12.11.2019

Architecture Design

Aleksandr Kasian; Oskar Boer

Table of contents

[Architecture Design: Rules 3](#_Toc24410362)

[Good Architecture Rules: 3](#_Toc24410363)

[S.O.L.I.D. Rules 3](#_Toc24410364)

[Architecture Design: Decomposition 4](#_Toc24410365)

[Hierarchy 4](#_Toc24410366)

[Functionality 5](#_Toc24410367)

[GUI [View] 5](#_Toc24410368)

[Business Logic 6](#_Toc24410369)

[Agent and Map 9](#_Toc24410370)

[Problem – Solver Package 12](#_Toc24410371)

[Conclusion 12](#_Toc24410372)

[Bibliography 12](#_Toc24410373)

# Architecture Design: Rules

## Good Architecture Rules:

|  |  |  |
| --- | --- | --- |
| **Rule** | Description | Availability |
| **Scalability** | The ability to expand the system and increase its productivity by adding new modules. | Yes |
| **Maintainability** | Changing one module does not require changing other modules. | No |
| **Swappability** | The module is easy to replace with another. | No |
| **Unit Testing** | The module can be disconnected from all others and tested / repaired | No |
| **Reusability** | The module can be reused in other programs and other environments. | Yes |
| **Maintenance** | A module-based program is easier to understand and maintain. | Yes |

## S.O.L.I.D. Rules

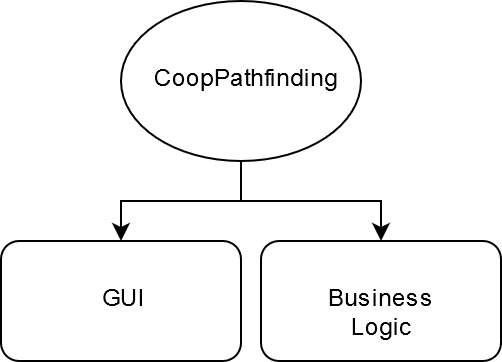
|  |  |  |
| --- | --- | --- |
| **Rule** | Description | Availability |
| Single responsibility principle | Class has one job to do. Each change in requirements can be done by changing just one class. | Yes |
| Open/closed principle | Class is happy (open) to be used by others. Class is not happy (closed) to be changed by others. | No |
| Liskov substitution principle | Class can be replaced by any of its children. Children classes inherit parent's behaviours. | Yes |
| Interface segregation principle | When classes promise each other something, they should separate these promises (interfaces) into many small promises, so it's easier to understand. | Yes |
| Dependency inversion principle | When classes talk to each other in a very specific way, they both depend on each other to never change. Instead classes should use promises (interfaces, parents), so classes can change as long as they keep the promise. | Yes |

# Architecture Design: Decomposition

## Hierarchy

Let’s use the MVC (Model – View – Controller) design pattern:

