1. Block 4.2: CS6514: Software Architecture Design Portfolio

Fluctuating Finite Element Analysis (FFEA) Case Study

Art Ó Liathain

September 2025

2. Table of Contents

Contents

1. Block 4.2: CS6514: Software Architecture Design Portfolio 1
2. Table of Contents 2
3. Tech Stack 3
4. Use Case Diagram 5
5. Utility Tree 6
6. SysML Diagram 7
7. Codesense Diagram 8
8. 4+1 Diagram 9
    8.1. Logical View 9
    8.2. Process View 9
    8.3. Developmental View 9
    8.4. Physical viewa 9
9. SonarQube Diagram 10
10. DV8 11
11. Understand 12
12. C4 Diagram 13
13. Dependency Graph 13
14. Class Diagram 14
15. References 14

List of Figures

Figure 1 Tech Stack 3
Figure 2 Use Case Diagrams 5
Figure 3 Utility Tree 6
Figure 4 SysML Diagram 7
Figure 5 Codesense Architecture Diagram 8
Figure 6 SonarQube Complexity Diagram 10
Figure 7 Understand Function Analysis 10
Figure 8 Understand Function Analysis 11
Figure 9 Understand Function Analysis 11
Figure 10 Understand Function Analysis 12
Figure 11 Understand Function Analysis 12
Figure 12 C4 Diagram 13
Figure 13 Dependency Graph 13
Figure 14 Class Diagram 14

3. Tech Stack

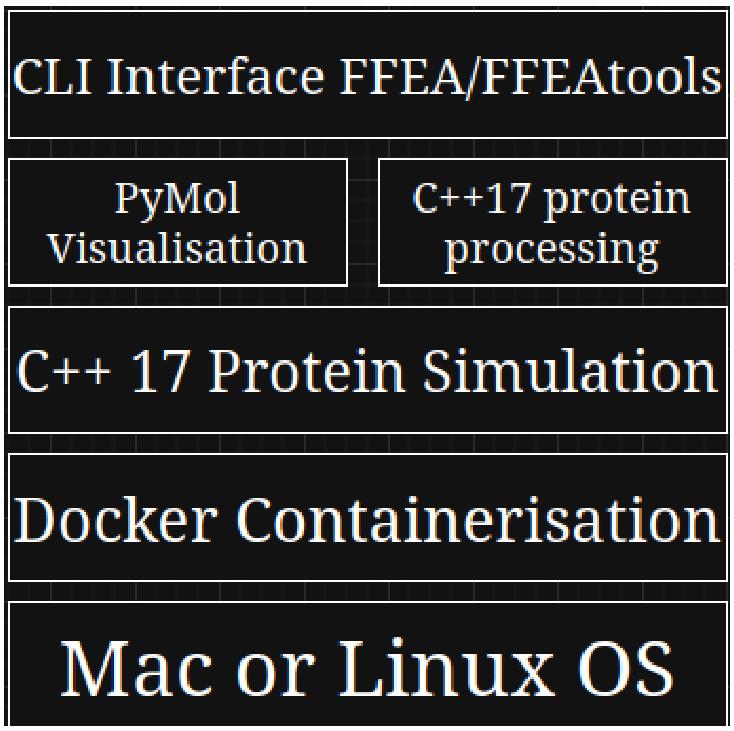
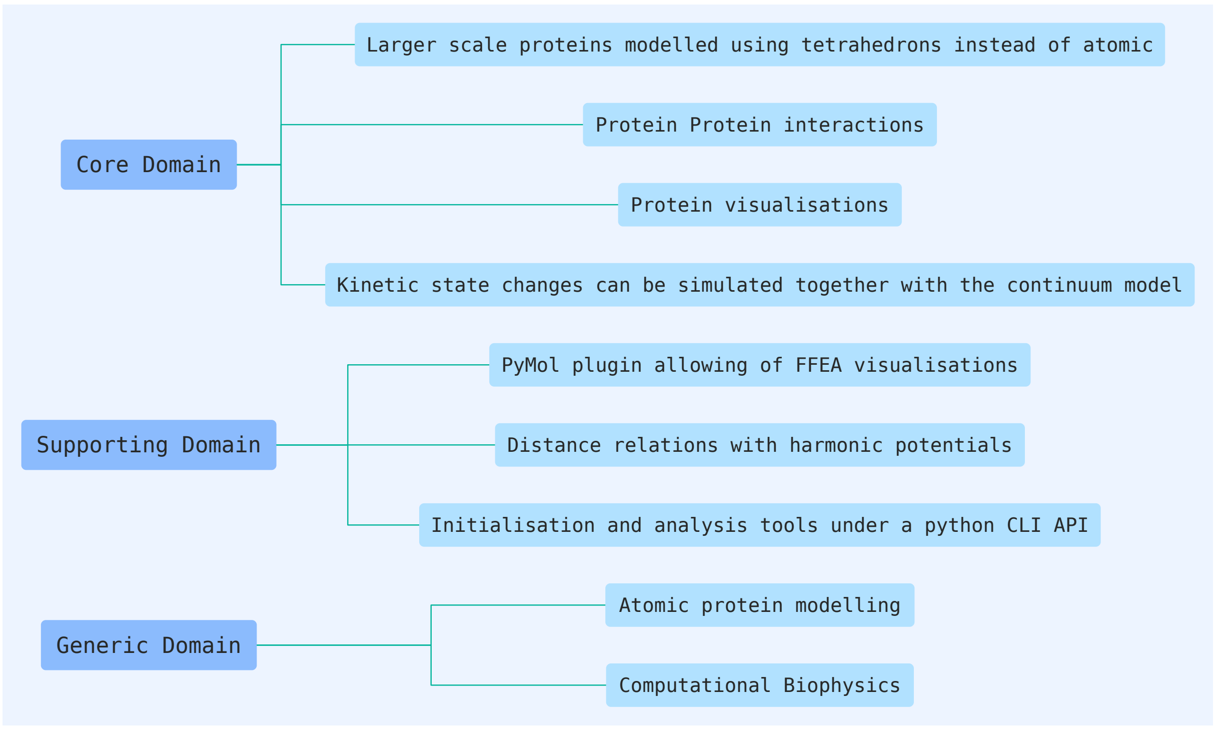


