Architecture and Design Report

Software Engineering Project (SEPR)

Team “EEP”

Richard Cosgrove, Yindi Dong, Alfio E. Fresta, Andy Grierson, Peter Lippitt, Stefan Kokov

Department of Computer Science

University of York

The full UML class diagram for the architecture of the Game can be found in the Appendix. To facilitate understanding of the Game architecture, small excerpts of the class diagram are referred in the following paragraphs of this document. The architecture is similar to that of the proposed architecture in the first assessment, however, updates have been made and are stated.

Game and Players

The two main entities for a two-player game are the “Game” and the “Player”. The Game class represents the game being played by two Players which will hold all the information about the progress of the game, such as the turn being played, the Player objects, and the Game Map. Following the Network architecture we adopted, the Game class is also used as the main data container for the network exchange between the Players. The Player class represents the two players of the Game which includes the properties of the Player’s nickname, their score and the number of objectives accomplished by the Player.

Update: No significant update to this section of the architecture.

Justification: A master “Game” class is needed as a unique point of access to all of the Game information and facilitates the collection of data exchanged in the network. Each Player is be able to edit its own Player object by specifying their nickname or, more generally, playing the game (e.g. by buying a resource, completing a goal, positioning a train, and so on). The design decision was discussed thoroughly amongst the team prior to implementation and was suggested as the best approach by the lead programmers because it would be more efficient to use, implement and maintain by creating one location where all data is exchanged. This was also less complicated for the development team to understand, meaning very little, if any, training time would be required.

Related Requirements: FR no. 1.2, 1.3 and 1.4.

Resources, Inventory and Store

We classified Resource as any type of Spendable or Usable resource within the game. These classes are used to distinguish between which Resources are used by the player on the game board (Usable) and which Resources are used to purchase additional Resources (Spendable) in the Store. All Resources have a short name and a description associated with them. The two Spendable Resources Gold and Metal are essential elements of the game that can be used by the player and are gained every turn (to comply with the clientele requirements, see requirements specification document). They are used to buy other Usable resources when the player can afford them in the Store to add a unique strategic method to the game that gives the Player the freedom to choose which Resources they want, rather than being given any at Random.

All Usable Resources have a price property that is expressed in quantities of Gold and Metal, and are usable on Trains of both players. For example, a Train Speed Modifier is a Resource that specifies a Speed Factor to be applied on a train. Examples of Modifiers could either be a speed boost for a Player’s own Train, or any Impeder that slows down an opponent’s train. All Usable Resources are placed into the Inventory of the Player to track which Usable Resources they can use. Once a Usable Resource is used, it is removed from the Player in question’s Inventory to stop them from repeatedly using the resource and comply with the clientele requirements stated previously in the requirements specification.

The Store class represents the in-game shop that contains a collection of Usable Resources available for the Players to buy. The role here is to give the Player a choice in which Usable Resources they wish to purchase by using the Spendable Resources. It adds a strategic element to the game that Players may enjoy more, rather than relying on pot luck to get the Resource they desire most. The Store class is not required, however, it is used to give a purpose behind the Spendable Resources that meets the requirement of having two resources gained per turn. The Store is technically a singleton class, meaning that there only is one store in each game and it should never be instantiated more than once. It can be used to contain all of the Usable resources, including Trains, that the players can buy in the Game.

As the “Inventory” of a Player we define the set of all the Resources currently owned by the Player - both those that are currently being used and those that are not. The purpose here is to provide the Player with the information of which Resources they have on them, their spare resource slots and those that are in-play. This will provide clarity for the Players when playing the game, which will help keep them engaged in the game more. This will be linked with the GUI as a visual aid to the Player to easily identify the previous reasons.

Justification: This Architecture allows us to have some common properties for objects that are very different (e.g. Gold and a Speed Upgrade), which will help improve the readability and memory efficiency of the code. The Store class is very helpful to keep a reference of all the available Resources that can be bought in the Game and is needed as part of the Network architecture to share, between the players, the Game Store and the list of Resources that can be bought.

Update: Whilst the Store class has been developed and would be ready for production, because of time constraints we decided not to include a Store in the User interface at this point. Instead, the Player receives each turn up to two random Usable resources - instead of two Spendable resources that could be used in the Store to buy some Usable. This means it would still comply with the essential requirements stated by the clientele at this development stage, however, the clientele will not be as happy due to not implementing a feature that was recommended. A future development team may wish to create the Store class to improve the clientele’s satisfaction and would help provide an additional selling point of the game to encourage users to play.

