SOFTENG 306 - Project 2

System Refactoring Report

Team 8

***Note to Marker**:* Team 8's refactored code does not persist or obtain data from csv files, but rather uses JSON files to store system data. The system no longer uses csv files and is not compatible with them. The team got Reza's approval before making this change, but thought it important to mention in case the markers test the system using csv files.

Introduction

A modern day university management system involves different categories of users, each with a specific set of tasks they wish to achieve via the system. Representing a university management system using object-oriented principles and design in Java allows for the separation of concerns (SoC) via the use of different class modules and applying the SOLID design principles. There are several ways to design a system that provides a given set of functionality, but the system provided to the team suffered from several design smells, which are discussed further below.

This report discusses the design smells identified in the as-is system and the proposed solutions to the identified smells by means of applying the SOLID design principles. To illustrate the changes made, the new class diagram for the refactored system has also been provided. Additionally, the metrics used to objectively evaluate the as-is system have been recalculated for the refactored system, in addition to a few new metrics introduced to determine the design quality of the refactored system. This allows the two systems to be objectively compared against each other and quantify any improvements made.

Finally, the design of the refactored system is analysed against the as-is system with respect to maintainability, reusability, coupling and coherency. This report is the culmination of the design cycle followed by the group and the product of the agile sprints undertaken by the team.

1. Design Smells in the As-Is System

1.1. Rigidity

1.1.1. Variables of Type ArrayList and HashMap

In the as-is system, the concrete type ArrayList was used to define variable types rather than the List interface type in classes such as Course, MarkMgr, CourseMgr and FILEMgr. This increases rigidity as the program relies on the specific implementation rather than the interface abstraction. If a different subtype of List such as Vector needs to be for a variable instead, changes would need to be made to the dependent modules which expect an ArrayList object to be supplied. This design smell was also observed with variable types declared using the HashMap concrete type rather than the Map interface type. This design violates Dependency Inversion Principle (DIP) (Section 3.5.1).

1.1.2. Large Number of Associations and Coupling Between Classes

The classes ValidationMgr, Course, MainComponent, and Student all have a large number of associations with classes throughout the system. This can be seen in the high values for both the $WMC_{Fan\text{-}In}$ (30, 85, 32, 40) and CBO (11, 14, 6, 10) for these classes. For example, Course has associations to Main, Professor, CourseRegistration, Mark and MainComponent among others. If any of these classes needed to be changed, this would likely require changes to several other classes associated with it. Thus, these classes are quite rigid in the as-is system.

1.2. Fragility

1.2.1. Main

The Main class stores all domain object lists as public fields, which can be accessed by all other classes. Despite Main not having any conceptual relationships to any of the domain classes, it stores instances of the domain classes. If any of the domain classes break, for example Course, then the Main class will also break as it is the central data storage class. This design is extremely fragile as it is hard to anticipate the code behaviour and debug problems in Main if the domain classes break. This design smell is indicated by the CBO value of Main indicating that it is associated with 8 conceptually unrelated classes. The Main class should simply be delegating responsibility to the manager classes.

1.2.2. FILEMgr

FILEMgr was meant to be responsible for reading in data from csv files and creating domain objects which were stored in the Main class. Additionally, FILEMgr was also responsible for persisting the domain data in the csv files after the main menu was quit. But the CBO value of FILEMgr in the as-is system was 12, indicating that it had associations with several other classes which it wasn't conceptually related to. For example, FILEMgr was directly iterating with CourseRegistrationMgr and StudentMgr, with which it had no conceptual relationship. If the manager classes FILEMgr was interacting with broke, then the fragile code would result in FILEMgr breaking as well, which is not what would be expected.

1.3. Immobility

1.3.1. Large Manager Classes

There were various manager classes in the as-is system that were very large, and contained a lot of functionality. For example, ValidationMgr was a class that contained methods for all the required validation of user inputs. As the system involved a large amount of user input, the ValidationMgr class therefore contained a lot of methods. If we wanted to extract the validation of Student-related inputs into a different module, we would have to examine and dissect the ValidationMgr class, which would require several changes to related modules. A similar issue was observed for HelpInfoMgr and FILEMgr.

1.3.2. Long Methods

Throughout the entire as-is system, methods appear to have been designed to contain all the functionality that the method required. For example, the addCourse() method inside CourseMgr was almost 400 lines long, and contained functionality for getting all required user input, error checking, outputting information to the user, and updating the system state. This makes the method very hard to separate into components that might need to be reused, such as just updating the system state or outputting information to the user. This same issue can be seen in many other methods in the system such as addStudent() in StudentMgr and setCourseWorkMark() in MarkMgr.

1.3.3. Heavy Coupling Between Modules

It could be observed from the UML diagram, as well as from CBO statistics, that there was a high degree of coupling throughout the system. Even the large WMC_{Fan-In} values for classes such as Course (85) and Student (40) were an indication of the coupling problems in the system. The large number of dependencies between classes would make it very hard to take a single class out of the system, as this class would rely on functionality within many other classes. Thus, the entire system has a large degree of immobility.

1.4. Viscosity

1.4.1. File Manager

FILEMgr in the as-is system insisted on being its own file management system. Indexes of columns for every property and every csv file in the system were all stored in a single FILEMgr class as static final fields, when in fact the indexes could be retrieved from the column headers themselves. The attempt to circumvent the norm is evidence of viscosity in the system, but was not explicitly seen in a metric.