Figure 1: Tech Stack

The tech stack is primarily focused on C++, the processing and the simulations are all carried out in C++, this is a natural choice as performance is a tenant of the program. The visualisations build on the PyMol library with a plugin. This leads to a natural decoupling where visualisation is separated from processing. This is a suitable tech stack for the project allow for a modular approach and follwing the qualities required for the project.


Core Domain
Larger scale proteins modelled using tetrahedrons instead of atomic
Protein Protein interactions
Protein visualisations
Kinetic state changes can be simulated together with the continuum model
Supporting Domain
PyMol plugin allowing of FFEA visualisations
Distance relations with harmonic potentials
Initialisation and analysis tools under a python CLI API
Generic Domain
Atomic protein modelling
Computational Biophysics

Listing 1: Domain Model

4. Use Case Diagram

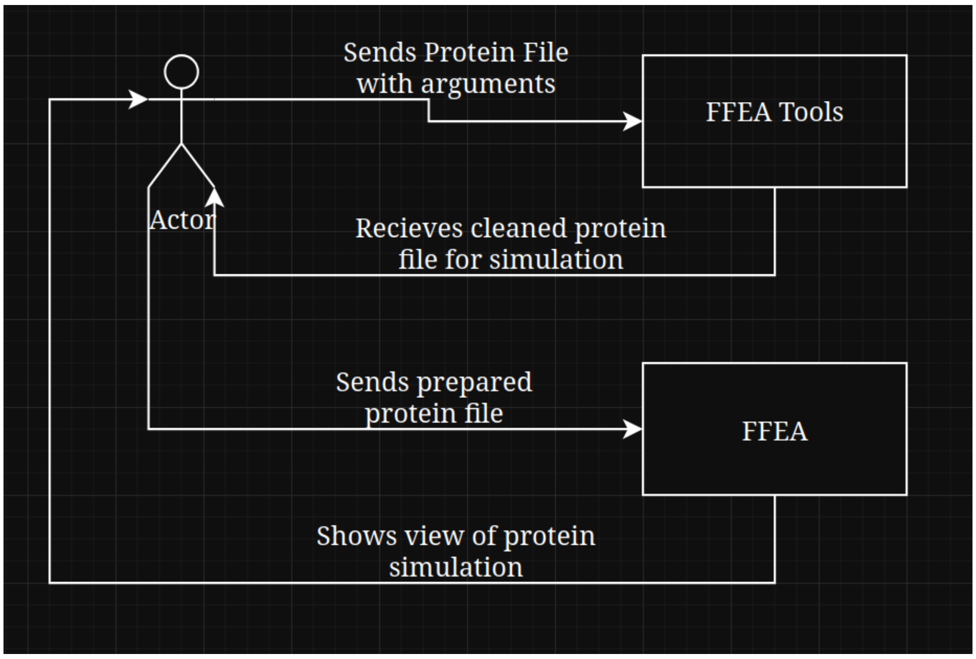

Sends Protein File with arguments
FFEA Tools
Actor
Recieves cleaned protein file for simulation
Sends prepared protein file
FFEA
Shows view of protein simulation

Figure 2: Use Case Diagrams

The primary use case for FFEA is to process protein files to then simulate under user desired conditions. The interface is rudimentary in terms of use as the complexity lies in both the simulation and parameters which means that a simple process still has layers of complexity. Looking at the use case there could be an argument to bundle the tooling as the whole purpose of ffeatools is to prepare files for ffea meaning a single unified process that processes then displays based on cli arguments would streamline the process.

