Traffic Simulation

CS4227 Software Design and Architecture

Semester 2 2019/20

Group J: E E E EEEE

16187504 – Adam O'Mahony
13132911 – Niall Dillane
16163842 – Joseph Tobin

Repository: https://github.com/joseph-tobin/CS4125-traffic-simulation

Table of Contents

Introduction

We have been contracted by the University of Limerick to create a simulation of traffic flow throughout the campus. Traffic congestion has grown exponentially over the years and they are looking for a piece of software to test different traffic control solutions, in order to facilitate easier movement in and out.

The simulation will primarily be used by the Buildings and Estates Department, who have been experimenting lately with opening an additional exit road in the North Campus, as well as deploying security personnel to congested junctions at various points in the campus. This has come with significant ongoing costs for staff, complications with acquiring permits for students to use the additional exit, and overall confusion.

The user interface will display a simple top-down map view of the university campus, with traffic congestion represented by moving vehicles on the roads, allowing quick and easy visual interpretation. This must be able to be played and paused as the user desires, for closer inspection. In addition, the program will allow users to save their current settings as well as a report on traffic metrics, for finer analysis.

The department need several functions available to them, to modify the simulation as they desire. Traffic flow – both inwards and outwards – must be adjustable, not only allowing them to simulate current levels but also plan for the future. They need to be able to add, remove and adjust existing and additional traffic control measures, including traffic lights, roundabouts and roads.

According to IRIX, Dublin suffered the third-worst loss in hours due to traffic in 2018, worldwide. This corresponds to at least €1600 per driver per year. This seems to indicate we are not particularly good at traffic management, and could use some data-driven help. Our simulation is focused on this small case, but in theory it could be expandable outwards in the future.

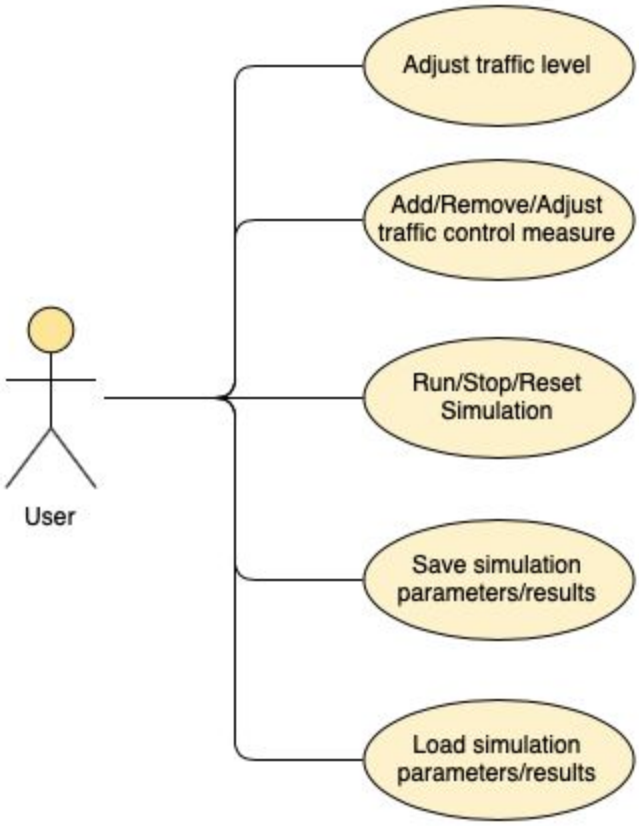
Note: This is a continuation of our previous semester's project in CS4125.

Requirements

Functional Requirements

Requirement	Relevant Use Case
Must be able to play and stop simulation	Run/Stop/Reset Simulation
Must be able to add/remove/modify traffic controls – traffic lights, roundabouts, roads	Add/Remove/Adjust Traffic control measures
Must be able to adjust traffic level (inwards and outwards)	Adjust traffic level
Must be able to save settings of current simulation for use later	Save simulation parameters/results
Must be able to load previously saved simulation parameters	Load simulation parameters/results

Use Case Diagram



Use Case 1	Adjust Traffic Level	
Goal in Context	User increases or decreases traffic flow to change the simulation	
Precondition	Value for traffic flow is within bounds of the program	
Success End Conditions	Traffic flow is adjusted to the according level	
Fail End Conditions	Traffic flow does not change or does not correspond to the requested change	
Primary Actor	End user	
Trigger	Slider interface is moved in the UI	
Descriptions	Step	Action
	1	User moves the traffic flow slider
	2	Loading icon shows while simulation adjusts
	3	Simulation updates with new traffic flow
Extensions	Step	Branching Action
	1a	User sets the traffic level high, in which case they are presented with a warning about performance
	2a	Loading is taking too long, so the user cancels it
Variations		Branching Action
	1a	User may use the slider, manually enter a value, or load a pre-existing simulation
	2a	If the simulation is paused, loading does not occur until it is run again
Related Information	Run/Stop/Reset Simulation	
Priority	High	
Performance	Depends on the change, but no more than several seconds	
Frequency	Potentially multiple times per minute	
Channel to Actors	Not yet determined	
Open Issues	N/A	
Due Date	Week 9	

Use Case 2	Add Traffic Control Measure (TCM)	
Goal in Context	User wants to add a TCM to judge its effect	
Precondition	TCM must be placed on valid location on the map	
Success End Conditions	TCM is added and traffic follows it accordingly	
Fail End Conditions	TCM is not added or traffic ignores it	
Primary Actor	End User	
Trigger	A TCM is dragged from the menu onto the map	
Descriptions	Step	Action
	1	User clicks, drags and drops a TCM onto the map
	2	Loading icon shows while simulation adjusts
	3	Simulation updates with traffic following the new TCM
Extensions	Step	Branching Action
	1a	User changes their mind and drags the TCM back to the menu, cancelling the action
	2a	Loading is taking too long, so the user cancels it
Variations		Branching Action
	2a	If the simulation is paused, loading does not occur until it is run again
Related Information	Run/Stop/Reset Simulation	
Priority	High	
Performance	Seconds	
Frequency	Potentially multiple times per minute	
Channel to Actors	Not yet determined	
Open Issues	N/A	
Due Date	Week 10	

Non Functional Requirements

Accuracy

- Reference data must be reasonably accurate, to have a worthwhile simulation.
- Road network must be accurately represented.
- Traffic flow must be accurately simulated.
- Vehicles must be able to path-find realistically.

Performance

- Performance must be managed to keep the simulation smooth.
- Many iterations will need to be run to find a good combination of parameters and traffic controls, so the simulation needs to adjust quickly.

Usability

- UI should be straightforward and intuitive.
- Users will not necessarily be tech experts.

Quality Attributes

Correctness, Reliability, Adequacy, Learnability, Robustness, Maintainability, Readability, Extensibility, Testability, Efficiency, Portability

There are many non-functional requirements with this project, since creating a functioning representation of traffic is only half the problem. The important part is having something genuinely realistic and applicable to the real world.

Java will be our programming language of choice, since it has a long history of being used in enterprise-level projects of this scale, and comes with many useful tools and libraries that will help optimise our project.

The Processing library will make the UI straightforward to create and thus we can spend more time optimising usability than worrying about scaling issues. This is widely used for GUI projects and fits our requirements perfectly.

IntelliJ is a Java Integrated Development Environment (IDE) that comes with built-in code analysis and optimisation, and easy access to plugins including Checkstyle, FindBugs, and PMD.

Architecture

Overview

The Traffic Simulation software will consist of a graph of connected nodes through which vehicles will navigate. Given a start and destination node, a vehicle will find the most optimal route and travel to its destination while following the rules set out by each node.

Language and Frameworks

Java is a very common language used in enterprise grade systems. We have selected Java as the language to use in this project due to it being a purely object oriented language and being platform agnostic, useful for simulation software.

Based on the Model View Control (MVC) pattern, we have divided the project into two layers, User Interface and Business Logic. We have omitted the data persistence layer due to the lack of data handling that will take place.

The JavaFX library will be used for UI rendering – a popular and long-supported set of functionality, it is the most modern core GUI toolkit for Java, compared to the earlier Swing and AWT. JavaFX also supports the incorporation of CSS for styling UI elements, which aids in making the application more extensible in future (separation of concerns).

The Maven build tool will be used for package management, streamlining the development process in our IDE of choice: IntelliJ. By having this common work environment, we hope to cut down on frustrating technical issues across team members.

Design Patterns

Factory

Devised by the "Gang of Four", the Factory pattern decouples construction of objects from the objects themselves (Gamma 1995). Creating objects directly in the class using it is inflexible and results in high coupling, but by use of the Factory method we decouple the two and defer instantiation to subclasses. This is more extensible and helps maintain the Open-Closed Principle: we can change how the object is created without having to go back and change each individual class that uses it.

For our purposes, this is very suitable for Vehicle creation, which we want to be extensible and open to other Vehicle types in future, as well as the possibility of refactoring the way they are created.

Memento

The memento design pattern is a behavioral design pattern that enables saving and restoring of object states without revealing the details of its implementation. These states are created by the originator (the object that the state originates in) which removes issues with private/hidden information as the originator has complete access to its own information.

Last semester we wanted to have the ability to save/load states of the simulation but never got around to completing it. After finding out about the memento design pattern, we determined that we could leverage it for the purposes of saving/loading simulation states.

Prototype

The Prototype pattern is a creational pattern, wherein a prototypical instance of an object is created and then future projects are cloned from it, instead of being created afresh. This saves on computation cost, and is beneficial in instances where you are creating a large amount of similar objects, or if creating these particular objects involves a significant amount of processing.

