Team DUS - System Architecture

Approach

Our general approach to the system architecture was based on both user-facing qualities and developer-facing qualities. We determined that the user must experience a game that performs well and is reliable whilst the developer must have reliable, easily modifiable and easily understandable code. Thus, the architecture was based around meeting the following criteria.

The system architecture should allow the game to perform actions quickly. As a result, data structures should be quick to access and as efficient as possible. User interaction should have no noticeable delay and the time taken switching between players should be negligible.

The architecture must be reliable to the player and to the developer. The player should not experience any crashes and errors should be handled correctly. Most importantly, the game should not act sporadically; for example, the speed of the game's responses should not slow down the longer the user has been playing the game. It should respond in a predictable way for developers so they will get the correct data returned from functions or an error will be raised correctly.

The architecture should be capable of being extended or modified easily. This involves having modular design and highly cohesive classes and low coupling. For instance, modifying a type of engine should not have to involve editing any class outside the Engine class. This is important in making the game reusable so that further ideas can be built using this game as a core framework or classes already built can be altered to add more functionality.

Finally, the system architecture should be consistent throughout the whole game. This means that the code should follow all coding standards, all Java standards and have meaningful variable names. Where functionality is not obvious, the code should be well commented to make this clear. The architecture should also follows the principles of encapsulation.

Architectural Decisions

Early on, we decided to not use a game library as we did not want to add unnecessary complications to our project as it was felt that the game was not complex enough to warrant needing a game library as its basis. Despite this, we did explore the option of a few game libraries. The first of these was LWJGL, however this was found to be generally for using the GPU to create 3D graphics using OpenGL. As our game is strictly a 2D game, this was deemed

inappropriate for our purposes. Slick2D and libgdx were also investigated but ultimately it was decided that these too were unnecessary for the level of detail in our game.
Both would involve working through very large amounts of documentation to figure out which bespoke functions to use in different situations and why. This would mean many hours would have to be spent simply working out how to implement something which could be implemented, albeit to a lesser graphical quality, with standard Java libraries. Furthermore, using a game engine would involve creating many more classes than would otherwise be necessary (e.g. texture classes) that we decided would be too complex to be fully exploit in the given timescale. Again, this would require spending allot of time and effort learning how the library worked rather than concentrating on the quality of the back-end (e.g. data structures and general standard of code).

We therefore decided to create all graphics in the game from scratch and all windows/forms using the standard Swing GUI toolkit as part of Java. There is allot of documentation for this, varying from easy online tutorials to fully explained Oracle documentation. As several members of the team had some experience using this, we decided this would be the best approach. Ultimately, although this decision may have limited the overall capability of our program, we made this decision so that we did not have to learn specialised code for the chosen library. We felt this would not only produce in general a better game but would increase our general Java knowledge which will help us later on in the project.

Game Architecture and Interactions

The architecture is the underlying conceptual design of the game involving object classes and the relationships between these classes. As stated by Sommerville: “these classes define the objects in the system and their interactions”. Hence, we define the architecture of the game to involve the data structures (both concrete and abstract) that make up each class whereas the interactions of the game are the ways in which the objects of a class may use objects or other systems (such as reading from a file) and are linked when a class has been instantiated.

For clarity, the design decisions taken for each part of the project shall be discussed class-by-class from the bottom of the hierarchy to the top (i.e. from lowest-level classes where methods are independent of other classes to highest-level classes where methods tend to be composites of lower-level methods). This will also cover the implications of these particular decisions. References to the original analysis document shall be made where there are significant changes from the original design. In addition to this document JavaDocs have been generated for our system.

Point:

This class is used in place of the standard Java.awt.Point class. We found this to be necessary as our node locations are stored as fractions of the overall screen size to accommodate for varying screen sizes. The original Point class only supports integer x and y coordinates which

would not be suitable for storing decimals. In addition to changing the x and y attributes to be floats, a constructor was written that takes a string as an input, then converts this string to an instance of Point. This constructor was created to handle creating the node locations directly from the strings saved in the map JSON file (as discussed in the External Files section below). This was done to prevent having to write the code outside the class which converted the string into two floats and then passed these as arguments to the Point constructor. This keeps the code more concise as well as maintaining high cohesion by keeping all relevent code internal to the class which uses it. This prevents code repetition and makes updating the code that creates Points easier, if the way that they are stored changes in future updates to the game.

