BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
SPECIALIZATION SOFTWARE ENGINEERING

DISSERTATION THESIS

CODE SMELL PRIORITISATION

Supervisor
PROF. DR. MOTOGNA SIMONA

Author
SUCIU ANDREI-MIRCEA

2023

UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA INGINERIE SOFTWARE

LUCRARE DE DISERTAŢIE

PRIORITIZARE DE CODE SMELL

Conducător științific
PROF. DR. MOTOGNA SIMONA

Absolvent
SUCIU ANDREI-MIRCEA

2023

ABSTRACT

This paper seeks to create a valuable tool for software developers to help in the prioritisation of code smell refactoring by analysing four types of smells: God Class, Data Class, Brain Method and Dispersed Coupling. The user can input metrics generated by the MetricsTree IntelliJ plugin, and the tool computes code smells detected with configurable formulas and thresholds for each metric while supplying many valuable graphs and tables to aid in discovering trends and practices in the analysed project. This paper will dive into the Metrics used for each Code Smell, detailing how those smells appear in a code base and how to avoid and eliminate them. After that, it will explore the algorithm created to compute the priority, delve into details of the tool design and implementation, and analyse how it can improve code bases and practices. Concluding the paper are some final remarks and further research ideas.

Contents

Chapter 1

Introduction

As software projects grow, a need appears for the design to change and adapt to fit the new requirements, and such changes are done via refactoring. But understanding what needs to be refactored and when can be time-consuming, so automatic software inspection has become an indispensable tool for software engineers. An automatic, static source code inspection is done via metrics that identify and measure code smells: bad design and programming techniques, which the industry learned through experience. [vEM02]

This thesis aims to explore the domain of software quality by addressing the relevance and distribution of code smells in large complex projects. The approach is taken software metrics as indicators for symptoms corresponding to different code smells, thus offering a classification method for them. It is also our purpose to provide an inspection tool that prioritises the detected code smells so that a developer knows the most urgent issues to tackle in refactoring. The tool has a simple GUI and provides statistics, taking into consideration different versions of the project.

As a short description of what is tackled in this paper:

- Chapter 2 provides definitions and sources of the software metrics that are used for the code smell detection
- Chapter 3 describes the code smells, their metric detection thresholds and detection formulas
- Chapter 4 discusses the smell intensity of each code smell, which will be used in creating the refactoring prioritisation for the user
- Chapter 5 provides details for the application design, implementation and use cases
- Chapter 6 analyses the tool's usefulness and how it can improve code bases and their developers by leveraging the gathered data through graphs and valuable tables.

- Chapter 7 concludes the paper and proposes further work

The tool developed in this paper has several advantages: it is proven effective in prioritising code smell handling and refactoring by providing a straightforward graphical interface with deep analysis and valuable graphs.

Moreover, the application's modularity and the presence of a configuration file provide flexibility, facilitating its expansion by adding new code smell definitions or changing the existing ones.

The application can also be used just for generating a CSV for each defined smell containing all instances of its detection, bypassing the GUI.

The generation of CSVs and the file structure used by the application also provide an easy way of archiving different versions of the analyzed program for later analysis with newer versions in order to evaluate the efficiency of each maintainer's refactoring policy.

Chapter 2

Metrics

Software Metrics are a way to measure and evaluate the quality of a software application by making a static analysis of the source code [IEE94]. Leveraging those metrics enables developers to understand better what needs to be refactored or more exhaustively tested. Many types of code metrics analyze different levels, such as method, class, module or project levels. Usually, those metrics have a so-called "Quality Gate", a value of the metric that, if exceeded, would render warnings to the developer. Those Quality Gates are not a set standard, but they are usually defined on a company basis, and many papers seek to define the best limit for each metric, as they are very context-dependent. Thus, metrics can be used on their own to measure specific information for any analysis level, but they can also be utilised in another way. The use of metrics in this paper is that they will be grouped into different units called code smells, which provide a quantifiable way to measure design problems. More information on code smells is presented in the following chapter.

Definitions of Code Metrics may vary, so this chapter defines the source and definition of each metric used in the code smell detection phase.

2.1 ATFD (Access to Foreign Data)

The ATFD metric [LM06] calculates the number of attributes from external data that are accessed directly or through accessor methods. External data refers to data that resides outside the class or module, such as data from other classes, global variables, or data retrieved from databases or files. When a class or module accesses many external data items, it increases its coupling with other components. It can result in increased complexity, reduced modularity, and decreased maintainability.

2.2 WMC (Weighted Methods for Class)

This metric is part of the popular CK metrics [CK94], and it is calculated as the sum of all the complexities of the methods in the class. Originally, CK considered a value of 1 for each method, but the modern approach uses the CYCLO metric [McC76] for each method. The higher the value, the greater the functional complexity of the class. This becomes a problem in maintainability and code reuse, as such classes tend to be problem-specific.

2.3 CYCLO (McCabe's Cyclomatic Number)

McCabe's Cyclomatic Complexity [McC76], named after Thomas J. McCabe, is a quantitative measure used to evaluate the complexity of a software program. It provides a numerical value that indicates the number of independent paths through a program's source code. However, as stated above, we will limit this complexity calculation to a single method.

Cyclomatic Complexity is based on the control flow of a method and is determined by the number of decision points, such as conditional statements (if, switch, case) and loops (for, while) in the code. Each decision point adds to the overall complexity of the method. The formula to calculate CYCLO is shown in 2.1:

$$M = E - N + 2P \tag{2.1}$$

- M represents the cyclomatic complexity of the method

  In the method's control flow graph:

- E is the number of edges
- N is the number of nodes
- P is the number of exit points

CYCLO measures the number of possible paths through the method. Higher cyclomatic complexity indicates increased program complexity, implying incredible difficulty in understanding, testing, and maintaining the code.

2.4 TCC (Tight Class Cohesion)

The TCC metric measures the grade of cohesion within a class or module. It reveals how closely related the methods and attributes of a class are to each other. The metric is calculated by examining directly connected methods within a class [BK95].

Two visible methods are directly connected if they access the same instance variables of the class. TCC analyzes the cohesion between methods and determines the proportion of directly connected methods that are actually invoked together compared to the total number of possible method pairs.

A high TCC value indicates strong cohesion, meaning that the methods within a class frequently interact with each other, suggesting that the class focuses on one responsibility, while a low TCC value suggests weak cohesion and thus may indicate that the class has multiple responsibilities.

2.5 NOPA + NOAM

These are two simple metrics: NOPA is the Number Of Public Attributes, and NOAM is the Number Of Accessor Methods (getters and setters).[LM06]

2.6 LOC (Lines of Code)

One of the most basic metrics, LOC, describes the number of lines of code in a method, class, package, or project. But even this basic metric can be refined into various definitions [AFP20]:

SLOC - Source Lines of Code: doesn't add comments or white lines to the final metric

LLOC - Logical lines of Code: highly dependent on the programming language, it refers to the executable lines of code

RLOC - Relative Lines of Code: measures the percentage of LOC of the method compared to LOC of its class. Useful to identify large methods or god classes, which should be broken down into multiple units during refactoring.

This metric is often misunderstood and debated [BTP12], as it is often used incorrectly, such as programming effort estimation [Eva]. The impact and complexity of a solution cannot be measured by the LOC metric, as a method can be written verbose and less readable at the same time. That said, it can be helpful when comparing the magnitude of two different projects (10.000 LOC vs 1.000.000 LOC) or the impact of a proposed solution (1000 LOC might be too much for a single commit).

The usefulness of this metric is in identifying bloat and signalling what component should be refactored into smaller parts to keep the code readable and maintainable. Also it is used in computing other metrics.

For the Brain Method code smell, SLOC will be used.

2.7 MAXNESTING (Maximum Nesting Level)

This metric measures the maximum depth of nested control structures(e.g. loops, conditionals) within a method. A high MAXNESTING value can lead to difficulties debugging and maintaining the method.

2.8 NOAV (Number of Accessed Variables)

This metric measures the total number of variables accessed directly from the method. Variables are defined as parameters, local variables, and global variables.

2.9 CINT (Coupling Intensity)

This metric measures the number of distinct operations called by the measured operation.

Chapter 3

Code Smells

The term code smell was coined by Martin Fowler in the book ”Refactoring” and popularised and expanded upon by Robert C. Martin in the book ”Clean Code”. A code smell was defined as an indication in the source code that suggests a possible violation of good code practices. It indicates an area of code that would benefit from refactoring [Fow99]. The presence of code smells might reveal a maintainability issue with the code, and they often appear when design principles are misused or even not used at all [MC09]. Another reason for the code smell appearance in source code is the prioritisation of feature delivery over code quality and maintenance. Addressing code smells will make future development more accessible and might prevent bugs and failures, limiting technical debt.

There has been much research done, and additions contributed to the domain of code smells, which prompted most software development companies to use some code quality checking tools such as Sonar to analyse their code. A problem that often arises is that large projects generate many code smells, and there needs to be a way to prioritise them in a way that would make refactoring cost-effective.

This chapter details the various code smells and their metrics thresholds used to detect them, which be analysed for their intensity. The detection algorithms of code smells used in this paper are considered according to the definitions from the book of Lanza and Marinescu. [LM06] Code smells are composed of code metrics combined into a smell detection formula using different thresholds for each metric.

The thresholds LOW, MEDIUM, and HIGH presented for each metric are taken from the analysis of [AFFZY15]. This paper presents a data-driven method that respects the distribution of each metric: the Quantile Functions(QF) of each metric show that the distributions are extremely skewed: the QF grows slowly, and after a certain point, it grows very fast, so using this data fixes an issue of the [LM06] paper, which assumed a normal distribution.

To those three thresholds, I will add VERY-LOW (50% of the LOW threshold) and VERY-HIGH (50% of the HIGH threshold) to compute an intensity for each smell.

This paper presents four code smells: two class-related(God Class, Data class) and two method-related(Brain Method, Disperesed Coupling).

3.1 God Class

This smell identifies a class that is too complex, has many unrelated methods, does too much work on its own, and uses data from other classes.

God Class characteristics:

- Bloated size: a large number of methods and attributes
- High complexity: convoluted control flow, numerous conditional statements, and many dependencies
- More than a few responsibilities: not conforming to SOLID
- Low cohesion: methods inside the class are not related, making it difficult to assess the behaviour of the class
- Strong coupling with other classes

This type of smell violates the single-responsibility principle, creating maintainability issues and duplicate code. Also, the complexity becomes unmanageable, and changes to the code may have unpredictable consequences [Car].

To identify a God Class, formula 3.1 is used:

$$ATFD > LOW \wedge WMC \geq HIGH \wedge TCC < LOW \Rightarrow GodClass \tag{3.1}$$

Table 3.1 shows the thresholds used for each metric.

Metric	VERY-LOW	LOW	MEDIUM	HIGH	VERY-HIGH
ATFD	5	10	13	22	33
WMC	20	25	36	64	96
TCC	0.25	0.33	0.5	0.66	0.75

Table 3.1: God Class Metrics Thresholds

3.2 Data Class

This code smell refers to a class that mostly serves as a container for data and is without any significant logic. The lack of functional methods indicates that the class is likely coupled with other classes and violates the OO design. Some characteristics are:

- Lack of functionality: no meaningful methods or behaviour other than getters and setters
- No data encapsulation: expose internal data with public fields or properties
- Minimal or no validation
- Manipulating data that might belong elsewhere

This type of code smell increases code complexity, reduces maintainability and increased coupling.

To identify a Data Class, formula 3.2 is used:

$$NOPA + NOAM > LOW \wedge WMC < HIGH \Rightarrow DataClass \tag{3.2}$$

Table 3.2 shows the thresholds used for each metric.