5. Utility Tree

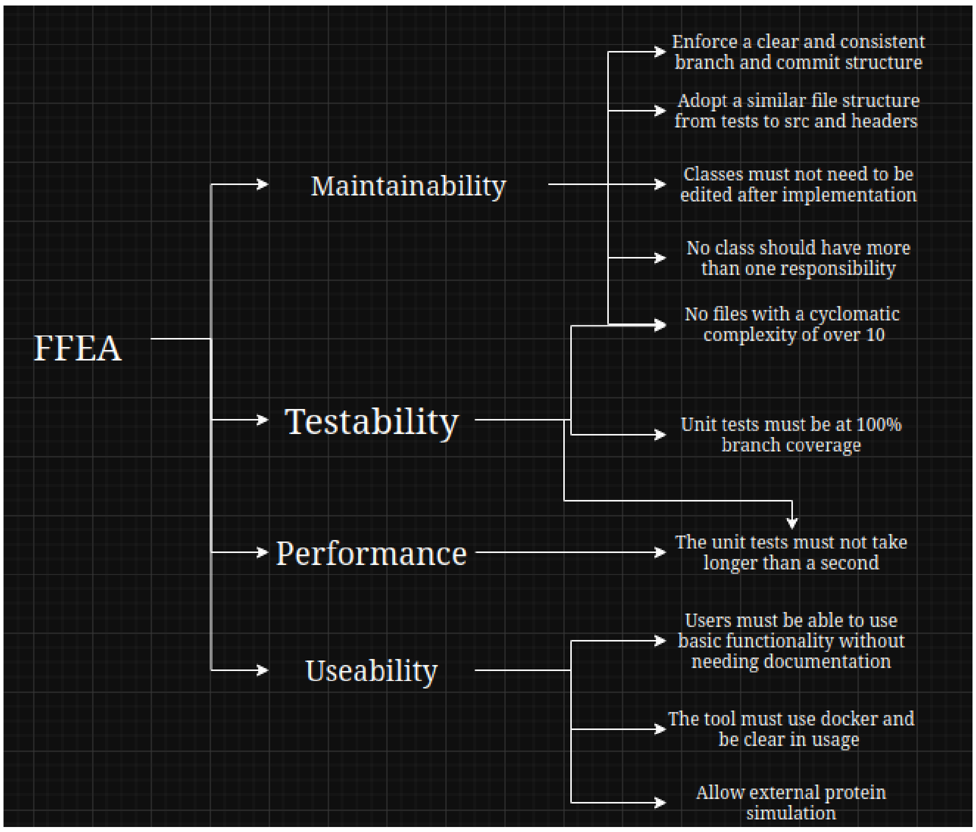

FFEA
Maintainability
Testability
Performance
Useability
Enforce a clear and consistent branch and commit structure
Adopt a similar file structure from tests to src and headers
Classes must not need to be edited after implementation
No class should have more than one responsibility
No files with a cyclomatic complexity of over 10
Unit tests must be at 100% branch coverage
The unit tests must not take longer than a second
Users must be able to use basic functionality without needing documentation
The tool must use docker and be clear in usage
Allow external protein simulation

Figure 3: Utility Tree

The utility tree is quite idealistic of aspirational long term goals of the project. There is a heavy focus on maintainability as that must be a tenant of the software for the longevity of the project and research. The maintainability goals have been selected as the foundational work needed in terms of refactoring to move the project to a useable state before any new features are added.

Testability is primarily focused on having a robust test suite to test against to allow simpler refactoring. The goal of 100% test coverage is lofty but a necessary requirement to allow for refactoring with confidence.

Performance is a tenant that is key to the project through the functional requirements but in terms of ilities it is much harder to focus non fucntional requirements through that lens as the testable performance is not easily described.

Useability is targeted towards allowing this tool to be run anywhere regardless of OS as Windows is not supported now but docker can resilve this. THe tool must allow for users to interact with it without needing the wiki to see the help commands and simple processing as well to allow for easier and more vaired user testing and feedback.

6. SysML Diagram

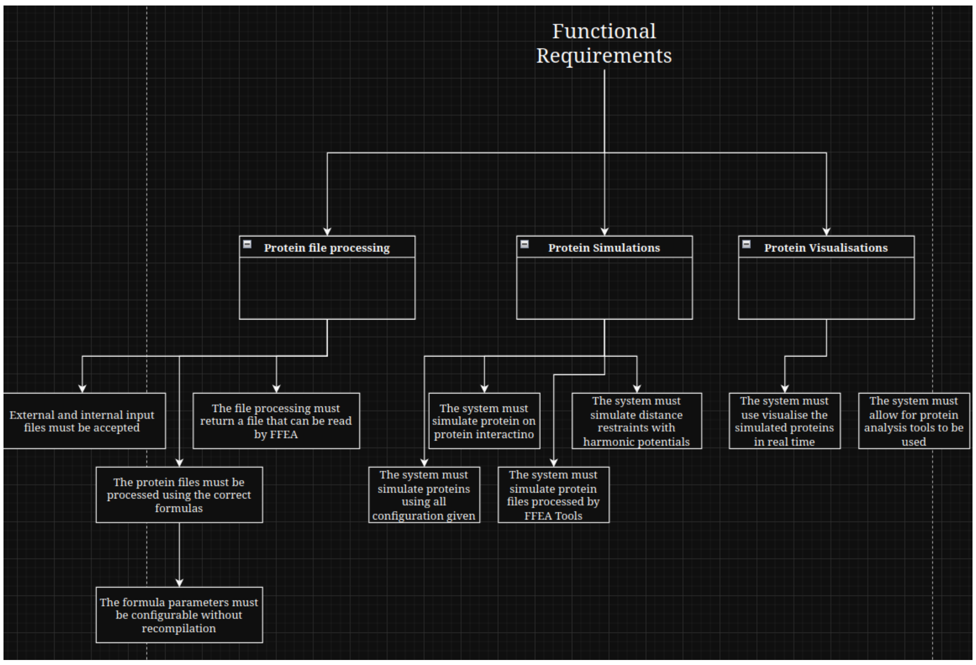

Functional Requirements
Protein file processing
Protein Simulations
Protein Visualisations
External and internal input files must be accepted
The file processing must return a file that can be read by FFEA
The protein files must be processed using the correct formulas
The formula parameters must be configurable without recompilation
The system must simulate protein on protein interactino
The system must simulate distance restraints with harmonic potentials
The system must simulate proteins using all configuration given
The system must simulate protein files processed by FFEA Tools
The system must use visualise the simulated proteins in real time
The system must allow for protein analysis tools to be used

