VIETNAM NATIONAL UNIVERSITY OF HO CHI MINH CITY
DEPARTMENT OF INFORMATION TECHNOLOGY
SUBJECT: **OBJECT-ORIENTED PROGRAMMING**

GUIDELINE

CARO

HCM CITY, 25th June 2020

CONTENTS

1 Introduction

In this part, we use some class techniques and basic data structure to build a simple caro. To complete this project, we need to know some basic knowledge: object-oriented design, file processing, two-dimensional array… This guideline only helps students to build the basic game. You should research many problems and do your best.

2 Screenplay

Firstly, Game shows the board, and user controls the keys 'W', 'A', 'S', 'D' to move. When, user keys the 'enter', the mark 'X' or 'O' will be appeared with the appropriate turn.

When one of the players wins with caro rule, the screen prints a line showing the winner. Then, the program ask the players if they want to continue. If the players choose "y", the program will restart. Otherwise, we exit the game.

When there is no place in caro board, the game show "two players draws". Then, the program ask the players if they want to continue or not.

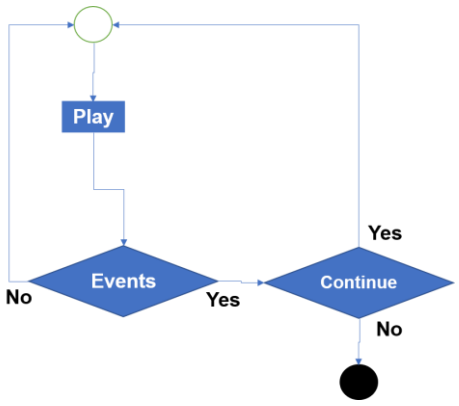

Figure 1: Screen-play diagram

3 Some steps to build the game

In this part, we go through all the steps to build the basic game. Note that this is only the guideline, group of students can make your own design.

Step 1: In this step, we create a common class called _Common. Actually, this is a pseudo-class includes some common functions. Function "fixConsoleWindow" fixes the screen with reasonable size. This helps to prevent the players from resizing the screen,

and make our computation be incorrect. Furthermore, function “gotoXY” helps us to move the cursor to the appropriate position.

Lines	
1	`class _Common{`
2	`   public:`
3	`      static void fixConsoleWindow();`
4	`      static void gotoXY(int, int);`
5	`};`
6	`void _Common::gotoXY(int pX, int pY) {`
7	`   COORD coord;`
8	`   coord.X = pX;`
9	`   coord.Y = pY;`
10	`   SetConsoleCursorPosition(GetStdHandle(STD_OUTPUT_HANDLE), coord);`
11	`}`
12	`void _Common::fixConsoleWindow() {`
13	`   HWND consoleWindow = GetConsoleWindow();`
14	`   LONG style = GetWindowLong(consoleWindow, GWL_STYLE);`
15	`   style = style & ~(WS_MAXIMIZEBOX) & ~(WS_THICKFRAME);`
16	`   SetWindowLong(consoleWindow, GWL_STYLE, style);`
17	`}`

In above code, HWND is a “special handle” to Console window. To work with these graphical object, we need such that type. Flag GWL_STYLE is a mark for function “GetWindowLong” to get the features of Console window. Returned-type of “GetWindowLong” is a long, and we edit at line 4. The goal is to disable the button “maximize” and ask the players not to resize the window. Next, we use “SetWindowLong” to assign the edited-result back to. In board, there are many positions we want to print there, so we need to move to all the positions of console window. In this code, we use struct _COORD (COORD). This is a structure reserved for console coordination processing. We assign x-value and y-value to coord variable, then set the position with “SetConsoleCursorPosition”. Note: this function needs a main object which is console window, so we also need a handle to this window (HANDLE could be a void*). We have it by calling “GetStdHandle” with the flag parameter called STD_OUTPUT_HANDLE.

Step 2: Next, we build a class used to support the players tick the board. We let each position be a point including x and y values. Furthermore, to differentiate ‘X’ and ‘O’, we add a variable for signaling. For example, variable _check, with _check = -1 to print ‘X’ and _check = 1 to print ‘O’.

