Cairo University
Faculty of Engineering
Computer Engineering Department
CMP 103 & CMP N103

Spring 2017

بسم الله الرحمن الرحيم

"نرفع درجاتٍ من نشاء وفوق كل ذي علم عليم"

Programming Techniques Project Requirements

Paint for Kids

Spring 2017

Introduction

A fancy colorful application is an effective way to teach kids some computer skills. Educational games are another enjoyable way for kids teaching.

In this project (Paint for Kids) we are going to build a simple application that enables kids draw fancy shapes and also play some simple games with those shapes. Your application should help a kid draw a number of figures, fill them with different colors, save and load a graph, and so on. The application should provide a game playing mode to teach kids how to differentiate between figures types, colors, sizes … etc.

Your Task:

You are required to write a C++ code for Paint for Kids application. Delivering a working project is NOT enough. You must use ***object oriented programming*** to implement this application and respect the **responsibilities** of each class as specified in the document. See the evaluation criteria section at the end of the document for more information.

NOTE: The application should be designed so that the types of figures and types of operations can be easily extended (*using inheritance*).

The rest of this document describes the details of the application you are required to build.

Project Schedule

Project Phase	***Deliverables***	***Due Week***
Phase 1	**Input-Output Classes**	**Week 9**
Phase 2	**Final Project Delivery**	**Week 15**

NOTE: Number of students per team = **3 to 4 students.**

Table of contents

Main Operations

The application supports 2 mode: **draw mode** and **play mode**. Each mode contains 2 bars: **tool bar** that contains the main operations of the current mode and **status bar** that contains any messages the application will print to the user.

The application should support the following operations (actions) in each mode. Each operation must have an icon in the tool bar. The user should click on the operation icon from the tool bar to choose it.

***Note:** Any **percentage** written below next to any operation is its percentage from **phase 2 grade**. See the evaluation criteria at the end of the document for more details.*

The main operations supported by the application are:

[I] Draw Mode: [60%]

Note: See the "General Operation Constraints" section to know the general constraints that must be applied in any operation.

1- [2.5%] Add Figure: adding a new figure to the list of figures. This includes:
- Adding a new **line**, a new **rectangle**, a new **triangle**, or a new **circle**

2- [2.5%] Change Current Colors/Border: changing the current drawing color, the current filling color, the background color, or the border drawing width. Existing figures will NOT change but any subsequent figures will be drawn using the changed drawing and filling colors (until they're changed again and so on).

3- [5%] Select Figure(s): selecting one or more of the figures.
- User first chooses the "select" icon from the tool bar
- Then clicks inside the figure or on its border (***same for filled and unfilled figures***)
- If the user re-clicks on a selected figure, this will un-select it
- If the user selects one figure,
  - This figure should be highlighted
  - All information about the selected figure should be printed on the status bar.
    For example, the application can print (depending on figure type): the figure ID, start and end points, center, radius, width, height, area...etc.
- If the user selects more than one figure,
  - All selected figure should be highlighted
  - The number of selected figures should be printed on the status bar.

4- [5%] Change Figure(s) Colors/Border Width: changing the drawing, filling colors, or border width for the selected figure(s). You have to select figure(s) first before changing their drawing or filling color.
Note: the color used for highlighting must not appear in the palette of colors.

5- [2.5%] Resize Figure(s): resizing the selected figure(s) by 1/4, 1/2, 2 or 4 times their current size**.**
The user steps are as follows:
- First select the figure(s) you want to resize using the "select" operation
- Then choose "resize" operation from the tool bar
- Ask the user to choose 1/4, 1/2, 2 or 4 and resize accordingly

6- [2.5%] Zoom in/out Graph: zooming the whole graph in or out.
Note: Resize makes permanent changes to figures dimensions but Zoom does not. Zoom just displays them in new sizes without actually updating their original sizes.

7- **[5%] Delete Figure(s):** deleting the selected figure(s).

8- **[5%] Move Figure(s):** moving the selected figure(s).
   - ❑ The user first selects the figure(s) then clicks move icon.
   - ❑ Move action needs one extra click from the user to make it the move destination.
   - ❑ If more than one figure should be moved, you can make the destination click the center or the top left corner of the first figure in the selected figures but the relative positioning of the moved figures should remain the same.

