Task 1

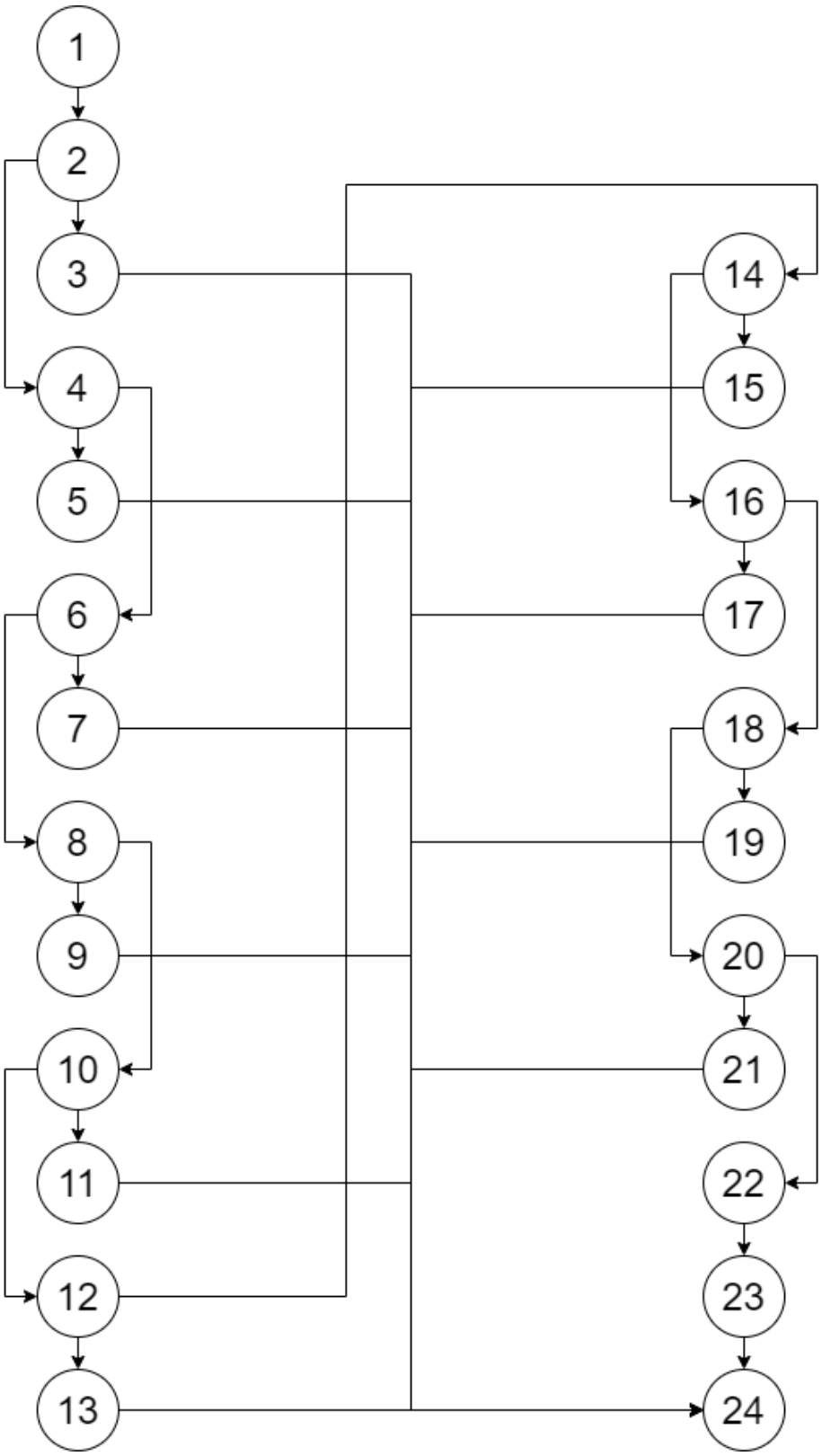

Figure 1. Control Flow Graph for initialiseVehicle *in Controller.java*

Task 2

The first metric for measuring the Java project is **Coupling Between Object classes (CBO)**, which, if reduced, would increase both modularity and encapsulation. By reducing the interaction between objects, like method calling or variable usage between classes, modularity is improved. Furthermore, by increasing modularity, similar data and connected methods are bundled together, thus improving encapsulation.

