Tools for improvement of code quality — A systematic review

Marius Kohmann

September 2019

Abstract

This paper takes a systematic approach to find areas of improvement for tools for static code analysis. By looking at 30 static analysis tools for Java, a classification of domains for static analysis has been created and suggestions for future research has been found. Most notably, an unexplored category of tools which uses design principles to help developers take good decisions has been found. In addition this paper proposes more research into topics System Architecture Reconstruction (SAR), Architectural Pattern Detection (APD), Design Pattern Detection (DPD), anti-patterns, design principles and best practices with respect to testability.

Contents

1 Introduction

Writing maintainable and high quality code is an important part of software engineering. Ignoring quality enhancing tasks like refactoring[1] and testing will introduce technical debt[2], that will have negative impact on further development. Some consequences include increased development time, inaccurate estimations causing lost deadlines and higher costs of introducing new developers to the project.

Various people define software maintainability differently, but a commonly accepted collection of quality attributes include extensibility, modularity, testability and understandability. For developers it is a tough challenge to fulfil all of these aspects and therefore a lot of tools for code analysis have been developed. By using code analysis one is able to detect and resolve problems, before the problems arise in a production environment. In this paper we will look into the tools for analysing the source code, also known as tools for static code analysis.

The tools have many forms ranging from Command Line Interface (CLI)s, Integrated Development Environment (IDE)s, IDE-plugins to online services and native applications. Some tools are multilingual, supporting a wide range of languages, while others are specific for a certain programming language or framework. Some tools are broad, targeting many quality attributes while others have a limited area of focus.

With the ever increasing number of tools it is difficult to get an overview of what type of tools that exists, which areas they cover, and what they can't do. With the intention of contributing to the area of tools for code analysis, the first step will be to identify areas for contribution. I will therefore in this paper do a systematic review of existing tools and their capabilities, categorize them and find out where the tools can help us and where we still need further development. This analysis of tools will take a general approach, looking at shortcomings on the area of code analysis, rather than shortcomings of individual tools.

The outline of the paper is as follows; Section 2 gives some background information and an introduction to the topics of writing maintainable software and code analysis. Section 3 describes the methodology of the research process. Section 4 gives some insight in related work in the area and section 5 presents the results. Section 6 discusses the results, and lastly in section 7 we conclude the paper.

2 Background

To be able to identify shortcomings of tools for code analysis, we need to look at what software quality is and how developers can write software of high quality, more specifically that the code is *maintainable*. The following sections will

[1]From Wikipedia: Process of restructuring existing code without changing its external behavior [7].

[2]From Wikipedia: Concept in software development that reflects the implied cost of additional rework caused by choosing an easy (limited) solution now instead of using a better approach that would take longer [33].

briefly mention a list of commonly accepted quality attributes of maintainable code and ways of adhering to those attributes.

2.1 What is software quality?

The software community is highly opinionated and software quality is measured differently based on (but not limited to) domain, programming language and business requirements. Therefore, measuring software quality and creating rules without exceptions is extremely hard. However, what we know is that the ratio of time spent reading versus writing code is well over 10 to 1 as Robert C. Martin states in [22].

To reduce the amount of time spent on reading code (i.e understanding code), we need to ensure that the written code is understandable. We need to ensure that it is easy to understand what the code does and why it does what it does. It should be easy to locate what needs to change, easy to make changes and easy to ensure that the changes does not create unwanted side effects.

More formally, developers have defined a set of quality attributes that will help ensure that the code is of high quality. A commonly accepted collection of quality attributes include extensibility, modularity, testability, understandability, performance, reliability and security. Martin Fowler did a useful distinction using the terms *internal attributes* and *external attributes* [16]. The distinction is whether the attribute is visible for the user or customer. The internal quality attributes correspond to maintainability, that is our focus.

Following are the definitions of the internal quality attributes with most importance in this paper:

- Extensibility - "Extensibility is a measure of the ability to extend a system and the level of effort required to implement the extension. Extensions can be through the addition of new functionality or through modification of existing functionality. The principle provides for enhancements without impairing existing system functions." [12]

- Modularity - "Modular programming is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality." [24]

- Testability - "Software testability is the degree to which a software artifact supports testing in a given test context. If the testability of the software artifact is high, then finding faults in the system (if it has any) by means of testing is easier." [30]

- Understandability - "Understandability is defined as the attributes of software that bear on the users' (programmer) efforts for recognizing the logical concept and its applicability." [36]

2.2 Achieving maintainable code

To write code that is maintainable a set of concepts, principles and conventions including; Architectural patterns, design patterns, anti-patterns, design principles, metrics and best practices is used amongst developers. Some of them are well defined, and can easily be verified through source code analysis. Others are more abstract in nature and requires reasoning from the developers and is harder to verify.

An *architectural pattern* is a general, reusable solution to a commonly occurring problem in software architecture within a given context [27]. An example is the Model-View-ViewModel (MVVM)-pattern for mobile development [23]. It is a well defined pattern and correct use or misuse could be verified through testing tools like ArchUnit [37].

A *design pattern* is similar to an architectural pattern, but more limited in scope. An example is the Adapter pattern [1]. Detection of design patterns is possible through mining [35]. The absence of patterns is harder to detect as the absence of a design pattern is not clearly defined.

Definitions of *architectural anti-patterns* and *design anti-patterns* have also been made. They are the exact opposite of architectural-patterns and design-patterns. In other words ways one should not solve a common problem. They are often called architecture-smells and design-smells. An example of architectural anti-pattern is the Cyclic Dependency [5] and could be detected through dependency analysis. An example of design anti-pattern is the God-Object [13], and is as stated about design patterns, not easily verifiable. However, metrics such as a high value of coupling and Lines Of Code (LOC) could imply possible violations.

Design principles are a set of guidelines that programmers should follow to avoid bad design. According to Robert C. Martin [21] there are three characteristics of bad design that the design principles will help reduce:

1. Rigidity - It is hard to change because every change affects too many other parts of the system.
2. Fragility - When you make a change, unexpected parts of the system break.
3. Immobility - It is hard to reuse in another application because it cannot be disentangled from the current application.

A common set of design principles that often is referred to is the **S**ingle responsibility, **O**pen–closed, **L**iskov substitution, **I**nterface segregation, **D**ependency inversion (SOLID) principles [31] and the Don't Repeat Yourself (DRY) principle. The design principles are often abstract and verification requires knowledge and reasoning about the business domain. They are therefore hard to verify using code analysis. One could easily think that DRY could be verified through looking at similar code, but it turns out that it is not that simple. Two code snippets could be exact copies of each other, but still not violating DRY if the two snippets would change for different reasons.

Metrics have been developed to measure how well one adheres to the design principles. Examples include cyclomatic complexity and coupling. *Coupling* is the degree of interdependence between software modules [9]. *Cyclomatic complexity* is used to indicate the complexity of a program [10]. They are easily calculated using code analysis.

