FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Fachhochschule Aachen, Campus Jülich

Fachbereich 9
Medizintechnik und Technomathematik

Bachelorarbeit

In degree program of "*Angewandte Mathematik und Informatik*"

Redesign of the software „*MICpad*“ applying SOLID principles and MVC pattern: from redesign to a running software

---

Name	Raja Babu Chauhan
Matriculation Nr.	3194210
Supervisor	Prof. Dr. Karola Merkel
Advisor	Ralph Altenfeld (Dipl.-Inf.)

Aachen, 23.08.2021

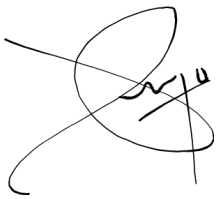
Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Bachelorarbeit mit dem Thema

> Redesign of the software „MICpad“ applying SOLID principles and MVC pattern: from redesign to a running software

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war. Ich verpflichte mich, ein Exemplar der Bachelorarbeit fünf Jahre aufzubewahren und auf Verlangen dem Prüfungsamt des Fachbereiches Medizintechnik und Technomathematik auszuhändigen.

Name: Chauhan, Raja Babu
Aachen, den 18. August 2021



Unterschrift des Studenten

Table of Content

1. Introduction

1.1 Task Overview

We can never design something perfectly. It is impossible to fulfill all the possible objectives for any creation. There are always some trade-offs between requirement, availability and possibility. Even if we find an optimal design, it will be temporary because a design is static, but the universe is chaotic. Provided all the constraints, the best we can hope for our design is to be agile. In other words, we need to design our code, keeping in mind that it would be easy and quick to change the current design. So, redesign / refactor is the fundamental aspect of any software program.

The purpose of redesign is to help in maintenance of a software program. Adding new features, resolving bugs and improving old features are the tasks of maintenance. These operations require the program to be dynamic. In other words, if we add new features, we don't need to modify old features or if we delete an old feature, the program does not crash. This property of being dynamic in nature is generally called loose coupling. It means the modules of a software are not tightly bound, i.e. modules are independent of one another. For this property we need to implement SOLID principles **[Martin]** a short form of five principles (Single Responsibility Principle, Open Close Principle, Liskov's Substitution Principle, Interface Segregation Principle and Dependency Inversion principle).

SOLID principles dictate what a class should do, how a class should interact with another class and what kind of relation it should fulfill. They can be used to check the consistency among modules, create loose coupling and bring high cohesion. In short, SOLID principles make maintenance of a program easy.

Unlike SOLID principles which are used to create designs, the design pattern MVC (model, view, controller) is a ready-made design for the GUI application. The program business logic remains in the model section, while views and controllers are device specific components. Thus, it creates interdependence among modules. This provides the first layer of abstraction to simplify our program. We then use SOLID principles to design and verify if there is any code which requires modification.

The task of this *Bachelorarbeit* is about redesigning a software called *MICpad*. *MICpad* is a GUI editor for *MICRESS* **[ACCESS e.V.]**, a simulation program for metallurgical processes. It provides various functions to ease running simulations and analyzing the result. The task here is to implement *MICpad* in MVC design pattern and make it compliant with SOLID principles. The analysis of this task is already done in the seminar thesis. Now in this bachelor thesis we implement those analyses. After completion of the implementation phase, we should be able to help in the software maintenance and improve the readability of the code.

1.2 Task background

The task is the extension of the work carried out in the seminar thesis **[Chauhan]**. There we have analysed the code of the *MICpad* and have figured out which part of the code required refactoring. We have implemented the process of redesigning for some of the classes to extract the steps which are listed below:

1. Creating the folders and files
2. Mapping the code
3. Removing the empty class / files
4. Fixing the data structure
5. Running the Application

To sum it up, we create an empty project with three subfolders as model, view and controller. We copy each class and file to each of the folders as a first step. Then we go through each class and take a method, and determine which part of the code (model, view and controller) it belongs to. The third step is to remove the empty files and rename some classes. This step is necessary to clean up the classes. Next we fix the data structure for each of the classes, and finally we try to run the application. In order to run the application we start with the view as it can be run with the least dependencies.

The next part of the task is the verification of the newly created classes with SOLID principles. At first, we verify each class with the Single Responsibility Principle (SRP). For this, we check for each class whether it does only one task. We count the number of methods in the class. If the class does only one task, then the number of methods should also be low per class. In the case of many methods, we divide the class into smaller classes. After verifying SRP, we verify with a second SOLID principle called Open Close Principle (OCP) which states a class should be open for extension but closed for modification. This principle is more about how to add a new feature. It promotes the use of interfaces over making a child class to override the parent methods. Since we can't know in advance which features can be added, we examine with a new feature which will be added in MICpad and analyse how the existing design allows us to add new features without modifying the existing code. The third principle to verify is the Liskov Substitution Principle (LSP) in which we see the parent-child relation and determine whether the parent class can be replaced with child class or not. This principle is all about consistency between parent and child classes. It says a child class should be able to replace its parent class without giving errors. According to the fourth principle: Interface Segregation Principle (ISP), the interfaces in the class should not have many methods. We verify this by listing all the interfaces and counting the number of methods it contains. If any child class is forced to implement a method it does not need, we need to refactor the interface into smaller interfaces. Finally we verify the Dependency Inversion Principle (DIP) by checking whether a high level module does not directly use the low level module and vice versa. For this we generate a graph of all the classes using all other classes. We then group the classes according to the module. We check that if a class which belongs to one module uses the class of another module, then this principle is violated. We solve this by creating an interface to make these classes communicate.

At last, we check whether our code has improved by measuring the quality of code through static code analysis. For this, we measure the cyclomatic complexity and lines of code per class and per method. These quality measuring metrics are described in Section 2.2. For example, if the number of lines per class has reduced after redesigning, our code has improved. In the same way, we also measure the complexity number per method. If the complexity number is less than before redesigning, we conclude the complexity of the code has reduced which would make the code easier to understand. With these numbers we would also be able to figure out which part of the code requires refactoring in the future, i.e. the part of code which is still complex and difficult to understand.

2. Tools and Concepts

There are not many software tools used during implementation. The language is C++. The framework used is Qt. We use one of the Python libraries to measure the quality metrics. We also write some routines to determine the progress of the project.

2.1 Software tools

MICpad is written in C++ with the help of GUI library Qt **[Qt]**. Qt comes with an IDE (integrated development environment) called Qt creator. It is used for building GUI applications for mobiles, desktop and embedded platforms. It provides features like syntax highlighting, code completion and warning regarding memory leaks which are very important in writing C++ code. Furthermore, it provides visual debugger, and visual GUI creating tools called Qt designer (following the principle of WYSIWYG).

We use Python to measure the implementation progress. To measure the code quality, we use a Python library called lizard **[Yin]**. It measures cyclomatic complexity which is a metric to measure code quality. This way we can compare new design with old design and we would be able to see how much the code has improved.

GitLab **[GitLab]** is a git repository manager tool. Git is a version control system which keeps track of different versions of code. It also provides various functions for a team of programming staff collaborating on the same project. The code of MICpad is hosted on an internal GITLAB server.

Draw.io **[Alder]** is a free online software application for building diagrams. It is used to make class diagrams, control flow graphs, tables, and increments tracking.

yUML **[Pocketworks]** is an online tool for creating simple UML diagrams with simple syntaxes. It is used to make the dependency graph for verifying the dependency inversion principle.

Ipywidgets **[Project Jupyter]** is a Python library for making GUI interfaces in jupyter notebook. It is used to make a small GUI for some preprocessing tasks while implementing the MVC design patterns.

2.2 Metric

Just like in any implementation, we also need to keep track of progress. Furthermore, we need a way to measure the quality of the newly designed project. This way we can compare the code quality before and after redesigning. Thus, we need at least two metrics for the implementation: one for tracking the progress and another for checking the code quality.

2.2.1 Progress metric

There are different features of the statical code like number of lines of code (LOC), number of methods, number of classes, etc which can be used for measuring the progress. For example, we use the LOC of redesigned MICpad and of old MICpad and get the ratio to calculate the progress. For example, if we have added 200 lines in the redesigned MICpad and the old MICpad has a total of 20000 LOC, then the progress would be 1%. This is fairly simple enough but LOC is very unstable. Since we are redesigning and refactoring old code, then clearly the final number of LOC would not be the same as LOC of old MICpad. For example, when we implement the first step of redesigning to MVC pattern: Creating files and folders, we make a copy of the same file three times once in each module: model, view and controller. This would make the number of LOC greater than the old MICpad and we get unexpected progress greater than 100%. The same goes for other features like number of methods, number of classes, etc.

However, if we look at the use case of MICpad, we know for sure that after redesigning it should perform all of its functionality. Thus, we can count the number of use cases of the old MICpad and the number of use cases of *MICpad* while designing and make a ratio to get the progress of the project. For example, if the old MICpad has 50 use cases it must fulfill, and while implementation we finished 10 use cases, then the progress would be 20%. This way we can keep the track of the progress of implementation. Mathematically,

$$progress \quad = \quad 100 \quad \times \quad \frac{number\ of\ use\ case\ completed}{total\ number\ of\ use\ cases}$$

The 100% of this approach means the implementation task is completed. But the downside of this approach is that not all use cases have the same weight. To put it more simply, not all use cases require the same number of lines of code or take the same amount of time and effort.

2.2.2 Quality Metric

We know the fundamental properties of a good program are loose coupling and high cohesion. In order to achieve this we use SOLID principles. These principles are like stepwise implementation and verification of those properties. However, it is very difficult to quantify SOLID principles because these principles themselves are interpreted differently by different people. For example, the single responsibility principle (SRP) says a class should do only one task. But it does not specify what is meant by one task. Everyone is free to choose what one task would mean for a certain class. Because of this, it is difficult to quantify SRP values.

Even if we can't measure SRP, we still can agree that if a class does one task, the total lines of code would be no more than a certain number, say 200 lines. If it is more than that, it is probably performing a lot of tasks. This way we can quantify the SOLID principles with some metrics which have direct positive correlation with SOLID principles. These metrics are also called static properties of programs **[Hopkins]** i.e. properties which can be directly measured from the source code. These properties are source lines of code (LOC), path count, cyclomatic complexity, etc. These are not perfect metrics but sufficient enough to compare old design with new design.

Cyclomatic Complexity

There are different ways a code can get complicated and complex. Complicated codes are difficult to understand. However, not all complicated codes are complex. For example, if we have a function with 10 lines code but the variables names are $a, b, c,$ .. Then the function would be difficult to understand but it is not complex.

So the complexity here is not about naming issues but rather the number of independent paths or the different ways a method or a piece of code can run. This is what actually makes a code complex. For example, if a method has 10 if-statements there are $2^{10} = 1024$ possible ways it can execute. This makes the code difficult to manage.

There are different methods to measure the complexity of a code. The most famous method is cyclomatic complexity. Cyclomatic complexity **[MCCABE]** is a quantitative measure of logical complexity. It is based on graph theory. It gives the number of independent paths in code. To calculate cyclomatic complexity we take the code to create the control flow graph (CFG), and with the help of this graph we can then calculate the complexity as follows.

$$M = E - N + 2P$$

where

E = the number of edges of the graph.
N = the number of nodes of the graph.

P = the number of *connected components.*

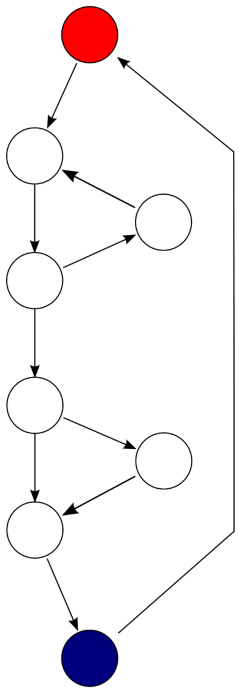


Fig 1 Graph with one connected component **[Wikipedia 1]**

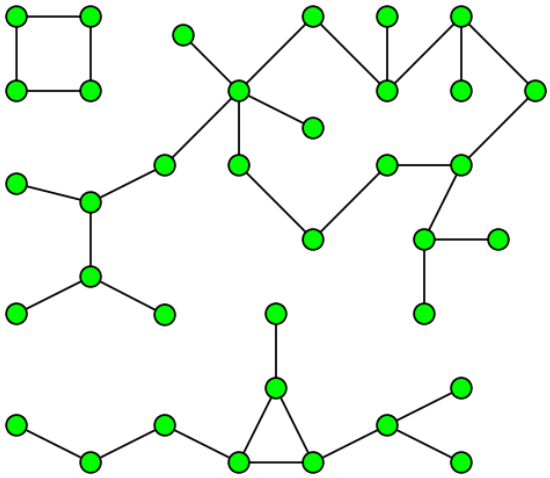


Fig 2 Graph with three connected components **[Wikipedia 2]**

Connected components are the number of individual subgraphs which are connected by edges in a graph. An example with 1 vs 3 connected graphs is shown ***Fig 1*** and ***Fig 2***. Generally, we always have one graph for program code i.e. P = 1, thus we can simplify the formula as:

$$M = E - N + 2$$

In ***Fig 1*** we can see the number of independent paths from start (red node) to end (blue node) is 3 (one is straight, another one is first loop and third one through second cycle). Thus, the complexity is 3. With the help of formula we also get the same result; $m = 9 - 8 + 2 = 3$

In order to calculate cyclomatic complexity of some source code, we need a control flow graph (CFG). CFG is a directed graph created with each statement as node and transition from one statement to another as edge. With each conditionals (loops , if else statements) they form new paths. We take the code from ***Listing 1*** and generate its CFG (***Fig 3***).

```
def error(X, y):
    out = Perceptron.predict(X)
    n = 0
    for i, val in enumerate(y):
        if(val != out[i]):
            n += 1
    return n/len(out)
```

Listing 1: Code to calculate error for a binary classifier

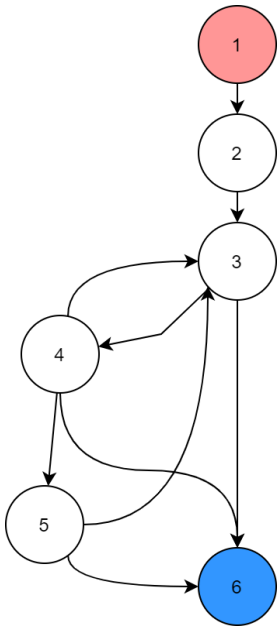


Fig 3: Control Flow Graph for ***Listing 1***

Now with the help of this CFG (***Fig 3***), we can calculate the corresponding complexity value as follow

$$n = 8 - 6 + 2 = 4$$

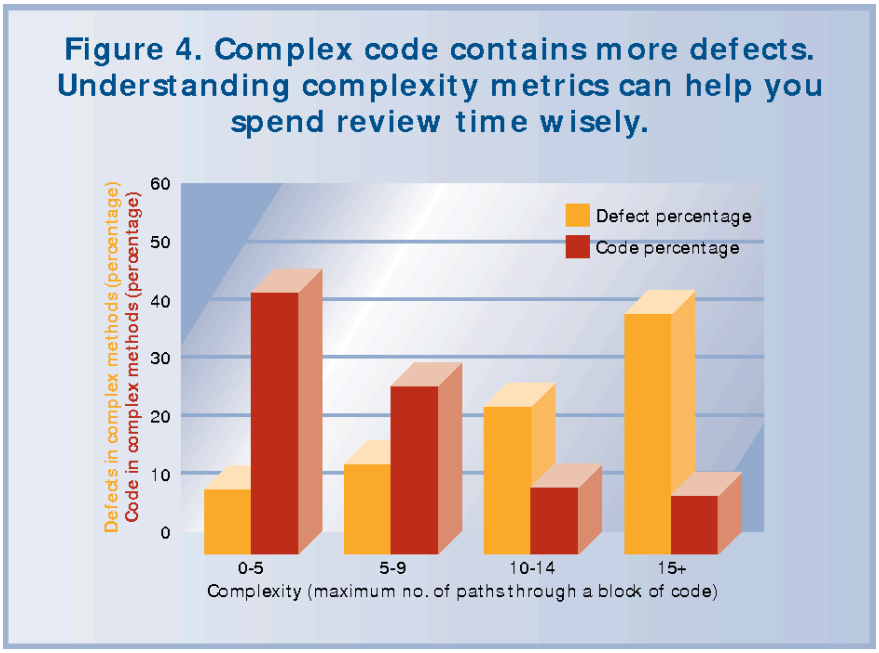


Fig 4: defects vs complexity **[Schroeder,** Fig 4**]**

Lines of Code (LOC) per method / class

LOC per class or per method is the simplest metric. Although we could not use this metric to calculate the progress, we can use it to calculate the quality of the code. In General,a class with fewer lines of code is easy to maintain. This is because of the Single Responsibility Principle (SRP) of SOLID principles which states a class should do only one task. This makes a class hardly have many lines of code.

Higher numbers of LOC also have a direct correlation with cyclomatic complexity. For example, if a method has **m** cyclomatic complexity, just by increasing some lines of code say **k,** we increase the complexity by many factors. Let's say **k** contains **n** more cyclomatic complexity. The new cyclomatic complexity is **m * n** not just **m + n**. However, if the new lines were in a new method the complexity could have been only **m + n.** Thus, more lines of code per method also tends to make code difficult to maintain.

Although we can argue there should be an optimal number of LOC per method, there is no valid argument for optimal number of LOC per class. We can however deduce the optimal number of lines per class from ***Table 1*** which displays statistics of famous libraries of Java. If we take the average of mean LOC per class, it would be 130 with an average deviation 160. In short, the maximum number of LOC per class should be around 290. The same result can be also seen from ***Table 2*** which is around 286 **[Schroeder]**.

	junit	fitnesse	testNG	tam	jdepend	ant	tomcat
max	500	498	1450	355	668	2168	5457
mean	64.0	77.6	62.7	95.3	128.8	215.9	261.6
min	4	6	4	10	20	3	12
sigma	75	76	110	78	129	261	369
files	90	632	1152	69	55	954	1468
total lines	5756	49063	72273	6575	7085	206001	384026

Table 1: Java Libraries code summary **[Bob]**

Table 1. Benchmarks derived from analysis of 18 PowerBuilder systems.

