Cairo University
Faculty of Engineering
Computer Engineering Department
CMP N103 Fall 2019

CMPN103
Programming Techniques
Project (Phase 2)

Game

Main Classes

You are given a ***code framework*** where we have ***partially*** written code of some of the project classes. For the graphical user interface (GUI), we have integrated an open-source ***graphics library*** that you will use to easily handle GUI (e.g. drawing grid on the screen and reading the coordinates of mouse clicks …etc.).

You should **stick to** the **given design** (i.e. hierarchy of classes and the specified job of each class) and complete the given framework by either: extending some classes or inheriting from some classes (or even creating new base classes).

Below is the class diagram then a description for the basic classes.

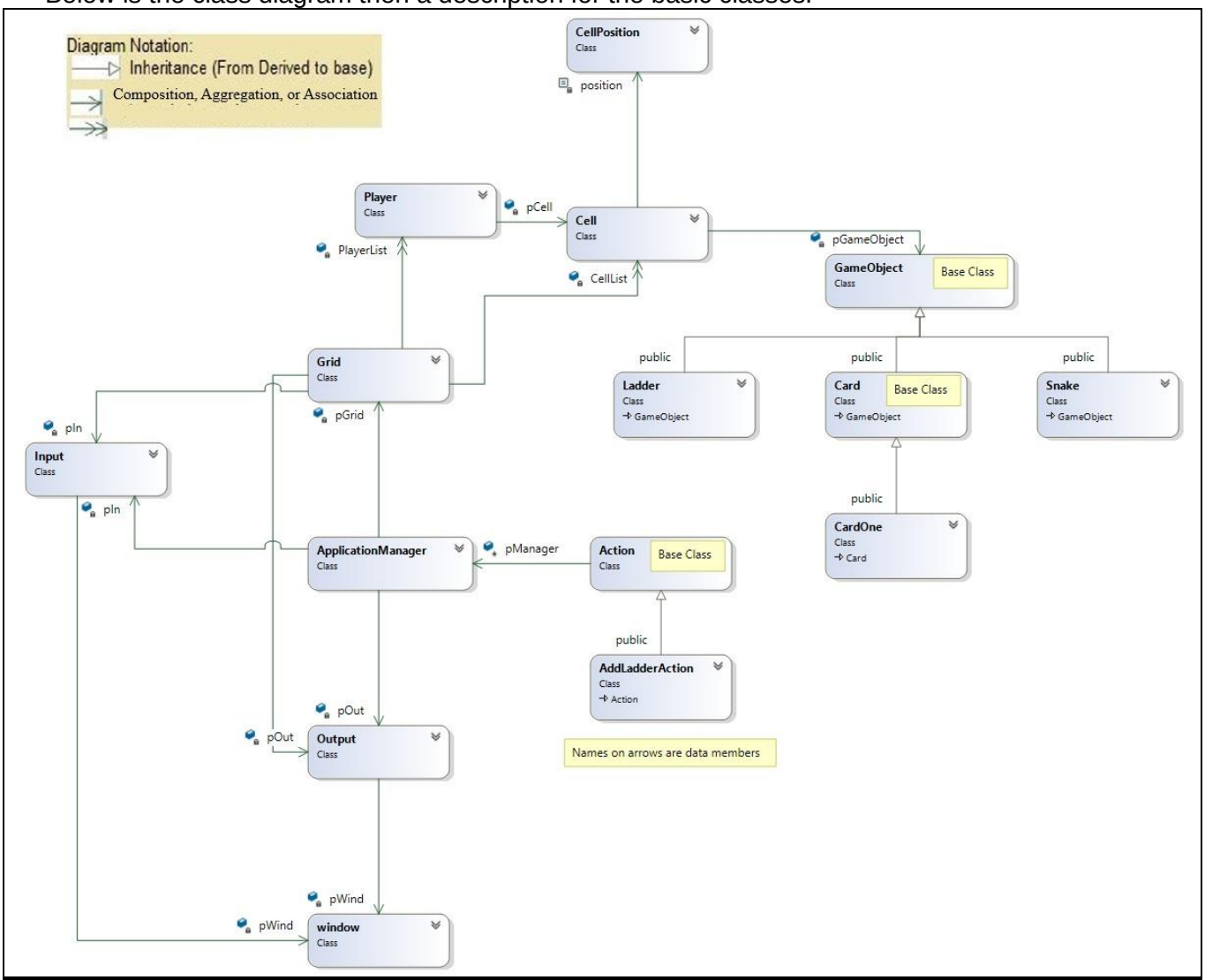

Figure 1 – Class Diagram

Input Class:
ALL user inputs must come through this class. If any other class needs to read any input, it must call a member function of the input class. You should add suitable member functions for different types of inputs.

Output Class:
This class is responsible for **ALL** GUI outputs. It is responsible for toolbar and status bar creation, grid and game objects drawing, and for messages printing to the user. If any other class needs to

make any output, it must call a member function of the output class. You should add suitable member functions for different types of outputs.

Notes: - No input or output is done through the console. All must be done through the **GUI window**.
- Input and Output classes are the **ONLY** classes that have **access to GUI library**.

Action Class:
Each operation of the design or play mode (corresponding to each icon) must have a ***corresponding action class***. This is the base class for all types of actions (operations) to be supported by the application. To add a new action, you must **inherit** it from this class. Then you should override virtual functions of class **Action** (ReadActionParameters and Execute functions). Each action may have action parameters. **Action parameters** are the parameters needed to be read from the user, after choosing the action icon, to be able to execute the action. You can also add more details or functions for the class Action itself if needed.

GameObject Class:
This is the **base class** for all types of game objects (ladders, snakes or cards). Each game object type must **inherit** from this class (GameObject), then you should **override** its virtual functions **(**e.g. Draw, Save, …etc.). You can also add more details or functions for class GameObject itself if needed. One of the important **virtual** functions of GameObject class is **Apply()** function. It applies the GameObject's effect on the passed player, for example: **in the Ladder GameObject**, Apply() moves the player to the end cell of the ladder, and **in the Card GameObject**, Apply() makes the effect of this specific card type on the player.

CellPosition Class:
This class represents the cell position in the grid by having two data members called vCell and hCell. This class does NOT deal with real coordinates, it deals with the vCell, hCell and cellNum instead.

Cell Class:
This class represents the cells of the grid. It holds an object of CellPosition class. It holds a pointer to the game object occupying the cell. For snakes and ladders, they are set as the game objects of only their **start cell**, so for example, if a ladder starts at cell 1 and ends at cell 12, then the game object pointer of cell 1 will point to that ladder, however, the game object pointer of the end cell 12 will NOT point to that ladder.

Grid Class:
This class represents the game grid (the vertical and horizontal cells that game objects can move in). It contains a **2-Dimensional Array of Pointers to Cells** (called **CellList**). This list keeps track of the grid cells. This is the **ONLY** class that can operate directly on the **CellList** which means that NO other classes can Get this List or a copy of it and operate directly on it. It also includes an array of 4 Player pointers, **PlayerList**, that represents the four players of the game. In addition, the Grid class has pointers to the Input and the Output class objects.

The Grid class responsibility is to maintain the **CellList** (e.g. by providing public functions like `AddObjectToCell` and `RemoveObjectFromCell`, …etc.) but **it must NOT make any further logic** in its functions. There may be functions inside the Grid class like SaveAll that loops on the 2D array and **blindly** call the virtual function Save for each GameObject pointer in each cell, but the functions of the Grid class only loop and call functions (do NOT make any further logic).

ApplicationManager Class:
This is the ***maestro*** class that controls everything in the application. It creates the Grid, Input and Output objects and have pointers of them as data members. Its job is to create an object of the action class corresponding to the action chosen by the user then executes it. **ApplicationManager** just manages or instructs other classes to do their jobs (**NOT** to do other classes' jobs).