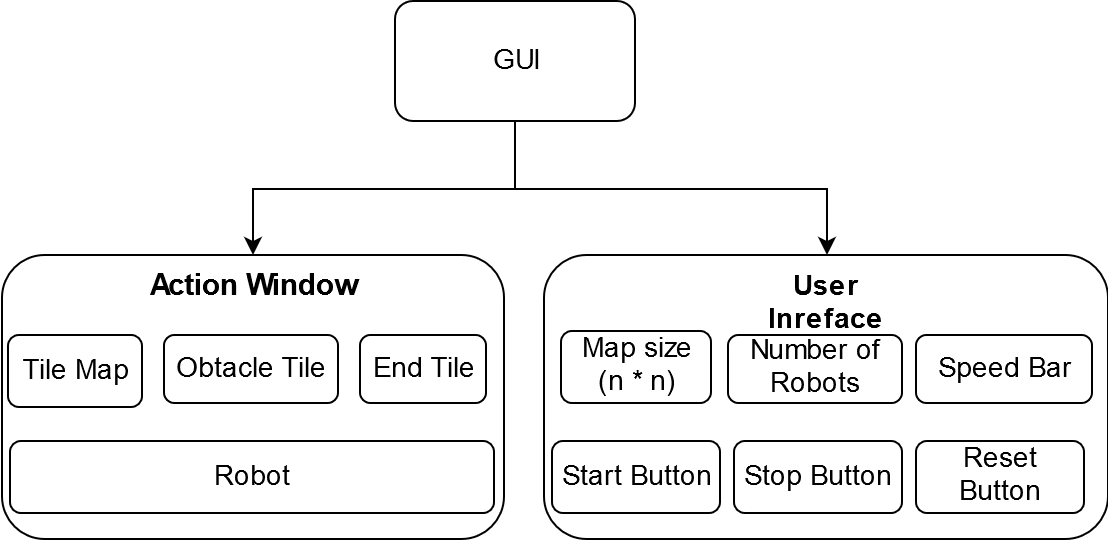
|  |  |  |
| --- | --- | --- |
| Part Number | Description | Part Name |
| 1 | It’s main logic of this application (business logic). | Model |
| 2 | An interface for this application. (Representation of information) | View |
| 3 | Accepts input and converts it to commands for the model or view | Controller |

1.  The ***model*** is responsible for managing the data of the application. It receives user input from the controller.
2. The ***view*** means presentation of the model in a particular format.
3. The ***controller*** responds to the user input and performs interactions on the data model objects. The controller receives the input, optionally validates it and then passes the input to the model.

## Functionality

### GUI [View]

|  |  |  |  |
| --- | --- | --- | --- |
| Submodule name | Object name | Description | Importance |
| Action Window | Tile Map | This is a map W\*H which consists of square tiles. Basic tile which is clear (neither Obstacle nor Finish) **is white**. | Important |
| Obstacle Tile | This is Obstacle Tile. Walls for the Robot (If Robots go on the black tile then he crashing). The color is black. | Important |
| End Tile | The End Tile is a finish for the each Robot  These tiles may or may not be the same. The color of this tile is the same as Robot color. If several number of Robots have the same finish than tile ??? | Important |
| Robot | Robot it’s a cyrcle with their own number and probably color. | Important |
| User Interface | Map size | List with size W\*H. | Important |
| Number of Robots | Input field where user set the number of Robots. Number of the Robots should be in the range [1; N\*M] where N/M is size (row/column)  16 | Important |
| Speed Bar | This is bar which represent velovity of animation. | Not Important |
| Start Button | Starting animation | Important |
| Reset Button | Reclaim start positions | Important |
| Stop Button | Stop animation | Important |



### Business Logic

|  |  |  |
| --- | --- | --- |
| Data Name | Description | Type |
| Agent | This is the Robot. I will Describe this class a little bit later. | Class (type) |
| Map | The Map-class. =\ | Class (type) |
| Problem Solver | This is the package which contain the algorithms for the solving. | Package (module) |
| Hash Map | Standard library package | STL Package |

Overall objective is collaboratively pathfinding for all agents, make sure they don’t collide, and reach goal safe and sound. System initiates when all agents are deployed at their own starting position on the map. System shall end when all agents reach goal position or there is no pathfinding solution. System returns an error message when agents cannot find path.

