BBM485 Software Architecture (2023-24'Fall)

Concrete Architecture Evaluation Report

Team Members:
2200356083 Bartu Bayazıt

1. OSS PRODUCT OVERVIEW

Felix Bertram started the TuringTrader project in September 2018, based on bits and pieces dating back to 2011. The project aims to provide a more productive environment for researching quantitative investment strategies.

Table 1. Description of the OSS product

Name of OSS Product:	**Turing Trader**
Description of OSS product:	An open-source solution for end-of-day trading, a backtesting engine/market simulator. Allowing traders to create and test their own investment strategies with ease and accuracy.
URL of OSS product:	https://www.turingtrader.org/
Size of OSS product (KLOC):	58

2. TOOLS USED FOR ANALYSIS

<List the tools that you used in the analysis of the OSS product.>

Table 2. The list of tools used for analysis

Tool Name	**Purpose of Use**	**URL**
SciTools Understand	to obtain maintainability metrics to create dependency graphs	https://scitools.com/

3. GQM TREE FOR EVALUATING OSS MAINTAINABILITY

<Summarize the goal and the questions to evaluate the maintainability of the OSS product by software metrics. It is suggested that your questions are answered by some indicators such as graphs (figures) or tables.>

Table 3. The goal, questions, and metrics to evaluate OSS maintainability

Goal:	To understand, the maintainability of the OSS product, from the viewpoint of *<architect/developer>*	
List of questions to answer while evaluating the goal:		*List of metrics used to answer the question:*
Q(1):	How does the overall code complexity and cohesion effect maintainability ?	Cyclomatic Complexity (CC) Lack of Cohesion in Methods (LOCM
Q(2):	How do the relationships between classes and their hierarchies affect code understandability?	Coupling Between Objects (CBO) Depth of Inheritance (DIT)
Q(3):	Which classes/modules demonstrate potential coupling issues and complex method interactions ?	Coupling Between Objects (CBO) Number of Nested Levels Response for a Class (RFC)
Q(4):	How does the complexity of inheritance hierarchies correlate with the depth of method invocations and the class structure ?	Depth of Inheritance Tree (DIT) Weighted Methods per Class (WMC) Response for a Class (RFC)
Q(5):	How does the inheritance hierarchy and class composition impact system scalability ?	Depth of Inheritance (DIT) Weighted Methods per Class (WMC) Number of Children (NOC)

4. DESCRIPTION OF SOFTWARE METRICS

<Provide a unique list of software metrics to answer the questions in your GQM tree, each with its name, type, source (the name of the tool that provides value for it), and formula/description provided by the source.>

Table 4. List of software metrics used in analysis

Metric Name	Type	Source	Formula (or description) by Source
Cyclomatic Complexity	Complexity	Understand	The cyclomatic complexity of any structured program with only one entrance point and one exit point is equal to the number of decision points contained in that program plus one. Understand counts the keywords for decision points (FOR, WHILE etc.) and then adds 1. For a switch statement, each 'case' is counted as 1. For languages with macros, the expanded macro text is also included in the calculation.
Number of Nested Levels	Quality	Understand	Maximum nesting level of control constructs (if, while, for, switch, etc.) in the function.
Coupling Between Objects (CBO)	C&K	Understand	The Coupling Between Object Classes (CBO) measure for a class is a count of the number of other classes to which it is coupled. Base classes and nested classes are not counted. Class A is coupled to class B if class A uses a type, data, or member from class B. This metric is also referred to as Efferent Coupling (Ce). Any number of couplings to a given class counts as 1 towards the metric total.
Lack of Cohesion in Methods (LOCM)	C&K	Understand	100% minus the average cohesion for package entities. 100% minus average cohesion for class data members. Calculates what percentage of class methods use a given class instance variable. To calculate, average percentages for all of that class's instance variables and subtract from 100%. A lower percentage means higher cohesion between class data and methods.
Depth of Inheritance Tree (DIT)	C&K	Understand	Maximum depth of class in inheritance tree. The depth of a class within the inheritance hierarchy is the maximum number of nodes from the class node to the root of the inheritance tree. The root node has a DIT of 0. The deeper within the hierarchy, the more methods the class can inherit, increasing its complexity.
Weighted Methods Per Class (WMC)	C&K	Understand	Number of local (not inherited) methods.
Response for a Class (RFC)	C&K	Understand	Number of methods, including inherited ones.
Number of Children (NOC)	C&K	Understand	Number of immediate subclasses, i.e., the number of classes one level down the inheritance tree from this class

5. EVALUATION OF MAINTAINABILITY

<Please evaluate the maintainability of the OSS product in relation to the questions in your GQM tree.