Best practices are informal rules that have been learned over time, or practice that have become part of the language "culture". The best practices can in some ways be equal to the design principles, but are often simpler and more limited in scope. Even if limited in scope, the range of different best practices is huge. Best practices includes but is not limited to, code patterns that are probable bugs, styling of code and readability. An example of best-practice in the Java language could be to use camel case (camelCase) [4] on variable-names, or to not have empty else-blocks. They are well defined and are verified using code analysis tools like SonarQube [6].

2.3 Code analysis

To help developers of software systems adhere to these concepts and conventions tools for code analysis have been developed. In code analysis we differentiate between two types of code analysis, dynamic code analysis and static code analysis. *Dynamic code analysis* is done by analysing programs being executed on a processor, while *static code analysis* is purely based on analysis of the source code. Since dynamic analysis is based on program execution it has the advantage of being able to measure the actual CPU, memory and energy performance of the application. It can also gather information about the system that would be hard to detect using static code analysis. Examples include discovery of dynamic dependencies using reflection, or actual application usage. However, that does not mean that static code analysis is not able to target performance or dynamic aspects of source code.

As we search to improve the source code, we have chosen to focus on static code analysis. The static analysis is done by parsing the source code, creating an Abstract Syntax Tree (AST) and then analyzing the tree for violations of the aforementioned concepts and conventions. Figure 1 shows a simple example AST where a static analysis tools could detect that the expression `x == 1` always evaluates to `true` and that the variable `y` is never used. The tool could then suggest that the branching is unnecessary and that the `y` variable is removed.

3 Methodology

The theoretical foundation of the tools, programming languages, compilers and static analysis are heavily based on scientific research. However, the applications and the usage of these theoretical concepts have not been driven by researchers. The creation of tools have been driven by a community of programmers, wishing to improve their code, or by businesses that wants to provide a valuable service to other developers. Initial investigations suggested an approach focusing more

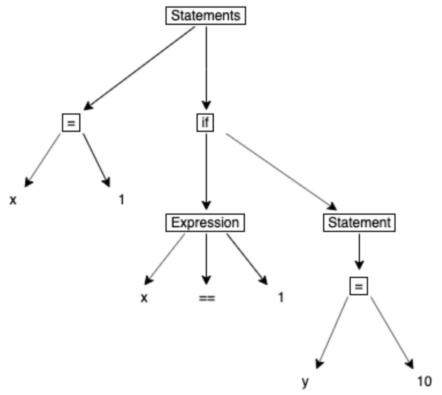


Figure 1: Simple AST of x=1; if (x == 1) {y = 10;}

on tools rather than literature, due to lack of literature on the subject. Therefore a traditional Systematic Literature Review (SLR) only studying literature is not suitable for this paper. However, the scientific method of an literature review is interesting and is very applicable for the practical domain of tools. I will therefore adapt this scientific method into the domain of tools and do a Systematic Tool Review (STR) that includes both tools and literature. Figure 2 is taken from [25] and shows the process of the SLR.

3.1 Goal and research questions

Since the target with this paper is to identify areas for contribution, the main objective of this preliminary study will be to provide a classification of the current tools for static analysis, and to provide suggestions in which areas further research needs to be done.

To know in which areas further research needs to be done, we need a classification of domains within static analysis with respect to the maintainability of code. Hereby these domains within static analysis are referred to as *maintainability domains*. The first goal is therefore to provide a classification of maintainability domains.

The second goal is to identify which of the maintainability domains needs most research to continue to improve the current state of the tools.

These two goals can be summarized as two research questions:

RQ1: Which maintainability domains exists?
RQ2: Which maintainability domains has weak support and high potential for further research?

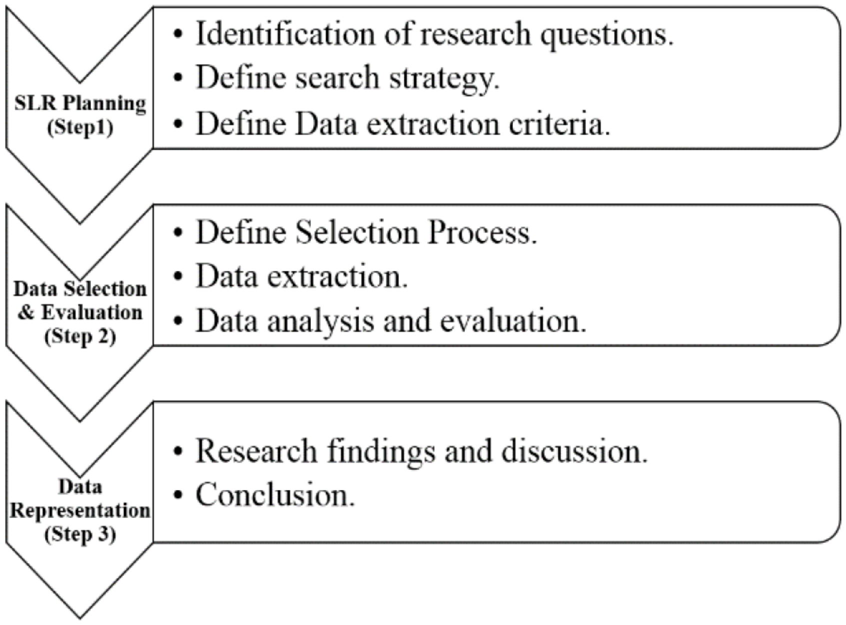

SLR Planning (Step1)
• Identification of research questions.
• Define search strategy.
• Define Data extraction criteria.
Data Selection & Evaluation (Step 2)
• Define Selection Process.
• Data extraction.
• Data analysis and evaluation.
Data Representation (Step 3)
• Research findings and discussion.
• Conclusion.

Figure 2: Steps in a systematic literature review

3.2 Research strategy

To answer **RQ1** a detailed insight in the current tools and their support to improve maintainability is needed. The first step in the process will therefore be to find all the tools that needs to be investigated using a defined search strategy (3.2.1). After the execution of the search, the tools needs to be filtered according to the list of inclusion criteria (3.2.2). With the final list of tools, a manual inspection and extraction of maintainability-domains needs to be done for each tool. This extraction needs to follow the defined domain-extraction strategy (3.2.3). The result will be a mapping from tool to related maintainability-domains. This mapping will be used to create the final classification of tools.

To answer **RQ2**, we have to see if all of the rules, concepts, principles and practices that have been developed are targeted with the current set of tools. A review of each will be done to find shortcomings, and a literature search will be done to see if it is an area that needs more research. Also, we need to consider if a combination of all the tools and their capabilities could provide more value into new tools.

