Concrete Architecture Evaluation Report (T1)

Delivery Date: Apr 30, 2023

Team Members:
Yunus Emre Terzi
Gamze Ergin

1. OSS PRODUCT OVERVIEW

Apache SkyWalking

Table 1. Description of the OSS product

Name of OSS Product:	*Apache SkyWalking*
Description of OSS product:	This is an open-source APM(application performance monitor) system. It is designed for: ● microservices, ● cloud-native, ● container-based architectures. It includes monitoring, tracing, and diagnosing capabilities for a distributed system in Cloud Native architecture.
URL of OSS product:	https://github.com/apache/skywalking
Size of OSS product (KLOC):	204.133

2. TOOLS USED FOR ANALYSIS

Table 2. The list of tools used for analysis

Tool Name	**Purpose of Use**	**URL**
MetricsReloaded	● *to obtain maintainability metrics,* ● *to create dependency graphs.*	https://plugins.jetbrains.com/plugin/93-metricsreloaded
MetricsTree	● *to obtain maintainability metrics,* ● *to create dependency graphs.*	https://plugins.jetbrains.com/plugin/13959-metricstree

JDeodorant	● to obtain code smells	https://marketplace.eclipse.org/content/jdeodorant
Understand	● *to obtain maintainability metrics,* ● *to create dependency graphs.*	https://www.scitools.com/?pk_vid=ddb297d31fb25c701682038687e99189
PMD	● to obtain code smells	https://plugins.jetbrains.com/plugin/1137-pmd

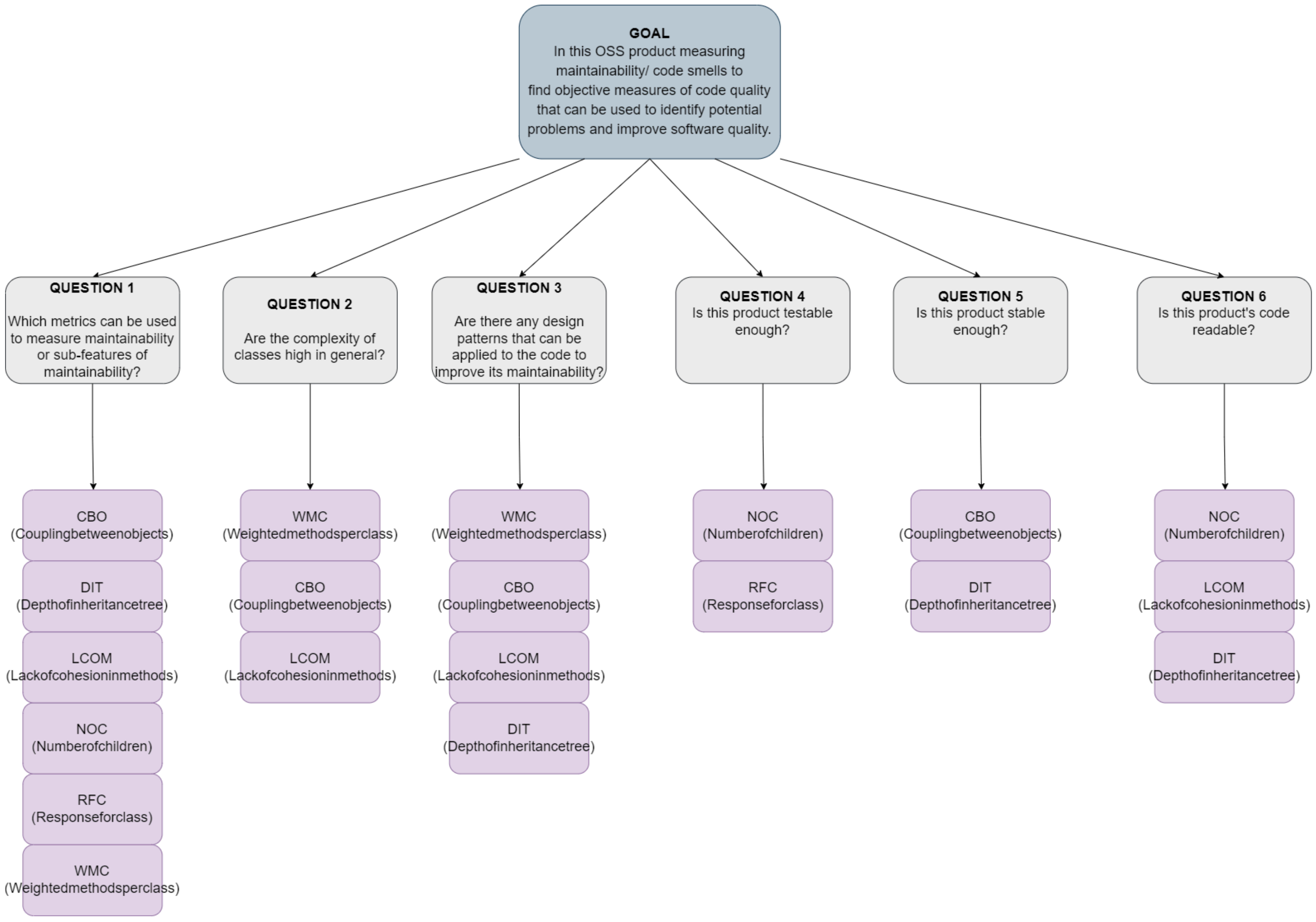
3. GQM TREE FOR EVALUATING OSS MAINTAINABILITY