5.1. Answer to <Question-1>

Question 1 uses the metrics Cyclomatic Complexity(CC) and Lack of Cohesion in Methods (LOCM).

Cyclomatic Complexity is simply the measure of the linearly independent paths in the source code, which provides an understanding of the code complexity where a higher cyclomatic complexity indicates higher complexity and potentially harder to maintain source code. The minimum value we obtained for cyclomatic complexity from understand is 0, which indicates some methods are straightforward and have a linear flow. The maximum value we obtained is 48, which indicates highly complex methods with many decision points. The average value of 1.59 indicates that, on average, methods are relatively less complex, with a number of methods being lot more complex. In general, since the average cyclomatic complexity of methods is below the recommended values of around 5, we can say that maintenance of these methods wouldn't impose too much of a big challenge.

Lack of Cohesion in Methods metric measures the lack of relationship between methods in a class. A higher LCOM value indicates that methods within a class are less cohesive or have less in common. Cohesiveness of methods within a class is desirable, since it promotes encapsulation. Lack of cohesion implies classes should probably be split into two or more subclasses. Any measure of disparateness of methods helps identify flaws in the design of classes. Low cohesion increases complexity, thereby increasing the likelihood of errors during the development process.
When we look at the values we obtained, minimum value of 0 indicates that there are some classes with high cohesion, the maximum value of 100 suggests that there are classes with complete lack of cohesion, which have unrelated methods. An average value of 24.47 shows that, on average, there is a moderate level of cohesion among methods within classes. As high cohesion provides improved readability, understandability and reliability, our product having moderate level of cohesion in methods is beneficial in terms of maintainability.

5.2. Answer to <Question-2>

Question 2 uses the metrics Coupling Between Objects (CBO) and Depth of Inheritance Tree (DIT).

Coupling Between Objects (CBO) metric measures the dependencies between classes, indicating how much one class relies on another. Excessive coupling between object classes is detrimental to modular design and prevents reusability. The more independent a class is, the easier it is to reuse it in another application. In order to improve modularity and promote encapsulation, inter-object class couples should be kept to a minimum. The larger the number of couples, the higher the sensitivity to changes in other parts of the design and therefore maintenance is more difficult. A measure of coupling is also useful to determine how complex the testing of various parts of a design are likely to be. The higher the inter-object class coupling, the more rigorous the testing needs to be.
A lower metric value in this case indicates less interdependence between classes. The minimum value of 0 indicates that some classes are independent and have minimal or no dependencies on other classes. The maximum value of 63 indicates that certain classes are highly interconnected, potentially leading to complex interdependencies and difficulties in understanding the relationships between these classes. The average value of 8.92 signifies moderate coupling between objects on average, with some classes having relatively higher dependencies while others exhibit fewer connections. High coupling can lead to increased interdependencies between classes, making it harder to understand and modify the code without effecting other parts of the system. Classes with excessive dependencies (max CBO of 63) might require more effort to comprehend due to their complex relationships with other classes, potentially reducing code maintainability. But the average value of 8.92 implies that, maintainability of our product's source code is relatively good.

Depth of Inheritance Tree (DIT) measures the level of inheritance within the class hierarchy , showing how deep the class hierarchy goes. The deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behavior. Deeper trees constitute greater design complexity, since more methods and classes are involved. The deeper a particular class is in the hierarchy, the greater the potential reuse of inherited methods.

A minimum value of 0 suggests that there are classes with no inheritance, meaning there are standalone classes. The maximum value of 5 indicates that there are classes nested within several levels of inheritance, potentially creating complex hierarchical structures. An average value of 1.42 implies a relatively shallow inheritance tree on average, indicating that classes in general classes don't have extensive levels of inheritance, keeping the hierarchy relatively simple.
A deeper inheritance tree (max DIT of 5) might imply more complex hierarchies, which could impact code understandability by making it harder to trace the origin of functionalities and behaviors. However, the relatively low average DIT of 1.42 suggests that, on average, the inheritance tree is not extensively deep, which could contribute to better code understandability and in turn better maintainability.

5.3. Answer to <Question-3>

Question 3 uses the metrics Coupling Between Objects (CBO), Number of Nested Levels and Response For a Class (RFC).