Related Requirements: FR no. 4.1, 4.2, 4.3 and 4.4.

Regions, Vertices, Junctions and Stations

The Game’s map is be internally represented as an undirected weighted graph, a collection of Vertices and Edges interconnecting them. A Vertex in the map can either be a Station or a simple Junction - in the case of the first, the node will also have a recognisable Station name (e.g. “London King’s Cross”). In the game, in fact, Vertex is an abstract class that has a list of edges and a pair of coordinates used to draw the Vertex on the map. The Vertex class is inherited both by the Station and Junction classes.

An Edge is an arc between two vertices and has a variable length that represents the distance between the two vertices. The length of the path is used to calculate the progress of the train on the edge at the end of each turn, depending on the speed of the train. This length can be used to calculate the shortest path between any two vertices in the graph whenever the Player decides to move a train. In the future, this could be used to calculate the score of each goal.

In the game architecture proposal, we wrote about implementing Regions to categorise Stations and Random Obstacle Events. As the latter are not required at this stage, and because of the time constraints imposed by the assessment deadline, we decided to leave it for the next team to eventually implement. When implemented, it would add an extra dynamic to the game’s experience through additional interactivity and can be considered as a selling point of the game for the clientele. Both reasons may encourage users to play the game.

Route finding

It is possible to find the route between any two vertices in the map using a shortest path finding method. The method implements a version of Dijkstra's Algorithm that has been optimised with the use of priority queues. Dijkstra's algorithm has been chosen for its efficiency, as the method is supposed to be called very often in the game.

Whilst the method is implemented and has been tested - unit tests for the shortest path algorithm are provided as part of the game code - as part of the GUI design process we decided not to allow users to click on any two vertices on the map and let the game calculate the path, as this would have strongly affected the freedom of the gameplay. Instead, we decided to allow the user to build a Journey edge by edge, and use the shortest path algorithm to calculate what would be the shortest path between the starting and ending point. The length ratio between the Player's chosen Journey and the optimal is subsequently used to compute the score awarded to the Player if the journey accomplishes a goal.

It is important to notice that - even if the game code is already capable of calculating scores and rewarding a player that accomplishes a goal, the function to increment the score has been deactivated as it is outside the scope of this assessment. It is left to the team that will continue the development of this game to activate and fine tune the formula for the calculation of the scores.

Update: The Region model will not be implemented for the assessment as it was not an essential requirement that needed to be included. The development team did not have enough time to implement the Region model and would have been a complicated task to complete based on our coding experience, taking up too much time to learn and implement. The Random Obstacle Events will not, consequently, be part of the game at this point in time. Therefore there will be no relationship between a Vertex on the map graph and a Region, nor between a Region and Random Obstacle Events.

Justification: The team decided that nature of the railway network could not be implemented in a more suitable structure than a graph. It was the only method the team thought of in how to implement it and team members already knew how it could have been programmed, which meant no extra training would be required that could be distributed elsewhere along with the team's resources. The graph would also be easier to read in the code than other methods as it could be visually understood by all developers with diagrams if required. The requirement by the clientele was that a railway network would be included in the game and did not state how it would be implemented, as long as it was efficient. For simplicity, we decided to avoid including route directions, and instead decided that an undirected graph would best fit our needs due to the unnecessary complexity involved. It also adds an element of realism to the game as generally there are two train tracks to enable trains to go in both directions to and from a city and it may discourage users from playing the game. The idea could be implemented in the next stage as an obstacle to add an extra dynamic to the game.

Related Requirements: FR no. 2.3, 2.5, 2.6 and 2.8.

Trains, Journeys and Goals

Goals represent the objectives that the Player needs to achieve in order to be awarded points, advance stage (and ages as a consequence) and finish the game. Each Goal has properties such as a general name and description and a starting and an ending Station that a Player's train needs to travel between to achieve the Goal. The Goal class also has a method used to calculate the reward when it is accomplished by a player. This method can make use of factor such as the optimal journey (calculated using Dijkstra's shortest path algorithm) and the actual journey chosen by the player, as well as other methods to check for accomplishment. The reward calculation has deliberately left approximate, as the scoring is not required at this stage, and ready for the next team to implement.