This should be useful for our system for vehicle creation. Each vehicle is assigned a random start and end point, and independently does its own route-finding. This is implemented with the A* algorithm which, while fast, is still computationally expensive. By cloning vehicles with the same start and end points, we can avoid this. At least to a certain point, after which we may wish to recheck that the particular route is still best.

Command

The Command design pattern is a behavioural pattern that encapsulates a request into a standalone object, containing all necessary information about the request in that object. This enables the parameterization of methods with different requests, support undoable operations, delay requests, log requests and many more.

The classic implementation for the command pattern is for UI requests being sent to the application logic layer. Encapsulating the requests sent when buttons are pressed on the UI layer, creates a layer of abstraction between the UI and application layer, disconnecting the UI with the implementation of the requests. For our implementation of the command pattern we will use this UI use case and implement undo and redo operations.

Pluggable Adapter

The pluggable adapter is an extension of the adapter pattern which is "used to translate the interface of one class into another interface." (GoF Design Patterns). This is achieved through an adapter class which maps or translates the functionality of the incompatible interface to one that is known. The pluggable adapter differs in that it may not know the type of the class that it is to adapt, known as the adaptee. Due to this, the pluggable adapter is not hard-coded to a specific adaptee, but creates an adapter that can handle a desired adaptee. For our application, this pattern was well suited to be used for outputting logs and was implemented with that in mind.

Interceptor

The Interceptor is an architectural design pattern which allows applications to extend a framework transparently by registering out-of-band services with said framework, via predefined interfaces. These are triggered on certain events (Schmidt et al. 2013). By implementing this pattern, and deriving concrete interceptors from the interface, you offer a way to change the processing cycle of the system, for example by introducing: load balancing, logging, security etc. This is most often found in middleware (Curry, Chambers and Lyons 2004).

A dispatcher is provided to allow applications to register their concrete interceptors with the framework. Then, when an (interceptable) event occurs, the framework notifies the relevant dispatcher to invoke the callback of its registered concrete interceptor, with the necessary context object. Context objects are used to allow the interceptor to control certain aspects of the framework, providing methods to access and modify the framework's internal state.

We felt this would be useful for our project for the purposes of logging at various points. Namely, the vehicle creation and route finding processes, which bring some complication and need to be noted for debugging purposes on occasion. The Interceptor will provide an easy mechanism for extending the framework with this new logging service.

Producer – Consumer (Independently Researched)

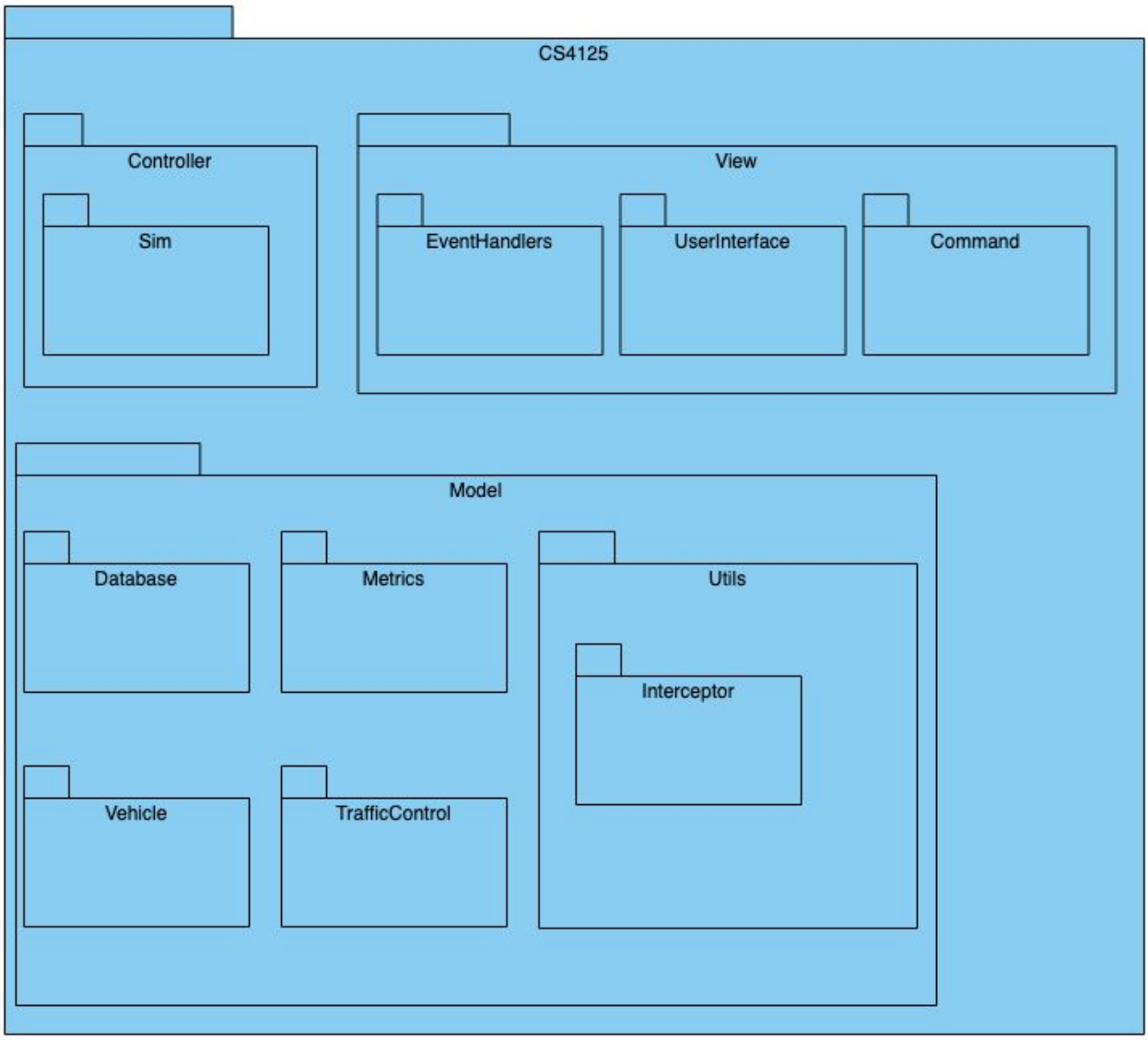
Producer-Consumer is a behavioural design pattern useful for decoupling and deferring work. It comprises three main components - Producer, Buffer and Consumer. A separation of concerns is introduced through this design pattern as the producer is responsible only for the creation of units of work. It does not care how or when those units of work are executed. Whereas the consumer, takes the work that has been produced, regardless of the source (Producer) and executes the work accordingly. The work is passed through a mutually accessible buffer, allowing both consumer and producer to run

concurrently in different threads. The buffer must be thread safe to avoid inconsistencies in the execution of the overall program, e.g. java BlockingQueue. The buffer should also inform the producer when it is full, to avoid the loss of work or data if work is being produced faster than it is being consumed. Unlike the producer, which is dependent on the buffer having available space, the consumer can constantly poll the buffer checking for work. If the buffer contains work, the consumer retrieves it, executes and then continues to poll the buffer.

Last semester, we encountered serious performance issues when running the simulation as we were not sure how to properly animate the calculations that were taking place on the backend of the simulation. Originally, we tried to use a naive callback design, in which the model would send information to the view to be animated and wait on a 'finished' message response to continue execution. This is a terrible design and introduces a huge bottleneck to the system. The producer-consumer design pattern was the best solution for this, allowing vehicle threads to run independently of the view, while also animating the movements of vehicles in the correct order without bottlenecking the simulation performance.

Diagrams

Package Diagram

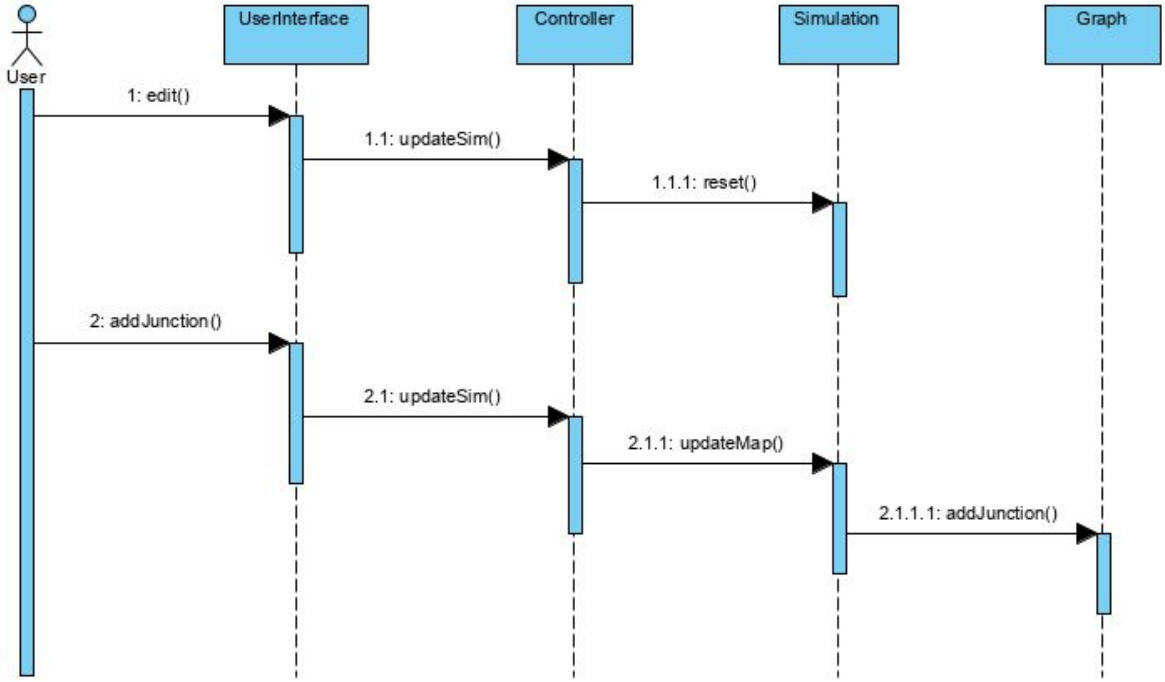


Class Diagram

Too large for display in document; see file in github repository (./classDiagram.png).