Player Class:
Each player object contains: player number, step count (initially 0) and wallet (initially 100). It also contains a pointer to the cell the player currently occupying. Add any other needed data members. One of the important functions in class Player is function **Move(…)** which makes the player move with the passed dice number.

Ladder Class:
This class represents ladders and their location. Ladders also are types of game objects.

Snake Class:
This class represents snakes and their location. Snakes also are types of game objects.

Card Class:
This is the **base class** for all types of card items. To create a new card type (CardOne for example), you must **inherit** it from this class. Then you should override virtual functions of class **Card.** You can also add more details or functions for the class Card itself if needed. One of the **virtual** functions of class Card is **ReadCardParameters ( )** which reads the parameters of a particular type of cards (for example, the wallet amount to decrease in CardOne)

Example Scenarios

Example Scenario 1: Add Ladder Action (in Design Mode)

Here is an example scenario for **adding a ladder** in the Grid in Design Mode. It is performed through the four steps mentioned in '**Appendix A - implementation guidelines**' section. These four steps are in the "main" function of phase 2 code. You **must NOT** change the "main" function of **phase 2**.

The 4 steps are as follows:

Step 1: Get user input

1- The **ApplicationManager** calls the **Input** class and waits for user action.
2- The user clicks on the **"Add Ladder" icon** in the tool bar to add a ladder.
3- The **Input** class checks the area of the click and recognizes that it is a "add ladder" operation. It returns **`ADD_LADDER`** (an **"enum"** value representing the **ActionType**) to the manager.

Step 2: Create a suitable action

1- **ApplicationManager::ExecuteAction(ActionType)** is called to create an action object of type **AddLadderAction** class:

```
Action* pAct = NULL;
// in a switch case on the passed ActionType
case ADD_LADDER:      pAct = new AddLadderAction(this);
```

Step 3: Execute the action

1- **ApplicationManager::ExecuteAction(…)** calls the virtual **AddLadderAction::Execute( )**

```
pAct->Execute(); // Execute the AddLadderAction object
```

2- AddLadderAction::Execute( )

a. Calls **AddLadderAction::ReadActionParameters( )** which calls function **GetCellClicked()** twice from the **Input** class to get the action parameters of **AddLadderAction** (i.e. the start and end cell positions, **startPos** and **endPos**).

Notice that when **AddLadderAction** wants to print messages to the user on the status bar, it calls some functions from the **Output** class.

b. Creates (allocates) an object of class **Ladder**:

```
Ladder * pLadder = new Ladder(startPos, endPos);
```

c. Asks the **Grid** to set the created ladder object to the GameObject pointer of its Cell (the start cell of the ladder) by calling **Grid::AddObjectToCell (…)** function and passing the new ladder to it:

```
pGrid->AddObjectToCell(pLadder);
```

Function Grid:: AddObjectToCell (`GameObject * pNewObject`) will do the following (Note: the passed pNewObject now points to the newly created ladder object):

1. Get the cell position of the ladder object (which represents its **start** cell):

```
CellPosition pos = pNewObject->GetPosition();
```

2. **Set the passed ladder object to the GameObject pointer of its start cell in the CellList:**

```
CellList[pos.VCell()][pos.HCell()]->SetGameObject(pNewObject);
```

Step 4: Update Interface

1- ApplicationManager::UpdateInterface( ) calls **Grid:: UpdateInterface ( )**

2- The **UpdateInterface()** of the **Grid** class *in the Design Mode* redraws all cells with their game objects (cards, ladders and snakes) using the following steps:

a. It iterates on each Cell in the CellList of Grid class and Call the following function to redraw each cell and its card if it contains a card:

```
CellList[i][j]->DrawCellOrCard(pOut);
```

b. It iterates again on each Cell in the CellList and Call the following function to redraw the ladder/snake of each cell:

```
CellList[i][j]->DrawLadderOrSnake(pOut);
```

☐ **Notes:**

a. **UpdateInterface()** draws all cells with cards first before the ladders/snakes to avoid drawing a cell above a drawn ladder/snake.

b. **DrawCellOrCard(…)** function of class **Cell** calls the **Draw(…) virtual function** of the **GameObject** class if the cell contains a card game object. Similarly, the **DrawLadderOrSnake(…)** function of class **Cell** calls the **Draw(…) virtual function** of class **GameObject** if the cell contains a ladder or a snake game object.

c. The **Draw(…)** function of **GameObject** class is a **pure virtual function** which is overridden in class **Ladder** to call **Output::DrawLadder(…)** of class Output that is responsible for Drawing the ladders in the Interface.