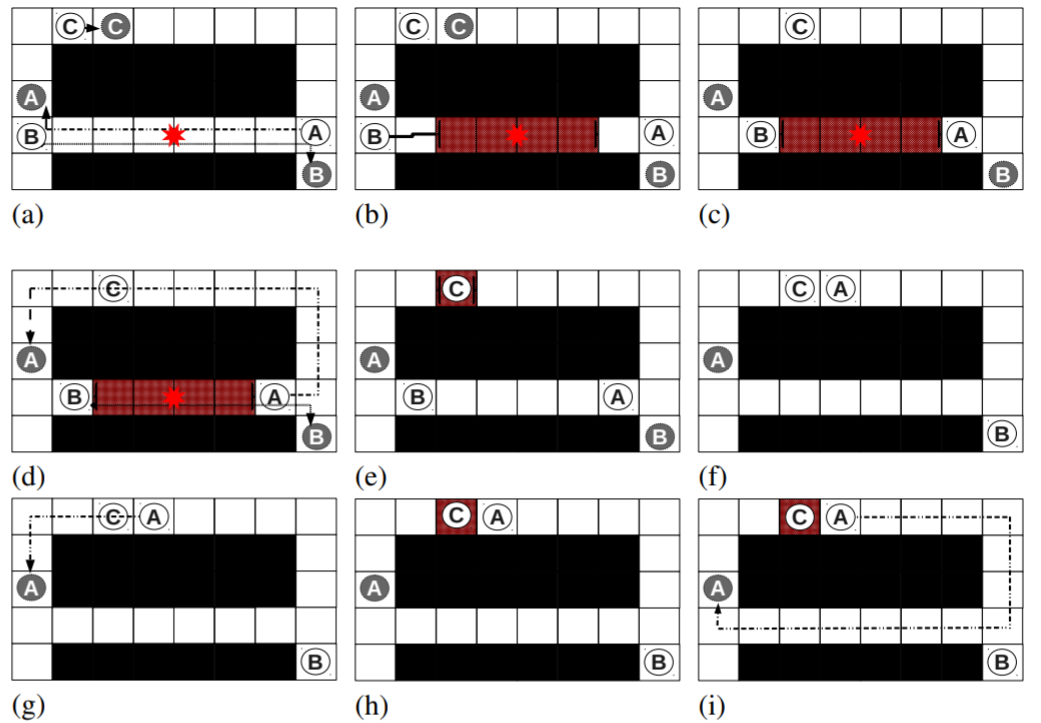
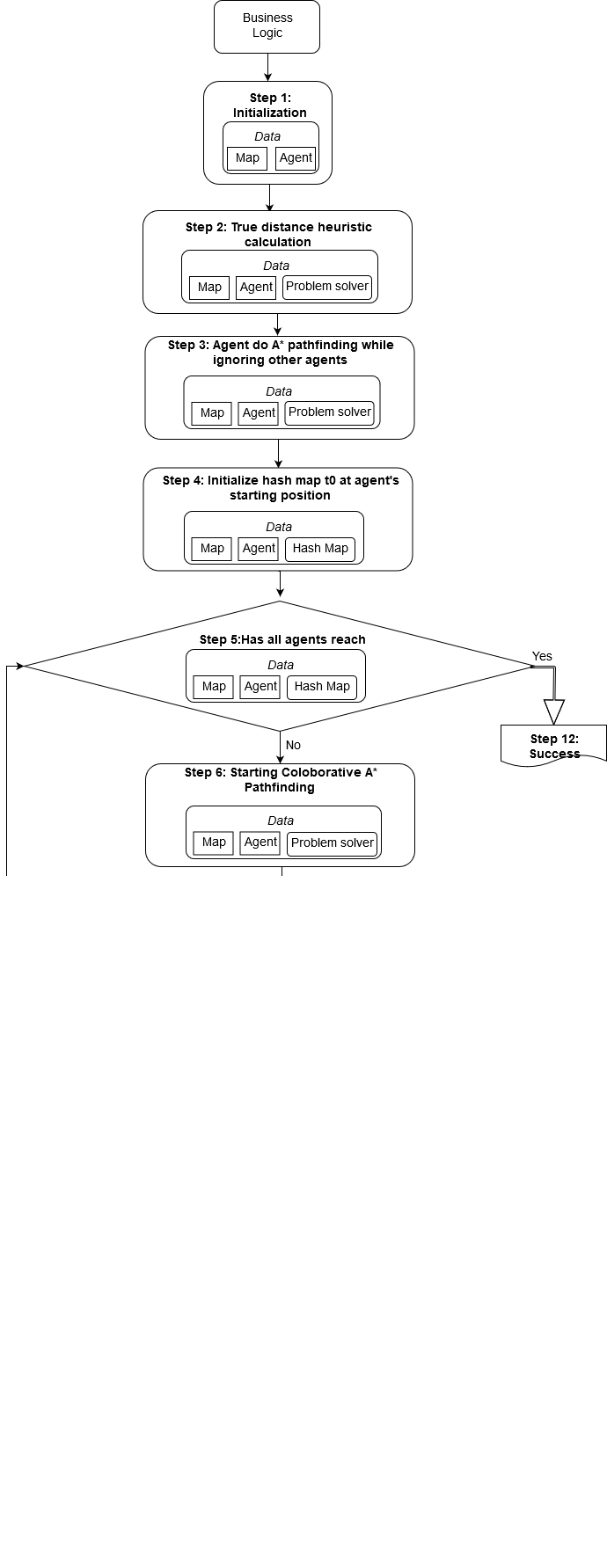
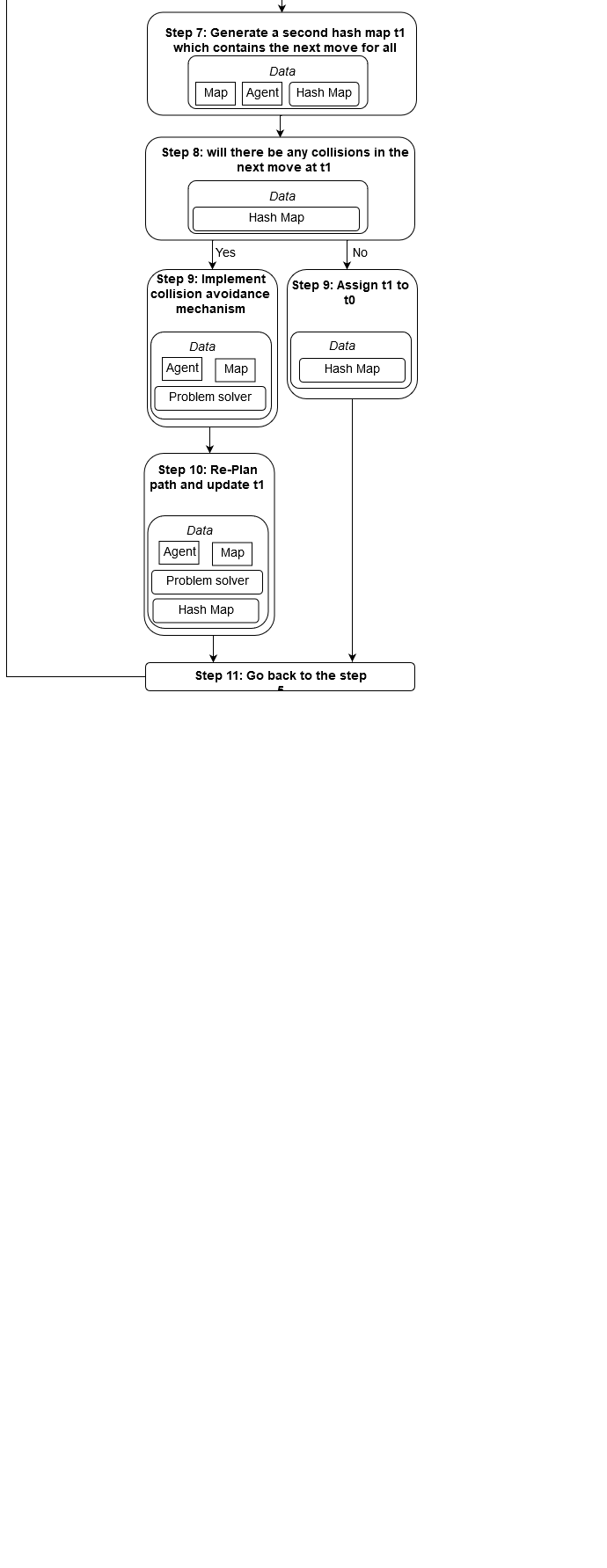
“To demonstrate the main purpose, consider Figure 1. Initially, agents A and B (agents locations are marked with white circles), each need to cross the entire grid to get to their goals while agent C is one step away from its goal and the reservation table is empty. The first plan-move cycle begins and each of the agents find a path to their goal (Figure 1(a)). By simulating the paths, the first conflict is found between the paths of agents A and B at time point t (marked with a red X). In this example, agent B is chosen arbitrarily as the conflict master. We assume a fixed window size of size W and that the reservation window should be positioned evenly around the conflict. Thus, we select wstart and wend such that W st-points around the conflict are reserved by agent B (Figure 1(b)). Next, the move phase begins (Figure 1(c)) and the agents proceed to their locations at wstart. A new plan phase begins at step (d). Since agent B reserved the interval adjacent to conflict c, agent A plans a detour - a path that conflicts with agent C. Now, C gets to reserve the interval (Step (e)) and the next move phase advances agent A near the location of c (Figure 1(f)). Next, a new plan is found and all agents. Agent C again reserves the interval around the conflict and agent A plans an alternative path to its destination. This time, none of the agents have conflicts. The space requirements of the reservation table used in WHCA\* and HCA\* depends on the number of agents (W · |A| in WHCA\*) since all agents reserve st-points at each phase. In CO-WHCA\*, the reservation table is usually a vector of size W since most of the time, only one agent uses the reservation table for a limited duration. One exception to this is when multiple conflicts occur at the same time.”