A GQM tree is a hierarchical model that links software development goals to specific questions and metrics that can be used to measure progress toward those goals. The model starts with high-level goals, such as improving software quality or increasing development efficiency, and breaks them down into more specific sub-goals, questions, and metrics. In this OSS the goal, questions, and metrics are shown in Table 3.

Table 3.

Goal:	In this OSS product measure maintainability/ code smells to find objective measures of code quality that can be used to identify potential problems and improve software quality.	
List of questions to answer while evaluating the goal:		*List of metrics used to answer the question:*
Q(1):	Which metrics can be used to measure maintainability or sub-features of maintainability?	● CBO ● DIT ● LCOM ● NOC ● RFC ● WMC
Q(2):	Are the complexity of classes high in general?	● WMC ● CBO ● LCOM
Q(3):	Are there any design patterns that can be applied to the code to improve its maintainability?	● WMC ● DIT ● CBO ● LCOM
Q(4):	Is this product testable enough?	● NOC ● RFC

Q(5):	Is this product stable enough?	● CBO ● DIT
Q(6):	Is this product’s code readable?	● NOC ● LCOM ● DIT


GOAL
In this OSS product measuring maintainability/ code smells to find objective measures of code quality that can be used to identify potential problems and improve software quality.
QUESTION 1
Which metrics can be used to measure maintainability or sub-features of maintainability?
QUESTION 2
Are the complexity of classes high in general?
QUESTION 3
Are there any design patterns that can be applied to the code to improve its maintainability?
QUESTION 4
Is this product testable enough?
QUESTION 5
Is this product stable enough?
QUESTION 6
Is this product's code readable?
CBO (Couplingbetweenobjects)
DIT (Depthofinheritancetree)
LCOM (Lackofcohesioninmethods)
NOC (Numberofchildren)
RFC (Responseforclass)
WMC (Weightedmethodsperclass)
WMC (Weightedmethodsperclass)
CBO (Couplingbetweenobjects)
LCOM (Lackofcohesioninmethods)
WMC (Weightedmethodsperclass)
CBO (Couplingbetweenobjects)
LCOM (Lackofcohesioninmethods)
DIT (Depthofinheritancetree)
NOC (Numberofchildren)
RFC (Responseforclass)
CBO (Couplingbetweenobjects)
DIT (Depthofinheritancetree)
NOC (Numberofchildren)
LCOM (Lackofcohesioninmethods)
DIT (Depthofinheritancetree)

4. DESCRIPTION OF SOFTWARE METRICS

Software metrics refer to quantifiable measures that are used to assess the quality of software, the efficiency of the software development process, and the performance of software products.

Table 4. List of software metrics used in the analysis

Metric Name	Type	Source	Formula (or description) by Source
WMC	*C&K*	https://www.aivosto.com/project/help/pm-oo-ck.html	the number of methods defined in the class.
DIT	C&K	https://www.aivosto.com/project/help/pm-oo-ck.html	maximum inheritance path from the class to the root class.
NOC	C&K	https://www.aivosto.com/project/help/pm-oo-ck.html	the number of immediate sub-classes of a class.
CBO	C&K	https://www.aivosto.com/project/help/pm-oo-ck.html	the number of classes to which a class is coupled.
RFC	C&K	https://www.spinellis.gr/sw/ckjm/doc/metric.html#:~:text=The%20metric%20called%20the%20response,is%20invoked%20for%20that%20object	the number of different methods that can be executed when an object of that class receives a message.
LCOM	C&K	https://www.spinellis.gr/sw/ckjm/doc/metric.html#:~:text=The%20metric%20called%20the%20response,is%20invoked%20for%20that%20object	the number of sets of methods in a class that is not related through the sharing of some of the class's fields.