Example Scenario 2: Save Grid Action (in Create-Grid Mode)

- **In General, Save/Open** has NO relation to the **Input** or **Output** classes. They save/open grids to/from **files** not the graphical window.

- Here we explain the calling sequence in the execute of '**SaveGridAction'** action as an example. **Note** the **responsibility of each class** and how each class does only its job or responsibility.

- There is a **save function** in **Grid** class and in each type of **GameObject** but each function performs a different job:

1. **GameObject :: Save (… , Type)**
   It is a **virtual** function in **GameObject** class. Each class derived from GameObject class should **override** it with its own implementation to save itself because each GameObject type has different information and hence a different way or logic to save itself. The function takes a "*Type" parameter* which could be an *enum* or an *integer* that represents the GameObject type that should be save (either card, ladder or snake). So, each GameObject class type will check the "Type" parameter sent to the function. If it is the same type of the class, it will save its information to the file, otherwise, return without saving.

   This "**Type**" parameter is sent because as shown in the "**File Format**" section mentioned below, all **ladders** should be saved first, then all **snakes** then all **cards**.

2. **Grid :: SaveAll (… , Type)**
   It is the function responsible for **calling** the **GameObject :: Save(…, Type)** function for the GameObject of each cell (if any) in the Grid's **CellList** because **Grid class** is the only class that has **CellList** and no one else can access it. Note that it only gets the GameObject of each cell and calls function save of the game object; **ONLY calling without making the save logic itself** (not the responsibility of **Grid** but the responsibility of each game object class). **This note is important and has a huge grade percentage**.

   Note: dynamic_cast is NOT needed in this function because polymorphism will automatically call the Save function of the correct object type for each GameObject pointer in the list.

3. **SaveGridAction :: Execute( )**
   It does the following:
   - first reads action parameters (i.e. the filename)
   - then opens the file
   - and calls **Grid::SaveAll (…, LaddersType)** to save all ladders
   - and calls **Grid::SaveAll (…, SnakesType)** to same all snakes
   - and calls **Grid::SaveAll (…, CardsType)** to same all snakes
   - then closes the file

   Note: if any information is available for SaveGridAction class without breaking class responsibilities, it should write it to the file by itself.

   Important Note: Don't abuse the "*Type*" enum that you will create for this function. Whenever virtual functions and polymorphism could be applied, apply them. If any use of Type enum in your project will replace virtual functions and break class responsibilities, this will be grade-penalized.

File Format

The "Grid" File Format:

Your application should be able to **save/open a grid** to/from a simple text file. In this section, the file format is described together with an example.

- **File Format**

```
Number_of_Ladders(n1)
Ladder_1_start_cell        Ladder_1_end_cell
Ladder_2_start_cell        Ladder_2_end_cell
……..
Laddern_1_start_cell       Laddern_1_end_cell
Number_of_Snakes(n2)
Snake_1_start_cell         Snake_1_end_cell
Snake_2_start_cell         Snake_2_end_cell
……..
Snake_n2_start_cell        Snake_n2_end_cell
Number_of_Cards(n3)
Card_1_type    Card_1_Cell    Card_1_parameter_1(if any)    Card_1_parameter_2(if any)
……..
Card_n3_type   Card_n3_Cell   Card_n3_parameter_1(if any)   Card_n3_parameter_2(if any)
```