Lines	
1	`class _Point{`
2	`    int _x, _y, _check;`
3	`    public:`
4	`        _Point();`
5	`        _Point(int, int);`
6	`        bool setCheck(int);`
7	`        int getX();`
8	`        int getY();`
9	`        int getCheck();`
10	`        void setX(int);`
11	`        void setY(int);`
12	`};`
13	`_Point::_Point(){_x = _y = _check = 0;}`
14	`_Point::_Point(int pX, int pY){`
15	`    _x = pX; _y = pY;`
16	`    _check = 0;`
17	`}`
18	`int _Point::getX(){return _x;}`
19	`int _Point::getY(){return _y;}`
20	`int _Point::getCheck(){return _check;}`
21	`void _Point::setX(int pX) {_x = pX;}`
22	`void _Point::setY(int pY) {_y = pY;}`
23	`bool _Point::setCheck(int pCheck){`
24	`    if(pCheck == -1 || pCheck == 1 || pCheck == 0) {`
25	`        _check = pCheck;`
26	`        return true;`
27	`    }`
28	`    return false;`
29	`}`

Step 3: Next, we create the class _Board. In this class, we have some necessary properties, such as boardsize (_size), left-top point of boardgame (_left & _top). Finally, we have a level-2 pointer called_pArr with _Point type. For convenience, we create a set of get functions (get…), for example getSize, getLeft, getTop, getXAt(row, col) and getYAt(row, col).

Lines	
1	`class _Board{`
2	`    private:`
3	`        int _size;`
4	`        int _left, _top;`

```
5         _Point** _pArr;
6     public:
7         int getSize();
8         int getLeft();
9         int getTop();
10        int getXAt(int, int);
11        int getYAt(int, int);
12    };
13    int _Board::getSize(){return _size;}
14    int _Board::getLeft(){return _left;}
15    int _Board::getTop(){return _top;}
16    int _Board::getXAt(int i, int j){
17        return _pArr[i][j].getX();
18    }
19    int _Board::getYAt(int i, int j){
20        return _pArr[i][j].getY();
21    }
```

Step 4: Next, we add give the class _Board the constructor and destructor. The constructor's goal is the setting the initial values and allocate the memory for some pointers. The destructor's goal is the freeing the memory allocated by constructor.

```
Lines
1   class _Board{
2      _Board(int, int, int);
3      ~_Board();
4      //…
5   }
6   _Board::_Board(int pSize, int pX, int pY){
7       _size = pSize;
8       _left = pX;
9       _top = pY;
10      _pArr = new _Point*[pSize];
11      for(int i = 0; i < pSize; i++) _pArr[i] = new _Point[pSize];
12  }
13  _Board::~_Board(){
14      for(int i = 0; i < _size; i++) delete[] _pArr[i];
15      delete[] _pArr;
16  }
```

Step 5: Next, we give the class _Board a function called "resetData" to restart the data for 2-D array, and function called "drawBoard" to draw the boardgame

Lines	
1	`class _Board{`
2	`  //…`
3	`  void resetData();`
4	`  void drawBoard();`
5	`};`
6	`void _Board::resetData() {`
7	`if(_size == 0) return; // Firstly calling constructor before calling resetData`
8	`  for(int i = 0 ; i < _size ; i++){`
9	`    for(int j = 0 ; j < _size ; j++){`
10	`      _pArr[i][j].setX(4 * j + _left + 2); // x-value of boardgame`
11	`      _pArr[i][j].setY(2 * i + _top + 1); // y-value of boardgame`
12	`      _pArr[i][j].setCheck(0);`
13	`    }`
14	`  }`
15	`}`
16	`void _Board::drawBoard(){`
17	`  if(_pArr == NULL) return; // firstly call constructor`
18	`  for(int i = 0; i <= _size ; i++){`
19	`    for(int j = 0; j <= _size ; j++){`
20	`      _Common::gotoXY(_left + 4 * i, _top + 2 * j);`
21	`      printf(".");`
22	`    }`
23	`  }`
24	`  _Common::gotoXY(_pArr[0][0].getX(), _pArr[0][0].getY());//move to the 1st cell`
25	`}`

Step 6: Finally, we add function called “checkboard” to update the property _check at the position the player enters. Depending on turn, the variable called _check is set to 1 or -1. Furthermore, we add function called “testBoard” to check the result (win, lose with caro rule). Below code is just a demo, you implement such that the function returns -1 (the 1st wins), 0 (draw), 1 (the 2nd wins) or 2 (no one win).