Sequence Diagram

Add Traffic Control Measure



Code Snippets

Factory Pattern

Model.Vehicle.VehicleCreator

```
32          vFactory = new VehicleFactory();

66                  v = vFactory.makeVehicle("car", routeStartEnd[0], routeStartEnd[1]);
```

The Factory pattern is fairly straightforward: instead of directly creating new Vehicles, the VehicleCreator thread creates a VehicleFactory, passing it the parameter of the type of Vehicle it wishes to create, with the accompanying route.

```
5   public class VehicleFactory {
6       public IVehicle makeVehicle(String vehicleType, ITCM start, ITCM end){
7           if (vehicleType == "car")
8               return new Car(start, end);
9           return null;
10      }
11  }
```

The Factory itself only allows for the creation of one type of Vehicle at this point, but allows for ease of extensibility in the future.

Prototype Pattern

Model.Vehicle.VehicleCreator

```
21      private Map<String, IVehicle> premade;
22      private Map<IVehicle, Integer> premade_count;
```

Two parallel maps are maintained:

- premade: the key being the concatenated start and end Nodes in a string, associated with the value of the Vehicle that was created for that path
- premade_count: the Vehicle paired with the number of times it has been cloned. This is because pathfinding depends on the congestion of the various edges between Nodes, so we don't want to clone indefinitely. The premade list should be reset occasionally.

```
47              if(premade.containsKey(routeString)) {
48                  // checking if this vehicle has been used to copy more than 5 times, remove after this
49                  IVehicle toCopy = premade.get(routeString);
50                  int count = premade_count.get(toCopy);
51                  if(count > 5) {
52                      Simulation.INSTANCE.logger.info("deleting vehicle: " + toCopy);
53                      v = vFactory.makeVehicle("car", routeStartEnd[0], routeStartEnd[1]);
54                      premade.replace(routeString, v);
55                      premade_count.remove(toCopy);
56                      premade_count.put(v, 1);
57                  }
58                  else {
59                      Simulation.INSTANCE.logger.info("copying vehicle: " + toCopy);
60                      v = toCopy.makeCopy();
61                      premade_count.replace(toCopy, count + 1);
62                      toCopy = null;
63                  }
64              }
65              else {
66                  v = vFactory.makeVehicle("car", routeStartEnd[0], routeStartEnd[1]);
67                  premade.put(routeStartEnd[0].toString() + routeStartEnd[1].toString(), v);
68                  premade_count.put(v, 1);
69              }
```

This is the implementation of cloning itself: checking if the Vehicle has been made before and, if so, if it has been cloned more than 5 times. In this case, the premade lists are reset and the Vehicle is cloned one more time, starting the list afresh.

Model.Vehicle.IVehicle

```
10  public interface IVehicle extends Observer, Runnable, Cloneable {
```

By having the IVehicle interface extend Cloneable, we are able to use the IVehicle.makeCopy() method seen on line 60 above, which is implemented in Car, as it would be in all Vehicle types.

Model.Vehicle.Car

```
35          public IVehicle makeCopy(){
36                  Car carObj = null;
37
38                  try {
39                          carObj = (Car) super.clone();
40                  } catch (CloneNotSupportedException e) {
41                          e.printStackTrace();
42                  }
43
44                  return carObj;
45          }
```

Memento Pattern

Our memento implementation for saving states consisted of a Memento inner class inside the Simulation class. This class contained all the necessary information to save the state of the Simulation class.

Controller.Sim.Simulation

```
328         public class Memento {
329
330                 private List<ITCM> nodeList;
331                 private HashMap<String, IVehicle> routeMap;
332                 private List<IVehicle> vehicles;
333                 private Queue<Move> moveQueue;
334                 private UIController controller;
335                 private IVehicleCreator vc;
336
337
338                 public Memento(List<ITCM> nodeList, HashMap<String, IVehicle> routeMap, List<IVehicle> vehicles, Queue<Move> moveQueue, UIC
339                                 vc) {
340                         this.nodeList = nodeList;
341                         this.routeMap = routeMap;
342                         this.vehicles = vehicles;
343                         this.controller = controller;
344                         this.moveQueue = moveQueue;
345                         this.vc = vc;
346                 }
```

The necessary steps to restore a memento are done in the restoreFromMemento() command, which is performed on a Memento instance. This command is activated through the UI using the command pattern. The collections shallow copied between lines 363 and 369 are necessary to avoid referencing issues.

Controller.Sim.Simulation

```
348                 public void restoreFromMemento() {
349                         // delete all old nodes
350                         for (ITCM n : Simulation.this.getNodeList()) {
351                                 controller.deleteNode(n);
352                         }
353
354                         // re-add memento nodes
355                         for (ITCM n : this.nodeList) {
356                                 logger.info(n.getLabel());
357                                 controller.addNode(n);
358                                 for (ITCM value : n.getAdjacent()) {
359                                         controller.addEdge(n, value);
360                                 }
361                         }
362
363                         Simulation.INSTANCE.nodeList.clear();
364                         Simulation.INSTANCE.nodeList.addAll(this.nodeList);
365                         Simulation.INSTANCE.vehicles.clear();
366                         Simulation.INSTANCE.vehicles.addAll(this.vehicles);
367                         Simulation.INSTANCE.routeMap = (HashMap<String, IVehicle>) this.routeMap.clone();
368                         Simulation.INSTANCE.moveQueue.clear();
369                         Simulation.INSTANCE.moveQueue.addAll(this.moveQueue);
370                         Simulation.INSTANCE.vc = this.vc;
```

Command Pattern

View.UserInterface.Command.ICommand

```
3  public interface ICommand {
4      abstract void execute();
5      abstract void undo();
6      abstract void redo();
7  }
```

View.UserInterface.Command.AddTCMCommand

```
30      public void execute() {
31          Simulation.INSTANCE.addNode(
32                  thisType, thisTCM,
33                  Integer.parseInt(x_inputText), Integer.parseInt(y_inputText), endpoint);
34      }
35
36      /**
37       * Removes the last added node, determined by the history stack in CommandExecutor
38       * Adds the node to deletedNodes, allowing for redo when the node is deleted
39       * As a deleted node has no reference in nodeList in Sim. We must maintain the reference here.
40       */
41      public void undo() {
42          deletedNodes.add(Simulation.INSTANCE.getNode(thisTCM));
43          Simulation.INSTANCE.deleteNode(thisTCM);
44      }
45
46      /**
47       * Re adds the last undo'd removal. Retrieves the nodeToRedo from deletedNodes
48       * Adds the node and readds the adjacencys
49       */
50      public void redo() {
51          ITCM nodeToRedo = deletedNodes.pop();
52          if (nodeToRedo != null) {
53              Simulation.INSTANCE.addNode(
54                      thisType, thisTCM,
55                      Integer.parseInt(x_inputText), Integer.parseInt(y_inputText), endpoint);
56              for (ITCM adjacentNode : nodeToRedo.getAdjacent()) {
57                  Simulation.INSTANCE.addEdge(nodeToRedo.getLabel(), adjacentNode.getLabel());
58              }
59          } else {
60              Simulation.INSTANCE.logger.error("No node found to redo");
61          }
62
63      }
64
65  }
```

In the above example, the concrete command AddTCMCommand (inherited from the abstract ICommand interface) implements the necessary concrete implementations of execute(), undo() and redo(). To enable undo and redo operations a stack (deletedNodes) is maintained, this is necessary to maintain a reference to deleted nodes that need to be ‘redo’d’.

View.UserInterface.Command.CommandExecutor

```
11  public class CommandExecutor {
12      private Stack<ICommand> history = new Stack<>();
13      private Stack<ICommand> undoHistory = new Stack<>();
14      private LoggingAdapter logger = LoggingAdapter.createLogger(BasicLogger .class);
15
16      public void executeOp(ICommand command) {
17          history.push(command);
18          command.execute();
19      }
20
21      public void undo() {
22          if (!history.isEmpty()) {
23              ICommand cmdToUndo = history.pop();
24              cmdToUndo.undo();
25              undoHistory.push(cmdToUndo);
26          } else {
27              logger.info("No commands in the undo stack");
28          }
29      }
30
31      public void redo() {
32          if (!undoHistory.isEmpty()) {
33              ICommand cmdToRedo = undoHistory.pop();
34              cmdToRedo.redo();
35              history.push(cmdToRedo);
36          } else {
37              logger.info("No commands in the redo stack");
38          }
39      }
```

This is our invoker. It enables undo and redo actions, by keeping a history of commands executed and undone. Further implementation such as a logging etc. can be easily added here.