A Journey represents the current position, if any, of a Train on the map, and its position in the path. It is represented by a set of three elements: a Path, that is defined as an ordered list of the Edges of the whole journey, a number that indicates the current route of the list and a Route Completion figure that specifies at which point of the current route the train is positioned.

Specifically, a Journey is a child class of a Path. A path extends an ordered list of Edges, as this unequivocally determines a directed path between two vertices. Some other methods, such as the possibility to build a path vertex by vertex instead of using edges has been added to facilitate the integration with the UI (as the player clicks vertex by vertex when setting a train's journey).

As an Edge of our unweighted graph has no intrinsic direction, adding by vertices also helps by eliminating the ambiguity about the direction when the path is composed by only one edge. In all other cases, the direction of an Edge in a path is extracted in the context of the following and preceding Edges (e.g. the ending vertex of an edge is the one in common with the following edge in the path). All of these methods are implemented in the Path class.

The Train class is used to represent a single Player's train. A train is strictly related to an Age (e.g. can either be a Steam Train or a Electric Train), some properties such as a Make and a Model, a Base Speed (that can be altered using an Upgrade) and, when it is placed on the map (as opposed to being idle in the player's Trains Depot), it also has a Journey object representing the current position and route, or null if the train is in the Player's trains inventory.

At this time, whilst the game is potentially able to display and coordinate multiple trains on the map, the ability to acquire trains has not been implemented - so the Players are limited to one train that is generated and placed for them on the map at the beginning of a game. This is left to the next team to improve.

Update: A new class named Journey has been introduced to replace the class that in the architecture proposal was called "Status".

Justification: The game requires Goals to be able to progress in the game and win and are part of the clientele requirements at this development stage. These must therefore be implemented by the development without delay and will be prioritised when coding the game, otherwise the clientele will not be happy when the system is handed in for the deadline. The Goals are set up in this manner to link directly with the graph and pull in the required information for the Goal to be successfully completed. The attributes are defined to easily output the necessary information to the user through the GUI, helping the development team and those of the future to understand and maintain our code in an efficient manner (e.g. not needing to get assistance from our development team).

Related Requirements: FR no. 2.3, 2.5, 2.8 and 2.10.

Ages and Environment Obstacles

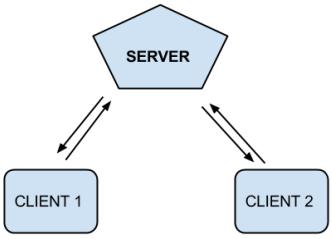
The Age class represents an Age of the Game. An Age is related to a number of Random Obstacle Events that can occur throughout the Game while the Player is at that particular Age, the Usable Resources that can be acquired and a set of Trains built during that Age (e.g. Steam Trains for the Steam Age). To facilitate integration and categorisation of objects for different Ages, we introduced a specific "Ages" enumerator type in the game code. An Age object can therefore be instantiated given an enumerator.

Update: The Random Obstacle Events as designed in the architecture proposal are not a part of the game at this point in time due to not being one of the requirements at this development stage. The relative class has therefore not been defined and can be implemented by a future team if they wish. Ages are still a fundamental part of the game and are used in many other circumstances such as the categorisation of Trains and Usable Resources to provide different stats based on their Age.

Justification: The whole Game is based upon the concept of having Ages succeeding and allowing the user to progress through the past and an ipotetic future. The idea is to have an object that represents any playable Age to classify a number of other entities in the game, such as Trains as well as Random Obstacle Events that could compose the “story” behind a Game. From a developer’s perspective, this would increase the readability and maintainability of the code, making easier for the programmers to understand and adapt the code. For a user, this would add an interesting gameplay experience as it would provide more resources as the game progresses, so that users are not overwhelmed at the beginning of the game,and gradually increase the pace of the game for a better experience. Ages were also suggested in our prototype to the clientele and were recommended to be implemented.

Related Requirements: FR no. 2.7.

Network Architecture



We adopted a very simple Client-Server Network architecture to allow Players to compete against each other in two-players Games. We also tried to keep the Server architecture as simple as possible to avoid having excess complexity in the Project.

