bad smell identification. The history of changes are extracted through the versioning system.

A novel approach is proposed, called HIST (Historical Information for Smell deTection); it detects five classes of code smells: Divergent Change, Shotgun Surgery, Parallel Inheritance, Feature Envy and Blob.

Using the source history of a project is the only way to detect some bad smells; for example, the Parallel Inheritance smell definition cannot be decoupled from tracking the changes in the code. In other words, the very nature of such smell needs to be able to analyze the changes through time of the system.

There are other kinds of smells that do not strictly need historical data but can benefit from using it; for example, Divergent Change smell and Shotgun Surgery have literature that shows detection approach using last-snapshot information only.

The authors compared the accuracy and recall of HIST with alternative approach and their approach tend to perform better. It is reported that HIST is able to detect smells missed by others (the recall is in range 58% and 100%). The previous evaluation was achieved with an empirical study on twenty Java projects; it comprised accuracy and recall calculated against a manually-produced oracle.

A second empirical study represents the closing edge in the loop of the detection process: feedback from the developers of the projects. The goal was to assess at what

extent the programmers agreed on the smell detected by HIST: 75% of the anomalies detected were reported as true problems by the people contacted by the authors (20 developers of 4 projects). This paper makes available a comprehensive replication package.

2.3 Technical Debt and Self-Admitted Technical Debt

This section briefly describes the relevant literature to this thesis on *technical debt* and later on, more specifically, on *self-admitted technical debt*.

2.3.1 Technical debt literature

Guo et al. [25] Tracking technical debt - An exploratory case study

This paper aims to highlight and make evident the effect of technical debt on the cost of a software project. Through the tracking of a single delayed task in a real project, the authors analyze the consequences of such technical debt. They created a framework for the explicit management of TD and then applied it, with a simulation, to the real scenario under scrutiny. The objective of this study is:

- determine technical debt effects on the project and evaluate their impact
- after the application of the simulation, determine if the provided framework gave real gain and uncovered benefits.

The results of this simulation made a clear statement that careful planning and analysis of TD is of high importance: in retrospect, the cost of the delayed task almost tripled the cost for the project.

Klinger et al. [36] An enterprise perspective on technical debt

This study explains the design of an interview whose purpose is to elicit general responses about technical debt; such interviews were conducted with four IBM technical architects. One of the authors' goal was to broaden the view on TD from the perspective of a single developer to the perspective of an enterprise.

Starting from the premise that TD can be leveraged as a financial asset (i.e. incur in TD today to gain competitive advantage and repay tomorrow) the study and the interviews are structured to assess how an enterprise handles TD; these are the standpoints:

- How decisions to acquire TD are conducted.
- The leverage gained contracting TD.

These are some of the findings that were observed:

- Two different sources of unintentional contraction of TD: from non-technical stakeholders (e.g. fixing a stringent release date at the expense of software quality) and from external forces (e.g. changes in the market and acquisitions).
- The process of acquiring TD was informal. The decision had no written records or written analysis on the impact, effects and expectations of such choices.
- A scarcity of knowledge and awareness on the consequences of taking on TD, insufficient channels of communication and lack of a common vocabulary to express contracted costs.

Kruchten et al. [37] Technical debt: From metaphor to theory and practice

This article expands the original metaphor of technical debt by Cunningham [14] in search of a better definition that enables reasoning on a variety of technical debt. The authors want to lay a theoretical foundation so to be better prepared to take on the challenge of dealing with TD. These are the main points covered by this work:

- TD Landscape. It's a possible organization of the many aspects of software improvement. It divides between visible elements (e.g. new features and defects) and mostly invisible (e.g. architecture and code). The idea is that TD is limited to those hidden part.
- Tackling of TD. The authors reason about the root causes of TD concerning quality and maintainability issues (so, not directly related to time pressure, e.g. carelessness, lack of education and poor processes) and describes which steps can effectively handle TD (e.g. awareness, explicit management, understand what tools can and cannot do, nurture architecture, documentations).
- Unified theory. It is observed that the challenge is making the right sequence of changes to improve the software; with respect to this, perhaps the financial and economic models could be the underlying layer to the TD landscape (i.e. expressing all the changes in relation to their cost and value over time).

Lim et al. [44] Technical debt: towards a crisper definition report on the 4th international workshop on managing technical debt

Lim et al. conducted an interview with 35 practitioners aimed to define the perceived characteristics of technical debt and in what context TD was encountered. What emerged is most of the teams know well TD and it is an unavoidable necessity in the business reality. Because of its certainty, one key factor is active management: recognition, tracking, analysis, cogent decision and prevention of worst consequences.

The participants were queried with both specific and open questions. Aside from general demographic questions, they were asked to describe an example of TD alongside its properties, causes, effects and benefits. The answer pointed to a different root cause than sloppy programming and poor discipline. Most of the testimony acknowledged that TD was acquired through intentional decisions; some of which were the

results of short-term thinking, yielding to the pressure of the moment. The negative effects of TD were perceived as long term consequences (e.g. the fear to change code expecting to break other parts of the system). In some cases, it was clear that the benefits were far repaid, in others it was not clear if the balance was positive. The respondents provided many examples of situations that described the crucible for TD (e.g. contracts with a stringent deadline, exploiting market opportunity windows).
The interviewees reported some of their strategies to handle TD:

- Do nothing. In those parts where low maintenance is required, it's safer to leave things as they are.
- Establish a policy to allocate development resource to fix TD (5 to 10 percent on total resources).
- Communication and open dialog about TD between all parties involved (technical, non-technical stakeholders and customers).
- Make TD explicit and visible to all the developers (e.g. through audits) and keep track of the discoveries.

Zazworka et al. [80] A case study on effectively identifying technical debt

This paper conducted a study to compare manual and automatic technical debt detection.

The manual detection was implemented through a questionnaire undertaken by five developers in the same team. The automatic detection was performed using three stable and established tools.

All questionnaire participants reported different debt (except in on case) so there is almost no consensus in the human component, on the other hand, the results show a good overlap between manual and automatic detection regarding defect debt. Human intervention is still needed for the other types of debt: documentation, design, testing and usability debt; they were, for the most part, unrecognized by an automatic tool.

Spinola et al. [67] Investigating technical debt folklore: Shedding some light on technical debt opinion

The goal of this paper is to provide some guidance on new research questions about TD. Exploiting the folklore extracted from grey literature, the authors gather 14 statements on technical debt; then they proceeded to survey 37 practitioners asking their level of agreement/disagreement on those statements. The most agreed upon was the following:"*technical debt is not managed effectively, maintenance costs will increase at a rate that will eventually outrun the value it delivers to customer*".

The underlying observation of this paper is that common belief, traditional stories and customs (i.e. folklore) can help the discovery of interesting topics; then, the agreement (i.e. the interview results) of knowledgeable people on those concepts could give

a measure of value and worthiness and guide possible future research.

Alves et al. [5] Towards an ontology of terms on technical debt

Alves et al. proposed an ontology of terms on technical debt. They developed a *lightweight domain ontology*, designed the quality criteria, conducted a systematic literature mapping and finally submitted the result to a specialist for an evaluation. The followings are the contributions of this work:

- The collection and gathering of information that was previously spread out.
- The organization of a common vocabulary for the technical debt field.

Through the description of a common knowledge ground, the authors want to help researchers and practitioners evolving the Technical Debt Landscape [30]. The first contribution is the organization of 13 types of TD: architecture, build, code, defect, design, documentation, infrastructure, people, process, requirement, service, test automation and test debt. The second contribution consists in the organization of indicators themselves; these indicators were used to support the identification of the TD.

2.3.2 Self-Admitted Technical Debt literature

Storey et al. [68] TODO or to bug: Exploring How Task Annotations Play a Role in the Work Practices of Software Developers

This empirical study has the goal to shed light on how the developers behave on personal and team tasks, with respect to source code annotations (i.e. comments). The authors analyze the relations that annotations have with commonly used tools like, e.g. wikis, issue and bug trackers. They gathered and combined data coming from a mix of methods, divided into two phases:

- Phase 1. Conduction of a survey targeting users of Eclipse IDE. The topic was about annotations: if they wrote them, which types, and how they used them..
- Phase 2. Contextual interviews with developers on three open-source projects. Then, augmentation of the answer from the interview with direct analysis on many versions of the source code, related to the annotation in question.

The conclusion reports how these finding can be useful to improve the tooling and software process.

Potdar and Shihab [59] An exploratory study on self-admitted technical debt

The authors conducted an empirical study on four open-source projects, focusing on three main research questions reported in the following summary:

- Finding the concentration of SATD in the projects
- Discovering the reasons for introducing the SATD
- Calculating the percentage of SATD removal after its introduction

The first contribution is the definition of 62 comment patterns that indicate the presence of a SATD (*e.g. fixme, todo, fix this crap*); this list of patterns was refined through manually reading 101,762 code comments mined from five large open-source projects. Using those patterns, they found that between 2.4% and 31% of the files contained SATD. Another interesting finding is that experienced practitioners are the most likely to introduce SATD. On the other hand, a counter-intuitive discovery is that the amount of SATD correlates with neither complexity nor time pressure. The removal ratio was found to be roughly between 0.26 and 0.63.

