System Design Document for the Chaotic Traffic Simulator of Group 5

Gustaf Ringius, Andreas Löfman, Robert Wennergren, Felix Bärring

May 20, 2014

Contents

1 Introduction

The application is an experiment more than an application to be served as a finished flawless product to a customer since there simply isn't enough development hours to produce such a product considering the amount of backbone that is needed to develop a base to even present everything in a potential GUI. This software will also experiment with audio handling, GUI building from only a basic canvas and implementing advanced data structures such as for example a weighted directed graph to simulate the traffic that we as humans face every day.

1.1 Design goals

The main design goals for this application are extendability and modularity to make it possible to add additional layers and functionality to the core model since we will probably continue to play with this application even after the final date of the course. In simple terms the application should have as high cohesion and as low coupling as possible. Classes should only have one responsibility to follow the single responsibility principle. The application should naturally attempt to follow SOLID and MVC. All calls between the model and GUI should go through a centralized front controller. The core code of the model that we have written must be tested to at least 70% coverage. There should be no circular dependencies between packages. Design patterns should be chosen with care and the choices will be explained and discussed further down in this document. The software should not consist of the worst code smells and hard coded values should be avoided as much as possible. There shouldn't be any copy paste code and no code duplication. The importance of a fully functioning GUI is not of very high priority. There should be a nice sleek GUI but it doesn't have to support all the bells and whistles currently implemented in the model at the final presentation date for the project. The software should be platform independent and all plugins and extensions used need to be available for at least windows, mac and GNU/Linux platforms.

1.2 Definitions, acronyms and abbreviations

- Open/Closed, means that the software should be open for extension and closed for modification.[1]
- Liskov substitution principle, means that subclasses shouldn't be able to change the behavior [2] it inherits from the super class by overriding the methods and using another specification that violates the specification of the methods in the super class.
- Subtype, an instance of a sub class that doesn't violate Liskov's substitution principle. [2]
- Decorator pattern, (sometimes Wrapper) is a pattern that adds behaviors to individual objects without changing the behavior of all instances of the object's class. [3]
- Strategy pattern, is a pattern that makes it possible to select algorithm behavior at runtime. [3]
- Observer pattern, is a pattern where an object contains a list of listening objects and notify the listeners upon a state change. [3]

- MVC, (model, view, controller) is a software architectural pattern that splits an application into three distinct areas of responsibility that have a very low coupling between each module and very high cohesion inside each module. [3]
- Singleton, is a way to make sure that there only exists one instance of an object at all times. [3]
- Encapsulation, means limiting the scope of certain data, see scope for more details. [3]
- Scope, is the reachability for data. How far the data is available in the application hierarchy. [3]
- Graph, consists of a set of nodes and edges that connects all the nodes. It can be directed which means that there are specific directions from one node to another. It can also be weighted which means that going from one node to another will have a certain weight, or cost if you will, and this will be attached to each edge between the nodes. [4]
- Dijkstra, Edsger Dijkstra was a computer scientist that received many awards and contributed very much to the field of computer science with both languages and algorithms that are used all over the world today. [5]
- Shortest path algorithm, is a brainchild from the aforementioned Dijkstra that is a recursive algorithm. For more information on recursion see recursion. The algorithm will iterate through all possible paths in a graph and one could then estimate the weight for each path to choose the shortest of all the available paths from point A to point B. [6]
- Recursion, means a function calling itself in a chain until it hits a base case and returns the result to the call before itself in the chain. "To understand recursion you must first understand recursion" [4]
- Concurrency, is the possibility to run something sequentially or in parallel. [7]
- Thread pool, is a pattern that spits a task into pieces and give these to threads that are organized in a queue for execution in a concurrent manner to improve system performance on multi threaded systems. [7]
- Producer/Consumer, is a well known concurrency problem that is solved with synchronization so that the consumer and consumer doesn't wait indefinitely on each other on some resource that they share between each other. [7]
- Deadlock, a situation where two or more parties want to use the same exclusive resource at the exact same time leading to the parties waiting indefinitely for each other. [7]