1.4.2. Dummy Streams

Within ValidationMgr, many of the validation methods included calls to methods that involved printing output to the user. However, this output was not wanted, as the validation method was just checking the validity of an input. To prevent output, the system output stream was set to a dummy stream, diverting all output so that it would not be displayed to the user. This is a hack solution of preventing output, as the output is still there, the user just can’t see it. Dummy streams were used many times for this purpose, so the wrong solution was easier to implement than the correct one, indicating that the ValidationMgr suffers from high viscosity. This is an implementation problem, and is not seen from the statistics.

1.4.3. Enum Usage

In many places where enum values were passed into methods, such as in ValidationMgr methods, the string representation of the enum was used. This string was then connected to an enum value. Dealing with strings can be easier than dealing with enums, however it is a hacky solution that introduces viscosity, and has caused complexity in the form of conversions and extra error checking.

1.5. Needless Complexity

1.5.1. Groups

The classes TutorialGroup, LabGroup, and LectureGroups were mostly empty classes, containing only a constructor that called the super constructor in the Group class. All uses of these three classes involved using methods within the Group parent class. Thus, these classes did not encapsulate any information, did not have any responsibility, and were repetitive structures that offered no use other than for type checking. If the system were to be extended, then there may be a need for operations on different group types. However, the current system does not require this, and thus these child group classes are units of needless complexity and repetition. This is supported by the statistics of $\text{WMC}_{\text{Fan-In}}$ and CBO, which are both relatively small for all three child classes.

1.5.2. ProfessorMgr

The class ProfessorMgr is completely unused in the system. It is not called by any other class in the system as the main flow of the program does not provide any functionality to add or modify Professors. Thus this class as a whole provides no benefit to the system and is considered as a source of needless complexity. Evidence of this is also seen in the metrics, which shows that the ProfessorMgr has a $\text{WMC}_{\text{Fan-In}}$ of 0, showing that no other classes use the methods of ProfessorMgr.

1.5.3. Unused Methods

Outside of ProfessorMgr, there were two unused methods. CourseworkComponent.printComponentType() and FILEMgr.writeProfIntoFile() were both unused in the system. Thus, they represent units of needless complexity that were likely implemented in case of future use.

1.5.4. Unused Enum Classes

The Enum classes Department, CourseType and Gender were introduced in the as-is system with the intent of using them in the domain classes such as Student, Course and Professor, but were never used in these classes as field types. These classes were added in an attempt to prepare for future changes, which led to needless complexity.

1.6. Needless Repetition

1.6.1. Groups

All three group subclasses were identical, apart from the class name. This was hinted at by the statistics, as the WMC_{Fan-In}, DIT, and CBO for all group subclasses were equal. Following on from the needless repetition of the classes, the usages of the classes were also duplicated. For example, when entering a new course, the user was prompted to enter LabGroups, LectureGroups, and TutorialGroups for the new course. There were separate blocks of code for each different group type, however the necessary functionality, and the code itself, was the same (apart from the group name). This created a large amount of repetition, as each code block was quite large.

1.6.2. Regex Validation

Within ValidationMgr, several of the valid input checking methods followed the same structure and were mostly duplicate code. For example, there were five methods that used a regex string to validate an input string. These were refactored into a method that takes a regex string and an input string, and validates the input string using the regex string.

1.7. Opacity

1.7.1. Long Methods and Classes

In many areas of the as-is system, certain methods were excessively long, such as MarkMgr.setCourseWorkMark() reaching 92 lines, as well as CourseMgr.addCourse() reaching 393 lines of code. The general system complexity was further worsened by the unreasonable amount of nesting, reaching at a peak of 10 indentations in CourseMgr.addCourse(). This resulted in code that was overall difficult to understand, thus inducing general system opacity. Very large classes were present in the as-is system such as CourseMgr with 691 lines of code (LOC) and FILEMgr with 1163 LOC. These issues significantly reduced the readability, understandability and maintainability of the system. These issues link to the issues discussed in the sections 1.3.1 and 1.3.2.

1.7.2. Method Names

Some method names were inaccurate. These methods claim to do one thing, but perform another, which means code which calls these methods are harder to understand. Method names like checkStudentExists() and checkCourseExists() suggest they are checking for the existence of a given student and course respectively in the system, when in actuality they prompt the user to input an existing student/course ID. This difference between rational logic and reality is evidence of the system smelling of opaqueness.

1.7.3. Overloaded Methods

In ValidationMgr, several methods of the type 'check<Class>Exists' were overloaded. These methods had the same method names but had separate parameters. Depending on the parameters, the methods did separate actions: if it had no parameter it acted as an IO class prompting user input, whereas if it had a string input, it acted as a getter for the specific item with the ID of the string. This overly complicated code is not explained and makes the system opaque to read.

1.7.4. Large Constructors

In the Course domain class, the constructor consisted of 10 input parameters. The existence of this constructor created excessively long lines of code, which contributed to the problem of readability. Furthermore if there are multiple parameters with the same type, it is easy to accidentally switch the parameters due to lack of type checking.