View.UserInterface.UIView

```
commandExecutor.executeOp(new AddTCMCommand(tcmType, thisTCM, x_input.getText(), y_input.getText(), endpoint));
```

Example usage of the invoker using the concrete command AddTCMCommand in the client class UIView. This command is used to add any type of TCM, specified by the tcmType parameter.

```
142                 Button refreshBtn = new Button("Undo");
143                 refreshBtn.setOnAction(event -> {
144                         commandExecutor.undo();
145                 });
```

Example usage of the undo operation. This tells the invoker to pop the previous command executed and undo it.

Producer - Consumer Pattern

Data describing vehicle movements are created in the individual vehicle threads seen below and passed to an accessible BlockingQueue in the simulation singleton. A move is offered to the queue each time a vehicle enters a new road.

Model.Vehicle.Car

```
92          /**
93           * Add Move object to moveQueue in Simulation
94           */
95          private void addToMoveQueue() {
96                  Simulation.INSTANCE.getMoveQueue().offer(new Move(prevNode, currentNode));
97          }
```

Controller.Sim.MoveConsumer

```
7   /***
8    * Consumer thread that will take Move objects as they become available and send them to the UI
9    * thread to animate them.
10   */
11  public class MoveConsumer extends Thread {
12      private Queue<Move> moveQueue;
13
14      public MoveConsumer(Queue<Move> moveQueue) {
15          this.moveQueue = moveQueue;
16      }
17
18      @Override
19      public void run() {
20          consume();
21      }
22
23      /**
24       * Method to constantly take Move object out of a queue and pass it to Simulation to UIController to be animated
25       */
26      public synchronized void consume() {
27          Move m = null;
28          while (true) {
29              m = moveQueue.poll();
30              if(m != null)
31                  Simulation.INSTANCE.addMoveAnimimation(m);
32          }
33      }
34  }
```

The move consumer shown above has been started from within the simulation instance and constantly polls the BlockingQueue for new move objects. When it pulls an object out of the queue, it then passes that object to the View to be animated via the simulation instance.

Pluggable-Adapter Pattern

Model.Utils.LoggingAdapter

```
5  /**
6   * Abstract class to be extended by all concrete adapters that wrap a specific logging framework
7   */
8  abstract public class LoggingAdapter {
9
10     public static LoggingAdapter createLogger(Class<?> adaptee) throws UnknownTypeException {
11         // switch on (known) supported adaptable types
12         switch(adaptee.getName()) {
13             case "CS4125.Model.Utils.BasicLogger": {
14                 System.out.println(">>> BasicLogger.class passed to LoggingAdapter");
15                 return new BasicLogger();
16             }
17             default: throw new UnknownTypeException(null, "Unsupported class " + adaptee.getName() + " passed to LoggingAdapter
18         }
19     }
20
21     // Abstract methods to be implemented by all concrete adapters
22     abstract public void info(String message);
23     abstract public void debug(String message);
24     abstract public void error(String message);
25
26 }
```

Fig. x

For this project, we based the pluggable adapter around a logging system. Seen above in fig x. Is the implementation we used. We chose to use an abstract class as it allows us to use a static factory method to create an adapter specific to an adaptee passed to the method. Lines 22-24 define abstract methods that all concrete classes should implement. Although the current implementation is rather simple, it allows for easy extension in the future with clients having only to create a concrete adapter for their desired adaptee. In this project, we used a simple logger that formats a string with the associated log level (Debug, Info or Error), but this design pattern would allow the use of a more fully featured logging framework (e.g. Logback) without having to change each line where the logger is called by creating an adapter for that specific adaptee. Logging is achieved by calling one of the abstract methods defined above on the concrete logging adapter created.

Interceptor Pattern

Model.Utils.Interceptor.Frameworks.Framework

```
12      private static Framework instance;
13
14      private Framework() {
15          loggingDispatcher = LoggingDispatcher.getInstance();
16      }
17
18      public static Framework getInstance() {
19          if (instance == null)
20              instance = new Framework();
21          return instance;
22      }
23
24      public boolean registerLoggingInterceptor(LoggingInterceptor interceptor) {
25          return loggingDispatcher.register(interceptor);
26      }
27
28      public boolean removeLoggingInterceptor(LoggingInterceptor interceptor) {
29          return loggingDispatcher.remove(interceptor);
30      }
31
32      public void onLogEvent(Context context) {
33          loggingDispatcher.onLogEvent(context);
```

The Framework is implemented as a singleton, and also creates a Dispatcher on creation.

Model.Utils.Interceptor.Dispatchers.LoggingDispatcher

```
9   public class LoggingDispatcher implements Dispatcher {
10      private ArrayList<LoggingInterceptor> interceptors;
11
12      private static LoggingDispatcher instance;
13
14      private LoggingDispatcher(){
15          interceptors = new ArrayList<>();
16      }

32      public void onLogEvent(Context context) {
33          for (LoggingInterceptor interceptor : interceptors) {
34              interceptor.onLogEvent(context);
35          }
36      }
```

Dispatchers are similarly implemented as singleton, in this case the LoggingDispatcher which maintains a list of its attached Interceptors and notably, the onLogEvent method which calls them.

Model.Utils.Interceptor.Interceptors.LoggingInterceptor

```
28      public Context onLogEvent(Context context) {
29          logging.info(context.getMessage());
30          if (context instanceof RouteContext) {
31              logging.debug(((RouteContext)context).getRoute());
32          }
33          return context;
34      }
```

Again a singleton, here we have integration with the PluggableAdapter, creating the specific kind of logger that the user wants. In this case, we only have one implemented.

Furthermore, we have the ultimate onLogEvent method, which calls the logger with the info that the user wishes to display, using the ContextObject provided.

Model.Utils.Interceptor.Contexts.RouteContext

```
5   public class RouteContext extends Context {
6       private List route;
7
8       public RouteContext(String message, List route) {
9           this.message = message;
10          this.route = route;
11      }
12
13      public String getRoute() {
14          return route.toString().replace("CS4125.Model.TrafficControl.", "");
15      }
```

The RouteContext class, extending from the base Context class which simply contains a String with getters and setters, provides access to the route that is generated on vehicle creation, along with some formatting for readability.

Main

```
12          Framework framework = Framework.getInstance();
13          LoggingInterceptor loggingInterceptor = LoggingInterceptor.getInstance("BasicLogger");
14          framework.registerLoggingInterceptor(loggingInterceptor);
15
16          UIView.main(args);
```

On application start, Framework (which creates the Dispatcher) and LoggingInterceptor instances are created and the Interceptor registered. They are now attached and listening for interceptable events.