Lines	
1	`class _Board{`
2	`  //…`
3	`  int checkboard(int, int, bool);`
4	`  int testBoard();`
5	`};`
6	`int _Board::checkBoard(int pX, int pY, bool pTurn){`
7	`  for(int i = 0; i < _size; i++){`
8	`    for(int j = 0; j < _size; j++){`

```
9       if(_pArr[i][j].getX() == pX && _pArr[i][j].getY() == pY && _pArr[i][j].getCheck() == 0){
10        if(pTurn) _pArr[i][j].setCheck(-1);// If current turn is true: c = -1
11        else _pArr[i][j].setCheck(1);// If current turn is false: c = 1
12        return _pArr[i][j].getCheck();
13      }
14    }
15  }
16  return 0;
17 }
18 int _Board::testBoard(){return 0;} // Defaultly returns 'Draw'
```

Step 7: Next, we create class _Game to execute the game. Class _Game includes a boardgame with _Board type, variable called _turn with bool type to represent two turns, current coordination (_x, _y) of cursor, variable called _command receiving the input-key from the players, and the variable called _loop to check the finish of game.

```
Lines
1  class _Game{
2      _Board* _b;// a board game
3      bool _turn;// turn: true for the 1st player and false for the 2nd player
4      int _x, _y; // current position of cursor
5      int _command; // input-key from the players
6      bool _loop; // decision bool variable to exit game or not
7      public:
8          _Game(int, int, int);
9          ~_Game();
10 };
11 Game::_Game(int pSize, int pLeft, int pTop){
12     _b = new _Board(pSize, pLeft, pTop);
13     _loop = _turn = true;
14     _command = -1; // Assign turn and default key
15     _x = pLeft; _y = pTop;
16 }
17 _Game::~_Game(){delete _b;}
```

The constructor initializes the simple variables and calls the _Board object's constructor to allocate the memory for matrix. Destructor will free the object called _b (because declaring the pointer, we must explicitly call operator delete).

Step 8: We give the functions used to provide which key the players tick (variable called _command) or check if the players go out or not (variable called _loop).

```
Lines
```

```
1  class _Game{
2      // …
3      public:
4          _Game(int, int, int);
5          ~_Game();
6          int getCommand();
7          bool isContinue();
8          char waitKeyBoard(); // Receiving keyboard from players
9          char askContinue();
10 };
11 int _Game::getCommand(){ return _command;}
12 bool _Game::isContinue(){ return _loop;}
13 char _Game::waitKeyBoard(){
14   _command = toupper(getch());
15   return _command;
16 }
17 char _Game::askContinue(){
18 _Common::gotoXY(0, _b->getYAt(_b->getSize() - 1, _b->getSize() - 1) + 4);
19   return waitKeyBoard();
20 }
```

In addition to “getCommand” and “isContinue”, we add two convenient functions called “waitKeyBoard” to receive the input-key from the players and “askContinue” to ask if the players to continue or not when they finish the game or hit “ECS”. Note: actually, “askContinue” and “waitKeyBoard” are different in that: in “askContinue” we need to move to the reasonable position where the player hits the key, but in “waitKeyBoard” the player hit the key in the area of boardgame.

Step 9: We need the function called startGame and exitGame to start and finish

```
Lines
1  class _Game{
2      // …
3      public:
4          //…
5          void startGame(); // Function to start the game
6          void exitGame(); // Function to exit the game
7  };
8  void _Game::startGame() {
9    system("cls");
10   _b->resetData(); // Setting the original data
11   _b->drawBoard(); // Draw boad
```

```
12    _x = _b->getXAt(0, 0);
13     _y = _b->getYAt(0, 0);
14  }
15  void _Game::exitGame() {
16    system("cls");
17    //Maybe save game before stopping
18    _loop = false;
19  }
```

Step 10: In Caro, we need to process when the players hit 'enter' at the position of the boardgame, then we check the result of that turn leading to win/lose/draw/nothing. Also, we need to move up/down/left/right when the players hit the moving keys.

```
Lines
1   class _Game{
2       // …
3       public:
4           //…
5           int processFinish();
7           bool processCheckBoard();
8           void moveRight();
9           void moveLeft();
10          void moveUp();
11          void moveDown();
12  };
14  bool _Game::processCheckBoard() {
15    switch(_b->checkBoard(_x, _y, _turn)){
16      case -1:
17        printf("X");
18        break;
19      case 1:
20        printf("O");
22        break;
23      case 0: return false; // Tick the cell marked
24    }
25    return true;
26  }
27  int _Game::processFinish() {
29    // Move to the reasonable place to print string win/lose/draw
30    _Common::gotoXY(0, _b->getYAt(_b->getSize() - 1, _b->getSize() - 1) + 2);
31    int pWhoWin = _b->testBoard();
32    switch(pWhoWin){
```

```
33      case -1:
34        printf("The player %d won and the player %d lost\n", true, false);
35        break;
36      case 1:
37        printf("The player %d won and the player %d lost\n", false, true);
38        break;
39      case 0:
40        printf("The player %d draw with the player %d\n", false, true);
41        break;
42      case 2:
43        _turn = !_turn; // change turn if nothing happen
44    }
45    _Common::gotoXY(_x, _y);// Return the current position of cursor
46    return pWhoWin;
47  }
48  void _Game::moveRight() {
49    if (_x < _b->getXAt(_b->getSize() - 1, _b->getSize() - 1)){
50      _x += 4;
51      _Common::gotoXY(_x, _y);
52    }
53  }
54  void _Game::moveLeft() {
55    if (_x > _b->getXAt(0, 0)) {
56      _x -= 4;
57      _Common::gotoXY(_x, _y);
58    }
59  }
60  void _Game::moveDown() {
61    if (_y < _b->getYAt(_b->getSize() - 1, _b->getSize() - 1)){
62      _y += 2;
63      _Common::gotoXY(_x, _y);
64    }
65  }
66  void _Game::moveUp() {
67    if (_y > _b->getYAt(0, 0)) {
68      _y -= 2;
69      _Common::gotoXY(_x, _y);
70    }
71  }
```