Metric	System average	90th-percentile benchmarks: Class	90th-percentile benchmarks: Method
Class and method size metrics			
LOC per class	145	286	N/A
LOC per method	21	N/A	33
Function calls per class	67	127	N/A
Function calls per method	9	N/A	14
Number of methods per class	10	21	N/A
Public method count per class	N/A	18	N/A
Number of attributes per class	4.40	8	N/A
Number of instance attributes per class	N/A	8	N/A
Coupling and inheritance metrics			
Class fan-in	N/A	4	N/A
Class fan-out	2.98	6	N/A
Class inheritance level	3.44	5	N/A
Number of children per class	0.88	1	N/A
Class and method internals			
Number of global/shared references per class	0.38	0	N/A
Method complexity	0.13[1]	N/A	7
Number of public attributes per class	2.75	4	N/A
Lack of cohesion among methods	0.89	1	N/A
Class specialization index	0.33	0.5	N/A
Percent commented methods[2]	63%	N/A	N/A
Number of parameters per method	N/A	2	N/A

[1] The system-level value is the percentage of methods with complexity of 10 or greater.
[2] This metric represents a lower bound.

Table 2: defect analysis **[Schroeder, Tab.1]**

2.3 Implementation process model

There are different stages in software development life cycle (SDLC). The important stages are

1. Requirement analysis
2. Planning
3. Software design such as architectural design
4. Software development
5. Testing

SDLC model describes how and in what order these stages would be carried out. They ensure the success in the process of software development. The most frequent SDLC models are waterfall, iterative, incremental, spiral, agile etc. Each model is suited for different kinds of software tasks. For example, if the task is small and exact like calculating the shortest route or minimum price between cities, one can use a waterfall model. When the requirement is not clear but evolves with time, a spiral model can be used. If the project is very large, requirements are not clearly defined and requires constant feedback from owner and consumers, an agile model can be used.

We use the incremental model for this redesign implementation. This is because the requirements are well-defined, the program contains many features which can be independently added. This way we can always get a running application with the addition of each increment. Furthermore, after adding each feature we would be able to calculate the use case progress metric along with LOC progress (see **Section 2.2.1**) to keep track of the implementation progress.

Incremental Model

The ***Fig 5*** illustrates how incremental approach and iterative approach works. Generally, it is easy to understand the incremental model by comparing it with the iterative model. In the first row we can see the framework of the picture is made first and then only the contents are filled and improved with each iteration. This is an iterative approach. Meanwhile, in the second row of the picture we can see the picture is divided into parts. When the first part is completed then only the next part is started. This is an incremental approach.

The iterative model is useful when the requirements are not clear in the beginning. With each iteration, the program comes to completion. Incremental model is helpful if the requirements and modules / boundaries are already clear. The entire task needs to be divided into individual independent features. These are also called increments. Each increment goes through all the stages of development: analysis, planning, design and testing. After each increment the product provides more functionality.

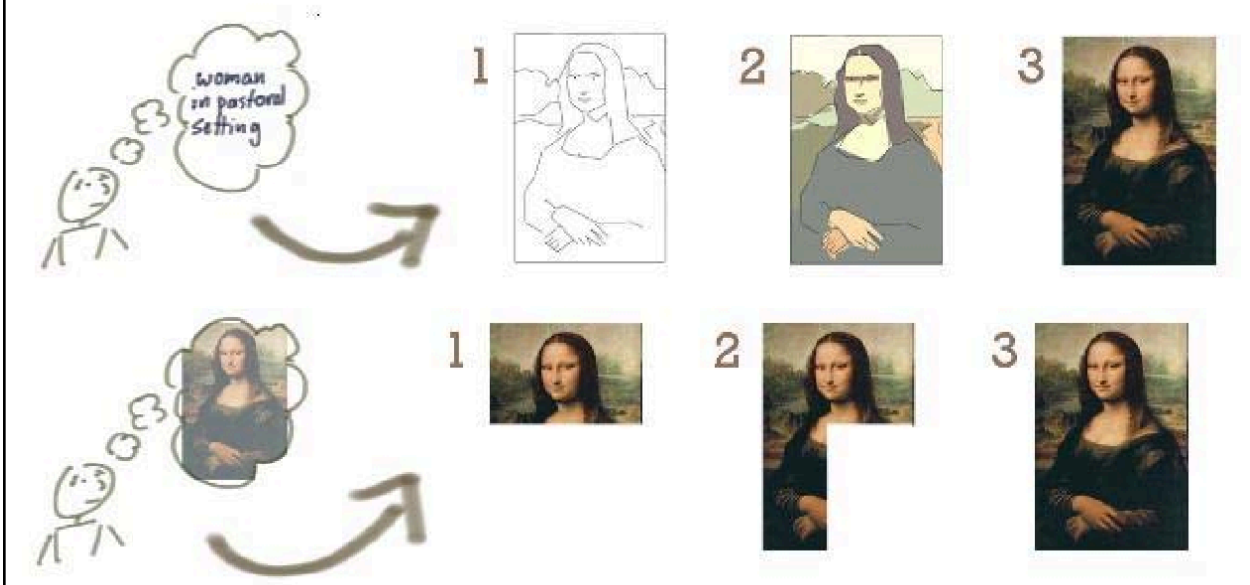


Fig 5 : Iterative vs Incremental approach **[Ljubovic, Fig 1]**

3. Implementation process

Steps of execution:

1. Creating the folders and files
2. Mapping the code
3. Removing the empty class / files
4. Fixing the data structure
5. Running the Application

We go through each step described in the seminar paper. These steps are listed above. As we can see, not all steps are the same, i.e. some steps are simple while others require more time and effort. For example, making the copy of the original file can be done by creating a small subroutine in Python. Thus, we classify the entire steps into preprocessing steps and the main redesigning steps. The first three steps are preprocessing steps: making a copy of the original file, mapping the code and removing/renaming files. The last two steps are the main steps: fixing the data structure and running the application.

For the implementation of the main steps we use the incremental model where we divide the entire tasks in increments and for each increment we calculate implementation progress. These groups are further explained below.

3.1 Preprocessing

In the first step of preprocessing, we create three folders: model, view and controller. We create an empty file in each of these folders for every file from the old *MICpad*. For example, if there is an editor.cpp file in the old MICpad, we make one empty editor.cpp in model, one in view and one in controller. Altogether we make three empty files with the same name. This step is done easily with a Python subroutine.

Fig 6: Summary of old *MICpad*

The next step is to map each function from the old MICpad to the corresponding modules (model/view/controller) in the new MICpad. There are around 55 files and 47 classes in the original project. So doing this manually might take a lot of time since we are going to create 3 * 55 files and more for some classes. This step is also a little confusing. This is because to do so, we need to open four files with the same name in a text editor. For example, we need to open *editor.cpp* from the old MICpad and one each *editor.cpp* from modules model, view and controller. Altogether we have 4 files of the same name in a text editor. Since we need to switch to the correct module when pasting, the chances of making mistakes is very high. Since there are 55 files and we need to keep track all time, we make a simple GUI ***Fig 7*** which would do the task in background. With this GUI we only need to open one *editor.cpp* from the old MICpad, then we copy/cut the content from here and on clicking the correct module button, it copies clipboard content to corresponding file and module.

Fig 7: GUI created for implementing the first two steps

And as for the third step, we go through each file in the new MICpad and check whether the file is empty. If the file is empty we remove the file and if the file is not empty, we rename the file. For example: the files of name *editortab.cpp* are named to *EditorTabView.cpp, EditorTabController.cpp* and *EditorTabModel.cpp.*

3.2 Main Task

MICpad usecases

1. new tab
2. open
3. reload
4. recent file
5. save
    1. nav update
    2. update results if result directory is changed
    3. closed plot if result directory is change
    4. check for conflicted file
6. search in driving file
7. save as
8. close
9. exit
10. start
11. stop / kill
12. refresh result
13. automatic refresh
14. color scheme
15. show result
16. convert 6.4 to 7.1
17. check 6.4
18. check 7.1
19. copy IN to DRI file
20. phase Diagram
21. 2D anisotropydiagram
22. online documentation
23. about
24. select binaries
25. font size change
26. remove sim tab
27. show result tab
    1. show table
        1. plot colum
        2. export column to csv
        3. hide / show columns
    2. show text
    3. close plot on closing the sim tab
28. give warnings on closing running sims

Fig 8: use cases / functions list of *MICpad*

Increments

Increment 1

- main GUI
- new tab
- reload
- slotOpen
- config widget
- show results navigation
- result display area

Increment 2

- Save
- saveAs
- close
- exit
- phase diagram
- anisotropy diagram
- close Simulation tab functionality
- show results on selecting on result navigation
- plot columns
- expoting columns
- font change

Increment 3

- start simulation
- stop simulation
- pause simulation
- refresh results
- update results for running simulations
- update plots

Increment 4

- colorscheme
- syntax conversion
- syntax checking
- about
- online documentation
- copyINtoDRIFile Options

Fig 9: tasks divided into increments

For the main task: the fourth and the fifth step of the implementation, we use the incremental model described earlier in **section 2.3**. This is because these steps require carefully running and implementing individual classes and finally combining them together to run as a whole. For this we divide the entire process into small increments as described by the incremental model. Each increment is defined with tasks (***Fig 9***). With the completion of each increment, we would be able to have a running application. After each increment we measure the progress metric based on use cases (***Fig 8***) to keep the track of the implementation progress.

3.2.1 Increment 1: **Main View and Layout**

The first increment tasks are to implement the main view overview. To do this we also need to refactor the main controller i.e. *MainEditorController* partially, especially *open file* and *new file* menu so that we would be able to implement 90% of the view. The rest of the view is added as new functionalities are added. So the task of this increment is to focus on the MVC code separation for the main functions like *open file* and *new file* (***Fig 11***) and to display the main view first.

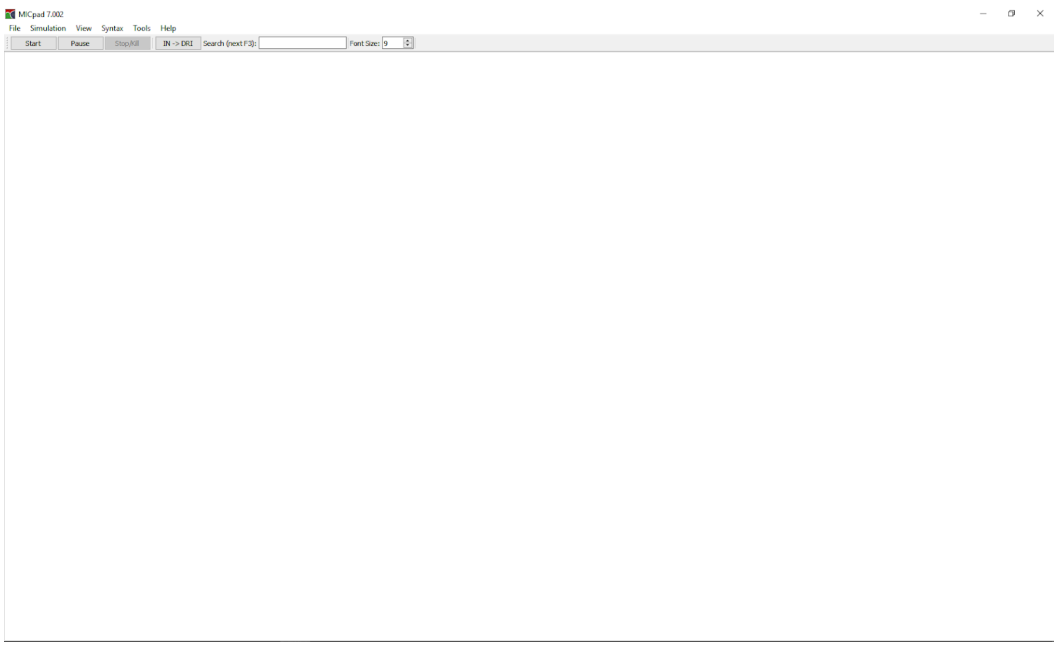


Fig 10: Main View (no editor)

At first the main GUI without any editors is refactored which is shown in ***Fig 10***. This main window contains all the menus bars and toolbar views (***Fig 11***). Since the callback functions are not added yet, the menu options don't perform any operations. This is however only a small part of view as it is still missing the layout for file editing. To implement the other view, the main editor view for editing the driving file, the $new\ file$ menu option is refactored in *MainEditorController*. The result of this implementation is shown in ***Fig 12***. Finally the $open\ file$ functionality is added.

This task is further divided into mapping the code for input navigation, input content area, configuration widget, output navigation and output result area. The ***Fig 13*** and ***Fig 14*** shows the result of these features.

With the separation of main view, editor layout, binary selection and result layout from the old MICpad, we complete around 13.15% of the task in this increment.

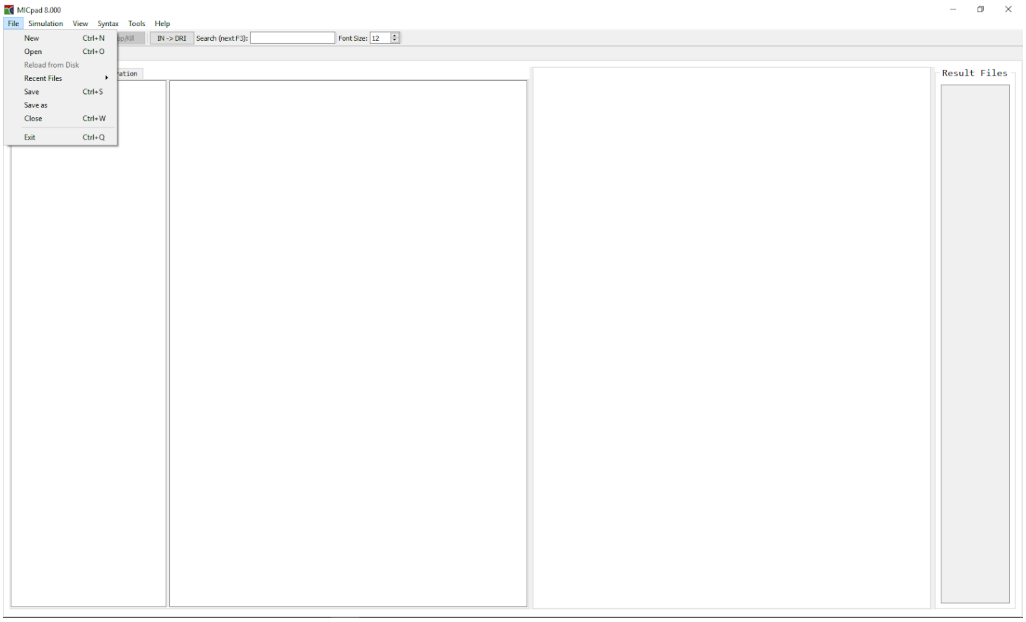


Fig 11: File menu options

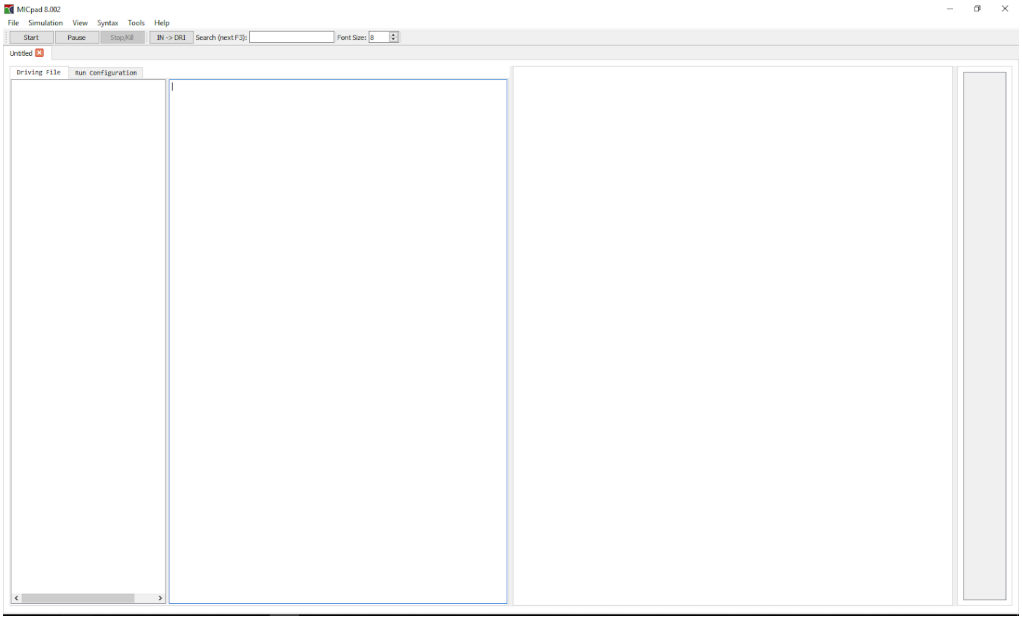


Fig 12: Editor View (on selecting new file)

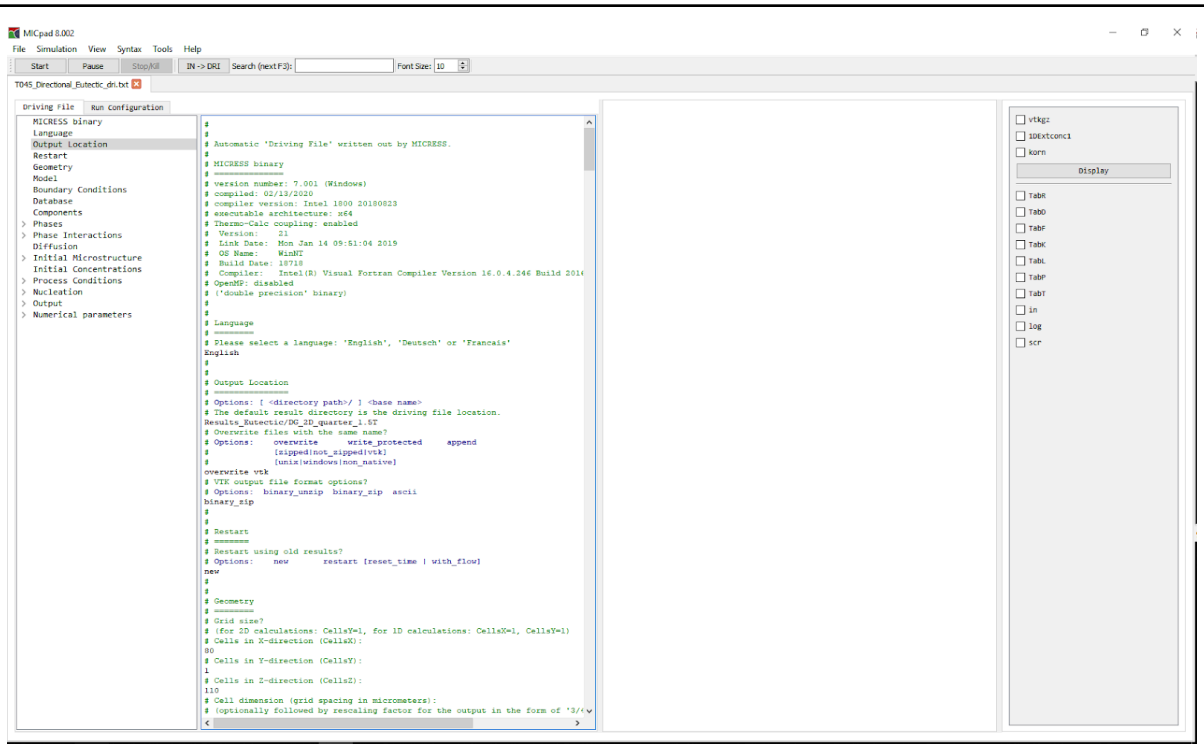


Fig 13: Editor View (on selecting open file)