Bavota and Russo [7] A large-scale empirical study on self-admitted technical debt

This study is a differentiated replication of the work by Potdar and Shihab. It is based on the mining of a large source code corpus: 159 software projects that accounted for 600K commits and 2 billion of comments.
These are the main findings:

- Diffusion: 51 SATD instances on average per project, that constitutes 0.3% of the comments.
- Types: code debt (30%), defect and requirement debt (20%) and design debt (13%).
- Quality: no correlation is found between the number of SATD and code file internal quality.
- Evolution
  - Growth: during the history of the project, on average, only 57% of the introduced SATD are fixed; thus the number of total instances increases over time.
  - Persistence: those SATD that are fixed (57%) show a remarkable survivability; they stay in the system, on average, for over 1000 commits.
  - Removal: 63% of SATD are removed from the same developer that introduced them in the first place; the rest of the time a different and more experienced developer fixes the SATD.

Maldonado and Shihab [49] Detecting and quantifying different types of self-admitted technical debt

The contribution of this paper is the classification of SATD types in four open-source projects. The first author manually classified 33,093 comments; these are the findings

with the range of presence across projects: design debt (42-84%), requirement debt (5-45%), defect debt (4-9%), test debt (0- 7%) and finally documentation debt (0-5%).

The projects were chosen in the Java realm in different domains with well-commented sources: Apache Ant, Apache JMeter, ArgoUML, Columba and JFreeChart. Using JDeodorant as comment extractor, the authors gathered more than 166K comments. This number decreased to roughly 33K thanks to processing and filtering of those comments with a low likelihood of being SATD. Such operation was conducted through four simple heuristics that targeted the following cases: license comments (removal), commented source code (removal), javadoc (removal), multi-line comments instead of block comments (joining).

The classification process made evident that one SATD can belong to multiple categories (e.g. a design debt can also be a defect debt at the same time). For the sake of clarity this paper associates only one SATD label to the comment.

The set of possible SATD classes was taken from Alves et al. [5]. It is observed that not all 13 original TD classes are found in the selected open-source projects; Maldonado and Shihab argue that some technical debt are not likely to be reported in written comments (e.g. people and infrastructure debt). The authors note that the personal bias and subjectivity can be a threat to internal validity: the manual classification was executed by only one person. Other factors on internal validity: quantity and quality of comments could be affected by biased filtering. About external validity, the authors consider the domain of the projects: it is diverse but all of them are open-source Java projects; thus, the results may not generalize to other languages or market segments.

Wehaibi et al. [78] Examining the impact of self-admitted technical debt on software quality

This empirical study on five open-source projects explores the relation between self-admitted technical debt and defects in source code. It was discovered that that there is an increase in defects after the introduction of SATD. It's also clear that introducing a SATD makes the development on the related code much harder.

2.4 TD and machine learning

This section reviews the literature on machine learning and technical debt.

Khomh et al. [35] BDTEX: A GQM-based Bayesian approach for the detection of antipatterns

An anti-pattern is defined using natural language; thus, to judge if one is present or not, a human is needed for the decision process. This work presents an approach

to automatically detect anti-patterns while handling the uncertainty inherent to the human-centric process.

The authors propose BDTEX (Bayesian Detection Expert), an approach to build BBNs (Bayesian Belief Networks) from the definitions of an anti-pattern capable of ranking what it detects. First, the authors apply BDTEX on the Blob anti-pattern and describe the advantages of this approach with respect to rule-based techniques. Second, the method is validated with three anti-patterns: Blob, Functional Decomposition and Spaghetti code. Then the results are compared with those of DECOR [53]; this tool, differently from BDTEX, employs a rule-based approach and do not provide a ranking. The comparison was made on the satisfaction measured on the behavior of quality analysts; they where asked to judge the detection reported by the tools. In all cases except one, the utility perceived by the analysts was higher for BDTEX than for DECOR.

Fontana et al. 2013 [20] Code smell detection: Towards a machine learning-based approach
Fontana et al. 2016 [21] Comparing and experimenting machine learning techniques for code smell detection

These two papers employ multiple machine learning algorithms for finding code smells. They both use datasets created from the Qualitas Corpus repository [69] and manual labelling for creating the ground truth. The results show an accuracy between 90% and 95%.

Di Nucci et al. [15] conducted a study to investigate possible limitations of the two studies above; they found two potential issues that might compromise the effectiveness of the high performance reported. First, a single dataset contained smelly samples of only one type. Second, the dataset was unbalanced with respect to the proportion between positive class (smelly) and negative (non-smelly) classes.

Amorim et al. [6] Experience report: Evaluating the effectiveness of decision trees for detecting code smells

The goal of the authors' is the evaluation of a decision tree algorithm on finding code smells. For this study, they use labeled data and code metrics on four Java projects. They compare their approach with the manual oracle and with the results from three automated techniques. The comparison shows that the decision tree approach has better performance in most of the cases.

Amorim et al. used the C5.0 algorithm with a 10-fold cross validation and a dataset composed of 7,952 samples. Each sample, which represents an OOP class, has 62 features and 12 binary labels. The features are real numbers depicting source code metrics, for example: line of code, number of parameters and depth of inheritance. The labels represent which code smell affects the class, for example: antisingleton, blob and long parameter list.

The labels related to code smells originated from a work of Khomh et al. [34], on

the other hand the 62 code metrics are a concatenation between the results obtained by two automated tools (CKJM and POM).

The authors compare the performance of their approach with a rule-based technique (using the combined output of four tools) and two machine learning algorithms (SVM and Bayesian Belief Network). The last experiment reported in the paper, involving the addition of genetic programming, gives an improvement in the performance of the basic approach.

Maldonado et al. [65] Using natural language processing to automatically detect self-admitted technical debt

This paper shows an approach to automatically detect design and requirement self-admitted technical debt using natural language processing (NLP).

The experiments use ten Java open-source projects as test bench; the results display a remarkable improvement of performance in respect to the state of the art. The study exposes the words that best correlates to the studied SATD (design and requirement self-admitted technical debt). It is reported that the model behaves very well even using only a fraction (5-23%) of the training dataset.

They manually classified 29,473 comments extracted from six open-source Java projects. This number was much greater before an automatic filtering using five heuristics: the manual cleaning process is similar to another work from the same authors [49] with an additional filter (removal of comments created by IDEs). The joining of the newly created dataset with the one from previous work, yields a total number of 62,566 manually classified comments, spanning ten open-source Java systems. These are the SATD classes found: design, defect, documentation, requirement and test debt.

The classification model employed is the Stanford Classifier which is provided by The Stanford Natural Language Processing Group [50]. It is a Java software that implements a maximum entropy classifier, it is roughly equivalent to a multi-class logistic regression model.

The training was conducted only on the ordinary comments (i.e. the non-SATD), design and requirement SATD, thus on 61,664 samples. The results were compared with two baselines with related performance improvement:

- 7.6 and 19.1 times better for design and requirement SATD, respectively compared to a random baseline;
- 2.3 and 6 times better for design and requirement SATD, respectively compared to comment pattern matching baseline.

The authors discuss the characteristics of the vocabulary used by the programmers that strongly indicates SATD (i.e. ‘hack’, ‘workaround’, ‘yuck!’ for design SATD and ‘todo’, ‘needed’, ‘implementation’ for requirement SATD).

Ren et al. [62] Neural network-based detection of self-admitted technical debt: from performance to explainability

This study proposes a CNN-based approach applied to the identification of SATD. The authors provide the model that includes a method to explain its predictions; the result shows that humans mostly agree with the key phrases extracted by the model to explain itself.

It is observed that classic text-mining approaches have limitations with respect to the following measures: performance, generalizability and adaptability. The method described in the paper addresses these issues.

This work documents the SATD aspects that influence the measures mentioned above: variant term frequency, project uniqueness, variable length, semantic variation and imbalanced data.

The experiment is conducted on ten open-source projects with 62,566 code comments.

It is known that the average of SATD over the total number of comments is very low (e.g. between 0.2-2.6% with average 0.3%, Bavota and Russo [7]; between 3.2-16.8% with average 7.42%, Maldonado and Shihab [49]; at class level between 0.4-3.33%, Potdar and Shihab [59]). Ren et al. report their SATD concentration to be between 0.41-5.57% with average 1.86%; to deal with this unbalanced dataset, they designed and employed a weighted version of the cross-entropy loss and compared the results with the basic loss function: the benefit yields an average improvement of 6.22% on F1-score. The experiments on tuning reveal which hyper-parameters are the most influential: size of word embedding, number of filter and combination of window size. The authors conclude that their method improves on explainability and deployability. The first is proven with the high agreement between the SATD summary by the model and humans. The second is explained with a cross-project experiment: the model can be fine-tuned (i.e. transfer learning) with a small amount of data and the result proves that it is a highly effective approach.

Maipradit et al. [48] Automated Identification of On-hold Self-admitted Technical Debt

An on-hold SATD is a SATD that can be removed only after a condition is met; in other words, there is an external factor that needs to be waited for before proceeding to remove the TD (e.g. a bug in an issue tracker).

The authors propose a system to detect on-hold SATD based on regular expression and machine learning; the empirical experiment shows that 8% of the comments referring to issues are on-hold SATD. The dataset was created from ten open-source projects yielding 133 on-hold SATD on a total of 1,530 comments containing issue references. The classifier developed has an average F1-score of 0.73% and an average AUC of 0.97%.

To evaluate the results, Maipradit et al. collected feedback from the projects' devel-

oper about the detected on-hold SATD: there was agreement that they should be fixed or removed.