1.7.5. Comments as Deodorant

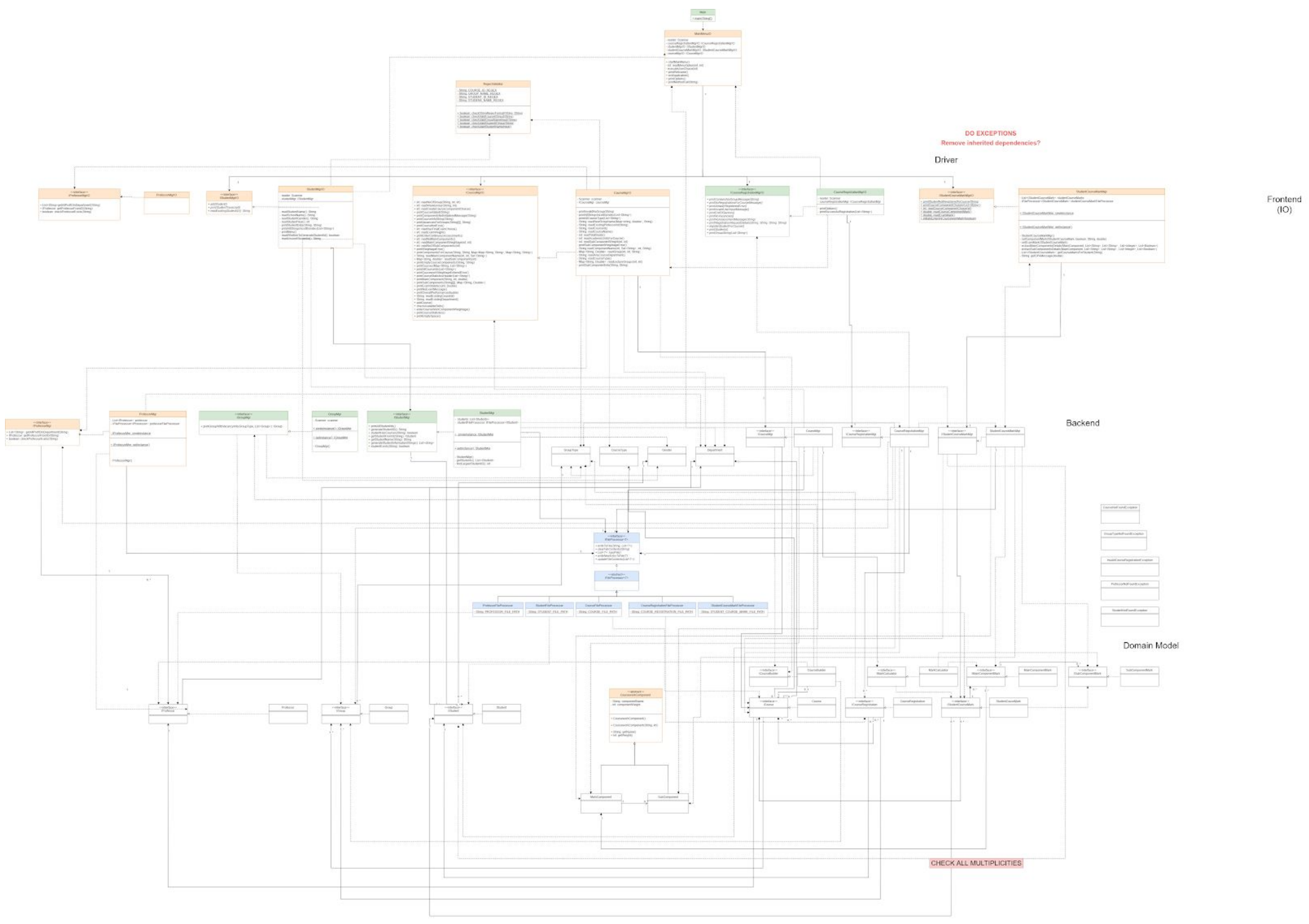
Excessive comments were included in the as-is system. For example, the field groupName in the Group class was commented with “The name of this group”. This should be self-explanatory by the field name, and adds nothing to the understanding of the code. It also adds extra clutter that the programmer has to read when attempting to understand the code. Adding comments as deodorant to the code indicates poor readability and design practices.

2. Class Diagram of the Refactored System

2.1. Assumptions

- Assume that class X getters which return lists of class Y are not uses of class Y.
- Wherever a class/object is called from a method, we consider it as a dependency (dotted line). If the object is stored as a field, we use an association (solid line).
- If A stores a field of type B and B inherits from C, then the association is drawn from A to B and no dependency needs be drawn from A to C, even if the method is defined in C. This is done as it is believed that an association with B automatically satisfies the dependency on C.

2.2. New Class Diagram



Link to new diagram here:
https://github.com/SoftEng306-2020/project-2-team-8/raw/master/documentation/refactored-class-diagram.png

The class diagram above distinctly follows the model-view-controller (MVC) architecture with a model (domain classes), view (IO classes) and the controller (manager classes) linking them together. This allows for SoC.

3. SOLID Design Principles Applied in the Refactored System

3.1. Single Responsibility Principle (SRP)

3.1.1. Manager Input/Output (IO) Classes

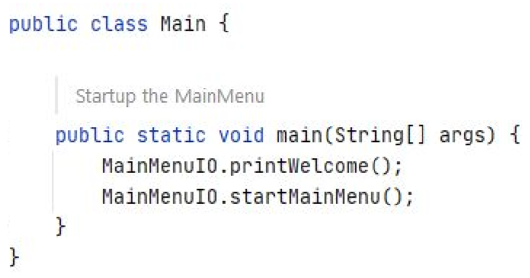
In the as-is system, the manager classes such as CourseMgr, ValidationMgr and StudentMgr were all directly reading user inputs and printing output to the console. Alongside this, these manager classes were also responsible for abstracting the logic to manage the respective domain objects. For example CourseMgr was responsible for abstracting the logic to add a course and iterate through a list of courses whenever required. This resulted in design smells being introduced into the system such as rigidity, immobility, viscosity, opacity and needless repetition.

In the refactored system, new IO managers were introduced, which are solely responsible for reading user input and printing output to the console. Each manager class has its own respective IO manager responsible for interacting with the user. As these classes only handle IO operations, they follow SRP. This allows for the separation of concerns, improved readability, maintainability and decreased coupling with for the manager classes.