9- **[5%] Copy Figure(s):** copying the selected figure(s) to the application clipboard (build a suitable data structure for your application clipboard).

10- **[5%] Cut Figure(s):** cutting the selected figure(s) to the application clipboard.
   - ❑ The cut figure(s) should be removed from the drawing area as soon as the cut operation is chosen (don't wait for the paste operation to remove them).
   - ❑ Note that the copy and cut operations use the same application clipboard, so each copy/cut should overwrite it.
   - ❑ This means, for example, if a cut operation is followed by a copy/cut operation before pasting the first cut figures, the figures of the first cut will be lost and the paste operation will paste only the last copied/cut figures.

11- **[5%] Paste Figure(s):** pasting the last copied/cut figure(s); pasting the figure(s) that exist in the application clipboard.
   - ❑ The application allows the user to paste the same copied/cut figures multiple times, so the user can make many paste operations.
   - ❑ When pasting the figure(s), the application should ask the user to click at the new location to paste the figures to (as the same way of the move action).
   - ❑ When pasting the copied figure(s), if the user copied some figures then updated some of them (e.g. their color) before pasting, the paste operation should paste the copied figure(s) without the recent changes.

12- **[5%] Save Graph**: saving the information of the drawn graph to a file (see "file format" section). The application must ask the user about the filename to create and save the graph in (overwrite if the file already exists).

13- **[5%] Load Graph:** load a saved graph from a file and re-draw it (see "file format" section).
   - ❑ This operation re-creates the saved figures and re-draws them.
   - ❑ The application must ask the user about the filename to load from.
   - ❑ After loading, the user can edit the loaded graph and continue the application normally.
   - ❑ If there is a graph already drawn on the drawing area and the load operation is chosen, the application must ask the user if he/she wants to save the current graph. Then, any needed cleanup of the current drawn graph and load the new one.

14- **[2.5%] Switch to Play Mode:** by loading the tool bar and the status bar of the play mode. The user can switch to play mode any time even before saving.

15- **[2.5%] Exit:** exiting from the application (after confirmation).
   - ❑ If the drawn graph is not saved before or changed after the last save, user should be prompted to save the graph before exiting.
   - ❑ Perform any necessary cleanup (termination housekeeping) before exiting.

[II] Play Mode: [35%]

In this mode, the graph created in the draw mode (or loaded from a file) is used to play some simple kids games. The operations supported by this mode are:

1- **[15%] Pick & Hide:** The user is prompted to pick a specific figure. When he picks it, it should disappear and the user picks the next figure with the same properties. The application should print counters to count the number of valid and invalid picks done by the user. Finally, when the user picks all similar figures, a grade is displayed for him.
The user should be able to pick figures by
- **Figure Type:** e.g. pick all rectangles.
- **Figure Fill Color:** e.g. pick all red figures, all non-filled figures …etc.
- **Figure Type and Fill Color:** e.g. pick all blue triangles.
- **Figure Area:** e.g. pick the largest (or smallest) rectangle then pick next largest and so on.

2- **[15%] Scramble & Find:** when the user clicks on this option,
- The graph is resized to half its size and is placed at the left half of the drawing area.
- In the right half, the application recreates the graph figures but in random order and at random locations
- The application then highlights a figure in the left side and asks the user to find it at the right side. When the user finds it and clicks on it, both figures should disappear.
- The above steps should be repeated until all figures disappear from both sides.
- Counters for valid and invalid user trials should be displayed and a final grade should be shown at the end.

Note:
At any time the user can restart the game or start another game by clicking its menu icon. When the user does so:
- The original graph should be restored
- All changes done in the current game should be discarded
- Don't forget to make any needed cleanup.

3- **[5%] Switch to Draw Mode**: At any time, user can switch back to the drawing mode.
- The original graph should be restored
- All changes done in the play mode should be discarded
- Don't forget to make any needed cleanup.

[10%] [Bonus] Operations:

The following operations are bonus and you can get the full mark without supporting them.

1- **[5%] Undo and Redo Action:** undoing and redoing all the above draw mode actions except save and load. The maximum number of possible consecutive undo or redo operations is 100 actions.

2- **[5%] Choose any 2 of the following operations:**

- **Move Figure(s) by Dragging**
- **Resize Figure(s) by Dragging**
- **Rotate Figure:** rotating the selected figure(s) by angles 90, 180, or 270.
- **Send to Back** / **Bring to Front**: sending the selected figure(s) to the back or the front of other figures.