Figure 1: WHCA\* Running





### Agent and Map

Agent

|  |  |  |
| --- | --- | --- |
| Name | Description | Type |
| Node | The basic unit of a path of an agent, include x, y coordinates, f\_score, g\_score. | Class (type) |
| Agent | An object with a name (number, of aggent), starting point and goal point. This object will move across the map, from the start point toward the goal point. | Class (type) |
| (x, y) | Point, basic unit of space map | uint |
| g\_score | The g(x) function, which estimate the distance travelled by agent so far from start to Node. | uint |
| f\_score | The f(x) = g(x) + heuristic distance (Manhattan Distance Heuristic), meaning the estimation of distance from start to goal. Implemented in hash table <Node, f\_score> | uint |
| closed\_set | Explored points (Nodes), meaning its parent, g\_score, f\_score is calculated | unordered\_map |
| open\_set | The set of points (Nodes) newly found, but unexplored. | prior\_queue |
| came\_from | A hash table <child, parent> indicates that child node came from parent node. | unordered\_map |
| space\_map | A two – dimensional grid map consists X and Y coordinates. | unordered\_map |
| name | The name, namely number – string | string (method) |
| get\_name | Returns the name of an Agent | string (method) |
| set\_start | Set starting position | void (method) |
| get\_start | Returns starting position | Node (method) |
| set\_goal | Set goal position | void (method) |
| get\_goal | Returns goal position | Node (method) |
| set\_whole\_path | Using A\* Search to set complete | void (method) |
| get\_path | Get agent path | void (method) |
| set\_portion\_path | Set agent portion path, i.e. each agent walks n steps in turn, where n is the window size. | void (method) |
| get\_portion\_path | Get agent portion path | void (method) |
| set\_current\_node | Set agent current position | void (method) |
| get\_current\_node | Get agent current position | Node (method) |
| set\_prev\_node | Set agent previous position | void (method) |
| get\_prev\_node | Get agent previous position | Node (method) |
| get\_next\_node | Get agent next position, return false if agent is at end of window | bool (method) |
| print\_path | Print agent’s whole path using basic A\* Search | void (method) |
| get\_path\_length | Get agent path length | uint (method) |
| insert\_path\_after\_front | Insert a path after current node | void (method) |
| pop\_front\_node | Pop out current node from path | void (method) |
| get\_front\_node | Took at agent current position | void (method) |