Metric	VERY-LOW	LOW	MEDIUM	HIGH	VERY-HIGH
NOPA + NOAM	4	7	10	17	26
WMC	11	21	32	57	86

Table 3.2: Data Class Metrics Thresholds

3.3 Brain Method

This code smell refers to a method with a large amount of complex logic or performs multiple tasks. Methods usually become Brain Methods by adding functionality to them over time. They tend to centralize the class's behaviour, similar to how a God Class centralized the behaviour of a system. Some characteristics are:

- Method is too large: challenging to read, making future modifications difficult and tends to be more error-prone by having more than one functionality
- Excessive branching: a large number of possible paths through the code
- Many variables used: can induce errors from the developer

This type of code smell creates hard-to-understand and debug methods and reduces maintainability and reusability.

To identify a Brain Method, formula 3.3 is used:

$$\begin{aligned} &LOC > HIGH \wedge CYCLO \geq HIGH \wedge \\ &\wedge MAXNESTING \geq HIGH \wedge NOAV > HIGH \Rightarrow BrainMethod \end{aligned} \tag{3.3}$$

Table 3.3 shows the thresholds used for each metric.

Metric	VERY-LOW	LOW	MEDIUM	HIGH	VERY-HIGH
LOC	10	20	28	45	68
CYCLO	4	8	10	15	23
MAXNESTING	2	4	4	5	8
NOAV	6	11	14	20	30

Table 3.3: Brain Method Metrics Thresholds

3.4 Dispersed Coupling

Dispersed coupling refers to a situation where the methods of a class rely on many operations from a large number of other classes. In other words, when a class depends on a significant number of operations across multiple classes, it is considered to have dispersed coupling with those provider classes. Some characteristics include:

- Method Calls: Affected classes have many methods calls to other classes.
- Moderate Communication Intensity: The affected class interacts with external classes it is coupled with, but the communication is not extensive. Each affected method typically calls one or a few methods from each coupled class.
- Nested Conditionals: The calling methods within the affected class exhibit a non-trivial level of nested conditionals, often seen as nested IF-ELSE statements.

The problem arises when a small change is made in one of those provider classes, which would propagate to all the dispersed classes, making the development of new features intensive.

To identify a Dispersed Coupled method, formula 3.4 is used:

$$MAXNESTING > LOW \wedge CINT > MEDIUM \Rightarrow DispersedCoupling \tag{3.4}$$

Table 3.4 shows the thresholds used for each metric.

Metric	VERY-LOW	LOW	MEDIUM	HIGH	VERY-HIGH
MAXNESTING	2	4	4	5	8
CINT	4	7	9	12	18

Table 3.4: Dispersed Coupling Metrics Thresholds

Chapter 4

Smell Intensity

The idea of smell intensity was first proposed in the paper ″Towards a prioritization of code debt: A code smell Intensity Index″ by Vincenzo Ferme et al. It proposes a code smell intensity index that quantifies the amount of each smell [FAFZR15]. This paper uses the intensity level intervals defined by Ferme et al. :

1. VERY-LOW: [1, 3.25)
2. LOW: [3.25, 5.5)
3. MEDIUM: [5.5, 7.75)
4. HIGH: [7.75, 10)
5. VERY-HIGH: [10, 10]

The algorithm I used to map each smell to an intensity level differs, though, as I do not use quantiles as in the Intensity Index paper, but fixed values, so I had to develop my own algorithm.

After each smell is detected and has values associated with each metric, the smell intensity for each metric is calculated. In other words, in order to calculate the intensity of the smell, the intensity of each metric needs to be computed in the following steps:

4.1 Define the intervals for the specific metric

1. Each computed metric has a threshold value ***a*** and an extremity threshold value ***b***.
    - We define ***a*** to be the threshold value used in the formula of the smell's computation. For example, in the God Class smell, ATFD metric, ***a*** would be the value of **LOW**

- We define ***b*** to be the most extreme value of a metric's thresholds following the comparison sign in the formula of the smell. So, for a metric using $>$ or $\geq$, ***b*** is the value of **VERY-HIGH**. For $<$ or $\leq$, ***b*** is the value of **VERY-LOW**.
- For the following, we will consider ***b*** to be **VERY-HIGH**. To compute for **VERY-LOW**, the values ***a*** and ***b*** just need to be flipped and have the interval [***b***,***a***]

2. Those two values form an interval [***a***,***b***), which needs to be split into 4 equal parts, matching the intensity level intervals, excluding VERY-HIGH. Thus, each part would have the length $\delta = \frac{b-a}{4}$
3. The new intervals are
   - VERY-LOW: $[a, a + \delta)$
   - LOW: $[a + \delta, a + 2\delta)$
   - MEDIUM: $[a + 2\delta, a + 3\delta)$
   - HIGH: $[a + 3\delta, b)$

As an example, let us map the intervals for WMC for the God Class smell.

The formula states that $WMC \geq HIGH$, so ***a***= WMC(HIGH)= 64 and because we have the $\geq$ sign, ***b***= WMC(VERY-HIGH)= 96. Thus $\delta = \frac{96-64}{4} = 8$, so the intervals are:

- VERY-LOW: $[64, 72)$
- LOW: $[72, 80)$
- MEDIUM: $[80, 88)$
- HIGH: $[88, 96)$

Figure 4.1 shows an example of MAXNESTING metric:

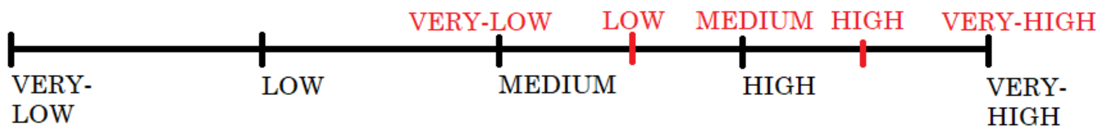


Figure 4.1: MAXNESTING metric visual example

4.2 Normalize value of given metric

The value of a metric needs to be normalized to fit the intensity level intervals. Having all metrics normalized will enable the computation of a value for the smell.

1. For the given metric value *x*, we find its matching interval, $x \in [i_{min}, i_{max}]$. If $x \geq b$, return 10 (it belongs to the VERY-HIGH intensity level interval) and skip the following steps.

2. For the matching interval, $x \in [i_{min}, i_{max}]$, get the target intensity level interval $[t_{min}, t_{max}]$.

3. Linearly scale *x* to $[t_{min}, t_{max}]$ with formula (4.1)

$$x_{scaled} = \frac{x - i_{min}}{i_{max} - i_{min}} \times (t_{max} - t_{min}) + t_{min} \tag{4.1}$$

   Where:

   - $\frac{x-i_{min}}{i_{max}-i_{min}}$ maps x to $[0, 1]$
   - multiplying by $(t_{max} - t_{min})$ maps x to $[0, t_{max} - t_{min}]$
   - adding t_{min} maps x to $[t_{min}, t_{max}]$

4. return the normalized *x*

As an example: Let us say that we want to compute the normalized value of the WMC metric in the God Class smell with a value of $x = 87$.

$x \in [80, 88)$ so it belongs to MEDIUM, which is $[5.5, 7.75)$. Now we use the formula to map normalize x:

$x_{scaled} = \frac{87-80}{88-80} \times (7.75 - 5.5) + 5.5 = 7.46$

This represents the value of the metric, which will be used in the smell intensity calculation.

Figure 4.2 shows a visual representation of the mapping:

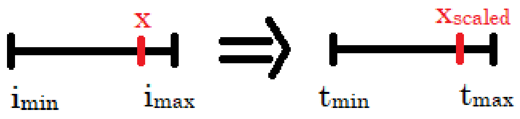


Figure 4.2: Mapping visual representation

4.3 Compute smell intensity

With all the values of each metric calculated, the smell intensity is determined by a simple average. The value obtained is then mapped to an intensity interval.

So if we used the earlier computed value for WMC and define the values 9.32 for ATFD and 8.45 for TCC, we would compute smell intensity S_{WMC}:

$S_{WMC} = \frac{9.32+8.45+7.46}{3} = 8.41 \implies S_{WMC} = HIGH$

Note that for a smell to be VERY-HIGH, all its metrics need to be computed to VERY-HIGH. This is by design so that VERY-HIGH warnings are seen as critical and necessitate immediate attention and refactoring.

This smell intensity is shown to the user in the form of a priority: the higher the intensity, the higher the need to prioritize refactoring to prevent future complications.

Chapter 5

Tool for Code Smell Priority

The purpose of this application is to provide a tool that helps the user to prioritise code refactoring by signalling code smells and their refactoring priority. The application has three main modes: viewing a single version of the analysed code, comparing two versions, and comparing multiple versions.

- While viewing a single version, the user has an ordered list containing all the smells and graphs the number of smells and priorities. Also, there are multiple tabs with details for each smell.
- Two version comparison shows lists of new, persisted, resolved, worse, and better smells.
- Multiple version comparison shows graphs of total smells per version, smell categories per version, and smell priorities per version.

The application backend is modular and flexible, featuring a configuration file so the user can modify smell formulas and thresholds or even add new smells. Also, the analysis is done in different steps, so it is possible to add or modify the data at different points in its analysis.

The frontend provides a simple yet clean presentation of the data, and it is completely decoupled from the backend, meaning the user can choose not to use the provided frontend but use the backend as an API or build their own frontend. It is also easy to add new tabs in the views to reflect new code smells.

All project analyses are saved in CSV files, making them humanly readable and saving different version analyses for archiving purposes or for easier version comparison at a later date. New versions can also easily be added to existing projects.

5.1 Analysis and Design Details

5.1.1 Simple Scenario

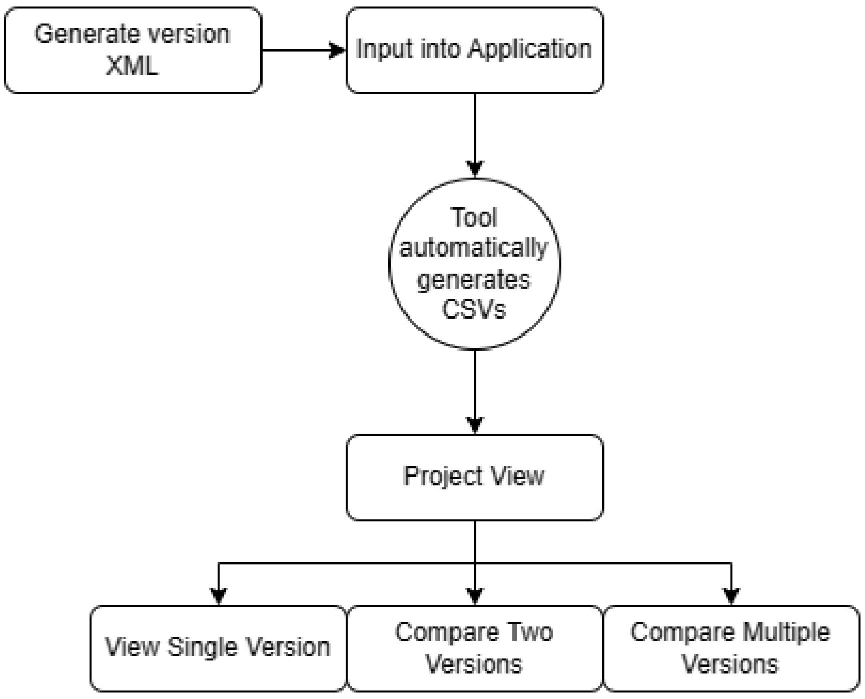


Figure 5.1: Simple Scenario

Figure 5.1 shows a simple utilisation scenario that I will use to explain the basic flow of the application:

- The user generates an XML for each version of their target application using the MetricsTree IntelliJ Plugin. I have chosen this metric generation tool as it is one of the most advanced tools for the computation of Java static code analysis, having 61 code metrics, the most extensive set of any similar tool [BB23].
- Any number of XMLs are input for the application for a specific project. More versions can be added at a later date.
- The tool generates CSVs containing computed smells, their metrics value, and the overall intensity of the smells.
- The user is redirected to the Project View window in which they can choose any number of versions for analysis.
- A new window is opened for any Single Project View, Compare Two Projects View or Compare Multiple Versions View.
- The user can jump directly into Project View without doing the analysis again, as all data is saved in the CSVs and can be accessed at any point.