Cruz et al. [13] Detecting bad smells with machine learning algorithms: an empirical study

This paper adds empirical evidence on machine learning behavior performing automatic detection of bad smells.

The authors use the Qualitas Corpus [69] and extract 20 projects as a test bench for this work; they focus the research on four bad smells using seven ML techniques. It is observed that Gradient Boosting Machine and Random Forest achieved good performance with two bad smells out of four; God Class with F1-score of 0.86 and Refused Parent Bequest of 0.67.

Another contribution is the application of SHAP (SHapley Additive exPlanations, a approach to explain the output of any machine learning) [45]. The authors provide an insight on the most meaningful features for the prediction.

Cruz et al. created a new dataset to assure quality and perform a comparison on an unbalanced dataset (CSV are available for download). There are four datasets, one for every bad smell, two at class level (God Class, Refused Parent Bequest) and two at method level (Long Method and Feature Envy). Every class has 17 metrics (e.g. Coupling Between Objects and Depth of Inheritance Tree) and every method has 12 metrics (e.g. Lines of Code and Quantity of Loops). These features are calculated with two tools, VizzMaintenance for the class metrics and CKMetrics for the method metrics.

The ground truth of bad smells was created using five tools: PMD, JDeodorant, JSpirit, DECOR and Organic. The decision if the subject was positive or not was taken by consensus between tools. The authors kept only those smells that could be detected at least by three of the tools; agreement by two of the detectors made a positive case. The seven algorithms used are: Naive Bayes, Logistic Regression, Multilayer Perceptron, Decision Tree, K-Nearest Neighbors, Random Forest and Gradient Boosting Machine.

Wang et al. [76] Detecting and Explaining Self-Admitted Technical Debts with Attention-based Neural Networks

This paper proposes HATD, an attention-based deep learning model, capable of detecting and explaining SATDs. The authors describe the important aspects of SATD that need to be analyzed and taken care of to successfully implement their proposal. HATD outperforms the state-of-the-art SATD detectors, provides insightful feedback to achieve explainability and is effective when used in real-world projects.

The proposed model is composed of the following parts:

- ELMo algorithm for word embedding. In opposition to static word embedding techniques ELMo can deal with polysemy.

- Single-head Attention Encoder (SAE). This is used for the explainability of the

model.

- Multi-head Attention Encoder (MAE). This serves to encode better the current word considering the positional information of the other words in the comment.
- A fully connected layer with a weighted cross-entropy loss. This part is feeded with both SAE and MAE output.

The training dataset employed is provided by Maldonado et al. [65]; it was created on ten open-source Java projects and it is known to be imbalanced; to counter this issue, the authors of this paper, introduce a weighted cross entropy loss.

The initial evaluation of the model is done within- and cross-project. The within-project evaluation is performed with a 10-fold cross-validation using all ten project. The cross-project evaluation is implemented using nine projects to train the model and using the remaining project as a test set.

The authors assess the capability of HATD to adapt to real-world projects; they select ten additional popular projects from Github. These are written not only in Java but also in Python and Javascript.

The total number of comments, for these ten projects, are 38,280; HATD detects 543 SATDs, which is 1.33% of the total. The average concentration of the ten Java projects by Maldonato et al. is 6.57% on average. To check the quality of the predictions, the authors randomly select 100 comments (50 SATD and 50 non-SATD); then they engage five programmers and ask them to classify those comments. The result shows an accuracy of 83% against the human oracles.

Rantala et al. [60] Prevalence, Contents and Automatic Detection of KL-SATD

This paper defines KL-SATD to be Keyword-Labeled self-admitted technical debt. It is a subset of SATDs, i.e. those SATDs that contain specific words that indicate the presence of technical debt admission. The authors focus on four specific keywords: TODO, FIXME, HACK and XXX. The goal is to exploit the information assets contained in KL-SATDs and detect those SATDs that are not KL-SATDs.

The dataset used is provided by Lenarduzzi et al. and includes 33 Java projects [40]. After pre-processing, the experiment has access to 507,254 comments. One of the first reported finding is the average and median KL-SATD concentration per repository: 2.29% and 1.52%, respectively.

The paper highlights the difference in the vocabulary used by KL-SATDs and non KL-SATDs. Through a word cloud the authors show which are the most common words used (after keywords removal for the KL-SATDs).

A logistic regression classifier is trained to recognize the two above-mentioned classes. The 10-fold cross validation has an AUC of 0.88.

A new dataset is created, formed only by the non KL-SATD comments. Then, the classifier is executed to make a prediction using this filtered dataset. The authors are interested in those instances labeled as KL-SATD with a confidence over 70%: these

do not contain keywords but could be SATD. To verify this assumption and check if the classifier has learned to spot SATD, two of the authors randomly selected 100 comments and labeled them manually. Using the results, it is estimated that 2/3 of the instances may contain technical debt.

2.5 Summing Up

Here I explain why what I did is different

Chapter 3

Using Deep Learning to Detect Technical Debt

We aim to create a system that automatically detects technical debt in a class method. To achieve this goal, we took the following three steps: dataset creation, classification and hyperparameter tuning. The dataset was automatically created by mining and processing open-source projects histories. It is a balanced dataset by construction and the two classes are referred to as: SATD and fixed.

We initially identify technical debt through the presence of SATD in the comments of the source code, using keyword labels pattern matching [59] [60]. The identification of a SATD/fixed method pair is based on a strong assumption: when the SATD comment disappears, due to a commit, we suppose that the technical debt is fixed; so, we regard the new code as TD-free, belonging to the fixed class. The classifier is a neural network capable of representing snippets of code with a fixed-length vector, conceptually similar to how word2vec works. The learned vector representation (code embedding) is associated with a semantic label (i.e. SATD or fixed). Lastly, to increase the performance of the system we select a set of hyperparameters and perform a distributed grid search for tuning their values. The rest of this chapter explains each part in greater detail.

3.1 Mining SATD Instances and their Fixes

These are the fundamental activities involved in the creation of the dataset:

- Github repository URL mining: using Github API, we extracted 248,872 projects' URL matching our search profile.

- Repository cloning and filtering: some projects can be discarded only after the commit history is available for a precursory analysis to determine if a project qualifies for the next activity.

- Commit history processing: the directed acyclic graph of the commits is traversed and the source code is parsed in search of acceptable SATD/fixed pair.

The following sections describe these activities in detail.

3.1.1 Github repository url mining

We know from experience and the literature that the concentration of SATD is very low [7] [49] [59], for this reason we created a dataset from an initial set of open-source projects as big as possible. The search was conducted for all public Java GitHub repositories in a 20 years time window (from 2000 to 2019). The retrieval process of this URLs list was challenging in itself. We dealt and solved the following issues:

1. GitHub search API limit of 1000 hits.
2. GitHub search API number of request per minute maximum quota.
3. Filtering to remove low-quality repositories
4. Unexpected interruptions

We addressed the first issue with two different intertwined phases of queries: probe and harvest. The probe requested only the number of the repositories that were created in a specific time window; if the number exceeds the 1000 limit, it iteratively divides the probe query into two sub-queries using half of the time window for each and adds them to the job queue. It was not enough to specify the day interval in the query; the time of the day was also needed because in the recent years there are multiple instances of more than 1000 repositories created in a single day.

The second issue was solved using an authentication token, as specified by the GitHub documentation.

The third issue was the reason we switched to the GitHub GraphQL API; we quickly realized that the total number of repositories in the selected period was more than seven million and we needed a way to trim down the list to those most meaningful repositories. We empirically defined, through a trial and error process, a metric to keep those repositories with a good likelihood to contain *higher quality source code*. The GraphQL Repository API can be queried to return additional information; for each repository, we requested the issue count and the commit count so to discard quickly those with less or equal than 100 commits and 100 issues. It must be noted that the GraphQL query automatically excluded some repositories because of atypical structures (e.g. Subversion to git converts and repositories with non-standard naming).

The last issue affects all long-running processes: unexpected interruptions, e.g. network outages, program crashes, API service unavailability. To fix these problems, all failed queries are repeated for a maximum of 50 times, then the program is halted with an exception. The program can recover from a crash because every query is cached to a

file; when a query is issued and the cache is present the network request is skipped and the file content is used instead. This was important because the URLs mining execution took roughly 150 hours and a system to reuse past expended resources were needed.

The URLs mining tool creates two text files: one with the complete list and one with the URLs that match our acceptance criteria. Both files contain the following columns:

- Repository creation date.
- User name and repository name.
- Issue count.
- Commit count.

The total number of URLs retrieved was roughly seven million but we accepted only a subset of them: around 250 thousand repositories.

3.1.2 Repository cloning and filtering

This section describes the process of cloning 250 thousand repositories and applying further filtering.

The repository clone task was conducted without the checkout, i.e. only the commit history was downloaded without creating the working copy for the last commit. This approach saved disk space and processing time, particularly when the last commit contained a high number of files.

Once the repository was locally accessible, using the library JGit[1], we tested the presence of a few files that indicated an Android OS project; if it was the case, the repository was rejected. The problems related to the Android OS was that there were more than 1,700 forks/clones and those repositories counted between 300,000 and 550,000 commits; it is one of the biggest project encountered in this endeavour and it posed a significant bottleneck to the commit history processing: traversing the commit tree could take a couple of days only for one Android OS repository. The exclusion of Android OS in this phase was established only for performance reasons.