This helped remove several smells from the system. Firstly, the class size for the managers and IO classes significantly reduced, decreasing code opacity, immobility and rigidity as there are fewer associations which allows for smaller modules to be extracted easily in the future. Dummy streams are no longer required as the validation of input is separate from the printing of output, which helps decrease viscosity. As IO is now abstracted into a separate class, the needless repetition smell is removed as single methods, for example in CourseMgr, have been extracted rather than using method duplication due to IO restrictions.

3.1.2. Main and MainMenuIO Classes

The responsibility of the Main class previously was quite varied. It contained the students, courses, course registrations, marks, and professors. It was used as the data store for the entire system. In addition to this, it also handled the main menu loop and loaded the files on startup. In contrast with these responsibilities, the only responsibility of the new Main class is to startup the main menu, which follows SRP.


```
public class Main {

    Startup the MainMenu
    public static void main(String[] args) {
        MainMenuIO.printWelcome();
        MainMenuIO.startMainMenu();
    }
}
```

The MainMenuIO class was introduced into the system to follow the delegation pattern and its sole responsibility is to delegate calls to the other IO classes based on the main menu options. These changes help remove rigidity, viscosity (invoking methods directly from main), opacity and fragility (as Main should not have any conceptual relation to the domain classes).

3.1.2. Manager Classes

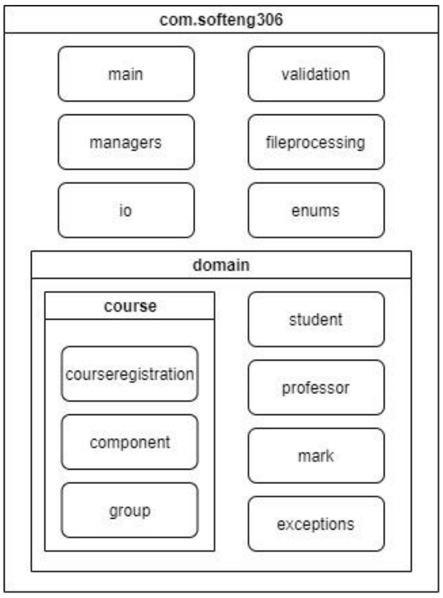
All the manager classes now only encapsulate the logic required to modify the domain classes related to them, for example StudentMgr for the Student domain class. The IO responsibility has been moved out of these classes. The fields stored in the Main class in the as-is system have now been moved to their respective managers for encapsulation. The validation of user inputs has been moved into the respective managers for increased cohesion as the managers can check against their stored list of domain objects whether an object with a given ID exists in the system or not. This helped remove the viscous, immobile and rigid ValidationMgr class (Sections 1.1.2, 1.3.1 and 1.4.2). The manager classes follow the singleton pattern for data consistency in the system.

The old system violates SRP as FILEMgr is responsible for reading and writing every domain class. To solve this issue, the new system splits up the FILEMgr into composable interfaces, which reduces the responsibilities of FILEMgr into smaller sub components.

3.1.4. Common Closure Principle (CCP) - Package Structure

The package structure of the original system was very basic. The only nested package was Enums, which held the enum classes. This was problematic as it meant if anything had to change in the system, then the entire package would be affected.

The proposed package structure groups together similar classes, such as the managers, the io classes, and the file processors. This means if the data source needs to change, then all the classes in the managers package would need to change. Similarly, if the display of the data to the users needs to change, such as converting to a web interface, then the io package would need to be rewritten. Therefore, the components only have one reason to change.



3.2. Open-Closed Principle (OCP)

3.2.1. Interfaces for Variables

Within the original system, there were many places with variables declared as concrete types, such as ArrayList, where an interface could have been used. In the refactored system, we declare all variables as interfaces where possible. For example, all ArrayLists are declared as Lists, and all HashMaps are declared as Maps. This means that if a different implementation needs to be used, none of the functionality within the code needs to be modified. This supports rigidity, as it allows for easy modifications.

3.2.2. Interfaces For all Complex Classes

A large structural change to allow the system to follow the open-closed principle was the addition of interfaces. Every concrete class within the system, except Main, MainMenuIO, RegexValidator, and the Exception classes, either extends an abstract class or implements an interface. This means that for any domain class, manager, or IO class, if the implementation needs to be changed, a new concrete class can be made and used instead. This allows for very easy extension to the system, without having to modify existing functionality.

This change allows for better opacity, as it is very easy to see the intent and meaning of a class, by looking at it's interface. It also significantly reduces the rigidity of the system, as we can very easily change any part of the implementation without affecting other parts of the system. Immobility is reduced, as the system is built as a collection of parts that can be swapped out.

3.3. Liskov Substitution Principle (LSP)

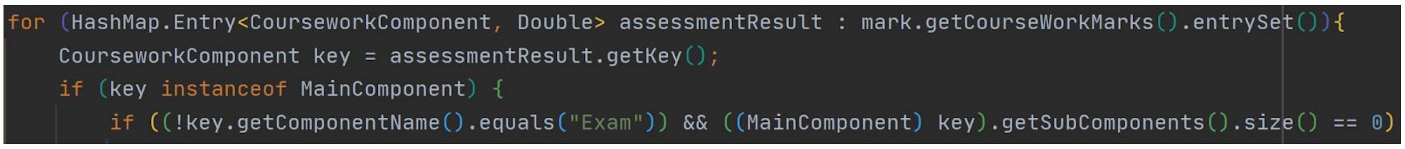
3.3.1. Group Type as an Enum

The as-is system was incorrectly using LSP with respect to the Group hierarchy, resulting in the system being vulnerable to errors. For example, the arraylists for the lecture, lab and tutorial groups used in CourseRegistrationMgr.registerCourse() were declared to be of type Group. This meant that while LSP was being satisfied, an object of type LectureGroup could be added to the labGroups list, without any error checking being performed.