5. EVALUATION OF MAINTAINABILITY

5.1. Answer to the question "Which metrics can be used to measure maintainability or sub-features of maintainability?"

We will use C&K metrics for our answers. But first, we should define what quantitative is good for each metric.

- ***WMC:*** There should be a maximum of 10% of classes that have more than 24 methods.

  High WMC:
  - ***Increased likelihood of faults:*** Classes with a high WMC are more likely to contain bugs and errors.
  - ***Limited reuse:*** Classes with many methods are often more application-specific, which limits their potential for reuse.
  - ***Higher development and maintenance effort:*** The more methods a class has, the

more time and effort are required to develop and maintain it.
    - ***Increased impact on derived classes:*** Classes with a high WMC can have a significant impact on derived classes since derived classes inherit some of the methods from the base class.

  Low WMC:
    - ***Overuse of inheritance:*** There may be a tendency to create many small, specialized classes, which can lead to an overuse of inheritance.
    - ***Reduced cohesion:*** If a class has too few methods, it may lack cohesion and be responsible for too many different tasks.
    - ***Duplication of code:*** If methods are separated into different small classes that can cause this problem.
    - ***Increased method call overhead:*** There may be an increase in method call overhead, which can lead to performance issues in the software system.

  DIT: It should be 5 or less.

  High DIT:
    - ***Increased complexity***
    - ***Decreased maintainability:*** A high DIT can make the code harder to modify because changes to a base class can have ripple effects throughout the entire inheritance tree.
    - ***Reduced flexibility:*** It can become difficult to add new functionality without creating a new subclass.
    - ***Increased coupling:*** If many classes inherit from a common superclass, there may be a risk of increased coupling between those classes. This can make the code more difficult to test and can lead to potential errors.

  Low DIT:
    - ***Code duplication***
    - ***Decreased modularity:*** A low DIT can lead to a less modular design, because there may be fewer opportunities to group related functionality together in a common superclass.

- **NOC:** It should be a value between 1 and 3.

  High NOC:
    - ***High reuse of the base class:*** In general, it is preferable for a base class to have a high NOC since it encourages software reuse and lowers development effort. But also the changes that have been made in base classes can have bigger impacts in the child classes. It’s better to pay more attention to maintenance and testing.
    - ***Complexity at the top of the class hierarchy:*** A high NOC potentially influences a large number of descendant classes. This can be a sign of poor design, and a redesign may be required to break down the class into smaller, more focused ones.
    - ***Improper abstraction of the parent class:*** If the sub-classes don't share a common abstraction with the parent class, then the inheritance relationship may not be meaningful, and the high NOC can indicate misuse of inheritance.

    - ***Misuse of sub-classing:*** If a high NOC is the result of a lot of unrelated sub-classes in order to reduce the number of sub-classes and enhance the design, it could be required to group relevant classes.

  Low NOC:
    - ***Lack of reuse***
    - ***Tight coupling:*** A base class and its customers may be tightly coupled if there are not enough subclasses. This makes the system more difficult to manage because changes to the base class can significantly affect the client classes.
    - ***Reduced flexibility:*** It could be challenging to add new functionality or adjust to changing requirements if there aren't enough sub-classes.
    - ***Overgeneralization:*** Low NOC can cause overly generic design and lack sufficient attention to particular use cases.

- **CBO:** It should be less than 14.

  High CBO:
    - ***Reduced maintainability***
    - ***Reduced reusability***: Because a class is closely tied to its dependencies, high coupling between classes may prevent code from being reused.
    - ***Increased complexity***
    - ***Reduced flexibility***
    - ***Increased likelihood of errors:*** As changes to one class may introduce flaws in other dependent classes, high CBO can increase the likelihood of errors in the system.

  Low CBO:
    - ***High cohesion:*** A class with a low CBO is likely to be highly cohesive and to have a distinct role to play in the system. The class may become simpler to comprehend, test, and maintain as a result.
    - ***Limited reuse:*** A low CBO can also be a sign that there aren't many options for a class to be reused elsewhere in the system.

- **RFC:** It should be a value between 6 and 36.

  High RFC:
    - ***High complexity***
    - ***High coupling:*** Classes having a lot of cross-class interaction typically have higher RFC values. High coupling between classes may result from this, making it more challenging to maintain and modify the code.
    - ***Poor design:*** It may suggest that a class is trying to do too much and that its responsibilities should be split into smaller, more focused classes.