2 System design

2.1 Overview

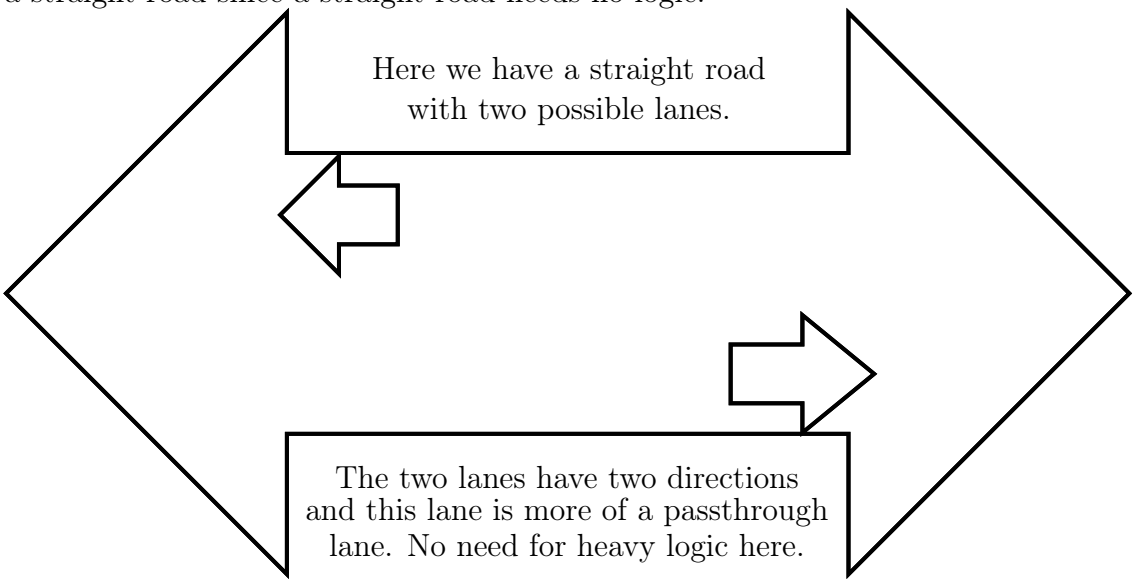
The system is designed following SOLID and a modified MVC model.

2.1.1 The model functionality

The model's functionality is exposed through the IWorldMap interface that is the main controlling part of the model. The worldmap class will act as a handler for the components in the rest of the model and will have lists of these if necessary. The WorldMap will have a list of entities. Entities are all the objects in the actual world that will later be painted and displayed in the GUI. All Entities have a set of environment variables like the weather and time that all entities must obey and these are static so once updated all entities will have the updated variables no matter where in the inheritance hierarchy the object's class is. TrafficEntities has the basic functionality of all entities, Vehicles inherits from TrafficEntities and in an alpha state there are cars and trucks that are specific type of vehicles. No methods except abstract are possible to override to make sure that all cars are true subtypes of a trafficEntity which satisfies Liskov's substitution principle and by that the open/closed principle.