The second metric is **Lack of Cohesion Of Methods (LCOM)**, which, if decreased, would also increase both modularity and encapsulation. If LCOM is low, it implies that data and methods are bundled well; therefore, encapsulation is at a satisfactory level. Furthermore, low LCOM implies that there is low interaction between methods; thus, modularity holds.

The third metric is **Response For a Class (RFC)**, which is directly linked with complexity because it is a description of the number of method in that class and the number of methods called in that class. Therefore, lowering RFC would lower the complexity of the class.

Task 3

The most important class in *VehicleControlSystem* is ***Controller*** in the *oose.vcs* package, which is indicated by the relatively high measurements of all three metrics described in Task 2. Firstly, *Controller's* CBO is 28, which is approximately 7 times larger than the mean (2.7), which implies that modularity and encapsulation are worsened by this class. Secondly, its RFC is 31, which is about 3 times more than the mean (9.63), implying that the class is increasing complexity. Finally, the class' LCOM is 28, which is around 5 times more than the mean (5.7), again decreasing modularity and encapsulation. Furthermore, by analysing the code, one can tell that *Controller* is a god-class that has too many unconnected responsibilities and that it is incredibly repetitive; thus, it has a lot of room for improvement.

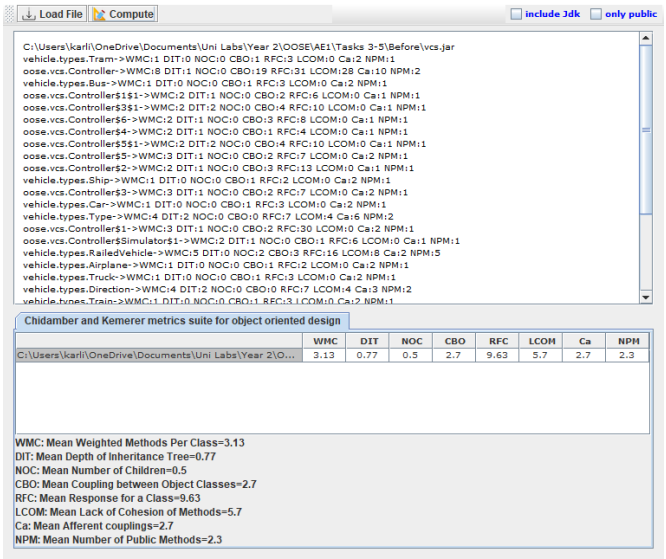

Figure 2. CKMetrics of Controller and mean.

Task 4

One of the refactorings was moving the Simulator class to a different file, which reduced complexity of the *Controller* class and, by definition, improved modularity. Although *Controller's* metrics did not change much, the mean CBO decreased by 1%, which demonstrates that both modularity and encapsulation are increased.