In the refactored system, the group type has been abstracted into the GroupType enum in the Group class, allowing for more effective type checks being performed when dealing with group objects. The advantages of LSP were not being used in the as-is system as there was no case where it would be beneficial for any child type of Group to be used in place of the parent type to achieve a better design. Hence, the type checking was abstracted out into an enum class for better encapsulation. Additionally, this design helped remove the needless repetition and needless complexity design smells. There were identical blocks of code being used in CourseMgr.addCourse() in the as-is system to handle the different group types. This functionality has now been abstracted into a single method in CourseMgr in the reactored system. Additionally, the Group child classes were not encapsulating any functionality or information, leading to needless complexity. This has been resolved with the new Group class design.

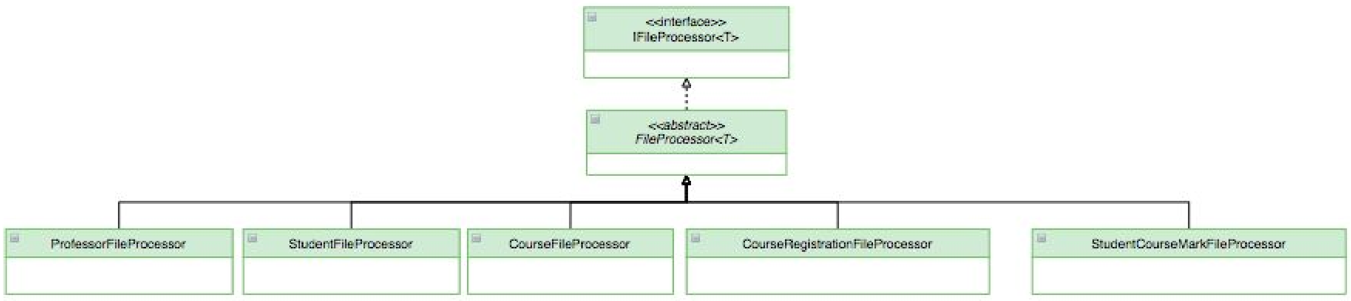
3.3.2. The Component Hierarchy

The component hierarchy has remained unchanged from the as-is system. But the misuse of LSP in the as-is system has been resolved in the refactored system. In MarkMgr.setCourseWorkMark() in the as-is system, a HashMap with the key type of CourseworkComponent was being used to retrieve the marks associated with an assessment. But there was a large amount of typecasting being used to convert CourseworkComponent objects to MainComponent objects as conceptually, it only made sense for MainComponent objects to be used as the key to obtain mark values. This was an incorrect use of LSP as any SubComponent object could also be used as a key, which would be incorrect as then the SubComponent object would be cast to a MainComponent object, whereas that should not be possible. Hence, this incorrect use of LSP was resolved in the refactored system as no unsafe type casting is performed anywhere, hence, following good object-oriented design and practices.



```
for (HashMap.Entry<CourseworkComponent, Double> assessmentResult : mark.getCourseWorkMarks().entrySet()){
    CourseworkComponent key = assessmentResult.getKey();
    if (key instanceof MainComponent) {
        if ((!key.getComponentName().equals("Exam")) && ((MainComponent) key).getSubComponents().size() == 0)
```

3.3.3. File Processing Hierarchy



We have implemented a file processing hierarchy to resolve the serialization and deserialization of our data. We obeyed LSP by allowing each child file processor to implement all the methods defined by our interface and abstract file processor. The ProfessorFileProcessor doesn’t require implementation for the methods, so we have put dummy methods in there for the superclass methods so that we can still substitute it for the superclass. All other child classes also obey LSP as they implement all the methods of the superclass.

3.4. Interface Segregation Principle (ISP)

3.4.1. User-Specific Interfaces

In the as-is system, there were many classes that provided various kinds of functionality for other classes. For example, the ValidationMgr, FILEMgr, and HelpInfoMgr classes were all used by many other parts of the system. In the refactored system, the team has created interfaces that isolate the functionality required by different components of the system. For example, there are different interfaces for each type of file manager, each domain manager, and each area of IO operations. With this segregation, a component in the system needs only to depend on the interface relevant to its operations, and will not have to depend on functionality that it doesn't use. A component that manages students, for example, needs only to use interfaces that provide responsibilities for student-related operations, such as the student file manager and student manager IO. The component will not be exposed to or depend on functionality for courses, professors, or any other part of the domain. This reduces the rigidity and fragility of the system, as changes to functionality will only affect the parts of the system that depend on that functionality. It makes the code easier to maintain, reuse and easier to read with decreased opacity.

3.4.2. User Interfaces for all Classes

Relying on interface types for all classes makes the code more flexible and coherent as when defining a public interface, it can be ensured that only the methods that a particular user needs are a part of the interface. The code is less viscous with the new design as wrong coding practices such as using a dummy stream in the ValidationMgr class can easily be avoided as the manager interfaces don't allow for any IO operations anymore. The benefits of this and the other design smells it removes has been further discussed in section 3.5.2.