  Low RFC:
    - ***Lack of interaction:*** A class may not communicate with many other classes in the system if its RFC value is low. This can mean that the class isn't serving its original function or isn't being used to its full potential.
    - ***Poor design:*** A class with a low RFC value can be poorly designed, have few

functionalities, or have little room for reuse.
- ***Limited Impact:*** Changes to a class with a low RFC value may have a minimal effect on the system as a whole, which means that the system as a whole will probably not be significantly impacted by the class's changes.

- **LCOM:** It should be a value between 1 and 3.

  High LCOM:
  - ***Increased complexity:*** This can be caused because methods are not well-organized.
  - ***Maintenance issues:*** When LCOM is high the lack of clarity in each method's purpose might make it challenging to maintain and modify the code.
  - ***Reduced reusability*****:** When methods are not well-organized, it can limit the reusability of the class.
  - ***Increased coupling:*** A high LCOM can lead to increased coupling between methods, which can make the class more tightly coupled and harder to change.
  - **Increased testing effort:** It can increase the testing effort required to ensure that the class is working correctly because methods are not well-organized.

  Low LCOM:
  - ***Tightly coupled classes:*** This makes the system less flexible and harder to maintain.
  - ***Code redundancy*****:** This problem can lead to larger and more complex classes.

WMC	DIT	NOC	CBO	RFC	LCOM
There should be a maximum of 10% of classes that have more than 24 methods.	It should be 5 or less.	It should be a value between 1 and 3.	It should be less than 14.	It should be a value between 6 and 36.	It should be a value between 1 and 3.

5.2. Answer to the Question "Are the complexity of classes high in general?"

Software complexity is a natural byproduct of the functional complexity that the code is attempting to enable. When a system becomes more complex, it becomes harder to understand how different parts of software interact with each other. Thus it becomes more difficult the maintain. So lower complexity indicates better maintainability.

We must investigate the complexity of the overall project. The complexity can be evaluated with all of the C&K metrics, but we choose critical three of them; WMC, CBO, and LCOM. Since we are working on the overall project, the average of metrics taken.

Our metric values,

- WMC: 5,76 ***(There should be a maximum of 10% of classes that have more than 24 methods.)***
- CBO: 7,85 ***(<14)***

- LCOM: 1,54 ***(<3)***

In our example, we have 85 classes that they have more than 24 methods and 1933 classes in total. So less than 10% of classes have more than 24 methods and it means our WMC value is good.

Since our CBO value is less than 14, we can say that we have a good CBO value.

Since our LCOM value is less than 3, we can say that we have a good LCOM value.

Since all metrics have good value, we can conclude that our project does not include complex classes in general.

5.3. Answer the Question "Are there any design patterns that can be applied to the code to improve its maintainability?"

Design patterns provide a common set of practices that can be used to improve quality and decrease complexity. Since it is a common set of practices, it makes it much easier to understand how software design work. So if there is a design pattern that can be applied to the code, it will make the product more maintainable.

If we want to improve our code we should look at the WMC, DIT, CBO, and LCOM metrics.

For example, if our value of WMC is higher than the threshold we can make some changes in our design pattern for reducing code complexity in our classes. These can be dividing classes into smaller classes with more clear descriptions of their use in the product. Implementing the Single Responsibility Principle (SRP) is another useful option. Also, we can use inheritance. But in our code, our metrics values are:

- WMC: 5,76 ***(There should be a maximum of 10% of classes that have more than 24 methods.)***
- CBO: 7,85 ***(<14)***
- LCOM: 1,54 ***(<3)***
- DIT: 1,29 ***(<5)***

all of them are in the boundaries so we don't need to apply any design patterns to improve its maintainability.

5.4. Answer the Question "Is this product testable enough?"

Testability is a characteristic of software that refers to how easy it is to create automated tests for the code. In general, testable code is more modular, clear, and well-defined. This makes it easier to isolate and fix bugs when they are found. These practices also indicate for maintainability of the product. So when the product becomes more testable, it becomes more maintainable.

For to be able to comment on the testability of the product we are looking at the values of NOC and RFC metrics. The reason why we are choosing these metrics is the complex hierarchy and high coupling can make our code hard to be tested. If these metrics values are not in the acceptable value boundaries it will get harder to test them. If code is highly dependent isolating and testing individual units of code can be hard to succeed. If they are highly independent our tests can not perform effectively because there are limited opportunities.