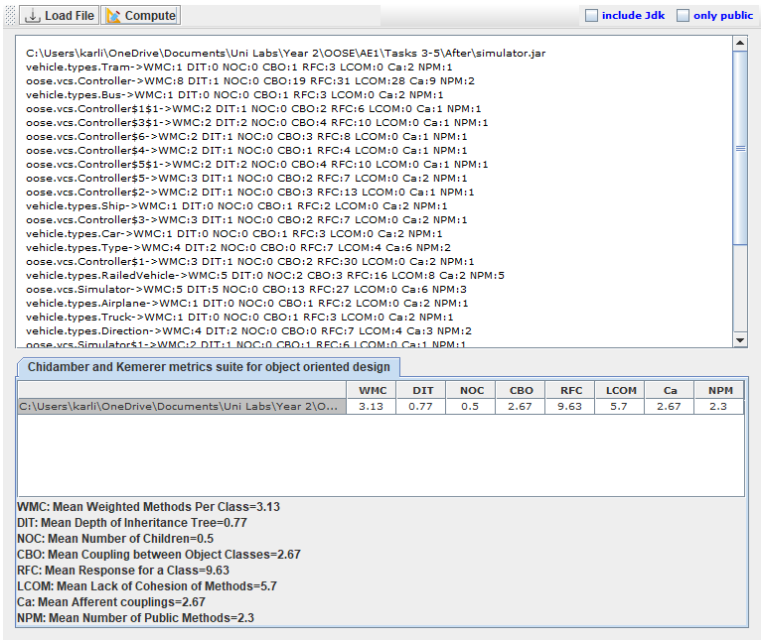

Figure 3. CKMetrics after Simulator refactoring.

The second refactoring was moving *initialiseVehicle()* to another class for increased modularity and encapsulation. Although the mean CBO grew, *Controller's* CBO fell by 53%, as did both the mean and Controller-specific metrics for RFC and LCOM, highlighting the fact that the overall complexity of the project was reduced.

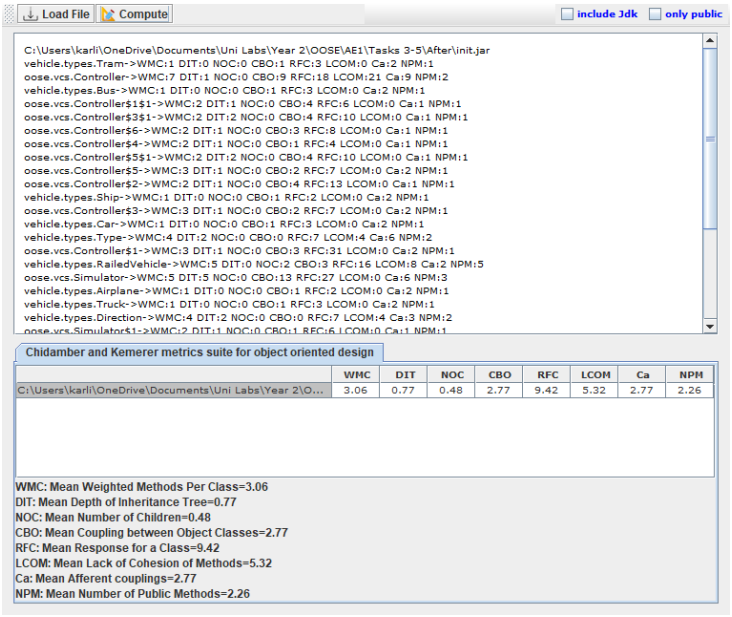

Figure 4. CKMetrics after initialiseVehicle refactoring.

The third refactoring was separating configuration methods out of *Controller*, which was beneficial for separation of concerns, as shown by the 95% decrease in *Controller's* LCOM metric, 72% decrease in the class' RFC, the means of both decreasing for the whole project as well.