3.2.1 Search strategy

To obtain a list of tools for analysis an open source project on GitHub were used [32]. This comprehensive list contains over 400 tools for static analysis.

Because there are too many tools to examine them all, we will have to limit the number of tools to include in our analysis. To provide a review of the current state of statical analysis, we want to keep the diversity of the tools high.

Initial thoughts suggested an approach sorting the tools on popularity. However it ended up with a selection of tools with low diversity, only giving the linters[3] in the most popular languages high score. Based on the observation that the most used languages should have the highest amount of tools, only looking at tools supporting Java is a good choice. According to the TIOBE-index seen in figure 3, Java has been the most popular programming language the latest 20 years [34]. Tool support for Java will therefore be our reference of the current state of the art.

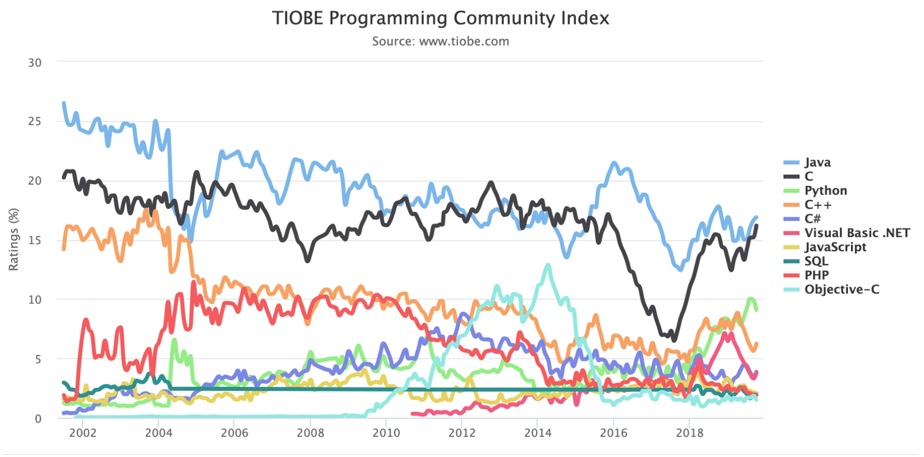


Figure 3: Most popular programming languages from 2002 - 2018

To obtain the list of relevant tools for analysis, the following sections in [32] will be used: Java, Multiple languages, IDE-plugin, Supporting Tools, Writing and Web services. In addition, the industry leading Java IDE will be considered, namely IntelliJ.

3.2.2 Data extraction criteria

To ensure that the tools are relevant for analysis a list of inclusion criteria have to be defined. The first inclusion criteria is that the tools relevant for this review needs to be based on static code analysis. This rules out tools that involves dynamic code analysis. The second inclusion criteria is that the tool supports Java source code. The third inclusion criteria is that the tool should directly affect code maintainability. It will therefore rule out tools that support source code analysis, but are designed for managing teams and indirectly improving

[3]From Wikipedia: A tool that analyzes source code to flag programming errors, bugs, stylistic errors, and suspicious constructs [20].

the development process. It will also rule out tools for model-checking as model-checking is more related to proving the correctness of programs than improving the maintainability. The fourth criteria will rule out tools that for various reasons are not available anymore, and thus could not be reviewed. The last criteria rules out tools that combines other static analysis tools without giving more value. The inclusion criteria are summarized in the table below.

Criteria identification	**Inclusion Criteria**
IC1	Based on static code analysis
IC2	Supports analysis of Java source code
IC3	Directly affects Java code maintainability
IC4	Project available for use
IC5	Is not a wrapper around other tools

3.2.3 Domain extraction strategy

To extract maintainability-domains from the static analysis tools this strategy needs to be followed: For each tool that is considered these steps needs to be answered:

1. Does the tool itself classify the domains of static-analysis? If so, which? If not, try to categorize the features based on the information that is available.
2. Is it possible to find any domains that the tool does not consider? Sub-domains that are not considered? These domains will be important to uncover as there may be the some maintainability-domains that is not considered by any tool.

When all the maintainability-domains have been extracted it is important to see if there is any similarities and if some extracted domains can be considered equal or synonyms. For example the maintainability-domains; error-prone and probable bugs can be considered the same thing.

4 Related work

As tools will be the main concern of this paper, and as stated in section 3, the literature on the actual tools are limited. And to the best of my knowledge no classification of tools for static analysis exists.

However two papers have been somewhat useful carrying out the classification of tools. Z. Mushtaq, G. Rasool and B. Shehzad provides a comprehensive literature review of multilingual source code analysis tools in [25]. They provide some valuable information about domains for analysis of multilingual applications that can be transferred to my use in analysis of single language applications. However the evaluation of the tools are not of much use in my analysis, as their focus is on multilingual analysis tools. Because the multilingual tools need to focus on a higher level of abstraction, their direct feedback on code quality is limited.

A classification of tools to support maintenance has been done by V. Lenarduzzi1, A. Sillitti and O. Taibi [19] and has been useful in classification of the tools in this paper. However, as their focus was on classifying the most popular tools used in industry, the classification is not broad enough to cover all the different domains of analysis i will consider in this paper.

5 Results

Following are the results of my classification and my suggestions for future research. Initially 108 tools were considered. After filtering using the inclusion criteria there were 30 left for domain extraction. The initial list of tools for review can be seen in A.1. The final list of tools after filtering can be seen in A.2. The extraction of features for all the tools can be seen in A.3.

5.1 List of tools and maintainability-domains

By following the domain-extraction strategy, a lot of similarities in features of the tools appeared. By selecting the most frequently appearing domains the first level of the classification were created. Every tool fits into one or more of these first-level categories.

The category *structure analysis* contains tools that analyzes projects at a high level. They often uses code metrics and different forms of dependency-analysis to provide insights and visualizations about the software.

The *style* category contains tools that enforces use of styling conventions ranging from source file structure to programming practices and formatting.

The *metrics* category contains all the tools that in some way calculates metrics, either as data for creating visualizations of software, or as checks that is meant to notify the developer if the value is reaching a certain threshold.

The *probable bugs* category contains all the tools that checks for common sources of bugs.

Each sub-category in the classification was found looking at the features of the tool or by finding un-discovered domains. While extracting maintainability-domains i found that the domains **Probable bugs** and **Style** were covering a huge amount of topics and is heavily developed. As i search to find un-developed areas for code-analysis the focus on mapping out all the domains within these areas have been limited and therefore only a few is shown.