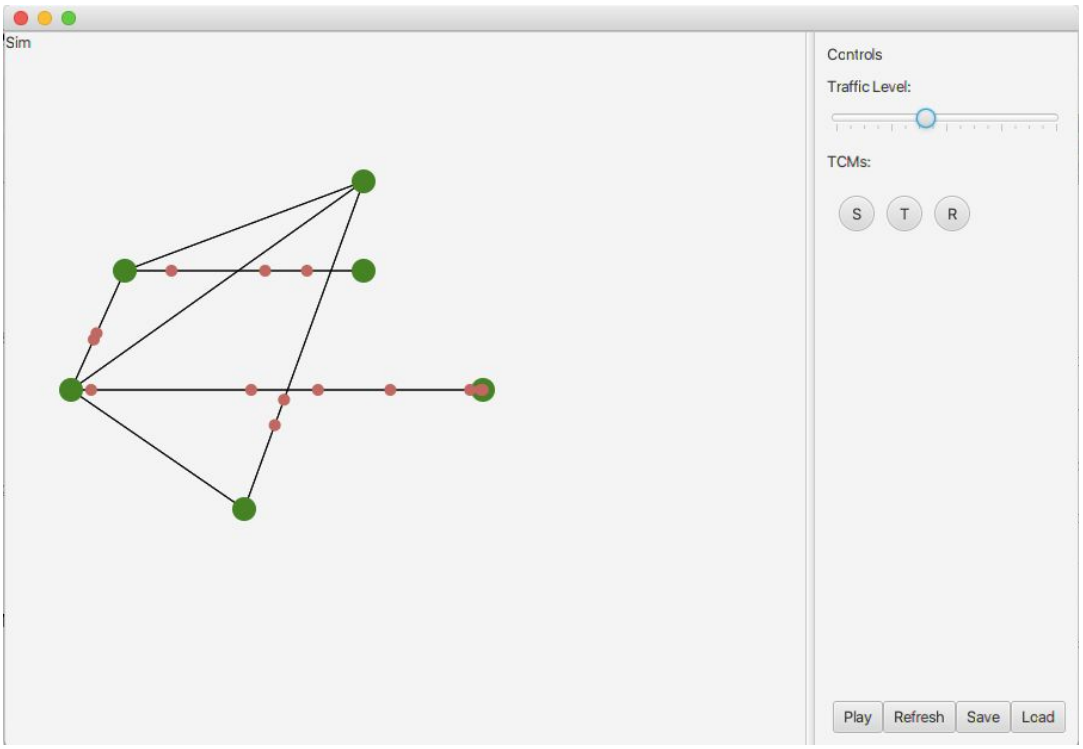
Model.Vehicle.Car

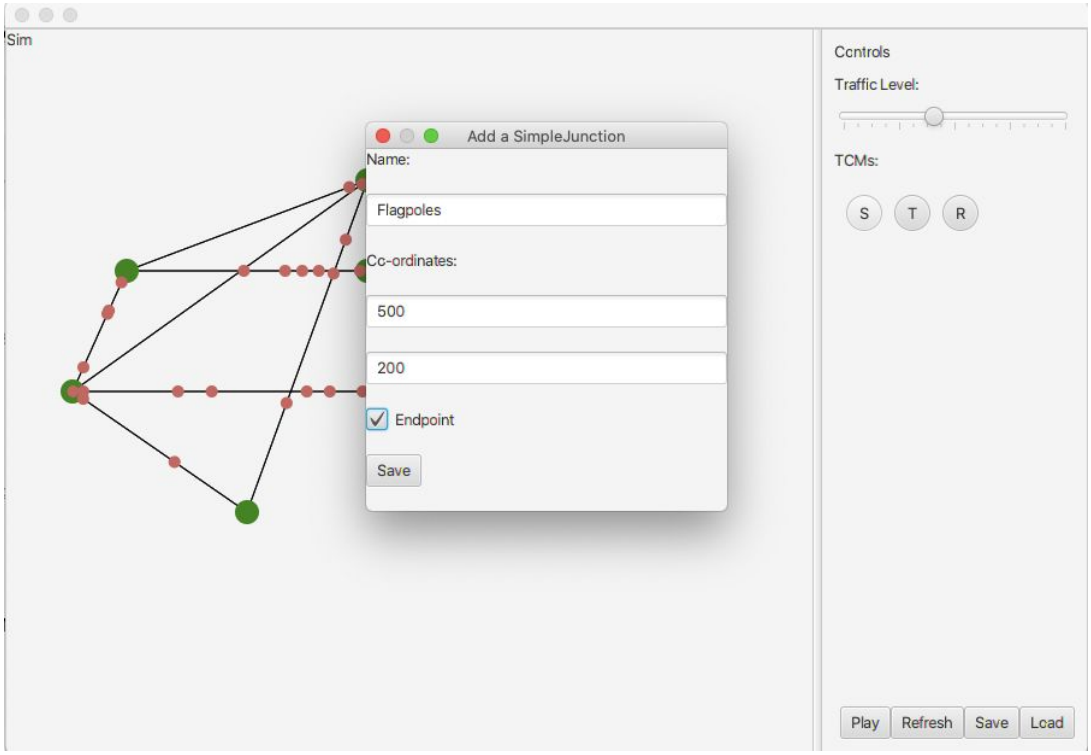
```
26          public Car(ITCM start, ITCM end){
27                  startNode = start;
28                  currentNode = start;
29                  endNode = end;
30                  route = A_Star.findRoute(start, end);
31                  logRoute();
```

```
126          private void logRoute(){
127                  Framework.getInstance().onLogEvent(
128                                  new RouteContext(String.format(
129                                                  "'%s' class created route:",
130                                                  this.getClass().getName()
131                  ), route));
132          }
```

Finally, this is where interception actually occurs. When a Vehicle (in this case Car) is created, it makes a call to logRoute(), which triggers the interceptors in the Framework, passing along a RouteContext object of the route that has just been created.

GUI Screenshots





Added Value

Build Tool: Maven

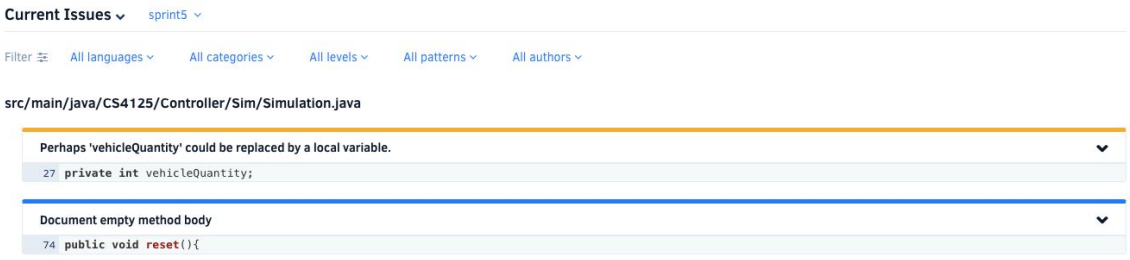
We decided early in the design phase that we would use the Maven build tool through this project. We chose Maven for a few reasons, first of all, there is a huge repository of dependencies and plugins that exists from which we can pull. This allowed us to be dependant on a specific version of a library without having to ensure that the correct version was included locally on our classpath. Changes that we made were simple with the use of xml in the pom, Maven's build file. Any difference in dependencies were immediately downloaded and reflected throughout the project. Another benefit was the easy inclusion of different build plugins that modify how the project was built and packaged. An example of this was our use of the shade plugin in combination with a JavaFX plugin, which allowed us to build an executable fat jar with all the necessary dependencies.

Another reason we chose Maven was the ability to invoke each stage of the build lifecycle separately. The ability to check whether or not our program would build without having to run a full compilation and package lifecycle stages definitely sped up implementation and debugging.

Finally, for us, one of the nicest features of the Maven build tool is the inclusion of the Test lifecycle stage. We were able to continuously test our implementation with every build. Using Maven's default project structure layout, we could place test classes in a testing directory where, upon calling the Maven test lifecycle, test cases are automatically executed and feedback was provided.

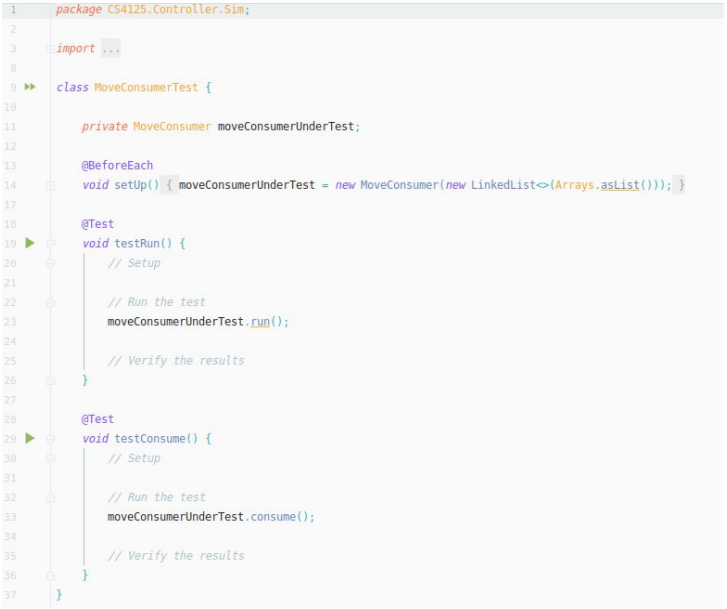
Bad Code Smells: Codacy & GitHub

We enabled a webhook on our GitHub repository (see: References) to use a service called Codacy, which analyses each commit and reports on the quality of code. This wasn't extremely helpful due to the entire project being in a "Work in Progress" state throughout and such there were many times when variables weren't used when we were testing things and many times when code quality wouldn't be what would be expected in a consumer facing system. However, it did help to identify some bad code smells and report on those. For example:



Automated Testing: Squaretest

Squaretest is a plugin for Intellij that aids in automated testing. Using the assigned shortcut in a given file, Squaretest will create boilerplate code in the testing directory. While there was already a similar utility built into Intellij, the code generated by Squaretest was more detailed and closer to a working test cases. An example of a test class generated by Squaretest can be seen here. Private member variables and setup methods are generated for unit test cases, with mock items being created using the Mockito library for any integration tests.

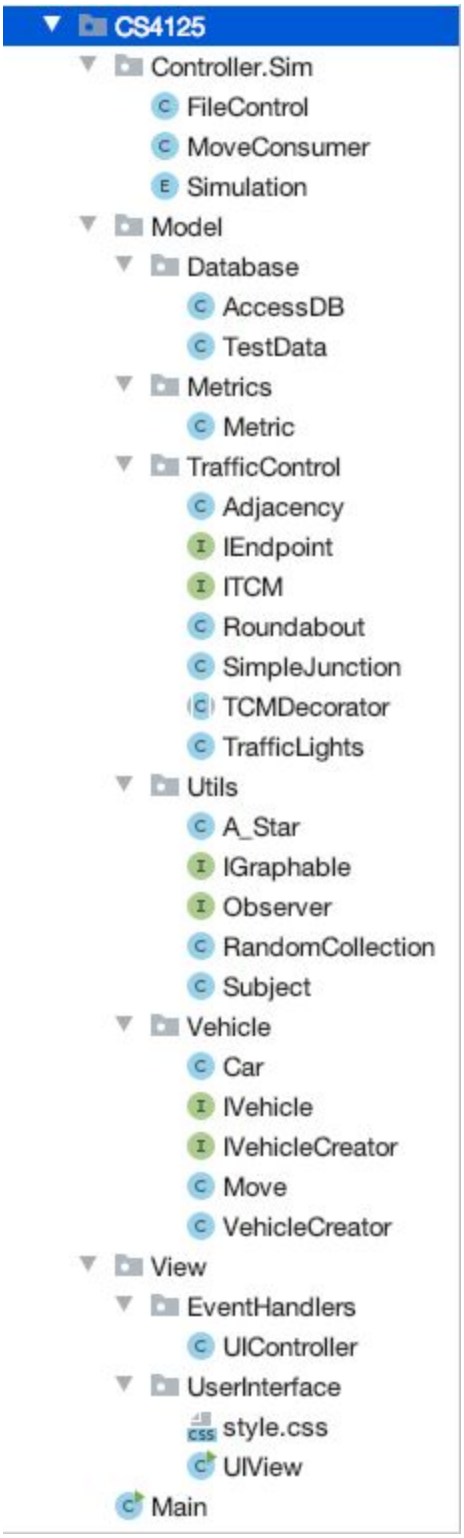


```
package CS4125.Controller.Sim;

import ...

class MoveConsumerTest {

    private MoveConsumer moveConsumerUnderTest;

    @BeforeEach
    void setUp() { moveConsumerUnderTest = new MoveConsumer(new LinkedList<>(Arrays.asList())); }

    @Test
    void testRun() {
        // Setup

        // Run the test
        moveConsumerUnderTest.run();

        // Verify the results
    }

    @Test
    void testConsume() {
        // Setup

        // Run the test
        moveConsumerUnderTest.consume();

        // Verify the results
    }
}
```

Below is an example test case for the A* search algorithm, which shows how little needs to be added to the boilerplate test code generated by Squaretest, decreasing testing time and helping with continuous integration.

```
42      @Test
43      void testFindRoute() {
44          // Setup
45          start.setAdjacent((Arrays.asList(B)));
46          B.setAdjacent((Arrays.asList(start, end)));
47          end.setAdjacent((Arrays.asList(B)));
48
49          final List<IGraphable> expectedResult = Arrays.asList(start, B, end);
50
51          // Run the test
52          final List<IGraphable> result = A_Star.findRoute(start, end);
53
54          // Verify the results
55          assertEquals(expectedResult, result);
56      }
```

Architecture: Model-View-Controller



As you can see from the screenshot, we decided to use a Model-View-Controller architecture. Furthermore, we broke these packages down into sub-packages which we felt reflected distinct components of the project. This helped enforce a separation of concerns and increased comprehension for us while developing the project, as well as for others seeking to understand or contribute in the future.