Map

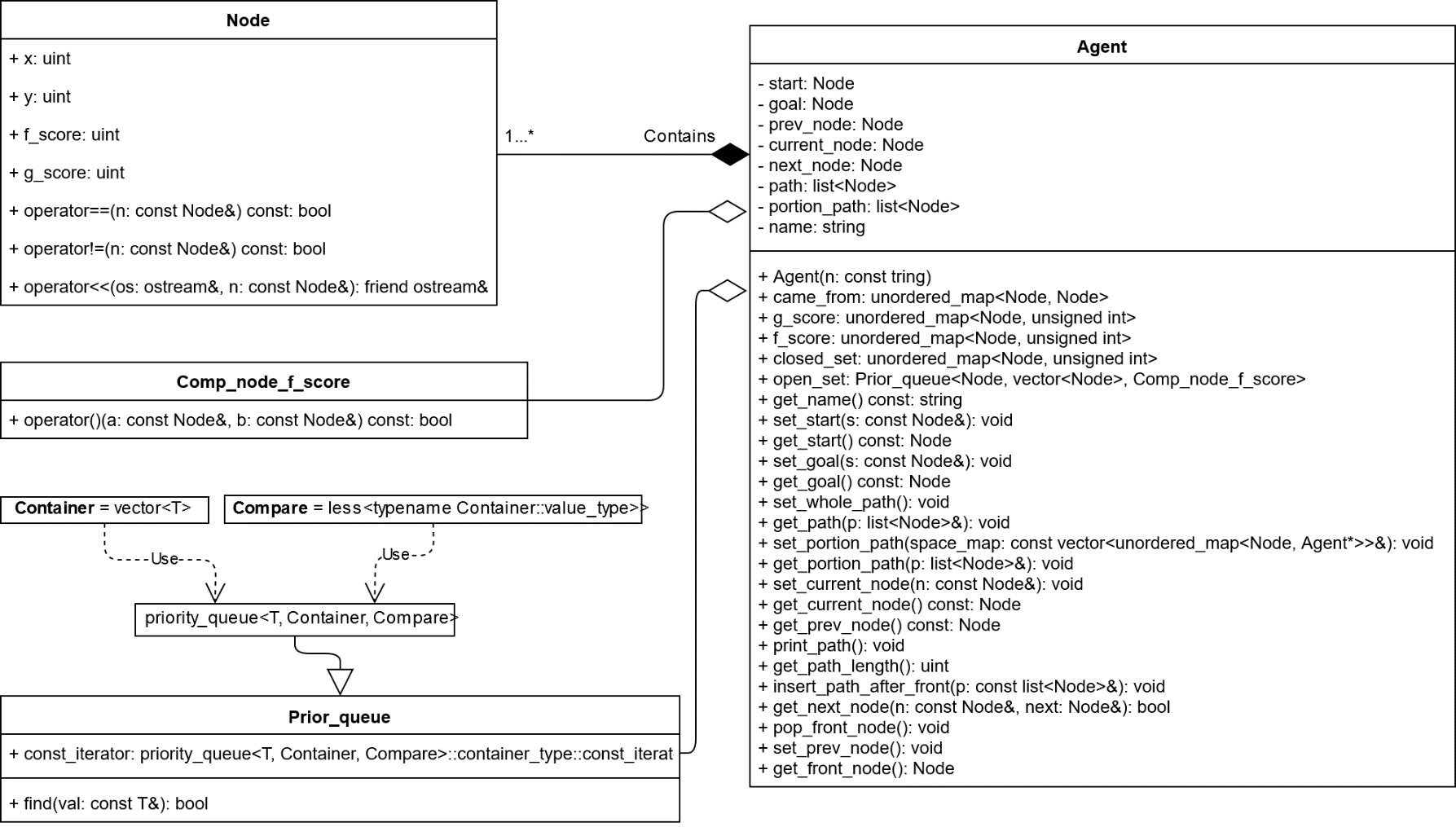
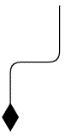
|  |  |  |
| --- | --- | --- |
| Name | Description | Type |
| Map | The map where the agents will find the paths. | Class (type) |
| (w, h) | Width and height of the map. | uint |
| obstacle\_percent | The percent of obstacles relative to the entire map. | uint |
| square | The square of the map, w\*h. | uint |
| get\_height | Get height of the map | uint (method) |
| get \_width | Get width of the map | uint (method) |
| get \_square | Get square (W\*H) of the map | uint (method) |
| get\_map\_data | Get all data of the map. (include start/end point, agents, obstacles e.t.c.) | char\*\* (method) |
| set\_obstacle | Set obstacle on the map | void (method) |
| get\_obstacle | Get obstacle | void (method) |
| is\_obstacle | Is a node as an obstacle | bool (method) |
| get\_neighbors | Get neighbors and we don’t care either node is an obstacle or node is empty or it’s an agent e.t.c. | void (method) |
| get\_clean\_neighbors | Get neighbors which are not contains an agent. | void (method) |
| update\_map | Update the map | void (method) |
| print\_map | Just print a map | void (method) |
| print\_map\_with\_agents | Print a map with agents | void (method) |
| print\_agent\_path | Print agent path. | void (method) |
| print\_time\_space\_map | Print time space map | void (method) |