5.1.2 Data Flow

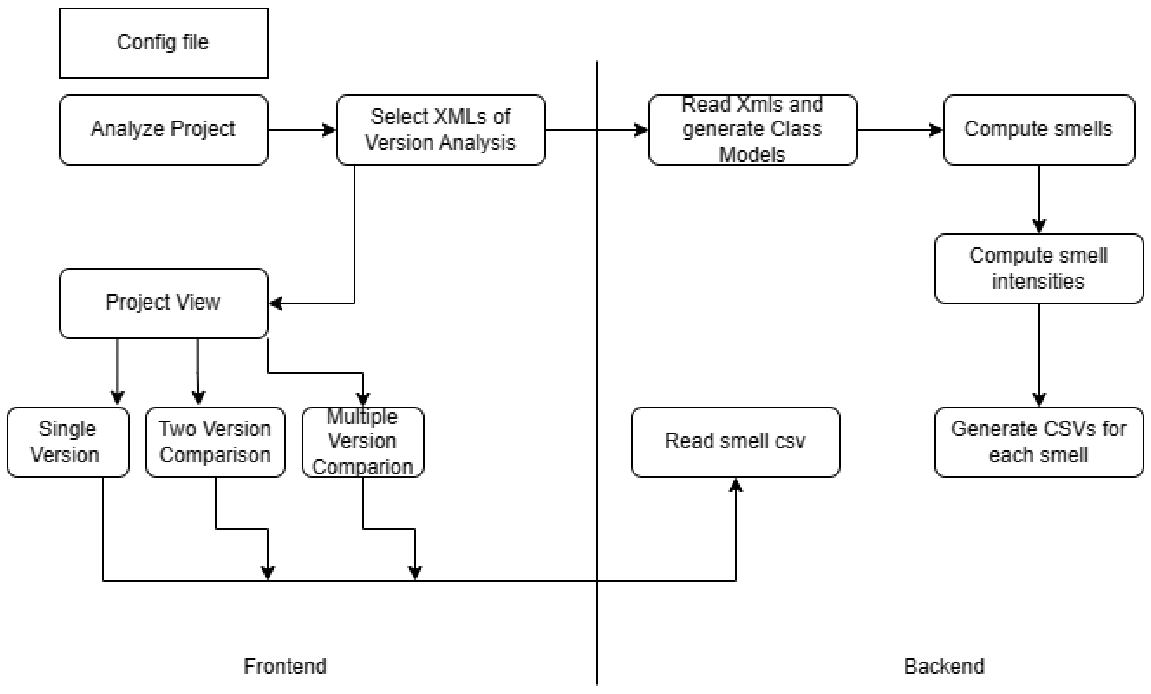

Config file
Analyze Project
Select XMLs of Version Analysis
Read Xmls and generate Class Models
Compute smells
Compute smell intensities
Project View
Single Version
Two Version Comparison
Multiple Version Comparion
Read smell csv
Generate CSVs for each smell
Frontend
Backend

Figure 5.2: Data Flow

Figure 5.2 shows a representation of the data flow in the application.

- The config file can be modified before runtime to set or modify the Smell Formulas and their Metrics Thresholds. This provides a way to fine-tune the detection algorithm and add new smells or metrics, which must be present in the input XML.
- The user inputs generated XMLs into the application
- The XMLs are read, and Class Models are generated, containing metrics for the class itself but also for the methods inside the class
- Those Class Models are passed to a Smell Checker component, which generates Smell Models based on the Smell Formula and Metrics Thresholds set in the config file.
- The next step is to compute smell intensities by passing the Smell Model to a Smell Intensity calculator, which generates an intensity for each smell.
- The backend now writes all smell intensities to a file for each type of smell containing the name of the class/method, priority and class metrics. The smell intensity is presented as a priority to the user, as higher intensities necessitate immediate attention, while VERY-LOW intensities can be ignored in the short term.

- The CSVs are read by the backend and passed on to the frontend, where data is organised and logically for easy reading and statistics generation.

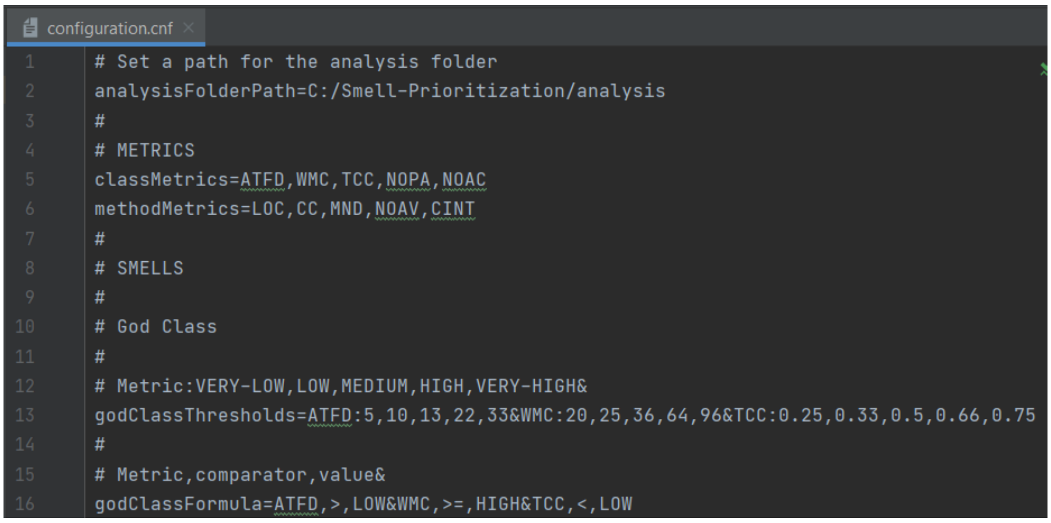
5.2 Implementation Details

5.2.1 Backend

The Backend part of the application is written in Java 15, and the code is modular to provide a flexible way to add/modify/delete data between each step of the analysis process. This is why smells and smell intensities are computed at different steps, for example.

Configuration File

Before runtime, the configuration file shown in figure 5.3 can be modified by the user to set a path for the folder in which all analyses of various projects are stored, define the metrics to be read from the XMLs, and define smells, formulas for the smells and threshold values for each metric. At present, adding new smells requires the user to also add a few trivial lines of code in the backend. This known limitation would require a small amount of further work to improve the configuration file.

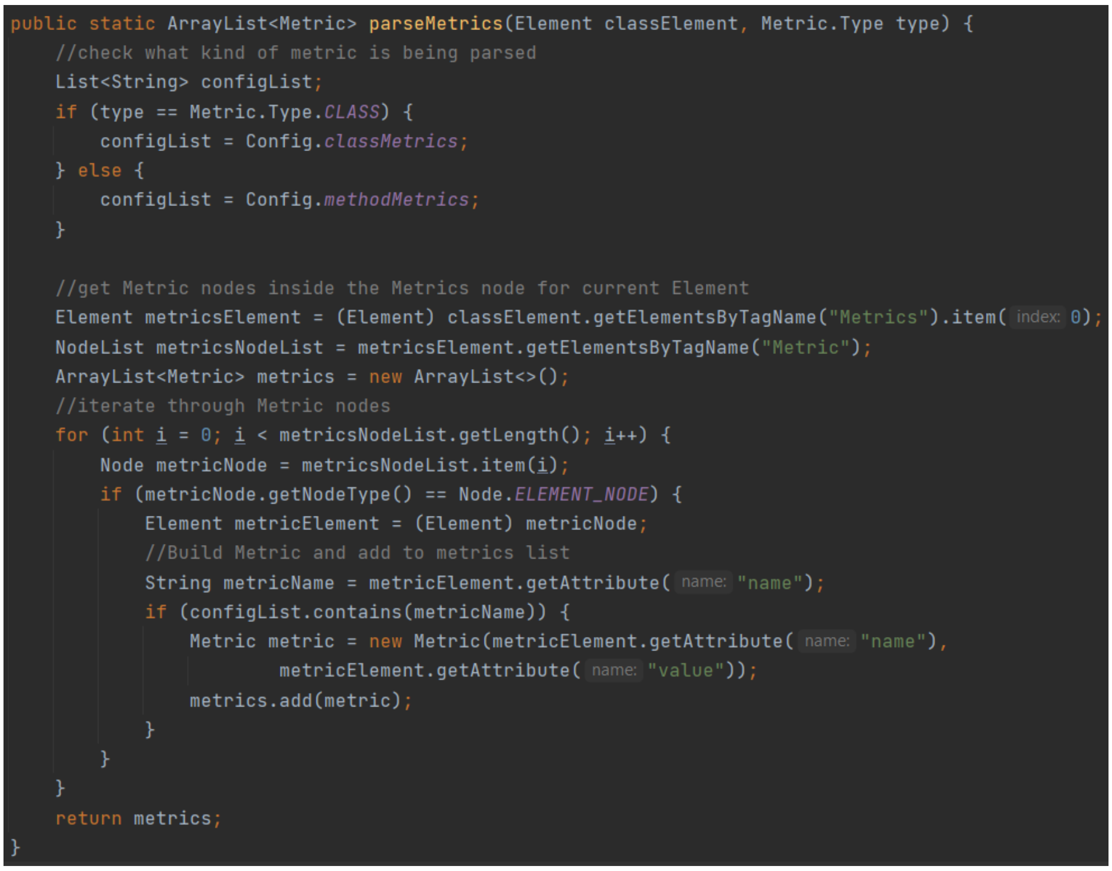


```
# Set a path for the analysis folder
analysisFolderPath=C:/Smell-Prioritization/analysis
#
# METRICS
classMetrics=ATFD,WMC,TCC,NOPA,NOAC
methodMetrics=LOC,CC,MND,NOAV,CINT
#
# SMELLS
#
# God Class
#
# Metric:VERY-LOW,LOW,MEDIUM,HIGH,VERY-HIGH&
godClassThresholds=ATFD:5,10,13,22,33&WMC:20,25,36,64,96&TCC:0.25,0.33,0.5,0.66,0.75
#
# Metric,comparator,value&
godClassFormula=ATFD,>,LOW&WMC,>=,HIGH&TCC,<,LOW
```

Figure 5.3: Config File

XML Parser Step

At the Parser step, a parser reads an input XML, building a ClassWithMetrics model, which holds a class name, metrics for the class and an array of MethodWithMetrics. Each MethodWithMetrics contains a method name and metrics of the method. Only metrics defined in the config file are passed to the models. The parser builds the name of each entity by iterating through the parent nodes and building a full package name, which also enables the analysis of inner classes. Figure 5.4 shows the code for parsing metrics for a class/method. Note that an XML needs to be named after the version of the analysed project. The naming convention should follow the Semantic Versioning method [Tom], or variations of such if the user wants the versions to show up properly ordered in the Project View and the Multiple Versions Comparison View