3.5. Dependency Inversion Principle

3.5.1. Using List and Map instead of ArrayList and HashMap

Defining variables using the concrete type such as ArrayList and HashMap was representative of the rigidity design smell in classes such as MarkMgr and Course as discussed in section 1.1.1. This design smell was fixed by applying the dependency inversion principle (DIP) in the affected classes. Instead of using the concrete class of ArrayList and HashMap, variable types were declared using the interface types List and Map respectively, which improves both, information hiding and allows for dependency injection to be used in case we wish to initialise objects using a different child class. This means that in the refactored system, no client code needs to be changed in case the object it depends on changes to a different sub-type. As a result, the code has become more flexible to future change through the effect of polymorphism. While this is not reflected in the class diagram, it does remove dependencies from concrete classes. This helped remove the rigidity design smell as modules are no longer associated with concrete implementations.

3.5.2. Associating an Interface with each Concrete Class

DIP has also been applied in the entire refactored system. Each concrete class implements an interface, allowing for all classes in the system to be associated with interfaces of other classes. This is beneficial as interfaces are less likely to change than concrete classes, ensuring that the methods defined in the interface can always be invoked as child classes need to implement them. Additionally, if in the future any new child classes are added to the system, different implementations of the methods can easily be invoked via dependency injection without having to force several changes in multiple modules.

For example, if in the future the university wishes to use a different GPA calculation scheme for one semester (similar to the one used in semester 1 with the grade increase), then we do not need to change the MarkCalculator class. A new class implementing the IMarkCalculator interface can be introduced into the system and no other changes would need to be made to other modules. DIP was used to remove rigidity as modules are now dependent on interfaces rather than concrete abstractions. It also helped remove fragility as interfaces were designed first, meaning that only conceptually correct associations were introduced into the system and no unrelated modules interact with each other. Modules can now easily be extracted from existing modules, helping remove immobility. The design also removed opacity and needless complexity with cleaner code overall.

4. Re-Measured System Metrics for the Refactored System

4.1. Assumptions

- The lines of code (LOC) for every class was calculated by starting the counting from the opening brace after the declaration of the class, and stopping at the line of the closing brace. Similarly, to calculate the lines of code per method, the count was started from the opening brace after the declaration of the class, and stopped at the line of the closing brace.
- Assume that the LOC per method cannot be calculated for interfaces, as they do not implement methods.
- Assume DIT only involves classes that are within the system, and any classes extending classes outside the system are only counted as having a singular parent regardless of how deep the hierarchy is.
- When calculating the DIT value for a class, only inheritance is considered and implementing interfaces does not affect the DIT value.

4.2. Refactored System Metrics

Link to metrics for the as-is (original) system including LOC per class and average LOC per method for comparison purposes:
https://github.com/SoftEng306-2020/project-2-team-8/raw/master/documentation/statistics/Old%20System%20Metrics.pdf

	Metric				
Class	**WMC$_{Fan-In}$**	**DIT**	**CBO**	**LOC**	**Average LOC/Method**
Main					
Main	0	0	1	13	4
Domains					
CourseworkComponent	23	0	0	41	3
MainComponent	29	1	3	36	3.3
SubComponent	24	1	3	24	3
ICourseRegistration	10	0	1	13	N/A
CourseRegistration	11	0	10	60	4.5
IGroup	11	0	5	29	N/A
Group	12	0	5	55	3.4
ICourse	69	0	13	211	N/A
Course	73	0	15	209	3
ICourseBuilder	14	0	3	29	N/A
CourseBuilder	18	0	7	99	3.9
IMainComponentMark	17	0	5	18	N/A
MainComponentMark	18	0	5	76	4.3

IStudentCourseMark	16	0	6	47	N/A
StudentCourseMark	17	0	7	103	8.4
IMarkCalculator	5	0	2	20	N/A
MarkCalculator	6	0	4	104	17
ISubComponentMark	8	0	4	7	N/A
SubComponentMark	9	0	4	32	3.2
IProfessor	6	0	1	22	N/A
Professor	6	0	1	40	3.2
IStudent	22	0	8	64	N/A
Student	23	0	8	74	3.1
Exceptions					
CourseNotFoundException	1	1	N/A	6	3
GroupTypeNotFoundException	1	1	N/A	6	3
InvalidCourseRegistrationException	1	1	N/A	6	3
ProfessorNotFoundException	1	1	N/A	6	3
StudentNotFoundException	1	1	N/A	6	3
SubComponentNotFoundException	1	1	N/A	6	3
Enums					
CourseType	2	0	2	27	8
Department	6	0	7	26	7.5
Gender	2	0	3	27	8
GroupType	15	0	7	25	3
File Processing					
CourseFileProcessor	20	1	6	60	11.3
CourseRegistrationFileProcessor	20	1	6	61	11.3
FileProcessor	20	0	5	26	8.5
IFileProcessor	20	0	5	35	N/A
StudentCourseMarkFileProcessor	20	1	6	56	11.3
ProfessorFileProcessor	20	1	6	40	6.7
StudentFileProcessor	20	1	6	59	11.3
IO					
ICourseMgrIO	41	0	3	205	N/A
CourseMgrIO	61	0	12	900	13.7
ICourseRegistrationMgrIO	11	0	2	47	N/A
CourseRegistrationMgrIO	13	0	5	127	6.8
MainMenuIO	12	0	4	148	15.9
IStudentCourseMarkMgrIO	11	0	1	55	N/A
StudentCourseMarkMgrIO	12	0	3	109	8.2
IProfessorMgrIO	1	0	1	9	N/A
ProfessorMgrIO	1	0	1	11	7