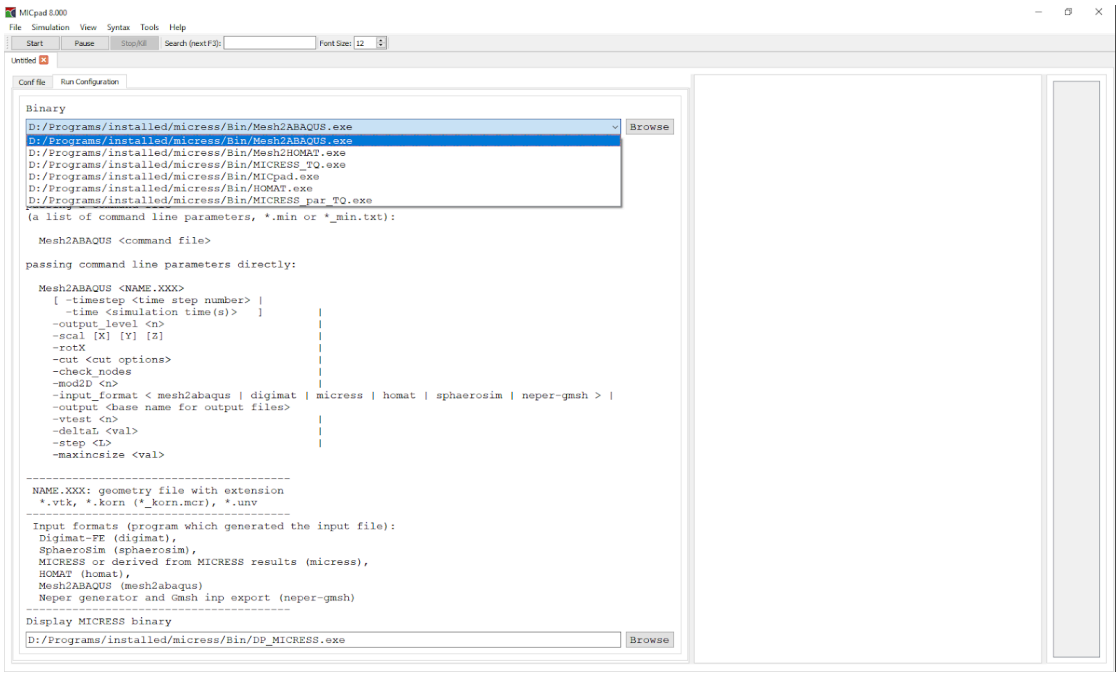


Fig 14: Configuration widget for selection different kind of MICRESS binary

3.2.2 Increment 2: **Static GUI functionality**

In the second increment, we separate more statical GUI functionalities like saving file, closing an editor, updating the diagram (phase diagram and anisotropy diagram from Tool Menu) and displaying results with its functionalities from old MICpad.The results display can be subdivided into tasks like display the results of simulation on selecting the result navigation.

There are two kinds of result files: binary and text files. These text files are also either normal text or table data. The text files are displayed the way it is (***Fig 16***). The tabular text result is displayed as a table (***Fig 15***). The table representation also contains a context menu like plotting, exporting, hiding or showing the selected columns (***Fig 17***). Finally we filter out the font size change functionality to change the font size of all the widgets (***Fig 19*** and ***Fig 20***) and phase diagram menu from the toolbar (***Fig 21***).

With the completion of these redesigning tasks, we complete about 60% of the task. We carry out the refactoring process in the third increment where we make these results and diagrams live on running the simulation.

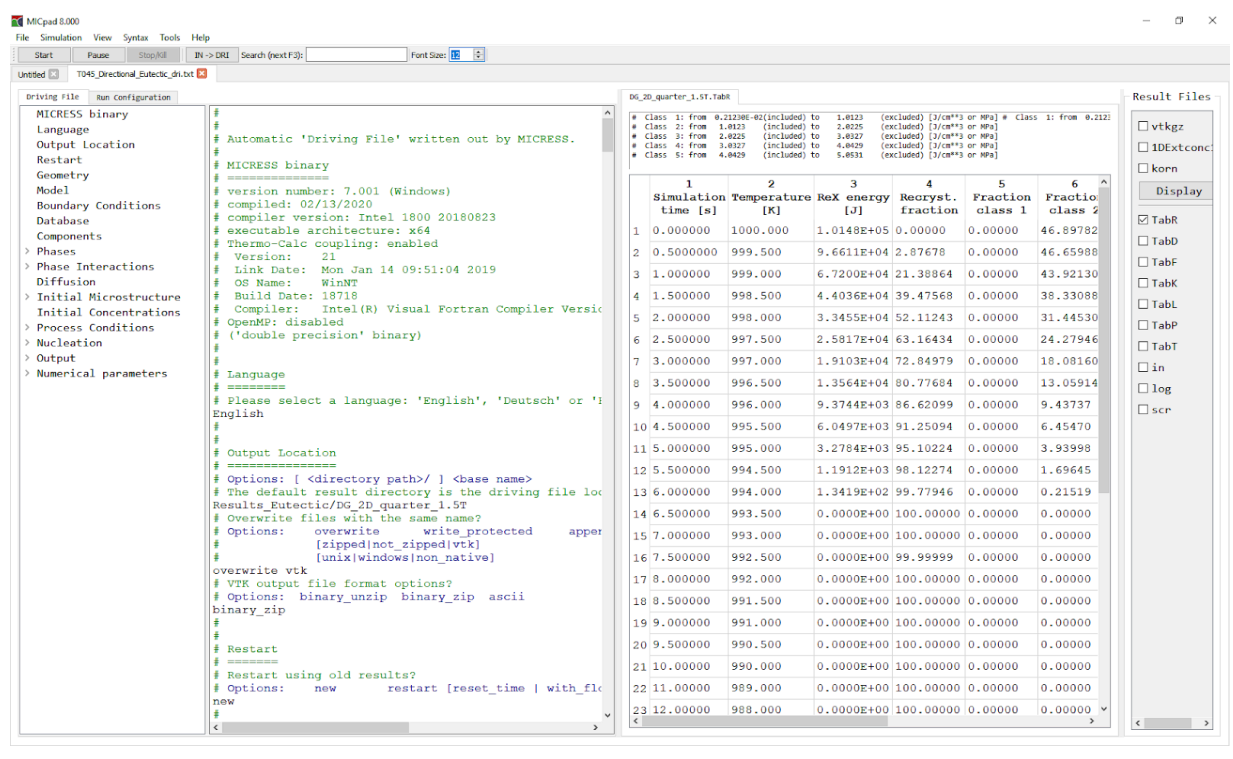


Fig 15. result display (on selecting output navigation)

