TAU - Advanced Topics in Programming, Semester B 2025

Assignment 2 (Link to Assignment 1)

Note: This document is in DRAFT mode till May-5th 23:00. While in draft mode, it may have changes without highlighting them. Even though it is still a draft, you may (and should) start working on it!

During the draft period: you are mostly welcomed to add questions directly into this document, as comments. Please assign your comments and questions to kirshamir@gmail.com so I will get an email when you add them. Then please follow my answer and if it resolves your comment or question please close it.

After the draft period: commenting on the document will be closed and all questions on the assignment would be submitted in the relevant forum in the course site! Changes in the doc, if any, but then they would be highlighted in the document. Note: if a change creates more work for you (e.g. you already implemented things differently), please raise the issue in the assignment forum before rushing to change your code!

Submission deadline: June-8th 2025, 23:30.
Note: here are the Submission Instructions - please make sure to follow them!

Assignment 2 - Requirements and Guidelines

In this assignment, the API and file format would be strictly defined, so your code can work with algorithms implemented by other teams and with house files created by other teams.

Game Map Input File - TEXT file

The format of the game map file shall follow the following strict instructions:
Line 1: map name / description - internal for the file, can be ignored by the game manager
Line 2: MaxSteps for the game on this map, as: **MaxSteps = <NUM>**
Line 3: NumShells per tank, as: **NumShells = <NUM>**
Line 4: Number of Rows in house, as: **Rows = <NUM>**
Line 5: Number of Cols in house, as: **Cols = <NUM>**
Lines 6 and on: the map itself, as described below

Notes:

1. All values in lines 2-5 may or may not have spaces around the = sign
2. If the file is invalid you can reject it and print the reason to screen

The actual size of the map is determined by the values in lines 4 and 5. If there are missing lines, they should be filled in with empty lines, if there are missing columns they should be filled in with spaces. In case the file contains additional rows and/or cols beyond the numbers as set in lines 4 and 5, the program shall ignore the redundant lines / cols.

The following characters shall be used in the file for the map:

Empty Space (“corridors”): space as well as any character which does not have other mapping below
Wall: #
Mine: @
A tank that belongs to player 1: 1 (can be any number, including 0)
A tank that belongs to player 2: 2 (can be any number, including 0)

If one of the players doesn’t have any tanks, the player loses when the game starts (before round 0!). If both do not have any tanks, it’s an immediate tie.

Position of the tanks’ cannons on start would be *Left* for player 1 (thus shooting left) and *Right* for player 2. The cannon direction also sets the forward movement direction of the tank.

Example of valid houses:

```
1-Small battlefield
MaxSteps = 5000
NumShells=20
Rows = 4
Cols =10
#### @ @
# 1    @
#  2
```

```
2-Another Small battlefield
MaxSteps=2000
NumShells=200
Rows= 4
Cols=12
############ 0
# 2    2   # 1
   # 1 @ #   2
############ 3
012345678901
```

API

You should use the following enums, classes and interfaces, as provided, without changing them at all, each should sit in the defined location (folder/filename):

common/ActionRequest.h

```
enum class ActionRequest {
	MoveForward, MoveBackward,
	RotateLeft90, RotateRight90, RotateLeft45, RotateRight45,
	Shoot, GetBattleInfo, DoNothing
};

// Note:
// All the actions in the enum work exactly as in exercise 1
// with a new action: “GetBattleInfo” which will be explained below
```

---

common/TankAlgorithm.h

```
class TankAlgorithm {
public:
	virtual ~TankAlgorithm() {}
	virtual ActionRequest getAction() = 0;
	virtual void updateBattleInfo(BattleInfo& info) = 0;
};
```

---

common/BattleInfo.h

```
class BattleInfo {
public:
	virtual ~BattleInfo() {}
};
```

---

common/SatelliteView.h

```
class SatelliteView {
public:
	virtual ~SatelliteView() {}
	virtual char getObjectAt(size_t x, size_t y) const = 0;
};
```

The class SatelliteView represents the view of the battlefield from a satellite.
Its function “getObjectAt” returns the object at the requested location, as a char.
Locations are indexed from (0, 0) at the top left corner of the board, to (max_x, max_y) at the lower right.
The char provided back by this function would be:

#	For a “wall”
1	For a tank owned by player 1
2	For a tank owned by player 2
%	For the tank which requested the info
@	Mine
*	For artillery shell, in case an artillery shell is in the air above a mine, the satellite sees the artillery shell and misses the mine
Space	If there is nothing at this location
&	If the location is outside the battlefield

Note that the satellite cannot see or report the *movement directions* of moving objects.