Our metrics:

- NOC: 0,30 ***(1,3)***
- RFC: 7.98 ***(6,36)***

  Our NOC value should have been between 1 and 3 but we can see it's lower than that. This can seem good because our code is not complex and it does not require more setup and dependencies but low NOC gives limited opportunities. If the base class is not properly abstracted, it may require more testing and lead to increased testing effort. A low NOC can make it more difficult to test and maintain the codebase.

5.5. Answer the Question "Is this product stable enough?"

Stability is the ability of a software system to remain consistent and reliable over time, even in the changes and updates to the system. A stable software system is less likely to come up with unexpected bugs when there is a change or update, and this makes it easier to maintain the product. So more stable system implies that more maintainable system.

For this question, we will look at CBO and DIT metrics. They are used for this purpose because they provide us with information about interdependencies between different classes in the codebase. This information can be used to obtain ideas about the product's stability.

- CBO: 7,85 ***(<14)***
- DIT: 1,29 ***(<5)***

The values are in acceptable boundaries. This means the product's code is stable and it's reliable and robust. It has a low frequency of crashes or bugs. This software code performs intended functions consistently and reliably over time without unexpected errors or failures.

5.6. Answer the Question "Is this product's code readable?"

Readability is a measurement of how easy it is to understand code. It refers to the clarity and organization of the code, as well as the use of clear and understandable names for variables, functions, and classes. Readability makes it easier to understand the software system, identify new bugs, and make changes to the software. These features have importance on the maintainability too. So the product is maintainable if it is readable.

For this question, we need to examine the values of NOC, LCOM and, DIT metrics values.

The values are:

- NOC: 0,30 ***(1,3)***
- LCOM: 1,54 ***(<3)***
- DIT: 1,29 ***(<5)***

Our NOC value is low. We can look at the effect of this within an example:

For example, if we have

- Parent: Vehicle
- Subclasses: Truck, Car

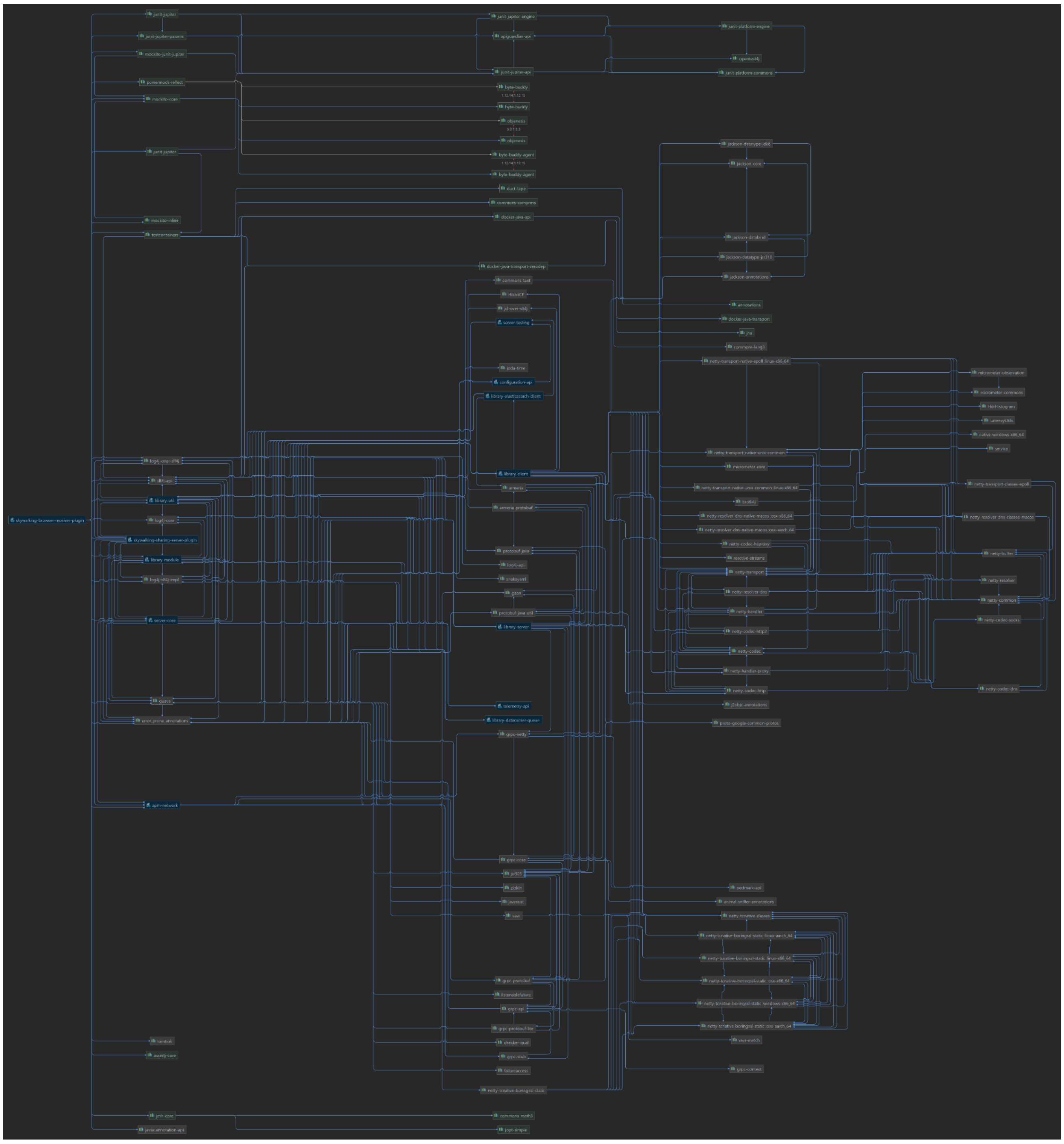
The attributes and methods for each type of vehicle may be put together in the Vehicle class if the class has a low NOC value, which indicates that there are fewer subclasses. Since it contains a lot of code pertaining to various types of vehicles, this can make the Vehicle class very large and challenging to read. The fact that changes to one type of vehicle may also affect others can make it more difficult to maintain and adjust the code.