The general issue of detecting forked (i.e. duplicated) repositories will be better explained in the next section.

3.1.3 Commit history processing

This section describes the task of traversing all the commits of a repository and collecting the code snippets to our SATD/fixed dataset.

What we are generally looking for is a method body that is affected with SATD and after a commit is not affected by it anymore, i.e. the SATD was removed. What follows

[1] https://www.eclipse.org/jgit/

from it is that we do not need to blindly parse all the Java files but only those that are changed by a commit.

For each repository we iterate on every commit and every file change; but we care only of *modify* type operations; we ignore the rest of git file change operations: add, delete, rename and copy.

At this stage we just have a pair of Java source code texts: the old version before the commit and the new after it; they are also called *before image* and *after image*. Next, we parse the before image and run the SATD detector; if hits are found, then we parse also the after image.

Now we tackle the previous explanation with a different abstraction level in mind: not from the source file point of view but at the method level. What we have is a list of methods affected by SATD coming from the before image and another list of methods created from the after image; we couple the items of these two lists by method name and accept as viable candidates only those pairs whose after image method is not affected with SATD. The following code implements the semantic described before (old stands for before image and new stands for after image):

```
//now old and new contains matching methods instances
oldSatd.forEach { old : Method ->
    val new : Method = methods[old.name]!!
    if (old.hasSatd && new.exists && !new.hasSatd)
        candidateForDb(old, new, newCommitId)
}
```

There are a few considerations to ponder:

- The appearance order of the methods in the source code does not affect the process.
- When a method is removed it is automatically excluded through the lack of pairing between the two lists.
- The renamed methods are lost; they are treated as removed.

The rest of this section develops some aspects that were not fully explained before but that are important to the creation of the dataset.

Keyword label pattern matching. To detect the SATD, we use keyword label techniques using the 62 patterns reported by Potdar and Shihab [59] (they are actually 63[2]). Out of these patterns we create regular expressions that are applied on all the source comments extracted with JavaParser.

[2]http://users.encs.concordia.ca/ eshihab/data/ICSME2014/satd.html

We did an initial experiment to verify the effectiveness of those patterns and we excluded the following two: “there is a problem” and “bail out”. After manually verifying roughly one hundred matches we noticed that those two patterns were often used as documentation in ‘catch’ Java blocks and were not documenting SATD. The final list used in our tool is found in listing A.1.

Snippets preprocessing and cleaning. Every sample in the dataset contains the verbatim method source code pair (before image and after image). It also contains a new pair (called ‘clean’) cleaned by all comments and string literals. The string cleaning process replaces all the non-null and non-empty strings to a constant: “–##string##–”. In other words, the string type values are constrained to this set: null, empty value and “–##string##–”.

Precomputed features The neural model used in this thesis needs a specific representation for each code snippet (see section 3.2.1 for details). The computation of such representation is resource expensive and we decided to store it in the database alongside with the sample (columns old_clean_features and new_clean_features).

Handling forks and duplicated code. In our research we found that many repositories were duplicates or clones (GitHub keeps track of forks but do not track duplicates). We need to be very careful not to introduce noise in the dataset. To ensure a better quality for our samples we implemented the following steps:

- We concatenated the clean images (before and after images) and computed the hash of such a string. This hash is stored in a database field with a unique index on it; this ensures that we do not have duplicated pairs.
- Two additional hashes are calculated: one for the clean before image and a second for the clean after image. Then we discard all pairs that are found having a hash in common between any before and after clean image hash; this ensures that we cannot have the same snippet in both before and after image.
- We did the same process as described in the previous point but applied to the precomputed features.

Rejected snippets. We implemented unit tests[3] to be sure to reject specific Java constructs that would pose issues for the pipeline. For example, methods containing inner named methods (explicit interface implementations) were discarded because the parser in code2vec recognized them as two distinct methods, which is not correct.

3.2 The Deep Learning Model

The model described in this section is called code2vec [3]. The following three sequential processes can be considered a chain that progressively transforms the source code

[3] https://github.com/simonegiacomelli/code2vec-satd-classifier/blob/master/satd-classifier/src/jvmTest/kotlin/satd/step2/JavaMethodTest.kt

into the desired target (i.e. the two classes SATD/fixed):

- Decompose
- Aggregate
- Predict

Each of these steps can be viewed as a process that takes an input, creates an intermediate representation and generates an output for the next process. In the rest of this section we give some details about each one of them and introduce some technical terms.

Decompose. The input of this process is the source code. The output generated is a bag of path-context. The following list gives more detail on the process and the intermediate representations:

- Parsing and creation of the abstract syntax tree (AST) of the method source code.
- Extraction of all AST-paths (up to a fixed limit).
- Encoding of the AST-paths into a bag of context-vector.
- Transform each context-vector into a path-context (so to obtain a bag of path-context).

Aggregate. This process aggregates the bag of path-context (the output from the previous process) using an attention vector. The final result is a code-vector that represents the snippet of code as a continuous distributed vector, i.e. a 'code embedding'.

Predict. The code-vector is fed to a fully connected neural network that performs the classification using the desired classes (i.e. SATD/fixed).

In the following sections we expand and dig deeper into these concepts.

3.2.1 Representing code using AST-paths

This section describes how to capture semantic information from code snippets and create a representation that can be used to predict properties of the snippet, for example a label (e.g. SATD/fixed). To better illustrate how the decomposition is done, we use the simple code snippet in listing 3.1 as an example .

Listing 3.1. Example code to show decomposition

```
String METHOD_NAME() {
```

```
        if(somePreCondition())
          while(!completed())
            doWork();
        return "ok";
    }
```

Using JavaParser[4] the AST is extracted from the source code; see an example in figure 3.1.

We identify three sets of node types in the AST: values, terminal and non-terminal nodes. Values are the leaves (this set is identified with X), terminal nodes are the immediate parent of a leaf and the rest are the non-terminal ones. *AST-path definition*: it is a path connecting two terminal nodes and it must include one non-terminal node which is a common ancestor of the two terminal nodes. The representation of the program is the *set of all its AST-paths*.

To wrap up this section, we explain the last AST-path in figure 3.2 and, for convenience, this AST-path is also reported here:

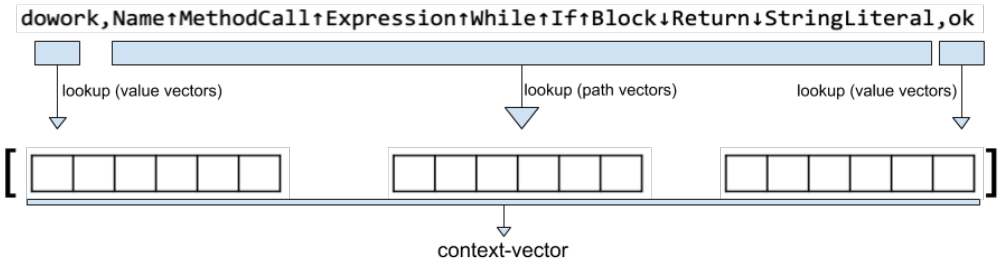
```
dowork,Name↑MethodCall↑Expression↑While↑If↑Block↓Return↓StringLiteral,ok
```

The two value nodes are the first and the last words, 'dowork' and 'ok' respectively; the reader can also identify these leaf nodes in the bottom-right part of figure 3.1. The central part of the AST-path above is the connecting path between those two value nodes: it lists all the intermediate nodes and it also specifies the direction (up or down) one needs to take to traverse the tree. The common ancestor, for this example, is the intermediate node called 'block'.

What is seen in figure 3.2 is a bag of context-paths (another name for the set of all AST-paths) and it is the representation for the code snippet. The following section explains how these AST-paths are transformed into fixed-length vectors.

3.2.2 Context-vector

A context-vector c_i is the vector representation for an AST-path. The process of transformation is applied to all AST-paths producing a bag of context-vectors. The following picture shows how the context-vector is formed:



[4] https://javaparser.org/

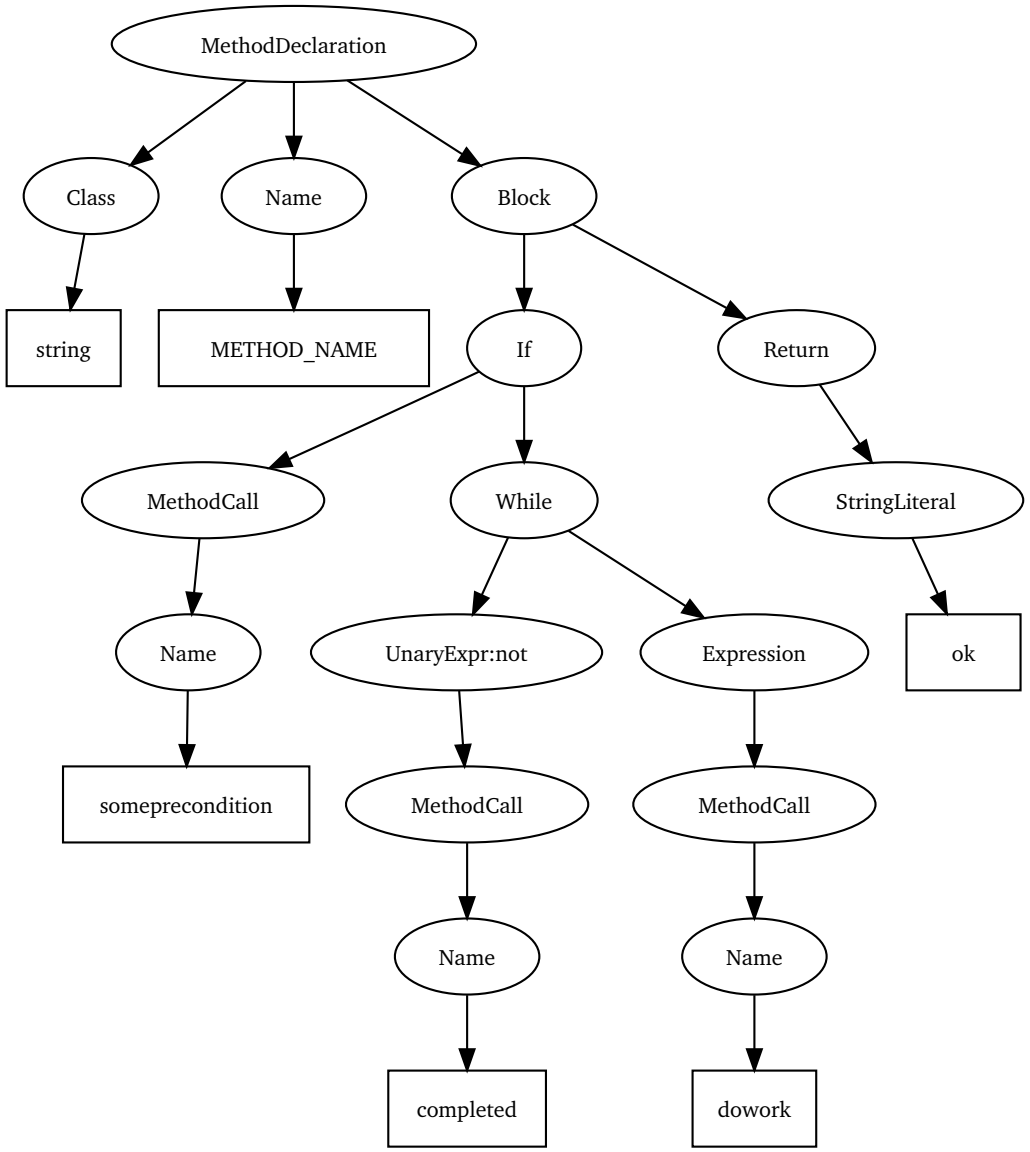


Figure 3.1. Abstract syntax tree of the source code presented in listing 3.1.

```
string,Class↑MethodDeclaration↓Name,METHOD_NAME
string,Class↑MethodDeclaration↓Block↓If↓MethodCall↓Name,someprecondition
string,Class↑MethodDeclaration↓Block↓If↓While↓UnaryExpr:not↓MethodCall↓Name,completed
string,Class↑MethodDeclaration↓Block↓If↓While↓Expression↓MethodCall↓Name,dowork
string,Class↑MethodDeclaration↓Block↓Return↓StringLiteral,ok
METHOD_NAME,Name↑MethodDeclaration↓Block↓If↓MethodCall↓Name,someprecondition
METHOD_NAME,Name↑MethodDeclaration↓Block↓If↓While↓UnaryExpr:not↓MethodCall↓Name,completed
METHOD_NAME,Name↑MethodDeclaration↓Block↓If↓While↓Expression↓MethodCall↓Name,dowork
METHOD_NAME,Name↑MethodDeclaration↓Block↓Return↓StringLiteral,ok
someprecondition,Name↑MethodCall↑If↓While↓UnaryExpr:not↓MethodCall↓Name,completed
someprecondition,Name↑MethodCall↑If↓While↓Expression↓MethodCall↓Name,dowork
someprecondition,Name↑MethodCall↑If↑Block↓Return↓StringLiteral,ok
completed,Name↑MethodCall↑UnaryExpr:not↑While↓Expression↓MethodCall↓Name,dowork
completed,Name↑MethodCall↑UnaryExpr:not↑While↑If↑Block↓Return↓StringLiteral,ok
dowork,Name↑MethodCall↑Expression↑While↑If↑Block↓Return↓StringLiteral,ok
```

Figure 3.2. AST-paths of listing 3.1.

To explain the picture above we need to introduce two matrices:

$$value\_vocab \in \mathbb{R}^{|X| \times d}$$
$$path\_vocab \in \mathbb{R}^{|P| \times d}$$

The embedding size d is a hyperparameter. X is the set of values of the AST terminals that were observed during training; in our recurring example this set is composed of: string, METHOD_NAME, someprecondition, completed, dowork and ok. P is the set of all AST-paths across all snippets.

These matrices are initialized randomly and are learned by the model during the training. An embedding (either from value_vocab or path_vocab) is looked up selecting the appropriate row in its matrix.

The previous notions tell us that:

$$c_i \in \mathbb{R}^{3d}$$

The two matrices value_vocab and path_vocab do not need to be of the same width d but for convenience it was chosen so.

3.2.3 Path-context

The previous section describes the definition of the context-vector c_i. Applying a fully connected layer to it we obtain the path-context vector $\widetilde{c}_i$, also called combined context-vector. The following equation describes the computation of this layer:

$$\widetilde{c}_i = \tanh(W \cdot c_i)$$

where W is the weight matrix and

$$W \in \mathbb{R}^{d \times 3d}$$

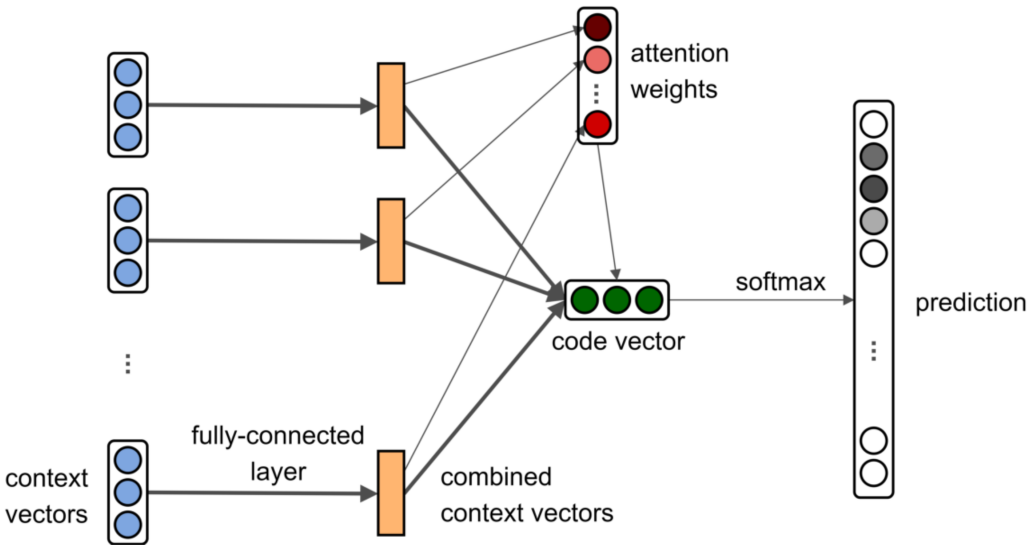


Figure 3.3. Code2vec architecture. Alon et al. [3]

The height of W is for convenience of the same size as before (d); it does not need to be strictly so: it can also be of different height. One other way to look at this layer is that it compresses the context-vector c_i of size $3d$ into a combined context-vector of size d.

3.2.4 Attention mechanism and the code-vector

The previous section left us with a set of path-contexts. The goal of this part of the model is to combine them all into a code-vector. This step employs an attention vector a (see figure 3.3; it can be described as a weighted average that . The weights are initialized randomly and learned with respect to the bag of path-contexts.

3.2.5 Training and prediction

The code-vector is used as input for a binary classifier. During the training, the model learns how to classify two classes: SATD and fixed. In the prediction phase it will calculate the probability that a specific class should be assigned to the given method body.

3.3 Hyperparameter Tuning

This section describes the process of tuning the hyperparameters of the model.

The initial experiments led us to quickly understand that using shorter snippets yielded better performances. For example, a dataset with snippets shorter than 25

tokens has 4,256 samples and gives a 73% test accuracy. A dataset with snippets shorter than 200 tokens has 106,272 samples and gives a 58% test accuracy.

Using a smaller set of the dataset decreases the usefulness of the trained model because of less capability in generalization. For the tuning of the hyperparameters, we empirically defined them to keep those snippets with less than 200 tokens; an example of a method composed of 199 tokens is shown in listing 3.2.