IStudentMgrIO	5	0	1	16	N/A
StudentMgrIO	15	0	7	256	16.3
Managers					
ICourseMgr	27	0	3	73	N/A
CourseMgr	45	0	10	442	11.2
ICourseRegistrationMgr	3	0	1	13	N/A
CourseRegistrationMgr	11	0	10	276	20.7
IGroupMgr	1	0	6	10	N/A
GroupMgr	3	0	6	60	14.3
IStudentCourseMarkMgr	8	0	4	28	N/A
StudentCourseMarkMgr	22	0	11	225	14.1
IProfessorMgr	3	0	3	18	N/A
ProfessorMgr	5	0	6	65	7.6
IStudentMgr	10	0	2	31	N/A
StudentMgr	16	0	6	132	7
Validation					
RegexValidator	8	0	2	74	6.2

4.3. Analysing Metrics

4.3.1. Weighted Methods Per Class: Fan-In (WMC_{Fan-In})

The average WMC_{Fan-In} for the original design was 17.0. In comparison, the new design has an average of 14.8. This is a definite improvement. The highest individual WMC_{Fan-In} value was 85 in the original, compared to 73 in the new system, as the Course domain class still encapsulates a lot of information, but lower than in the as-is system. This improvement in the average WMC_{Fan-In} comes from the separation of large classes into smaller classes. For example, the removal of the ValidationMgr, HelpInfoMgr and FILEMgr which interacted with most of the other classes.

A significant improvement was seen in the WMC_{Fan-In} value for the Student, which reduced from 40 to 23 and that of Mark (now called StudentCourseMark), which decreased from 23 to 17. On the other hand, the WMC_{Fan-In} value of the manager classes has increased. For example, for StudentMgr the value increased from 1 to 16. A similar trend can be seen for the other manager classes. But this isn't necessarily indicative of a worse design as the system has classes with increased abstraction, encapsulation and information hiding and all classes follow SRP.

In the as-is system, the domain classes were handling a lot of the responsibilities of the managers and hence, in the refactored system, these methods have been moved to the respective managers. The domain classes in the refactored system simply represent a real world abstraction with no extra functionality, and all the logic for interacting with the domain objects is encapsulated in the manager classes. Hence, the WMC_{Fan-In} value has decreased for domain classes, while it has increased in the managers, which is what was expected with the new design.

4.3.2. Depth in Inheritance Tree (DIT)

The DIT metric has not changed significantly from the original system. The range of DIT values for both systems are from 0 to 1. DIT decreased for Group from 1 to 0 as the subclasses were deemed to be unnecessary as they provided no extra functionality, and an enum value was used instead to differentiate between different types of Groups. Apart from Group, the DIT value for no other class which also existed in the as-is system has changed. The DIT values of the Component classes have remained the same. The various exception classes are assumed to have a DIT value of 1 as they extend the Exception class, which is a Java standard library class.

The DIT value of the new file processing classes is also 1 as they simply inherit from the FileProcessing class. There was limited scope for inheritance in the proposed new system and hence, DIT is not the most comprehensive measure to compare the improvements made as the values haven't changed much between the systems.

4.3.3. Coupling Between Objects (CBO)

The average CBO is an indicator of coupling, rigidity, and reusability within a system as it shows how many classes rely on the method implementations of the class for which the value is being calculated. A higher CBO generally means higher coupling, resulting in higher rigidity, lower reusability and higher maintenance costs. The average CBO has reduced by 2.0 from 6.9 to 4.9. Because our CBO has reduced, we can infer that the system is in general less rigid, more reusable and less costly to maintain than the previous system.

An example of a class that had a reduction in CBO is the Main class. The main class originally had a CBO of 13, however in the new system it has dropped to 1. This improvement is a result of the proposed MainMenuIO class, which handles the running of the main menu, and the managers being responsible for storing the domain data. The CBO values for the domain classes and managers have increased on average. For example, the CBO value for Student increased from 7 to 8 and for StudentMgr increased from 5 to 6. But considering the several new classes in the system added for dividing the responsibilities and abstracting data better, this small increase is quite insignificant and surprising. This is testament to the design of the new system as the CBO value for the domain and manager classes has only increased slightly with a large increase in the number of system classes.

4.3.4. LOC per class

The calculation for the new system results in 80.5 lines of code per class. LOC is an indication of rigidity because in general the more lines you have in a class, the more lines you will have to change if you want to add or remove functionality from that class. Lines of code is also an indication of immobility. The average LOC in the original system was 173.6. This is a huge improvement overall and means that we can infer that our system has become less rigid overall. Some of this difference in lines of code comes from breaking down large methods into smaller methods, and others are from reducing duplication in methods such as reading in groups of different types.

Although the individual LOC has fallen for CourseMgr, the total code to handle its responsibilities has increased overall with CourseMgr and CourseMgrIO summing to 1342 LOC, and the original being 683 LOC. Therefore, for this particular instance, the metric has worsened. The reason for this is overhead in breaking down the methods into smaller reusable methods, and moving some responsibilities into CourseMgr such as validation.

Another issue that was found in the original code base was unused fields, methods, and classes. For example, the ProfessorMgr class was unused. By removing this unused code, the overall amount of LOC is lowered.

4.3.5. Average LOC per method

Overall average LOC per method fell from 30.2 to 7.5, which is a reduction of 75%. Overall, the LOC differences imply a large increase in readability of the code, as methods and classes are generally handling smaller tasks which reduces opacity. One point of interest is the average LOC per method for CourseMgr. The original system had an extreme value of 163.8, while the current system has an average of 11.2 for CourseMgr and 13.7 for CourseMgrIO. This significant difference is a result of the separation of concerns of IO out of CourseMgr and also the refactoring of large methods to handle smaller tasks.