General Operation Constraints

The following rules must be satisfied even if not mentioned in the operation's description:

1. **The overlapping rule:** "the above figure (only if filled) hides the overlapped parts of the below figures".

2. **The moving rule:** "relative positions of the moved figures should remain the same".

3. **The drawing borders rule:** Any drawing should be inside the drawing area borders.
    a. If any operation will draw any figures outside the borders, display an error message in the status bar to the user and do not perform the operation.

4. Any needed **cleanups** (freeing any allocated memory) must be done.

5. Many operations need some figures to be selected first. If no figures are selected before choosing such operations, an error message is displayed and the chosen operation will make no change.

Main Classes

Because this is your first object oriented application, 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 figures 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 but after instructor approval)

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

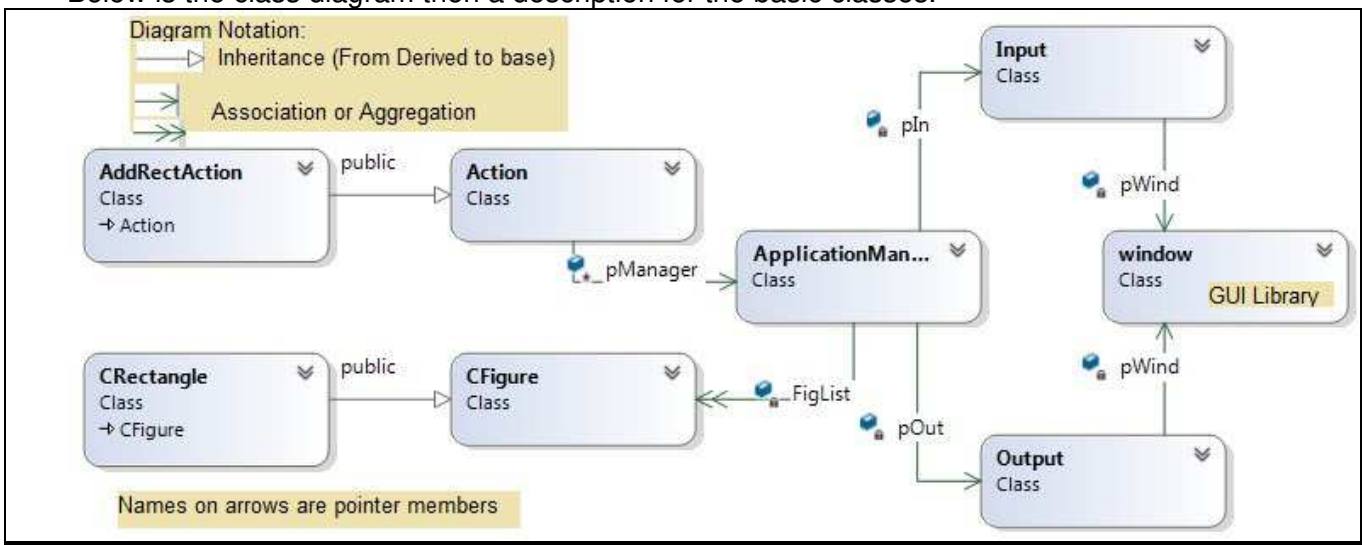

Figure 1 – Class Diagram of the Application

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, figures 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 thought the console. All must be done thought the GUI window.
- Input and Output classes are the **ONLY** classes that have direct access to GUI library.

ApplicationManager Class:

This is the ***maestro*** class that controls everything in the application. Its job is to instruct other classes to do their jobs (**NOT** to do other classes' jobs). It has pointers to objects of all other classes in the application. This is the only class that can operate directly on the figures list (FigList).

CFigure Class:

This is the base class for all types of figures. To create a new figure type (Circle class for example), you must **inherit** it from this class. Then you should override virtual functions of class **CFigure (**e.g. Move, Resize, Draw, etc.). You can also add more details for the class CFigure itself if needed.

Action Class:

Each operation from the above operations 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**. You can also add more details for the class Action itself if needed.

Example Scenarios

The application window in draw mode may look like the window in the following figure. The window is divided to **tool bar**, **drawing area** and **status bar**. The tool bar of any mode should contain icons for all the actions in this mode (***Note***: the tool bar in the figure below is not complete and you should extend it to include all actions of the draw mode).