Node:

Node is an abstract class as in the original design. Both Junction and Station inherit the attributes and methods of Node. All three attributes (id, location and name) are final so cannot be changed from the creation of the object as the nodes for the map should never change and hence do not need to be variables. This is an added level of protection to prevent misuse of the node class. The attribute location consists of a Point object which represents the coordinates of where the Node is on the map.

Junction and Station:

These are subclasses of Node. The two separate subclasses are used to differentiate particular nodes from being stations or junctions. Due to limitations in how much can be implemented for future assessments, there is currently very little difference between Stations and Junctions. We decided to keep the two subclasses of Node in the game, as a future version of the game may only allow one train per junction but multiple trains in any station. Keeping the subclasses separate makes it easier to distinguish between the two types of nodes. This is still useful within the current iteration of the game as Goals are created by randomly selecting Stations but not Junctions. This can done by using the instanceOf comparator to check whether a Node object is a Station or Junction. Overall, although their use has been limited, the internal design of Junction, Station (and hence, Node) has not changed from the original design.

Route:

The Route class contains an ArrayList of Nodes - this allows the route to consist of both Junctions and Stations. Using an ArrayList makes it easy to add and manipulate a route or retrieve the Node object at a particular position in the list. The original design of the Route class also contained a method to determine whether this route was blocked by an obstacle. This method was removed as it was unnecessary once obstacles were removed from the game.

Goal:

This class stores all relevant data about each goal and has remained largely the same as in the original design. Every goal contains exactly 2 Node objects which refer to the start and end position of each goal.

Resource:

Resource is an interface for the classes ‘engine’ and ‘upgrade’. This enabled us to guarantee to any user of upgrade or engine that they would both contain basic name, description and use methods. The Use() method is polymorphic so can be called in the same way but is implemented differently based on whether it is called by an upgrade or an engine. e.g. the same method, Use(), can be called by any resource object regardless of whether the object is an engine or upgrade. This makes it simpler to utilise these objects and methods throughout the code. An interface was chosen, rather than an abstract class as in the original design, as no methods were being inherited from the superclass (i.e. they would always be overridden) and hence it made most sense to have the resource as an interface - a contract that certain methods and attributes would always exist in the two classes which implement it.

Engine:

This class implements the Resource interface and has remained almost identical to the original design. An enumerated type restricts the types of engines that can be set and all attributes the engines (e.g. speed) are derived from this. We decided to do this as it made programming assignment of a new engine to a train very easy and less likely to produce user errors. It was also an effective way of ensuring that the attributes of comparable engines remained constant throughout the program without needing to look up which engines required which attributes (and potentially entering the wrong data). This is in line with the principles of encapsulation and information hiding.

The Engine class contains a constructor to randomly select a type of engine, using the java.util.Random class. This is contained within the class as the probabilities of each type of engine being generated should be internal to the engine rather than external classes having to select one of the possible types. Again, this leads to highly cohesive code which is more reliable and easier to keep track of as a developer.

Upgrade:

The upgrade class largely follows the same design as the engine class - both implement the Resource interface and use an enumerated type to set their attributes. The ‘Use’ method within the upgrade class, however, must account for different types of upgrades. For example, one upgrade simply doubles the speed of a train whereas another allows the train to teleport to anywhere on its route. Furthermore, the attribute ‘reapply’, which was not in the original design, was added to stores whether each upgrade needs to be reapplied when an engine changes. e.g. if a train has a ‘double speed’ upgrade and then a new engine is added, the system checks the ‘reapply’ attribute to determine if the double speed upgrade should be reapplied after changing the engine.