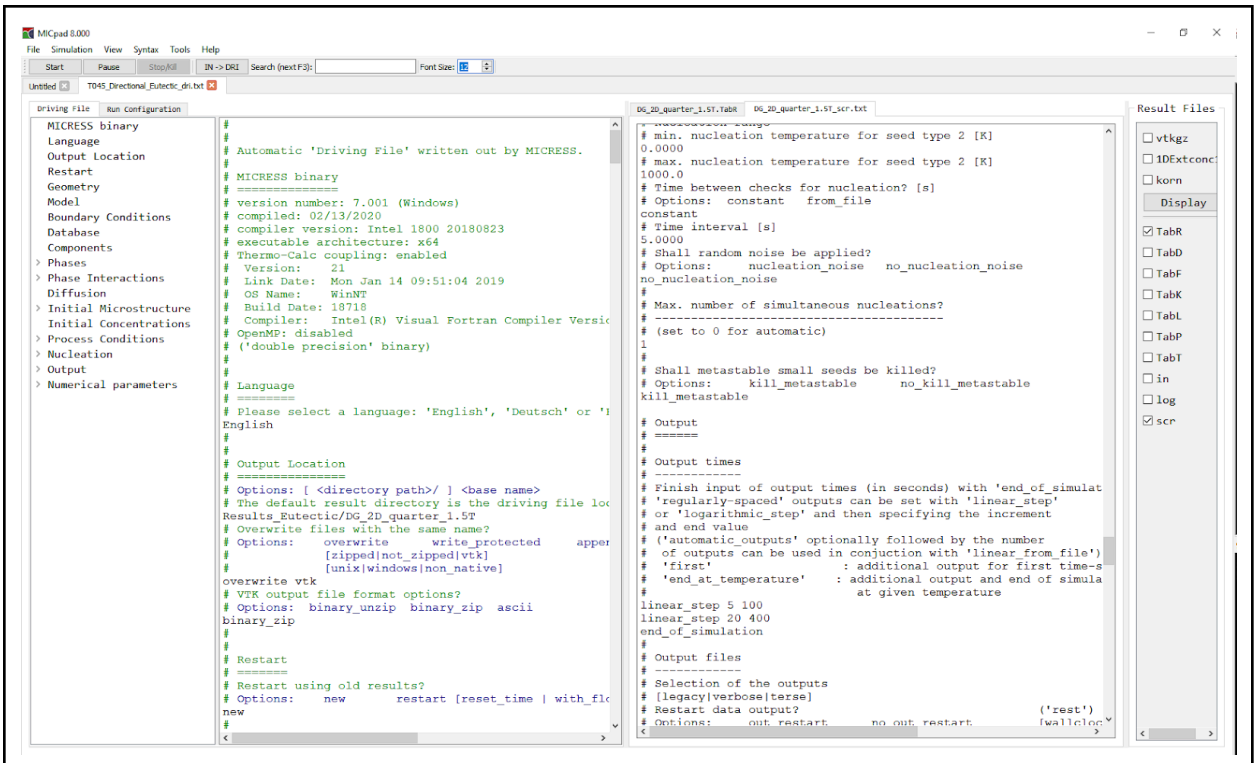


Fig 16. Result displayed as text

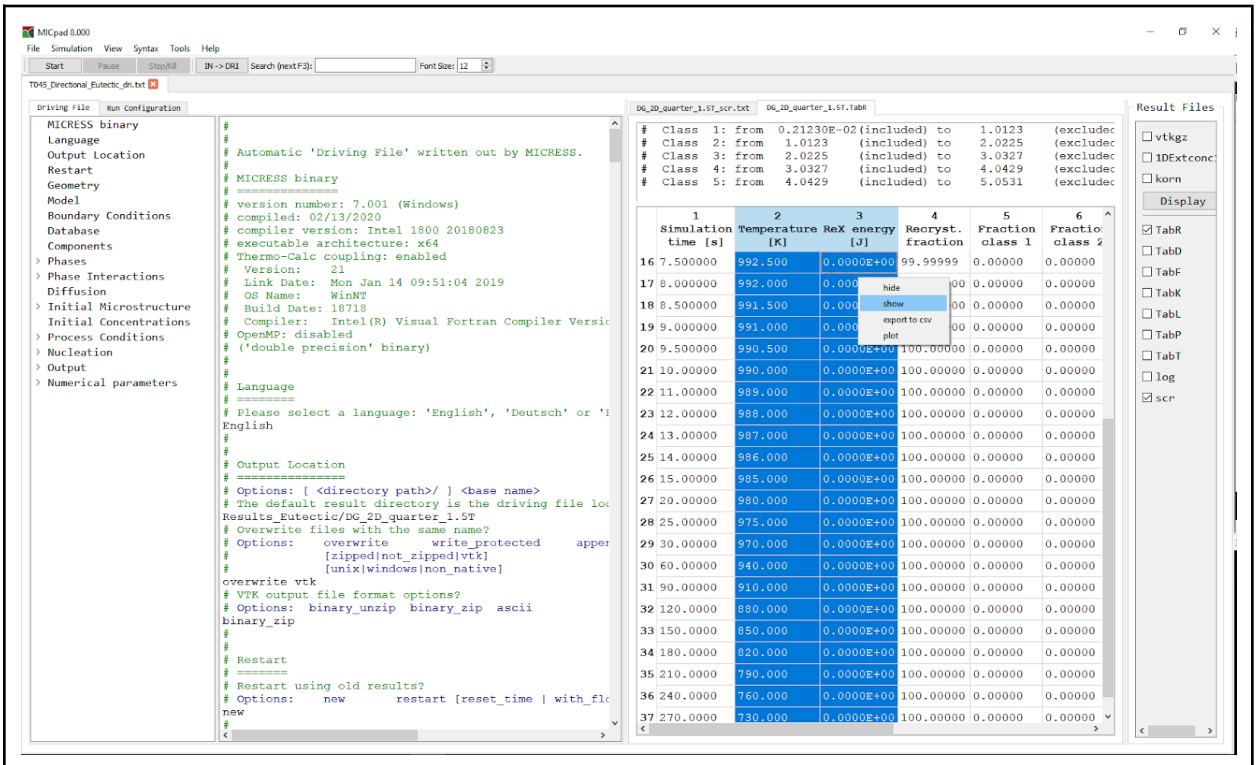


Fig 17. Context menu on result files

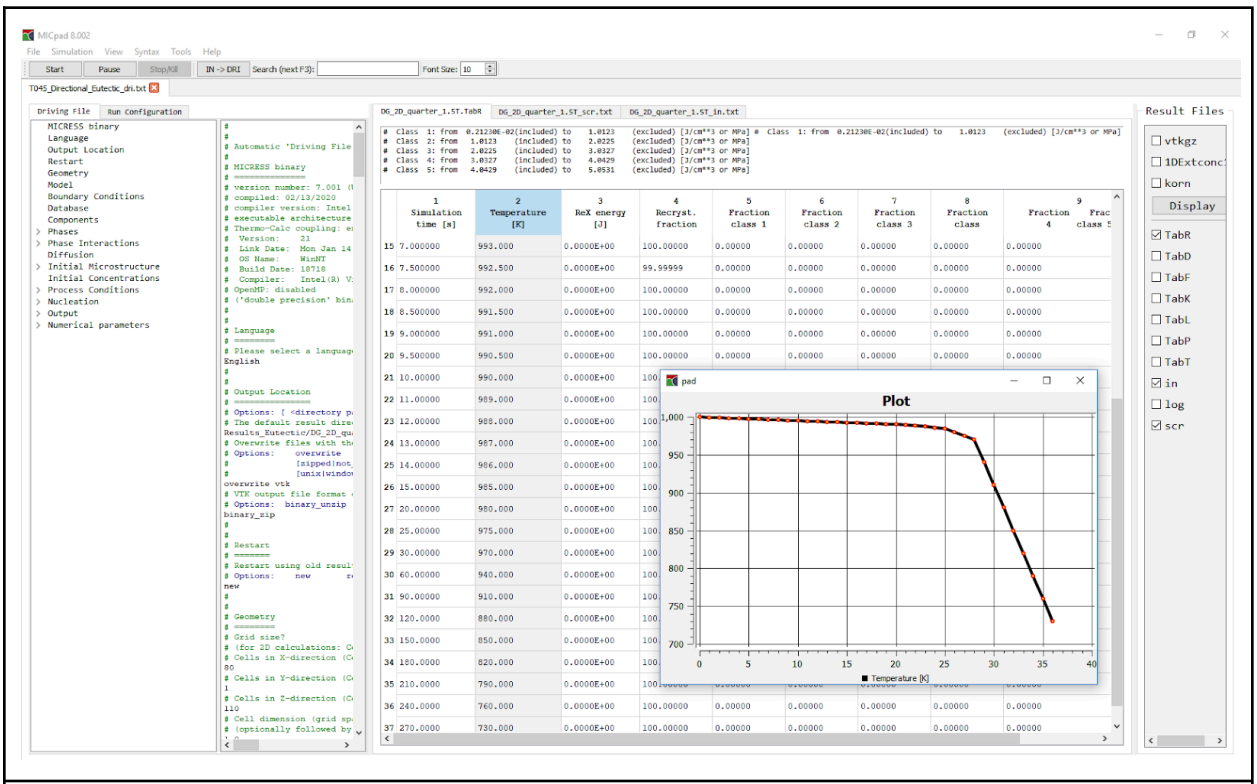


Fig 18. Plot function for tabular results

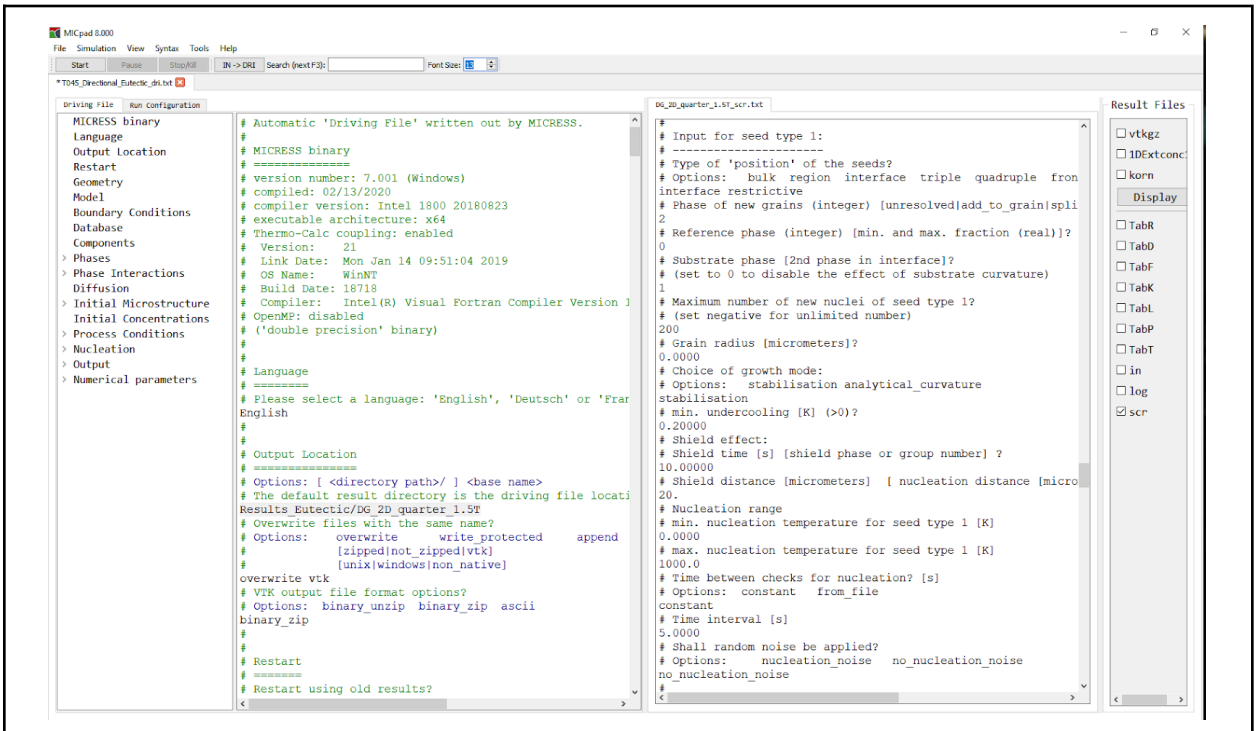


Fig 19. Font change 1

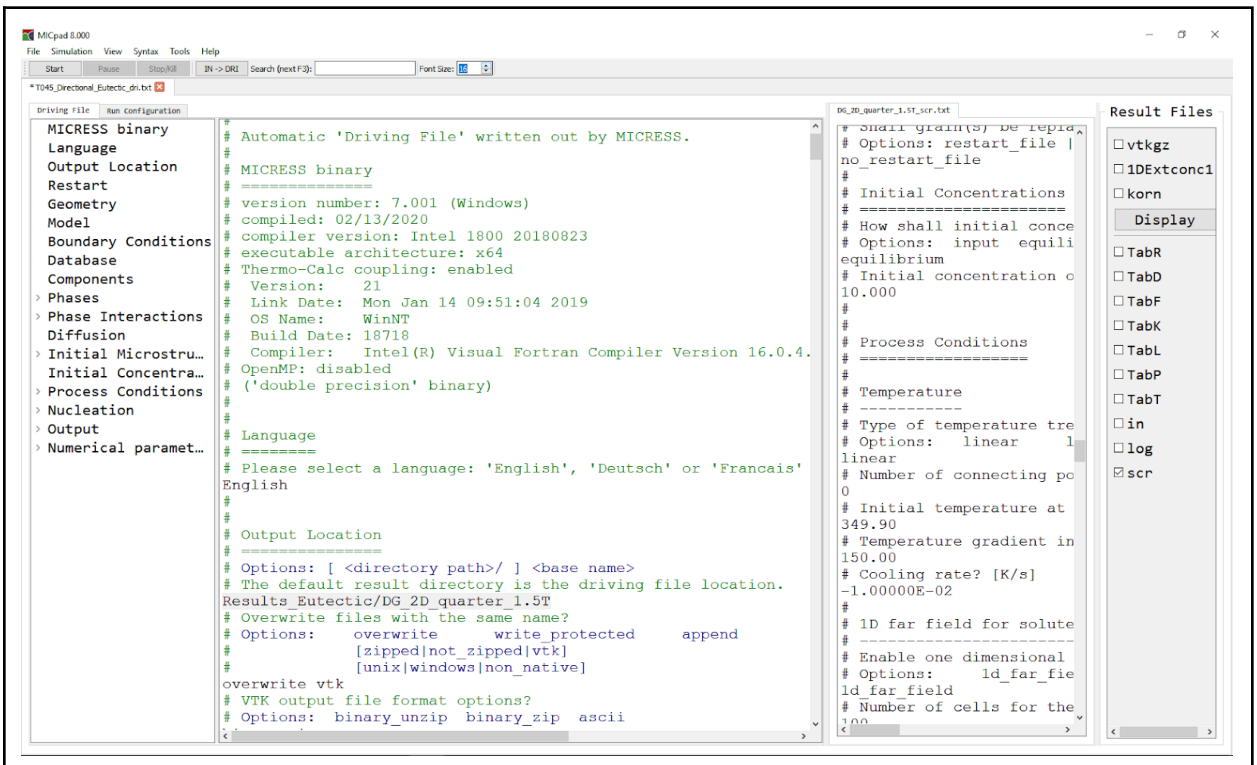


Fig 20 Font change 2

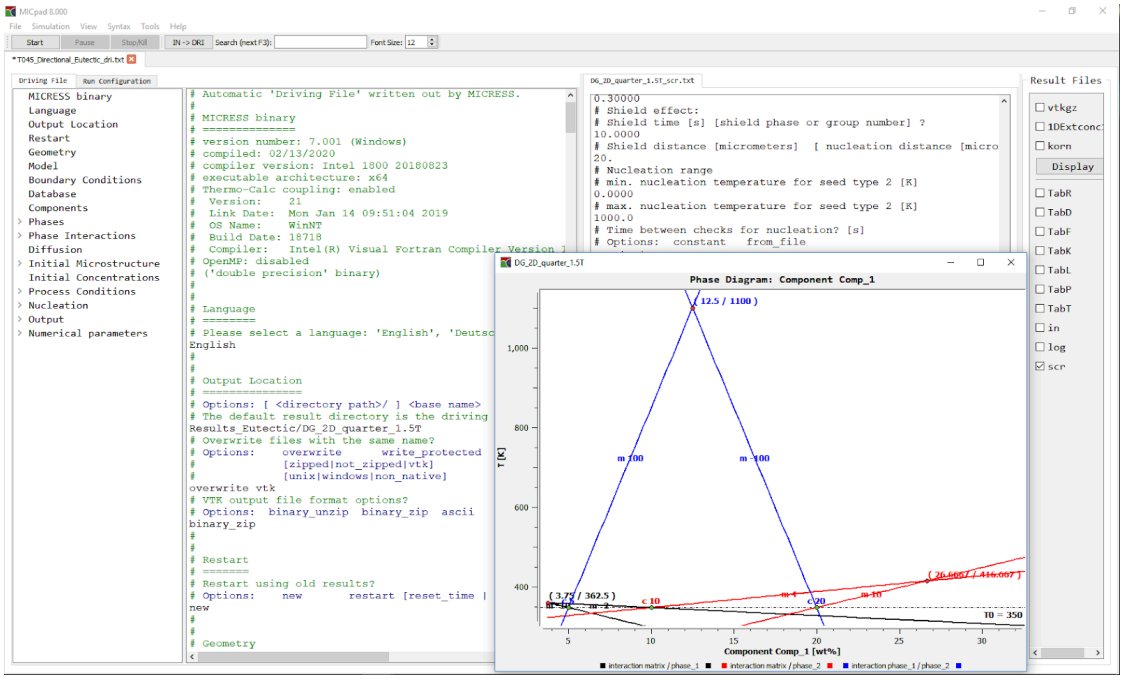


Fig 21. Phase Diagram plot

3.2.3 Increment 3: **Running Simulations**

The task of this increment includes all the functionalities related to running simulation. These tasks include starting the simulation, updating the results navigation and updating the opened results and plots. These tasks require modifications in the model and in the controller part.

These tasks are carried out in the old *MICpad* differently. We have a class $JobList$ which keeps the track of all the jobs there. Each job contains an id, which is saved in each $EditorTab$ class. The result is then updated with $slotUpdateOutput$ with the help of a timer $QTimer$ as a simulation runs.

We change the updating process in the redesigned MICpad. Since we already have the controller for each $DrivingFileEditor$, we do not need to keep track of jobs with class $JobList$. Instead, we create a $ProcessJob$ in the model to handle all the operations related to simulation like starting, stopping and suspending the simulation. $ProcessJob$ has also a timer which gives a signal for updating the results. This signal is connected to $syncResult$ of $DrivingFileOutputController$. When the simulation finishes or stops, the $ProcessJob$ emits another signal $TaskCompleted$. This way, the model notifies the controller with the signals. The controller then connects to these signals and performs the respective action preserving the loose coupling principle.

The results of these implementations can be seen in ***Fig 22***, ***Fig 23***, ***Fig 24*** and ***Fig 25***. The progress after this implementation is around 80%. In the next increment, we implement all the remaining tasks which are left.

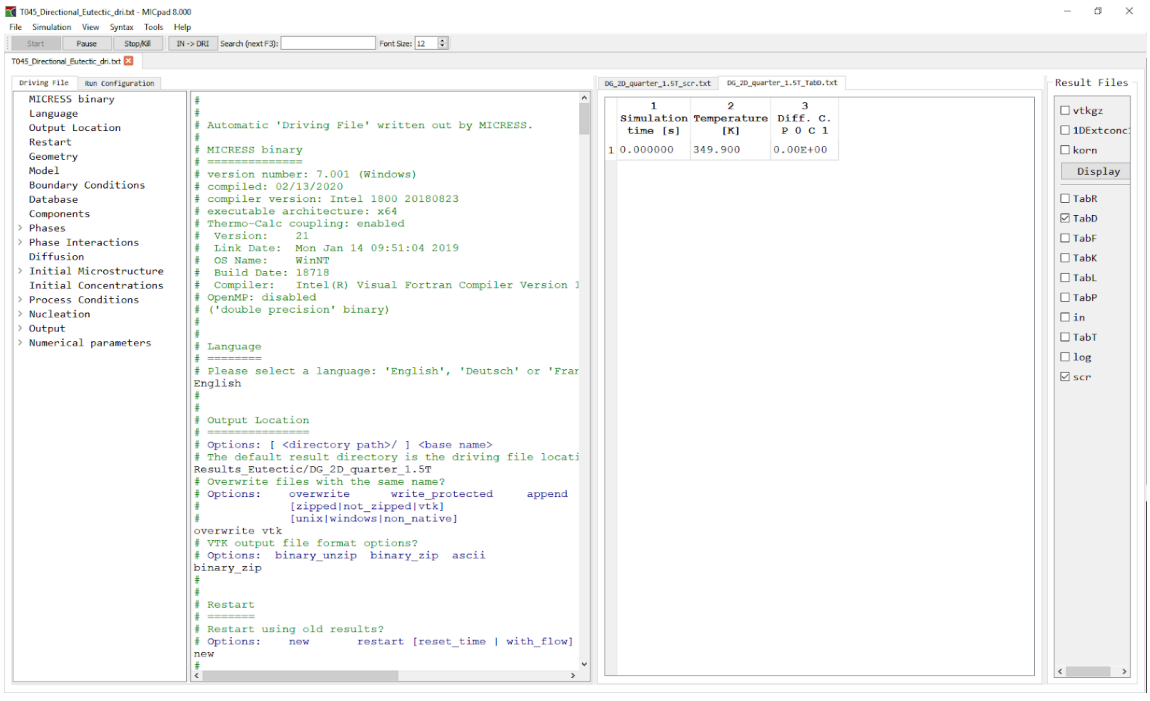


Fig 22. Running simulation updating result 1

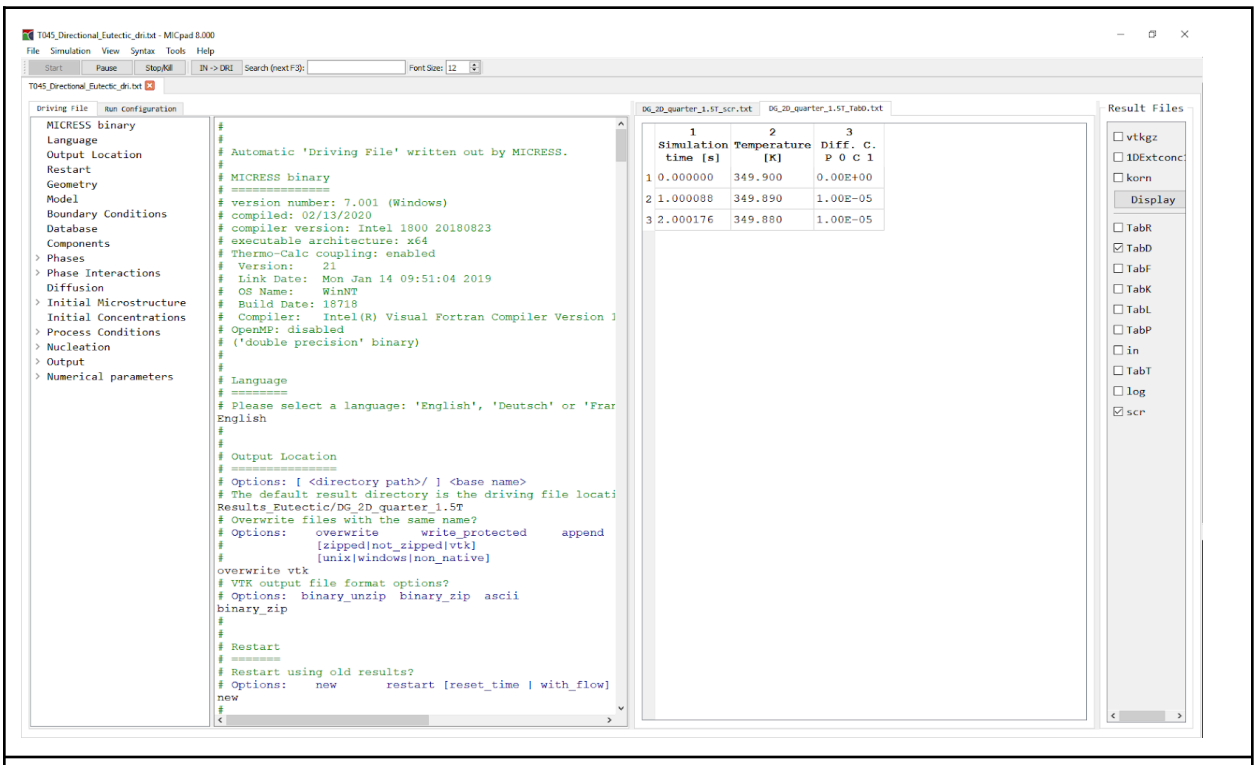


Fig 23 Running simulation updating result 2

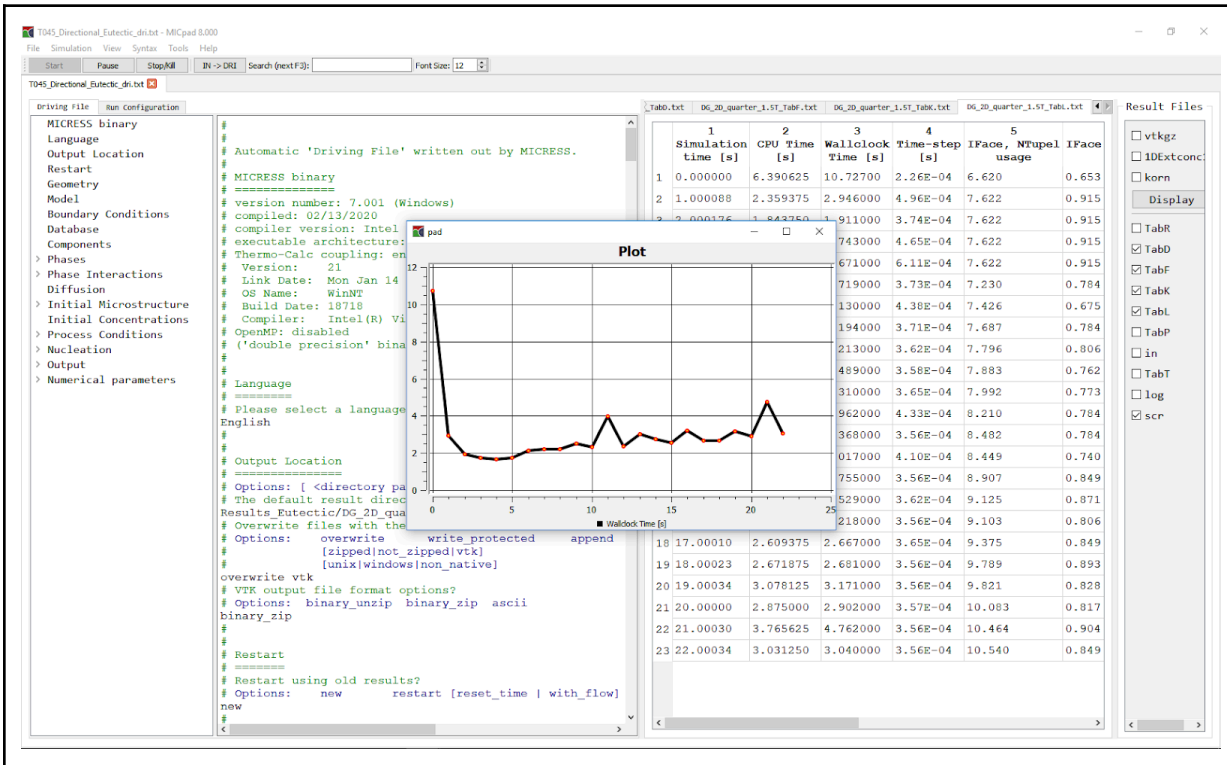


Fig 24. Running simulation updating result plot 1

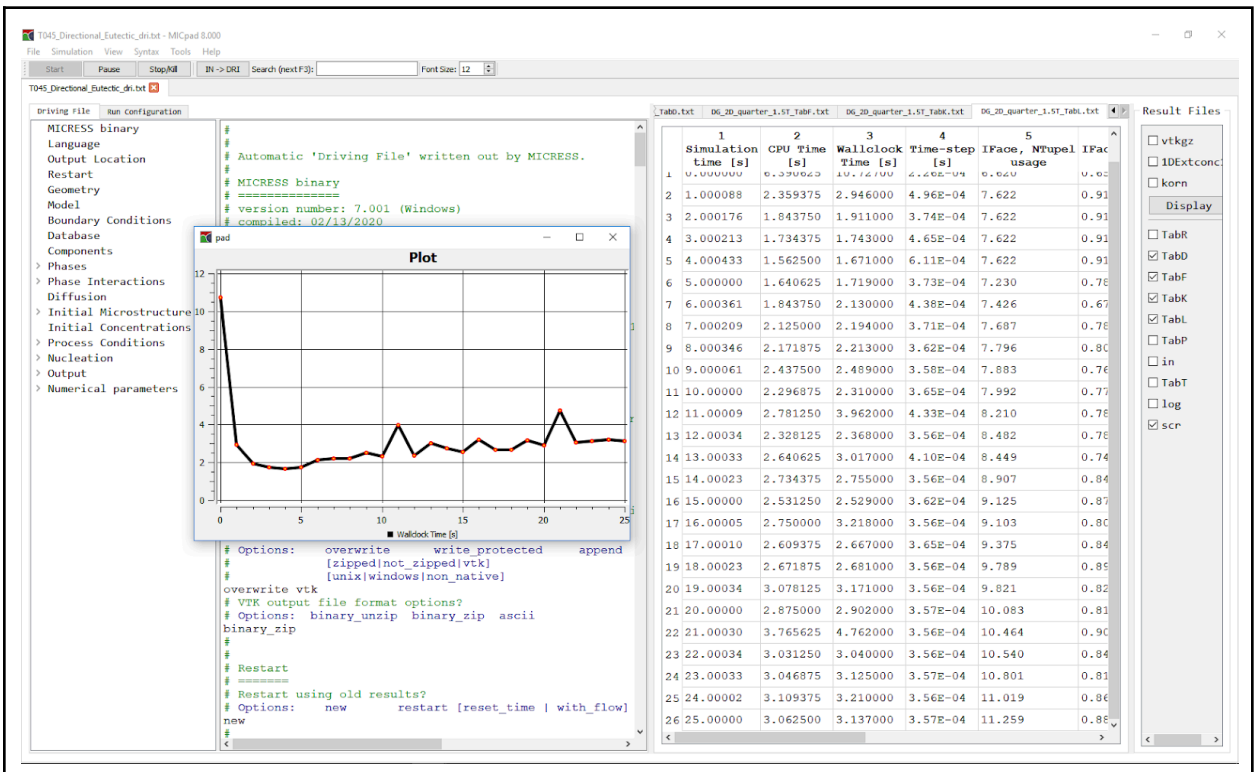


Fig 25. Running simulation updating result plot 2

3.3.4 Increment 4: **MICRESS Tools and remaining use cases**

At last, we separate the rest of the code into model, view and controller in this increment. The tasks of these increments can be grouped as MICRESS functionality tools like syntax checking, syntax conversion and copying result IN file to Input DRI file and other GUI related / remaining functionality like help, documentation, color scheme selection, showing message on closing simulation tab, etc.