As we mentioned before, a lower CBO value implies less dependence between classes. Higher coupling between objects can lead to increased complexity, making it harder to understand and maintain the code due to intertwined dependencies. Even though our product has classes that demonstrate high coupling, ones that have around 63 CBO, the average coupling value of 8.92 implies moderate coupling.

Number of Nested Levels metric measures how deep the nested control structures go. A deeper nesting level implies increased complexity and poor readability in terms of source code. Also, if these nested levels share the variables accessed and modified, this can impose a problem when it comes to code maintenance. Our product shows a minimum nesting level of 0, meaning there are flat structures throughout our source code, while a maximum value of 7 shows a deep level of nesting and an average value of 1.72. Even though the deeper nesting levels like 7 can make the code harder to follow, an average value of 1.72 indicates a generally shallow level of nesting, which improves readability and maintainability.

Response For a Class metric, as the name suggests, counts the number of methods in a class, including the inherited ones. If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes more complicated since it requires a greater level of understanding required on the part of the tester. The larger the number of methods that can be invoked from a class, the greater the complexity of the class. A worst case value for possible responses will assist in appropriate allocation of testing time.
A higher RFC value in this case suggests potentially complex method interactions between classes, which might increase the cognitive load required to understand and maintain these components. In the case of our product, minimum number of 0 for this metrics suggests that there are classes with almost no method interactions, while a maximum number of 126 implies that there are classes with potentially complex interactions. The average value of 48.11 indicates that, our product in general has a moderate level of methods invoked per class.

These three metric values together, when examined for each class, can help identify complex classes. A class that has one or more of these metrics with a value from "more than average" through "max" requires careful attention for refactoring or restructuring when it comes to code maintenance.

While the average values for these metrics suggest moderate levels of coupling, nesting, and method interactions on average, it's essential to focus on specific classes/modules with higher-than-average values (approaching the maximum) for potential issues. Addressing classes/modules exhibiting elevated metrics could contribute to improving the overall maintainability of the OSS product by refactoring complex interdependencies and method interactions, thereby enhancing code comprehensibility and maintainability.

5.4. Answer to <Question-4>

Question 4 uses the metrics Depth of Inheritance Tree (DIT), Weighted Methods per Class (WMC) and Response For a Class (RFC).

Depth of Inheritance Tree metric, as discussed previously, measures the level of inheritance in the class hierarchy. The minimum value of 0 for our product indicates that there are standalone classes with no inheritance, while a maximum value of 5 shows that there are classes with several levels of inheritance. The average value of 1.42 implies that in general, our product does not have classes with deep inheritances, which improves the understandability and maintainability of our source code.

Weighted Methods per Class (WMC) metric is used to measure the complexity and size of a class by counting the number of methods within that class. It assigns weights to each method based on complexity factors such as method invocations, loops, conditionals, etcetera, and then sums these weights to derive a total WMC for the class. The number of methods and the complexity of methods involved is a predictor of how much time and effort is required to develop and to maintain the class. The larger the number of methods in a class, the greater the potential impact on children, since children will inherit all the methods defined in the class. Classes with high number of methods are more likely to be application specific, limiting the possibility of reuse.
When it comes to maintaining the OSS product, the WMC metric provides insights into the potential complexity and maintainability challenges within individual classes. A higher WMC value often indicates increased complexity within a class, potentially leading to decreased code maintainability. More methods could mean increased intricacy in understanding, debugging, and modifying the class. Highly complex classes with elevated WMC values might be more susceptible to causing ripple effects or unintended consequences when modified. This could potentially hinder the ease of future maintenance.
High WMC values can signal areas for refactoring, where complex classes can be broken down into smaller, more manageable parts. This refactoring effort aims to improve maintainability by reducing the complexity of individual classes.

Response For a Class metric, also as discussed above, counts the number of methods in a class, including the inherited ones. The range describes the number of methods invoked within classes, ranging from minimal method interactions (minimum value of 0) to potentially complex interactions (maximum value of 126). The average value of 48.11 indicates a moderate level of method interactions within classes. Higher RFC values suggest potentially complex method interactions within classes, impacting their understandability and maintainability.

The metric values obtained indicate a mainly shallow inheritance hierarchy, moderate complexity in terms of methods and their interactions within classes. The moderate values suggest a reasonable balance, but when managing and refactoring the classes that skew towards the maximum values requires greater caution as they might present potential complexity and maintainability challenges.