Other values are in boundaries. This means the code's structure and organization is well designed.

6. DEPENDENCY GRAPH(S)

We used IntellijIDEA to create dependency graphs. We can analyze modules and their dependencies with respect to the color of the arrows.

- Blue arrows are reserved for libraries and modules dependencies
- Green arrows are reserved for test dependencies
- Red arrows are reserved for circular dependencies



Since the dependency graph is too big for the format, we put a drive link for a detailed look.

https://drive.google.com/file/d/1gb_dg90b5uhCDfLGK7KvgaVbdIPKck3O/view?usp=sharing

7. OVERALL EVALUATION OF CONCRETE ARCHITECTURE

We can comment on the architecture of the product by investigating the answers to the questions we mentioned in section 5. Also, the dependency graph in section 6 will help us to comprehend the concrete architecture of the product.

Maintainability is an important consideration in software architecture because it has a significant impact on the long-term sustainability of a software system. A software system that is difficult to maintain will become heavy and hard to manage over time, making it more expensive and time-consuming to modify or enhance the system. Designing a software system with maintainability can reduce costs, increase agility, and make software system answer to changes quickly while delivering high-quality and reliable products.

By looking at section 5, we can observe that the implementation of the software system satisfies the needs of maintainability. So we can conclude that our product is maintainable. Since maintainability has a higher importance on software architecture, we can say that our product has a well-designed architecture that promotes modularity, loose coupling, understandable code organizations, and testability.

Also, the C&K metrics we mentioned earlier have good values, except NOC. We use C&K metrics to understand the maintainability of the product but actually, they are very informative when it comes to evaluating software architecture too. Having good values of metrics implies well-designed architecture.

By looking at section 6, we can observe that some of the modules of the product have complex relationships between other modules and libraries. Unlike the others, some of them have a very simple organization. In general, the simpler ones are in the majority. We concluded the similar expression in section 5, the product is not complex in overall. But we can observe that some of the classes are too complex beside their minority, by looking at the dependency graph. The project needs some adjustments in this issue. Again, we can conclude that the product's architecture is well-designed.