- Structure analysis
  - Architectural patterns
    - Architecture reconstruction (SAR)
    - Compliance (SACC)

        - Detection
            - Patterns
            - Anti-Patterns
    - Design patterns
        - Verification
        - Detection (DPD)
            - Patterns
            - Anti-Patterns
    - Design principles
    - Visualization
        - Technical debt
        - System metrics
        - Dependency
            - Dependency-matrix
            - Dependency-graph
        - Tree-map
        - Call-graphs
        - Coupling graph
        - Path graph
        - All paths graph
        - Cycle graph
        - Class inheritance
    - Querying
    - Duplicate code detection
    - Testability
    - Extensibility
    - Modularity

- Style
    - Conventions
        - Encapsulation
        - Unused constructs
        - Redundant constructs
        - Structure
        - Confusing code
        - Using idioms
        - ...
    - Readability

        - Formatting
        - Spellchecking

- Metrics
    - Project
    - Class
    - Method
- Probable bugs
    - Inspection of source code
        - Concurrency
        - Internationalization
        - Null analysis
        - Initialization
        - Index
        - ...
    - Using pluggable type-systems (Annotations)
    - Wrong usage of API

A table of all the tools and their domains can be seen in figure 4. It is important to notice that for each tool, only the extra functionality that the tool is providing is added to the matrix - possible integrations with other tools are ignored.

Figure 5 is showing the distribution on the domains with most focus.

The bar for structure analysis can be misleading as many tools include some form of structural analysis, but very limited in scope. This means that the actual number of tools that does comprehensive structure analysis is lower than what can be seen from the graph.

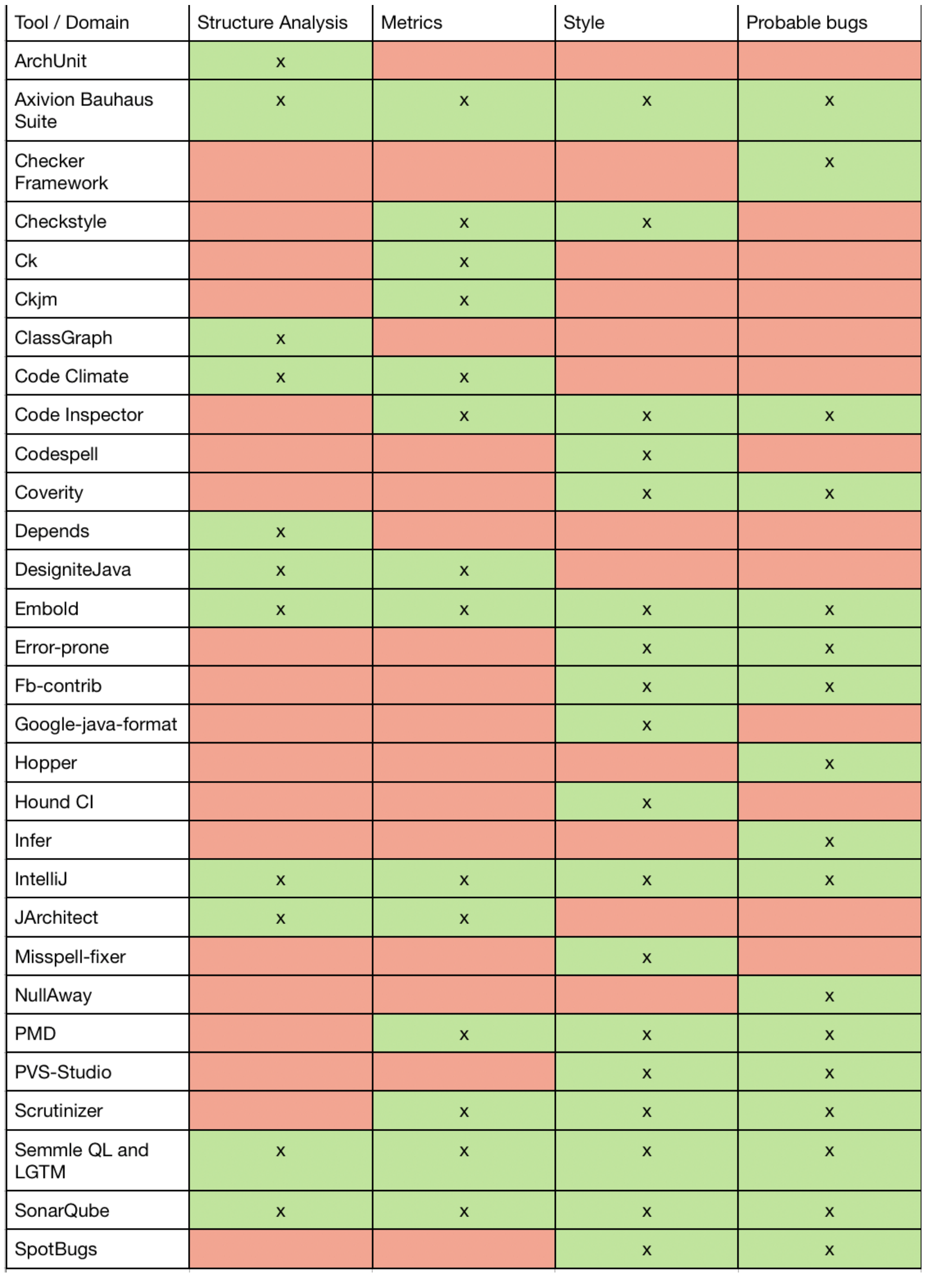
5.2 Suggestions for future research

The domains found in the classification above correspond to the concepts, principles and conventions mentioned in section 2.2. For each of these, an outlook into future research is proposed.

5.2.1 Architectural patterns

Architectural conformance-tools would enforce proper use of architectural patterns and could create automated documentation that would be helpful in program and structure comprehension activities.

The investigated tools include features for Software Architecture Compliance Checking (SACC) by testing of architecture, and querying of code. This approach suits all possible architectures, but the downside is that effort in writing



Tool / Domain	Structure Analysis	Metrics	Style	Probable bugs
ArchUnit	x			
Axivion Bauhaus Suite	x	x	x	x
Checker Framework				x
Checkstyle		x	x	
Ck		x		
Ckjm		x		
ClassGraph	x			
Code Climate	x	x		
Code Inspector		x	x	x
Codespell			x	
Coverity			x	x
Depends	x			
DesigniteJava	x	x		
Embold	x	x	x	x
Error-prone			x	x
Fb-contrib			x	x
Google-java-format			x	
Hopper				x
Hound CI			x	
Infer				x
IntelliJ	x	x	x	x
JArchitect	x	x		
Misspell-fixer			x	
NullAway				x
PMD		x	x	x
PVS-Studio			x	x
Scrutinizer		x	x	x
Semmle QL and LGTM	x	x	x	x
SonarQube	x	x	x	x
SpotBugs			x	x

Figure 4: Tools and which domains they cover

such tests and queries has to done. However, many such tests can be reused between projects with equal architectures, and can be checked into version control.