Figure 4: SysML Diagram

The SysML diagram highlights the standout requirements of the tool to allow for accurate protein simulations to be done with respect to user input. There are undoubtedly many more functional requirements but assuming the simulations as primarily a black box the requirements centre around accurate simulations to be done and visualised based on the user configuration allowing for the correct interactions to be observed. The requirements are broken into three large logical sections being the processing, simulating and visualing. This SysML structure allows for room for growth as more requirements are discovered.

7. Codesense Diagram

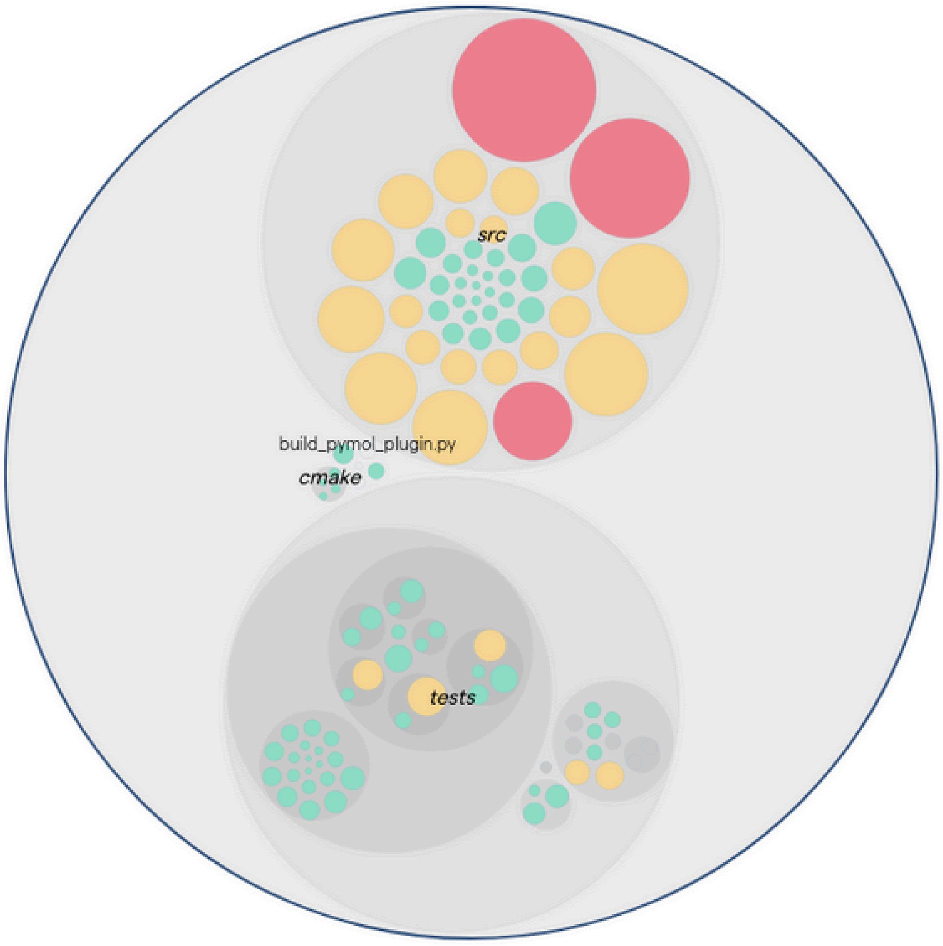


Figure 5: Codesense Architecture Diagram

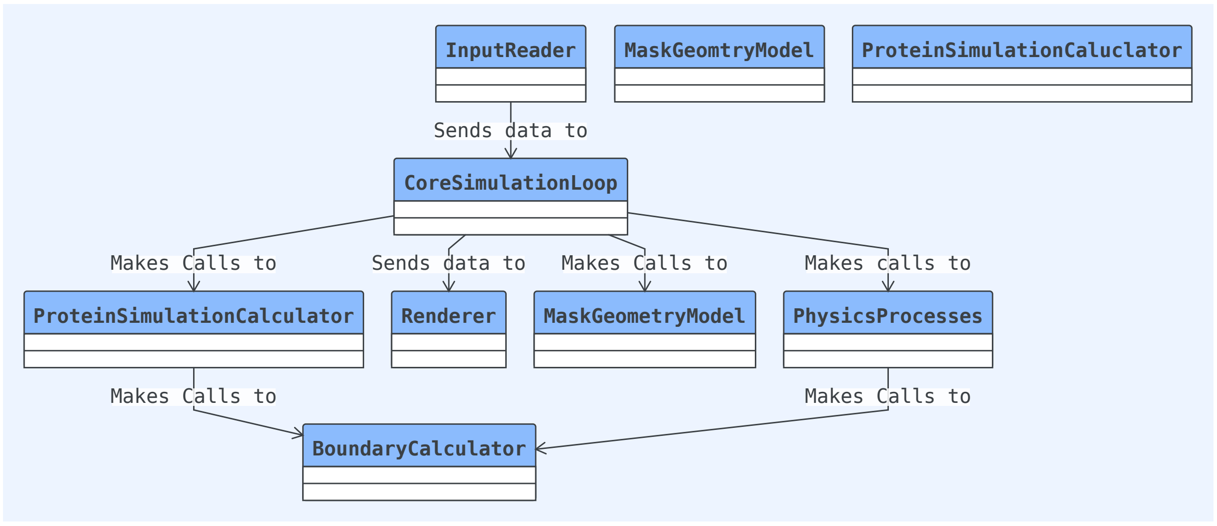
This digram highlights three key things about the architecture decisions taken for this project. The first feature being the flat structure taken for src, this means that all files are the same level and there is no logical grouping for any of the src files which makes grouping much more difficult. The is in spite of the fact that the test files are very well structured.

