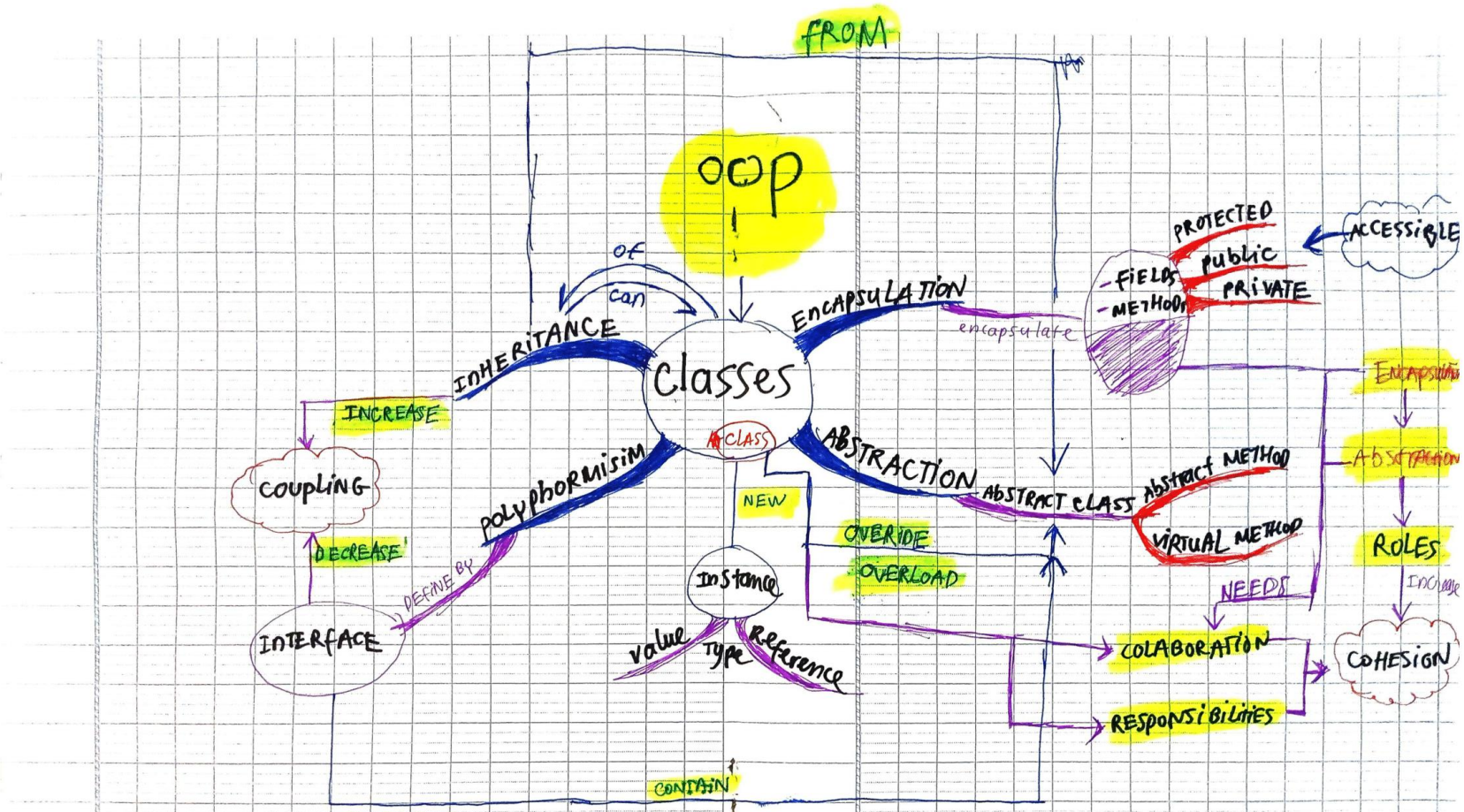
Name: Le Minh Kha

StudentID: 104179506

Key Object-Oriented Concepts report

Object-oriented programming (OOP), places a greater emphasis on objects rather than the logic and function of the structural approach. OOP is particularly suitable for complex, frequently updated, and large-scale applications because programmers have the flexibility to modify and arrange the items within a program. The accompanying image captures the fundamental concepts of OOP:

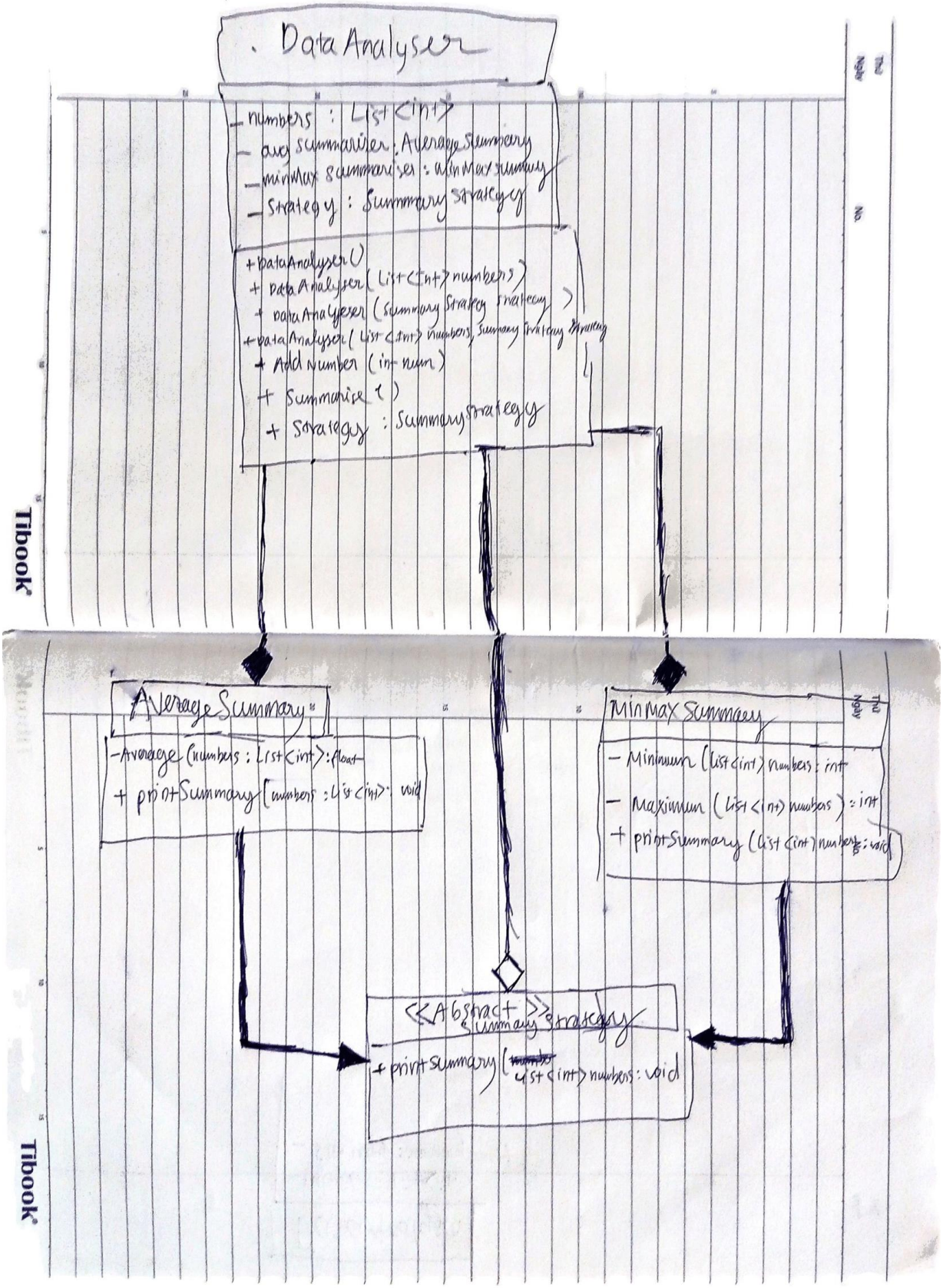


THE OOP CONCEPT MIND-MAP

-s104179506-

One of the beauties of OOP is that it captures the fundamental concept of the real world, where everything can be represented as an object with unique attributes and features. With encapsulation, OOP allows the bundling of data and methods within a class, protecting them from external access. This encapsulation also provides abstraction, enabling programmers to work with objects without needing to understand their underlying implementation details. OOP enables the creation of classes or blueprints of objects and the instantiation of instances from them. Objects of the same class can differ from one another due to polymorphism, inheritance, and encapsulation.