---

common/Player.h

```
class Player {
public:
      Player( int player_index,
              size_t x, size_t y,
              size_t max_steps, size_t num_shells ) {}
      virtual ~Player() {}
      virtual void updateTankWithBattleInfo
            (TankAlgorithm& tank, SatelliteView& satellite_view) = 0;
};
```

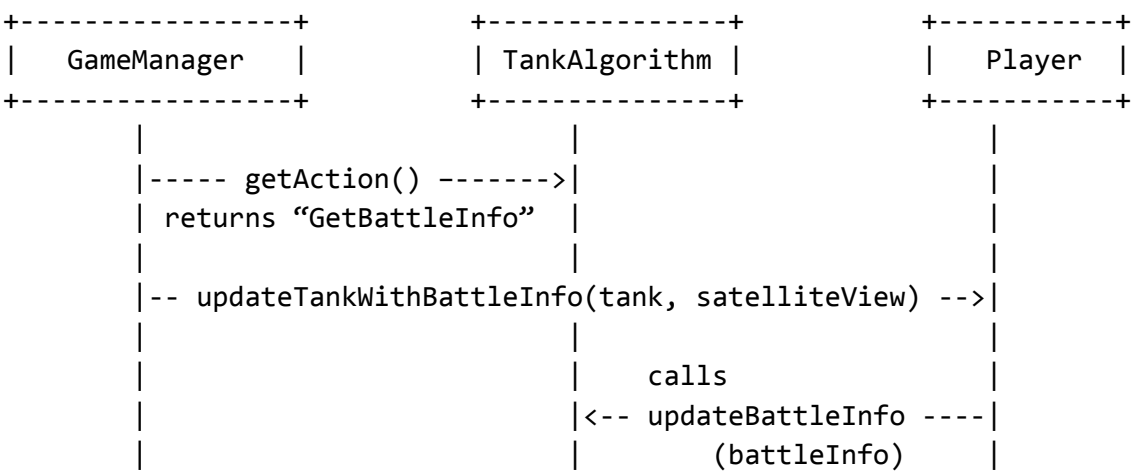
“Player” is an interface for the players in the game.
Given 2 players, there would be 2 “Player” objects.
You are required to implement different concrete “Player” classes for the two players.

Player can coordinate its tank movements when providing battle info to its tanks. You should think about how to perform that.

When a player object is born it gets in the ctor the dimensions of the battlefield and the index of the player, as 1 or 2, max steps for this game and max shells per tank for this game. It is clueless regarding any other information.

When a tank algorithm requested action is “GetBattleInfo”, the GameManager reacts immediately with a call to the tank’s player object function “updateTankWithBattleInfo”.

The sequence will look as following:



Notes:

1. The provided argument “satelliteView” in *updateTankWithBattleInfo* is a view of the board **as in the previous round** (all actions performed in this round are NOT updated yet in the view).
2. The view is valid for immediate use, but **cannot** be copied and **cannot** be stored for later use as the reference may expire! If the player wants to remember information it should use its own data structure.
3. Player class role is to coordinate between all the tanks of the same player. The data sent to the algorithm in *updateBattleInfo* would be a derived class of BattleInfo. The algorithm knows which actual type to expect and can cast to the right type.
4. The algorithm gets a non-const reference to the BattleInfo object, it can use functions on the actual type for sending back data to the Player object (such as its location, number of remaining shells etc.).
5. It is **forbidden** for the algorithm to store a reference or a pointer to the provided battleInfo object (doing so allows communication between the Player and the Algorithm in ways that are not permitted, it is considered as “cheating” and would reduce points!).

Your GameManager class should get in its constructor two factory methods, one for creating TankAlgorithms and another for creating Player.

