The architectural erosion problem in Andorid Apps

Identifying and Detecting Architectural Erosion in Android Apps

Juan Camilo Acosta Rojas

April 22, 2025

This thesis is submitted in partial fulfillment of the requirements for a degree of Master in Systems and Computing Engineering (MISIS).

Thesis Committee:

Prof. Name Last name (Promotor)	Universidad de los Andes, Colombia
Prof. Reviewer 1 Reviewer 1	Institution, Country
Prof. Reviewer 2 Reviewer 2	Institution, Country

Identifying and Detecting Architectural Erosion in Android Apps

The Software Design Lab

Systems and Computing Engineering Department

Faculty of Engineering

Universidad de los Andes

Bogotá, Colombia

To my family. The fruit of all their efforts be shown here.

Abstract

Software engineering involves different efforts with the same purpose: high-quality software development. Among these efforts, we can find the software architecture definition process, which gives preference to the various quality attributes according to the system's needs. However, for different reasons, the solution development can deviate from the original architecture design; it could generate performance problems, and as a consequence, it affects the user experience. This impact on software quality could be represented, at a greater rate, in a mobile environment due to its limited resources. Previous research with this approach has focused on problem study and detection in different areas like security, connectivity, etc. However, in mobile development, the concept of "architectural erosion" has not been deeply studied. The main goal of this research project is to detect and locate architectural erosion bugs in mobile applications with automated static and dynamic analysis based on deep learning models.

Acknowledgements

Contents

List of Figures

Chapter 1

Introduction

In the beginning of system building, we might undergo a strong design process, where you define the system components and resources you need to use. This process is named *software architecture definition*. After that, you have to align the development process with those architectural rules to achieve goals in performance, security, availability, and others. However, due to growth in software development, as time goes by, software tends to violate architectural rules, affecting system quality attributes. This deviation is called *architectural erosion*. In mobile development, it can reduce application performance and other system quality attributes, affecting device resources and, ultimately, the user experience. In recent research, there have been significant advances in bug resolution across different approaches. For example, in security, connectivity, and code smells, there are various tools and components in both stacks (Frontend and Backend) that detect and offer solutions for different bugs related to these approaches.

However, in the architectural erosion analysis, despite 73 research studies tackling this concept, some of these propose a toolset; it is not clear how we can resolve architectural erosion insights or a set of directives for facing this problem in the mobile development ecosystem. For this reason, it is important to find the impact of architectural erosion in mobile applications and how we can detect, give alerts, and provide solution recommendations for fixing architectural erosion bugs with static analysis and the help of natural language processing techniques.

1.1. Why make researching efforts in architectural erosion?

Due to different researches, the costs in terms of human resources, technological resources, time and money would grow after the first release and deployment of the software project. By this, is necessary to establish the different reasons and artifacts that affect the software sustainability, the software performance in short, middle and long term. For this task, there are many some approaches that analyze different software project components like its design, its code, its architectural rules. One of the most important, and one of the most studied too, has been the static code analysis approach. In this approach, has been created a lot of tools for detect architectural erosion in specific code fragments, showing a analysis along time. This tools has been very helpful for the developers and for the teams, but it has not been enough in mobile development ecosystem, where system resources are mire limited that side-server environment. Is necessary to build a component with specific purpose in Android Apps and show the importance of detect architectural erosion in mobile app code and recover of this issues in a early way.

1.2. Machine learning models in architectural erosion

Recent research in Machine Learning models, specifically in Natural Language Processing, has concluded that the use of pre-trained models for the classification or generation of words in a specific language, their performance with programming languages is better than natural languages (in this NLP approach, ex: English, Spanish, e.t.c), due to their syntax. This idea could be recreated in issues detection in source code repository analysis. Issues like masked code, pattern design detection, and other ones have been implemented with pre-trained machine learning models. In architectural erosion, it could be useful for detecting the first symptoms of this and making early detection for, after that, using specific rules of architecture in static code analysis. In resume, machine learning models could detect the first symptoms of architectural erosion in an Android project and create a component with optimized performance for detecting and recovering issues about this.

Chapter 2

Definitions

2.1. Development Process of a Software Project

When you have to make a software project plan, you have to define the main requirements that achieve the main goals of the business idea within the software project realization motivation. First, you have to define the functional requirements. In software engineering, functional requirements define the main features of each software component involved in the project. Requirements like design, basic functionalities, and defined flows are built on the functional requirements definition process. After that, you have to define the main requirements that do not directly impact the user experience, which are called nonfunctional requirements. Nonfunctional requirements indirectly affect your software project; bad planning about those requirements could progressively affect the user performance and the software project architecture, becoming slower, heavier, or less useful for a single application user. In this process, you must design a solution that satisfies the functional and non-functional requirements. This process is called the design process of a software project. Inside this stage, you must define the number of components that give the software its functionalities. Furthermore, you have to each component infrastructure, into a process named Software Architecture Definition Laplante [5]. The software development process involves some steps:

2.1.1. Discovery

This stage consists of the definition of the main features of the software application. The discovery process defines the objective of the application and its main requirements. Furthermore, this gives an initial version of the behavior and the design of the software application.

2.1.2. Design

After the discovery of the application requirements, it is possible to define the design of the components of the software application and their relationships. In this stage, we define the functional and non-functional requirements. The functional requirements define the main features and use cases of the software application, defining a complete flow with the final user. The non-functional requirements define the performance metrics that the application must achieve. Based on the ISO rule, which defines the quality assurance of any software application. There is a set of defined non-functional requirements categories for achieving good performance in a software application: Scalability, availability, security, latency, integration capacity, and modification capacity (We will talk about these categories later). Once the prioritized non-functional requirements are defined, we define the rules and standards that will manage the performance of the software application.

2.1.3. Development

With the initial design of the software application and its components, we can implement and develop the software application. With a bad design process or a bad development process, it is possible to generate a deviation from the initial design of the application or a violation according to the initially defined standards of the sotwa|re application.

2.1.4. Testing and Quality Assurance (QA)

With the design of the software application and its implementation, it is possible to test the main features and flows based on the designed use cases of the software application. In this stage, it would report code smells or bad performance metrics, according to the prioritized non-functional requirements.

2.1.5. Release

Once the application is tested, it achieves its performance metrics. The software application deploys on a production environment, with the final users' direct interaction. If the design process or the development process is implemented badly, it would affect the user performance of any software application. Focused on Android applications development, the user experience would decay significantly, decreasing the number of active users or increasing the resources used in Android devices as the main consequences.

2.1.6. Maintenance

After the release and deployment stage. For a guarantee of the application's sustainability. It should create a refactoring process. That process consists of libraries updating, code smells fixing, and fixing bad design or development issues. The costs of one software application in terms of human resources, time, and money could increase the time goes by if the latest stages are implemented badly.

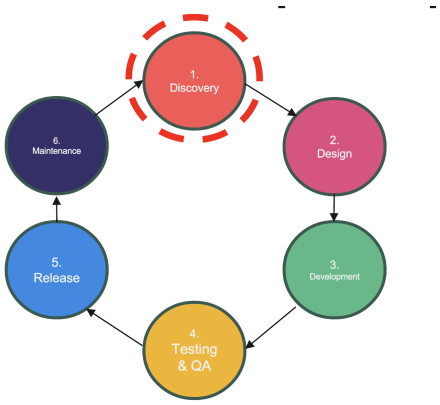

Figure 2.1: The development process of a Software Application

2.2. Architectural design in software engineering

For an effective and efficient software project development, we have to realize the respective investment in each stage. From the functional requirements creation to the design of each

component and its features and constraints. To do this, it is necessary to make an architectural design. The architecture of a software project. In general, defines the components that require the entire system and the connections and relationships (and their types). Furthermore, each component has to be defined in this design. In that order, different standards ensure the system's quality according to its architectural design.