Listing 3.2. Code snippet with 199 tokens

```
private CompoundWorkflow finishCompoundWorkflow(
  WorkflowEventQueue queue,
  CompoundWorkflow compoundWorkflow,
  String taskOutcomeLabelId,
  String userTaskComment,
  boolean finishOnRegisterDocument,
  List<NodeRef> excludedNodeRefs) {
    if ((finishOnRegisterDocument &&
      compoundWorkflow.isStatus(Status.FINISHED)) ||
      (!finishOnRegisterDocument &&
      checkCompoundWorkflow(compoundWorkflow,
      Status.IN_PROGRESS,
      Status.FINISHED) == Status.FINISHED)) {
        if (log.isDebugEnabled()) {
            log.debug("--##string##--" + compoundWorkflow);
        }
    } else {
        setWorkflowsAndTasksFinished(queue, compoundWorkflow,
            taskOutcomeLabelId, userTaskComment,
            finishOnRegisterDocument, excludedNodeRefs);
        if (finishOnRegisterDocument || excludedNodeRefs != null) {
            stepAndCheck(queue, compoundWorkflow);
        } else {
            stepAndCheck(queue, compoundWorkflow, Status.FINISHED);
        }
        boolean changed = saveCompoundWorkflow(queue,
            compoundWorkflow, null);
        if (log.isDebugEnabled()) {
            log.debug("--##string##--" + compoundWorkflow);
        }
    }
    CompoundWorkflow freshCompoundWorkflow =
        getCompoundWorkflow(compoundWorkflow.getNodeRef());
```

```python
def objective(trial):
    default_embeddings_size = int(trial.suggest_discrete_uniform('default_embeddings_size', 32, 192, 16))
    max_contexts = int(trial.suggest_discrete_uniform('max_contexts', 250, 350, 10))
    dropout_keep_rate = trial.suggest_discrete_uniform('dropout_keep_rate', 0.05, 0.7, 0.05)

    evaluation, evaluation_detail, info, output = ('', '', '', [])
    try:
        evaluation, evaluation_detail, info, _ = full_pipeline.run(clean_token_count_limit=200
                                                                   , default_embeddings_size=default_embeddings_size
                                                                   , max_contexts=max_contexts
                                                                   , dropout_keep_rate=dropout_keep_rate
                                                                   , output=output)
        accuracy = float(prop2dict(evaluation)['accuracy'])
        return accuracy
    except Exception as ex:
        handle(ex)
        return None
```

Figure 3.4. Simplified version of the objective function - optuna_worker.py

```
    if (!finishOnRegisterDocument && excludedNodeRefs == null) {
        checkCompoundWorkflow(freshCompoundWorkflow, Status.FINISHED);
    }
    checkActiveResponsibleAssignmentTasks(
        freshCompoundWorkflow.getParent());
    return freshCompoundWorkflow;
}
```

We based the tuning operation on these three hyperparameters:

- default_embeddings_size: this value defines the length of the code vector, i.e. the vector representation of the snippet (see section 3.2.4).
- max_contexts: it is the maximum number of AST-paths used by the model (see section 3.2.1).
- dropout_keep_rate: the dropout is a random removal of neurons to prevent excessive adaptation to the training values and in so doing, reduce the likelihood of the network overfitting.

To explore the best values for these parameters we created a distributed experiment using Google Colab Pro and a tool called Optuna .

Optuna is defined as "A next-generation hyperparameter optimization framework". It is capable to construct the parameter search space dynamically and it implements both searching and pruning strategies [1]. The initial experiments were conducted with a competing tool, Hyperopt [8], but it was abandoned in favor of Optuna.

The Optuna distributed worker was easier to setup and the distributed experiment gave less friction than Hyperopt. Figure 3.4 shows a simplified version of the objective function required by Optuna where we define the search space for each hyperparameter. Our objective value is the test prediction accuracy.

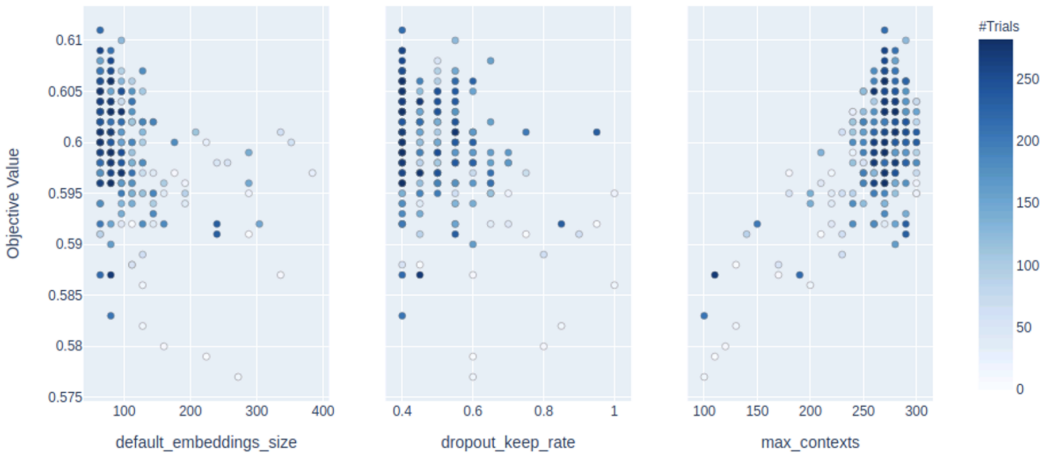


Figure 3.5. Slice plot for first grid search experiment

The first rounds of experiments yielded good values for max_contexts parameter: we found that the optimal value lies between 250-300. The other two parameters, default_embeddings_size and dropout_keep_rate, needed a different search space: figure 3.5 clearly shows the first two value bubbles crushed on the left. This suggested a new experiment with a different value search window shown in figure 3.7.

Google Colab Pro. The distributed experiment was executed on Google Colab infrastructure. We used 12 concurrent sessions, both with GPU and CPU. The Python notebook contained only a few lines of code (see listing 3.3). The first operation was to clone the GitHub repository with the source code for the distributed experiment. The second task was to invoke a Python function to setup the Colab environment: download PostgreSQL binaries, download and restore the backup with the dataset, install all the code2vec libraries and dependencies. The third and last step starts the Optuna worker. The results were stored in a central database as per Optuna design.

Listing 3.3. Google Colab notebook code

```
!cd /content; cd code2vec-satd-classifier && git pull || git clone \\
        https://github.com/simonegiacomelli/code2vec-satd-classifier
%cd /content/code2vec-satd-classifier/code2vec-satd
import satd_colab_starter as starter; starter.main()
!python3 optuna_worker.py
```

The best hyperparameters configuration found was the following:

- default_embeddings_size: 112
- max_contexts: 290
- dropout_keep_rate: 0.2

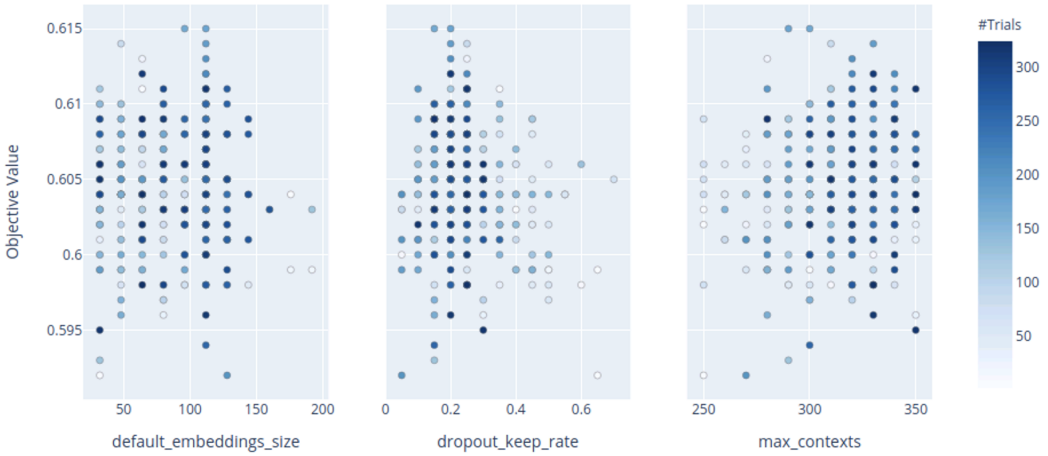


Figure 3.6. Slice plot for second grid search experiment with a centered search space

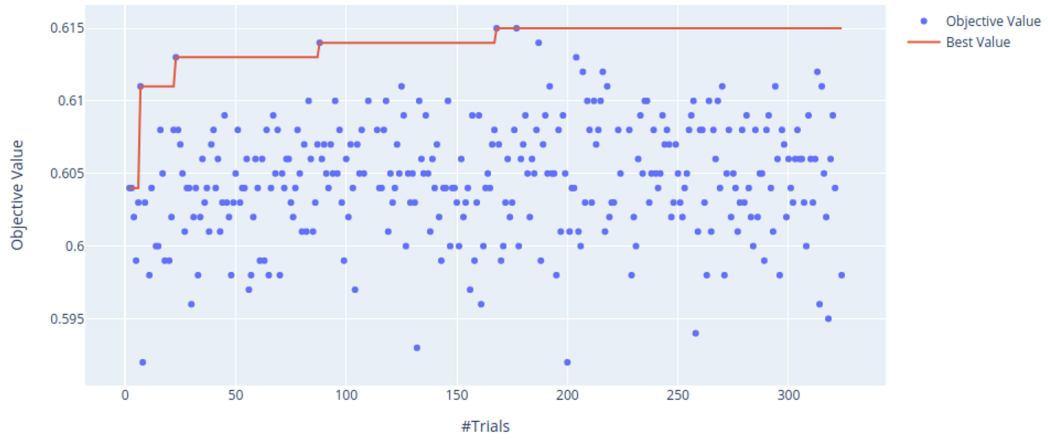


Figure 3.7. Second grid search experiment optimization history plot

This setup raised the accuracy from 58% (using default values) to 61.5%.

Chapter 4

Empirical Study Design

The goal of this exploratory study is to evaluate the accuracy of our approach for technical debt identification. This section provides details about the design and planning of the study aimed at assessing our model performances.