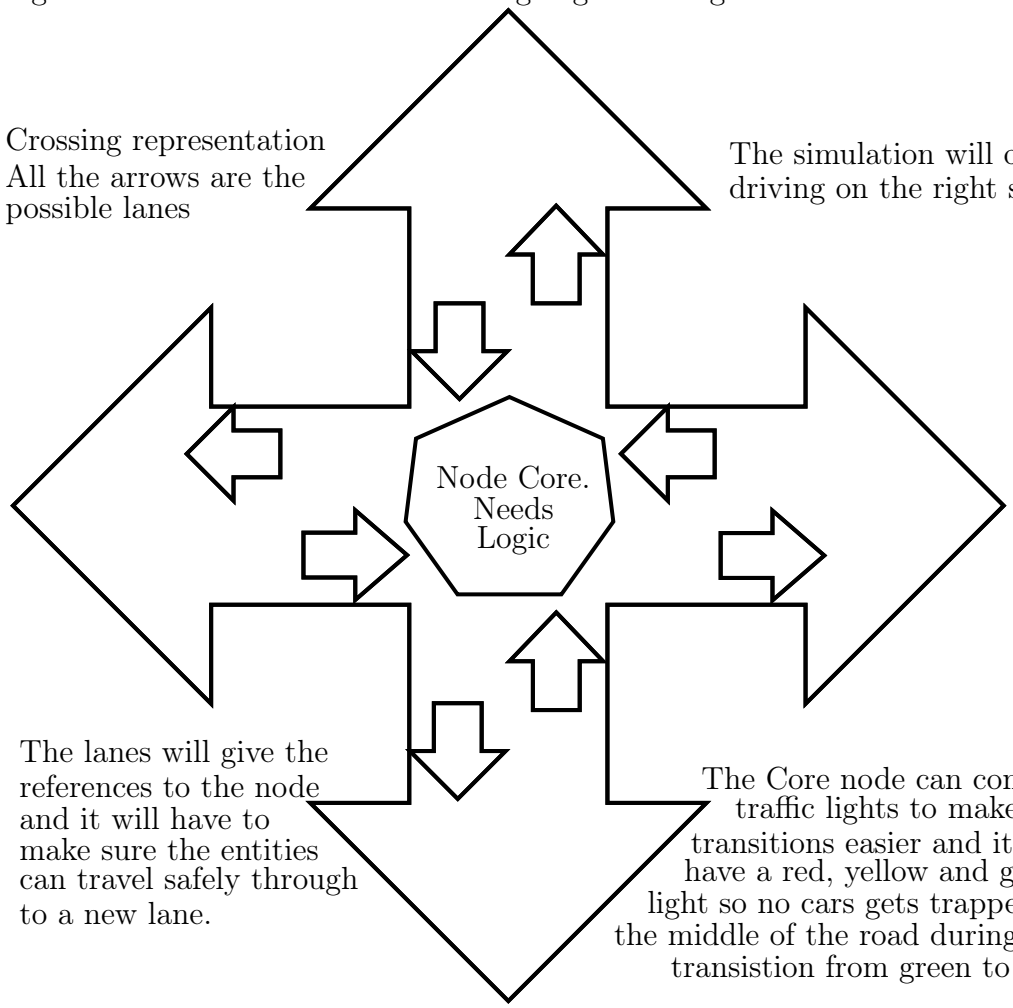
The worldMap contains lists of trafficEntities as well as lists of lanes where lanes are the roads or queues that will be filled with cars during a simulation. The worldMap will hold the entire world which means that it will have to hold the graph that represents the road network currently on the worldMap. This network is represented with nodes and lanes where the lanes are the edges in the graph. The graph is weighted and directed where every lane has a weight associated with it depending on both how many cars there are currently in that lane as well as the length of that lane. This makes it natural that the responsibility for calculating the shortest path for each car should fall on the worldMap. The cars will at the spawning or beginning of their journey calculate/search for the currently shortest path. It will not work in a GPS like fashion in an alpha state.

Nodes have a certain amount of lanes associated with it. The easiest "node" is the node with only four lanes. A lane is just the direction that the car can travel in. We do not create nodes for a straight road since a straight road needs no logic.



It is just a straight road you drive forwards on until you meet a curve or intersection. The easiest node in our network will be to represent a curve. We currently only support roads with lanes in fixed places which means that all curves are 90 degree "curves" and we need some minor

logic here to handle the transition from one traveling direction to the other at such a curve. With an intersection which is also a node there will be 8 different possible lanes that goes to and from that node. Two lanes in each cardinal direction, south, north, west and east from the node's perspective. It is important to note that the perspective of the node does not have to be on par with the perspective of the worldMap in this case since the node or intersection could plausibly be skewed in some direction to a certain amount of degrees but still consider some direction from its perspective to be north and another south. To be completely frank the node is completely unaware of these cardinal directions in the first place and it is only concerned with moving one car from one lane to another going in the right direction without any collisions.



The cars are pretty much only state holders with a very specific think method for its type. The cars only live inside either a lane or for a brief time within a node. So the lanes will have in its queue all the references to car objects that are currently traveling through and when the car is at the end of the lane, the lane will have to ask its neighboring node if there is room for another car over there and if the response is positive that means there's still room for at least one more car and the reference will be passed on to the neighboring lane that will transport the reference to the other node it is connected with.