Train:

The Train class links the Route, Goal, Engine and Upgrade classes. Every Train has exactly one Route, one Engine, but can have any number of Upgrades (upgrades that have been applied to the train) which are stored in an ArrayList. This ArrayList can be iterated through to check for

upgrades that must be reapplied (as described above) and then the upgrades can be placed back in the ArrayList at the index they were originally in. The train class remains mostly the same as in the original design. An attribute ‘speed’ was added as the speed of the train was dependent on both what engine has been applied and whether any upgrades had been applied. Rather than check all applied upgrades and what engine had been applied each time the speed of the train was required, it made most sense to have an attribute speed which could be recalculated each time an engine or upgrade was applied and then return this immediately on request, rather than calculate it each time. The attribute route was also added, as every train has exactly one route which it will travel through, which was omitted from the initial design. A train can have zero or one Goal. This decision was taken as it enabled us to be more flexible with Goals (this is explained further in the Player class description below) as a train is only set a Goal once it is set a Route.

Connection:

This class is used to store the distance between two nodes on the map and has largely remained the same as the one originally designed. As we have removed obstacles from the game at this stage, the connection class was not strictly necessary as we could have stored the length of the connections directly in the matrix. However, we decided that there was no real downside to keeping the class as it shall allow easier expansion of the game later on, if obstacles are to be added. Therefore the attribute ‘obstacle’ (and all associated methods) was removed from Connection, but the rest remains the same. The attribute distance is a final as the distances between two nodes will never change once they are loaded from the file (see External File section below for more detail). Thus, it make no sense for any class, internally or externally, to be able to change this value.

Map:

The map class links the node objects (from the subclasses junction and station) with connection objects as a graph stored in the form of a list of nodes and an adjacency matrix of connections. It contains 2 ArrayLists and an 2D array of connections. The ArrayLists consist of the list of filenames where the the various map nodes are stored and a list of nodes (of class Node) on the map. This is populated by reading from a randomly selected file from the list of filenames. The format of this file and the way it is read into the system is discussed under the title ‘external files’ below.

The 2D array of connections acts as an adjacency matrix as this was the simplest way of storing the graph structure we are using to implement the locations of the game and the tracks between them. The indices of the 2D array correspond with the IDs of each node in the ArrayList of nodes. Hence to check if there is a connection between two nodes, the ID of each node is found and the element in the 2D array corresponding these indices is checked for a connection object. Using an adjacency list would be much less efficient for checking if connections exist in this way.

Example of connections matrix:

	0	1	2
0	null	2	3
1	2	null	4
2	3	4	null

This represents the connection between node 1 and 0 being 2, node 2 and 0 being 3 and node 1 and 2 being 4. As can be seen in the matrix, it is mirrored about the leading diagonal which is null as a node cannot have a track that leads to itself. It is mirrored so that when programming the game, the developer does not need to work out which node IDs should be the first and second indices of the array to look at - the correct distance will be returned no matter which order these IDs are given.

Player:

The Player class remains very close to its initial design. The string attribute ‘name’ is final as the player’s name should not change throughout the game. The inventory of upgrades and engines are stored in two separate ArrayLists, for ease of manipulation. Keeping them separate means that each ArrayList can be managed independently; whilst using ArrayLists allows us to store the actual Upgrade or Engine objects that a player has rather than a list of names that must then be cross referenced to objects when they come to be used. There is also an ArrayList of Trains and Goals. We decided to separate the Player’s goal from the Goal attribute of each train as this enabled us to manage goals before they were assigned to trains. This meant we could implement rejecting unwanted goals if the player already has 3 goals which would not be possible if there was a 1:3 relationship between Player and Train and a 1:1 relationship between Train and Goal. Thus, the Player class has between 0 and 4 Upgrades, between 0 and 3 Engines, between 1 and 3 Trains and between 0 and 3 Goals.

Game:

This is the main class that runs the program. This class contains all variables which need to be read from and manipulated based on the state of the game. The attributes and methods of Game are static so that they can be accessed outside of the class. We chose to do this as it makes programming the game far easier as there are no issues with accessing what we need from the class in any part of the program. For example, the Game class creates a GUI object but the GUI must respond to clicking the ‘End Turn’ button with a call to a function that ends the current player’s turn; if this method was not static it would be far more complicated to program and organise an architectural structure.