We answer the following research question: How does our approach perform in detecting technical debt in methods source code?

4.1 Context Selection

The context of the study consists of 245,243 public Java GitHub repositories.

We targeted GitHub open-source projects written in Java because of three reasons:

- The large number of accessible projects available.
- The availability of the commits history.
- The ease of parsing Java sources thanks to feature rich options of modern parsers.

We used GitHub GraphQL API [1] to retrieve 7,265,342 URLs of public Java repositories created between the start of year 2000 and the end of 2019, thus spanning a time window of 20 years. The GraphQL API allowed us to request two additional information for each repository: the number of issues and the number of commits. To exclude smaller and less meaningful repositories, we kept only those URLs with issue or commit count greater than 100. This reduced the number of repository URLs to 248,872.

After the clone phase we were left with 245,243 repositories because 3,629 were rejected or failed: 1,726 Android OS repository clones were rejected, 587 clones failed because they were removed or made private and 1,316 for other mixed reasons.

The processing of 245,243 repository commit histories yielded 141,400 method bodies annotated with keyword label SATD and their 141,400 fixed counterparts. For each SATD/fixed pair we collected the following information:

[1] https://docs.github.com/graphql

- Pattern: the word or sentence that identified the SATD.
- Commit message: the commit message that removed the SATD.
- Commit hash id.
- Repository name and url.
- The verbatim bodies involved in the change: before the commit (affected with SATD) and after (fixed).
- The 'cleaned' version of the bodies from the previous point (see 3.1.3 for details).

Of the 141,400 collected samples, 48'339 were rejected leaving us with 93,061 viable pair. Those rejections were enforced for the following reasons:

- Merge commits (i.e. commits with more than one parent). The reason being that we were specifically looking for a commit fixing a method SATD and a 'merge' operation is not likely to do that.
- Empty methods.
- Methods containing inner methods.
- Methods with one throw exception statement.
- Methods with unparsable statements.

For the distributed experiment of hyperparameter tuning, xx were selected amongst the 93,061 viable pairs.

continue describing the parsed information and the subsequent manual and automatic filtering ...

4.2 Data Collection and Analysis

To answer the research question we conducted a distributed experiment using code2vec Keras model (described in sections x and y) and optimized for the selected hyperparameters. We did x run with 50 epochs and tested on the one with the best score on the validation dataset.

4.3 Replication Package

A replication package is available on GitHub:

- The source code for repository mining and deep learning model [2].

[2]https://github.com/simonegiacomelli/code2vec-satd-classifier

- Two Postgresql database backups. [3] containing:
  - The dataset with the mined GitHub repository urls and all the method bodies collected.
  - The Optuna database containing the data of the distributed experiment (i.e. hyperparameters tuning) with all Keras outputs.

[3] https://github.com/simonegiacomelli/code2vec-satd-classifier-dataset

Chapter 5

Results Discussion

. . . results discussion . . .

5.1 Quantitative Results

Report precision/recall for different confidence thresholds

5.2 Qualitative Results

Discuss interesting cases in which your approach succeeds/fails

Chapter 6

Threats to Validity

. . . Discussion on the limitations of my study . . .

Chapter 7

Conclusion

. . . conclusion . . .

Appendix A

Additional material

Listing A.1. 61 patterns for SATD detection

```
# a line that starts with an hash is ignored
hack
retarded
at a loss
stupid
remove this code
ugly
take care
something's gone wrong
nuke
is problematic
may cause problem
hacky
unknown why we ever experience this
treat this as a soft error
silly
workaround for bug
kludge
fixme
this isn't quite right
trial and error
give up
this is wrong
hang our heads in shame
temporary solution
causes issue
something bad is going on
```

```
cause for issue
this doesn't look right
is this next line safe
this indicates a more fundamental problem
temporary crutch
this can be a mess
this isn't very solid
this is temporary and will go away
is this line really safe
#there is a problem
some fatal error
something serious is wrong
don't use this
get rid of this
doubt that this would work
this is bs
give up and go away
risk of this blowing up
just abandon it
prolly a bug
probably a bug
hope everything will work
toss it
barf
something bad happened
fix this crap
yuck
certainly buggy
remove me before production
you can be unhappy now
this is uncool
#bail out
it doesn't work yet
crap
inconsistency
abandon all hope
kaboom
```

Bibliography

[1] Takuya Akiba et al. “Optuna: A Next-generation Hyperparameter Optimization Framework”. In: *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2019.

[2] Eric Allman. “Managing technical debt”. In: *Communications of the ACM* 55.5 (2012), pp. 50–55.

[3] Uri Alon et al. “code2vec: Learning distributed representations of code”. In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), pp. 1–29.

[4] Nicolli SR Alves et al. “Identification and management of technical debt: A systematic mapping study”. In: *Information and Software Technology* 70 (2016), pp. 100–121.

[5] Nicolli SR Alves et al. “Towards an ontology of terms on technical debt”. In: *2014 Sixth International Workshop on Managing Technical Debt*. IEEE. 2014, pp. 1–7.

[6] Lucas Amorim et al. “Experience report: Evaluating the effectiveness of decision trees for detecting code smells”. In: *2015 IEEE 26th international symposium on software reliability engineering (ISSRE)*. IEEE. 2015, pp. 261–269.

[7] Gabriele Bavota and Barbara Russo. “A large-scale empirical study on self-admitted technical debt”. In: *Proceedings of the 13th International Conference on Mining Software Repositories*. 2016, pp. 315–326.

[8] James Bergstra, Daniel Yamins, and David Cox. “Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures”. In: *International conference on machine learning*. PMLR. 2013, pp. 115–123.

[9] Terese Besker et al. “Embracing technical debt, from a startup company perspective”. In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2018, pp. 415–425.

[10] Mohamed Boussaa et al. “Competitive coevolutionary code-smells detection”. In: *International Symposium on Search Based Software Engineering*. Springer. 2013, pp. 50–65.

[11] WJ Brown et al. “AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.(1998)”. In: *Google Scholar Google Scholar Digital Library Digital Library* ().

[12] William R Bush, Jonathan D Pincus, and David J Sielaff. “A static analyzer for finding dynamic programming errors”. In: *Software: Practice and Experience* 30.7 (2000), pp. 775–802.

[13] Daniel Cruz, Amanda Santana, and Eduardo Figueiredo. “Detecting bad smells with machine learning algorithms: an empirical study”. In: *Proceedings of the 3rd International Conference on Technical Debt*. 2020, pp. 31–40.

[14] Ward Cunningham. “The WyCash portfolio management system”. In: *ACM SIGPLAN OOPS Messenger* 4.2 (1992), pp. 29–30.

[15] Dario Di Nucci et al. “Detecting code smells using machine learning techniques: are we there yet?” In: *2018 ieee 25th international conference on software analysis, evolution and reengineering (saner)*. IEEE. 2018, pp. 612–621.

[16] Dawson Engler and Ken Ashcraft. “RacerX: effective, static detection of race conditions and deadlocks”. In: *ACM SIGOPS operating systems review* 37.5 (2003), pp. 237–252.

[17] Dawson Engler et al. “Bugs as deviant behavior: A general approach to inferring errors in systems code”. In: *ACM SIGOPS Operating Systems Review* 35.5 (2001), pp. 57–72.

[18] Dawson Engler et al. *Checking system rules using system-specific, programmer-written compiler extensions*. Tech. rep. STANFORD UNIV CA COMPUTER SYSTEMS LAB, 2000.

[19] Cormac Flanagan et al. “Extended static checking for Java”. In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. 2002, pp. 234–245.

[20] Francesca Arcelli Fontana et al. “Code smell detection: Towards a machine learning-based approach”. In: *2013 IEEE International Conference on Software Maintenance*. IEEE. 2013, pp. 396–399.

[21] Francesca Arcelli Fontana et al. “Comparing and experimenting machine learning techniques for code smell detection”. In: *Empirical Software Engineering* 21.3 (2016), pp. 1143–1191.

[22] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[23] Patrice Godefroid, Robert S Hanmer, and Lalita Jategaonkar Jagadeesan. “Model checking without a model: An analysis of the heart-beat monitor of a telephone switch using verisoft”. In: *Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*. 1998, pp. 124–133.

[24] Yuepu Guo, Rodrigo Oliveira Spínola, and Carolyn Seaman. “Exploring the costs of technical debt management–a case study”. In: *Empirical Software Engineering* 21.1 (2016), pp. 159–182.

[25] Yuepu Guo et al. “Tracking technical debt—An exploratory case study”. In: *2011 27th IEEE international conference on software maintenance (ICSM)*. IEEE. 2011, pp. 528–531.

[26] Sudheendra Hangal and Monica S Lam. “Tracking down software bugs using automatic anomaly detection”. In: *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. IEEE. 2002, pp. 291–301.

[27] Reed Hastings and Bob Joyce. *Fast Detection of Memory Leaks and Access Errors. Winter 1992 USENIX Conference*. 1992.

[28] Klaus Havelund and Jens Ulrik Skakkebæk. “Applying model checking in Java verification”. In: *International SPIN Workshop on Model Checking of Software*. Springer. 1999, pp. 216–231.

[29] Konrad Hinsen. “Technical debt in computational science”. In: *Computing in Science & Engineering* 17.6 (2015), pp. 103–107.