8. REFERENCES

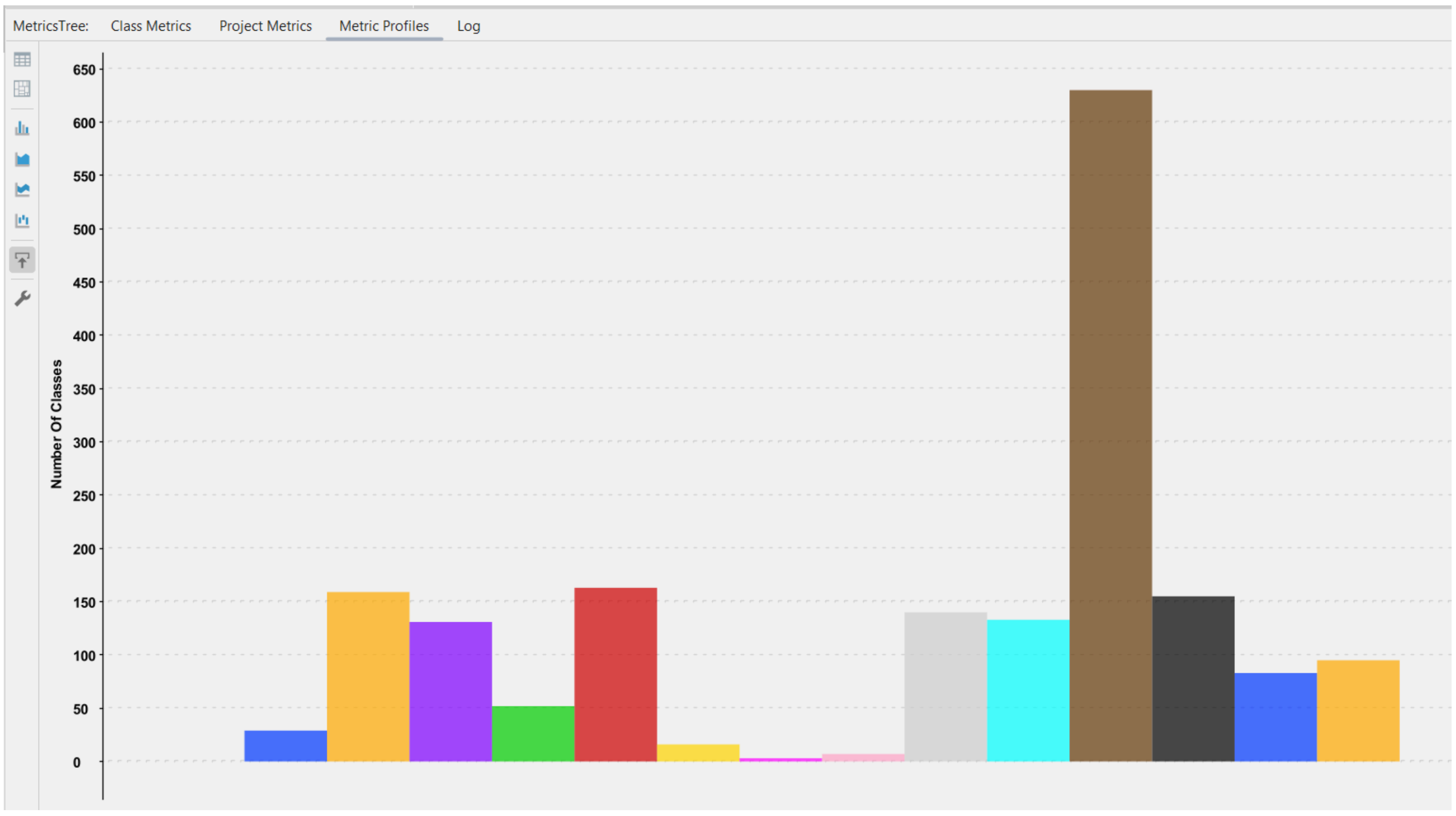
- **Chidamber-Kemerer Metric Values**
- https://www.castsoftware.com/glossary/software-complexity
- https://www.aivosto.com/project/help/pm-oo-ck.html
- http://www.openaccess.hacettepe.edu.tr:8080/xmlui/bitstream/handle/11655/3882/10158632.pdf?sequence=1&isAllowed=y
- https://dergipark.org.tr/tr/download/article-file/741010
- https://www.hindawi.com/journals/sp/2020/8840389/
- https://www.spinellis.gr/sw/ckjm/doc/metric.html#:~:text=The%20metric%20called%20the%20response,is%20invoked%20for%20that%20object
- https://en.wikipedia.org/wiki/Code_smell
- https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-class-coupling?view=vs-2022

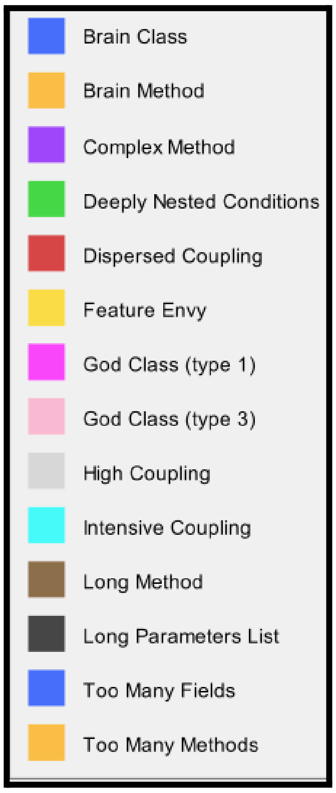
9. ALLOCATION OF RESPONSIBILITIES WITHIN TEAM MEMBERS