2.2.1. Architectural design in Android development

Architectural Design in the mobile approach has been advancing step by step; this depends on the creation of new technologies in each development layer since the data management layer with the implementation of new database libraries like the room to the User Interface layers with a different way of creating new screens inside the application (with the use of either JetPack Compose or Fragments organization). Depending on what simple should be an application, how many components it would have, which of them would be connected, and the reasons for those connections. It is necessary to review different architectural patterns used recently and why we focus on one of them for architectural erosion detection.

- MVC (Model View Controller): In this pattern, we divide the application components into the model, where we implement the connections with external platforms and internal data management. The controller component is used for setting the relation between the business logic and the User Interface (UI). The view component contains the UI. This pattern has been commonly used for the last 15 years due to its simplicity and popularity. However, the applications that use this pattern are very coupled, and their components depend strongly on others. It is usually to find business logic implementations and UI code fragments in the same code file, or data processing in business code components. At the beginning of Android, architectural issues weren´t as important as they are today. If an Android application is simple in terms of realization, it is possible to use an MVC pattern.

- MVP (Model View Presenter): This pattern is managed in a different way than the MVC pattern in the relations between business logic and UI. In this case, we use a component named presenter to manage the events and behavior of each UI view or screen. This pattern is commonly used for single Applications that do not have scalability or application

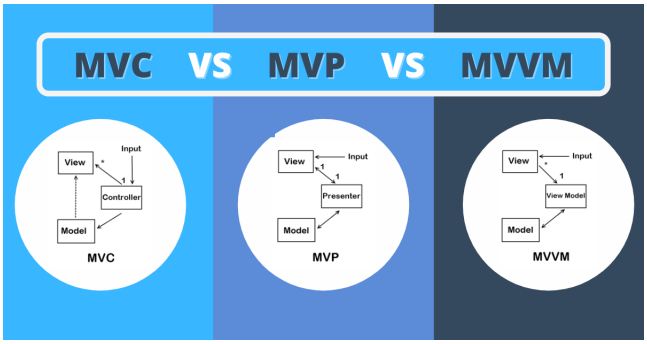

Figure 2.2: Similarities and differences between Android architectural patterns.]

overloading. This pattern divides the presenter features connected with the UI features. The disadvantages of this pattern are related to a high coupling rate inside their components and a big complexity for managing the life cycles of an application. Furthermore, it is difficult to implement new feature development and maintenance for large-scale Android applications.

- MVM (Model View View-Model): This pattern is one of the derivatives of clean architecture, a concept widely developed in Backend and Frontend applications architectures. This pattern uses some concepts of clean architecture, like use case organization, when we implement a new feature in an independent way of others. With this pattern, we use reactive components. The application components use libraries like Dagger Hilt to implement the use of reactive data; this reactive data changes depending on a UI event. With the creation of the JetPack Compose framework. The JetPack Compose is based on reactive UI and is more declarative than the traditional form (the use of XML files and fragments structure for managing different application screens)

In an Android application, it is not mandatory to use only an architectural pattern to achieve the functional and non-functional requirements of a software project. For example, it is very common to use the repository pattern declared in MVVM, divided into two: external connections and internal data management. Today, in the actual mobile development ecosystem, the most common framework that could be implemented with one or more architectural patterns is the three-layer pattern architecture: The UI layer, an optional layer named the Domain Layer, and

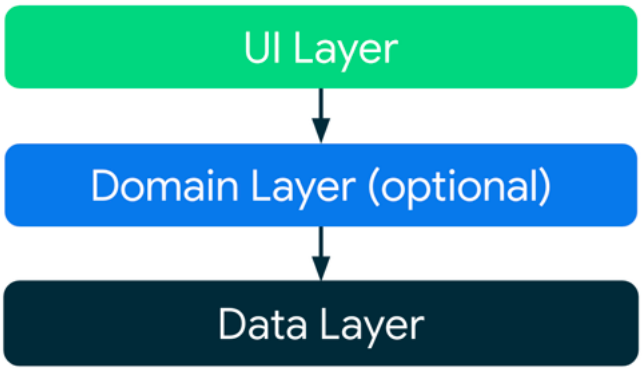

Figure 2.3: Three-layer framework for mobile development.]

the Data layer. Each layer could be or could not be implemented depending on each application's requirements set.

The three-layer architecture will be used as the main architectural framework to realize this study with the MVVM architectural pattern. These patterns are the most used in modern mobile development and will give a first approach for how the developers can detect and fix architectural violations in an Android application,

2.2.2. Quality Attributes

The consequences of the generation of architectural erosion could impact the different quality attributes contemplated. According to Bass Bass et al. [2]. there are six attributes based on ISO-25010. Those attribute qualities are:

- Latency: This attribute measures the response time of the components according to the implemented architecture. Response times of each architectural component are very important.
- Scalability: Measures the ability of a system to grow in critical situations of system inputs. It is important for system availability when the user's connection rate increases.
- Security: Nowadays, a system must give protection to its users and their data, since availability, integrity, and confidence of that data. This requirement is very important in the

legal environment of a system.

- Availability: When the system can be available when a system failure occurs (it doesn't depend on the fail type). This quality attribute measures the time that the system uses for recovering a fail(s).

- Integration capability: Measures the ability of a system to integrate with other(s) system(s), measures the time and effort that the system needs to make that integration in all their layers from the data layer till the UI layer is necessary.

- Modification capability: Similar to the last quality attribute, it measures the time that the system needs to change one or more components inside its system. The effort measure could be many relationships (human resources effort, time costs, money costs, etc.)

2.3. Architectural Erosion

The Architecture Erosion concept was treated a long time ago. Since 1992, architecture erosion has had a formal definition, giving the relationship of this concept with architectural rules violations Perry [8]. It can be treated from different perspectives, from the violation of rules, structure, and quality to the evolution perspective. Furthermore, the concept has a strong relationship with other synonyms, like, for example, degradation. In another modern definition, architectural erosion is defined as the set of architectural violations that reflect the deviation of the implemented architecture from the intended architecture over time. In resume, in a more concrete definition, architectural erosion could be considered a phenomenon that reflects the deviation of an implemented architecture due to an intended architecture in a software project.

2.3.1. Approaches and Perspectives

Due to the original concept of architectural erosion Ruiyin et al. [9]. It could be studied and analyzed by different approaches:

- Violation perspective: Denotes how the implemented architecture violates the design principles or main constraints of the intended architecture. These violations could occur in

two phases: the design phase and the maintenance and evolution phase, making different changes step by step in the short and long term.

- Structure perspective: Where the structure of a software system encompasses its components and their relationships.
- Quality perspective: it refers to the degradation of the system quality, due to architectural changes that would generate architectural smells. It could include all the quality attributes contemplated actually in the industry.
- Evolution perspective: It shows the architectural inflexibility that increases the difficulty of implementing changes in the project and, therefore, decreases the sustainability of the system.

2.3.2. Main Reasons and Symptoms

Different factors could provoke architectural erosion in different stages of software project development. Due to these reasons, is possible to make different solution approaches, and, therefore, the possibility for build different components based on that approaches. The main reasons founded are:

- Architecture modularization: Due to the business needs, is necessary to divide responsibilities between different components and layers in a software project. but, sometimes it could produce non-functional components and deviate from the initial intended architecture.
- Architecture complexity: if the intended architecture is very complex, is possible to deviate from it when the time goes by, producing the first symptoms of architectural erosion.
- Architecture size: Due to this attribute, and with no control over the maintenance of the software project, is not possible to have good software maintenance for a long time.
- Design Decisions: If the design decisions during the initial stages of the project don't have enough support (like documentation, reviews, etc.), it could generate problems with the maintenance of refactoring of different components, decreasing the code quality and generating the first symptoms of architectural erosion.