The results of these implementations are shown below in ***Fig 26***, ***Fig 27***, ***Fig 28*** and ***Fig 29***. The driving file is an example of version 7.0. That's why it shows a syntax error in ***Fig 27*** when syntax is checked for the version 6.4.

With the implementation of this increment, we have completed about 100% of the project.

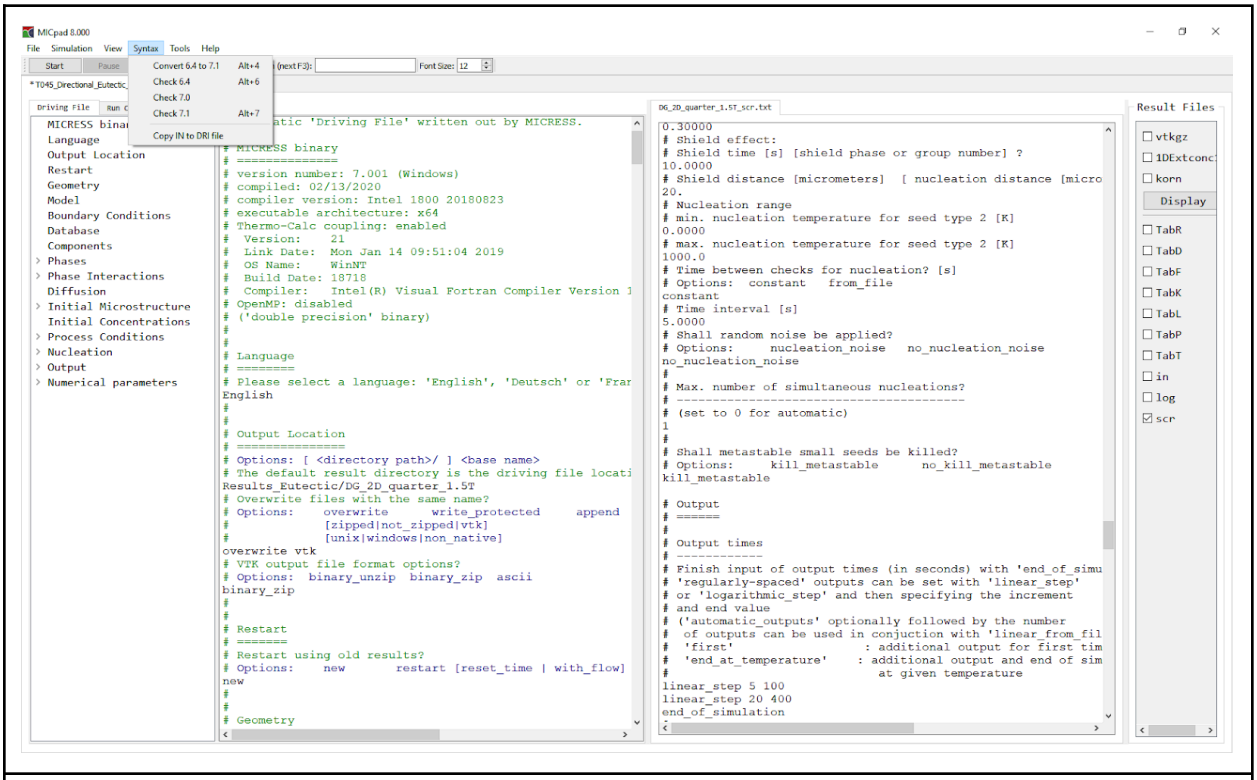


Fig 26. Syntax menus

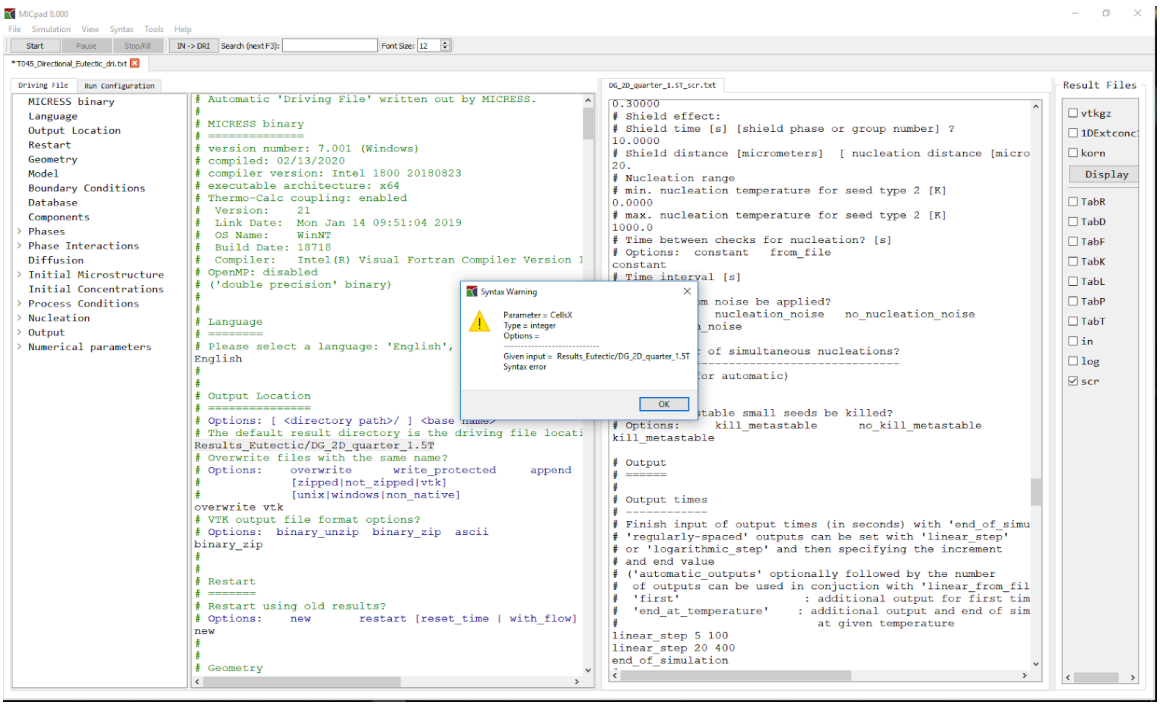


Fig 27. Syntax check for version 6.4

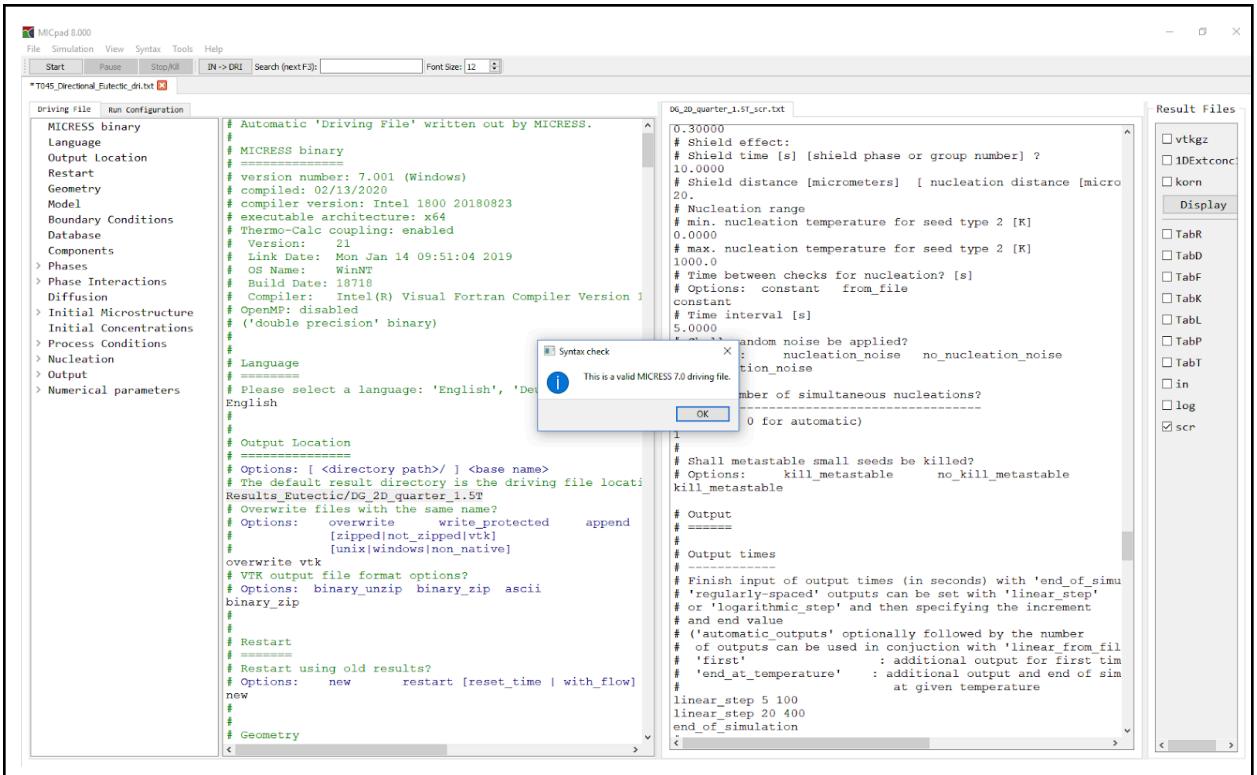


Fig 28. Syntax check for version 7.0

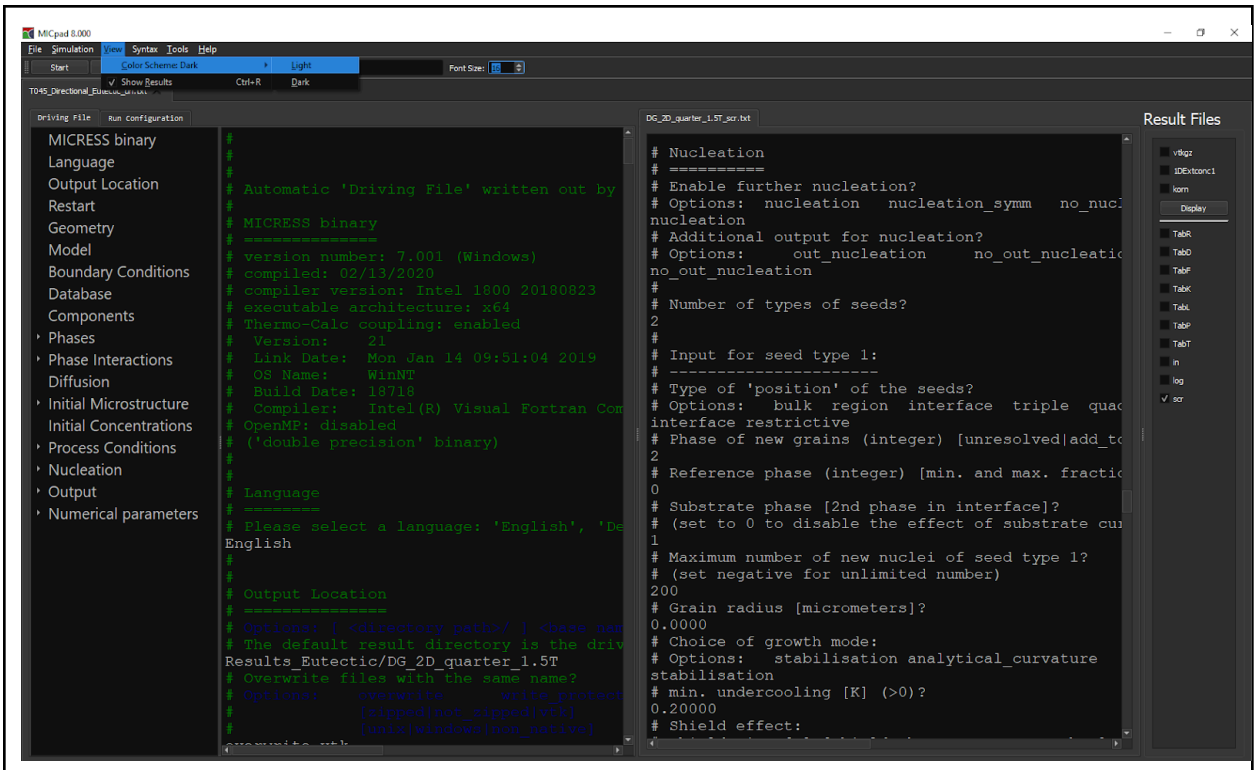


Fig 29. Color Scheme : Dark Mode

3.3 Obstacles

The first problem occurs while saving a *driving file.* The main obstacle here is that the view can not contain the information about the controller. There can be many driving files opened. When the user saves a driving file, we take the current view and try to save its content. The problem is here to get the corresponding controller from view. But according to MVC the view has no idea about its controller.

To solve this problem, we have given each view a specific id. This is done by creating an abstract class called *ContentView* (***Fig 30***) which has an attribute UUID (universally unique identifier). Every view is a child of this class. So every view class now has a unique identifier. When the controller is created with its view, these ids are stored in a lookup table as a hash table in the *MainEditorController*. Now we can get the view id from the view and get the corresponding controller from this lookup table. When the view is destroyed, the information about the view on this table is also removed.

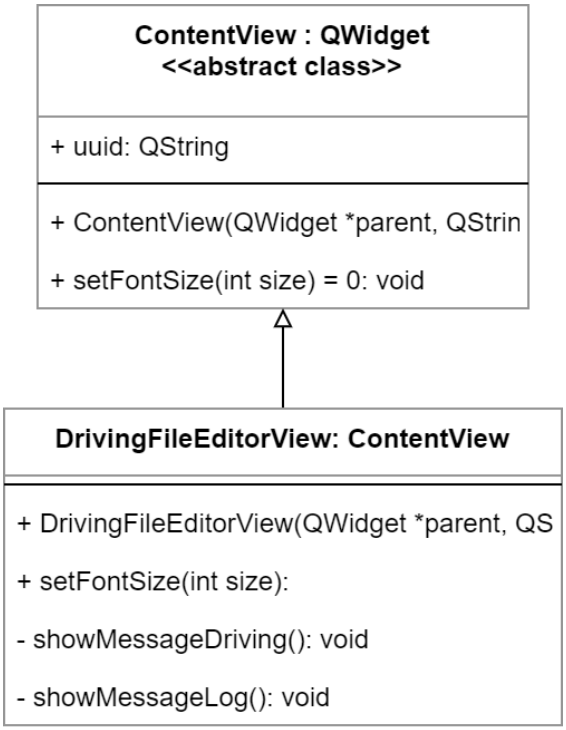


Fig 30 *ContentView*

4. Redesigned MICpad

MICpad now has a model, view and controller design. It also follows SOLID principles, which is analysed in the analysis section. The new design also follows the principle of loose coupling and high cohesion, i.e. the model is independent of the views and vice versa. They communicate through controllers through signal and slot mechanisms. For example, each view is connected to a specific controller method. So when this view component is clicked, the controller performs certain operations with the help of the model and the results/changes are then updated on views.

In this section we describe the structure and relationship among classes of each module: model, view and controller with the help of class diagrams. We also describe their purpose and the tasks of each class. We also try to divide these classes into further logical groups. However, these are just for the sake of explanation or design interpretation.

4.1 Views

There are three different layers of views. A layer is a kind of abstraction. The layer first means the main editor we see. The second layer would then be the view that pops up when we interact with the first view. The third layer would in the same way the view that pops out from the second layer and so on.

We can also interpret the above layer's interpretations as the parent-child class relation as shown in the class diagram ***Fig 31***. In the first layer, we see the *MainEditorView* and *EditorTabWidget*. They build the main layout. The *MainEditorView* provides a text editor tool with syntax highlighting for the *MICRESS* input file called DrivingFile (required to run the simulation). It also provides some operations in *MenuBar* for syntax conversion between different kinds of input files. There is a navigation area for different sections in *DrivingFile* (since the driving file generally has many lines (>500)). And there is a result display section in *MainEditorView* where a specific result file is opened from a list of result files.

The second layer is all the view classes connected to the first view as seen in the class diagram ***Fig 31***. The examples are *TextHighlighter*, *CustomTextEdit*, *PhaseDiagramView*, *AnisotropyPlot*, *RunConfiguration* and *DrivingFileOutputView*. *CustomTextEdit* is the main text editor widget where users edit/write the driving file. The words are highlighted with the help of the *TextHighlighter* class. *PhaseDiagramView* and *AnisotropyPlot* classes are diagram display widgets where phase and anisotropy plots are shown. *RunConfiguration* helps to select different kinds of binaries files for running simulations. Finally the *DrivingFileOutputView* widgets display output navigation and output display area where results are displayed either as *TableView* or *TextEditView.*

And the third layer contains classes like *RealTimePlot* and *GeneralPlotData*, which are responsible for displaying the plots from results and driving file respectively. The main difference of the new layout from the old layout is that all the views can be run independently. After connecting the controller, we can make the functionality work as specified by the use case of MICpad.