The factories would derive from the following interfaces:

---

common/PlayerFactory.h

```
class PlayerFactory {
public:
      virtual ~PlayerFactory() {}
      virtual unique_ptr<Player> create(int player_index, size_t x, size_t y,
                        size_t max_steps, size_t num_shells ) const = 0;
};
```

---

common/TankAlgorithmFactory.h

```
class TankAlgorithmFactory {
public:
      virtual ~TankAlgorithmFactory() {}
      virtual unique_ptr<TankAlgorithm> create(
            int player_index, int tank_index) const = 0;
};
```

Note:
The player_index provided in both factories above will always be 1 or 2.
The tank_index provided in the TankAlgorithmFactory will be zero-based, indexing each of the players' tanks (e.g. 0, 1, 2 for player 1 and 0, 1 for player 2 – if 1 has 3 tanks and 2 has 2 tanks).

Your tasks:
You need to implement the relevant concrete classes for the above abstract classes and to implement a GameManager class that manages the game, including writing to an output file as in assignment 1.

Main
Your main should look as the following (note: it doesn't have to be exactly the same, **but the API used below shall be the same**):

```
// getting command line arguments for the input file
int main(int argc, char** argv) {
      GameManager game(MyPlayerFactory{}, MyTankAlgorithmFactory{});
      game.readBoard(<game_board_input_file from command line>);
      game.run();
}
```

Algorithm

Note: your algorithm in this assignment MUST be deterministic, that is no randomness is allowed, to make sure that running your algorithm again on the same board will have the exact same results. (It is not for saying that randomness in algorithms is a bad thing, actually randomness is occasionally used in algorithms to get efficiency, but you should not, as we want to be able to always reproduce the exact same results when running a given algorithm on a given board).

It is nice if your Player class tries to actually coordinate between player's tanks, but it is not mandatory.

Output File - TEXT file format

Let's assume tanks are "sequenced" in the order they were "born" in the board, from top left, row by row (that is a higher tank in board is "born" before a lower one, two tanks on same row, the left one is "born" before the right one).
The output file will include a single line per game round.
Per each line: the action performed by all tanks, comma separated, in their order.
If a tank is killed, its action will appear with (killed) at the round it was killed, then as "killed" from the next round on.
If a requested action cannot be performed it will appear in file with (ignored) right after.
Action names are as in the enum.
Last line in file will announce the winner in the following format:
Player <X> won with <Y> tanks still alive
Or:
Tie, both players have zero tanks
Or:
Tie, reached max steps = <max_steps>, player 1 has <X> tanks, player 2 has <Y> tanks
Or:
Tie, both players have zero shells for <40> steps ← write the code so "40" will not be in the string ahead, e.g. use a constant

Output file example, for a case of 4 tanks and max_steps = 50:

```
MoveForward, MoveBackward, RotateLeft90, RotateRight90
RotateLeft45, RotateRight45 (ignored), Shoot, GetBattleInfo
DoNothing, DoNothing, Shoot (ignored), MoveForward
RotateLeft45, MoveForward (killed), MoveForward, MoveForward
MoveForward, killed, MoveForward, MoveForward (ignored) (killed)
Player 2 won with 2 tanks still alive
```

Note: if a move was ignored (as illegal) and the tank also killed in the same step, it will be indicated as: (ignored) (killed) – as presented in the last steps line above.

No *new* and *delete* in your code

Your code shall not have *new* and *delete*.
You can use std containers.
You can use smart pointers, make_unique, make_shared.
Note:

1. Do not use shared_ptr where unique_ptr can do the job.
2. Do not use allocations managed by smart pointers if stack variables can do the work.

Error Handling

Exact same instructions as in assignment 1.

Running the Program

tanks_game <game_board_input_file>

Submission

You should submit along your code 3 input files with 3 corresponding output files.

Bonuses

You may be entitled for a bonus for interesting implementation.
The following may be entitled for a bonus:

- adding logging, configuration file or other additions that are not in the requirements.
- having automatic testing, based on GTest for example.
- adding visual simulation (do not make it mandatory to run the program with the visual simulation, you may base the visual simulation on the input and output files as an external utility, it can be written in C++ or in another language).
- Implementing the game for any number of players between 2 to 9. If you implement this, add a proper 4th input file with more than 2 tanks, with a proper output file.

IMPORTANT NOTE:

1. In order to get a bonus for any addition, you MUST add to your submission, inside the main directory, a *bonus.txt* file that will include a listed description of the additions for which you request for a bonus.
2. If you already asked for a bonus in assignment 1 for a certain addition you shall not ask for the same bonus again. If you added something, focus in your bonus.txt file on the addition.