This leads into the second point of the severe size difference in size between tests and src, this signifies a significant amount of the code is untested making it much more difficult to add to the codebase with confidence.

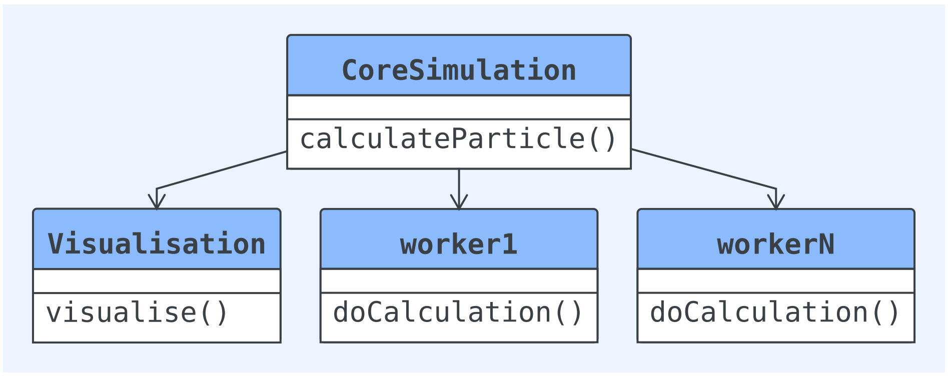
Lastly looking at the colours of the larger files three files are exceptionally difficult to modify and maintain, again a major hit to maintainability

8. 4+1 Diagram

8.1. Logical View



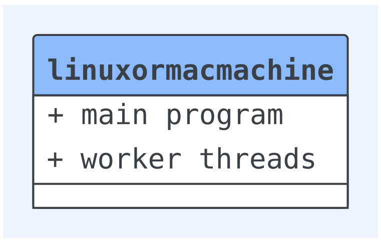
8.2. Process View



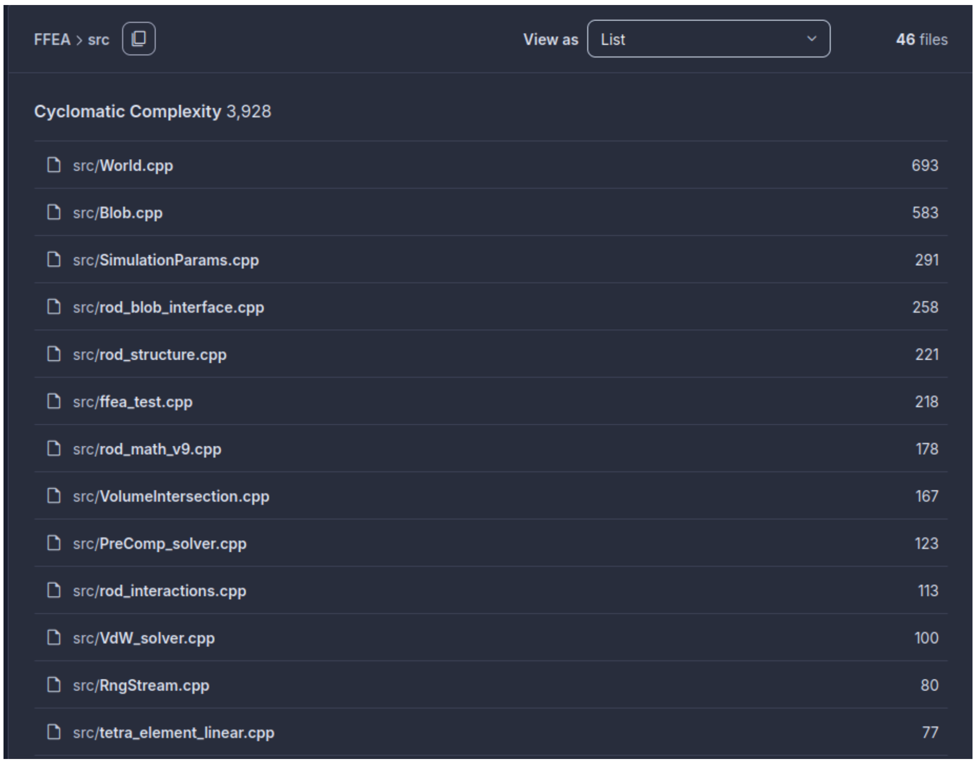
8.3. Developmental View

- The file structure is flat
- There is a high level of redundant code
- There is a god class called world
- The tests dont pass
- There are two main simulation sections being rods and blobs
- There is a logical separation between rendering and simulation due to pyton vs c++ being used