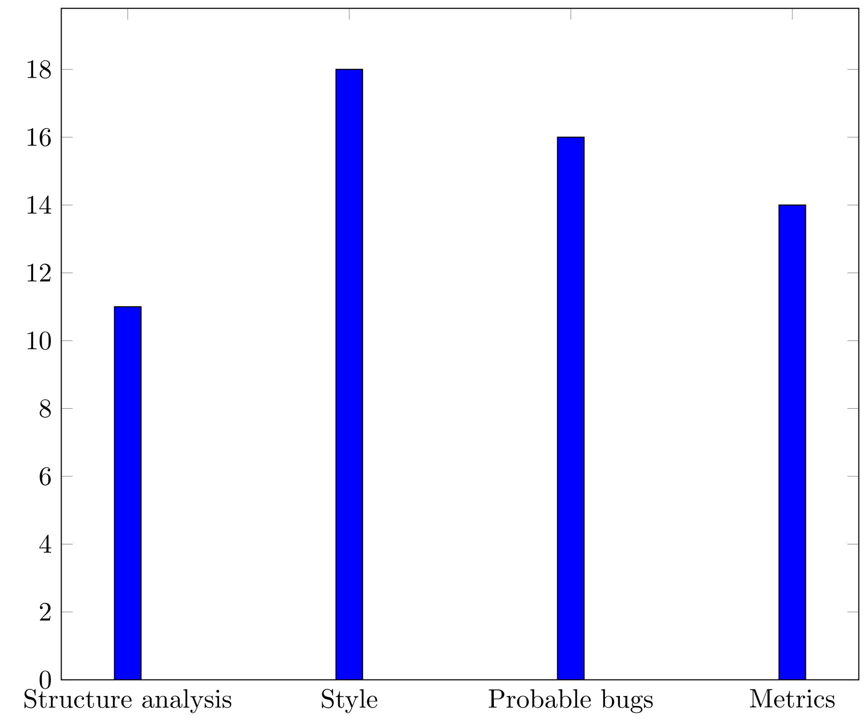


Figure 5: Number of tools for each domain

Other approaches include Graphical User Interface (GUI) tools like HU Software Architecture Compliance Checking Tool (HUSACCT) [14], but requires extensive configuration.

System architecture reconstruction is automatic detection of architecture and architectural patterns, and would be useful for reduced workload of testing and documenting architecture. It is researched in numerous papers e.g [26] and [18], and [11] even state that there are no automatic tools for this purpose and that only quasi-manual, semi-automatic and quasi-automatic tools exists. I wasn't able to find any easily available tools that could provide this either. This proposes more research into this topic.

Architectural anti-pattern detection is the detection of architectural smells. Of the investigated tools only DesigniteJava and Jarchitect included these features. Detection of anti-patterns is a heavily researched area. Azadi U., Arcelli F. and Taibi D. [3] did some good research about mapping architectural anti-patterns with support in tools. According to that paper only closed source products that are good are available, and some are not available anymore. These tools detect anti-patterns, but do not give any support for refactoring.

Samarthyam G., Suryanarayana G. and Sharma T. [29] did a study on the state of the art of architecture smell refactoring and concludes that the topic haven't received extensive focus yet. However, i found one paper [28] proposing a solution for automatic refactoring of the cyclic dependency. The proposed solution was developed as a plugin to the ARCitecture ANalyzer (Arcan) tool. Automatic refactoring for other anti-patterns is an area that needs development.

5.2.2 Design patterns

Design pattern detection could in the same way as architecture pattern detection serve as documentation and be useful for program and structure comprehension activities.

In some languages certain patterns are anti-patterns, because of language features that makes the patterns obsolete. An example is the builder pattern in Kotlin. Because Kotlin supports default parameters the need of the builder pattern in most cases is not necessary. Using detectors of patterns one could possibly avoid such anti-patterns.

A technique for detection and mining design patterns have been shown by [35] and is shown to be successful to find and detect design patterns. The findings in this paper is supposed to be features of a commercial analysis tool called CodeMR [8], but no information about design pattern detection could be found using their online resources. Implementation of such design pattern techniques are subject for more research.

Anti pattern detection is useful for technical debt estimation and for finding code that needs refactoring. A recent mapping study (September 2019) for anti pattern detection [2] concluded that "there is a lack of maturity in the tools and that will restrict their use and adoption in the industry". To accompany this finding, i did only find partly support for detection of design smells in the investigated tools. This proposes more research into this area and to the accompanying topic of automatically refactoring these design smells.

5.2.3 Design principles

Design principles is not directly targeted with the current set of tools. Many of the investigated tools target design principles indirectly. An example includes measuring cyclomatic complexity which could support the statement that the method or class have multiple responsibilities (violating Single Responsibility Principle (SRP)). What is missing is tools that are giving direct hints or advice about possible violations of design principles.

A probable reason for this is that such rules would cause high false-positive rates, causing a lot of noise. However, in this domain the usage of rules with higher false-positive rates could be of value.

For example in the setting of doing code review, rules that is checking for possible violations of

- Liskov Substitution Principle (LSP) when introducing inheritance

- SRP when creating or extending a class with new methods
- Interface Segregation Principle (ISP) when creating or adding methods to interfaces

could be of use to ensure that the reviewer (and the creator) have reasoned about their design during development. To support these rules not giving them a to high false-positive rate, metrics and code analysis could be used. Tools like Jarchitect [17] and NDepend [15] use Dependency Structure Matrix (DSM) and Lack of Cohesion Of Methods (LCOM) to help identify violations of SRP, but does not focus directly on other design principles. To provide more value more tools should be tightly integrated with the development process and focus on a broader range of principles. Therefore, more research into this topic is suggested.

5.2.4 Metrics

There exists a lot of metrics for software, and it is a quite developed area. However, more research into finding optimal values for the different metrics is suggested. Finding optimal values for different metrics could possibly support the identification of design principle violation. Other than that, we were not able to identify further areas that need attention.

5.2.5 Best practices

Best practices include style and error finding. This area is by far the most developed areas within static analysis. My findings support this. However, a few missing features related to testable code have been found.

Testable code is a concern we haven't found any good tools and approaches for. This includes that the code is written in a testable way, and to ensure that production code is not modified in a way that includes bad-practices to fix a testing problem. For example relaxing visibility to fix testability problems is a bad practice, or to provide extra constructors that is only used from tests. Approaches for analysis of inversion of control and testability analysis could be an area of research.

6 Discussion

The result section presented a classification of domains for static analysis and suggestions for further research. The provided classification of tools is not a complete classification of tools. To be complete, more tools have to be investigated and more subsections with regards to style and error-finding needs to be found. Its academic applicability is therefore limited, but for the purpose of suggesting areas for further research it has been useful.