For the Controller, this was simple enough, just consisting of a Simulation class which was effectively the core of the project – managing creation of vehicles/TCMs and their representations in the business logic and UI.

Model consists of all the various entity classes grouped into relevant packages for clarity. Some, like Database and Metrics, are not as yet fully implemented, but we felt it was important to support this extensibility.

View is straightforward enough, since the project in its current state only offers one main UI, but we did separate out the event handling which interacts with the Controller, in order to keep the visual UI elements decoupled.

Supporting extensibility is a common theme, and a benefit of MVC is allowing us to theoretically introduce many new views – metrics, database interaction, various simulations at the same time, just to name a few examples.

This made it easier for all of us to work on the application in parallel: one person could modify the View without worrying about changing the Model, since we all maintained a common understanding of what parameters were being passed around.

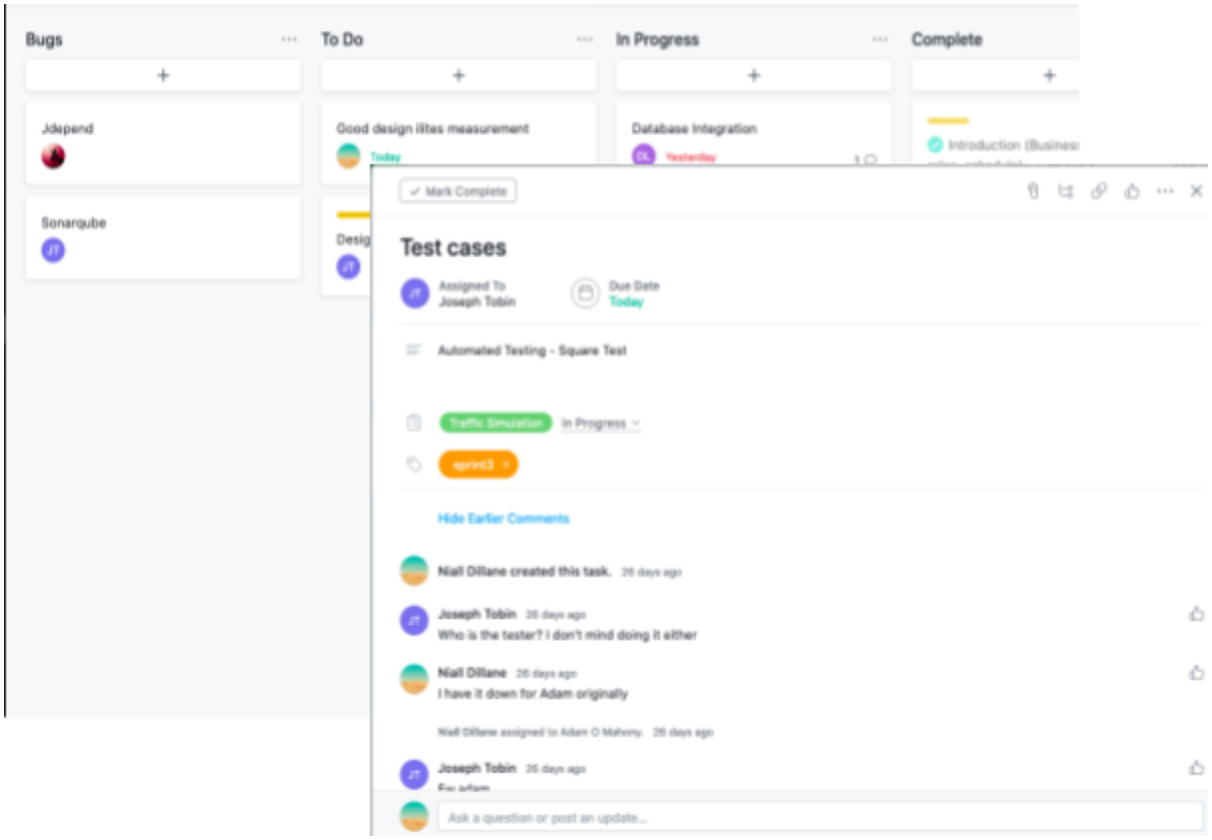
At various times we found ourselves mistakenly starting with animations in the View and aiming to replicate that in the Model, but we eventually realised that it should be the other way around, with the UIsimply being a reflection of the business logic. This was a good learning experience.

Project Management: Asana

We used Asana as a project management tool, which was very helpful for assigning tasks, tracking progress and issues, and generally making sure we were all on the same page with what needed to be done.

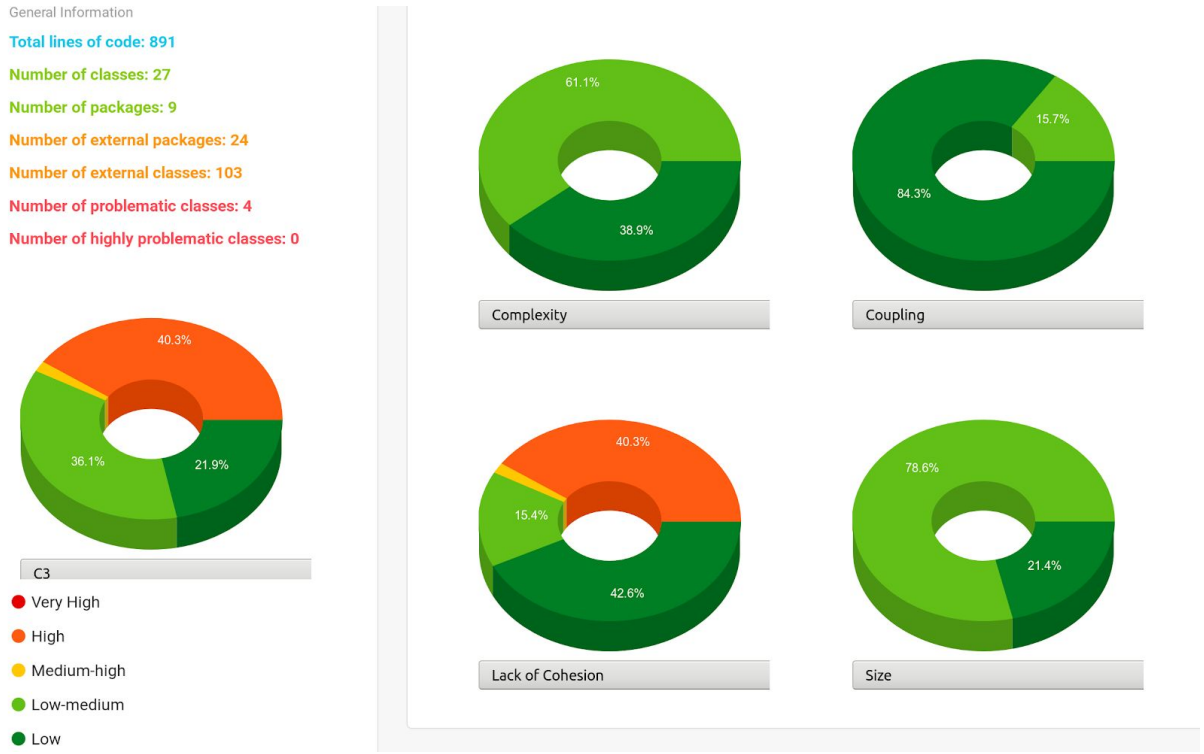
In our weekly meetings, we would discuss the progress made to date, update the running tasks accordingly, and add any new tasks which were discovered in the interim. We could set assignees, tags and deadlines, as well as comments and subtasks if we wished to update or clarify a task. This saved having to email, arrange a meeting, or resort to a cluttered group chat for every query.

A sample screenshot of our “Board” and a task popup is included below:

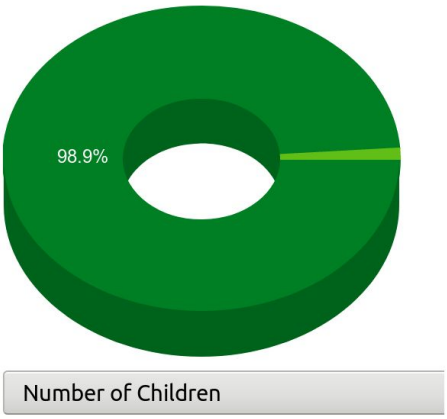


Code Quality Analysis: CodeMR

Learning about dependency management is one of the goals of this module, a tool to visualize this is essential. CodeMR is an IntelliJ plugin used to generate graphical representations of code quality such as cohesion, size, coupling and size. It can generate many different representations of a code base in a package structure, sunburst and tree map view. Package dependency can also be visualized. We used this feature throughout our development, creating five different models to view the quality of code and it's progression.
Screenshots of metrics given from our final code version can be seen below.