Name of Team Member	**Description of Responsibility**	**Allocation Unit**
Gamze Ergin	Filling section 1	section 1
Gamze Ergin	Filling section 2	*section 2*
Gamze Ergin	Filling section 3	*section 3 question 1, 2, 3*
Yunus Emre Terzi	Filling section 3	*section 3 question 4, 5, 6*
Yunus Emre Terzi	Filling section 4	*section 4*
Gamze Ergin	Answers for questions	*section 5 question 1, 2, 3*
Yunus Emre Terzi	Answers for questions	*section 5 question 4, 5, 6*
Yunus Emre Terzi	Filling section 6	*section 6*
Gamze Ergin	Filling seciton 6	*section 6*
Yunus Emre Terzi	Filling section 7	*section 7*
Gamze Ergin	Bonus Part	*A1.*
Yunus Emre Terzi	Bonus Part	*A2.*

APPENDIX. Bonus Part for T1 (10 pts)

A1. What are the types, numbers, and locations of the code smells detected in the source code of the OSS product?





Brain Class:

29 Brain Class detected.

Class tends to be complex and centralizes the functionality of the system.

Brain Method:

159 Brain Method detected.

Brain Methods are methods that centralize the intelligence of a class.

Complex Method:

131 Complex Method detected.

This smell occurs when a method has high cyclomatic complexity.

Deeply Nested Conditions:

52 Deeply Nested Conditions detected.

Nested conditionals are code smells because they make the code harder to understand.

Dispersed Coupling:

163 Dispersed Couplings detected.

A method suffers from Dispersed Coupling when it calls many other methods that are dispersed among many classes.

Feature Envy:

16 Feature Envy detected.

When a method is more interested in members of other classes than its own.

God Class:

10 God Class detected.

A god class is usually a huge class that concentrates a lot of responsibilities, controls and oversees many different objects, and effectively does everything in the application.

High Coupling:

140 High Coupling detected.

High coupling is a code smell that occurs when two or more classes or components are tightly coupled or dependent on each other.

Intensive Coupling:

133 Intensive Coupling detected.

When two or more components in a system are closely connected in a way allowing unnecessary communication between them.

Long Method:

630 Long Method detected.

When a method contains too many lines of code, it becomes hard to read and understand. **In this OSS this was the most concerned code smell.**

Long Parameters List:

155 Long Parameters List detected.

If the methods have a long number of parameters that creates this code smell.

Too Many Fields:

83 Too Many Fields detected.

Another name for this code smell is "Data Clumps" or "Data Class."

Too Many Methods:

95 Too Many Methods detected.

A long method list cause this code smell.

A2. Do you observe any relation(s) between the maintainability of the source code analyzed & the types, numbers, and locations of the code smells identified? If yes, explain the relation(s)?

Code smells are any characteristic feature in the source code of a program that possibly indicates a deeper problem. We can make assumptions about maintainability by investigating code smells.

Since it is a big and complex project, we can consider harmless the code smells that occur in the less than 100 classes because we have 1933 classes in total. So we can eliminate code smells *Brain Class, Deeply Nested Conditions, Feature Envy, God Class, Too Many Fields,* and *Too Many Methods*.

Brain Method, Complex Method, Dispersed Coupling, High Coupling, Intensive Coupling, and *Long Parameters List*. They are all relatable with maintainability and exist in more than 100 classes.

- Brain Methods are methods that centralize the intelligence of a class and manifest themselves as long and complex method that is difficult to understand. Since they are complex and hard to understand, we can say that they make it hard for developers to maintain the product.
- Coupling is a measure of how many classes a single class uses. High coupling indicates a difficult design to reuse and maintain because of its many interdependencies on other types.
- Long Parameters List is a method with usually more than three or four parameters. This occurs when the method tries to do too many things. It causes hard usage of method calls, hard reading of what a method does, and hard maintenance as a result.

They are all in the range of 100-200 classes. That means their percentage is %0.05 - %0.10. They indicate low maintainability but they are in the minority, not in the overall. So these smells don't imply that the project has low maintainability, but developers can deal with these smells to provide better maintainability.

Long Methods contain too many lines of code. They are hard to read and understand, thus hard to maintain. Nearly %30 of the classes have long methods. This shows us high percentage of the classes have long methods and this implies low maintainability. This is the most problematic code smell and developers need to handle this issue.

Except for *Long Methods,* number of the code smells indicates good maintainability, which is similar to what we found in the maintainability of the source code analysis. So we can infer that code smells are indicators of maintainability too.