The 4 main principles of OOP is Abstraction, Encapsulation, Inheritance and Polymorphism. Hurdle Test 1 will be used as a primary example to demonstrate the principles:


DataAnalyser
- numbers : List<int>
- avgSummariser : AverageSummary
- minmaxSummariser : MinMaxSummary
- Strategy : SummaryStrategy
+ DataAnalyser()
+ DataAnalyser(List<int> numbers)
+ DataAnalyser(SummaryStrategy strategy)
+ DataAnalyser(List<int> numbers, SummaryStrategy strategy)
+ AddNumber(int num)
+ Summarise()
+ Strategy : SummaryStrategy
AverageSummary
-Average(numbers : List<int>):float
+ printSummary(numbers : List<int>): void
MinMax Summary
- Minimum(List<int> numbers : int
- Maximum(List<int> numbers) : int
+ printSummary(List<int> numbers): void
<<Abstract>>
SummaryStrategy
+ printSummary(List<int> numbers: void

1. *Abstraction.*

   Abstraction is like a capsule that encapsulates any essential features of an entity while hiding the unnecessary details. In OOP, abstraction only expose the essential features through a well-defined interface for a group of related objects.

   Using the UML diagram above, it is evident that DataAnalyser class can interact with summary strategies through a common interface provided by the SummaryStrategy abstract class, rather than needing to know the details of each specific implementation. As a matter of fact, DataAnalyser class doesn't need to know whether the summary strategy calculates the average or the minimum and maximum values - it only needs to know that it can call the PrintSummary() method to get a summary of the data which make the code very flexible.

2. *Encapsulation*

   Encapsulation enables the compression of fields and methods within a class, adding layer of security and protecting them from external access. Encapsulation could be imagined like this, the Class act as a safe house; every data and methods are safely protected in it. The lock would be the public interface and a authorize key would be needed to unlock the house and access the data within. To obtain a access key, public method would be needed (get, set methods).

   Taking UML diagram as a example, DataAnalyser class has four private attributes. Those attributes are inaccessible from outside of the class, however, private variable _strategy is accessible due to a public get set method making it accessible externally. This ensure the class can enforce its own rules and maintain its internal state, while still allowing external objects to interact with it through a well-defined interface.

3. *Inheritance*

   Inheritance enable sub classes to “inherit” from the parent class, this might be properties and behaviors. Inheritance could be imagined as a family tree, there would be hierarchical relationship between classes, where subclasses(the child) inherit the properties and behaviors of their parent classes. This enables the subclass to reuse and build upon the functionality of the parent class, just as a child builds upon the legacy of

their ancestors. The child inherits all of the features of the parents and might develop their own way of achieving the methods whilst their parent could not.
In the UML diagram, AverageSummary class inherit from SummaryStrategy abstract class. Imagine AverageSummary class not having public method PrintSummary but normal class SummaryStrategy does have it but the method does return value and is not meant to be override. AverageSummary would be able to call the PrintSummary method without actually having the method in the class because it inherits that function from SummaryStrategy class.

4. Polymorphism

Polymorphism means to take many forms, meaning instances of different classes.
can be treated as if they are of the same type. This means methods can be reusable, as the same method can be used with different instances. Using the UML diagram above as an example:
Polymorphism was used to allow DataAnalyser class to switch between AverageSummary and MinMaxSummary, which is stored as objects of abstract class SummaryStrategy. We could essentially call the abstract method how many times we like for classes that inherit from abstract class SummaryStrategy without having to specify a particular implementation.

Beside the 4 principles, here are some concepts worth mentioning.

Roles refer to an object's functions and the assigned or required duties that are connected to the system object. A group of interconnected duties that an object can perform is also known as a role.

Responsibility is the obligations of the classifier or employee that are associated with the behavior of the object. This includes performing specific actions or having particular knowledge.

Coupling indicates the level of reliance and connection between two groups. Low coupling implies minimal connection, while strong coupling should be avoided in OOP design as it can make code more difficult to maintain and modify due to its nature of fault intolerance.

Cohesion refers to the measure of a class's functionality. A high cohesion indicates that the class is focused on its intended purpose, while a low cohesion means that it contains a broad range of methods and functions without a clear objective.

Overall, a well-designed OOP program should have high cohesion and low coupling.