The GUI is completely unaware of what it gets from the model and puts in the buffer to be drawn. This is achieved by having a data type Drawable that is a drawable representation of

all the components in the model. It is a class that all components in the model uses with the method getGraphics() that the worldmap will call on all its components that need to be sent to the controller and then to the gui drawing buffer to be rendered on the screen. The Drawable objects are immutable and the method getGraphics() always creates new objects that represent the drawable state of the components at the specific time it is called upon.

2.2 Software decomposition

2.2.1 General

The application is decomposed into 4 main packages.

- Sound: containing all the sound. Has an interface ISound to the world.
- Model: containing all the main model classes like the entity hierarchy and the worldMap. Has the interface IWorlMap to the world.
- Controller: containing the controller part in the MVC model.
- View: containing all the graphical user interface components.
- Util: containing all the utilities used by multiple levels in the MVC.

The scope of the encapsulated classes in the Model package are mostly protected, in other words package private, except for the WorldMap class. To reach any part of the Model one has to go through the WorldMap class that pretty much holds "the state of the world".

2.2.2 Design patterns

The Strategy pattern is used in this application to increase modularity and the ability do dynamically change the algorithm that calculates the risk taking of a car and the stress levels. This pattern makes it possible to have multiple different implementations of the algorithm behind a common behavior interface. The stress and risk assessment both uses the strategy pattern.

The Observer pattern is used in many places of the application. This pattern is heavily used in the GUI and goes under the name listeners there. For example behind the scenes the buttonListener's implementation is based on the observer pattern. The Map and car uses a custom observer pattern where the cars needs to notify the Map if it has crashed and the map needs to somehow kill the cars.

The Decorator pattern is used in the FileInputReader and it is also used in the Risk assessment of the application. The application has the ability to implement an interface to create base algorithms for risk assessment and then special extensions can be created by creating a class that extends the RiskDecorator to add further evaluation to a base calculation in runtime. This makes it possible to have one simple base algorithm that can be extended with a combination of other assessments in runtime like for example adding a drunk status to an Average driver or make the driver sleepy by wrapping the Average driver object in a Sleepy.

Singleton pattern is used to basically make sure that there is only for example one worldMap

during a simulation. A decision was made to make worldMap a singleton and add a special method called getInstance that will always return the same instance of the worldMap during a simulation. To clean the worldMap between simulations a special destructor is present that will set the variable holding the instance to null so next time the getInstance method is called it will initialize a new worldMap once and then repeat the aforementioned pattern.

Immutable objects are used on many places to make thread handling easier since immutable objects are simply objects without methods that can can change the state once an object has been instantiated. Such objects are inherently thread safe. For example all objects of the Drawable class are immutable and we have a static inner class named Event in the GuiDisplay class.

Facade pattern is used in the GUI to provide an easy to use interface while at the same time still having the option to call each underlying method explicitly without using the facading/simplifying interface.

Thread confinement is in the main loop of the GuiDisplay class. Events are stored in a LinkedBlockingQueue that will later be handled by the main thread. Thread safety is delegated to LinkedBlockingQueue.

Builder pattern is used in the DrawableObject class to eliminate the need for multiple different constructors. It receives each initialization parameter step by step and then returns the resulting constructed object at once.

2.2.3 Dijkstra's shortest path algorithm

Dijkstra's shortest path algorithm is used to find the shortest path in a vast graph from a single source node by going through all paths and comparing the combined weight between the paths. It was developed by Edsger Dijkstra and is the perfect algorithm for our needs and intended behavior. For more information see [6].

Our current implementation uses a priority queue as the only optimization but even if this is the only optimization we currently have this works very well and we have made tests with over 8000 nodes with more than 10000 paths.

A downside with this algorithm is that it in its current state only counts the weight on the edges in the graph while in a traffic simulator it might be neccessary to account for the weight of evey intersection(node) as well as just the weight on the edges between the nodes. There are papers published explaining the aforementioned and also how a possible improvement can be made to the algorithm to make it possible to execute the algorithm with multiple threads making it a lot more effective on multi threaded machines. This however will not be of priority during the time space of the course but can be of interest in future experiments after the final evaluation for the course.