The game class has a 1:1 relationship with the map. This was done because Map is a class which contains the data structures of the map layout but the game class contains the methods and attributes regarding the state of the game. These were kept separate as it enabled us to avoid having a ‘god class’ to try to maintain high cohesion and low coupling. In a sense, the

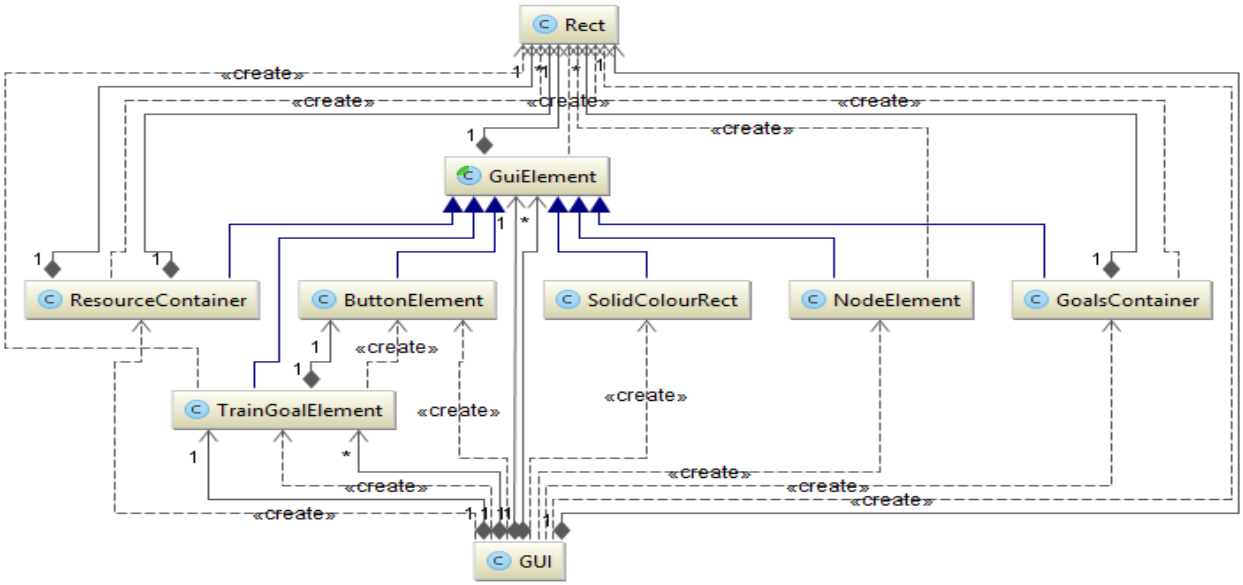
Game class ties together all of the back-end of the program by running the appropriate methods at the beginning and end of each turn without having a single class that contains all of the code for each object.

The game class simply serves as the overall controller of the game: managing which player is currently active, the state of the map and how many turns have been passed. This keeps these attributes and methods contained to one class to make it easier to use as a framework to build the game on top of this class.

The static method ‘main’ (i.e. when the game is run) creates a Map object and a new GUI object which then controls the user interaction with the rest of the game through the Game class. The architecture of the GUI package is described below.

GUI Package and Classes:

The GUI Package contains all of the classes which manage the user interface. The GUI was designed from scratch rather than using a library as described above. The main GUI creates objects from the subclasses of class GUIElement. These act as both the graphical images on screen as well as containing the code for user interaction with each object. As discussed at the start of this document, the GUI was designed from the ground up to be simple rather than to use a complex game library. In keeping with this, the GUI classes then call functions from the back-end classes that enable the functionality of the game. The relationships between the classes of the GUI package are shown below.



Omitted Classes:

Several classes were omitted that were originally intended to be implemented but were no longer necessary as a result of the restrictions imposed as part of the assessment. These included obstacle class and the bonus class (i.e. quantifiable goals for extra points) as it was

stated that the game must not exceed the brief for assessment 2. Removing these classes does not affect any other part of the game, it simply means that other classes linked to them no longer need to have specific attributes to refer to them.