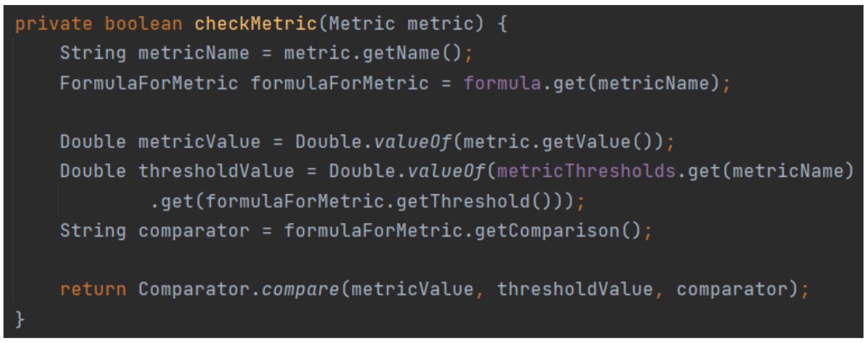
```
public static ArrayList<Metric> parseMetrics(Element classElement, Metric.Type type) {
    //check what kind of metric is being parsed
    List<String> configList;
    if (type == Metric.Type.CLASS) {
        configList = Config.classMetrics;
    } else {
        configList = Config.methodMetrics;
    }

    //get Metric nodes inside the Metrics node for current Element
    Element metricsElement = (Element) classElement.getElementsByTagName("Metrics").item( index: 0);
    NodeList metricsNodeList = metricsElement.getElementsByTagName("Metric");
    ArrayList<Metric> metrics = new ArrayList<>();
    //iterate through Metric nodes
    for (int i = 0; i < metricsNodeList.getLength(); i++) {
        Node metricNode = metricsNodeList.item(i);
        if (metricNode.getNodeType() == Node.ELEMENT_NODE) {
            Element metricElement = (Element) metricNode;
            //Build Metric and add to metrics list
            String metricName = metricElement.getAttribute( name: "name");
            if (configList.contains(metricName)) {
                Metric metric = new Metric(metricElement.getAttribute( name: "name"),
                        metricElement.getAttribute( name: "value"));
                metrics.add(metric);
            }
        }
    }
    return metrics;
}
```

Figure 5.4: Parsing a Metric

Smell Checker Step

Next, the Smell Checker step receives a class/method with their respective metrics, the formula for the checked smell, and the thresholds. It then iterates through each metric of the entity that also belongs to the smell and checks it against its threshold by comparing it using the sign defined in the formula. If all smells pass their threshold, the entity is considered smelly, and the smell checker returns true. Otherwise, it returns false. All smells defined in this paper use the $\wedge$ operator, so this implementation fits. If a smell containing the $\vee$ operator is to be defined, trivial changes must be made. Figure 5.5 shows a code snippet of how a metric is checked.



```java
private boolean checkMetric(Metric metric) {
    String metricName = metric.getName();
    FormulaForMetric formulaForMetric = formula.get(metricName);

    Double metricValue = Double.valueOf(metric.getValue());
    Double thresholdValue = Double.valueOf(metricThresholds.get(metricName)
            .get(formulaForMetric.getThreshold()));
    String comparator = formulaForMetric.getComparison();

    return Comparator.compare(metricValue, thresholdValue, comparator);
}
```

Figure 5.5: Smell Checking a Metric

Compute Smell Intensity Step

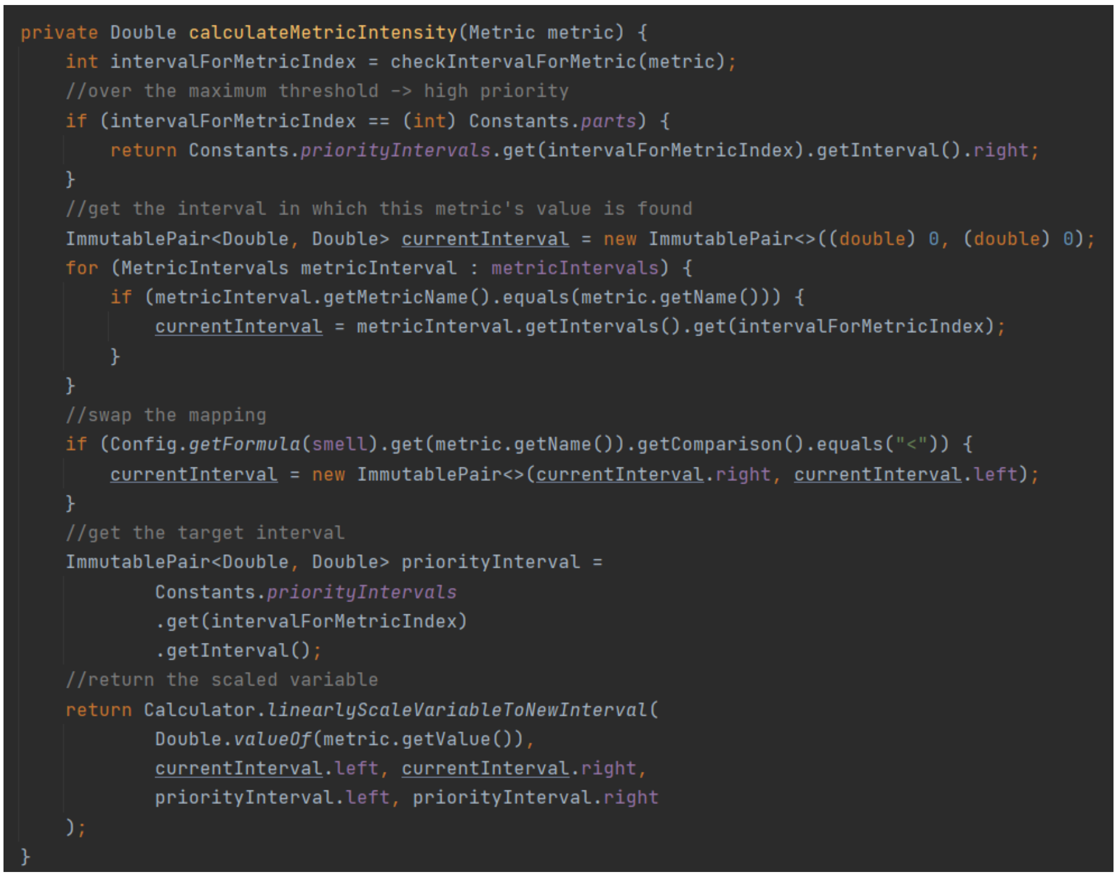
The last analysis step is to compute the smell intensity. The algorithm used to calculate this is detailed in Chapter 4. The smelly entities are passed to an Intensity-Calculator, which calculates the intensity for each smell by computing the intensity of each metric, as shown in figure 5.6, belonging to that smell and their average. This is then returned as a SmellIntensity object, containing the name of the entity, its metrics with name and value, and finally, the Priority of the smell, based on the average of the metrics intensities. Figure 5.6 shows a snippet of the algorithm detailed in Chapter 4. Note the steps: find the interval of the metric, swap the interval if the sign necessitates, find the intensity intervals and, finally, linearly scale the value of the metric to the intensity interval.

Write CSV Step

After the analysis is complete, the SmellIntensity objects are passed to a CSV Writer, which writes a CSV file for each type of smell in a folder named after the version of the project, having as the parent folder named after the project. All data is written in a human-readable format so that it is saved, and there is no need to analyse the version again, and the user can use the CSV in other ways than viewing it through the provided frontend. Figure 5.7 shows the analysis folder structure and an example of the Dispersed Coupling Smell.

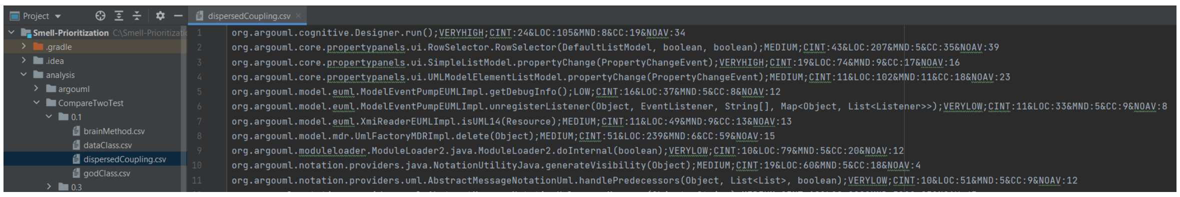
Read CSV Step

At the request of the frontend, CSVs can be read, having each line passed as SmellIntensity objects back to it.



```java
private Double calculateMetricIntensity(Metric metric) {
    int intervalForMetricIndex = checkIntervalForMetric(metric);
    //over the maximum threshold -> high priority
    if (intervalForMetricIndex == (int) Constants.parts) {
        return Constants.priorityIntervals.get(intervalForMetricIndex).getInterval().right;
    }
    //get the interval in which this metric's value is found
    ImmutablePair<Double, Double> currentInterval = new ImmutablePair<>((double) 0, (double) 0);
    for (MetricIntervals metricInterval : metricIntervals) {
        if (metricInterval.getMetricName().equals(metric.getName())) {
            currentInterval = metricInterval.getIntervals().get(intervalForMetricIndex);
        }
    }
    //swap the mapping
    if (Config.getFormula(smell).get(metric.getName()).getComparison().equals("<")) {
        currentInterval = new ImmutablePair<>(currentInterval.right, currentInterval.left);
    }
    //get the target interval
    ImmutablePair<Double, Double> priorityInterval =
            Constants.priorityIntervals
            .get(intervalForMetricIndex)
            .getInterval();
    //return the scaled variable
    return Calculator.linearlyScaleVariableToNewInterval(
            Double.valueOf(metric.getValue()),
            currentInterval.left, currentInterval.right,
            priorityInterval.left, priorityInterval.right
    );
}
```

Figure 5.6: Compute Smell Intensity for Metric



```
org.argouml.cognitive.Designer.run();VERYHIGH;CINT:24&LOC:105&MND:8&CC:19&NOAV:34
org.argouml.core.propertypanels.ui.RowSelector.RowSelector(DefaultListModel, boolean, boolean);MEDIUM;CINT:43&LOC:207&MND:5&CC:35&NOAV:39
org.argouml.core.propertypanels.ui.SimpleListModel.propertyChange(PropertyChangeEvent);VERYHIGH;CINT:19&LOC:74&MND:9&CC:17&NOAV:16
org.argouml.core.propertypanels.ui.UMLModelElementListModel.propertyChange(PropertyChangeEvent);MEDIUM;CINT:11&LOC:102&MND:11&CC:18&NOAV:23
org.argouml.model.euml.ModelEventPumpEUMLImpl.getDebugInfo();LOW;CINT:16&LOC:37&MND:5&CC:8&NOAV:12
org.argouml.model.euml.ModelEventPumpEUMLImpl.unregisterListener(Object, EventListener, String[], Map<Object, List<Listener>>);VERYLOW;CINT:11&LOC:33&MND:5&CC:9&NOAV:8
org.argouml.model.euml.XmiReaderEUMLImpl.isUML14(Resource);MEDIUM;CINT:11&LOC:49&MND:9&CC:13&NOAV:13
org.argouml.model.mdr.UmlFactoryMDRImpl.delete(Object);MEDIUM;CINT:51&LOC:239&MND:6&CC:59&NOAV:15
org.argouml.moduleloader.ModuleLoader2.java.ModuleLoader2.doInternal(boolean);VERYLOW;CINT:10&LOC:79&MND:5&CC:20&NOAV:12
org.argouml.notation.providers.java.NotationUtilityJava.generateVisibility(Object);MEDIUM;CINT:19&LOC:60&MND:5&CC:18&NOAV:4
org.argouml.notation.providers.uml.AbstractMessageNotationUml.handlePredecessors(Object, List<List>, boolean);VERYLOW;CINT:10&LOC:51&MND:5&CC:9&NOAV:12
```

Figure 5.7: Example of Dispersed Coupling Smell CSV

5.2.2 Frontend

The frontend part is written in JavaFX, an easy-to-use and flexible library to write native GUI for applications. Adding new smells would require a moderate amount of code alteration, but such is the limitation of GUI applications.