- **Example:** The grid file looks like that (*comments in green are just for explaining the example*)

```
2//number of ladders
13  57// start cell (13), end cell (57)
6   39// start cell (6), end cell (39)
2  //number of snakes
97  31// start cell (97), end cell (31)
92  48// start cell (92), end cell (48)
5 //number of cards
1   2 10// card of type 1 in cell 2. Decrement amount is 10
1   20 12// card of type 1 in cell 20. Decrement amount is 12
10  25 20 5// card of type 10 in cell 25. Cell price is 20 and fees for the card are 5
3   50// card of type 3 in cell 50 (no parameters for card 3)
10  80// card of type 10 in cell 80. Cell price is 20 and fees for the card are 5. Since each card
of type 10 has the same value and fees, the parameters should only be read once
```

Notes:

- ❑ You are allowed to modify this file format if necessary **but after instructor approval**.
- ❑ **The "Open" Action**:
  For lines in the above file, For Example for the ladders: the **OpenAction** first **creates** (allocates) an object of that type of ladder. Then, it **calls** **GameObject::Read** virtual function that can be overridden in the class of each game object type to make the object load its data from the opened file by itself (its job). Then, it **calls** **Grid::AddObjectToCell** to add the created ladder object to a cell in the **CellList**.

Project Phase2

Partially-implemented code frameworks for Phase 2 are given to you to complete them.

For **fast navigation** in the given code in **Visual Studio**, you may need the following:

- **F12 (go to definition):** to go to definition of functions (code body), variables, …etc.
- **"Ctrl" then "Minus":** to return to the previous location of the cursor.

□ Phase 2 (Project Delivery)

In this phase, the completed **I/O classes**, **DEFS.h** and **UI_Info.h** (without phase 1 test code) should be added to the project **framework code** given for phase 2 (and the **images folder** should be copies too) and the remaining classes of phase 2 should be implemented. Start by implementing the base classes then move to derived classes.

In the given code of phase 2, AddLadderAction of Create-Grid mode is almost fully implemented (needs only validation). In addition, some base classes are partially implemented with some guiding comments to follow.

You are required to:

- **Complete the Implementation of functions marked with ///TODO comment** as it is described in the comments. These are some useful functions that you can use in implementation of phase 2.
- **Add all the classes mentioned in the "Main Classes Section"** with full implementation of their functions and finalize the project to perform all the operations mentioned in the project documents in the 2 modes.
- **You may need to add more classes to make the code more organized and object-oriented** but you are NOT allowed to change the initial classes' design, hierarchy and responsibilities we mentioned in the "**Main Classes Section**".

Phase 2 Deliverables*:*

(1) **Workload division**: a **printed page** containing team information and a table that contains members' names and the actions each member has implemented.
(2) An email (or on elearn – will be announced later) that contains the following:
    a. **ID.txt file**. (Information about the team: names, IDs, team email)
    b. **Workload division** (described in the previous point)
    c. **The project code and resources files** (images, saved files, …etc.).
    d. **Sample grid files:** at least **three** different grids. For each grid, provide:
        i. Grid text file (created by save operation)
        ii. Grid screenshot for the grid generated by your program
        iii. Screenshot of one action of *design* mode

Note that Each project phase should be sent first **by mail/elearn** (same time for all groups). **No modifications are allowed after the mail delivery.** After that the face-to-face **discussion** of the project will be held. The week day of sending each project phase by mail/elearn and the discussion schedule of each phase will be announced later.

Phase 2 Evaluation Criteria

Percentages are added according to the **difficulty** of the task, so to divide the project load **equally** on team members, each member should take actions that their percentages add up to about **25%** of phase 2 (if 4 students in the team).