- Duplicate functionality: As a consequence of the latest reasons too, duplicate functionality reflects the bad connection between layers of an architecture, and, could be considered as the initial symptom of architectural erosion.

There are another reasons like bad documentation, bad programming features, but, in general, the main reasons were considered accord to the architecture design stage issues.

2.3.3. Consequences

Are several points of view about the real definition of consequences of architectural erosion violations inside a software project. The main problems that could generate the non correct maintenance and the deviaton of an intended architecture are:

- Costs of software maintenance: Due to the no implementation of architectural erosion issues, this could affect one of the non-functional defined requirements, and after that, affect the actual software infrastructure.

- Software Performance: When one of the quality attributes is affected by the initial architecture design planning, it incurs performance reduction of the intended architecture. In Android Apps, it could be more notorious, due to the limited resources of an mobile device, very different of a desktop device or server-type device.

- Software quality decrease: When architectural changes in a software project, it could affect the normal behavior of the application and, inside its source code, could imply bad code features implementation, decreasing the software quality, an important standard in a software project.

- Software Sustainability: The cost in terms of human resources, software infrastructure, time, money, etc, could increase if you do not attend the architectural erosion insights in your software project, affecting nonfunctional requirements, and, in a long time, affecting the user performance.

2.3.4. Metrics and treatments for architectural erosion

Today. exist some different metrics that can determine architectural erosion in different approaches. According to Baabad et al. [1]. Some metrics have been created to analyze architectural decay in open-source projects, analyzing possible reasons, indicators, and solution strategies for it. In the resume, exist around 54 metrics that could determine architectural erosion in different stages, from the design stage to the deployment stage. Those metrics have been classified by measured artifacts, level of validation, usability, applicability, comparative analysis, and support tools. Different classifications could be implemented into a tool with a specific measure strategy for analyzing architectural erosion in any system.

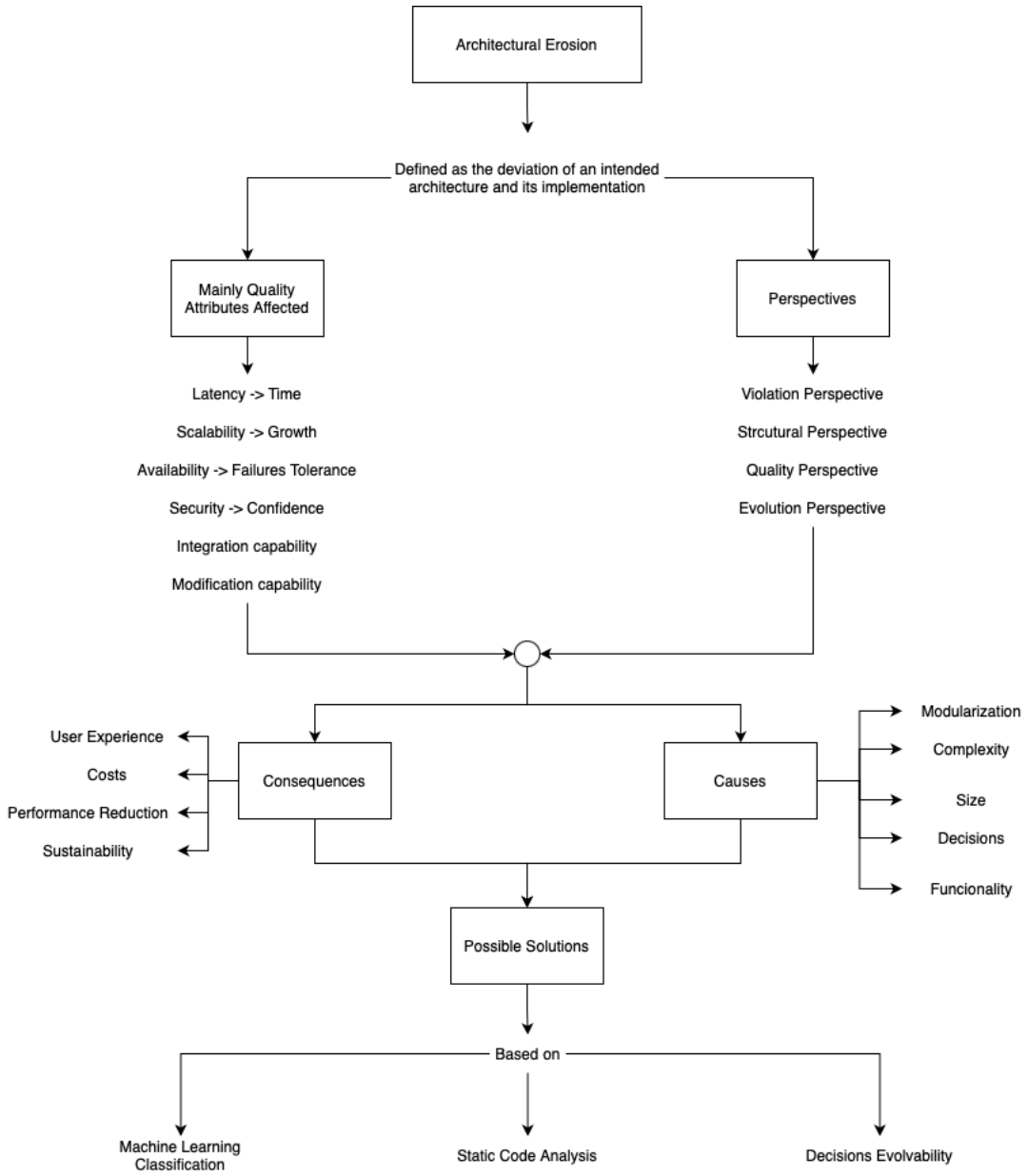

Figure 2.4: Main Concepts of Architectural Erosion in Software Engineering

2.4. Representative set of a sample

In statistics, several dataset sampling ways are effective in selection criteria, each alternative has a way for selecting data randomly that represents in a minor scale all the dataset. One way is to select a representative set from the original population. In this case, it is possible to make a weighted-randomly selection of some individuals from any population that represents the main features of all population *reference representative set*

2.5. Natural Language Processing in Software Issues Detection

Before solving architectural erosion insights into a software project, you have to detect the real violation types that could appear in your software project. There are some approaches for software issue detection using actual natural language processing methodologies that have been powered with Machine Learning Techniques' help. Natural Language Processing (NLP) gives the ability to extract relevant information for great text sets named corpus. Nowadays, NLP is very useful for many tasks like text generation or classification. Even with a different approach, the NLP actual tools have had a better performance of the same tasks in code due to its standard structure, which does not present different language variations that could present in a spoken language.

NLP has a set of different preprocessing methods for language models. Due to the complexity of text representation for computer processing, it is necessary to define a standard input structure for a model language. Before this, you have to make a series of processes for getting the standard structure of a given corpus of text. The process that enables those actions is named text normalization Jurafsky [4]. With the use of regular expressions, you could build a dictionary of words, which is useful for getting a standard structure and enable the text words with the modeling task. The main stages and process for the dictionary building are:

- Tokenization: Given a character sequence, in this case, a sequence of words of a given context, you can split that sequence into minimal processing units named tokens. These tokens are normally defined as terms of words. These tokens will create the base dictionary to begin the text processing into a language model Manning [7].