Main Page

Upon starting the application, the user is greeted with a main screen shown in figure 5.8, where they can select to analyze a project or, if they already have an analysed project, jump to the project view.

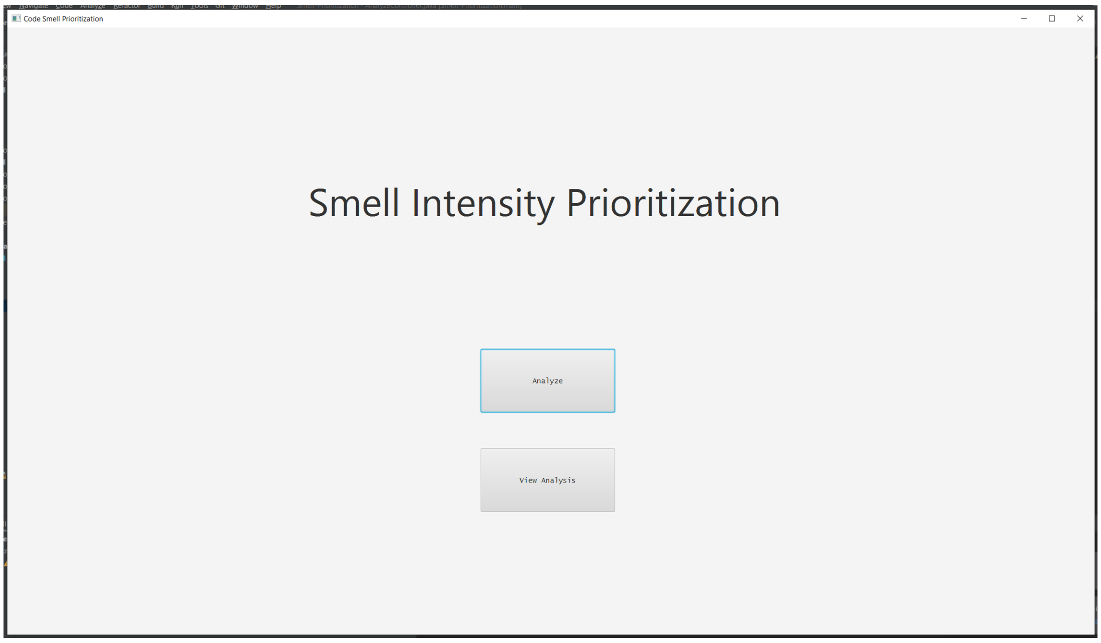
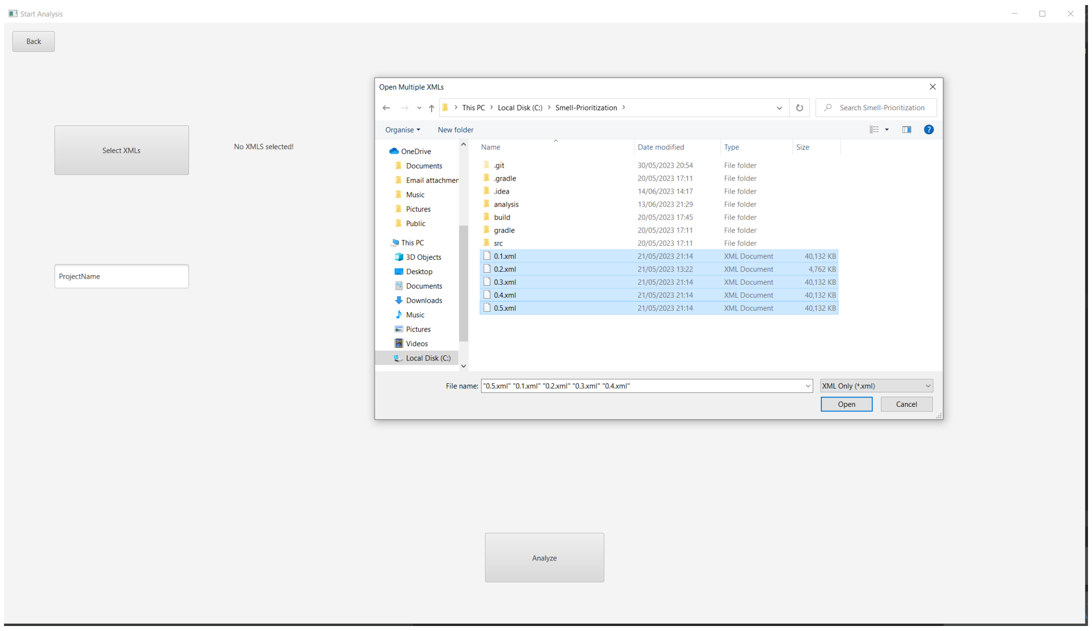


Figure 5.8: Main Screen

Analyze Project Window

Figure 5.9 shows that the user is directed to the Analyze Project View. If they wish to return to the main page, they can press the Back Button. Pressing the button selectXMLs, the user is prompted with a popup window where they can select multiple XMLs representing the various versions of the given project. A name can be set, and then press the Analyze button. When the analysis starts, as shown in figure 5.10, a loading bar appears, signifying the progress of the analysis. This is possible because a task is begun on a new thread so that the GUI thread is not blocked during the analysis. This task calls the backend to analyse every XML selected by the user.

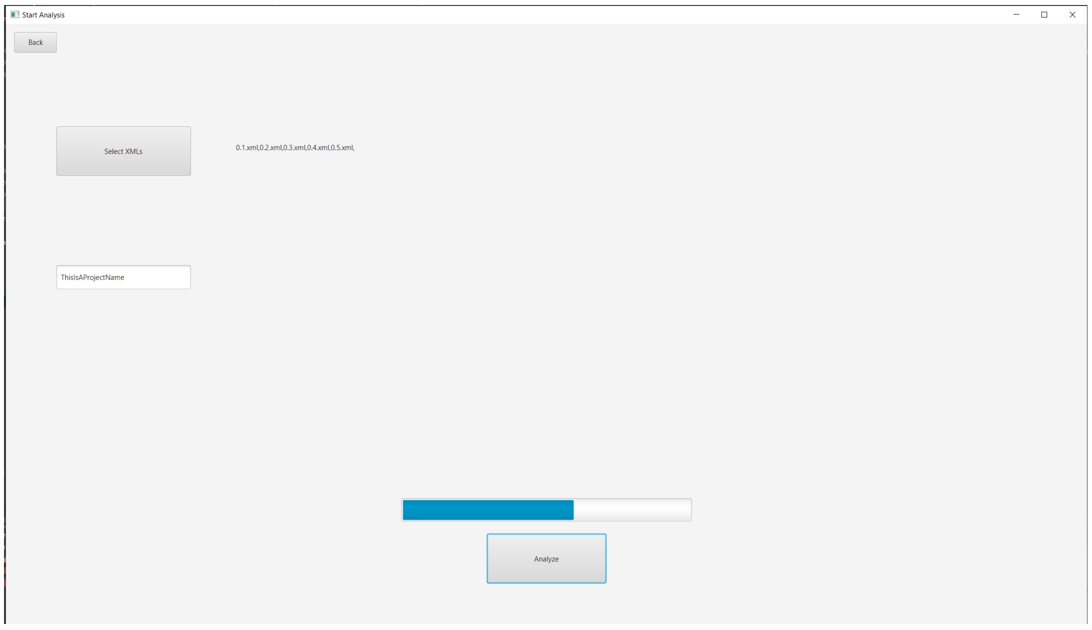


Name	Date modified	Type	Size
.git	30/05/2023 20:54	File folder	
.gradle	20/05/2023 17:11	File folder	
.idea	14/06/2023 14:17	File folder	
analysis	13/06/2023 21:29	File folder	
build	20/05/2023 17:45	File folder	
gradle	20/05/2023 17:11	File folder	
src	20/05/2023 17:11	File folder	
0.1.xml	21/05/2023 21:14	XML Document	40,132 KB
0.2.xml	21/05/2023 13:22	XML Document	4,762 KB
0.3.xml	21/05/2023 21:14	XML Document	40,132 KB
0.4.xml	21/05/2023 21:14	XML Document	40,132 KB
0.5.xml	21/05/2023 21:14	XML Document	40,132 KB

File name: "0.5.xml" "0.1.xml" "0.2.xml" "0.3.xml" "0.4.xml"

XML Only (*.xml)

Figure 5.9: Starting an Analysis on a Project



0.1.xml,0.2.xml,0.3.xml,0.4.xml,0.5.xml,

ThisIsAProjectName

Figure 5.10: Analyzing a Project

View Project Window

After the analysis is done, the user is immediately redirected to the View Project Window, as shown in figure 5.11. This window can also be accessed from the main window if the user has already had an analysis done. The Back button would return the user to the main page. The user has three options: select one, two or multiple project versions from the ordered list view. This would call the backend and read the corresponding CSVs, then open a new window for each action.

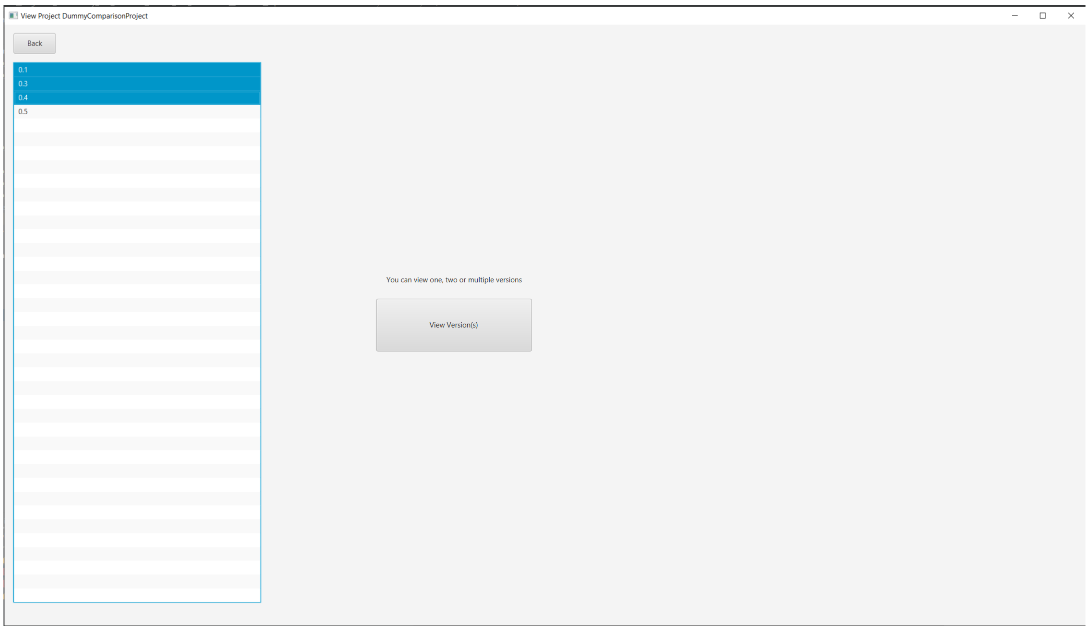
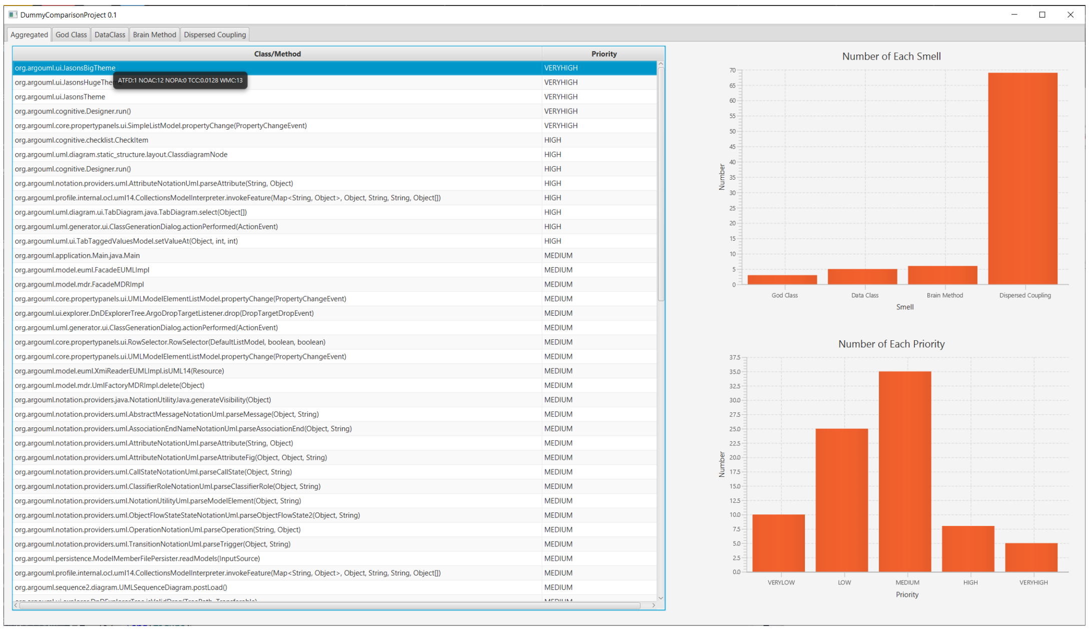