Toolbar Operations [40%]

- ❑ Each operation percentage is mentioned below.
- ❑ **[32%]** Design Mode
  - ❑ **[3%]** Add Ladder & Add Snake
  - ❑ **[8%]** Add Card and Edit Card
  - ❑ **[7%]** Copy & Cut and Paste Card
  - ❑ **[2%]** Delete Game Object
  - ❑ **[11%]** Save & Open Grid
  - ❑ **[1%]** Switch to play mode and Switch back to design mode (this icon in PlayMode)
- ❑ **[8%]** Play Mode
  - ❑ **[6%]** Roll Dice & Input Dice Value
    - This percentage includes ONLY what will be written inside the action classes of Roll Dice and Input Dice Value; mainly the **function calls** written inside them (not the implementation of each called function).
  - ❑ **[2%]** New game

Game Logic [55%]

- ❑ Any percentage mentioned next to a Game Object means: handing all the Logic of this object in the game:
  - ❑ **[5%]** Player movement and adding money after 3 turns
  - ❑ **[8%]** Taking Ladders & Snakes
  - ❑ **[42%]** Cards
    - **[4%]** Card 1 & 2
    - **[8%]** Card 3 & 4
    - **[6%]** Card 5 & 6
    - **[6%]** Card 7 & 8
    - **[3%]** Card 9
    - **[15%]** Cards [10-14]

- ❑ **Note: it is recommended that the student who will work on Add & Edit card work also on cards [10-14]**
- ❑ Note: we tried to combine similar actions/objects together in the same percentage because they share similar logic, so it is recommended not to separate them between more than one person (for example, Card 1 & 2 are written together, so give both of them to one person).

Code Organization & Style [5%]

- ❑ Every class in .h and .cpp files
- ❑ Variable naming
- ❑ Indentation & Comments
- ❑ Note: this percentage is divided between team members (Each should make the code organization of his code part)

Bonus [10%]

- ❑ Each player has 4 special attacks that he can use throughout the game.
- ❑ A player can choose during his "recharge the wallet" turn to launch a special attack instead of recharging his wallet. (If he is not denied from rolling dice)
  - ❑ A message is shown, "Do you wish to launch a special attack instead of recharging? y/n"
- ❑ **[1%]** Each player can use two unique special attacks at most throughout the game. This means a single player cannot use the same special attack twice.
- ❑ If a player chooses to launch a special attack, he is prompted to choose his special attack type
- ❑ The 4 special attacks are:
  - ❑ **[2%]** Ice: Choose a player to prevent from rolling the next turn.
  - ❑ **[3%]** Fire: Choose a player to burn. Deduct 20 coins from his wallet for his next 3 turns (total 60).
  - ❑ **[3%]** Poison: Choose a player to poison. For 5 turns, deduct 1 number from his dice roll.
  - ❑ **[1%]** Lightning: Deduct 20 coins from all other players.

General Evaluation Criteria for any Operation:

1. **Compilation Errors** → **MINUS 50%** of Operation Grade
   - ❑ The remaining 50% will be on logic and object-oriented concepts (see **point no. 3**)
2. **Not Running** (runtime error in its **basic functionality**) → **MINUS 40%** of Operation Grade
   - ❑ The remaining 60% will be on logic and object-oriented concepts (see **point no. 3**)
   - ❑ If we found runtime errors but in corner (not basic) cases, that's will be part of the grade but not the whole 40%.
3. **Missing Object-Oriented Concepts** → **MINUS 30%** of Operation Grade
   - ❑ **Separate class** for each item and action
   - ❑ Each class does its **job**. No class is performing the job of another class.
   - ❑ **Polymorphism:** use of pointers and virtual functions
   - ❑ **See the "Implementation Guidelines" in the Appendix which contains all the common mistakes that violates object-oriented concepts.**
4. **For each corner case** that is not working → **MINUS 10% to 20%** of the Operation Grade according to instruction evaluation.

Note: The code of any operation does NOT compensate for the absence of any other operation.

Individuals Evaluation:

Each member must be responsible for writing some project modules (e.g. some classes or some functions) and must answer some questions showing that he/she understands both the program logic and the implementation details. The work load between team members must be almost equal.