5.5. Answer to <Question-5>

Question 5 uses the metrics Depth of Inheritance Tree (DIT), Weighted Methods per Class (WMC) and Number Of Children (NOC).

The Depth of Inheritance Tree (DIT) metric, as we discussed before, measures the depth of the inheritance hierarchy within classes.
The values obtained indicate a range from no inheritance (minimum value of 0) to moderately deep inheritance (maximum value of 5) within classes. Shallow inheritance hierarchies, which are indicated by lower DIT values, generally lead to simpler class structures, aiding scalability due to reduced complexity. Classes with deeper hierarchies might introduce more complexity, impacting scalability negatively.

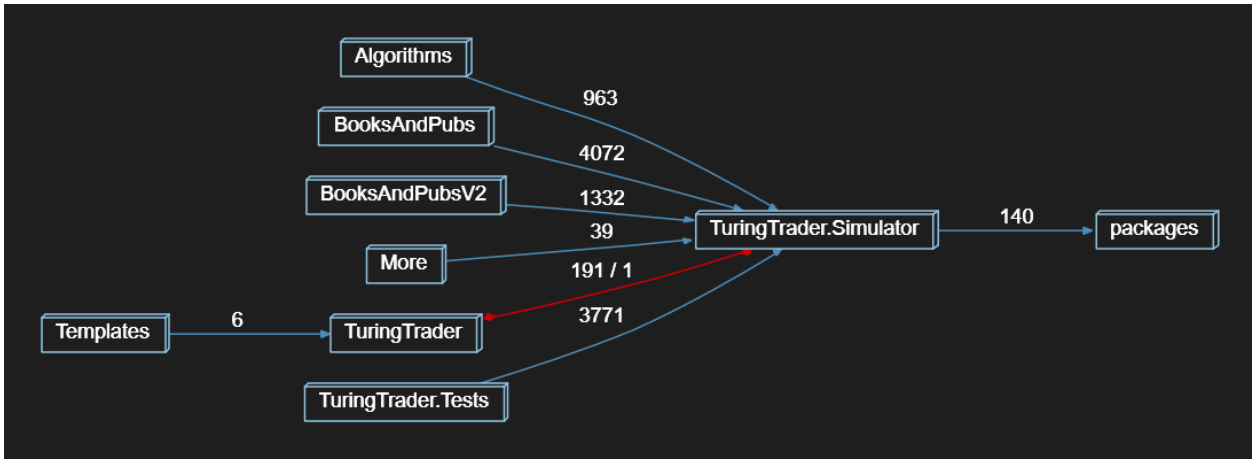
The Weighted Methods per Class (WMC) metric, as we also discussed previously, measures the complexity of methods within classes based on various factors such as method invocations, loops, conditionals and etcetera. The values obtained range from classes with few methods (minimum value of 0) to more complex classes (maximum value of 73). Classes with higher WMC values might indicate increased complexity, negatively impacting scalability due to increased cognitive load and maintenance efforts required.