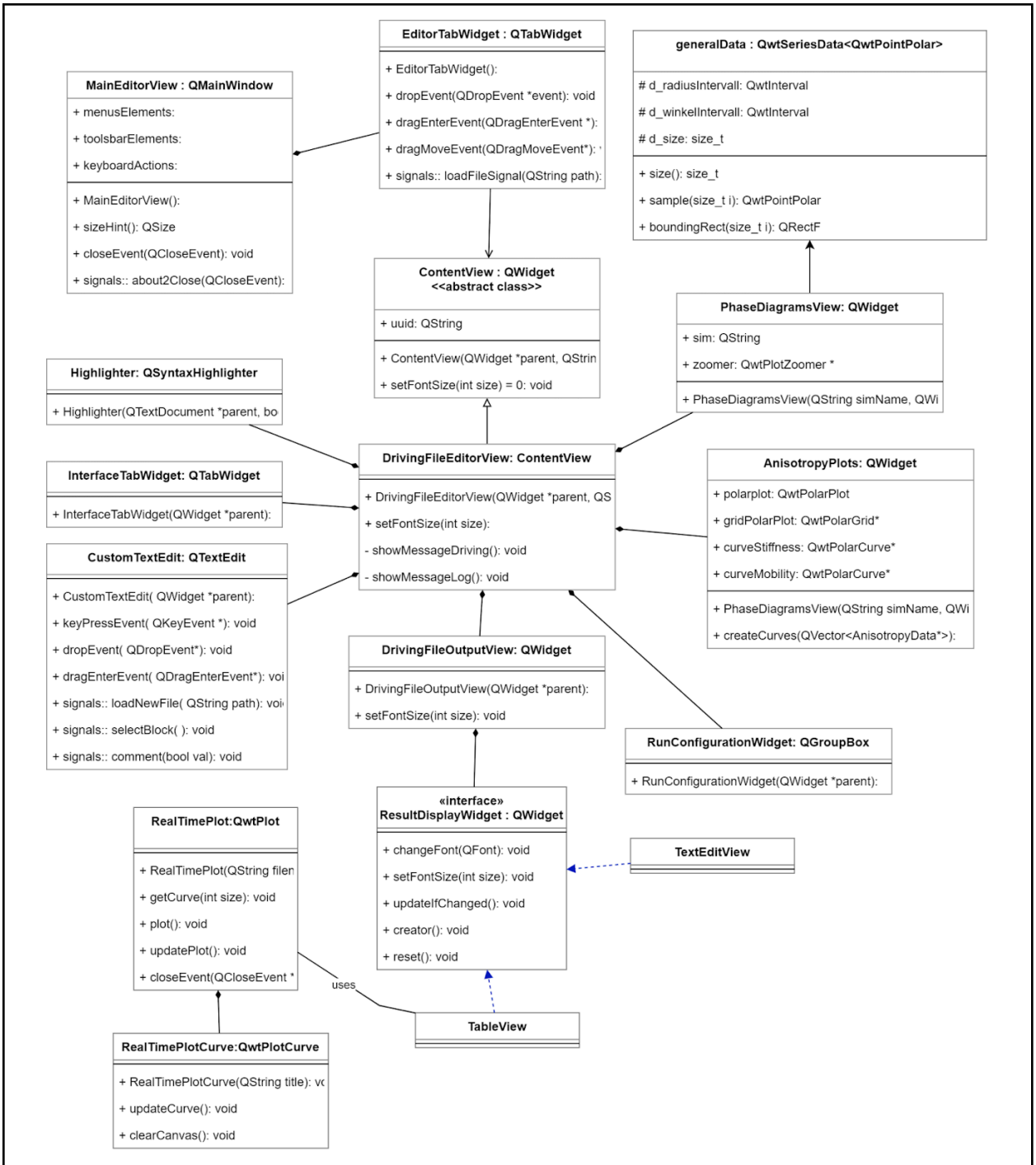


Fig 31: View class diagram

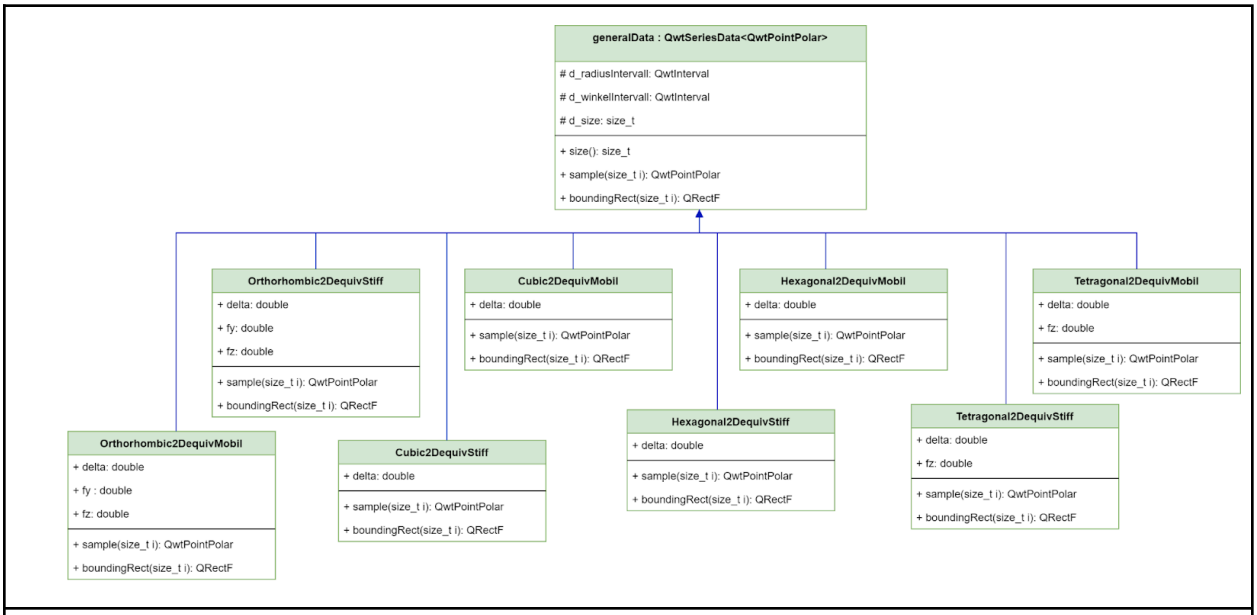


Fig 32: View class diagram for *GeneralData*

4.2 Model

The model contains the *MICpad* logic. The entire model classes can be categorized into Process model and display model. The class diagram for these modules is displayed in ***Fig 33***, ***Fig 34*** and ***Fig 35***. Process model contains the details parsing logic and running the job process, while the display logic also known as view model contains the information required to update the view element.

The main process model classes include *SyntaxAnalyser*, *DrivingFileEditorModel*, *JobList*, and *MicPadUtils*. The *SyntaxAnalyser* class helps to analyse the syntax of driving files (input file for the simulations). It also provides the functionality regarding conversion of syntax from one version to another. *DrivingFileEditorModel* sets up the result directory for the simulations. It detects the file changes and sync with the file on the local drive. It also keeps track of text status for the input editor. For example, when the driving is edited or modified, the text status changes to *not saved*. If the loaded file is analysed, the text status changed to parsed. ProcessJob helps to run, suspend and kill simulations. It also sends the signal taskCompleted() when the task is completed or some error occurs.This signal is then caught by the controller and updates the corresponding view elements like disabling the pause button, etc. *MicpadUtils* class contains static methods such as reading file, writing to file, searching in a string with regex, etc.

The Display model helps to keep the view element data. The main display classes are *PhaseDiagramsModel*, *AnisotropyModel* and *TreeModel*. The class *PhaseDiagramsModel* helps to extract the phase data from the driving file. This data is used to plot diagrams. Similarly *AnisotropyModel* helps to extract the polar data and phase symmetry data which are used to

plot anisotropy diagrams. The class *TreeModel* keeps the bookmarks' information for the driving file. This is because the driving file is a very large file and there are different sections in the driving file. To easily access different sections in driving, treemodel is used as a navigation section.

Finally, the result display model parses the results of the simulation. The results of simulation are either binary or text. Most of the text files are tables. These tables are parsed and plotted as tables, which are then used to analyse the results. The result display model holds the data required for displaying the text output as text or table format. As the simulations run, these data are updated correspondingly. The main classes are *DrivingFileOutputModel*, *TableParser*, *TableModel*, and *TextEditModel*.

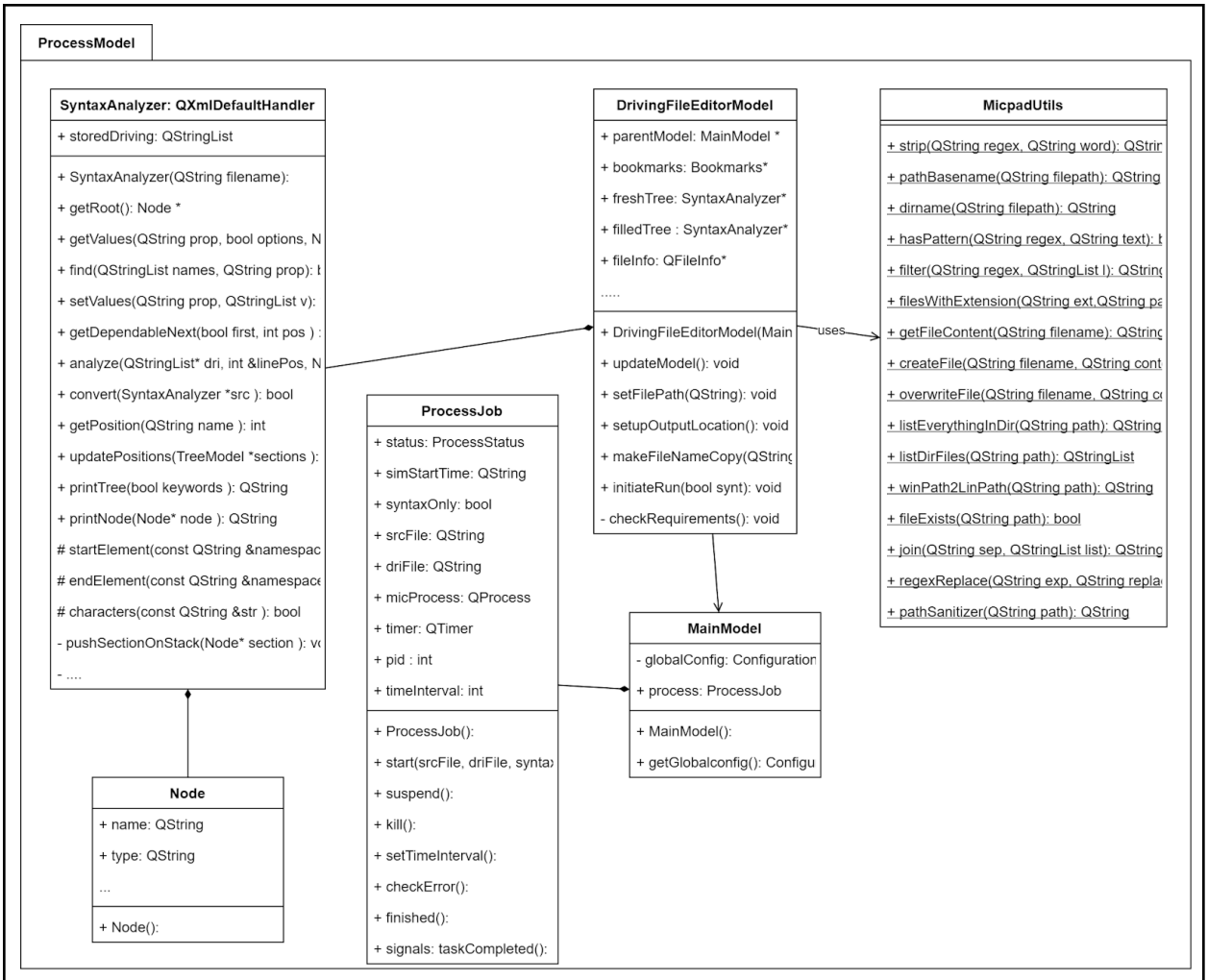


Fig 33: Process model class diagram

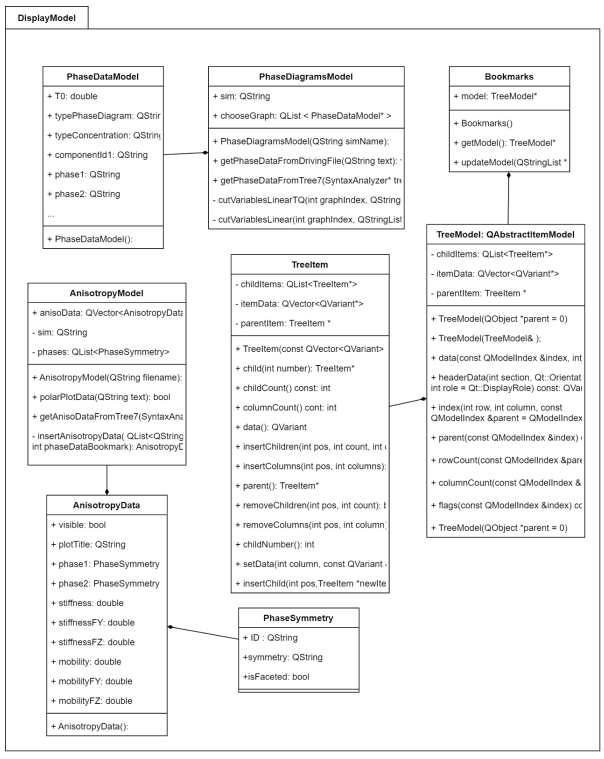


Fig 34: Display model class diagram

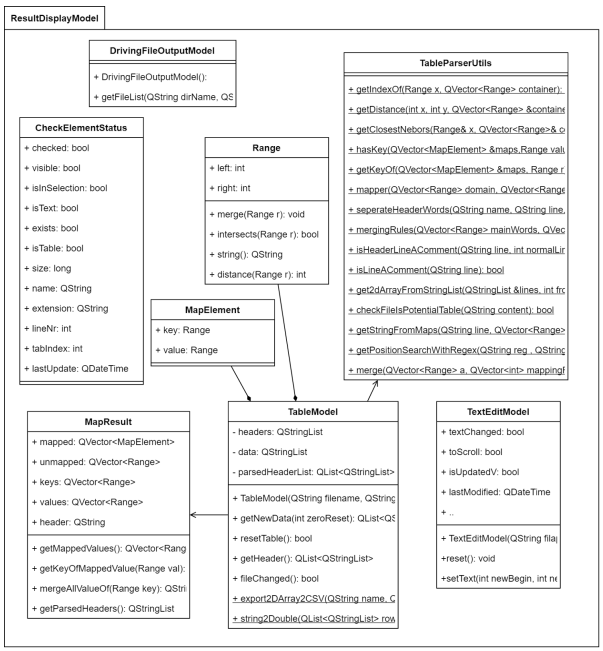


Fig 35: Result display class diagram

4.3 Controller

The controller handles the user actions and gives appropriate action. For each view there exists a controller. Every controller gets the parent controller on instantiation of the controller except for the main controller. The job of the controller is to modify views and model. It listens to the signals from model and view and takes respective action.

The main controller called MainEditorController is the first level controller. It is created when the application starts. The controller sets *MenuBar* and *ToolBar* actions to corresponding callback functions like opening new files, recent file menus, saving file, etc. It also creates an empty new editor when the GUI is loaded as a default behaviour.

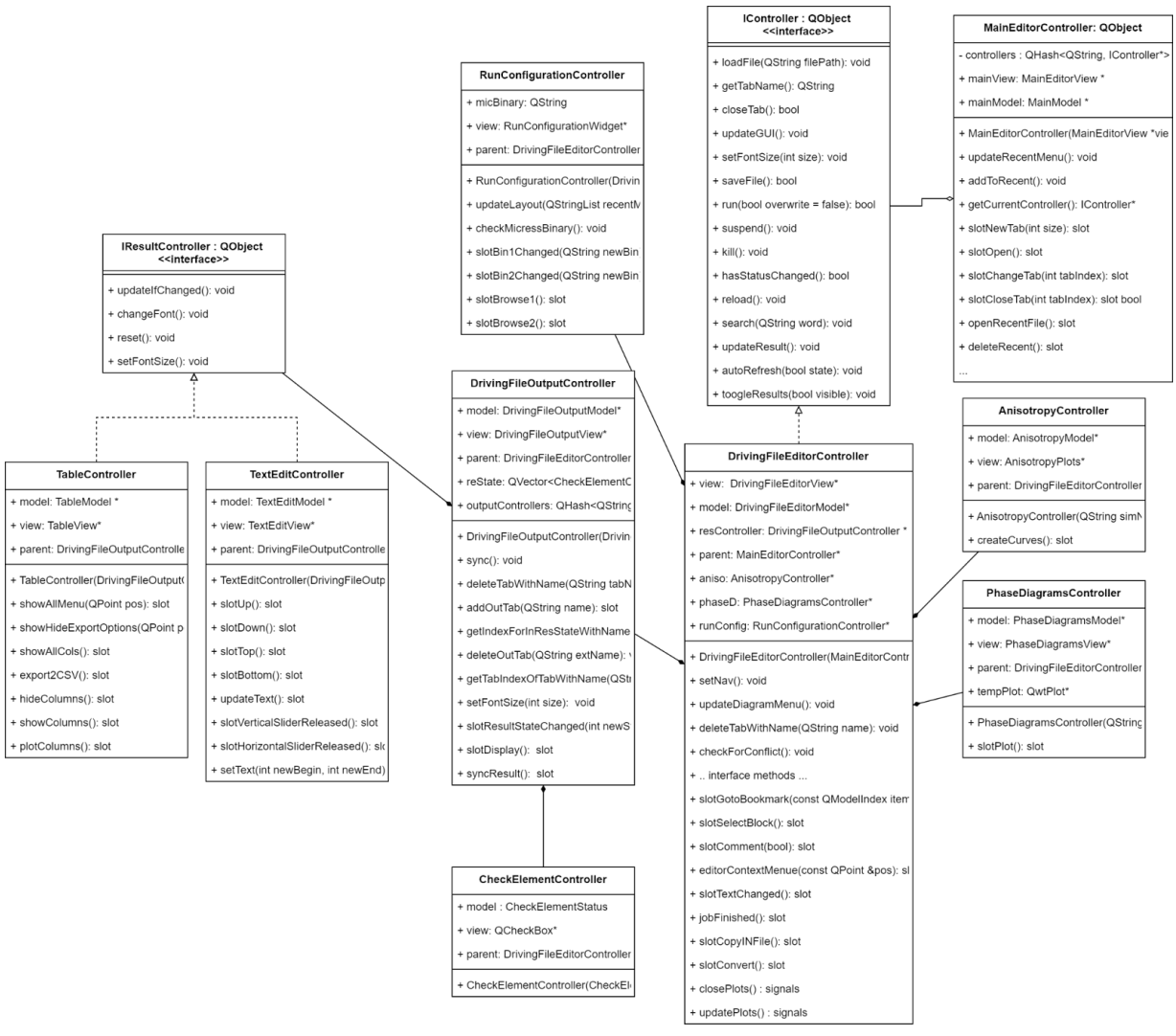


Fig 36: Controller class diagram

The next controller is *DrivingFileController* which is an instance of *IController*. It serves as an editor for driving files. It checks the syntax, converts syntax, runs simulations and displays results. The *IController* interface is created for the purpose of extending MICpad for other tools of *MICRESS* (simulation program). This follows the Open Closed Principle of SOLID principles. Now we can extend *MICpad* functionality by implementing *IController* without having to modify the existing class. These other tools of *MICRESS* are *HOMAT*, *Mesh2Homat* and *Mesh2Abaqus*. But for this thesis, we only focus on *DrivingFileEditorController*.