View Project DummyComparisonProject
Back
0.1
0.3
0.4
0.5
You can view one, two or multiple versions
View Version(s)

Figure 5.11: View Project Window

Single Version Window

In figure 5.12, notice several things: there are multiple tabs for an aggregated view of all smells and for viewing each smell. The table of smells is ordered by priority, and hovering over an entity reveals the metrics of that entity. There are also two graphs showing the number of smells for each smell type and the number of each priority. Clicking on another tab would show a table of only smells of that respective smell type and a graph for that data.



Class/Method	Priority
org.argouml.ui.JasonsBigTheme	VERYHIGH
org.argouml.ui.JasonsHugeThe	VERYHIGH
org.argouml.ui.JasonsTheme	VERYHIGH
org.argouml.cognitive.Designer.run()	VERYHIGH
org.argouml.core.propertypanels.ui.SimpleListModel.propertyChange(PropertyChangeEvent)	VERYHIGH
org.argouml.cognitive.checklist.CheckItem	HIGH
org.argouml.uml.diagram.static_structure.layout.ClassdiagramNode	HIGH
org.argouml.cognitive.Designer.run()	HIGH
org.argouml.notation.providers.uml.AttributeNotationUml.parseAttribute(String, Object)	HIGH
org.argouml.profile.internal.ocl.uml14.CollectionsModelInterpreter.invokeFeature(Map<String, Object>, Object, String, String, Object[])	HIGH
org.argouml.uml.diagram.ui.TabDiagram.java.TabDiagram.select(Object[])	HIGH
org.argouml.uml.generator.ui.ClassGenerationDialog.actionPerformed(ActionEvent)	HIGH
org.argouml.uml.ui.TabTaggedValuesModel.setValueAt(Object, int, int)	HIGH
org.argouml.application.Main.java.Main	MEDIUM
org.argouml.model.euml.FacadeEUMLImpl	MEDIUM
org.argouml.model.mdr.FacadeMDRImpl	MEDIUM
org.argouml.core.propertypanels.ui.UMLModelElementListModel.propertyChange(PropertyChangeEvent)	MEDIUM
org.argouml.ui.explorer.DnDExplorerTree.ArgoDropTargetListener.drop(DropTargetDropEvent)	MEDIUM
org.argouml.uml.generator.ui.ClassGenerationDialog.actionPerformed(ActionEvent)	MEDIUM
org.argouml.core.propertypanels.ui.RowSelector.RowSelector(DefaultListModel, boolean, boolean)	MEDIUM
org.argouml.core.propertypanels.ui.UMLModelElementListModel.propertyChange(PropertyChangeEvent)	MEDIUM
org.argouml.model.euml.XmiReaderEUMLImpl.isUML14(Resource)	MEDIUM
org.argouml.model.mdr.UmlFactoryMDRImpl.delete(Object)	MEDIUM
org.argouml.notation.providers.java.NotationUtilityJava.generateVisibility(Object)	MEDIUM
org.argouml.notation.providers.uml.AbstractMessageNotationUml.parseMessage(Object, String)	MEDIUM
org.argouml.notation.providers.uml.AssociationEndNameNotationUml.parseAssociationEnd(Object, String)	MEDIUM
org.argouml.notation.providers.uml.AttributeNotationUml.parseAttribute(String, Object)	MEDIUM
org.argouml.notation.providers.uml.AttributeNotationUml.parseAttributeFig(Object, Object, String)	MEDIUM
org.argouml.notation.providers.uml.CallStateNotationUml.parseCallState(Object, String)	MEDIUM
org.argouml.notation.providers.uml.ClassifierRoleNotationUml.parseClassifierRole(Object, String)	MEDIUM
org.argouml.notation.providers.uml.NotationUtilityUml.parseModelElement(Object, String)	MEDIUM
org.argouml.notation.providers.uml.ObjectFlowStateStateNotationUml.parseObjectFlowState2(Object, String)	MEDIUM
org.argouml.notation.providers.uml.OperationNotationUml.parseOperation(String, Object)	MEDIUM
org.argouml.notation.providers.uml.TransitionNotationUml.parseTrigger(Object, String)	MEDIUM
org.argouml.persistence.ModelMemberFilePersister.readModels(InputSource)	MEDIUM
org.argouml.profile.internal.ocl.uml14.CollectionsModelInterpreter.invokeFeature(Map<String, Object>, Object, String, String, Object[])	MEDIUM
org.argouml.sequence2.diagram.UMLSequenceDiagram.postLoad()	MEDIUM
org.argouml.ui.explorer.DnDExplorerTree.isValidDrag(TreePath, Transferable)	MEDIUM

Figure 5.12: View Single Version Window

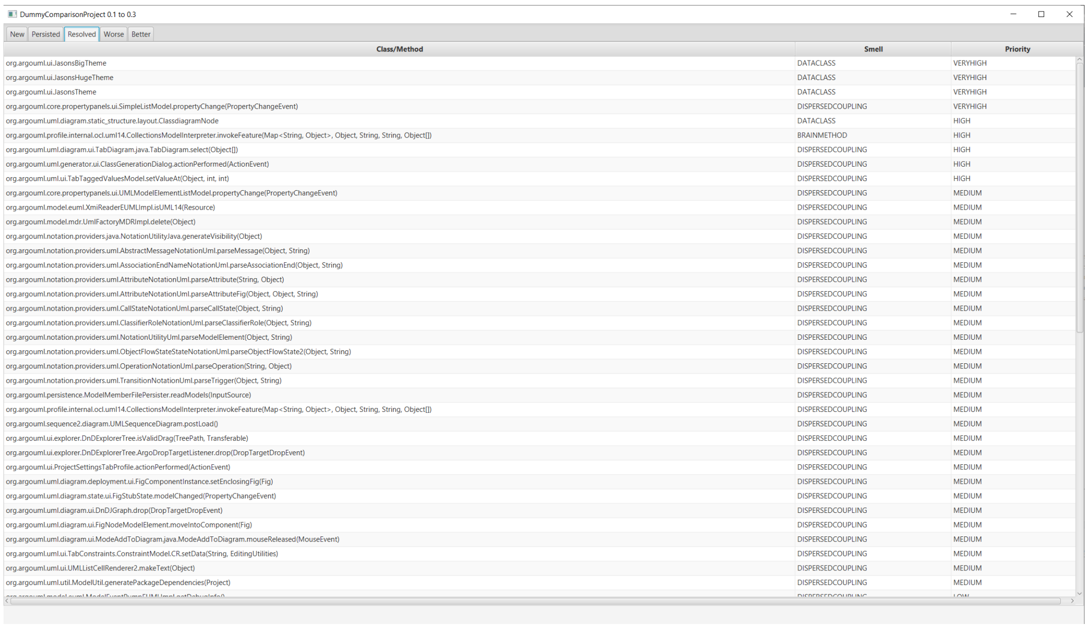
Two Versions Comparison Window

Figure 5.13 shows the window that opens if two projects are selected. There are multiple tabs:

- New - for new smells from one version to another
- Persisted - smells that are present in both versions
- Resolved - smells that are present in the earlier version but not in the newer version
- Worse - smells that are present in both versions but are of a HIGHER priority in the newer version
- Better - smells that are present in both versions but are of a LOWER priority in the newer version

In figure 5.13, the Resolved tab is selected, and it shows a table ordered by priority, with each line containing the class/method name, the smell type, and its priority. Again, hovering over an entity would reveal the metrics of that particular class/method.

This View is packed with valuable information, as the user can quickly see the effect of their refactoring and maintenance policy from one version to another. As this view opens in a new window, the user can have multiple two versions comparisons shown at the same time.



Class/Method	Smell	Priority
org.argouml.ui.JasonsBigTheme	DATACLASS	VERYHIGH
org.argouml.ui.JasonsHugeTheme	DATACLASS	VERYHIGH
org.argouml.ui.JasonsTheme	DATACLASS	VERYHIGH
org.argouml.core.propertypanels.ui.SimpleListModel.propertyChange(PropertyChangeEvent)	DISPERSEDCOUPLING	VERYHIGH
org.argouml.uml.diagram.static_structure.layout.ClassdiagramNode	DATACLASS	HIGH
org.argouml.profile.internal.ocl.uml14.CollectionsModelInterpreter.invokeFeature(Map<String, Object>, Object, String, String, Object[])	BRAINMETHOD	HIGH
org.argouml.uml.diagram.ui.TabDiagram.java.TabDiagram.select(Object[])	DISPERSEDCOUPLING	HIGH
org.argouml.uml.generator.ui.ClassGenerationDialog.actionPerformed(ActionEvent)	DISPERSEDCOUPLING	HIGH
org.argouml.uml.ui.TabTaggedValuesModel.setValueAt(Object, int, int)	DISPERSEDCOUPLING	HIGH
org.argouml.core.propertypanels.ui.UMLModelElementListModel.propertyChange(PropertyChangeEvent)	DISPERSEDCOUPLING	MEDIUM
org.argouml.model.euml.XmiReaderEUMLImpl.isUML14(Resource)	DISPERSEDCOUPLING	MEDIUM
org.argouml.model.mdr.UmlFactoryMDRImpl.delete(Object)	DISPERSEDCOUPLING	MEDIUM
org.argouml.notation.providers.java.NotationUtilityJava.generateVisibility(Object)	DISPERSEDCOUPLING	MEDIUM
org.argouml.notation.providers.uml.AbstractMessageNotationUml.parseMessage(Object, String)	DISPERSEDCOUPLING	MEDIUM
org.argouml.notation.providers.uml.AssociationEndNameNotationUml.parseAssociationEnd(Object, String)	DISPERSEDCOUPLING	MEDIUM
org.argouml.notation.providers.uml.AttributeNotationUml.parseAttribute(String, Object)	DISPERSEDCOUPLING	MEDIUM
org.argouml.notation.providers.uml.AttributeNotationUml.parseAttributeFig(Object, Object, String)	DISPERSEDCOUPLING	MEDIUM
org.argouml.notation.providers.uml.CallStateNotationUml.parseCallState(Object, String)	DISPERSEDCOUPLING	MEDIUM
org.argouml.notation.providers.uml.ClassifierRoleNotationUml.parseClassifierRole(Object, String)	DISPERSEDCOUPLING	MEDIUM
org.argouml.notation.providers.uml.NotationUtilityUml.parseModelElement(Object, String)	DISPERSEDCOUPLING	MEDIUM
org.argouml.notation.providers.uml.ObjectFlowStateStateNotationUml.parseObjectFlowState2(Object, String)	DISPERSEDCOUPLING	MEDIUM
org.argouml.notation.providers.uml.OperationNotationUml.parseOperation(String, Object)	DISPERSEDCOUPLING	MEDIUM
org.argouml.notation.providers.uml.TransitionNotationUml.parseTrigger(Object, String)	DISPERSEDCOUPLING	MEDIUM
org.argouml.persistence.ModelMemberFilePersister.readModels(InputSource)	DISPERSEDCOUPLING	MEDIUM
org.argouml.profile.internal.ocl.uml14.CollectionsModelInterpreter.invokeFeature(Map<String, Object>, Object, String, String, Object[])	DISPERSEDCOUPLING	MEDIUM
org.argouml.sequence2.diagram.UMLSequenceDiagram.postLoad()	DISPERSEDCOUPLING	MEDIUM
org.argouml.ui.explorer.DnDExplorerTree.isValidDrag(TreePath, Transferable)	DISPERSEDCOUPLING	MEDIUM
org.argouml.ui.explorer.DnDExplorerTree.ArgoDropTargetListener.drop(DropTargetDropEvent)	DISPERSEDCOUPLING	MEDIUM
org.argouml.ui.ProjectSettingsTabProfile.actionPerformed(ActionEvent)	DISPERSEDCOUPLING	MEDIUM
org.argouml.uml.diagram.deployment.ui.FigComponentInstance.setEnclosingFig(Fig)	DISPERSEDCOUPLING	MEDIUM
org.argouml.uml.diagram.state.ui.FigStubState.modelChanged(PropertyChangeEvent)	DISPERSEDCOUPLING	MEDIUM
org.argouml.uml.diagram.ui.DnDJGraph.drop(DropTargetDropEvent)	DISPERSEDCOUPLING	MEDIUM
org.argouml.uml.diagram.ui.FigNodeModelElement.moveIntoComponent(Fig)	DISPERSEDCOUPLING	MEDIUM
org.argouml.uml.diagram.ui.ModeAddToDiagram.java.ModeAddToDiagram.mouseReleased(MouseEvent)	DISPERSEDCOUPLING	MEDIUM
org.argouml.uml.ui.TabConstraints.ConstraintModel.CR.setData(String, EditingUtilities)	DISPERSEDCOUPLING	MEDIUM
org.argouml.uml.ui.UMLListCellRenderer2.makeText(Object)	DISPERSEDCOUPLING	MEDIUM
org.argouml.uml.util.ModelUtil.generatePackageDependencies(Project)	DISPERSEDCOUPLING	MEDIUM
org.argouml.model.euml.ModelEventPumpEUMLImpl.getDebugInfo()	DISPERSEDCOUPLING	LOW