8.4. Physical viewa



9. SonarQube Diagram



FFEA › src

View as List

46 files

Cyclomatic Complexity 3,928

src/World.cpp	693
src/Blob.cpp	583
src/SimulationParams.cpp	291
src/rod_blob_interface.cpp	258
src/rod_structure.cpp	221
src/ffea_test.cpp	218
src/rod_math_v9.cpp	178
src/VolumeIntersection.cpp	167
src/PreComp_solver.cpp	123
src/rod_interactions.cpp	113
src/VdW_solver.cpp	100
src/RngStream.cpp	80
src/tetra_element_linear.cpp	77

Figure 6: SonarQube Complexity Diagram

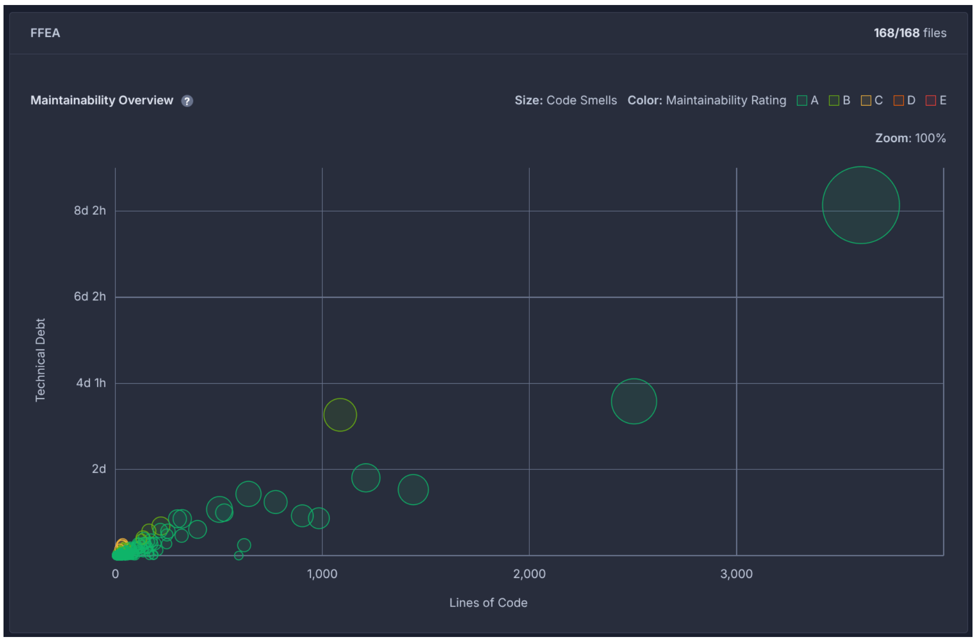
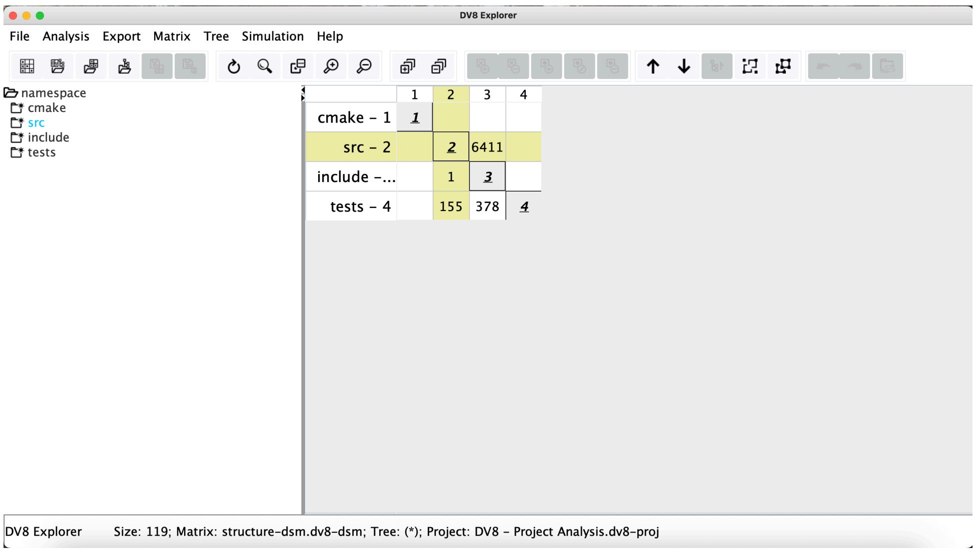


Figure 7: Understand Function Analysis

This is the cyclomatic complexity of the offending files and others from the codesense analysis.