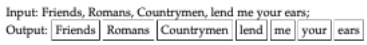

Figure 2.5: Tokenization process of one sentence Manning [7]

- Removing Stop Words: In the basic language modeling tasks, it is necessary to remove words that do not have relevant semantic information inside a text or a text corpus; those words are named Stop Words. Stop Words are words that do not contribute to the meaning of a sentence, like prepositions, articles, etc. In actual language models (Large Language Models), it is very important to maintain Stop Words to get a better specific context for next-word prediction or sentence classification. There are different strategies for removing those words; the most common is removing by collection frequency, due to the number of appearances in the corpus, which is enormous compared with relevant words. In document retrieval, the rare words are the most important for giving an efficient model over the text corpus Manning [7].

- Lemmatization: For new tokens controlling and token derivations, is essential to create tokens since the roots of the words, to reduce inflectional forms and related forms of words with the same root. In some cases, it is difficult to implement that process because you must have a root word dictionary to get root tokens. In this case, in different contexts, could generate conflicts for getting roots of specific context words [7].

- Stemming: This process consists of a heuristic process to cut off some characters at the end of each word, reducing the derivation of some words, with the same objective of the lemmatization process. In English, language could be an efficient technique but could have some conflicts with other languages.

2.5.1. Word Embeddings for Words Representation

As said in the last section, language models need to have a numerical representation of the corpus text for modeling tasks. To solve this, you must define a standard structure based on the decided model inputs. The most common structure is a word embedding representation, where you define a numerical vector for representing into a specific context (where the embedding was trained)

for use as input into a language model. This representation gives all model vocabulary a vector representation, where you can observe similar words, different words, and how much distance is between them. This representation is useful for similarity word management and getting the relationships between different features inside them Jurafsky [4].

Figure 2.6: Representation in two dimensions of Similar Words gives a Word Embedding Jurafsky [4]

2.5.2. Performance and Similarity Metrics in NLP

It is very important to maintain and define an objective in terms of performance. Different metrics represent the behavior of a classic or modern language model based on next-word probabilities (like the anagram model when you generate an n-tuple of words and calculate the occurrence probability of that word sequence). For information retrieval, when you, in the same case of word embedding representation, have a vector representation, you must use a metric based on the vector's components. In the same dimension, you could determine the similarity between two vectors and, in the case of NLP context, verify the semantic similarity between two words. One of the most used metrics in this approach is cosine similarity. This metric combines the product of the two vectors and the difference between their components. The Similitude Cosine metric is defined as:

$$cos(v, w) = \frac{v.w}{|v||w|} = \frac{\sum_{i=1}^{N} v_i w_i}{\sqrt{\sum_{i=1}^{N} w_i^2}\sqrt{\sum_{i=1}^{N} v_i^2}} \tag{2.1}$$

With this metric, we can conclude the similarity between two words (or two documents without another research context). If the similarity value is high, the words can be considered similar.

2.6. Programming Languages fundamental for Software Analysis

With the fundamentals of programming languages, it is possible to analyze the software building. Programming languages have a defined vocabulary, structure, and semantics. Those features make the use of AI tools and NLP techniques more effective. Furthermore, it is possible to create custom rules for checking a defined set of guidelines for a standard structure of different patterns in a software project. This is possible by detecting different semantical, grammatical, and lexical patterns inside the source code of any software project. In general, a programming language is defined by the next components:

- Lexical component: In that component, we define the vocabulary and the set of words that will have a meaning for the programming language. For example, the word function in JavaScript programming language means a function declaration, or int in Java, which means the Integer primitive data type. It is necessary to define all the words that could be used in any source code file of that programming language.

- Grammatical component: With a defined set of words in the lexical component, the next step is to define the order in which words could be written in a code block. It is essential to define all the possible structures that could be defined in any program and the different ways that could be written. For example, most programming languages used in the industry have a defined structure of if statements and all the possible ways to write them.

- Semantical component: If we have a set of words and a defined order to write them, it is possible to build a kind of translator for a programming language. In this step, we define the type of translator with two options: for executable program building, that is, a compiler, one example of it is C language programming, that a semantic visitor that generates an executable program, through a translation process between C code fragments and machine instructions. The other option is to make an interpreter, where in execution time, we visit the line by line of code and translate it into a machine instruction; an example of that approach is the Python programming language. In both approaches, we specify the

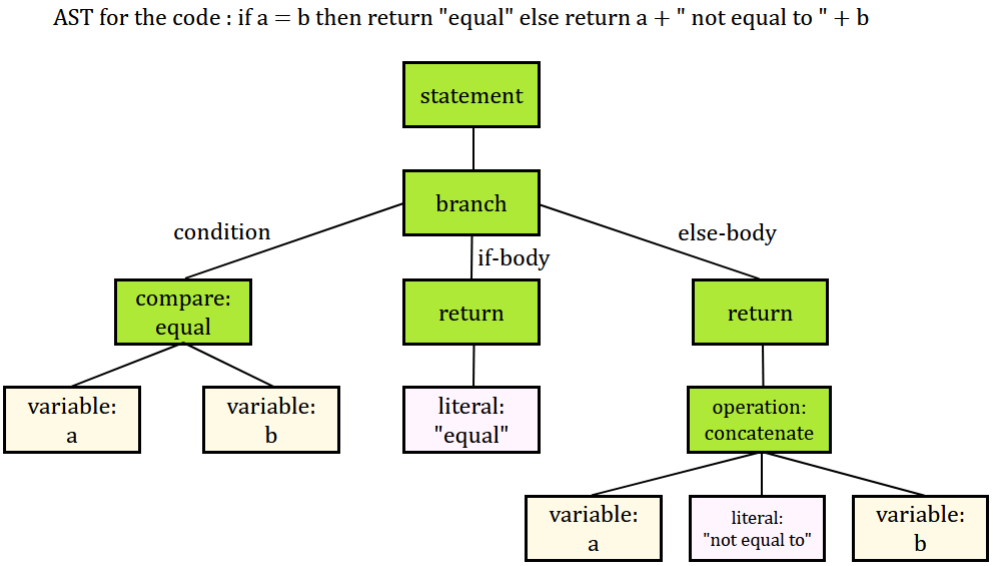

Figure 2.7: Example of an Abstract Syntax Tree (AST) statement.]

translation strategies mainly with two components: a visitor component and a call graph component.

- Visitor component: A visitor component consists of a structure built from grammatical and lexical components of a programming language. In that structure, we can find how all the statement code blocks are defined in all source code files of a software project. We can find the name of every parameter declared in any function and the name of any class of any code statement defined in any source code file. With this component, we can detect any pattern in names and data types inside all code fragments and combine them for more customized check rules (security issues, connectivity issues, and other).
- Call Graph component: The call graph component is very similar to the visitor component. The main difference is that we can find all the dependency relationships between all the source code files of a software project. With this dependency structure, we can detect the high dependency between components and their high coupling rate.

```
private static void Main(string[] args)
{
    if (!args.Any()) return;
    var integer = ParseSomeData(args[0]);
    Console.WriteLine(integer);
}

private static int ParseSomeData(string data)
{
    Console.WriteLine(data);
    return int.Parse(data);
}
```

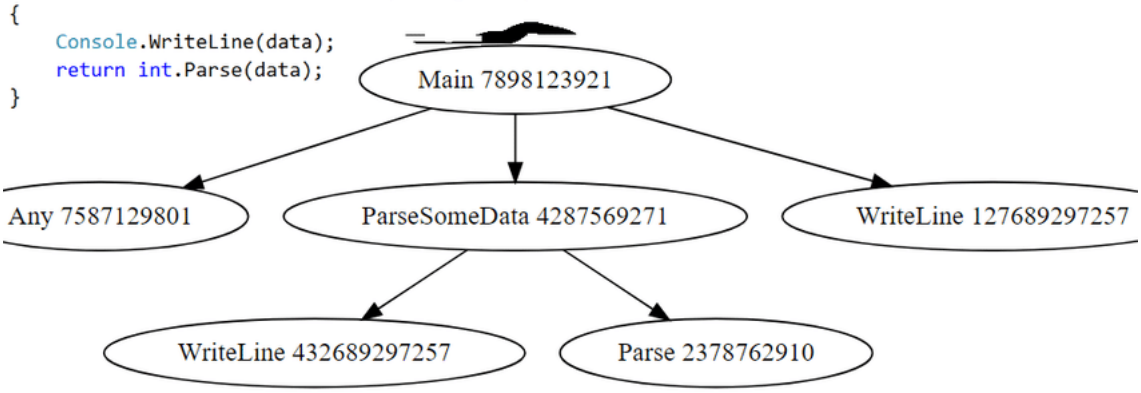

Figure 2.8: Example of an Abstract Syntax Tree (AST) statement.]