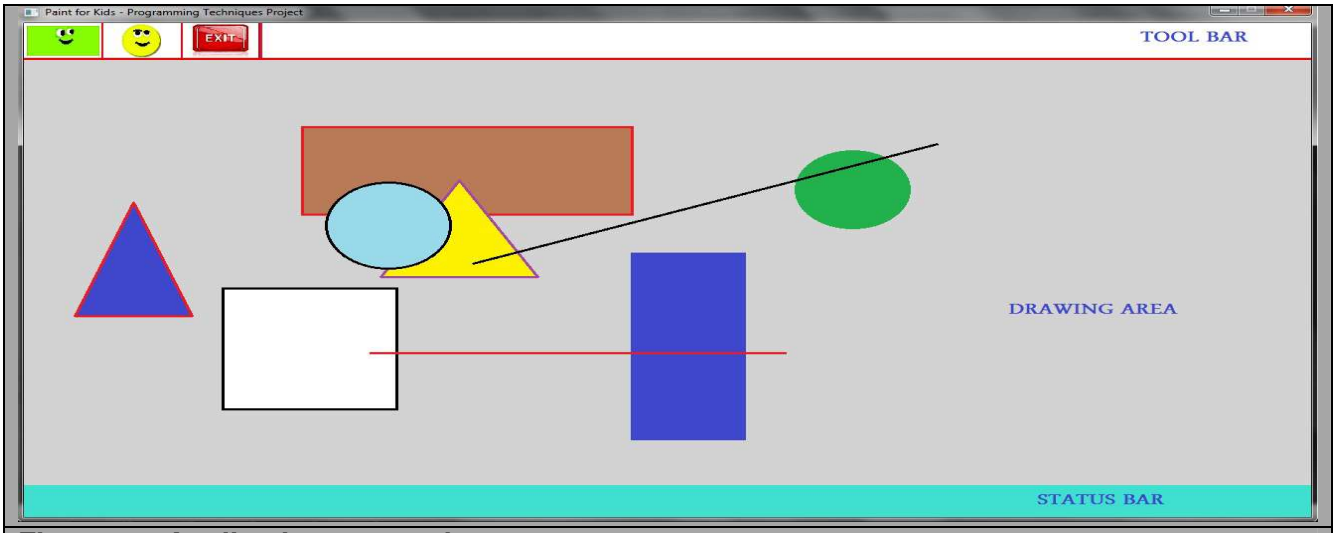

Figure 1 – Application screenshot

Example Scenario 1: AddRectAction

Here is an example scenario for **drawing a rectangle** on the output window. 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 Rectangle icon in the tool bar to draw a rectangle.
3- The ***Input*** class checks the area of the click and recognizes that it is a "draw rectangle" operation. It returns **`DRAW_RECT`** (enum value representing the action: ActionType) to the manager.

Step 2: Create a suitable action

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

Step 3: Execute the action

1- ***ApplicationManager::ExecuteAction(…)*** calls ***AddRectAction::Execute( )***
2- ***AddRectAction****::Execute( )*
   a. calls **AddeRectAction::ReadActionParameters( )** which calls the ***Input*** class to get rectangle parameters (i.e. the 2 corner points of the rectangle) form the user. **Notice** that when ***AddRectAction*** wants to print messages to the user on the status bar, it calls the **Output** class.
   b. Creates a figure object of type ***CRectangle*** class and asks the ***ApplicationManager*** to add it to the current list of figures by calling ***ApplicationManager*** **::AddFigure(…)** function.

At this step the action is complete but it is not reflected yet to the user interface.

Step 4: Reflect the action to the Interface.

1- The ***ApplicationManager::UpdateInterface( )*** is called to draw the updated list of figures.
2- ***ApplicationManager::UpdateInterface( )*** calls the virtual function ***CFigure::Draw( )*** for each figure in **FigList**. (in this example, function **CRectangle::Draw( )** is called)
3- ***CRectangle::Draw( )*** calls ***Output::DrawRect*(…)** to draw a rectangle on the output window.

This means there is a draw function in **Output** class for each figure which takes the figure parameters (e.g. the 2 corners of the rectangle and drawing color …etc.) and draw it on the window. The draw in each Figure class calls the function draw that draws that figure from Output class. Then the update interface function of ApplicationManager loops on FigList and only calls function Draw of each figure.