Suggestions for future research includes:

- **System Architecture Reconstruction**
- **Automatic refactoring of architectural anti-patterns**
- **Automatic refactoring of design anti-patterns**
- **Adherence to design principles**
- **Detection of untestable code**

All the areas considered for future research are pretty abstract and at a high level of abstraction. At this level of abstraction it is hard to develop meaningful but strict rules that the static analysis tools forces the developers to follow. Therefore, what i think we should focus more on is providing feedback to the developers so that the developers can make good decisions, instead of enforcing strict rules. Therefore the most interesting suggestion for future research is that no tools have a direct focus on design principles and the adherence to those. This suggestion proposes a new set of tools that could provide useful feedback on the high-level design and structure of code. Therefore, to directly answer **RQ2** i will consider the design-principles domain the least supported and with highest potential.

The other suggestions for research are interesting, but my main concern is that they target structure analysis at a high level where defining strict rules is becoming difficult and with limited potential. The one exception is detection of untestable code which possibly is easier to detect and write possible refactorings for.

To find even more suggestions for future research multiple measures could have been taken. Firstly, the research of tools has purely been based on available documentation of tools and literature on the domains of static analysis. Some of the commercial tools had limited information about their features, so there are possibilities that some tools had useful features that i haven't been able to find. However, to get the most amount of customers, they should provide detailed information about their product and its features, so the potential loss of information is there, but possibly low.

Secondly, by testing out both open source and commercial tools more areas for further research could have been found. In addition, if resources were not an issue, removing the restriction on only Java tools would have been useful.

Thirdly, an approach looking at tools based on machine-learning would have been interesting.

7 Conclusion

108 tools have been considered to find the domains of static analysis and to provide suggestions for future research. A classification of tools for static analysis have been provided and 5 areas for future research have been proposed. Tools for adherence to design principles have been considered the domain with the least support in the current set of tools and with the highest potential.

8 Acknowledgements

I would like to thank my supervisor Hallvard Trætteberg for giving me the opportunity to do research on the topic of static code analysis. I would especially mention the support i received with regards to methodology and the countless discussions about tools and possible opportunities for future research.

References

[1] *Adapter pattern - Wikipedia.* (Accessed on 10/08/2019). URL: `https://en.wikipedia.org/wiki/Adapter_pattern`.

[2] K. Alkharabsheh et al. *Software Design Smell Detection: a systematic mapping study.* 2019. URL: `https://doi.org/10.1007/s11219-018-9424-8`.

[3] U. Azadi, F. Arcelli, and D. Taibi. *Architectural Smells Detected by Tools: a Catalogue Proposal.* 2019. URL: `https://doi.org/10.1109/TechDebt.2019.00027`.

[4] *Camel case - Wikipedia.* (Accessed on 10/08/2019). URL: `https://en.wikipedia.org/wiki/Camel_case`.

[5] *Circular dependency - Wikipedia.* (Accessed on 11/25/2019). URL: `https://en.wikipedia.org/wiki/Circular_dependency`.

[6] *Code Quality and Security — SonarQube.* (Accessed on 11/25/2019). URL: `https://www.sonarqube.org/`.

[7] *Code refactoring - Wikipedia.* (Accessed on 10/14/2019). URL: `https://en.wikipedia.org/wiki/Code_refactoring`.

[8] *CodeMR — Better code, better quality!* (Accessed on 11/01/2019). URL: `https://www.codemr.co.uk/`.

[9] *Coupling (computer programming) - Wikipedia.* (Accessed on 10/24/2019). URL: `https://en.wikipedia.org/wiki/Coupling_(computer_programming)`.

[10] *Cyclomatic complexity - Wikipedia.* (Accessed on 10/24/2019). URL: `https://en.wikipedia.org/wiki/Cyclomatic_complexity`.

[11] Stephanne D. and P. Damien. *Software Architecture Reconstruction: A Process-Oriented Taxonomy.* 2009. URL: `https://doi.org/10.1109/TSE.2009.19`.

[12] *Extensibility - Wikipedia.* (Accessed on 10/14/2019). URL: `https://en.wikipedia.org/wiki/Extensibility`.

[13] *God object - Wikipedia.* (Accessed on 10/24/2019). URL: `https://en.wikipedia.org/wiki/God_object`.

[14] *HUSACCT: A Software Architecture Comformance Checking Tool for Java and C#.* (Accessed on 10/29/2019). URL: `https://github.com/HUSACCT/HUSACCT`.

[15] *Improve your .NET code quality with NDepend.* (Accessed on 10/25/2019). URL: https://www.ndepend.com/.

[16] *Is High Quality Software Worth the Cost?* (Accessed on 10/14/2019). URL: https://martinfowler.com/articles/is-quality-worth-cost.html.

[17] *JArchitect :: Java Static Analysis and Code Quality Tool.* (Accessed on 10/25/2019). URL: https://www.jarchitect.com/.

[18] S. Kang, S. Lee, and D. Lee. *A Framework for Tool-Based Software Architecture Reconstruction.* Mar. 2009. URL: https://doi.org/10.1142/S0218194009004167.

[19] V. Lenarduzzi1, A. Sillitti, and O. Taibi. *A Survey on Code Analysis Tools for Software Maintenance Prediction.* 2019. URL: https://doi.org/10.1007/978-3-030-14687-0_1510.1007/978-3-030-14687-0_15.

[20] *Lint (software) - Wikipedia.* (Accessed on 10/10/2019). URL: https://en.wikipedia.org/wiki/Lint_(software).

[21] Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices.* 2008. URL: https://www.amazon.com/Software-Development-Principles-Patterns-Practices/dp/013597444.

[22] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship.* 2008. URL: https://www.amazon.com/Clean-Code-Handbook-Software-Craftsmanship-ebook/dp/B001GSTOAM.

[23] *Model-view-viewmodel - Wikipedia.* (Accessed on 10/08/2019). URL: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93viewmodel.

[24] *Modular programming - Wikipedia.* (Accessed on 10/14/2019). URL: https://en.wikipedia.org/wiki/Modular_programming.

[25] Z. Mushtaq, G. Rasool, and B. Shehzad. *Multilingual Source Code Analysis: A Systematic Literature Review.* 2017. URL: https://doi.org/10.1109/ACCESS.2017.2710421.

[26] Z. Nayyar and N. Rafique. *Software Architecture Reconstruction Method, a Survey.* Dec. 2014. URL: https://doi.org/10.14569/IJACSA.2014.051219.

[27] N. Medvidović R. N. Taylor and E. M. Dashofy. *Software architecture: Foundations, Theory and Practice.* 2019. URL: https://www.amazon.com/Software-Architecture-Foundations-Theory-Practice/dp/0470167742.

[28] L. Rizzi, F. Arcelli Fontana, and R. Roveda. *Support for architectural smell refactoring.* Sept. 2018. URL: https://doi.org/10.1145/3242163.3242165.

[29] G. Samarthyam, G. Suryanarayana, and T. Sharma. *Refactoring for Software Architecture Smells.* 2016. URL: https://doi.org/10.1145/2975945.2975946.

