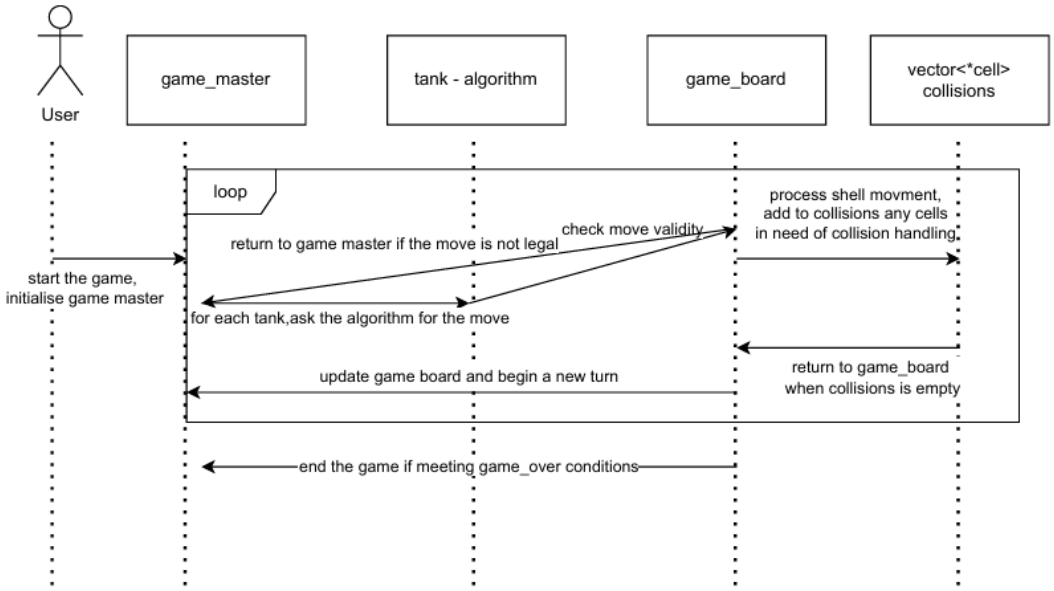
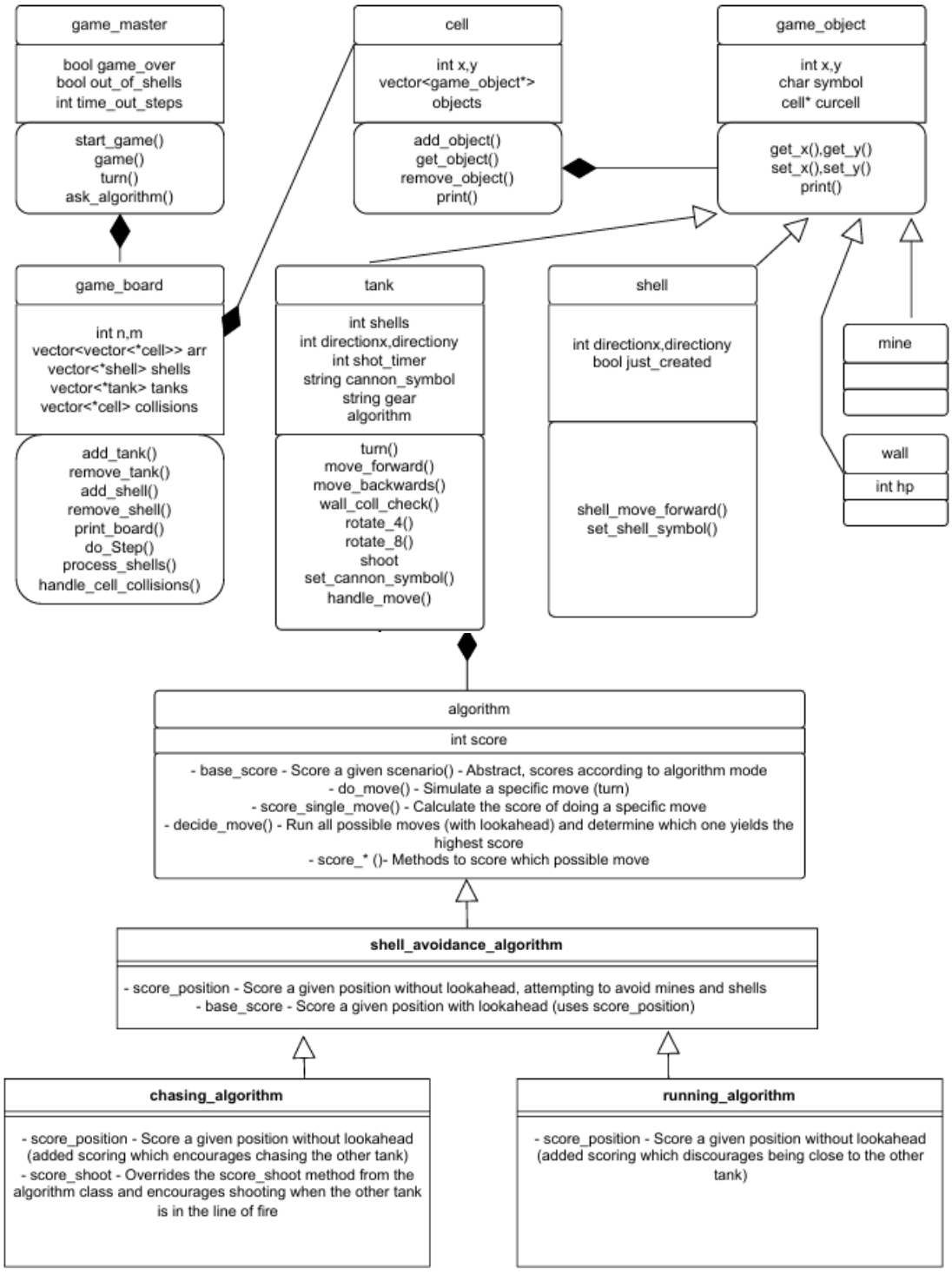
Flow chart