- ✓ The grade of **each student** will be divided as follows:
  - **[70%]** of the student grade is on his individual work (the project part he was responsible for).
  - **[25%]** of the student grade is on integrating his work with the work of ALL students who Finished or nearly finished their project part.
  - **[5%]** of the student grade is on cooperation with other team members and helping them if they have any problems implementing their parts (helping does NOT mean implementing their parts).
- ✓ If one or more members didn't make their work, the other members will NOT be affected **as long as** the students who finished their part integrated all their parts together AND their part can be tested to see the output.
- ✓ If a student couldn't finish his part, try to still integrate it with the other parts to at least take the integration grade.

You should **inform the TAs** before the deadline **with a sufficient time (some weeks before)** if any problems happened between team members to be able to take grading action appropriately.

APPENDIX A

[I] Implementation Guidelines

- **Any user operation is performed in 4 steps:**
  - Get user action type.
  - Create suitable action object for that action type.
  - Execute the action (i.e. function **Action::Execute()** which first calls **ReadActionParameters()** then executes the action).
  - Update Interface which updates the drawings on the window after each executed action

- **Use of Pointers/References**: Nearly all the parameters passed/returned to/from the functions should be pointers/references to be able to exploit **polymorphism** and **virtual** functions. **GameObject** in a cell should be a **Base Pointers (GameObject)** to be able to point to any game object type. Many class members should be pointers for the same reason

- **Classes' responsibilities:** Each class must perform tasks that are related to its responsibilities only. No class performs the tasks of another class. For example, when class **Ladder** needs to draw itself on the GUI, it calls function ***Output::DrawLadder*** because dealing with the GUI window is the responsibly of class **Output**. Similarly, class **Grid** must not contain any logic. It only should call functions. Read the "**main classes**" section to know the responsibility of each class.

- **Abusing Getters:** Don't use getters to get data members of a class to make its job inside another class. This breaks the classes' responsibilities rule. For example, do NOT add in **Grid** class function **CellList()** that gets the 2D array of cells to the other classes to loop on it and use it there. **CellList** and looping on it are the responsibility of the Grid class only.

- **Virtual Functions:** In general, when you find some functionality (e.g. saving) that has different implementation based on each type, you should make it virtual function in the base class and override it in each derived class with its own implementation.
  - A common mistake here is the **abuse** of **dynamic_cast** (or similar implementations like **type** data member) by checking the object type outside the class and perform this class's job there (this job should be inside that class in a virtual member function in it).
  - This does not mean you should never use **dynamic_cast** but do NOT use it in a way that breaks the constraint of class responsibilities or replaces the use of virtual functions.

- **Not all the actions** need to add a corresponding function inside **Grid**. This will make **Grid** perform the responsibility of these actions. However, some actions need to loop on **CellList** (e.g. ***SaveGridAction*** …etc.). In this case only (looping on Game Objects), you can add functions for them in **Grid** that loop on the list and just call functions **without making any further logic**.

- You are not allowed to use **global variables** in your implemented part of the project, use passing variables as function parameters instead. That is better from software engineering point of view.

- You need to get instructor approval before using **friendships.**

[II] Workload Division Guidelines

Workload must be distributed among team members. A first question to the team at the project discussion and evaluation is "who is responsible for what?" An answer like" we all worked together" is a **failure** and will be penalized.

Here is a recommended way to divide the work based on **Actions**

- Divide workload by assigning some actions to each team member. Each member takes an action, should make any needed changes in any class involved in that action then run and try this action and see if it performs its operation correctly then move to another action.
- For example, the member who takes action **'SaveGrid'** should create **'SaveGridAction'** and write the code related to **SaveGridAction** inside **'Grid'** and **'GameObject'** derived classes. Then run and check if the game objects are successfully saved. Don't wait for the whole project to finish to run and test your implemented action.
- It is recommended to give similar actions to the same member because they have similar implementation.
- After finishing and trying few related actions, it's recommended to integrate them with the last integrated version and any subsequent divided action should increment on this project version and so on. We call this ***'Incremental Implementation'***.
- It's recommended to first divide the actions that other actions depend on (e.g. adding and creating grids) then integrate before dividing the rest of the actions.
- In the game logic, you may divide by object type, …etc. Each member takes an object type and totally handles it.