[30] Clemente Izurieta et al. “Organizing the technical debt landscape”. In: *2012 Third International Workshop on Managing Technical Debt (MTD)*. IEEE. 2012, pp. 23–26.

[31] Marouane Kessentini, Stéphane Vaucher, and Houari Sahraoui. “Deviance from perfection is a better criterion than closeness to evil when identifying risky code”. In: *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 2010, pp. 113–122.

[32] Wael Kessentini et al. “A cooperative parallel search-based software engineering approach for code-smells detection”. In: *IEEE Transactions on Software Engineering* 40.9 (2014), pp. 841–861.

[33] Foutse Khomh et al. “A bayesian approach for the detection of code and design smells”. In: *2009 Ninth International Conference on Quality Software*. IEEE. 2009, pp. 305–314.

[34] Foutse Khomh et al. “An exploratory study of the impact of antipatterns on class change-and fault-proneness”. In: *Empirical Software Engineering* 17.3 (2012), pp. 243–275.

[35] Foutse Khomh et al. “BDTEX: A GQM-based Bayesian approach for the detection of antipatterns”. In: *Journal of Systems and Software* 84.4 (2011), pp. 559–572.

[36] Tim Klinger et al. “An enterprise perspective on technical debt”. In: *Proceedings of the 2nd Workshop on managing technical debt*. 2011, pp. 35–38.

[37] Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. “Technical debt: From metaphor to theory and practice”. In: *Ieee software* 29.6 (2012), pp. 18–21.

[38] Philippe Kruchten et al. “Technical debt: towards a crisper definition report on the 4th international workshop on managing technical debt”. In: *ACM SIGSOFT Software Engineering Notes* 38.5 (2013), pp. 51–54.

[39] Michele Lanza and Radu Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[40] Valentina Lenarduzzi, Nyyti Saarimäki, and Davide Taibi. “The technical debt dataset”. In: *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*. 2019, pp. 2–11.

[41] Zhenmin Li et al. “CP-Miner: Finding copy-paste and related bugs in large-scale software code”. In: *IEEE Transactions on software Engineering* 32.3 (2006), pp. 176–192.

[42] Ben Liblit et al. “Bug isolation via remote program sampling”. In: *ACM Sigplan Notices* 38.5 (2003), pp. 141–154.

[43] Elvis Ligu et al. “Identification of refused bequest code smells”. In: *2013 IEEE International Conference on Software Maintenance*. IEEE. 2013, pp. 392–395.

[44] Erin Lim, Nitin Taksande, and Carolyn Seaman. “A balancing act: What software practitioners have to say about technical debt”. In: *IEEE software* 29.6 (2012), pp. 22–27.

[45] Scott M Lundberg and Su-In Lee. “A Unified Approach to Interpreting Model Predictions”. In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon et al. Curran Associates, Inc., 2017, pp. 4765–4774. URL: http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf.

[46] Abdou Maiga et al. “Smurf: A svm-based incremental anti-pattern detection approach”. In: *2012 19th Working Conference on Reverse Engineering*. IEEE. 2012, pp. 466–475.

[47] Abdou Maiga et al. “Support vector machines for anti-pattern detection”. In: *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. IEEE. 2012, pp. 278–281.

[48] Rungroj Maipradit et al. “Automated Identification of On-hold Self-admitted Technical Debt”. In: *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE. 2020, pp. 54–64.

[49] Everton da S Maldonado and Emad Shihab. “Detecting and quantifying different types of self-admitted technical debt”. In: *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*. IEEE. 2015, pp. 9–15.

[50] Christopher Manning and Dan Klein. “Optimization, maxent models, and conditional estimation without magic”. In: *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology: Tutorials-Volume 5*. 2003, pp. 8–8.

[51] Radu Marinescu. “Detection strategies: Metrics-based rules for detecting design flaws”. In: *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* IEEE. 2004, pp. 350–359.

[52] Antonio Martini, Terese Besker, and Jan Bosch. “Technical Debt tracking: Current state of practice: A survey and multiple case study in 15 large organizations”. In: *Science of Computer Programming* 163 (2018), pp. 42–61.

[53] Naouel Moha et al. “Decor: A method for the specification and detection of code and design smells”. In: *IEEE Transactions on Software Engineering* 36.1 (2009), pp. 20–36.

[54] Matthew James Munro. “Product metrics for automatic identification of" bad smell" design problems in java source-code”. In: *11th IEEE International Software Metrics Symposium (METRICS’05)*. IEEE. 2005, pp. 15–15.

[55] Madanlal Musuvathi et al. “CMC: A pragmatic approach to model checking real code”. In: *ACM SIGOPS Operating Systems Review* 36.SI (2002), pp. 75–88.

[56] Nicholas Nethercote and Julian Seward. “Valgrind: a framework for heavyweight dynamic binary instrumentation”. In: *ACM Sigplan notices* 42.6 (2007), pp. 89–100.

[57] Rocco Oliveto et al. “Numerical signatures of antipatterns: An approach based on b-splines”. In: *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE. 2010, pp. 248–251.

[58] Fabio Palomba et al. “Mining version histories for detecting code smells”. In: *IEEE Transactions on Software Engineering* 41.5 (2014), pp. 462–489.

[59] Aniket Potdar and Emad Shihab. “An exploratory study on self-admitted technical debt”. In: *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE. 2014, pp. 91–100.

[60] Leevi Rantala, Mika Mäntylä, and David Lo. “Prevalence, Contents and Automatic Detection of KL-SATD”. In: *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE. 2020, pp. 385–388.

[61] D Rapu et al. “Using history information to improve design flaws detection”. In: *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.* IEEE. 2004, pp. 223–232.

[62] Xiaoxue Ren et al. “Neural network-based detection of self-admitted technical debt: from performance to explainability”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28.3 (2019), pp. 1–45.

[63] A.J. Riel. *Object-oriented Design Heuristics*. Object oriented technology. Addison-Wesley Publishing Company, 1996. ISBN: 9780201633856. URL: https://books.google.it/books?id=oHkhAQAAIAAJ.

[64] Dilan Sahin et al. "Code-smell detection as a bilevel problem". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24.1 (2014), pp. 1–44.

[65] Everton da Silva Maldonado, Emad Shihab, and Nikolaos Tsantalis. "Using natural language processing to automatically detect self-admitted technical debt". In: *IEEE Transactions on Software Engineering* 43.11 (2017), pp. 1044–1062.

[66] Frank Simon, Frank Steinbruckner, and Claus Lewerentz. "Metrics based refactoring". In: *Proceedings fifth european conference on software maintenance and reengineering*. IEEE. 2001, pp. 30–38.

[67] Rodrigo O Spı**********nola et al. "Investigating technical debt folklore: Shedding some light on technical debt opinion". In: *2013 4th International Workshop on Managing Technical Debt (MTD)*. IEEE. 2013, pp. 1–7.

[68] Margaret-Anne Storey et al. "TODO or to bug: Exploring How Task Annotations Play a Role in the Work Practices of Software Developers". In: *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE. 2008, pp. 251–260.

[69] Ewan Tempero et al. "The Qualitas Corpus: A curated collection of Java code for empirical studies". In: *2010 Asia Pacific Software Engineering Conference*. IEEE. 2010, pp. 336–345.

[70] Edith Tom, Aybuke Aurum, and Richard Vidgen. "A consolidated understanding of technical debt". In: (2012).

[71] Edith Tom, AybüKe Aurum, and Richard Vidgen. "An exploration of technical debt". In: *Journal of Systems and Software* 86.6 (2013), pp. 1498–1516.

[72] Guilherme Travassos et al. "Detecting defects in object-oriented designs: using reading techniques to increase software quality". In: *ACM Sigplan Notices* 34.10 (1999), pp. 47–56.

[73] Nikolaos Tsantalis and Alexander Chatzigeorgiou. "Identification of move method refactoring opportunities". In: *IEEE Transactions on Software Engineering* 35.3 (2009), pp. 347–367.

[74] Eva Van Emden and Leon Moonen. "Java quality assurance by detecting code smells". In: *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* IEEE. 2002, pp. 97–106.

[75] Alberto Villar, Santiago Matalonga, and Montevideo Uruguay. "Definiciones y Tendencia de Deuda Técnica: Un Mapeo Sistemático de la Literatura." In: *CIbSE*. 2013, pp. 29–42.

[76] Xin Wang et al. “Detecting and Explaining Self-Admitted Technical Debts with Attention-based Neural Networks”. In: *The 35th IEEE/ACM International Conference on Automated Software Engineering (ASE 2020)*. 2020.

[77] Bruce F Webster. *Pitfalls of object-oriented development*. M & T Books, 1995.

[78] Sultan Wehaibi, Emad Shihab, and Latifa Guerrouj. “Examining the impact of self-admitted technical debt on software quality”. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. IEEE. 2016, pp. 179–188.

[79] Fiorella Zampetti, Alexander Serebrenik, and Massimiliano Di Penta. “Automatically learning patterns for self-admitted technical debt removal”. In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2020, pp. 355–366.

[80] Nico Zazworka et al. “A case study on effectively identifying technical debt”. In: *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*. 2013, pp. 42–47.

[81] Pin Zhou et al. “AccMon: Automatically detecting memory-related bugs via program counter-based invariants”. In: *37th International Symposium on Microarchitecture (MICRO-37’04)*. IEEE. 2004, pp. 269–280.