Example Scenario 2: SaveAction

- **Note: Save/Load** has NO relation to the Input or Output classes. They save/load graphs to/from files not the graphical window.

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

- There is a save function in ApplicationManager and in each figure class but each one does a different job:

1. **CFigure::Save(…)**
   It is a pure **virtual** function in CFigure. Each figure class should **override** it with its own implementation to save itself because each figure has different information and hence a different way or logic to save itself.

2. **ApplicationManager::SaveAll(…)**
   It is the responsible for **calling** Save/Load function for each figure because ApplicationManager is the only class that has FigList. In addition, it only calls function save of each figure; ONLY calling without making the save logic itself (not its responsibility but the responsibility of each figure).

3. **SaveAction::Execute()**
   It does the following:
   - first reads action parameters (i.e. the filename)
   - then opens the file
   - and calls ApplicationManager::Save(…)
   - then closes the file

File Format

Your application should be able to save/load a graph to/from a simple text file. At any time during the draw mode, the user can save or load a graph. In this section, the file format is described together with an example and an explanation for that example.

- **File Format**

```
Draw_Color      Fill_Color      Background_Color
Number_of_Figures
Figure_1_Type   Figure_ID       Figure Parameters (coordinates, color, fill color ...etc.)
Figure_2_Type   Figure_ID       Figure Parameters (coordinates, color, fill color ...etc.)
Figure_3_Type   Figure_ID       Figure Parameters (coordinates, color, fill color ...etc.)
……..
……..
Figure_n_Type   Figure_ID       Figure Parameters (coordinates, color, fill color ...etc.)
```

- **Example:** The graph file will look like that

```
BLUE    GREEN   YELLOW
5
LINE    1   100   200   17   30   BLUE
RECT    2   20    30    154  200  RED   NO_FILL
TRIANG  3   10    20    70   30   220   190   BLACK   RED
LINE    4   10    200   45   90   BLACK
CIRCLE  5   120   97    30   RED  GREEN
```

- **Explanation of the above example**

```
BLUE    GREEN   YELLOW  //Draw, fill, and background colors respectively
5  //Total number of figures is 5

LINE    1   100   200   17   30   BLUE
//Figure1:Line, ID=1, start point (100,200), end point(17,30), color = blue
//note that lines cannot be filled so we skipped this parameter

RECT    2   20    30    154  200  RED   NO_FILL
//Figure2:Rectangle,ID=2, corner1(20,30), corner2(154,200),color = red, not filled

TRIANG  3   10    20    70   30   220   190   BLACK   RED
//Figure3:  Triangle,  ID=3,  corner1(10,20),  corner2(70,30),  corner3(220,190),
//color=black, fill=red

LINE    4   10    200   45   90   BLACK
//Figure4: Line, ID=4, start point (10,200), end point(45,90), color = black

CIRCLE  5   120   97    30   RED  GREEN
//Figure5: Circle, ID=5, center point(120,97), radius=30, color=red, fill = green
```

Notes:

- You can give any IDs for the figures. Just make sure ID is **unique** for each figure.
- You are allowed to modify this file format if necessary **but after instructor approval**.
- You can use numbers instead of text to simplify the "load" operation. For example, you can give each figure type and each color a number. This is done by using ***enum*** statement in C.
- **The LoadAction**:
  For lines in the above file, the LoadAction first **reads** the figure type then **creates** an object of that figure. Then, it **calls** CFigure::Load virtual function that is overridden in each figure type to make the figure load its data from the opened file. Then, it **calls** ApplicationManager::AddFigure to add the created figure to FigList.

Project Phases

1- Phase 1 (Input / Output Classes) [15% of total project grade]

In this phase, you will implement the input and the output classes as they do not depend on any other classes. The Input and Output classes should be **finalized** and ready to run and test. Any expected user interaction (input/output) that will be needed by phase 2, should be implemented at this phase.