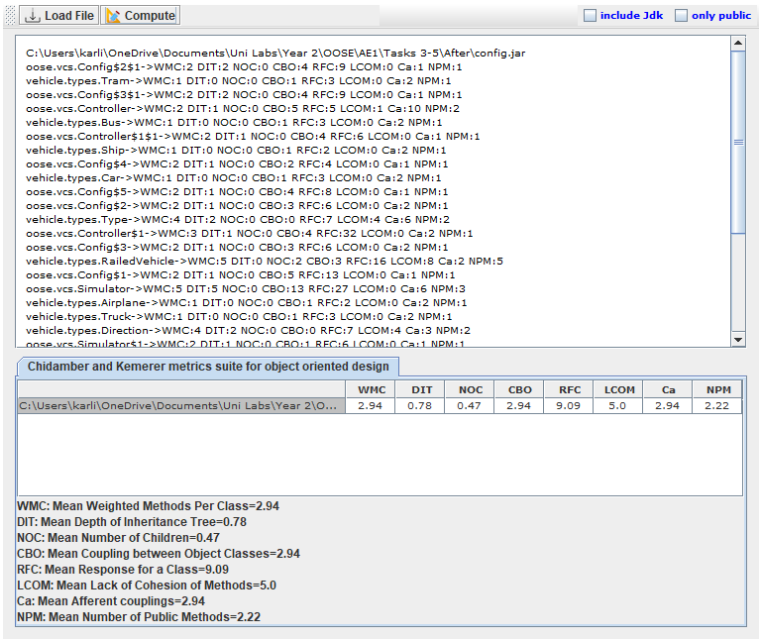

Figure 5. CKMetrics after Configuration refactoring.

Finally, the fourth refactoring was reducing button repetitiveness, which, although it worsened mean metrics, it improved the DRY and Open-Closed principles by increasing modularity and scalability for different features that might be added to the project with different buttons.

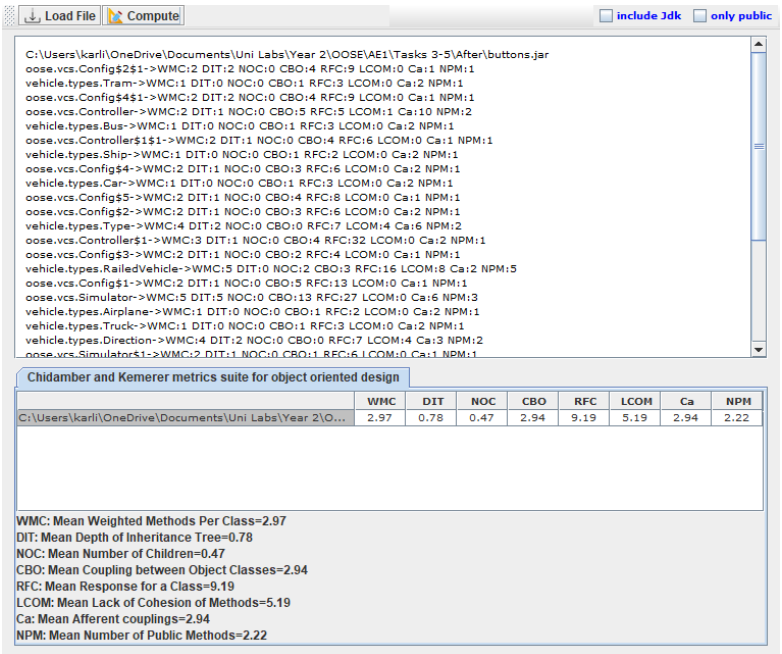

Figure 6. CKMetrics after Button refactoring.