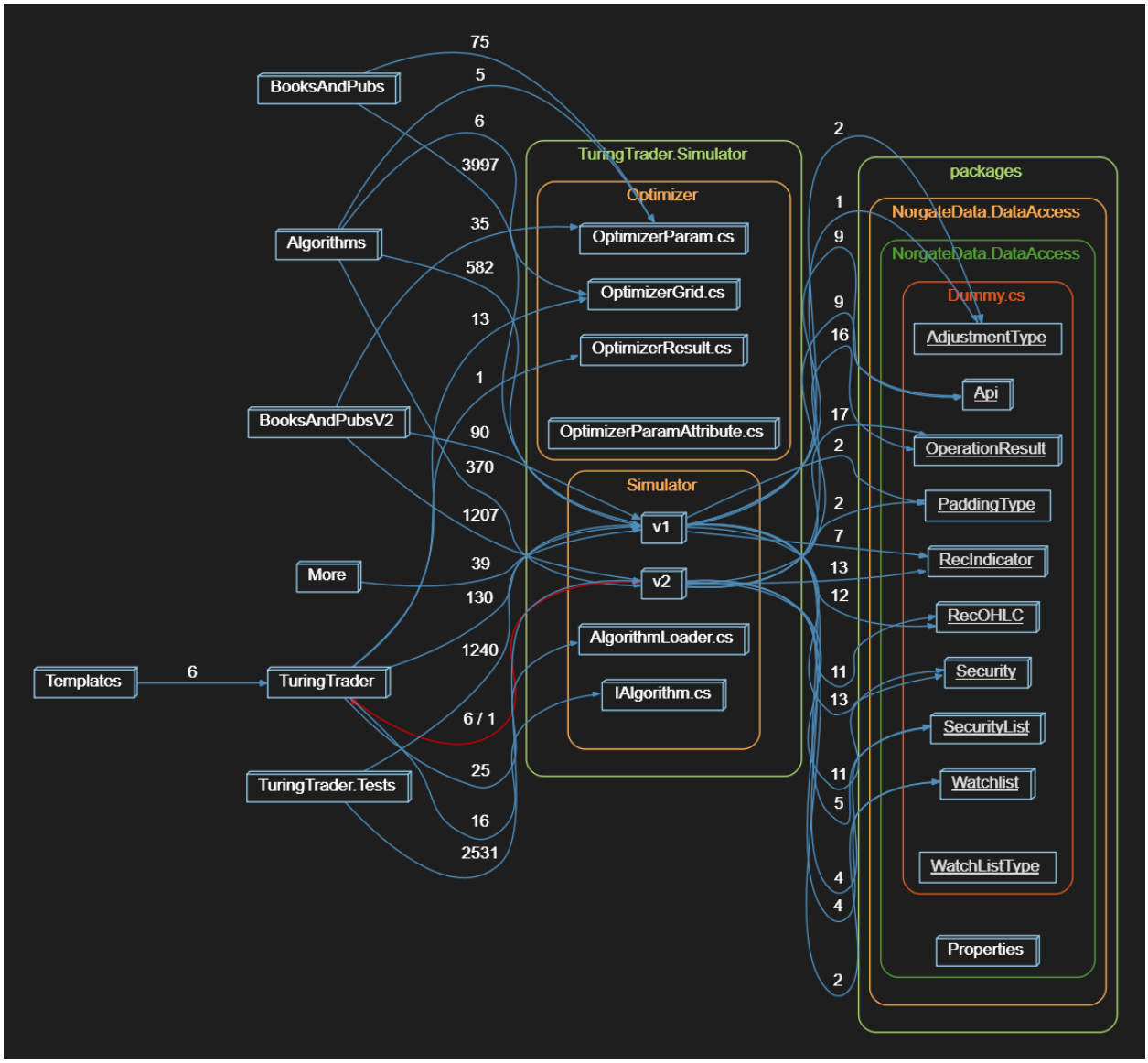
Number of Children (NOC) metric count the number of immediate subclasses, i.e., the number of classes one level down the inheritance tree from this class. This metric reflects the level of inheritance and class hierarchy within the codebase. Greater number of children, greater the reuse, since inheritance is a form of reuse. On the other hand, greater number of children might also indicate the likelihood of improper abstraction of the parent class. If a class has a high number of children, it may be a case of misuse of subclassing. Number of children also gives an idea of the potential influence a class has on the design. If a class has a large number of children, it may require more testing of the methods in that class.
A higher number of children inheriting from a single parent class can increase dependence among classes. Changes made to the parent class might have ripple effects on multiple child classes, potentially increasing coupling. This increased coupling can make the codebase more fragile and harder to maintain.

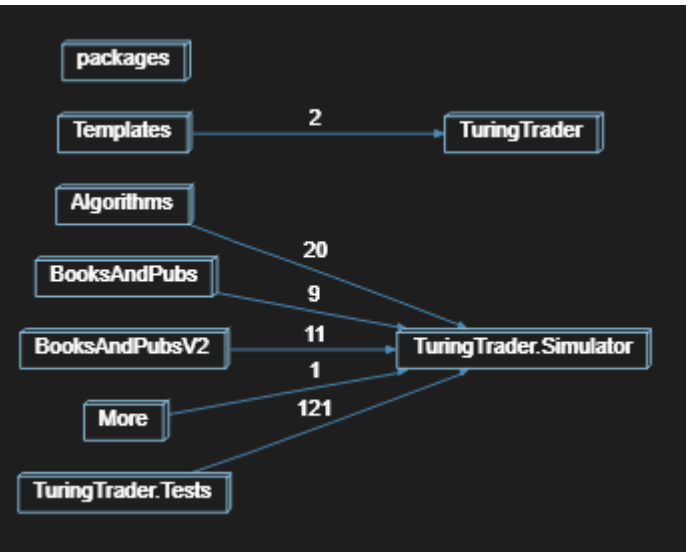
A larger number of children can potentially complicate code comprehension. Developers maintaining or modifying the code may find it more challenging to understand the relationships between the parent and numerous child classes, affecting maintainability. A well-designed inheritance hierarchy with a manageable number of children can enhance the flexibility and scalability of the codebase. However, an excessively large number of children might disturb the evolution of the system due to increased complexity and maintenance overhead. The obtained metric values range from no child classes (minimum value of 0) to classes with multiple children (maximum value of 100). A higher number of children might introduce tighter coupling and dependencies among classes, impacting scalability as changes to the parent class might affect multiple child classes.