Figure 5.13: View Two Versions Comparison Window

Multiple Versions Comparison Window

Selecting more than two versions opens a new window to compare them.

The Total Smells Tab, shown in figure 5.14, shows a graph for the total number of smells of all types per version.

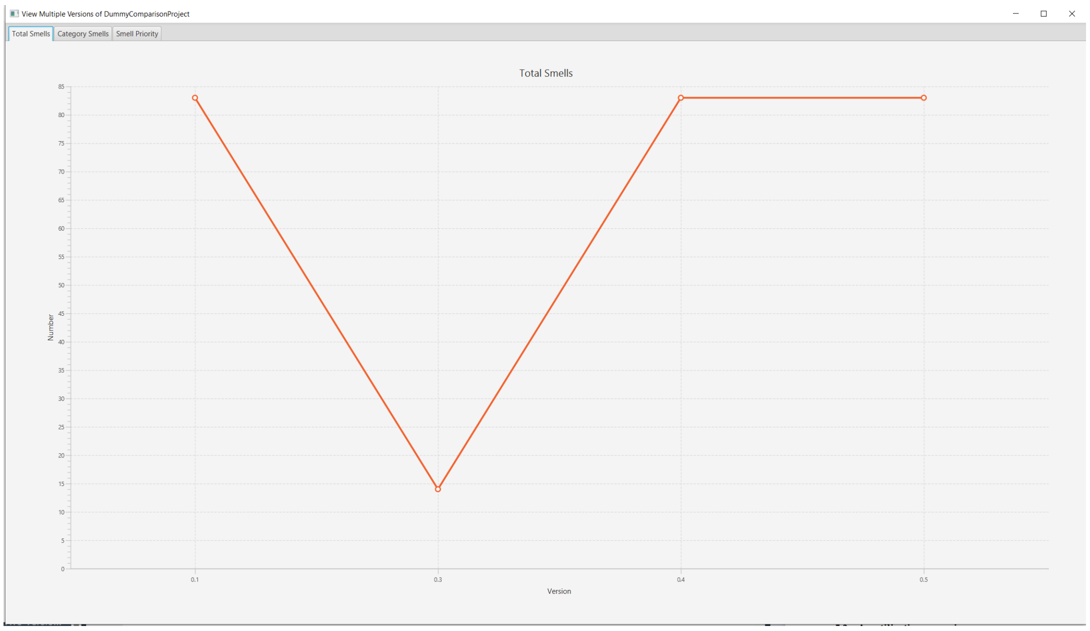


Figure 5.14: View Multiple Versions Comparison Window, Total Smells Tab

The Smell Category Tab, shown in figure 5.15, shows a graph comparing the number of different types of smells present in each selected version.

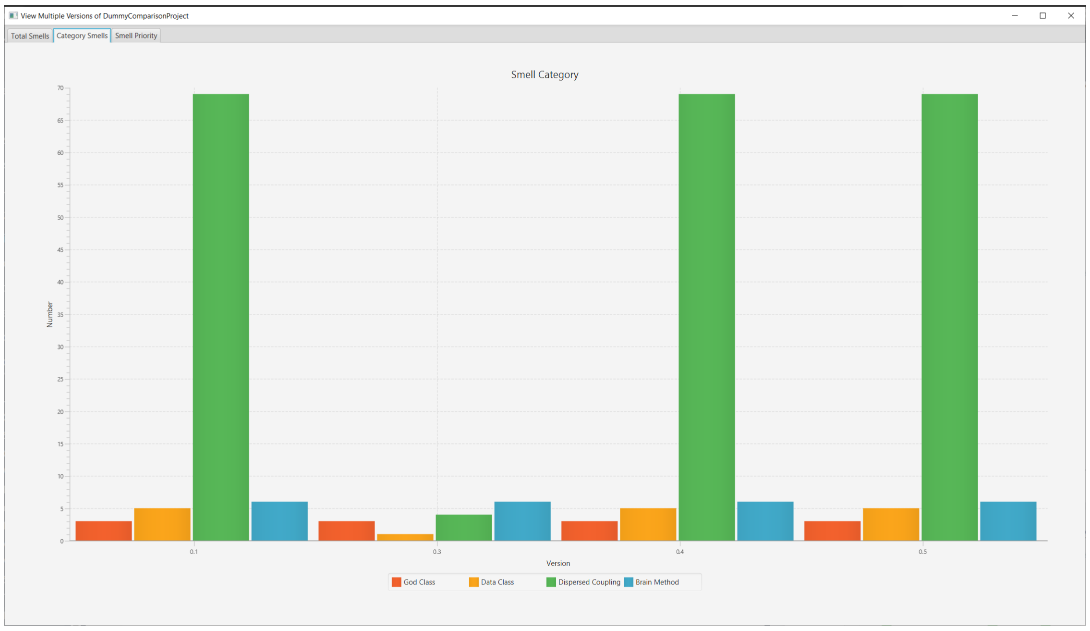


Figure 5.15: View Multiple Versions Comparison Window, Smell Category Tab

The Smell Priority Tab, shown in figure 5.16, shows a graph comparing the number of different smell priorities in each selected version.

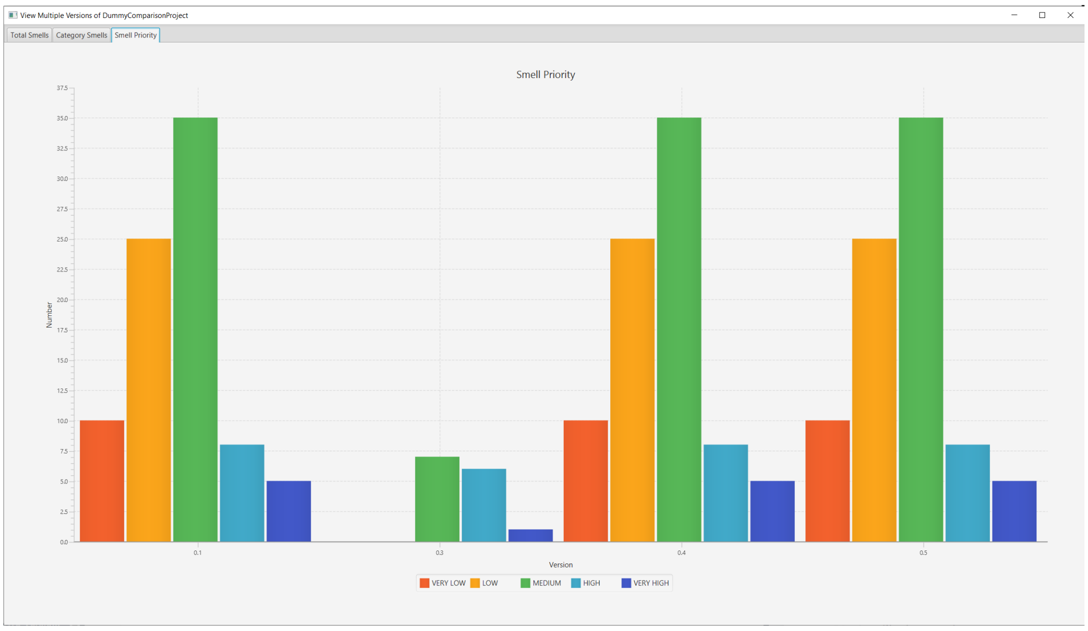


Figure 5.16: View Multiple Versions Comparison Window, Smell Priority Tab

Chapter 6

Analysis

In order to prove the benefits of our tool, we will consider a use case: four versions of ArgoUML, an open-source Java project, with the latest version having 177278 lines of code.

Let us start by looking at the graphs for the Multiple Versions Comparisons and see what kind of trends and outliers can be spotted to see the validity of how the tool could help a developer.

In figure 6.1, the Total Priority tab clearly shows a dip in smells at version 0.3, which indicates that various refactorings were done for this version, but the trend is going up in the subsequent versions, which shows why refactorings and code quality focused commits should be a constant throughout the development process: new smells add up, and it is harder to do big, sporadic refactorings, rather than smaller, constant ones. This is where the tool developed in this paper can help reveal new smells to be tackled as soon as they appear. This graph is not enough, though, as the new smells might be of lower priority, while the solved ones might be of higher priority, so more information is needed to pass correct judgement.