Input and Output Classes Code and Test Code

You are given a code for phase 1 (separate from the code of the whole project) that contains both the input and output classes partially implemented. Each team should complete such classes as follows:

1- ***Input Class***:
- **[1 function]** Complete the function Input::**GetUserAction(…)** where the input class should detect all possible actions of any of the 2 modes according to the coordinates clicked by the user.
- Add any other needed member data or functions

2- ***Output Class***:
- **[2 function]** Draw the full tool bars.
  Output class should create 2 FULL tool bars: one for the draw mode and one for the play mode. Each contains icons for each action in this mode.
- **[4 functions]** Draw function for each type of figures (Line, Rectangle, Triangle and Circle). These functions must be able to draw the figures: either filled or not filled, with all supported colors for borders or filling, and draw them either normal or highlighted.
- Add any other needed member data or functions

3- **Test Code**: (this is NOT part of the input or the output classes; it is just a test code)
- **Complete** the code given in TestCode.cpp file to test both Input and Output classes.

Deliverables*:*

Each team should deliver a CD that contains IDs.txt (team number, member names, IDs, email) and phase 1 code that has Input and Output classes and test program completed.

2- Phase 2 (Project Delivery) [85% of total project grade]

In this phase, the completed I/O classes (without phase 1 test code) should be added to the project framework code and the remaining classes should be implemented. Start by implementing the base classes then move to derived classes.

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) A CD that contains the following:
   a. ID.txt file. (Information about the team: names, IDs, team email)
   b. The workload division document.
   c. The project code and resources files.
   d. Sample graph files: at least three different graphs. For each graph, provide:
      i. Graph text file (created by save operation)
      ii. Graph screenshot for the graph generated by your program
      iii. Screenshot of one action of play mode

On CD cover, each team should write: **semester or credit**, **team number** and **phase number**.

Phase 2 Evaluation Criteria

Draw Mode [60%]

❑ Each operation percentage is mention beside its description.

Play Mode [35%]

❑ Each operation percentage is mentioned beside its description.

Code Organization & Style [5%]

❑ Every class in .h and .cpp files
❑ Variable naming
❑ Indentation & Comments

Bonus [10%]

❑ Each operation percentage is mention beside its description.

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** (check them below) → **MINUS** 30% of Operation Grade
   - ❑ Separate class for each figure and action
   - ❑ Each class is doing its job. No class is performing the function of another class.
   - ❑ Polymorphism: use of pointers and virtual functions

4. **For <u>each</u> corner case** that is not working → **MINUS** 10% to 20% of the Operation Grade according to instruction evaluation.

Notes:

- ❑ The code of any operation does NOT compensate for the absence of any other operation; for example, if you make all the above operations except the delete operation, you will lose the grade of the delete operation no matter how good the other operations are.
- ❑ **Each of the above requirements will have its own weight. The summation of them constitutes the group grade (GG).**

Individuals Evaluation:

Each member must be responsible for some actions and must answer some questions showing that he/she understands both the program logic and the implementation details. Each member will get a percentage grade (**IG**) from the group grade (**GG**) according to this evaluation.
The overall grade for each student will be the product of GG and IG.

Note: we will reduce the IG in the following cases:

- ❑ Not working enough
- ❑ Neglecting or preventing other team members from working enough

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).
  - Reflect the action to the Interface (i.e. function ApplicationManager::UpdateInterface()).

- **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. FigList is an array of CFigure pointers to be able to point to figures of any type (polymorphism). 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 CRectangle needs to draw itself on the GUI, it calls function ***Output::DrawRect*** because dealing with the GUI window is the responsibly of class output.

- **Abusing Getters:** Don't use getters to get data members of a class to make its job in another class. This breaks the classes' responsibilities rule. For example, do NOT add in ApplicationManager function GetFigList() that gets the array of figures to other classes to loop on it there. FigList and looping on it are the responsibility of ApplicationManager. See "Example Scenario 2".

- **Virtual Functions:** In general, when you find some functionality (e.g. saving) that has different implementation based on each figure type, you should make it virtual function in class CFigure and override it in each figure type with its own implementation.

   A common mistake here is the abuse of **dynamic_cast** to check for object type and perform object job outside the class not inside a class member virtual function.

[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.

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 makes its operation correctly then move to another action.

- For example, the member who takes action 'save' should create 'SaveAction' and write the code related to SaveAction inside 'ApplicationManager' and 'CFigure' hierarchy classes. Then run and check if the figures are successfully saved. Don't wait for the whole project to finish to run and test your action.

- It is recommended to give similar actions to same member because they have similar implementation. For example: copy, cut, paste together and save and load together … etc.

- 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 selecting figures, among the members then integrate before dividing the rest of the actions.