[30] *Software testability - Wikipedia.* (Accessed on 10/14/2019). URL: `https://en.wikipedia.org/wiki/Software_testability`.

[31] *SOLID - Wikipedia.* (Accessed on 10/08/2019). URL: `https://en.wikipedia.org/wiki/SOLID`.

[32] *Static analysis tools for all programming languages.* URL: `https://github.com/mre/awesome-static-analysis`.

[33] *Technical debt - Wikipedia.* (Accessed on 10/14/2019). URL: `https://en.wikipedia.org/wiki/Technical_debt`.

[34] *TIOBE Index — TIOBE - The Software Quality Company.* (Accessed on 10/07/2019). URL: `https://www.tiobe.com/tiobe-index/`.

[35] T. Umut and B. Feza. *A graph mining approach for detecting identical design structures in object-oriented design models.* 2014. URL: `https://doi.org/10.1016/j.scico.2013.09.015`.

[36] *Understandability for Reuse.* (Accessed on 10/14/2019). URL: `http://www.arisa.se/compendium/node39.html`.

[37] *Unit test your Java architecture - ArchUnit.* (Accessed on 11/25/2019). URL: `https://www.archunit.org/`.

A Appendix

A.1 Initial tool list

```
Checker Framework - https://github.com/typetools/checker-framework/
checkstyle - https://github.com/checkstyle/checkstyle
ck - https://github.com/mauricioaniche/ck
ckjm - http://www.spinellis.gr/sw/ckjm/
ClassGraph - https://github.com/classgraph/classgraph
CogniCrypt - https://www.eclipse.org/cognicrypt/
DesigniteJava - http://www.designite-tools.com/designitejava
Error-prone - https://github.com/google/error-prone
fb-contrib - https://github.com/mebigfatguy/fb-contrib
Find Security Bugs - https://find-sec-bugs.github.io/
google-java-format - https://github.com/google/google-java-format
Hopper - https://github.com/cuplv/hopper
HuntBugs - https://github.com/amaembo/huntbugs
JArchitect - https://www.jarchitect.com
JBMC - http://www.cprover.org/jbmc/
NullAway - https://github.com/uber/NullAway
OWASP Dependency Check - https://www.owasp.org/index.php/OWASP_Dependency_Check
qulice - https://www.qulice.com/
Soot - https://sable.github.io/soot/
Spoon - https://github.com/INRIA/spoon
```

SpotBugs - https://spotbugs.github.io/
Xanitizer - https://xanitizer.com/
AppChecker - https://npo-echelon.ru/en/solutions/appchecker.php
Application Inspector - https://www.ptsecurity.com/ww-en/products/ai/
AppScan Source - https://www.hcltechsw.com/wps/portal/products/appscan/home
APPscreener - https://appscreener.us
ArchUnit - https://www.archunit.org/
Axivion Bauhaus Suite - https://www.axivion.com/en/products-services-9#products_bauhaussuite
Checkmarx CxSAST - https://www.checkmarx.com/products/static-application-security-testing/
coala - https://coala.io/
Cobra - http://spinroot.com/cobra/
codeburner - https://github.com/groupon/codeburner
CodeFactor - https://codefactor.io
CodeIt.Right - https://submain.com/products/codeit.right.aspx
CodeScene - https://empear.com/
cqc - https://github.com/xcatliu/cqc
Coverity - https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast
DeepSource - https://deepsource.io/
Depends - https://github.com/multilang-depends/depends
DevSkim - https://github.com/microsoft/devskim
Fortify - https://software.microfocus.com/en-us/products/static-code-analysis-sast/overview
Goodcheck - https://github.com/sideci/goodcheck
graudit - https://github.com/wireghoul/graudit
Hound CI - https://houndci.com/
imhotep - https://github.com/justinabrahms/imhotep
Infer - https://github.com/facebook/infer
Klocwork - http://www.klocwork.com/products-services/klocwork
Kiuwan - https://www.kiuwan.com/code-security-sast/
oclint - https://github.com/oclint/oclint
pfff - https://github.com/facebook/pfff
PMD - https://pmd.github.io/
Pronto - https://github.com/prontolabs/pronto
pre-commit - https://github.com/pre-commit/pre-commit
PT.PM - https://github.com/PositiveTechnologies/PT.PM
PVS-Studio - https://www.viva64.com/en/pvs-studio/
Reviewdog - https://github.com/haya14busa/reviewdog
Security Code Scan - https://security-code-scan.github.io/
Semmle QL and LGTM - https://semmle.com/
shipshape - https://github.com/google/shipshape
SonarQube - http://www.sonarqube.org/
STOKE - https://github.com/StanfordPL/stoke
Synopsys - https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast
TscanCode - https://github.com/Tencent/TscanCode
Undebt - https://github.com/Yelp/undebt
Veracode - http://www.veracode.com/products/static-analysis-sast/static-code-analysis
WALA - http://wala.sourceforge.net/wiki/index.php/Main_Page

WhiteHat Application Security Platform - https://www.whitehatsec.com/products/static-applica
Wotan - https://github.com/fimbullinter/wotan
XCode - https://developer.apple.com/xcode/
ale - https://github.com/w0rp/ale
Attackflow Extension - https://www.attackflow.com/Extension
DevSkim - https://github.com/Microsoft/DevSkim
Puma Scan - https://github.com/pumasecurity/puma-scan
Security Code Scan - https://security-code-scan.github.io/
vint - https://github.com/Kuniwak/vint
LibVCS4j - https://github.com/uni-bremen-agst/libvcs4j
Violations Lib - https://github.com/tomasbjerre/violations-lib
After the Deadline - https://afterthedeadline.com/
codespell - https://github.com/codespell-project/codespell
languagetool - https://github.com/languagetool-org/languagetool
misspell-fixer - https://github.com/vlajos/misspell-fixer
Misspelled Words In Context - https://github.com/jwilk/mwic
proselint - https://github.com/amperser/proselint/
vale - https://github.com/ValeLint/vale
write-good - https://github.com/btford/write-good
Codacy - https://www.codacy.com/
Code Climate - https://codeclimate.com/
Code Inspector - https://www.code-inspector.com
Codeac - https://www.codeac.io?ref=awesome-static-analysis
CodeFactor - https://codefactor.io
CodeFlow - https://www.getcodeflow.com
Embold - https://embold.io
kiuwan - https://www.kiuwan.com/
Landscape - https://landscape.io/
Layered Insight - https://layeredinsight.com/
LGTM.com - https://lgtm.com/
Nitpick CI - https://nitpick-ci.com
PullRequest - https://www.pullrequest.com
QuantifiedCode - https://www.quantifiedcode.com/
Reshift - https://softwaresecured.com/reshift/
Scrutinizer - https://scrutinizer-ci.com/
SensioLabs Insight - https://insight.sensiolabs.com/
Sider - https://sider.review
Snyk - https://snyk.io/
SonarCloud - https://sonarcloud.io
Teamscale - http://www.teamscale.com/
Upsource - https://www.jetbrains.com/upsource/
IntelliJ