The reason for choosing a Network solution is to allow the players to interact without necessarily being in the same room and to provide more playing comfort by removing the need for players to switch seats at the end of each turn. This would increase the potential audience range for the clientele’s game as players from all over the world can connect and play each other. Moreover, we decided on a Client-Server-Client infrastructure to remove the need for the User to carry out any network configuration (e.g. setting up port forwarding for playing via the Internet, or remember a long IP to connect to an opponent). In fact, assuming the Server is always active and reachable via a known host address, the Players just need an Internet connection. After establishing the communication with the Server, the latter will pair the Players and forward any game data a Player sends to the other one.

The reason for the development team choosing a Network solution was because the team knew how to implement the architecture and had shown the idea to the clientele who wanted it over a local solution. The problem was considering the impact it may have on future development teams when maintaining the system, however, all the architecture has been created and thoroughly tested. It does not need to be changed and can be called in the code when required.

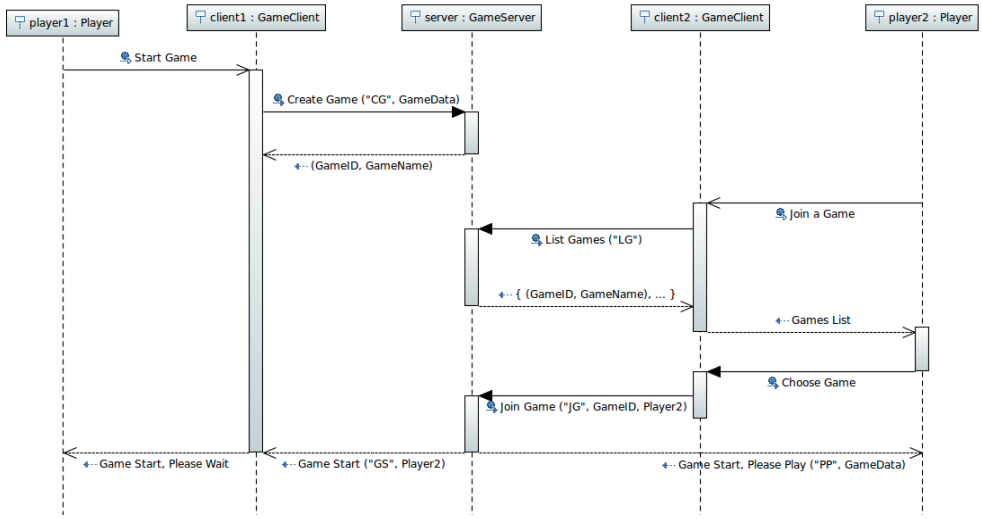
Master/Slave Network

We wanted to keep the Server architecture very simple and, more importantly, to keep the Server independent from the actual Game logic. The reason for this decision is to allow us and other future development teams to easily extend the game without having to modify the Server. To allow this, we chose a Master-Slave network architecture. In our architecture, in fact, the server’s only duty is to facilitate data exchange between the two players. All of the Game computation, such as creating the Game’s initial environment, calculate score, etc., is done by one of the two Game clients that will be called the “Master” Client.

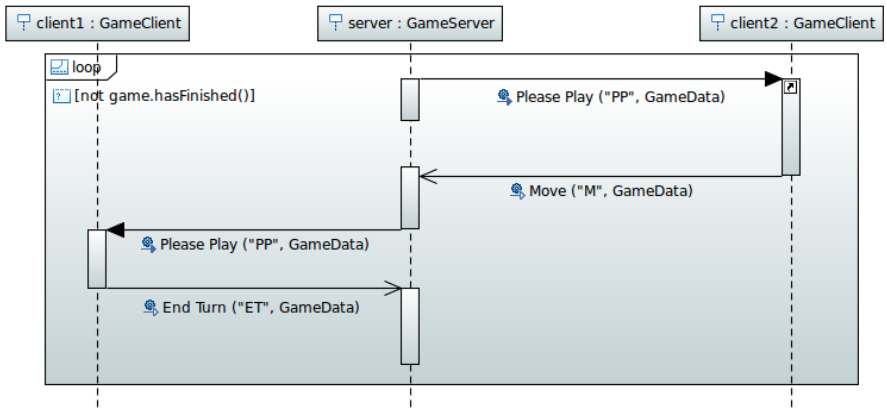
When the game is started, the Client of the player who creates the game is designated as the Master Client and the other player who joins the game is assigned as the Slave Client. The Slave will always play the first half of each turn, sending the move data to the Master Client through the server. This was because it was easier for our team to program the turns and it was not stated which player had to start. The Master Client will wait for the local Player to move and compute the end of turn scores, assign resources, environment obstacles and so on, where it will finally communicate to the other player the computed game data. This process will then repeat until the game will eventually finish.