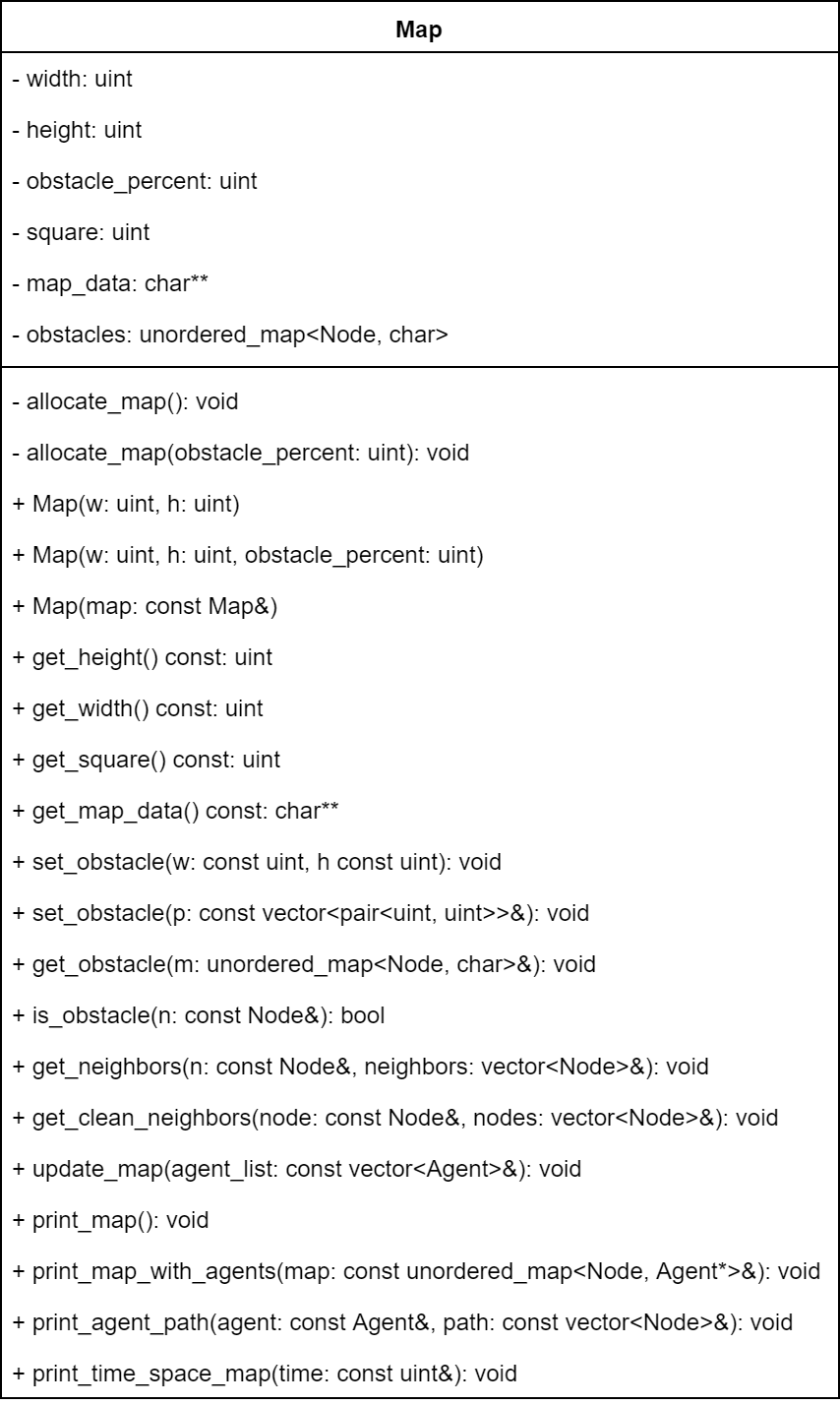
To seek high performance, Agent architecture is meant be designed as simple as possible.