In general, when considering these three metrics altogether; classes with lower DIT might simplify scalability due to reduced complexity and simpler class structures. They could potentially be easier to scale and maintain.
Classes with higher WMC values may impose challenges to scalability and maintainability due to increased complexity. They might require more effort for maintenance and could impact system scalability negatively. A higher number of children could lead to increased dependencies and potential maintenance challenges, impacting scalability if changes in parent classes affect multiple children.

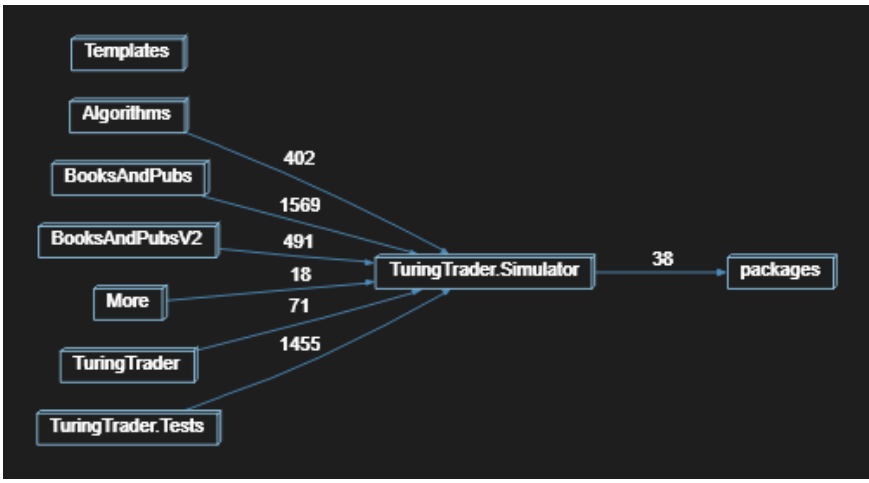
6. DEPENDENCY GRAPH(S)