The driving controller *DrivingFileEditorController* has many other child controllers like *AnisotropyController*, *PhaseDiagramController*, *RunConfigurationController* and *DrivingFileOutputController*. *AnisotropyController* and *PhaseDiagramController* are responsible for making plots from the driving file. The *DrivingFileOutputController* displays and syncs the result of simulations. It also displays the results based on the type of output. If the output is table, it is displayed as table else it is displayed as text. For displaying the output as text, *TextEditController* is used and for displaying as table, *TableController* is used. These are also the children of *DrivingFileOutputController. DrivingFileOutputController* also keeps track of all child controllers i.e. *TextEditController* and *TableController.* This way the results and plots of these results can be updated for running simulation. ***Fig 36*** displays the class diagram for all the controllers and their relations.

5. Analysis and Comparison

The purpose of this implementation is to help in maintaining the project by increasing loose coupling and high cohesion. This is done by implementing SOLID principles and MVC design patterns. In this section, we go through each SOLID principle and check how much the redesigned code follows SOLID principles. We also compare the cyclomatic complexity of the redesigned MICpad with the old MICpad. It would be an independent metric to check our code quality, especially the complexity / simplicity of the code besides SOLID principles.

5.1 Verification with SOLID principles

At first, we go through each principle of SOLID and verify whether SOLID principles are followed. There may be some classes which might not be following the SOLID principles completely even after redesigning MICpad. These classes and methods will be listed in the further improvement sections where the further necessary actions would be carried out.

5.1.1 Verify SRP

SRP states a class should do only one task. But it is very difficult to determine what that one task would be. So in order to test this principle, we assume if a class would do one thing, it cannot have more than 290 LOC or 21 number of methods (see section 2.2.2 ***Table 2*** for more) on average.

Fig 37 and ***Fig 38*** show the bar graph of classes and their LOC for redesigned MICpad and old MICpad. The classes like *PhaseDiagramModel*, *AnisotropyModel* and *SyntaxAnalyser* have LOC higher than 400. Since these classes require deep knowledge of *MICRESS* and *MICRESS* driving, they are not refactored. If we skip these classes, the other class with a higher number of LOC is *DrivingFileEditorController.* This is because it performs the navigation related operations, editor context menu operations and also implements the IController. The class is the same as the Editor class from the old MICpad which contains 2133 LOC. So it has improved significantly. This can be taken as the next improving class.

Beside *DrivingFileEditorController,* all the classes are within limit. *MainEditorController* also contains 310 LOC which can also be considered within limit. So all the classes follow the SRP.

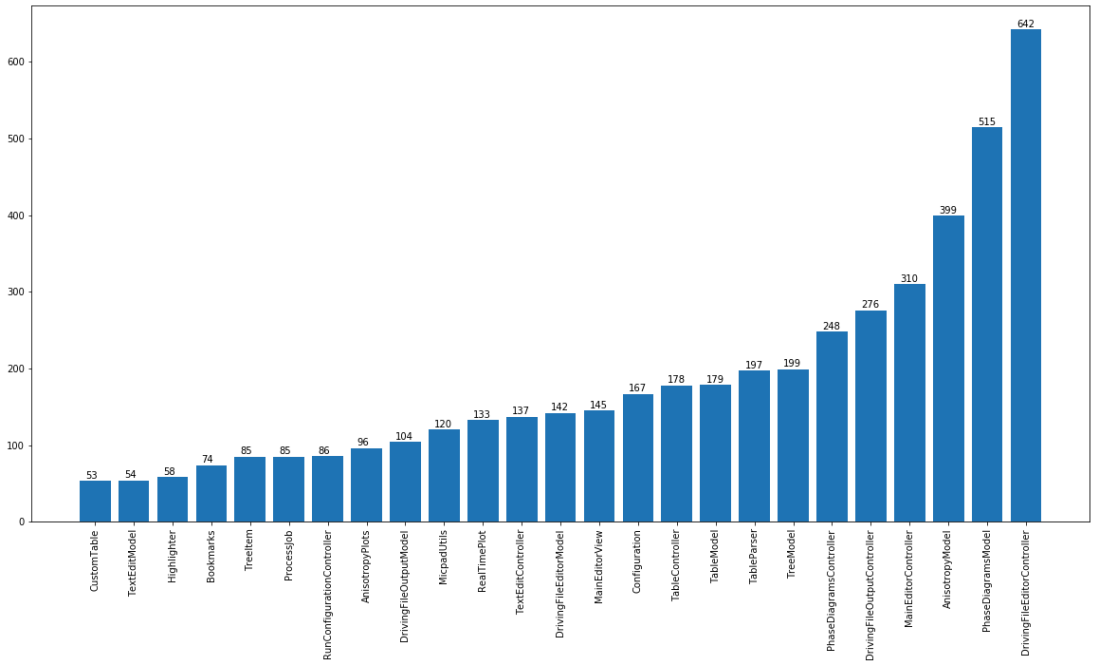


Fig 37: Lines of Code per class (redesigned MICpad)

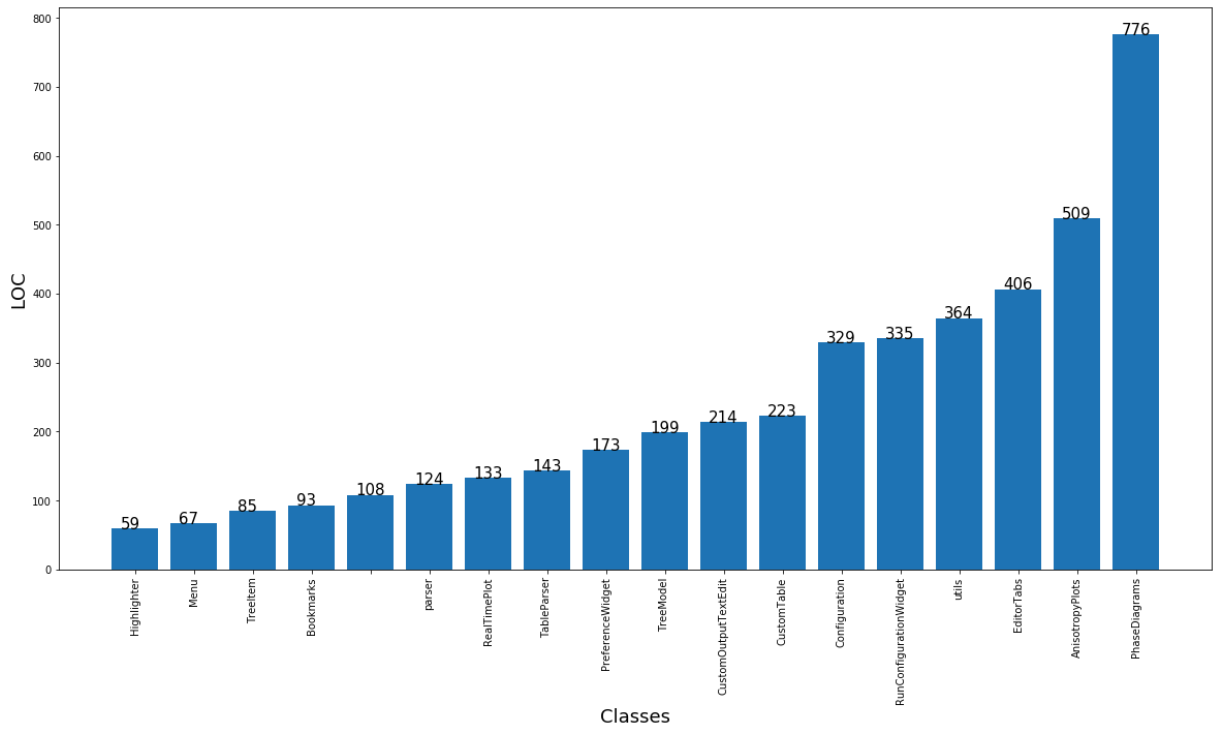


Fig 38: Lines of Code per class (old MICpad)

5.1.2 Verify OCP

This principle says that a class should be open for extension but close for modification. What this means is that we should add new functionalities without modifying existing classes. To verify this principle, we take the next features that need to be added in *MICpad* and see how the current design allows us to add new features.

The new functionality of *MICpad* to be added are adding an interface of *Mesh2Homat*, *Mesh2Abaqus* and *HOMAT*. The task is to recognize these files by its extension. The user should be able to run these input files for these binaries as well and show its corresponding results. This functionality can be extended by creating a view for each of these programs and connecting them through controllers. The main controller of this program should implement the *IController* interface (***Fig 39***) and that would enable it to add new functionality without having to modify the existing *DrivingFileEditorController*. In fact, they are quite independent of each other. Since the redesigning is done taking these in consideration, we can say extending these functionalities can be done following OCP.

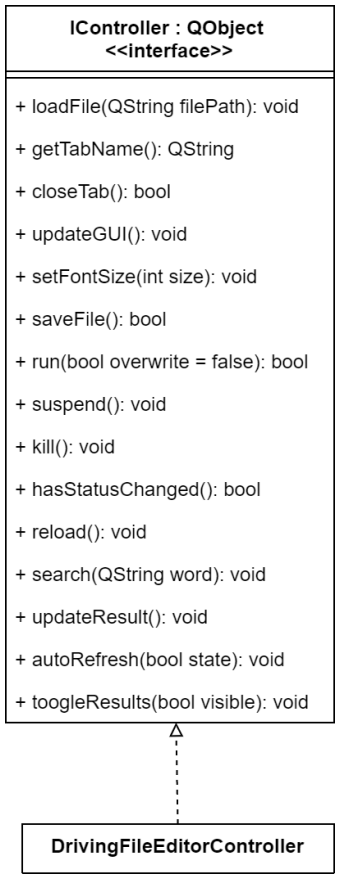


Fig 39: *IController* interface and its implementation

Now if we consider an arbitrary feature we want to add to *MICpad*, we would require its view, model and controller which can be added independently to existing modules without having to change the existing code. Thanks to the *MVC* pattern, we can easily add new features following OCP.

5.1.3 Verify LSP

According to this principle, “If S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program”. This means there should be consistency between parent and child relations. One way a child might violate this principle is by overriding the parent method and changing the parent's behaviour unexpectedly. So to verify this principle, we take a list of all classes which have parent classes and check where and how their parents’ behaviour have changed. ***Fig 40*** and ***Fig 41*** show the list of classes with parents. This list is further divided into two groups. The reason for this is *Qt libraries*. In Qt, if we want to implement a callback function (slot) in a class, it needs to implement *QObject* or a class from the *Qt library* so that the signals and slot can be implemented. So the classes which are child of *QObject* or *QWidget* class do not override any of its parent methods (***Fig 41***). So these can not possibly violate this principle. Children of *GeneralData* also do not violate this principle. This is already analysed in the Seminar thesis.

Most of the classes *DrivingFileEditorController*, *TableController*, *TextEditController*, *DrivingFileEditorView*, and so on in ***Fig 40*** do not modify parents behaviour either. So they are also fine according to this principle. The classes which modify its parent are *MainEditorView*, *EditorTabWidget, InterfaceTabWidget* and *PlotZoomer*.

Class *MainEditorView* modifies $onTabClose$ function so that the user does not close the file mistakenly without saving his works. So this does not violate this principle either. *EditorTabWidget* and *InterfaceTabWidget* only modify the drag and drop functionality so that users can open a file by dragging into MICpad. The principle is not violated here as well since the functionality is just as the expected behaviour.

```
void setZoomBase( const QRectF &base ){
    if ( !plot() )
        return;
    int currentIndex = zoomRectIndex();
    QStack<QRectF>* stack{};
    *stack = zoomStack();
    stack->replace(0, base);
    setZoomStack(*stack, 0);
    zoom(currentIndex);
}
```

Listing 2 class of PlotZoomer

The class *PlotZoomer* modifies the $zoom$ method *setZoomBase* to modify the zooming functionality for a running simulation. The code for *PlotZoomer* is shown in the ***Listing 2***. The $setZoomBase$ expects a base for zooming functionality when the escape button is pressed. For example, when the figure is zoomed and the user wants to return to the original figure before zooming, the user can press the escape button to reset the image. So the $setZoomBase$ function expects an initial position (current index) where it can return when the escape button is

pressed. This is required for a running simulation where the $zoomRectIndex$ is changing. So the overwritten function sets a zoom base as expected by the parent. So the LSP principle is followed. With this class, we conclude all the classes follow this principle.

class	parent
DrivingFileEditorController	IController
TableController	IResultController
TextEditController	IResultController
EditorTabWidget	QTabWidget
MainEditorView	QMainWindow
CustomTextEdit	QTextEdit
DrivingFileEditorView	ContentView
Highlighter	QSyntaxHighlighter
InterfaceTabWidget	QTabWidget
LogDialog	QDialog
RunConfigurationWidget	QGroupBox
RealTimePlot	QwtPlot
CustomTable	QTableView
PlotZoomer	QwtPlotZoomer
RealTimePlotCurve	QwtPlotCurve

Fig 40: class parent table group 1

class	parent
Logger	QObject
MainEditorController	QObject
AnisotropyController	QObject
PhaseDiagramsController	QObject
RunConfigurationController	QObject
CheckElementController	QObject
DrivingFileOutputController	QObject
ViewUtils	QWidget
AnisotropyPlots	QWidget
PhaseDiagramsView	QWidget
GeneralData	QwtSeriesData < QwtPointPolar>
Cubic2DequivStiff	GeneralData
Cubic2DequivMobil	GeneralData
Hexagonal2DequivStiff	GeneralData
Hexagonal2DequivMobil	GeneralData
Tetragonal2DequivStiff	GeneralData
Tetragonal2DequivMobil	GeneralData
Orthorhombic2DequivStiff	GeneralData
Orthorhombic2DequivMobil	GeneralData
DrivingFileOutputView	QWidget
TableView	QWidget
TextEditView	QWidget
ProcessJob	QObject
ContentView	QWidget

Fig 41: class parent table group 2

5.1.4 Verify ISP

This principle states a child class should not be forced to implement any method it does not require. ***Fig 39***, ***Fig 42***, ***Fig 43*** and ***Fig 44*** are all the interfaces of the redesigned *MICpad*. As we can see the *IResultController* has 4 methods (***Fig 43***) and two classes *TableController* and *TextEditController* implement these interfaces. Since the results need to be updated when the simulation is running, they both need to have *updateIfChanged* and *reset* methods. We also want the fonts to change if the font of the main editor changes, and for that we have two methods *changeFont* and *setFontSize*. So no class is forced to implement methods it does not need.