5. Analyse new Design

5.1. Maintainability

Custom exceptions were introduced to increase the maintainability of the system, as null values were returned previously. This is not maintainable as any developer of that method must account for this situation, but are not told of this possibility. The custom exception forces any developer to handle any errors in retrieving objects which may not exist. Also, a decrease of 2 in the average CBO value for the system indicates less coupling and increased maintainability. Similarly, a lower average $\text{WMC}_{\text{Fan-In}}$ value from 17 to 14.8 indicates a more maintainable system.

The refactored system uses JSON files and serialisation to persist data between sessions instead of the custom serialisation in FILEMgr, which reduces code size and increases readability. However, this change has introduced duplicated data. So even though the LOC for FILEMgr was originally 1,153, and has been reduced to 337 LOC for the file processing package, overall the refactored system may take up more memory. Using JSON makes the code more maintainable as less LOC are needed to make a change and the Jackson library handles a lot of the developer overhead such as data parsing and reading and writing values into the JSON file.

The introduction of different interfaces for each concrete class supports modifiability and maintainability. If any faults are discovered in the program, they will be localised to a given area of functionality. This makes them easier to discover, which makes the code easier to maintain. The use of interfaces also means that if any class becomes outdated, it is easy to use a different implementation.

5.2. Reusability

The 75% reduction in average LOC per method greatly increases reusability, as each method performs a smaller task which may be reused in other areas of the code. This metric also shows that each method itself is more coherent and maintainable as the responsibility can easily be understood and changed.

Something of note is the $\text{WMC}_{\text{Fan-In}}$ for the subclasses is always greater than the corresponding interface. The reason for this is that the other classes rely only on the interfaces, which requires less information, and the subclass performs internal processing using private methods which is not seen by the classes using it. This improves reusability and maintainability. This feature is not present in the original system.

5.3. Coupling

The functionality for different areas of the domain has been separated. For example, FILEMgr has been split into separate file processing classes, for each part of the domain. This takes the original associations to FILEMgr, and divides it between different classes. The FILEMgr class was coupled to every domain and manager class, which gave it a high CBO (12). Now, each individual subclass is only coupled to the relevant domain and manager classes that it uses. Similarly, there are separate classes for managing the different areas of the domain, and for the domain specific IO operations. This has greatly reduced the coupling within the system, as classes need only to be coupled to the classes that deal with the same domain objects.

One area of the system that was not refactored extensively is the CourseworkComponent hierarchy. These classes are coupled with many other classes, which is seen through their 3, 5 and 6 CBOs. Ideally, these classes would be refactored to reduce this level of coupling.

5.4. Coherency

The validator's responsibilities have been split into their respective manager classes, except for regular expression checking, which is handled by RegexValidator. This reduction is shown in the $\text{WMC}_{\text{FanIn}}$ which decreased from 30 to 8.

Functionality defined in the manager classes was altered so that they manage more specific parts of the system. For example, MarkMgr (now called StudentCourseMarkMgr) no longer contains code which directly computes grades; this functionality is now the responsibility of MarkCalculator. Also, the responsibility of printing course statistics was moved from MarkMgr (now called StudentCourseMarkMgr) to CourseMgr, increasing the coherence of functionality which manipulates and retrieves data from courses. This is shown by the lines of code in mark manager decreasing from 380 to 242.

Input and output functionality was removed from manager classes. This allows the logic of managing domain objects to be separated from the user's interaction with the system. Which makes each class more coherent.

Many parts of the new design for marks are incoherent. MainComponent does not manage its own SubComponent marks, and as a result, other classes such as MarkCalculator need to iterate through MainComponents and SubComponents to calculate marks. This is incoherent because functionality which should belong to classes such as MainComponent and SubComponent are instead spread out across many other classes which take on this responsibility. This problem existed in the original system and was never resolved due to time constraints.

Enums now encapsulate a lot more of their work, such as returning lists of strings of all the types. This is an improvement to the coherence of the system as related functionality is now found together inside the enums. This design replaced HelpInfoMgr, which encapsulated many unrelated methods and was very incoherent.

6. Roles and Responsibilities

Name	**Role**	**Responsibilities**
Leon	Testing, PR Reader, Documentation	Created the test suite, and dealt with maven pom.xml. Also implemented continuous integration using maven on GitHub. Generally helped team members debug the testing if needed.
Aiden	Team Coordinator, PR Reader	Organising meetings, creating meeting minutes, assigning tasks, reviewing PRs, resolving team conflicts. Worked on and introduced MgrIO refactor. General code refactoring.
Sreeniketh	Code Refactoring, Report Coordinator, PR Reader	Refactoring, using unused enums, making MarkMgr following SRP and refactoring it, working on the domain model, IO classes and improving the method and variable names. PR reviews, report writing, editing, and proofreading.
Sheldon	Code Refactoring, PR Reader	General code refactoring (moving responsibilities) and reducing duplication in many classes, refactored courseMgr and CourseMgr IO to work with the new architecture. UML. Report writing.
Elisa	Code Refactoring, JSON Implementation, PR Reader	Generic code refactoring, creating file processor hierarchy and JSON serialization, UML, review PRs.
Peter	Code Refactoring, PR Reader	Generic code refactoring, reducing class associations, javadoc and commenting, exceptions, component hierarchy refactor and abstraction.
Max	Code Refactoring, PR Reader, Report Coordinator, Documentation	General code refactoring, validation refactoring, javadocs and commenting for domain classes, interface creation, PR reviews, report writing and formatting.