With all the programming language components and the way of translating it to machine instructions, it is possible to make custom check rules for different violations detection to different standards. In architectural erosion detection, these concepts will be fundamental.

Chapter 3

Related Work

3.1. Research Methodology

For architectural erosion symptoms, causes, and consequences detection, during the research, it was necessary to realize a series of steps for identification and possible recovery processes. Due to the orientation of the actual architectural erosion solvers for solving them mostly for Backend and Frontend projects, it is necessary to identify architectural erosion symptoms in Android apps. After that, with human judges, these symptoms have to be confirmed, and, finally, those symptoms will generate architectural rules for detecting them and suggest different recovering ways during the coding stage of a system.

3.1.1. Architectural Erosion: An Initial Overview

As the first approach is to define, and explain the concept of architectural erosion in software engineering and to set the relationship between this concept and the mobile development ecosystem. In the first approach, we look for advances in static analysis solution approach in architecture quality gates in Server-Side and Frontend applications *reference paper uniandes*. In this case, we locate the cited related work in this research to search for the initial motivation for solving architectural erosion issues and find quality gates in terms of performance in the Android ecosystem.

The first overview we extracted about the related in Ruiyin et al. [9]. In this paper, we find

a Systematic Literature Review (SLR) that defines the definition, reasons, symptoms, consequences, and solution approaches to architectural erosion. In the resume, this approach finds 73 relevant papers about 8 research questions related to the last mentioned features. From this approach, we can execute the same query in different research papers databases for more actual research, due to the publication year of this research (2021), and the time period criteria (between 2006 and 2009) for finding recent research and advances during the last three years. The paper searches relevant papers in 7 research databases and makes a better-performed query, due to the relation between the "architectural erosion" concept in civil engineering and a phenomenon presented in buildings. We present the first used search queries and its databases:

Query
("software" OR "software system" OR "software engineering") AND ("architecture" OR "architectural structure" OR "structural") AND ("erosion" OR "decay" OR "degradation" OR "deterioration" OR "degeneration")

Table 3.1: Executed Query for related work search

Since those first results, we can extract relevant papers that include new developments and approaches for fixing architectural erosion issues, metrics, tools, and more related work. With this, we identify in each paper three main stages for solving those issues. In the detection stage, we use different identification alternatives through developer messages in versioning systems like GitHub. The Detection Stage, where we use Model Driven Development (MDD) and code patterns detection for detecting the identified architectural violations rules. Finally, based on the solution approach (Design approach, quality approach, etc.), we can suggest a proposal for solving the detected architectural violations. In the first search iteration, we added another filter related to the publication year. We selected the papers whose publication year is between 2021 and 2024, that papers include in their bibliography the first SLR that use as reference research. Despite different troubles with the query, when different databases show papers related to civil engineering or architecture (this is because the architectural erosion definition in that context is another research topic about building degradation). In this first research iteration,

we found researches that synthesize different solution approaches and give an overview from different perspectives, since the process of AER issues detection, metrics that could indicate a possible architectural violation in a software project like coupling metrics and the relationship between classes. We observed that the use of NLP techniques could improve the performance of AER issues detection based on commit messages analysis. With the use of pre-trained Word Embedding models trained in a specific context, different AER issues are detected, and a list of potential keywords could be generated as a type of alert of an architectural violation based on the architectural model base. In the symptoms and causes approach, an analysis of different base applications made in different programming languages was made. Different alternatives and methodologies were evaluated to determine the performance in architectural erosion issues detection and different solution approaches were proposed.

3.2. Architectural Erosion Symptoms Identification

In the stage of detection, the latest researches give feedback about identified architectural erosion symptoms and their types during different project stages. However, the detected and named symptoms are oriented to Frontend and Backend development. For mobile development oriented to Android technology, specifically made in Kotlin programming language, it doesn't exist a repository of possible architectural erosion symptoms. The most recent researches use NLP techniques based on GitHub commits. The researchers analyze the main keywords written in GitHub commits that could indicate a bad architectural issue implementation, identifying a possible architectural erosion issue. With a large amount of data from Git repositories is possible to make an NLP analysis of Github commits and define metrics that identify (from previously selected commits tagged by expert judges in software engineering and software architecture) similar keywords for architectural erosion. The metrics performance could be affected by the word embedding training context. The most recent research that implements that identification methodology, uses a Word Embedding model trained with 10 million of Stack overflow posts, a context that uses technical definitions for defining features, bugs, recommendations, and more of different programming languages. With this context, word embeddings could be more powerful and efficient for detecting architectural erosion issues keywords [6, 3]. Another approach is similar

to the one mentioned previously. However, those approaches are only based on architectural conformance checking. That consists of a set of different statements for a couple of judges who are professionals in software development and software architecture

3.3. Metrics that could indicate Architectural Erosion

Due to recent research, it is possible to determine the difference in metrics between an implemented architecture and an intended architecture, the main reason for the generation of the phenomenon of architectural erosion. In the resume, there are around 60 detected metrics in the software development process and in applications' source code analysis that could affect the maintainability of a software project. Research selected large software projects written in traditional programming languages like Python and Java and, with a set of judges, selected different source code implementations and past papers that try to describe and identify with similar methodologies. However, those metrics were detected in other development stacks like Frontend development and Backend development developed with traditional languages and frameworks. However, in the Android development context, the defined metrics could vary according to the mobile software development process for different reasons, like the recommended architectures for Android development and the mobile software development stages. For this reason, with the reviewed data for this research, we need to find different patterns that would indicate an architectural erosion issue and create a relationship between the reliability metrics found in recent research with the architectural erosion issues found in Android applications source code. In this research, it was employed different papers that found a set of metrics in different analysis code platforms that use different methodologies, mainly static analysis code techniques. The research consists of a Systematic Literature Review (SLR) that found different studies from different research databases. The study found 43 relevant papers for architectural erosion metrics definition. Founded metrics are defined with different criteria statements like historical data revision, architectural complexity, architectural dependency coupling analysis, or architecture size analysis. These metrics were found from open-source software development projects to industrial software development projects. Different measurement strategies established the effectiveness of the collected metrics. These strategies were found from different code analysis tools like SonarQube,

CKJM, etc. Most of the detected issues are mapped with any nonfunctional requirement inside a software development project, but most of them are related to maintainability architectural issues and customized quality gates determined by architectural deterioration or evolution.