On the contrary, the coordination mechanism in terms of solving interdependency issues shall be complex enough to consider all the scenarios. We want agents cooperate only when they have to. In other words, agents do not need to communicate with others if they know in advance they will not collide. Collison detection is the coordinator’s job. Once coordinator finish planning the path for all the agents, agents can walk without worry about collision.

The coordination mechanism is embedded inside each agent (Probably I will separate the coordination from each agent). From a single agent perspective, it has a name, a starting point, a goal point. The knowledge of each agent includes map, closed set, open set, g-score hash map, f-score hash map.

Most of coordination, interaction are embedded inside problem – solver functions. A star search, Reverse Resumable A Star Search, True distance calculation, Collision detection and avoidance are separated from agent architecture. I will describe them each in detail in the “Project-Explanation” document.





### Problem – Solver Package

|  |  |  |
| --- | --- | --- |
| Name | Description | Type |
| Manhattan Distance Heuristic | An estimated distance ignoring obstacles between two points. | uint |
| True Distance Heuristic | It distance taking into account of obstacles. | uint |
| A\*Search | Simple A\*Search. Pretty good description tou will find on the Wikipedia by following the [link](https://en.wikipedia.org/wiki/A*_search_algorithm) | bool |
| Reverse Resumable A\* Search | Back to the problem of calculating true distance heuristic, the solution is to use Reverse Resumable A\* proposed by David Silver. But I will make the only difference, I will make a full search once, without resume. I plan to do it on GPU, because this method to expensive. | bool |
| Collision Detection And Avoidance | Agents interact through the space\_time\_map windowed time (W steps) in turn based on other agents on that space\_time\_map. | Several types |

# Conclusion

First of all, it is not the end. In future I will add some features such as “Path Rejoining”, “Predictor” (I described these in “Project-Explanation” document), agent – velocity, diagonal – movement, bad agents (whose purpose is to stop other agents), and so on, and so on.

Secondly, pathfinding is indeed very interesting project, and complex to solve if other constraints are considered. The project is meant to be a learning experience for myself. It is far from completing, and the current research trend on multi-agent pathfinding is far ahead from this little project.

# Bibliography

[1] Silver D. (2005). Cooperative Pathfinding. ([link](http://www0.cs.ucl.ac.uk/staff/D.Silver/web/Applications_files/coop-path-AIIDE.pdf))

[2] Zahy Bnaya and Ariel Felner. Conflict-Oriented Windowed Hierarchical Cooperative A\*. ([link](http://www.bgu.ac.il/~felner/2014/COWA6p.pdf))