10. DV8

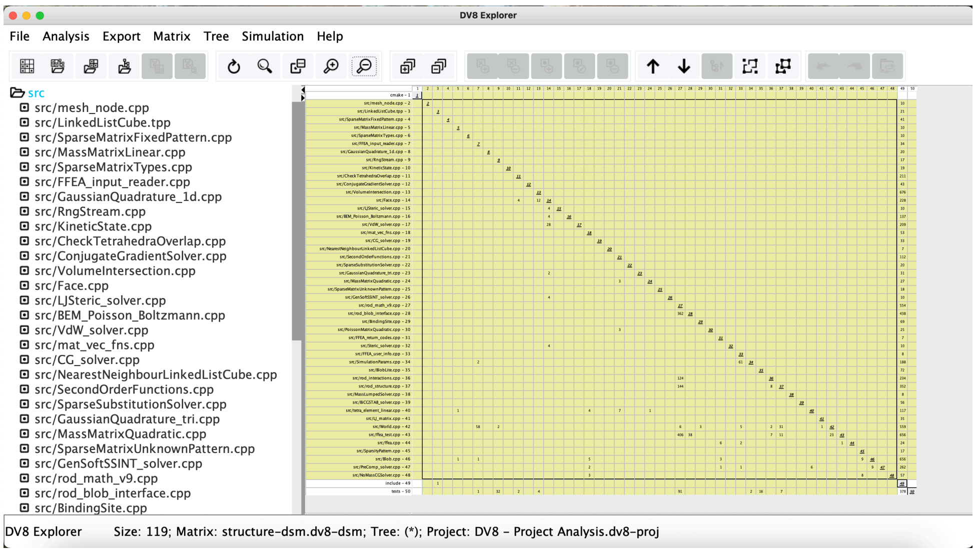


namespace
cmake
src
include
tests

	1	2	3	4
cmake – 1	1			
src – 2		2	6411	
include –...		1	3	
tests – 4		155	378	4

DV8 Explorer Size: 119; Matrix: structure-dsm.dv8-dsm; Tree: (*); Project: DV8 – Project Analysis.dv8-proj

Figure 8: Understand Function Analysis



src
src/mesh_node.cpp
src/LinkedListCube.tpp
src/SparseMatrixFixedPattern.cpp
src/MassMatrixLinear.cpp
src/SparseMatrixTypes.cpp
src/FFEA_input_reader.cpp
src/GaussianQuadrature_1d.cpp
src/RngStream.cpp
src/KineticState.cpp
src/CheckTetrahedraOverlap.cpp
src/ConjugateGradientSolver.cpp
src/VolumeIntersection.cpp
src/Face.cpp
src/LJSteric_solver.cpp
src/BEM_Poisson_Boltzmann.cpp
src/VdW_solver.cpp
src/mat_vec_fns.cpp
src/CG_solver.cpp
src/NearestNeighbourLinkedListCube.cpp
src/SecondOrderFunctions.cpp
src/SparseSubstitutionSolver.cpp
src/GaussianQuadrature_tri.cpp
src/MassMatrixQuadratic.cpp
src/SparseMatrixUnknownPattern.cpp
src/GenSoftSSINT_solver.cpp
src/rod_math_v9.cpp
src/rod_blob_interface.cpp
src/BindingSite.cpp

DV8 Explorer Size: 119; Matrix: structure-dsm.dv8-dsm; Tree: (*); Project: DV8 – Project Analysis.dv8-proj