Furthermore, there are different metrics and reasons to handle good programming practices with different architectural standards. Based on the main quality attributes in software architecture, it is possible to determine control metrics that could identify improvements with the suppression of identified AER issues in the source code of an Android application. In the availability quality attribute, one of the most common problems during the development stage of a software project is the bad implementation of exception handling. A bad exception handling in a software project could affect a complete flow that achieves a functional requirement and could generate catastrophic software failures and a possibility of a crash of an application. Despite this, some tools use custom lint check rules to detect common bad implementations of bad exception handling in software development. However, the tools spent considerable machine resources like computer processing time and RAM spent. [**?**]

Additionally, the latency and scalability quality attributes have been implemented with asynchronous programming techniques for better performance in the Android operating system. The recommended architecture standards and guidelines give politics about the inappropriate use of blocking library functions for data or behavior change operations. The use of synchronous programming techniques in any layer based on the MVVM architecture could affect the use of resources of Android devices. For that, it is necessary to implement the use of coroutines techniques and avoid synchronous operation in mobile application development. In the detection of AER issues stage, it will be very important for testing AER rules and detecting blocking functions in Android source implementations [**?**].

3.4. Giving Architectural Erosion Solutions

The latest researches give feedback about identified architectural erosion symptoms and their types during different realization project stages. However, the detected and named symptoms are oriented to Frontend and Backend development. For mobile development oriented to Android technology, specifically made in Kotlin programming language, it doesn't exist a repository of

possible architectural erosion symptoms. To get this, we use an artificial intelligence approach for detecting architectural changes, and those changes and their change messages (commits in git world) are useful for identifying possible symptoms and generating rules to prevent them [**?**].

3.4.1. Static Analysis Code techniques

Architectural erosion is a general phenomenon that occurs in all the fields in the software development process. Recent research for architectural erosion identification and detection has been focused on Backend development and Frontend development. For this, different static analysis tools have been created as plugins of different (Integrated Development Environment) IDEs. One example of that is the use of Antlr-determined grammars in the Eclipse plugin for bad pattern detection in (Data Transaction Object) DTO files. Those files on Backed development are used for service exposition and communication with other components inside a software project. There are similar components that use static analysis code techniques used in the industry. The most common platforms are SonarQube and different linters with custom check lint rules.

With static analysis code techniques, different advantages exist for architectural erosion issue detection. One of them is the easiness of detecting patterns in terms of dependency classes, coupling components, class name standards, and package name standards. Furthermore, we can extend that detection approach to detect customized architectural rules for a specific software development process.[**?**]

However, in the Android software development context, no platform considers the architectural erosion detection and identification process with architectural erosion metrics or customized quality gates. For this reason, it is necessary to define an identification process to detect different parents in different components. The pattern detection process must be based on a specific standard or specific recommended architectural pattern for Android application development. With the main concepts of programming languages, it is possible to define custom lint check rules with different tools like linters and IDE plugins. Static analysis code technique could help mobile developers to find architectural violations to defined nonfunctional requirements in the development stage.

3.4.2. NLP techniques and AI Models

For the architectural erosion identification process (and detection, but mainly AER identification process), different tools could be useful for architectural violation detection through Natural Language processing fundamentals. As an additional feature to that field, is possible to use different AI models powered by different training and contextualization techniques for getting a better performance in the AER issues detection process. Furthermore, the models present different ways to generate corrected code according to a detected issue in the AI training stage.

For the AER detection process, different AI models set have been implemented for the AER detection process based on developers' messages extracted from a code versioning platform like GitHub, GitLab, or OpenStack After a large identification process based on human judgments. With this identified AER issues dataset, basic AI models have been proved for potential key detection that could indicate architectural violations.

Chapter 4

Finding Architectural Erosion Issues

The recent research about architectural erosion identification followed the same process of detection of potential issues in natural language in the commit messages from different version platforms. Furthermore, the study used different NLP techniques like the use of static word embedding and word similarity metrics, for identifying potential words that have considerable semantic meaning in the development history of any application. With this base process, a list of potential keywords that could indicate an implemented bad architectural issue. However, the detection and selection process of different keywords were developed with large software Backend and Frontend software projects. Those projects were implemented with traditional programming languages like Python and Javascript. Those programming languages' documentation and support are greater than those languages oriented for mobile development. For this reason, it is necessary to adapt that identification methodology for the identification of potential words that have a semantic meaning inside commit messages inserted in a versioning platform for Android applications based on their source code. For this purpose, we extracted from 50 open-source Android applications and applied scraping techniques for GitHub history commit extraction. We can collect the committed messages and find similarities with the defined keywords in the research mentioned. A new version of the list of keywords will increase the accuracy of future selection and detection processes for Architectural erosion issues in the source code of Android applications.

4.1. Architectural Erosion Identification Methodology

4.1.1. Appplications Selection

Based on the methodology of the mentioned research, we designed a similar process of extraction and selection process for an initial version of the commits analysis. In this case, we explore different options and platforms for searching and selecting open-source Android projects whose source code is located on the GitHub versioning platform. We use two tools. The first one is SeART, a platform based on Data Mining repositories research [**?**]. This platform has different filter features that make it possible to customize a search of different GitHub open-source projects. The customized filter for this case is to select GitHub repositories whose main programming language is Kotlin, and the number inside each repository's commit history is in the range of 1000 commits and 30000 commits. This customized filter was selected due to the opportunity to collect a large number of commits for having a strong dataset to identify potential similar keywords with a high similarity metric. With this customized filter, we select the GitHub repositories with the longest number of commits. However, the Data Mining repositories techniques used are not the most effective to consider all the GitHub repository environments. For this reason, it is necessary to consider more tools for searching more GitHub repositories of open-source Android applications.

For a larger collection of Android applications for getting a better performance in AER identification and selection, we consider using an Android applications searching tool. The mentioned tool is F-Droid- F-Droid is an open-source platform for searching open-source Android applications. This platform gives specific details of every Android application, and one of those details is the GitHub repository URL. With that attribute and the type of license declared in the F-Droid platform.

Once the applications are selected, we implemented a program based on web scraping techniques with the PyDriller library and built a dataset with the main features of each commit in the GitHub repository of the source code of each Android application.

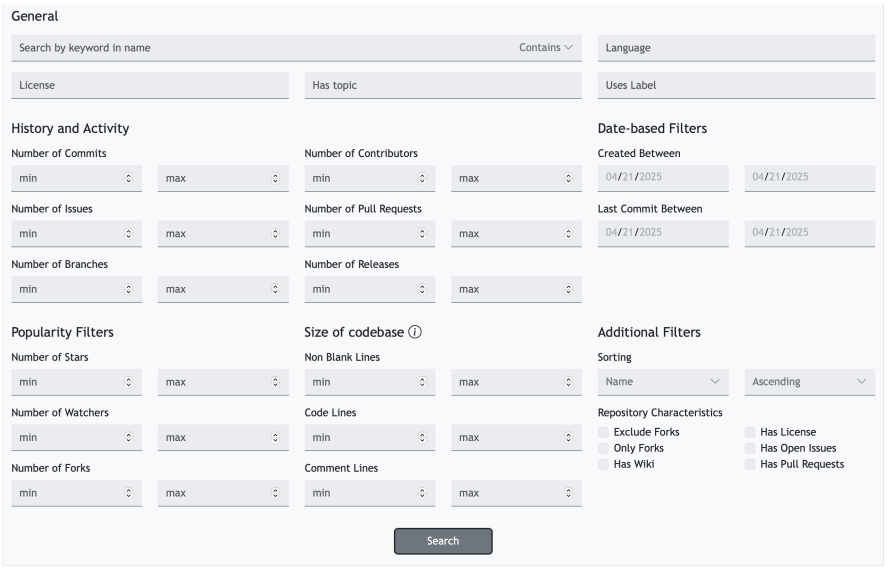

Figure 4.1: seART platform for GitHub repositories searching citet

4.1.2. Word Embedding and Similarity Criteria

With the collected set of commits from the mentioned Android apps collection, we made a preprocessing process for the developers' messages of those commits. In this case, we implemented a preprocessing flow with stemming, lemmatizing, and stop word removal. After that, we used the static pre-trained Word Embedding model, trained on 2 million Stack Overflow posts. With that model, we extracted the numerical representation of each word in the vocabulary of the GitHub commits dataset of selected Android applications. With the cosine similarity metric, we extracted the most similar words based on the keywords found in the related work.