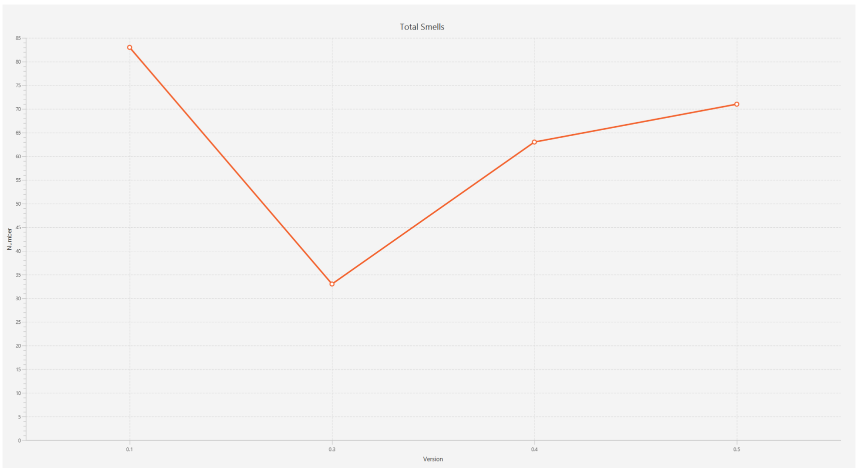


Figure 6.1: View Multiple Versions Comparison Window, Total Priority Tab

Figure 6.3 shows the different categories present throughout the versions. The clear outlier here is the Dispersed Coupling smell. This might indicate that developers do not use design patterns such as the Factory, Builder or Observer patterns or that proper encapsulation is not applied, such as having well-defined interfaces and abstractions that hide implementation details from their consumers. Developers can draw those conclusions from these graphs to see where they might improve themselves, as much as the code base. On a more positive note, we can observe that the Brain Method smell was a constant in versions 0.1, 0.3 and 0.4, but there is a dip in version 0.5, suggesting that developers have figured out how to tackle this type of smell. It is clear that this tool helps with both showing what went wrong, but also what went right.

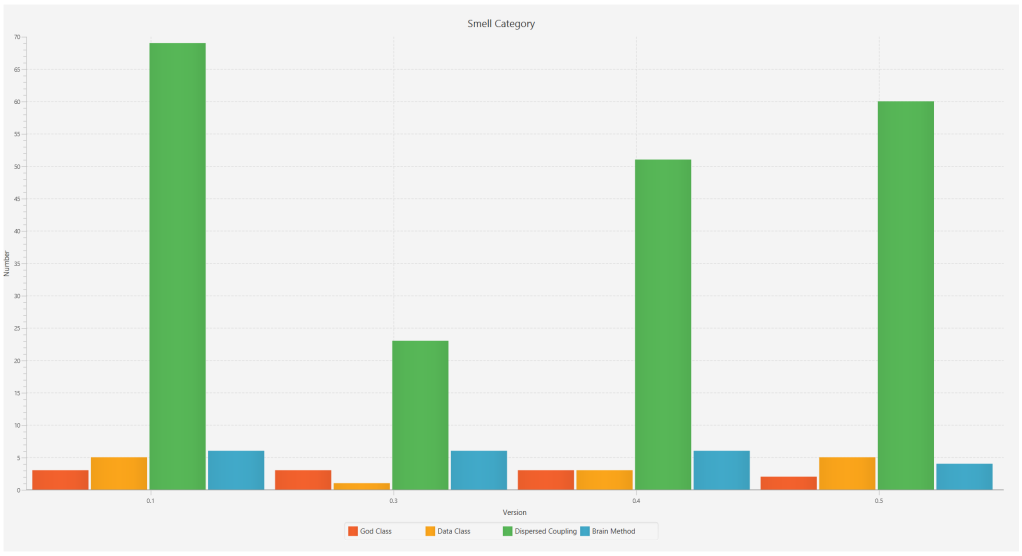


Figure 6.2: View Multiple Versions Comparison Window, Smell Category Tab

Last but not least, the Smell Priority tab reveals that most smells in the analysed application are of MEDIUM/LOW priority, which suggests that the code is of generally high quality as LOW, and VERY-LOW priorities can include false positives. A worrying aspect is that from version 0.3 to 0.5, the VERY-HIGH high priority is on the rise, and as explained in chapter 5, those are considered to be critical, as all the metrics in that smell have passed the VERY-HIGH threshold. This could help developers by alerting them that refactoring is crucial for the subsequent development steps to keep their application maintainable.

Another problem which reveals itself is that HIGH priority items remain fairly constant, so coupling this with the VERY-HIGH priority problem explained earlier, one could draw the conclusion that in the 6.1 graph, the dip mostly represents MEDIUM to VERY-LOW priorities, as the most significant ones were not sufficiently dealt with.

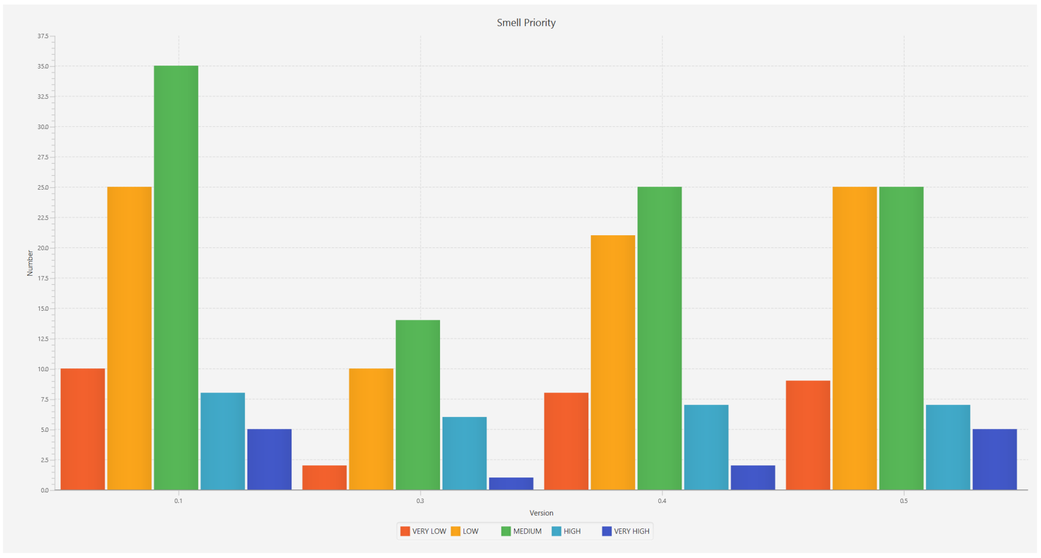
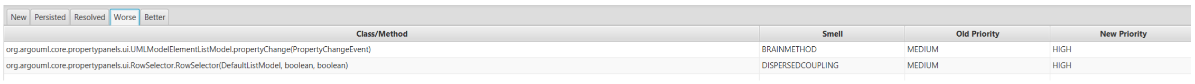


Figure 6.3: View Multiple Versions Comparison Window, Smell Priority Tab

One more thing to point out in this project would be that, on the Two Versions Comparison Window, seen in figure 6.4, the Worse tab reveals that two smells were made of a higher priority in the newer version, which should be unacceptable as a quality gate. The appearance of new smells might be deemed acceptable in the short term when new features need to be quickly delivered, but worsening smells should usually not be merged with the master branch.



Class/Method	Smell	Old Priority	New Priority
org.argouml.core.propertypanels.ui.UMLModelElementListModel.propertyChange(PropertyChangeEvent)	BRAINMETHOD	MEDIUM	HIGH
org.argouml.core.propertypanels.ui.RowSelector.RowSelector(DefaultListModel, boolean, boolean)	DISPERSEDCOUPLING	MEDIUM	HIGH

Figure 6.4: View Two Versions Comparison Window, Worse Tab

Chapter 7

Conclusions and Further Work

7.1 Conclusions

The thesis aim was to address software quality by exploring methods in which code smells may be identified and classified based on metrics defined at the level of source code. We provide a proof of concept in which we selected a set of meaningful code metrics and a set of code smells with high impact and showed how this approach can be applied. Practitioners can benefit from our findings as they might better control the distribution and evolution of code smells between versions and also evaluate the type of code smells with higher relevance to the project. From a research perspective, this is a real experiment based on previous relevant results in the literature.

In conclusion, the tool provided in this paper would be of much help to help the maintainability and quality of large projects and for analysing good and bad trends in design practices.

The flexibility provided by the configuration file allows users to define their tolerance thresholds to different smells and metrics, and the code's modularity allows for changes during the different analysis steps.

Both comparison windows provide easy access to trends and outliers, while the single version view enables quick access to all the particularities of a single version while highlighting the urgent issues that need to be addressed.

The CSV saving method allows users to use the backend as an API to integrate into workflows while also being a human-readable resource that can be used on its own. It can also serve archival purposes to help with trend analysis in the future.

7.2 Further Work

There are things that can be improved in future application versions, though.

One such thing is the configuration file. Although it is a strong point of the tool, I think it can be improved in several ways to make adding new smells a seamless process. At the moment, there is a need to add a trivial amount of lines of code to make that happen, but it can be improved to the point of that not being a necessity.

Speaking of code smells, there are more than 40 metrics that MetricsTree provides and are not used in the current version of the tool. I think more could be leveraged, and new smells could be added to improve the range of code quality coverage that this tool provides. Smells tackled here are from the [LM06] book, but it would be interesting to see how smells from other sources compare and analyse what benefits they would bring to this tool.

Last but not least, as seen in the Analysis Chapter 6, the multiple versions comparison window is incredibly interesting for noticing trends and forming plans on how to improve both the code base further, also as a professional developer, so adding more types of comparisons between versions would be an essential way of leveraging the already present data that this tool collects.

7.3 How I Improved

Developing this tool and writing the accompanying paper has had a profound effect on me as a professional software engineer. Reading books such as [LM06], [Fow99] and [MC09], as well as the various articles cited in this work in preparation for writing the paper, has opened my eyes to the importance of constant code refactoring by continuously checking for code smells. I have given many reasons why refactoring is a crucial step in the development cycle, and I will seek to implement what I have learned in my professional work in the future.

Bibliography

[AFFZY15] Francesca Arcelli Fontana, Vincenzo Ferme, Marco Zanoni, and Aiko Yamashita. Automatic metric thresholds derivation for code smell detection. In *2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics*, pages 44–53, 2015.

[AFP20] Kalev Alpernas, Yotam M. Y. Feldman, and Hila Peleg. The wonderful wizard of loc: Paying attention to the man behind the curtain of lines-of-code metrics. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2020, page 146–156, New York, NY, USA, 2020. Association for Computing Machinery.

[BB23] Vadim Burakov and Alexey Borovkov. Advanced metric analysis tool for java source code. *INFORMATION AND CONTROL SYSTEMS*, 1(122):17–28, 2023.

[BK95] James M. Bieman and Byung-Kyoo Kang. Cohesion and reuse in an object-oriented system. page 259–262, New York, NY, USA, 1995. Association for Computing Machinery.

[BTP12] Kaushal Bhatt, Vinit Tarey, and Pushpraj Patel. Analysis of source lines of code(sloc) metric. *IJETAE*, 2, 04 2012.

[Car] Carlos Schults. What Is a God Class and Why Should We Avoid It? `https://linearb.io/blog/what-is-a-god-class/`. Online; accessed 05 May 2023.

[CK94] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[Eva] Evan SooHoo. Did Elon Musk Really Fire People Using Lines Of Code As His Metric? `https://evan-soohoo.medium.com/did-elon-musk-really-fire-people-using-lines-of-code-as-his-me` Online; accessed 01 May 2023.

[FAFZR15] Vincenzo Ferme, Francesca Arcelli Fontana, Marco Zanoni, and Riccardo Roveda. Towards a prioritization of code debt: A code smell intensity index. 10 2015.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[IEE94] Ieee standard for a software quality metrics methodology. *IEEE Std 1061-1998*, pages 1–32, 1994.

[LM06] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer Berlin Heidelberg New York, Germany, 2006.

[MC09] Robert C. Martin and James O. Coplien. *Clean code: a handbook of agile software craftsmanship*. Prentice Hall, Upper Saddle River, NJ [etc.], 2009.

[McC76] T.J. McCabe1976. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[Tom] Tom Preston-Werner. Semantic Versioning 2.0.0. `https://semver.org/`. Online; accessed 20 May 2023.

[vEM02] E. van Emden and L. Moonen. Java quality assurance by detecting code smells. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, pages 97–106, 2002.