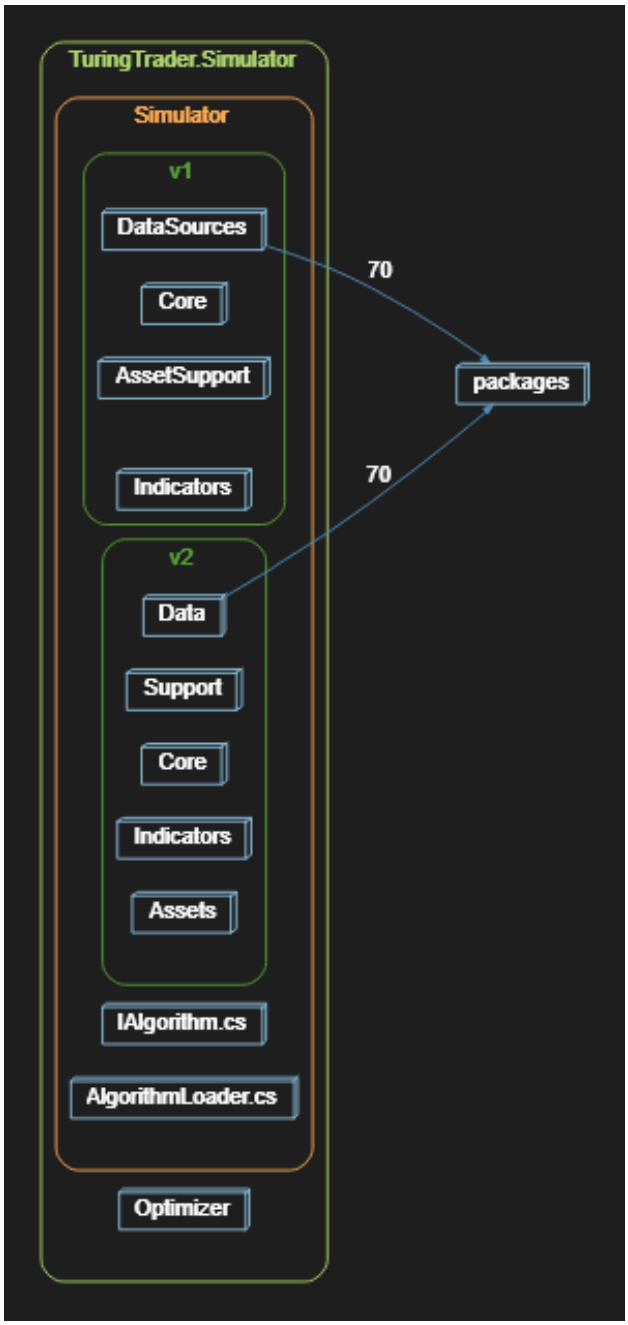


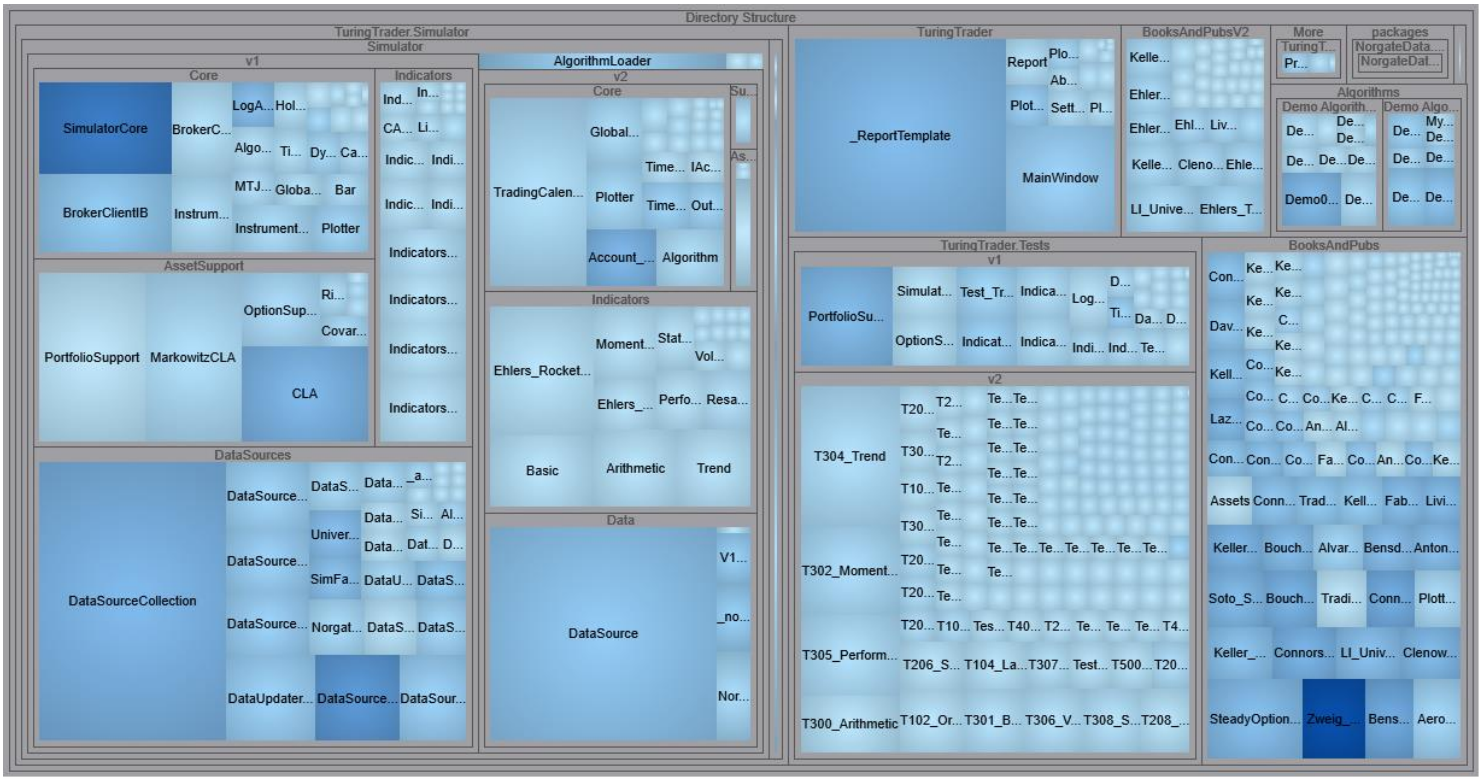


Base/Derived Classes



Function Calls





7. OVERALL EVALUATION OF CONCRETE ARCHITECTURE

As we can see from the dependency graphs, almost all parts of the product, directly or indirectly, depend on the part called "Packages". In "Packages" directory, everything is inside a directory called "Norgate Data Access". Since our product is a backtesting engine/market simulator for trading strategies, accessing the data to be processed is one of the main concerns. The "Norgate Data Access" part provides just this. The part in "Simulator" that depends on "Norgate Data" is, as expected the simulator itself. The "Optimizer" part does its work on its own. The "Simulator" has two subdirectories, namely V1 and V2. Inside these directories there are parts concerned with data collection and only "Norgate" variation of these data collection parts depend on the "Norgate Data" package. When looking at the metrics tree map with "max cyclomatic" metric, the two darkest parts are "SimulatorCore" part in V1 of the Simulator with cyclomatic complexity of 34 and the second most complex part is "Zweig_BondModel" inside "BooksAndPubs" with cyclomatic complexity of 48 (which is also the maximum cyclomatic complexity value of our product). The parts called "BooksAndPubs" and "BooksAndPubsV2" have interdependencies between their classes and also have classes that all other classes inside BooksAndPubs depend on, these are namely Indices, Assets and GlueLogic for "BooksAndPubs" and GlueLogicV2 for "BooksAndPubsV2", even though the dependency graphs of these parts look complicated, the cyclomatic complexity of the components are nowhere near the 34 of SimulatorCore or 48 of Zweig_BondModel. These parts seem to be the communication between the simulator core and the published trading strategies that are available in our product, through stock market indexes, asset tickers and glue logic. When we look at the dependency graph in terms of base/derived classes, we see that the part that depends on the Simulator the most is TuringTrader.Tests. When we look at the dependency