If we go out of the area of boardgame, we do not process (note: we access the position through the board of object called _b); otherwise, we move the cursor to the new position.

Step 11: Finally, we create main function to receive the input-key from the users.

Lines	
1	`void main(){`
2	`  _Common::fixConsoleWindow();`
3	`  _Game g(BOARD_SIZE, LEFT, TOP); // You yourself define some constants`
4	`  g.startGame();`
5	`  while(g.isContinue()){`
6	`    g.waitKeyBoard();`
7	`    if (g.getCommand() == 27) g.exitGame();`
8	`    else {`
9	`      switch(g.getCommand()){`
10	`        case 'A':`
11	`          g.moveLeft();`
12	`          break;`
13	`        case 'W':`
14	`          g.moveUp();`
15	`          break;`
16	`        case 'S':`
17	`          g.moveDown();`
18	`          break;`
19	`        case 'D':`
20	`          g.moveRight();`
21	`          break;`
22	`        case 13:`
23	`          //Mark the board, then check and process win/lose/draw/continue`
24	`          if(g.processCheckBoard()){`
25	`            switch(g.processFinish()){`
26	`              case -1: case 1: case 0:`
27	`                if(g.askContinue() != 'Y') g.exitGame();`
28	`                else g.startGame();`
29	`            }`
30	`          }`
31	`      }`
32	`    }`
33	`  }`
34	`}`

In main function, we fix the screen to prevent the players from resizing. Then, we initialize the object called _Game and call startGame() to prepare the data for playing. Main() **continuously** waits for the players, then reasonably responds to input-key from the players. Object called _Game controls all the flows.

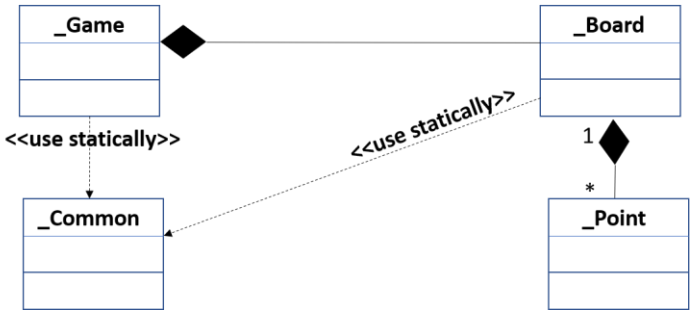

Figure 2: Class-diagram of game

4 REQUIREMENTS

In this guideline, we lack some basic features

4.1 Save/load – 2m

In this guideline, we cannot save and load the game. We need two features. When the players hit 'L', we show the line requesting the players provide the filename to save. When the players hit 'T', we show the line requesting the players provide the file to load.

4.2 Recognize win/lose/draw – 2m

Need to provide the caro rule to know win/lose/draw.

4.3 Provide animation of win/lose/draw – 2m

In this guideline, we only print the simple line showing win/lose/draw. Provide the vivid animation for this event.

4.4 Creating playing interface – 1.5m

When two players tick the board, we print some statistics of two players, for example, counting the number of ticking for each player … You yourself organize the interface such that it is clear and pretty.

4.5 Provide the main menu – 1.5m

When coming to boardgame, we print the menu game, for example, "New Game", "Load Game", "Settings", … So, this helps the players to easily choose actions they want.

4.6 Playing with machine (Alpha – beta pruning) – 1m

Allow the players to choose “play with machine”, and the players can choose the level, for example, easy-level if we allow machine to randomize the position, and hard-level if we apply the alpha – beta pruning.