An updated UML diagram for the System Architecture is contained on page 10.

External Files

The attribute listOfNodes in the Map class is populated by reading an external file. The nodes are stored as a JSON file because it is a very flexible file format. JSON is schemaless meaning that the structure of the file is very easy to change and update as the game evolves. In addition, the way that JSON stores all of the attributes of the file is very similar to how objects are stored in Java, making the loading of the file almost a one-to-one mapping with the Map class.

To read in the file, we use the java.io.FileReader library and then parse it using an external library called JSON-simple 1.1. This is due to the fact that the current versions of Java do not natively support JSON parsing and manipulation, so an external library was necessary to import meaningful data into the system from the file.

Below is the format of the JSON used to store the map:
{"goals":[<Array of goals stored as {start:<start-node>,end:<end-node>,points:<points>}>], "nodes":[<Array of nodes stored as {location:<Point stored as {(x,y)>,name:<Name of node>, type:<Junction or Station>}>], "connections":[[<Stored as an array of arrays which contain the distances between each node>]]}

Nodes with no connection are stored as null, as in the main program.

The JSON file storing the map is generated using a custom map creation tool that was created during development of the game, but could also be written manually using the format provided above.

Missing Requirements of TaxE Game

The game meets most of the requirements set out in Assessment 1. All but three of the high priority requirements were met. One of these requirements (High Priority Req 12) was that the “user shall be able to see how much time is left of their turn”. It was decided that setting up timing was going to be very difficult and would hinder the ability of the team to deliver other parts of the game to a high standard if so much time was spent on setting up timing. Similarly, the requirement (HP Req 8) to “see the total distance of their selected route and how long it will take (in terms of turns and game time)” was deemed to be less crucial than other requirements to the development of the project and final gameplay overall so was not implemented due to approaching deadlines.

Another requirement (HP Req 11) involving being able “to complete goals in order to earn points, when the user has enough points they win the game” involves scoring, which is to be implemented at a later stage; however, without a system of scoring, it is impossible to play a meaningful game (i.e if there is no scoring, how can there be a winner?). Therefore we interpreted this limitation on the game to mean that a complex system of scoring should not be implemented and instead we shall just declare whichever player has completed the most goals in a set number of turns the winner. This allows the game to still have meaningful gameplay and outcome, without drastically exceeding the brief that we have been given.

Three of the medium priority requirements (MP Req 3, 4, 6) were not met as they involve obstacles which are part of the deliverable product of Assessment 3, so could not be implemented yet.

Three low priority requirements were also not met (LP Req 2, 3, 4). The requirements to allow the user to “crash a train into an enemy train” and “freeze a train” were not implemented as they were considered too complicated to build into the game and it would be a more efficient use of time to build a simpler program which these features could potentially be added to later. The other requirement to “cancel a route” was not implemented as a result of the time constraints of the project. These three requirements, in particular, would require significant amounts of testings due to their complexity so it was decided that it was best not to implement these at this stage.

The game sticks to the majority of the brief for the game as laid out in the scenario section of the assessment document. The game fully meets the requirements of what users should be able to do each turn; however, not all of the others have been met.

The game does “support 10 different goals” - these are randomly generated by randomly selecting two Station object from the map. The two selected Stations are then checked to make sure they are not the same and are then presented as a goal to the user.

The requirement to “support at least 10 different resources” was not fully met as most of our ‘upgrade’ resources were based on obstacles and hence, could not be implemented as stated above. Nevertheless, these can be added to the Upgrade class relatively easily by adding new possible values for the enumerated type and by adding a new case to the switch() statement to code how they should affect the game - all within the Upgrade class. Engines are similarly simple to expand by adding new values to the enumerated type as part of the Engine class. Therefore the system can easily “support” 10 different resources though the current total falls short of this. With all engines and upgrades there are 7 resources; including future upgrades based on obstacles (adding, removing and moving obstacles) there will be at least 10.

The system has met all other user requirements to the best of our ability, as demonstrated in the validation section of the testing document.