TAU Advanced Topics in Programming - 2018B - Exercises

Requirements and Guidelines - Ex2

In this exercise you are required to implement two types of Algorithms for playing the rock–paper–scissors game:

(1) FilePlayerAlgorithm - this algorithm behaves the same as the file behavior in Ex1
(2) AutoPlayerAlgorithm - this algorithm plays automatically, according to the info provided to it

Both algorithms should be based on (i.e. inherited from) an interface (i.e. abstract class) of PlayerAlgorithm that would be explained below.
Together with actual Game that will use the Algorithms and manage a real game.

Note, In EX2 we still have (see legend for the below in Ex1):
M = N = 10
R = 2
P = 5
S = 1
B = 2
J = 2
F = 1
All classes can assume the values above (but please avoid using "magic numbers").

Your implementation must include the following methods. Please add brief documentation for and check all edge cases. The methods are self explanatory, you should use them in order to implement your program, the name of each class appears in bold (PlayerAlgorithm, Point, PiecePosition etc.), the full signature of each method is given below, all methods are public.

PlayerAlgorithm abstract class

virtual void *getInitialPositions*
    (int player, std::vector<unique_ptr<PiecePosition>>& vectorToFill);
virtual void *notifyOnInitialBoard*
    (const Board& b, const std::vector<unique_ptr<FightInfo>>& fights);
virtual void *notifyOnOpponentMove*(const Move& move); // called only on opponent's move
virtual void *notifyFightResult*(const FightInfo& fightInfo); // called only if there was a fight
virtual unique_ptr<Move> *getMove*();
virtual unique_ptr<JokerChange> *getJokerChange*(); // nullptr if no change is requested

Point abstract class

virtual int *getX*() const;
virtual int *getY*() const;

PiecePosition abstract class

virtual const Point& *getPosition*() const;
virtual char *getPiece*() const; // R, P, S, B, J or F
virtual char *getJokerRep*() const; // ONLY for Joker: R, P, S or B -- non-Joker may return '#'

Board abstract class

virtual int *getPlayer*(const Point& pos) const; // 1 for player 1's piece, 2 for 2, 0 if empty

FightInfo abstract class

virtual const Point& *getPosition*() const;
virtual char *getOpponentPiece*() const; // R, P, S, B or F (but NOT J)
virtual int *getWinner*() const; // 0 - both lost, 1 - player 1 won, 2 - player 2 won

Move abstract class

virtual const Point& *getFrom*() const;
virtual const Point& *getTo*() const;

JokerChange abstract class

virtual const Point& *getJokerChangePosition*() const;
virtual char *getJokerNewRep*() const; // R, P, S or B (but NOT J and NOT F)

Flow of the game:

[1]

Command Line - support the following options:

auto-vs-file	// first player "auto", second file - read files as player2, as in Ex1
file-vs-auto	// second player "auto", first file - read files as player1, as in Ex1
auto-vs-auto	// both players are "auto" - see FAQ below
file-vs-file	// both players are file based - read files for both, as in Ex1

[2]

Positioning

Game should call PlayerAlgorithm methods:

- getInitialPositions(int player); // with 1 - for player 1
- getInitialPositions(int player); // with 2 - for player 2

Then perform all fights on the initial positions and call PlayerAlgorithm methods:

- notifyOnInitialBoard(const Board& b, const std::vector<FightInfo>& fights);

Note: FilePlayerAlgorithm probably ignores the info provided in notifyOnInitialBoard, that is: the method is probably empty in class FilePlayerAlgorithm.

[3]

Moves

Player 1 starts.

[A TURN] For each turn, Game should call on the two PlayerAlgorithms:

- getMove();

For the player who just moved:

- notifyFightResult(const FightInfo& fightInfo); // only if there was a fight
- getJokerChange();

For the other player:

- void notifyOnOpponentMove(const Move& move);
- void notifyFightResult(const FightInfo& fightInfo); // only if there was a fight
  => go back to **[A TURN]** for this player

Output file and Errors

As in Ex1

Additional Notes

Tie result due to “no fight” for 100 moves

In case there is no fight for more than 100 moves the game is ended with a tie result (a move is counted as player’s move, so 100 moves = 50 moves of each player, note: a move is considered 1 move regardless of whether there was a joker representation change in this move or not, i.e. joker representation change is **not** considered as additional move - it is part of the move and the entire move is counted always as one move).