Task 5

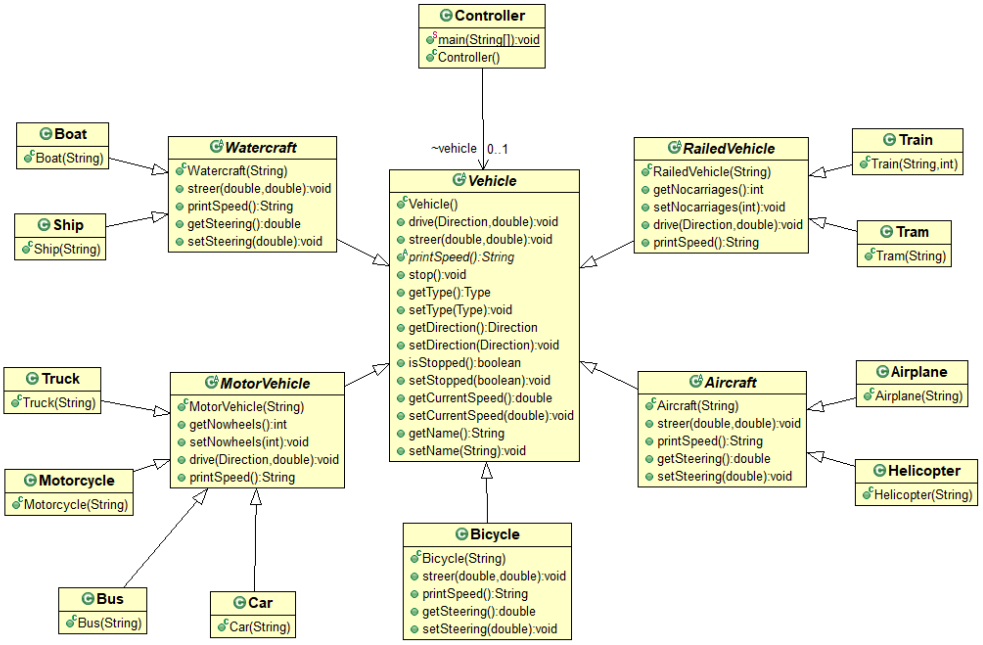

Figure 7. Diagram of VCS before refactorings.

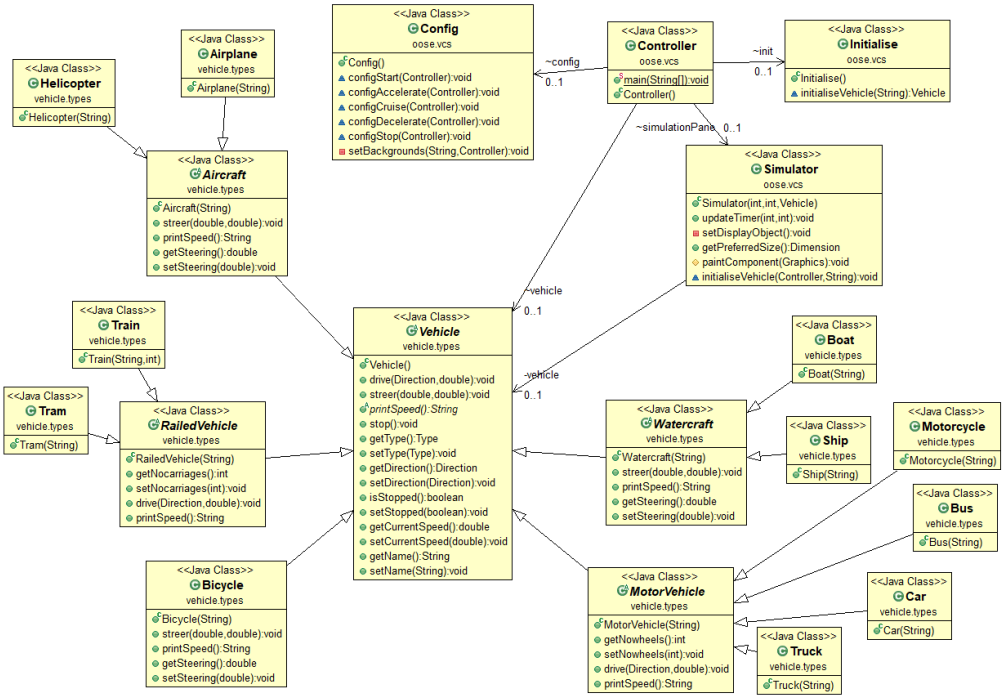

Figure 8. Diagram of VCS after refactorings.

The biggest structural difference after the four refactorings is that the *Controller* class is less of a god-class as three other classes (*Config, Initialise, Simulator*) have been taken out of it. By improving separation of concerns and, in turn, keeping to the Open-Closed Principle in SOLID, the new structure in Figure 8 increases modularity and encapsulation and reduces *Controller's* complexity, which helps with debugging, legibility of code and, by extension, developer fatigue. Firstly, the

Config class is responsible for starting configurations of the GUI and its responses to human interaction, which includes repainting the colours of the buttons and signalling about changes in the vehicle's behaviour. Secondly, the *Initialise* class handles initialising the correct vehicle class based on what is chosen in the combo box. Thirdly, the *Simulator* class manages the simulation of the vehicle's movement across the GUI, including its image and the timer of the system.