Apart from the mere data exchange facility, the server also implements a very simple lobby system, allowing Players to create games, list current “open” games and join a game - effectively pairing players to each other. This was to enable Players to have the freedom to choose their opponent and identify quickly which games are available to join. It was a necessity when implementing the Network architecture, following in suit of many other multiplayer games and was recommended by the clientele when shown in the prototype. The commands and payloads are explained in the diagram.

The following UML Sequence Diagram describes the interactions between the Players, their respective Game Clients and the Server while Starting a game:



Successively, while the game is in progress, the process described in the following UML Sequence Diagram is repeated, until the Game is finished.



Another important role of the server, is to check for constant network connectivity between the two players. In fact, if anyone of the two players goes offline, the server recognises that and immediately notifies the other player. At this time, when this happens, the other player is displayed a message explaining that their opponent has disconnected and that the game is therefore going to be terminated. It is possible for the next team to extend this simplistic lobby system with saving and loading functionalities (e.g. allow players to interrupt and continue a game at a later date). We did not implement this feature ourselves due to time constraints and not being an essential requirement at this development stage. It is also a complex system that would need additional time and training to complete.

Network Technology

While in the process of choosing the Network technology to adopt for the Game, we thought of the following requirements we need it to satisfy:

- We needed a Network technology that is simple for the Player to set up (does not need any configuration or have any particular requisite apart from a working internet connection). This would allow any player to play the game without the need of knowing any technical knowledge to stop them being discouraged from playing the game.;
- We needed a Network technology that allows real time communication. This is because Players play in turns and expect the system to be ready right after the opponent plays, otherwise the pace of the game would be reduced and may bore users. Examples could be TCP and UDP sockets, as opposed to *pull-type* communication over HTTP requests;
- We needed a Network technology that takes care of data loss, data integrity check and retransmission, as we don't have enough resources, time and skill to build it ourselves; this narrows the possible choices to those based on the TCP protocol;
- We needed a Network technology that is able to check the health of the connection, quickly detect disconnection, attempt to reconnect automatically in case of a temporary network failure, and when reconnection is not possible (e.g. when a client goes offline, or wifi is disconnected for a long time), promptly notify the server and the other player of the event;

After considering a couple options, we finally chose Socket.io[1] as our network infrastructure framework. Socket.io is an open source project[2], built on top of various technologies such as TCP sockets, that allowed us to create a very simple Server application written with NodeJS (an interpreter for the JavaScript language), while taking away all of the complexity that generally comes with a network application: the underlying framework takes care of all low-level duties such as keeping connections alive, correcting data-transmission errors and catching disconnection events. Moreover, the JavaScript programming language is very easy to read and - if ever any modification would be necessary - it would not be as complicated or hard for the next team to do it.

Many widely-used and tested implementations for Socket.io already exist for the language of our Game, Java[3]. Being very famous in the open source community, it also comes with excellent community support and, hence, would lower the risk for us through the adoption of an external library. In fact, during the development of the Game, we encountered a bug [4] in the Java's Socket.io implementation, but given the open source nature of the project, we were able to fix the bug and request the fix to be integrated into the official repository.

Update: No significant update to the network architecture of the Game. Socket.io has been used.

Development Patterns

Model-View-Controller Pattern

We chose to adopt a MVC software architecture pattern to develop the Game as the architecture is aimed at user interfaces, which is what is required by the clientele. This pattern is widely used and well supported in the Java programming community and has a number of advantages over other methods.

Firstly, the architecture pattern is used in industry. It enforces a good programming style and well defined separation between the data (models), the game logic (in the controllers) and the user interface (the view).This improves the readability of the system, making it simpler to understand the underlying architecture for future developers to maintain.

Secondly, it makes it easier for different team members to work on different aspects of the same part of the game, having for example someone to develop the user interface and, at the same time, enabling someone else to work on the logic for the same piece of software. By having a working environment that can be worked on

simultaneously, teams can distribute resources and time effectively to ensure they meet the time constraints of the development stages. This also makes it easier to branch the system in GitHub for team members so that they do not overwrite each others code or break it.