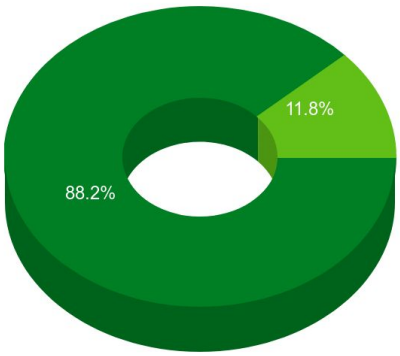


We are quite happy with our low and low-medium complexity and coupling. A challenge we faced throughout the module is trying to visualise what good dependency management looks like and whether or not the code we write has managed dependencies. CodeMR allowed us to visualise this.



Number of Children

Number of children represents the number of direct subclasses of a class, it is an approximate measure of how an application reuses itself. We have very low numbers of children for our classes, as a result, there is less responsibility on the maintainer of the class not to break the children's behaviour. Therefore making it easier to modify the class and require less testing.

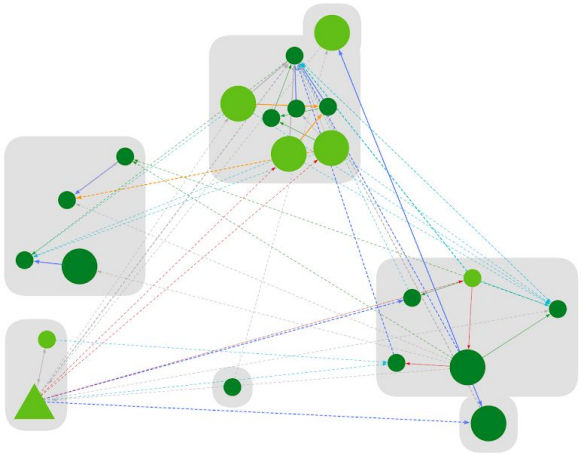


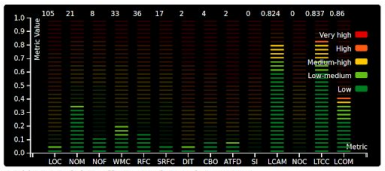
Access to Foreign Data

Access to foreign data is a good metric I think, it shows the number of classes whose attributes are directly or indirectly reachable from the investigated class. Classes with a high access to foreign data value rely strongly on data of other classes and can be a sign of a God Class. This was a worry for us with the Simulation class, but it looks like that is not the case.

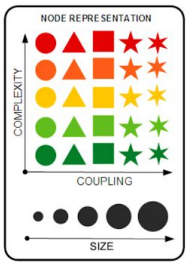
ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION
1	Simulation					140	low-medium	low-medium	high
2	UIView					134	low-medium	low	low
3	SimpleJunction					105	low-medium	low	high
4	Roundabout					58	low-medium	low	high
5	TrafficLights					56	low-medium	low	high

Our most problematic classes as seen above were Simulation and our traffic control measure classes. I think this was inevitable due to the traffic control classes being instantiated using decorator pattern and the Simulation class being the ‘main’ class of the program.





CS4125.Model.TrafficControl.SimpleJunction
Coupling: low
Complexity: low-medium
Lack of Cohesion: high



This is an example of a graph generated by CodeMR. The graphs are only graphical representations of the entire list given previously at different levels of abstraction. The graphs are interactable and are better served by opening the web page in the USB drive under the CodeMR folder html -> main report -> index.html.

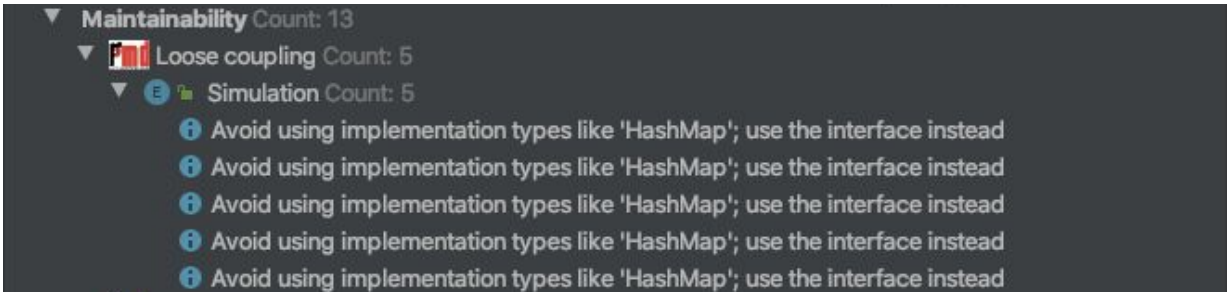
Refactoring

Bug fixing from CS4125's project

When nodes were removed and a new node was added, the removed nodes would still be available to be added as connections to the new node. This was due to the fact that the available nodes for connection was not being taken from the nodeList in Simulation. This nodeList is maintained when mementos are loaded and saved for example.

QAPlug

QAPlug is an IntelliJ plugin for code analysis, helping to find bad code smells and advise on solutions. There were some false positives and it didn't catch anything, but it was useful for identifying small issues that can add up over time. One such example was found here:



The code in question, in Controller.Sim.Simulation:

```
30          private HashMap<String, IVehicle> routeMap;
```

Which was refactored to:

```
28          private Map<String, IVehicle> routeMap;
```

Refactoring the logging system

Originally, throughout the application, the only form of logging was performed via System.out.print(). While this worked, it made understanding the running of and debugging the simulation more difficult. We saw a good opportunity to solve this issue through implementing the *Pluggable Adapter* design pattern.

```
[Mon Apr 06 19:58:16 IST 2020] {Logging Interceptor} INFO - 'CS4125.Model.Vehicle.Car' class created route:
[Mon Apr 06 19:58:16 IST 2020] {Logging Interceptor} DEBUG - [TrafficLights@59629429, TrafficLights@2d425072, TrafficLights@6840dbbe, TrafficLights@669c8eb7]
[Mon Apr 06 19:58:17 IST 2020] {Logging Interceptor} INFO - 'CS4125.Model.Vehicle.Car' class created route:
[Mon Apr 06 19:58:17 IST 2020] {Logging Interceptor} DEBUG - [TrafficLights@59629429, TrafficLights@2d425072, TrafficLights@27ba3b02]
[Mon Apr 06 19:58:18 IST 2020] {sim-inst} INFO - copying vehicle: CS4125.Model.Vehicle.Car@1e549f9f
exiting queue
```

Any class which needed output had a LoggingAdapter instance that it could call, either locally in the class or a public instance in the simulation. With access to these instances, all calls to system.out.print() were changed to a call to the logger, completing the refactor.

Refactoring animations to utilize the Producer-Consumer pattern

One issue we encountered was that of accurately representing the calculations that were taking place on the backend on the UI. The solution we had forced a tight coupling between the View and Model components and introduced a large bottleneck on the simulation. When a vehicle had completed a move from one intersection to another (moved along a road), an instance of that vehicle was passed to the UI controller to execute the corresponding animation. Here lies the bottleneck of the design, where a vehicle waited until its move() method was called by the UI controller before continuing execution. Line 118 below shows the call to a vehicle's move() method from within the UI Controller.

```
115                 t.setOnFinished(event -> {
116                         t.stop();
117                         view.getSimPane().getChildren().remove(c);
118                         v.move();
119                         if(v.getNextNode() != null)
120                                 addAnimation(v, Simulation.INSTANCE.getJourneyTime(v));
121                 });
```

1. As the UI needed only the start and end points and a journey time for an animation, we removed all references to the Vehicle type in the UI and introduced a new Move class. This new class modelled the action of a vehicle's movement from one intersection to another, without needing reference to the vehicle itself. This enabled us to represent what was happening in the simulation without referencing any of the vehicle objects.
2. Next we modified the addAnimation() method in simulation to handle the new Move objects instead of vehicles. Now the Vehicle objects can run in the background independently of the animations.

```
59          private void addToMoveQueue() {
60                  Simulation.INSTANCE.getMoveQueue().offer(new Move(prevNode, currentNode));
61          }
```

3. Once the front and backend was decoupled, we implemented a consumer thread that would constantly check for new Move objects to animate. The the consume() method of this thread can be seen below. By having vehicles offer their moves to a queue instead of directly to the UI, we could create a buffer, allowing the backend to run at full speed without overwhelming the frontend with too many animations.

```
21      public synchronized void consume() {
22          Move m = null;
23          while (true) {
24              m = moveQueue.poll();
25              if(m != null)
26                  Simulation.INSTANCE.addMoveAnimimation(m);
27          }
28      }
```

Now the vehicles continue travelling along their calculated route between intersections, passing relevant information about their movement to the simulation to be animated at a later time.