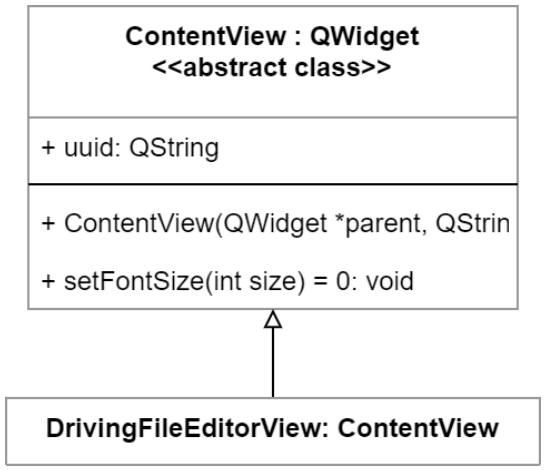


Fig 42: *ContentView* and its implementation

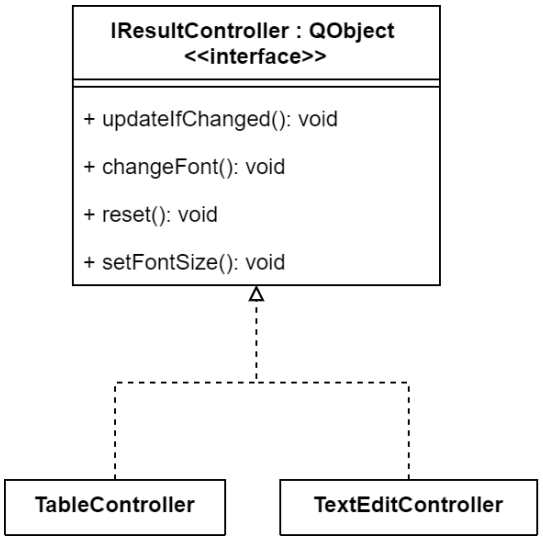


Fig 43: *IResultController* and its implementation

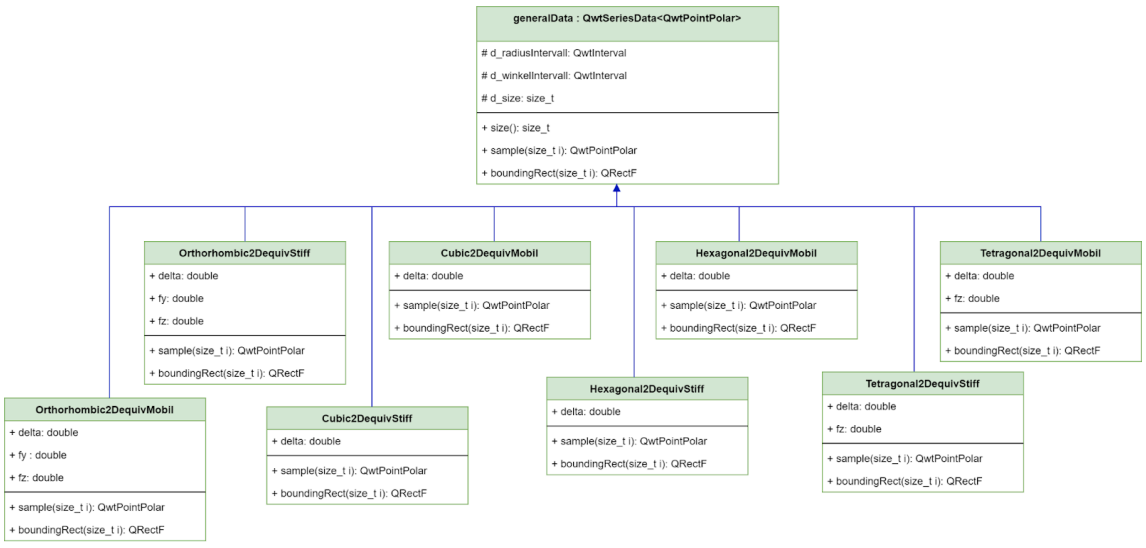


Fig 44: GeneralData and its children classes

The second interface which is *ContentView* (***Fig 42***) has only one method (the other one is a constructor since it is an abstract class). So this interface follows the ISP principle. Finally, the interface *IController* (***Fig 39***) has 15 methods which may cause it to violate this principle. However, there is only one class DrivingFileEditorController which implements this interface and all the methods of interface are used in *DrivingEditorController* class. So it does not violate this principle either. One can argue we do not need an IController interface if there is only one class that implements it. The reason for making the IController is the new extensions (*Mesh2Homat*, *Mesh2Abaqus* and *HOMAT*) which would be added in the future. Each of these features produces output and requires updating the results. So they will need the methods run, suspend, kill, updateResult, autoRefresh and toggleResults. Since they are files which can be saved and opened, they need loadFile, saveFile, hasStatusChanged and reload functions. For the GUI and tab functionality they need to implement *getTabname*, updateGUI, setFontSize, search and closeTab. So no child class would be forced to implement the methods it does not need. Thus, this principle is followed. However, we can't know in advance what methods they require and not, but for now we assume that the interface is minimal.

5.1.5 Verify DIP

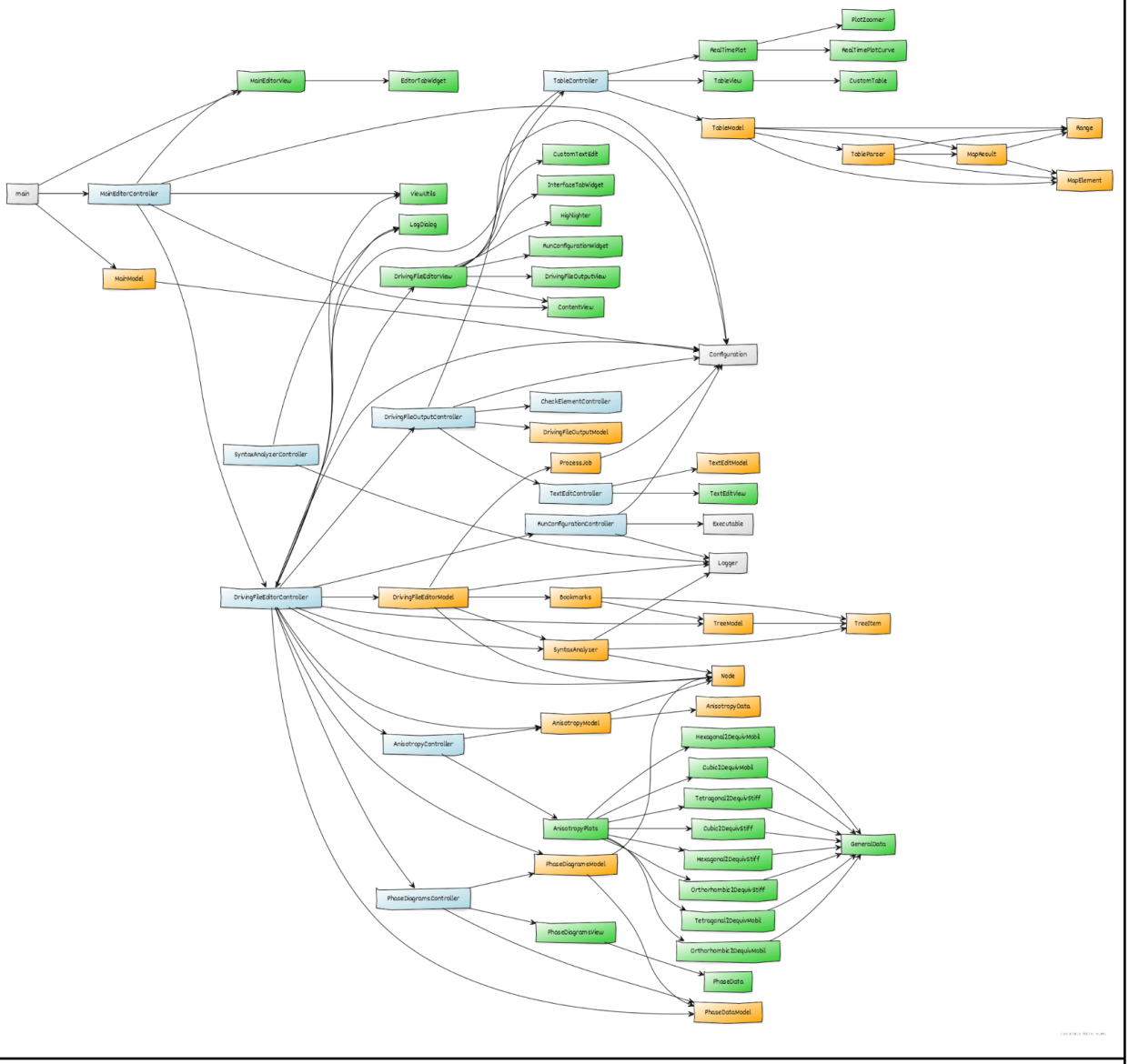


Fig 45: dependency graph of redesigned project

DIP states higher level modules should not depend on lower level modules and vice versa. They both should depend upon abstractions and finally abstractions should not depend upon details, rather details should depend upon abstractions. What this means is that a class should not depend upon another class from another module, however if they are in the same module they can use the class of the same module.

The ***Fig 45*** shows the dependency graph of MICpad classes. These nodes are classes which are encoded with color. Same colors mean the same module. The green color is the view module, orange is the model and light blue is controller module, grey color is the higher level module above MVC. We know from the seminar paper, by implementing the MVC pattern, we

implement DIP. This is because the controller acts like the interface/communication between model and view. From ***Fig 45*** we can see the model and view do not depend on each other (no connections between green color node and orange color node). The controller helps the communication between the model and view.

There are classes like *Logger*, *Configuration* and *Executable* which are higher level than MVC, but we see the controller and model classes depend upon these classes. All these dependencies are on static functions, i.e. *Logger* has static functions, *Configuration* uses a singleton pattern.

Only the *RunConfiguration* class uses the Executable class. Although *RunConfiguration* is inside the controller module, it is the same level as the *Executable* class. So we can ignore the dependency. The reason for the *RunConfiguration* class to be in the *MICRESS* controller module is because there is no implementation of *Mesh2HOMAT*, *Mesh2Abaqus* or *HOMAT* yet. Since all of these new editors also require the same configuration widget, we can say the *RunConfiguration* and *Executable* class lie at the same level. Thus, we can ignore their dependencies.

Logger and *Configuration* classes have static functions on which other classes depend. In fact, it's these static functions which cause the violation of DIP. We have two choices. We can ignore the DIP for these static functionalities, or we can remove these static functions and make them non-static. Since the logger class logs warnings or errors, it makes sense to keep it static. And also the *Configuration* class is also the same for all the editors, we can keep it the same. So we can ignore DIP for these two classes, since it makes sense. Another reason for ignoring this principle is the statement from Uncle Bob who says these principles are not law of the land. These can be bent according to the situation and conditions. But we must have a very good reason if we do bend these principles.

5.2 Comparing Cyclomatic complexity

The total complexity of the code of the old MICpad is 2277 and the total complexity for the redesigned MICpad is 1922. As we can see, there is a decrease in the complexity of the MICpad. To analyse how much the code has improved, we calculate the minimum complexity it could have. There are around 386 methods in the old MICpad and 397 methods in redesigned MICpad. If we assume 3.5 complexity per method is optimal, then the optimal total complexity of the redesigned MICpad would be $397 \times 3.5 = 1390$. If the complexity of the code would have decreased to 1390 (decreased by 887), we could call it a 100% percent decrease in complexity. But the complexity has decreased by 355, so the improvement in code complexity is about 40%.

There are however limitations to the cyclomatic complexity. Even if there is a decrease in the complexity, it does not tell anything about the individual methods. The next step would be to see how the code has improved with each method. But before we do that it is important to know that cyclomatic complexity sometimes gives false alarm. For example, although the code in ***Listing 3*** is fairly easy to understand, the complexity of the code is 11. We can agree that there are many possibilities in which the *result* variable can be true and there could be many scenarios in which it may not work as expected but the code is fairly simple to understand. So we might find some methods with fairly less number of LOC but high complexity. In this case there is a small possibility it can be a false alarm.

```
void checkError(){
    QString text = MicpadUtils::getFileContent(this->srcFile);
    bool result = (
        ( text.lastIndexOf("input error"          , -1, Qt::CaseInsensitive) != -1 ) ||
        ( text.lastIndexOf("reverting to keyboard", -1, Qt::CaseInsensitive) != -1 ) ||
        ( text.lastIndexOf("stop in routine"      , -1, Qt::CaseInsensitive) != -1 ) ||
        ( text.lastIndexOf("MESSAGE"              , -1, Qt::CaseSensitive  ) != -1 ) ||
        ( text.lastIndexOf("Unrecognised input"   , -1, Qt::CaseInsensitive) != -1 ) ||
        ( text.lastIndexOf("Press any key"        , -1, Qt::CaseInsensitive) != -1 ) ||
        ( text.lastIndexOf("no valid license"     , -1, Qt::CaseInsensitive) != -1 ) ||
        ( text.lastIndexOf("between file and keyboard input", -1, Qt::CaseInsensitive) != -1 ) ||
        ( text.lastIndexOf("Wall clock time"      , -1, Qt::CaseInsensitive) != -1 ) ||
        ( text.lastIndexOf("end homat"      , -1, Qt::CaseInsensitive) != -1 )
        );
    if(result) {
        kill();
    }
}
```

Listing 3: checkError function with 18 LOC but 11 cyclomatic complexity

Fig 46 and ***Fig 47*** show the complexity of the individual methods for complexity above 11. As we can see there are some methods whose complexity remains almost the same after the redesigning process. These can be taken as further improvements. Besides these methods, we also have methods related to classes like *PhaseDiagramModel*, *AnisotropyModel* and *SyntaxAnalyser* whose complexity has not decreased. This is because these classes require deep knowledge of MICRESS and its driving files to be able to decrease the complexity.

class	method	cyclomatic complexity nr	loc
Editor	toFinish(const QString & text)	11	18
Highlighter	highlightBlock(const QString & text)	11	25
utils	plot(QList> ysArr , QList x , PlotInfo info , QStringList ysLabel , bool skipFirst)	11	57
Editor	slotCloseTab(int akt)	12	72
Editor	loadFile(const QString & fileName , bool sameTab)	12	54
EditorTabs	~EditorTabs()	12	39
EditorTabs	setupOutputLocation()	13	55
	main(int argc , char * argv [ ])	16	108
Configuration	write(bool sync)	17	91
CustomOutputTextEdit	setText(int newBegin , int newEnd)	17	52
Editor	saveFile(QString & fileName)	17	72
Editor	slotConvert()	17	114
EditorTabs	analyzeText()	18	71
PreferenceWidget	getLayout(SyntaxAnalyzer * a)	20	82
Bookmarks	updateModel(QStringList * driText , QString syntaxType)	21	81
Editor	getValidParameterCount(QString * input)	26	46
Editor	slotUpdateOutput()	30	99
Editor	updateDiagramMenu()	30	116
Editor	getFileList(const QString & dirName , const QString & baseName)	32	116
Editor	slotRun(bool overwrite , bool syntaxOnly)	33	162
Configuration	read()	44	108
Editor	buildResultSelection()	85	216

Fig 46: Cyclomatic complexity per method (old MICpad)

class	method	cyclomatic complexity nr	loc
Highlighter	highlightBlock(const QString & text)	11	25
ProcessJob	checkError()	11	18
DrivingFileEditorController	run(bool overwrite)	12	34
TextEditModel	setText(int newBegin , int newEnd)	14	32
Bookmarks	updateModel(QStringList * driText)	16	62
DrivingFileOutputModel	getFileList(QString dirName , QString baseName)	27	102
DrivingFileEditorController	updateDiagramMenu()	28	111
DrivingFileOutputController	sync()	38	139

Fig 47: Cyclomatic complexity per method (redesigned MICpad)

5.3 Improvement possibilities

As we know, the purpose of this redesign process is to make the code more maintainable. By applying MVC design pattern and SOLID principles, we can add new functionality with ease. However, it may be still difficult to modify the existing code. This is because the MVC design pattern and SOLID principles have wrapped the complex code in some boundaries so it does not affect adding new functionality but the complexity of these functions are not significantly improved. As we can see from ***Fig 47*** there are around 7 methods with still a higher number of complexity. So these can be taken as the next steps to decrease the complexity.

Another step towards more maintainable code can be reducing the LOC in classes like *DrivingFileEditorController*, *PhaseDiagramController*, and *AnisotropyController*. This can be seen in ***Fig 37***. They have a fairly higher number of LOC than other classes. There are many functions related to input controller like context menu operations, navigation operations, etc in class *DrivingFileEditorController.* These can be further separated into individual controller. Similarly, the *PhaseDiagramController*, and *AnisotropyController* performs the model related tasks in the controller. These can be separated further.

There are some static functions in classes like *Configuration*, *ViewUtils* and *Logger* class which can be removed. This would help in the unit testing of individual classes. The *MicpadUtils* class also contains static functions, but it deals with string manipulation functions like regex search, list dir, *filter*, *filesWithExtension*, etc. So this class can be ignored since these functions have nothing to do with *MICpad* and act like some tools.

6. Summary and Conclusion

In this paper, we have implemented the analysis carried out in the seminar paper. In the seminar paper, the steps for redesigning the process were extracted by implementing a few classes. We also verified the SOLID principles partially for these implemented classes.

In this implementation, we have followed an incremental process model for our redesigning purpose. The progress is then measured during implementation with use case metrics. All the implementation processes are described in increments as described by incremental process. Finally, we perform analysis of the quality before and after implementing the redesigning process. We have verified the newly created classes with SOLID principles and suggested some more future improvements.

The study done in the seminar paper and the implementation in this bachelor thesis provide the foundation for the further improvements. The use of SOLID principles and MVC design pattern have made the code more modular. The classes are now more cohesive because of the Single Responsibility Principle. The relationships between the classes have improved because of Liskov's Substitution Principle. The use of interfaces is promoted to make the code loosely coupled, as dictated by Open Closed Principle and Interface Segregation Principle. The communication between modules is now well maintained with the help of the Dependency Inversion Principle. MVC has given Separation of Concern (SoC) in our code as the code is divided into model, view and controller. Even though the complexity of the code has not improved by much, the analysis has helped to detect the part of code which requires further attention in the future. With these improvements, the MICpad code would be simple to understand and easy to maintain.

List of Figures

List of code snippets

List of tables

Bibliography

[ACCESS e.V.] “Micress Introduction.” *Micress docs*, 2021, https://docs.micress.rwth-aachen.de/.

[Alder], Gaudenz. “Flowchart Maker & Online Diagram Software.” *app diagrams*, 2000, https://app.diagrams.net/. Accessed 10 08 2021.

[Chauhan], Raja B. *Analysis of the software "MICpad" based on SOLID principles and compliance with MVC design pattern*. december 2020. *docplayer*, https://docplayer.net/210637635-Analysis-of-the-software-micpad-based-on-solid-principles-and-compliance-with-mvc-design-pattern.html.

[GitLab]. “GitLab.” *GitLab*, 2014, https://about.gitlab.com/. Accessed 10 08 2021.

[Hopkins], Timothy, and Les Hatton. *Defect patterns and software metric correlations in a mature ubiquitous system*. researchgate, 2019.

[Martin], Robert C. *Clean Code*. Pearson Education, 2008.

[Pocketworks] Mobile Ltd. “yuml.” *yuml*, 2020, https://yuml.me/. Accessed 10 08 2021.

[Project Jupyter.] “ipywidgets.” *Jupyter Widgets*, 2021, https://ipywidgets.readthedocs.io/en/latest/. Accessed 10 08 2021.

[Qt]. “Qt.” *Qt*, 2021, https://www.qt.io/. Accessed 10 08 2021.

[Schroeder], Mark. *A practical guide to object-oriented metrics*. *Semantic scholar*, https://www.semanticscholar.org/paper/A-practical-guide-to-object-oriented-metrics-Schroeder/e3d66c47ee0ddb37868c51ca30840084263ee1f1.

[Wikipedia 1]. “Cyclomatic complexity.” *one graph*, https://upload.wikimedia.org/wikipedia/commons/thumb/2/2b/Control_flow_graph_of_function_with_loop_and_an_if_statement_without_loop_back.svg/375px-Control_flow_graph_of_function_with_loop_and_an_if_statement_without_loop_back.svg.png.

[Wikipedia 2]. “Component (graph theory).” *3 connected graphs*, https://upload.wikimedia.org/wikipedia/commons/thumb/8/85/Pseudoforest.svg/360px-Pseudoforest.svg.png.

[Yin], Terry. “pypi.” *lizard 1.17.9*, 2013, https://pypi.org/project/lizard/. Accessed 10 08 2021.

[Ljubovic], Vedran and H. Supic. “An iterative approach in development of the student information system: Lessons learned.” *2009 XXII International Symposium on Information, Communication and Automation Technologies* (2009): 1-6.

Additional Information

1.