Additional Requirements

Memory Management

It is not a requirement, but may give a bonus, if you manage all your heap memory needs with smart pointers (unique_ptr, shared_ptr) and / or with the collection classes.
In order to get this bonus add an empty text file to your submission with the name:

```
  no_raw_allocation_bonus.txt
```

Having this file in your submission while you do have raw allocations (you have in your code “naked new and/or naked delete”) -- may result with a fine fine. So please check yourself.

Documented Code

Document your classes with a descent header: author (full name, don’t get shy), description
Document complicated code
Do not document what the next line of code does (the code should say it) - document what you are trying to achieve in the next block, what is the approach, why you do that etc. (and only in the parts of “complicated code”).

Code Quality

Avoid unnecessary copy-paste.
Keep methods short and simple as possible.
Names are crucial: for classes, variables and methods. Invest thought in picking your names.

FAQ

Why do we need this exact class definitions?

In Ex3 we would run a tournament between AutoPlayerAlgorithm implemented by different teams. In order to allow it, the signatures of the classes and the flow should be exact - otherwise you would have extra work in Ex3.

How does the FilePlayerAlgorithm plays?

As in Ex1. But based on the classes and flow described above.
Files should be retrieved by the names and location as specified in Ex1.

Should we support a game with two AutoPlayerAlgorithms?

Yes. It should work with two instances of class AutoPlayerAlgorithm. The pieces positioning should not be the same - it can be random or based on player number, or both.

What do we do with all the abstract classes that are presented?

You implement the abstract class according to the documentation without adding any additional method or data member. Then you inherit from them with your own actual type. But you keep your code to work well with the interface (abstract class).

Can we add data members and member functions to the abstract classes?

No you cannot.
In Ex3 we will run your AutoPlayerAlgorithm with our implementation of all the abstract classes from the Game part, so you cannot rely on anything that is not in the interface. As for your own inherited classes - anything that you add in the inherited class is internal and cannot be assumed by the flow. You need something special for your AutoPlayerAlgorithm? Put it into an additional class which is not one of those which we described.

How sophisticated our AutoPlayerAlgorithms should be?

Not too much. But:

- It’s supposed not to make illegal moves (that will cause it to lose for an illegal move)
- It’s supposed to move with different tools, not all the time with the same tool
- It’s supposed to have *some wisdom* (of your choice), **for example**:
    - Try to remember the actual representation of the tools that were uncovered
    - If we know the actual representation of all tools except the flag - we know where is the flag and can try to conquer it
    - If we know the actual representation of an opponent's tool and we have a stronger tool near it, and this is our turn - we may want to attack
    - If we know the actual representation of an opponent's tool and we have a weaker tool near it, we run
    - If a joker is about to get into a fight with a tool that was uncovered (and let’s say that you know the other is not a joker), turn your joker (if you can, in the previous move before a fight) to something that will win the fight
- Above “wisdom” is **just a suggestion** - you can have your own
- In Ex3 we will have a tournament between your AutoPlayerAlgorithms, with a bonus (and TAU CS Pride and Glory Screenshot) for the winners.
- It’s OK to have a “dumb” algorithm as long as it plays legal moves and with different tools and tries somehow to remember the actual representation revealed to it and do something with it, even if minor.

Can we add or remove ‘const’ qualifier from the methods of the abstract classes?
No, it will break the signatures.

Can we add “include” to the abstract classes?
Only to <vector> and <memory>
Any additional include may make the users assume that there is such an include and it wouldn’t be added in our version of the abstract class, which may cause you problems.

What is unique_ptr?
We will talk about unique_ptr in class.
In the meanwhile you can peep at the slides of Smart Pointers lesson.