Finally, the style made a good starting for our development team. As a fair few of our development team lacked a lot of programming experience, the system could be set up in an understandable manner to make it easier to work on and progressively learn overtime.

Here is a brief description on how these components have been used in the game:

- **The Model** contains both the Network Client to communicate with the server, and therefore, the other player, and all of the data of the Game being played, such as the Player objects (with their resources inventory and trains), the Game Map, the Store Objects, etc. This data is updated and then synchronised with the other player at the end of every turn.

- **The View** extends the JFrame used for the GUI window and contains all of the user interface components for the game. It also incorporates a number of methods used to set the listeners for the Buttons and takes care of some view-dependent logic such as making a request for the player's name. It also accommodates methods to update the content of all of the interface components and to get the panel that is used for the map to be drawn on the screen.

- **The Controller** takes care of all of the logic of the game. It listens for events both from the view (clicks from the player) and the model (moves - updates to the game data - received over the network from the other player). It is responsible for pushing all of the updated information to the view every time something changes in the model, recognising player actions and updating and notifying the model accordingly.

Interface-driven development

In order to facilitate collaboration on the code for the first models, we decided to adopt an interface-driven development pattern. We firstly agreed on the programming interface that all of the essential game models would need to implement by creating a prototype to work off, and wrote the corresponding Java Interface definitions. Then we proceeded splitting the models between ourselves, using GitHub's Issues as the main system for coordination and keeping track of the work. Each team member would be distributed onto a task based on their skill. They would each be writing a different part of the game without compatibility problems thanks to the programming interface we agreed in advance. All of the Java Interface files are still included in the project and used to make sure classes always comply with the interfaces.

Version Control (GIT)

We used a GIT repository hosted on GitHub as the version control system for the game code. This helped us to keep track of all of the work done by each of us, discuss code and track, discuss and assign issues. Additionally GitHub is open source, allowing all of our team to acquire the code from one location online rather than exchanging the system locally to ensure we all had the latest version. This would help future development teams acquire the system in the next development stage too. We also used GIT branching to allow us to work on different parts of the Game that would otherwise have been problematic for multiple people to work on at the same time. In addition, a working implementation could be consistently kept on the master branch to indicate what was fully functioning at different Scrum sprints and the system could backtrack to an older version in the event of an error causing the system to crash. Overall, this would help to manage our resources and time as well as improving the maintainability and adaptability of our system.

Javadoc

In order to facilitate the understanding of our code, ease its reuse and to allow the next team to easily understand what the classes are and what they do, we decided to adopt the Javadoc standard for comments, listing parameters, return functions and describing operations at the level of each function, class or public property. We also keep an API reference that is generated from the code of the Game, accessible online from the Game's website, to enable developers to easily locate information about the game code.