2.2.4 Map generation algorithm

The map generation algorithm was made by the team during a brainstorm at a meeting. It uses a matrix to represent the worldMap like a grid pattern. This matrix is determined by width, height and a density variables. For example with the values d = 10, w = 50, h = 60 the resulting matrix will be five times six element. This will naturally be represented by a two dimensional

array. Since the map needs to be random a random starting point has to be created. The matrix will hold specially created objects named NElement that consists of a list of NElements that it is currently connected to. Remember that a node have four possible connection points and this is what this list keeps track of. The object also holds x and y coordinates as well as a content variable that will specify the current contents on a specific position in the matrix. This can be either empty, a road or a node.

At the initialization the algorithm will fill the matrix with NElement with empty contents. Then it will step through the two dimensional array and it has to do this in a controlled manner while at the same producing a random value and doing a calculation of the chance of connecting another road to a certain node based on the already connected roads. So for example the chance of adding another road to an intersection is 0 since it cannot hold any more roads than that. Then when a node gets a new road it has for each step calculate what direction to go taking into account that there may be roads just beside it and then determine the appropriate next step. If there is no free positions adjacent or if the algorithm encounters a road while stepping it will have to take one step back and create a node there or if it is possible create a node on the position that it encountered the road. If there is nothing in its way when it is stepping it will travel the whole calculated distance and then place a node there and repeat the process from the newly created node until it reaches the maximum allowed iterations that was specified.

2.2.5 Decomposition into subsystems

There are no subsystems in this application.

2.2.6 Layering Dependency analysis

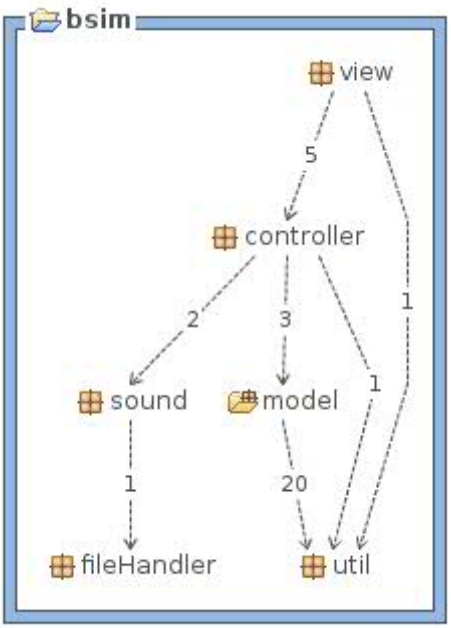


Figure 1: Stan layering and dependency analysis

2.3 Concurrency issues

There is a lot of concurrency in this application. The application's main thread will be running the Graphical user interface. One thread will run the simulation and one thread run and control the sound in the application. There will be one thread for event handling in the GUI.

We would have liked to extend to even more concurrency so the Dijkstra algorithm could run concurrently and in other words a lot faster and more effective on multi threaded systems.

References

[1] *Open/Closed Principle.* URL: `http://www.objectmentor.com/resources/articles/ocp.pdf`.

[2] *Liskov Substitution Principle.* URL: `http://www.objectmentor.com/resources/articles/lsp.pdf`.

[3] K. S. Eric Freeman Elisabeth Freeman and B. Bates. *Head First Design Patterns.* O'Reilly Media, 2004.

[4] R. L. R. Thomas H. Cormen Charles E. Leiserson and C. Stein. *Introduction to Algorithms.* MIT Press and McGrawâHill, 2001.

[5] *About Dijsktra.* URL: `http://www-history.mcs.st-and.ac.uk/Biographies/Dijkstra.html`.

[6] E. Dijkstra. *A note on two problems in connexion with graphs.* URL: `http://www-m3.ma.tum.de/foswiki/pub/MN0506/WebHome/dijkstra.pdf`.

[7] M. Ben-Ari. *Principles of Concurrent and Distributed Programming (2nd Edition).* Addison-Wesley, 2005-11.