4.2. Architectural Erosion Keywords in Commits

4.2.1. Architectural erosion symptoms Identification

Based on the last research, architectural erosion insights identification consists of a deep analysis of information based on development judgments. This judgment can be extracted from different software versioning systems, like OpenStack, a software versioning platform for large-scale software projects, or GitHub, the most used software versioning platform. From those messages, we

can classify and tag different code changes through code differences between code changes over time. In this process, the use of different Natural Language Processing (NLP) techniques, to find a standard architectural erosion commit definition. One innovative idea is to employ pre-trained Word Embeddings in software development contexts for potential words that could indicate an implemented architectural violation in server-side applications. The use of embeddings and word similarity metrics like Cosine Similarity allows us to calculate the similarity between the word embedding's numerical representation for every word, since semantic function. With this study, we can standardize the main cases of architectural erosion, the different metrics that could be identified in a software project, and potential solution approaches. With those NLP metrics, we can use them to create discriminatory models for issue detection. However, in an Android context, the architectural erosion identification hasn't been enough to standardize a set of rules for detecting it in software source code due to the Backend and Frontend solution approaches implemented in the study software projects (the two main projects are developed in Python). With the results of the papers extracted in the first related research overview, we can find a set of keywords extracted for the developer's code messages that indicates a potential architectural erosion issue in the implementation, that study was realized with the developer's judges and messages extracted from different version platforms like OpenStack (previously mentioned) and Github, the most popular and used versioning platform. In this case, the project had as a reference four large open-source software projects; all applications are server-side applications. Around 50 keywords were mentioned as potential keywords that indicate an air issue. However, the words were extracted for Backend development purposes. For this reason, we use the same word extraction approach in a mobile context. During this process, we extracted from 50 open-source Android applications published around 470k GitHub commits. This is enough detail to make a preprocessing of vocabulary implemented in each GitHub commit and create new rules that could be implemented with custom lint check rules. That topic will be treated in the next chapter.

Extracting keywords from Android Context

Based on the last mentioned research, it is possible to find potential keywords that indicate an issue or an insight inside a code implementation, with the help of NLP techniques, through

similarity measurements like cosine similarity (mentioned in the definitions chapter) and pre-trained Word Embeddings, due to the numerical representation of each word of the generated vocabulary in a specific context. First, we use the PyDriller library *url PyDriller*, a useful library for repository mining, for getting code source and its attributes of different open-source Android projects made in Kotlin. The selected projects were extracted from different open-source Android project catalogs like F-Droid and other data mining repositories found with different filters like development programming language used, number of commits, and keywords in the selection criteria *url fdroidurl search*. In the first overview, we extracted 50 Android projects that have around 470K commits. With these commits, we made a text pre-processing to build a standard vocabulary and tokenize with the help of NLTK library *url nltk*, a library for making NLP operations like tokenization, lemmatization, and stemming.

Keyword
architecture, architectural, structure, structural, layer, design, violate, violation, deviate, deviation, inconsistency, inconsistent, consistent, mismatch, diverge, divergence, divergent, deviate, deviation, architecture, layering, layered, designed, violates, violating, violated, diverges, designing, diverged, diverging, deviates, deviated, deviating, inconsistencies, non-consistent, discrepancy, deviations, modular, module, modularity, encapsulation, encapsulate, encapsulating, encapsulated, intend, intends, intended, intent, intents, implemented, implement, implementation, as-planned, as-implemented, blueprint, blueprints, mis-match, mismatched, mismatches, mismatching

Table 4.1: List of initial Keywords extracted of mentioned related work

Column	Description
Name Repo	Name of the GitHub repository of Android project source code
Url Repo	GitHub URL from source code repository
Commit Message	Message of a specific commit in GitHub commits history of each Android project
Commit Hash	Hash from GitHub commit, essential for commits analysis process
File Name	List of GiHub commit modified file names
Code Changes	String with the modified source code of each GitHub commit

Table 4.2: Features of commits dataset and their description

With all the corpus from GitHub commits, we implement a text cleaning process, removing stop words and performing stemming, due to the lemmatization process in a technical context is not very effective; some words do not have their respective lexical root, so the tokenizer would not consider those words. The stemming process is useful for semantic word derivation control. This process is essential because a lot of words do not have semantic relevance in each GitHub commit, so consider that words could affect similarity metrics. With the processed words, we use a pre-trained Word Embedding based on millions of Stack Overflow posts. With the Gensim library *librería de gensim*, we can load the Word Embedding model, get a numerical representation of each selected word, and use the cosine similarity metric for find similar words from the previous keywords. When the metric is generated, we select the 10 most similar words based on that metric. In the first overview implementation, we found around 5000 relevant commits with the updated keywords list. For efficient selection criteria, we extract a representative subset based on a weighted average made by the word frequency in the corpus text. With the first selection of a representative set, we extracted 357 GitHub commits.

Word	Cosine Similitude Average Value
notion	0.2674
respect	0.2627
formal	0.2482
high-level	0.2437
tend	0.2342
rigid	0.2315
kind	0.2256
stronger	0.2243
non-linear	0.2227
sane	0.2198

Table 4.3: Top 10 newfound words since Word Embedding cosine similitude metric

With this approach, it is possible to find potential words written in a development context that could indicate a potential issue related to different kinds of functional and nonfunctional requirements that a software project includes in its architectural design and its standards. This detection approach has many different development areas to detect different problems found in a software project. The future work related with this approach will be discussed in next chapters.

Chapter 5

Detecting Architectural Erosion Issues

With a set of possible causes presented in Kotlin source code files from different Android projects, it is possible to define a set of rules with the help of the Android Studio IDE tools ecosystem for Architectural Erosion issues detection. We detected architectural changes in the representative set of commits and extracted different rules and patterns that could handle an architectural violation inside the MVVM architectural pattern for Android Applications. Rules could be implemented in any IDE that supports mobile development oriented to Android, like, for example, Visual Studio Code or Sublime Text (despite its constraints).

5.1. UAST in Android Studio

The Android Studio Integrated Development Environment (IDE) is used mostly for mobile development and Android application building. Inside its ecosystem, there are a lot of built libraries, frameworks, and other programs that make the development process easier and get better performance in terms of the different non-functional requirements and architectural standards. In addition to those programs, other tools avoid committing any bad practice implementation in terms of different standards based on a specific architectural style. This tool is named linter. A linter is an integrated tool with an IDE for code insight detection and correction (if it is possible). When we write code in Android Studio with bad code practice, its interface shows and marks a possible insight into our code. After that, we select that marker and give a possible solution for that code insight. Every rule shows its name, its description, a possible reason for the insight,

and an optional code implementation solution.