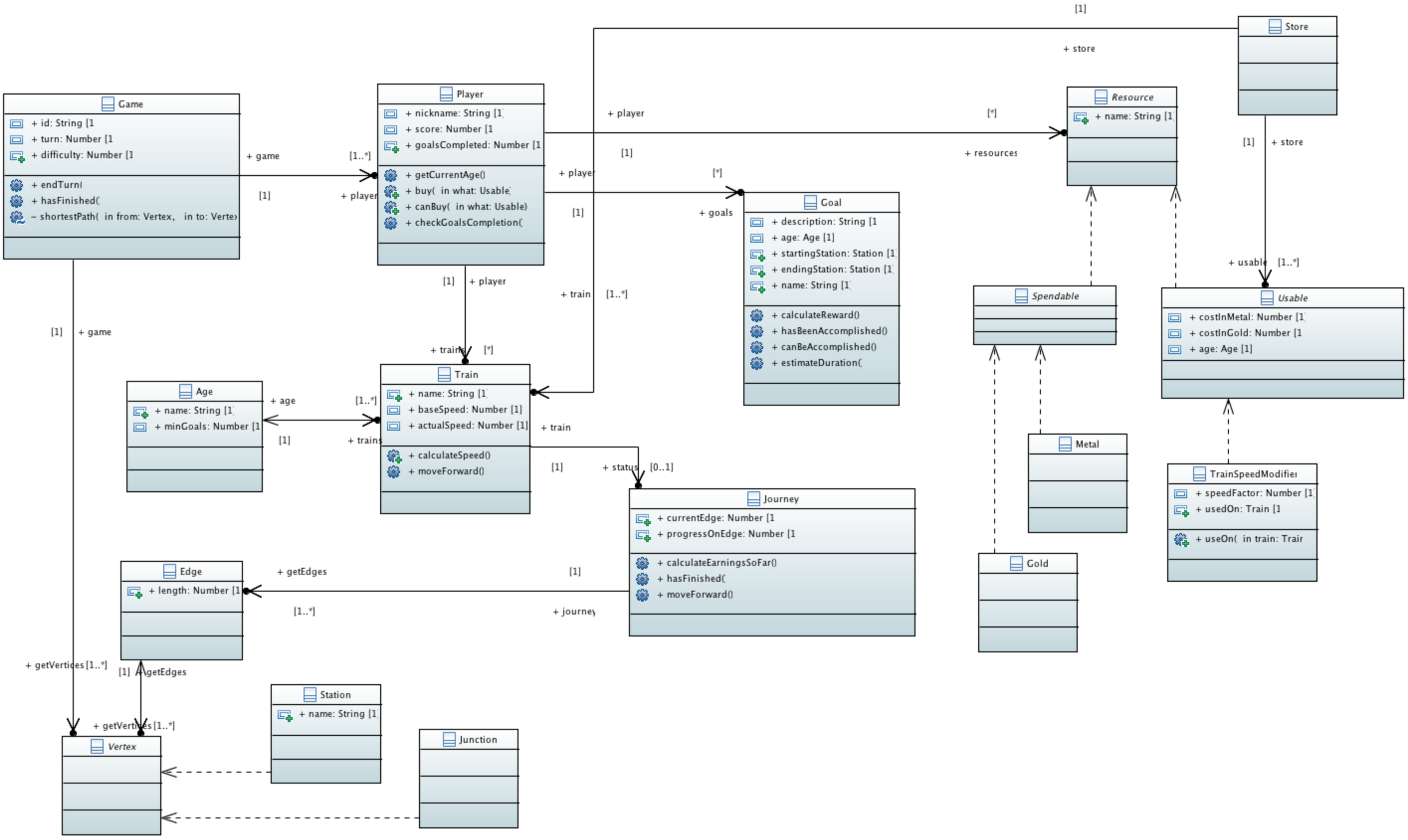
References

[1] http://socket.io/

[2] https://github.com/Automattic/socket.io

[3] https://github.com/Gottox/socket.io-java-client

[4] https://github.com/nkzawa/socket.io-client.java/issues/62


Game
+ id: String [1
+ turn: Number [1
+ difficulty: Number [1
+ endTurn(
+ hasFinished(
~ shortestPath(in from: Vertex, in to: Verte
Player
+ nickname: String [1
+ score: Number [1
+ goalsCompleted: Number [1
+ getCurrentAge()
+ buy(in what: Usable
+ canBuy(in what: Usable)
+ checkGoalsCompletion(
Goal
+ description: String [1
+ age: Age [1]
+ startingStation: Station [1
+ endingStation: Station [1
+ name: String [1
+ calculateReward()
+ hasBeenAccomplished()
+ canBeAccomplished()
+ estimateDuration(
Store
Resource
+ name: String [1
Spendable
Usable
+ costInMetal: Number [1
+ costInGold: Number [1
+ age: Age [1]
Metal
Gold
TrainSpeedModifier
+ speedFactor: Number [1
+ usedOn: Train [1
+ useOn(in train: Train
Age
+ name: String [1
+ minGoals: Number [1
Train
+ name: String [1
+ baseSpeed: Number [1]
+ actualSpeed: Number [1]
+ calculateSpeed()
+ moveForward()
Journey
+ currentEdge: Number [1
+ progressOnEdge: Number [1
+ calculateEarningsSoFar()
+ hasFinished(
+ moveForward()
Edge
+ length: Number [1
Vertex
Station
+ name: String [1
Junction
+ game
+ player
+ goals
+ resources
+ store
+ usable
+ age
+ trains
+ train
+ status
+ journey
+ getEdges
+ getVertices [1..*]
[1]
[*]
[1..*]
[0..1]