User
game_master
tank - algorithm
game_board
vector<*cell>
collisions
loop
process shell movment,
add to collisions any cells
in need of collision handling
check move validity
return to game master if the move is not legal
start the game,
initialise game master
for each tank,ask the algorithm for the move
return to game_board
when collisions is empty
update game board and begin a new turn
end the game if meeting game_over conditions

Class Design



Designing and testing process

We designed the project relying heavily on polymorphism concepts to assure a dynamic and maintainable codebase. The main parts of our code are:

-The game_object(s), which are all the objects which can be placed on the game boards (mines, shells, walls, tanks).

-The game_board itself, which holds an array of all the cells as well as some lists with direct references to certain types of objects (shells and tanks).

-The game_master, which manages the execution of the game (prompts the algorithms and runs each game step).

-The algorithm(s), responsible for deciding the moves of each players.

The main consideration point was regarding the implementation of the algorithms. The first most straight forward idea for the algorithm's design would be to pre-define a set of rules (if ..., then do ...). However, this design is both hard to maintain as all the rules have to be manually written down, and prioritising one choice over another becomes a very difficult task. Instead, we came up with a much more dynamic idea which also allows for much "smarter" agents. We still need to manually define some rules, but much less. Instead of defining rules to determine the moves themselves, we define rules based on which we "score" a given board state. This scoring allows us to test all the possible moves and yield the highest scoring one. Additionally, it allows us to have a look into the future and see which moves could benefit us later - we simply simulate the board's progress over time, and see how the scores progress. So to decide on a move we check for the combination of the next N (currently defined to be 3) moves which yields the highest combined score.

This design even helped us with our testing - since we try so many different move combinations (by brute-forcing all near-future possibilities), we ended up "unintentionally" testing lots of scenarios, and it helped us find lots of bugs which were hiding in some curious edge cases.

The - perhaps more difficult part, of managing these types of algorithms, is determining the scoring systems and the weights of certain characteristics (proximity to mines, shells, other tanks, line of fire availability, etc.). To fine-tune it, we tested many different board states, and each time we focused on one of the algorithms (chaser/runner), trying to optimize it. If the runner wasn't running fast enough from the chaser, we increased the negative weight of being close to another tank. If the tanks were too nonchalant regarding shells flying towards them, we increased that weight. We went on with this process for a while until we felt that the fight was balanced and the tanks were behaving in a relatively smart manner.