This linter implementation, integrated with the Android Studio IDE, is due to the UAST structure generated by that tool. The Unified Abstract Syntax Tree (UAST) is a defined structure generated by each software project opened by the IDE that contains the AST and the Call graph structures of the source code files of that project. With this structure, we can create different semantical components (similar to the visitors components mentioned in the definitions section) that detect any pattern in different structures and code fragments inside the source code files. We can instantiate the different structures based on the Kotlin programming language grammar since there are clauses to classes and other ones. Furthermore, we can access the different attributes of each structure, like name, source code file name, or package name, and their relationships with other components. In terms of rules, we can define any customized lint check rule, and with that structure's set of Kotlin programming language and its attributes, it is possible to detect any pattern and offer any solution suggestion.

Based on the standard rules implemented in the Android Studio IDE, we can similarly define custom lint check rules for aer detection based on the insights reviewed in the representative set of commits of open-source Android projects.

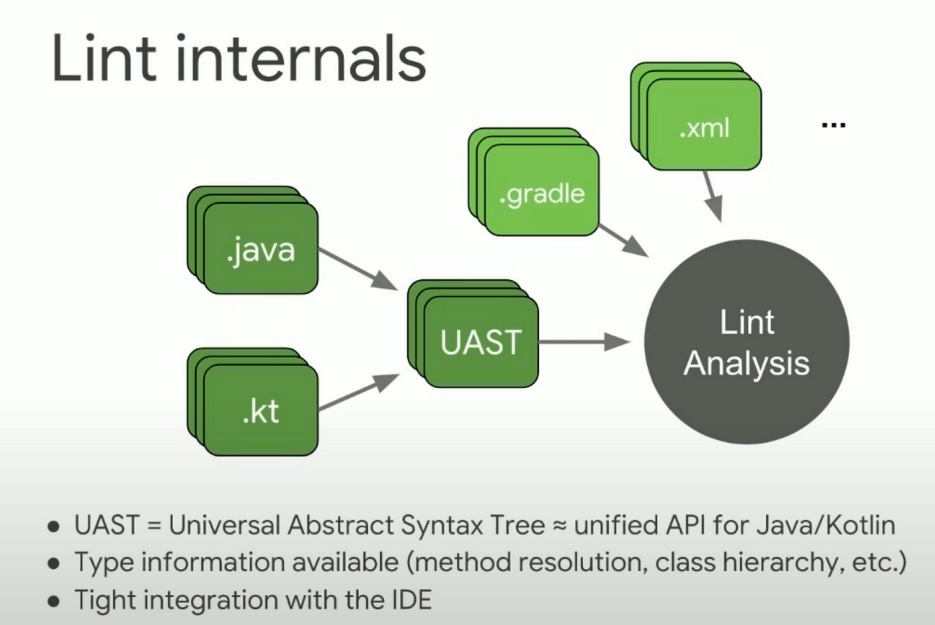

Figure 5.1: Definition of Android Studio UAST structure]

Inside the UAST structure in Android Studio, we can easily implement visitors of the abstract syntax tree. With that representation of the structure of the source code of an Android project, we can instantiate each one of them to analyze it in terms of patterns in its name, its structure, its declaration, and other features that could indicate an architectural erosion issue implemented in its structure.

5.2. Rules Definition

With the representative set extracted from the architectural erosion issues identification process, it is possible to analyze code fragments where the two judges confirmed that it could be a possible architectural violation of the original GitHub source code repository. With that, GitHub commits set and the Google architectural guidelines based on the MVVM architectural pattern], it is possible to find troubles in architectural changes. With these architectural erosion issues, we can infer different patterns in terms of class naming, method invocation, dependency injection, and other features that could mainly affect the application performance and other significant architectural requirements inside an Android project. In the table $table_rules$, we present the different found rule sets implemented in the Kotlin programming language whose implementation could indicate an architectural erosion issue inside an Android project.

5.3. AER Detector Implementation

To implement the mentioned architectural erosion detection rules for Android applications implemented in the Kotlin programming language, we need to use the lint API toolset given by the Android Studio IDE. It is necessary to understand the tool and how we can implement its functionalities inside any Android project, in terms of version compatibility of different implemented libraries versions and automatic dependencies injection tool versions like the Gradle plugin.

Firstly, we need to create an empty Android project. After that, we create a module that will be the main component for the custom lint check rules. We inject the main libraries to implement the linter's functionality into the project, in this case, the lint API libraries. When the project base has been implemented, we need to declare each one as an issue for the linter.

Each architectural erosion rule must be implemented by stating an issue object, where we define the architectural erosion rule issue and its main attributes like name, description, category (in terms of static analysis tools like lint rules, scope rules, and other ones) severity (if the rule could indicate a severity issue, a warning issue or another category). When we declare the issue object for each rule, we need to add a visitor component that acts as a detector component to each one. To create a visitor component, we need to inject the structures that we should analyze in each pattern of each architectural erosion rule. As an example, for a bad implementation of error handling in different components and layers of an Android application, we need to analyze the try-catch structures. With UAST libraries from Android Studio, we can inject the UTryCatchStatement, which instantiates the try-catch structures found in the Android source code repository. Furthermore, we can add different filters, of which try-catch statements, we can select by name and code block structure filters.

With the base components of each architectural erosion rule, we can create a custom lint registry template for setting the custom lint check rules inside any Android project. In this file, we create the architectural erosion rules issues based on the previously created issues. After that, we implement that component as an Android library. By this, we need to inject the .jar generated file into the Android application source code; this is possible by making the .jar libraries accepted by the application in the Gradle plugin configuration file. After that, we need to use an XML file, setting the specific name of every custom lint check rule.

With the architectural erosion rules, component creation, and the Gradle and environment parameter configuration. We execute the analysis by calling the functionality of Maven named lint. With this console command $maven --lintargs$, we execute the lint rules since the default implemented lint check rules and the custom lint check rules are declared in the AER component.

5.4. AER linter tool testing criteria

Once we have the AER component with all the implemented architectural erosion rules and the parameters configuration for compatibility of each Android application, we select specific criteria for testing the implemented custom lint check rules based on the analyzed open source applications and their commit set. With the hash commit attribute, we can select a specific

instant of that implementation inside the Android project source code repository and observe the custom lint check rules' effectiveness. Due to the different versions of all library ecosystems in Android development, we need to configure the test apps to set as environment variables the Java version compatibility and the Gradle plugin compatibility of the AER plugin. With these observations, we need to select specific test apps and set the environment variables for a correct lint process of each test application. Furthermore

5.4.1. Test apps selection

With the detection process previously made,

5.4.2. Building Configuration of AER Component

Chapter A

TITLE

Bibliography

The references are sorted alphabetically by first author.

[1] A. Baabad, H. Zulzalil, S. Hassan, and S Baharom. Characterizing the architectural erosion metrics: A systematic mapping study. *IEEE*, 2022.

[2] L. Bass, P. Clements, and R Kazman. *Software Architecture in Practice, 4th Edition*. O'reilly, 2021.

[3] V. Efstathiou, Chatzilenas C., and D Spinellis. Word embeddings for the software engineering domain. *IEEE*, 2018.

[4] J Jurafsky, D. Martin. *Speech ad Language Processing. An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models*. Stanford Web, 2024.

[5] M Laplante, P. Kassab. *Requirements Engineering for Software and Systems. 4th Edition*. O'reilly, 2022.

[6] P. Liang, P. Avgeriou, and L Ruiyin. Warnings: Violation symptoms indicating architecture erosion. *Cornell University*, 2023.

[7] P. Schutze H. Manning, C. Raghavan. *An Introduction to Information Retrieval*. Cambridge University Press, 2009.

[8] Alexander L Perry, Dewayne. Wolf. Foundations for the stufy of sotware architecture. *ACM SIGSOFT*, 1992.

[9] L. Ruiyin, L. Peng, P. Avgeriou, and M Soliman. Understanding software architecture erosion: A systematic mapping study. *Wiley*, 2021.

Acronyms