Prototype Refactor

Code snippets for the relevant parts discussed here can be found above, but for the purposes of documentation we have outlined our process of refactoring here. For reference, Vehicles were previously created like so:

```
28      @Override
29      public void run() {
30          while (true && !(Thread.interrupted())) {
31              ITCM[] routeStartEnd = getRandom();
32              IVehicle v = new Car(routeStartEnd[0], routeStartEnd[1]);
33
34              new Thread(v).start();
```

1. Vehicle (and its associated route finding) was identified as a class from which lots of similar objects were being created, with significant computational cost.
2. IVehicle was chosen as the prototype interface, which we made extend the Cloneable class, using Java's built in cloning feature.
3. Car, our concrete prototype, implemented the makeCopy() method which calls super.clone().
4. VehicleCreator, our client class, utilised this makeCopy() function on Vehicles which were to be created with the same route as previous ones, provided that Vehicle had not been cloned more than 5 times.

Ultimately, Vehicle creation now looks like this:

```
36      @Override
37      public void run() {
38          while (!(Thread.interrupted())) {
39              ITCM[] routeStartEnd = getRandom();
40              String routeString = routeStartEnd[0].toString() + routeStartEnd[1].toString();
41              IVehicle v = null;
42
43              /*
44                Prototype Design Pattern Implementation
45                checking if we already made a vehicle with these start & end points
46               */
47              if(premade.containsKey(routeString)) {
48                  // checking if this vehicle has been used to copy more than 5 times, remove after this
49                  IVehicle toCopy = premade.get(routeString);
50                  int count = premade_count.get(toCopy);
51                  if(count > 5) {
52                      Simulation.INSTANCE.logger.info("deleting vehicle: " + toCopy);
53                      v = vFactory.makeVehicle("car", routeStartEnd[0], routeStartEnd[1]);
54                      premade.replace(routeString, v);
55                      premade_count.remove(toCopy);
56                      premade_count.put(v, 1);
57                  }
58                  else {
59                      Simulation.INSTANCE.logger.info("copying vehicle: " + toCopy);
60                      v = toCopy.makeCopy();
61                      premade_count.replace(toCopy, count + 1);
62                      toCopy = null;
63                  }
64              }
65              else {
66                  v = vFactory.makeVehicle("car", routeStartEnd[0], routeStartEnd[1]);
67                  premade.put(routeStartEnd[0].toString() + routeStartEnd[1].toString(), v);
68                  premade_count.put(v, 1);
69              }
70
71              new Thread(v).start();
```

Command Refactor

1. We declare a command interface, ICommand with three methods, execute, undo and redo.
2. Extract relevant requests into concrete command classes. Any request sent from the UI to Simulation was extracted into a concrete class.
3. Create an invoker that sends commands only via the command interface ICommand. This is our CommandExecutor class. This will handle the execution of commands when invoked by the client, in our case, the UIView class.
4. Change the client, UIView to execute the equivalent concrete commands as opposed to the direct requests to the receiver, Simulation.

This loosened the coupling between UIView and Simulation, by abstracting the implementation of the commands to the concrete command classes.

Previously, for instance, the memento save request was called directly in the UIView class like so.

```
124                 Button saveBtn = new Button("Save");
125                 saveBtn.setOnAction(event -> {
126                         Simulation.INSTANCE.saveToMemento();
127                 });
```

This created tight coupling between UIView and Simulation, and thus was refactored through the use of the command design pattern as shown below.

```
123                 Button saveBtn = new Button("Save");
124                 saveBtn.setOnAction(event -> {
125                         commandExecutor.executeOp(new MementoSaveCommand());
126                 });
```

```
6   public class MementoSaveCommand implements ICommand{
7
8       public void execute() {
9           Simulation.INSTANCE.saveToMemento();
10      }
```

```
7   public class CommandExecutor {
8       private Stack<ICommand> history = new Stack<>();
9       private Stack<ICommand> undoHistory = new Stack<>();
10      private LoggingAdapter logger = LoggingAdapter.createLogger("Command Executor", BasicLogger.class);
11
12      public void executeOp(ICommand command) {
13          history.push(command);
14          command.execute();
15      }
```

Problems Encountered

While implementing the Prototype pattern, we initially made the mistake of not removing the old Vehicle being copied when it had exceeded the count limit. Instead, we were simply resetting the count and copying again. This meant that routes were not being recalculated at regular intervals and the pattern was not suitable for our use. This was rectified as seen above.

The Interceptor proved challenging, and significant research of explanations and previous implementations was required. Ultimately, we went with the singleton approach for simplicity, as a logging service didn't necessarily need a fresh dispatcher and/or interceptor for each logged event. We worried about this clashing with the PluggableAdapter which was also implemented with logging in mind, but actually it turned out that they meshed together perfectly, with the Interceptor using the Adapter.

Critique

Support for relevant NFRs through patterns selected (using scenarios)

Do the patterns selected support relevant architectural use cases? Alternatives?

The Factory pattern was an obvious choice for Vehicle creation, and something that probably should have been implemented in the previous semester's project. We wish to provide extensibility for different types of Vehicle in future, and this pattern is designed with that in mind. Perhaps, with the introduction of driver behaviour further down the line, this could be adjusted into an Abstract Factory.

We had discussed the possibility of cloning behaviour in the previous semester, but never pursued it due to concerns over how it would be implemented and take into account route recalculation. However, with the recognised conventions of the Prototype pattern in place, as well as making sure that routes are regularly updated for start–end pairs, I believe this pattern was ideal for our situation. Especially in cases where the user wishes to increase Vehicle production, objects can be created at a very fast rate, and cutting down on the heavy computation cost associated with A* route-finding is beneficial for performance.

The flyweight pattern was also considered, but deemed unsuitable due to the regular deletion of objects and the need to maintain timers for metrics. Perhaps with a more significant design overhaul this could be possible.

Typically used in middleware, the Interceptor was perhaps the least logical introduction to our system. It was an overkill implementation for logging in our local simulation (the PluggableAdapter alone was likely enough), but served the purpose of making us more aware of the potential of the pattern and how to implement a more complicated architectural design pattern. As such, we did find it a useful learning experience and were glad it was included as a mandatory pattern. We can see how it would be useful in other cases, such as security, too.

Each vehicle that is being modelled in the simulation gets its own thread to run on. This may introduce problems with not having sufficient resources to run more complex simulations. One solution that could have been explored but was not in this project was using a thread pool to limit the number of overall threads being used while running the simulation and reusing them where possible.

References

Curry, E., Chambers, D. and Lyons, G., 2004, May. Extending message-oriented middleware using interception. In *Third International Workshop on Distributed Event-Based Systems (DEBS'04), ICSE* (Vol. 4).

Explanation of Producer/ Consumer design pattern: https://www.geeksforgeeks.org/producer-consumer-solution-using-threads-java/

Gamma, E., 1995. *Design patterns: elements of reusable object-oriented software*. Pearson Education India.

GitHub Repo: https://github.com/joseph-tobin/CS4125-traffic-simulation

IRIX Traffic Data: http://inrix.com/scorecard/

Lines of Code per Contributor script: https://gist.github.com/amitchhajer/4461043

Schmidt, D.C., Stal, M., Rohnert, H. and Buschmann, F., 2013. *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects* (Vol. 2). John Wiley & Sons.

Package / Class / Authors / LoC

Package Name	Class/File Name	Author(s)	LoC
Model.Database	AccessDB	Lavelle	39
Model.Metrics	Metric	Tobin	14
Model.TrafficControl	Adjacency	O'Mahony	46
	IEndpoint	O'Mahony	10
	ITCM	O'Mahony	31
	Roundabout	O'Mahony	116
	SimpleJunction	O'Mahony	215
	TCMDecorator	O'Mahony	18
	TrafficLights	O'Mahony	114
Model.Utils	A_Star	O'Mahony, Dillane	153
	BasicLogger	O'Mahony	34
	IGraphable	O'Mahony	30
	LoggingAdapter	O'Mahony	26
	Observer	Lavelle	6
	Subject	Lavelle	31
Model.Utils.Interceptor.Contexts	Context	Dillane	24
	RouteContext	Dillane	16
Model.Utils.Interceptor.Dispatchers	Dispatcher	Dillane	5
	LoggingDispatcher	Dillane	37
Model.Utils.Interceptor.Frameworks	Framework	Dillane	35
Model.Utils.Interceptor.Interceptors	Interceptor	Dillane	4
	LoggingInterceptor	Dillane	35
Model.Vehicle	Car	Dillane, O'Mahony, Lavelle	133
	IVehicle	Lavelle, Dillane	20
	IVehicleCreator	Tobin	11
	Move	Tobin	29

	VehicleCreator	Tobin, Dillane	110
	VehicleFactory	Dillane	11
View.EventHandlers	UIController	Dillane, Tobin	200
View.UserInterface	UIView	Dillane, Tobin	266
	style.css	Dillane	3
View.UserInterface.Command	AddTCMCommand	Tobin	65
	CommandExecutor	Tobin	40
	ICommand	Tobin	7
	MementoRestoreCommand	Tobin	25
	MementoSaveCommand	Tobin	20
	SliderChangeCommand	Tobin	21
Controller.Sim	Simulation	Tobin, Dillane, O'Mahony	404
	MoveConsumer	O'Mahony	34
	FileControl	Tobin	12
~	Main	Tobin, Dillane	18

Student LoC

Student Name	Lines of code
Adam O'Mahony	992
Daire Lavelle (prev. semester only)	73
Joseph Tobin	733
Niall Dillane	718

Total LoC

Total Packages	Total Classes	Lines of code
13	41	2819

Note: Calculation Method

Package/Class/Total code calculated by the *IntelliJ Statistic plugin.*

Individual code contribution calculated by script in project root directory:

```
git ls-files | while read f; do git blame --line-porcelain $f | grep '^author
'; done | sort -f | uniq -ic | sort -n
```