graph from the point of function calls, the two parts that depend on the Simulator are Tests and BooksAndPubs. When we look at the metrics we obtained, specifically Cyclomatic Complexity, Number of Children and Response For a Class, and our dependency graphs, as a result we see that parts of our product where overall complexity and design affects maintainability the most seems to be "BooksAndPubs", "Tests" and "SimulatorCore".

8. REFERENCES

https://docs.google.com/spreadsheets/d/1y-v_SwidevQYOUCdGR3GVl_LyYbdw42wDorQAI4iE9o/edit?usp=sharing – Minimum, maximum, total and average metrics for general interpretation

https://docs.google.com/spreadsheets/d/1RzKJNywIDRwZqYrllSpOehlGRT9jd62ZabWdvkKr5VQ/edit?usp=sharing

– Raw metrics obtained from Understand

S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," IEEE Transactions on Software Engineering, vol. 20, no. 6, pp. 476–493, 1994.

T. J. McCabe, "A complexity measure," IEEE Transactions on Software Engineering, vol. SE-2, no. 4, pp. 308–320, 1976.

Samoladas, I., Gousios, G., Spinellis, D., & Stamelos, I. (2008). The SQO-OSS quality model: measurement based open source software evaluation. In Open Source Development, Communities and Quality: IFIP 20 th World Computer Congress, Working Group 2.3 on Open Source Software, September 7-10, 2008, Milano, Italy 4 (pp. 237-248). Springer US.

https://www.geeksforgeeks.org/cyclomatic-complexity/

https://www.aivosto.com/project/help/pm-oo-cohesion.html#LCOM4

https://www.geeksforgeeks.org/software-engineering-coupling-and-cohesion/

https://blog.codacy.com/reduce-cyclomatic-complexity

9. ALLOCATION OF RESPONSIBILITIES WITHIN TEAM MEMBERS

<For groups of 2-3 people, please describe the allocation of responsibilities within team members.>

<You may allocate responsibilities with respect to the questions in your GQM tree, to the sections of this report, or to the design elements (e.g. packages) in your product architecture. In any case, provide a detailed explanation to objectively give credit to each member's work.>

Name of Team Member	Description of Responsibility	Allocation Unit
		<e.g., question #, section #, package # or name>