A.2 Final tool list

```
ArchUnit - https://www.archunit.org/
Checker Framework - https://github.com/typetools/checker-framework/
checkstyle - https://github.com/checkstyle/checkstyle
ck - https://github.com/mauricioaniche/ck
ckjm - http://www.spinellis.gr/sw/ckjm/
ClassGraph - https://github.com/classgraph/classgraph
DesigniteJava - http://www.designite-tools.com/designitejava
Error-prone - https://github.com/google/error-prone
fb-contrib - https://github.com/mebigfatguy/fb-contrib
google-java-format - https://github.com/google/google-java-format
Hopper - https://github.com/cuplv/hopper
JArchitect - https://www.jarchitect.com
NullAway - https://github.com/uber/NullAway
SpotBugs - https://spotbugs.github.io/
Axivion Bauhaus Suite - https://www.axivion.com/en/products-services-9#products_bauhaussuite
Coverity - https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast
Depends - https://github.com/multilang-depends/depends
Hound CI - https://houndci.com/
Infer - https://github.com/facebook/infer
PMD - https://pmd.github.io/
PVS-Studio - https://www.viva64.com/en/pvs-studio/
Semmle QL and LGTM - https://semmle.com/
SonarQube - http://www.sonarqube.org/
codespell - https://github.com/codespell-project/codespell
misspell-fixer - https://github.com/vlajos/misspell-fixer
Code Climate - https://codeclimate.com/
Code Inspector - https://www.code-inspector.com
Embold - https://embold.io
Scrutinizer - https://scrutinizer-ci.com/
IntelliJ - http://www.jetbrains.com
```

A.3 Result of domain extraction

```
ArchUnit - https://www.archunit.org/
structure analysis
testing
architecture verification

Checker Framework - https://github.com/typetools/checker-framework/
probable bugs
null analysis
initialization checker
map key checker
optional checker
```

interning checker
lock checker
index checker
fake enum checker
regex checker
format string checker
internationalization errors
signature string checker
gui effect checker
units checker
signedness checking
aliasing checker
purity checker
constant value checker
reflection checker
subtyping checker

checkstyle - https://github.com/checkstyle/checkstyle) - checking Java source code for adher
style
annotations
block checks
design smells
coding
headers
imports
documentation
metrics
miscellaneous
modifiers
naming conventions
regex
size violations
formatting
readability

ck - https://github.com/mauricioaniche/ck
metrics

ckjm - http://www.spinellis.gr/sw/ckjm/
metrics

ClassGraph - https://github.com/classgraph/classgraph
structure analysis
dependency visualization
visualization
class indexing

```
resources indexing
querying

DesigniteJava - http://www.designite-tools.com/designitejava
structure analysis
metrics
architecture smells
design smells
smell visualization
dependency structure matrix
duplicate code detection
complexity detection
best practices

Error-prone - https://github.com/google/error-prone
best practices
probable bugs
style

fb-contrib - https://github.com/mebigfatguy/fb-contrib
probable bugs
style
complexity

google-java-format - https://github.com/google/google-java-format
readability
formatting
style
source file structure

Hopper - https://github.com/cuplv/hopper
probable bugs
index checker
null analysis

JArchitect - https://www.jarchitect.com) :copyright: - Measure, query and visualize your coo
structure analysis
querying
technical debt estimation
quality gates (pass/fail criterias)
metrics
dependency matrix
visualization
dependency graph
abstractness vs instability
dependency cycles
```

```
immutability and purity analysis

NullAway - http://errorprone.info/
probable bugs
null analysis
annotation processor

SpotBugs - https://spotbugs.github.io/
best practices
probable bugs
internationalization errors
multithreaded bugs
style

Axivion Bauhaus Suite - https://www.axivion.com/en/products-services-9#products_bauhaussuite
structure analysis
architecture verification
duplicate code detection
best practices
control flow issues
probable bugs
race condition analysis
metrics
dependency cycles
style

Coverity - https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast
probable bugs
style
resources leaks
null analysis
incorrect api usage
initialization checker
control flow issues
error handling issues
incorrect expressions
multithreaded bugs
insecure data handling

Depends - https://github.com/multilang-depends/depends
structure analysis
dependency analysis

Hound CI - https://houndci.com/
style
readability
```

Infer - https://github.com/facebook/infer
probable bugs
multithreaded bugs
null analysis

PMD - https://pmd.github.io/
metrics
best practices
style
design smells
documentation
probable bugs
multithreaded bugs

PVS-Studio - https://github.com/viva64/pvs-studio-check-list
probable bugs
style
best practices

Semmle QL and LGTM - https://LGTM.com
structure analysis
style
probable bugs
queries
metrics
best practices
testability
modularity
readability
portability

SonarQube - http://www.sonarqube.org/
structure analysis
probable bugs
code smell
metrics
style
brain-overload
best practices
cert
clumsy
confusing
convention
design smells
lock-in

```
owasp
pitfall
suspicious
unpredictable
unused
user-experience
duplicate code detection
api-design
leak
obsolete
regex

IntelliJ - http://www.jetbrains.com
structure analysis
metrics
best practices
style
duplicate code detection
probable bugs
Abstraction issues
arquillan
assignment issues
bitwise operation issues
class metrics
class structure
cloning issues
code maturity
style
compiler issues
concurrency annotation issues
control flow issues
data flow
declaration redundancy
dependency issues
encapsulation
error handling
finalisation
general
imports
inheritance issues
initialization checker
internationalization errors
j2me issues
java language level issues
java language level migrations aids
javabeans issues
```

```
documentation
unit
logging
memory
method metrics
modularisation issues
naming conventions
numeric issues
packaging issues
portability
probable bugs
properties files
reflective access
resource management
serialisation issues
testNG
threading issues
tostring issues
verbose or redundant code constructs
visibility
readability
spellchecking
formatting

codespell - https://github.com/codespell-project/codespell
readability
spellchecking

misspell-fixer - https://github.com/vlajos/misspell-fixer
readability
spellchecking

Code Climate - https://codeclimate.com/
structure analysis
metrics
duplicate code detection

Code Inspector - https://www.code-inspector.com
best practices
style
documentation
design smells
probable bugs
duplicate code detection
metrics
```

```
Embold - https://embold.io
structure analysis
metrics
style
design smells
probable bugs
code quality visualization
duplicate code detection
ai bug detection feedback

Scrutinizer - https://scrutinizer-ci.com/
metrics
style
duplicate code detection
```