Figure 9: Understand Function Analysis

11. Understand

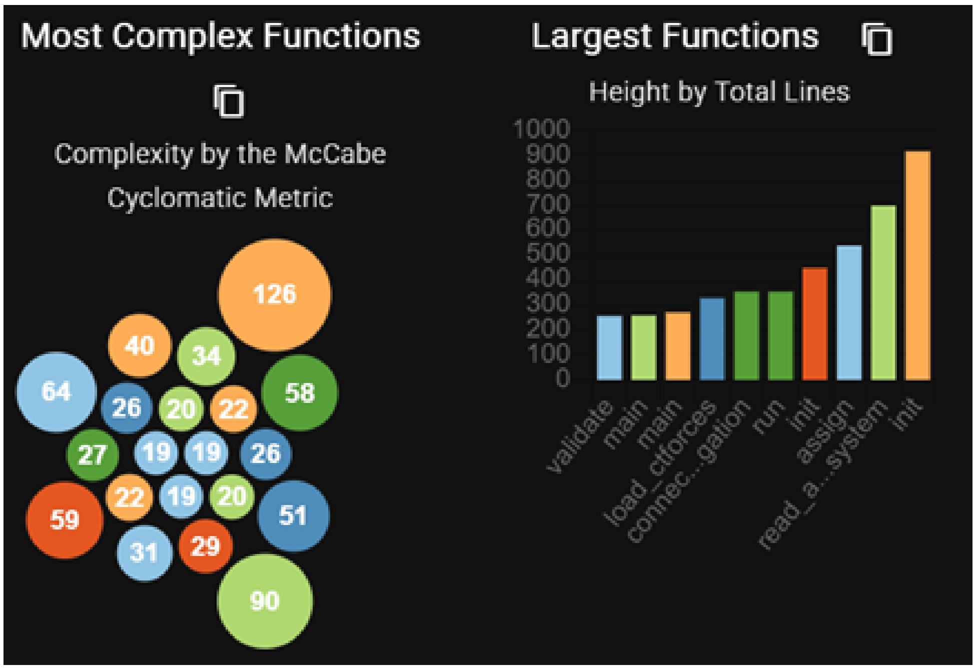


Figure 10: Understand Function Analysis

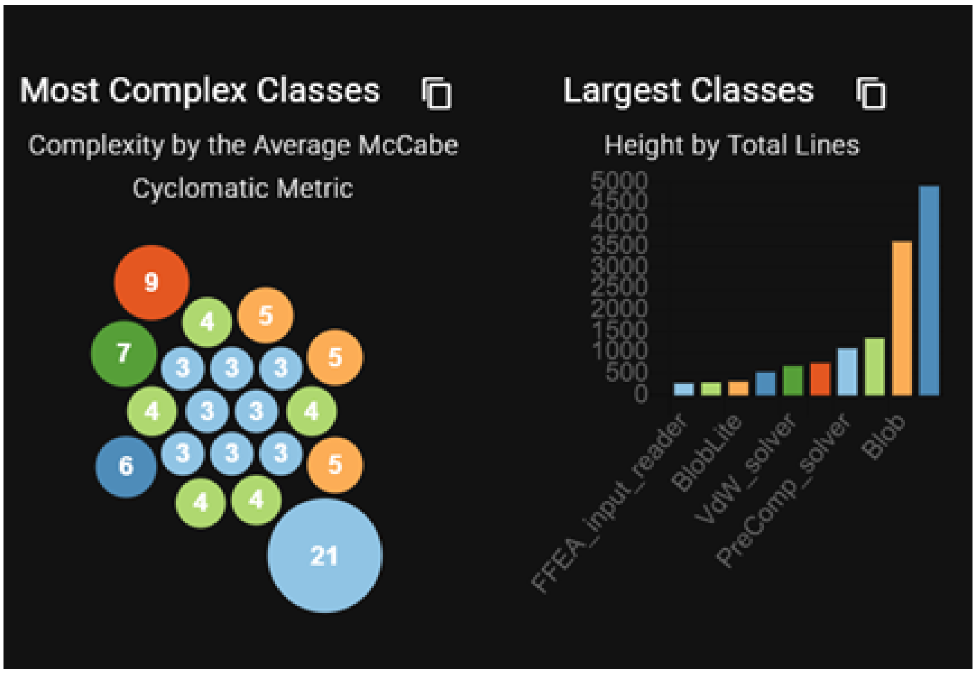


Figure 11: Understand Function Analysis

12. C4 Diagram

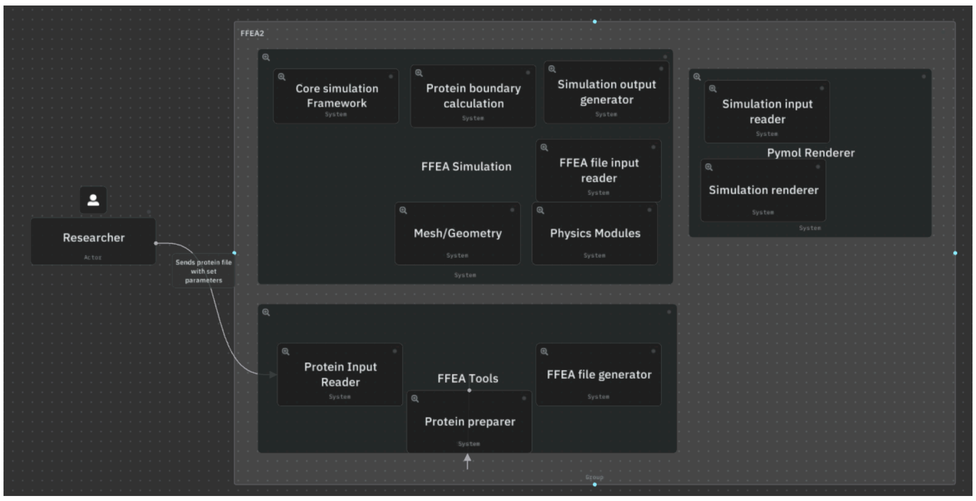


Figure 12: C4 Diagram

13. Dependency Graph

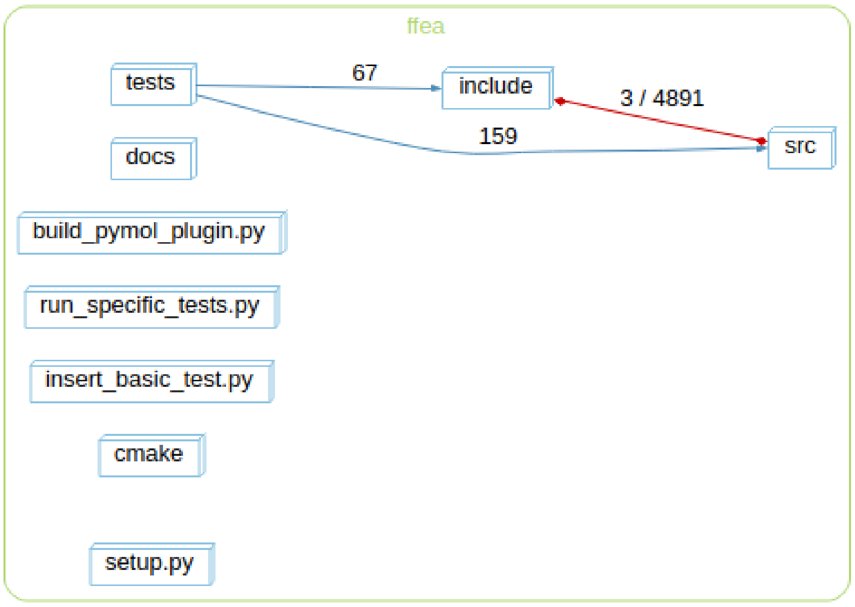


Figure 13: Dependency Graph

14. Class Diagram

Figure 14: Class Diagram

15. References

- Typst Documentation: https://typst.app/docs/