Cairo University
Faculty of Engineering
Computer Engineering Department
Programming Techniques Course

Programming Techniques Project Requirements (Phase 2)

Paint for Kids

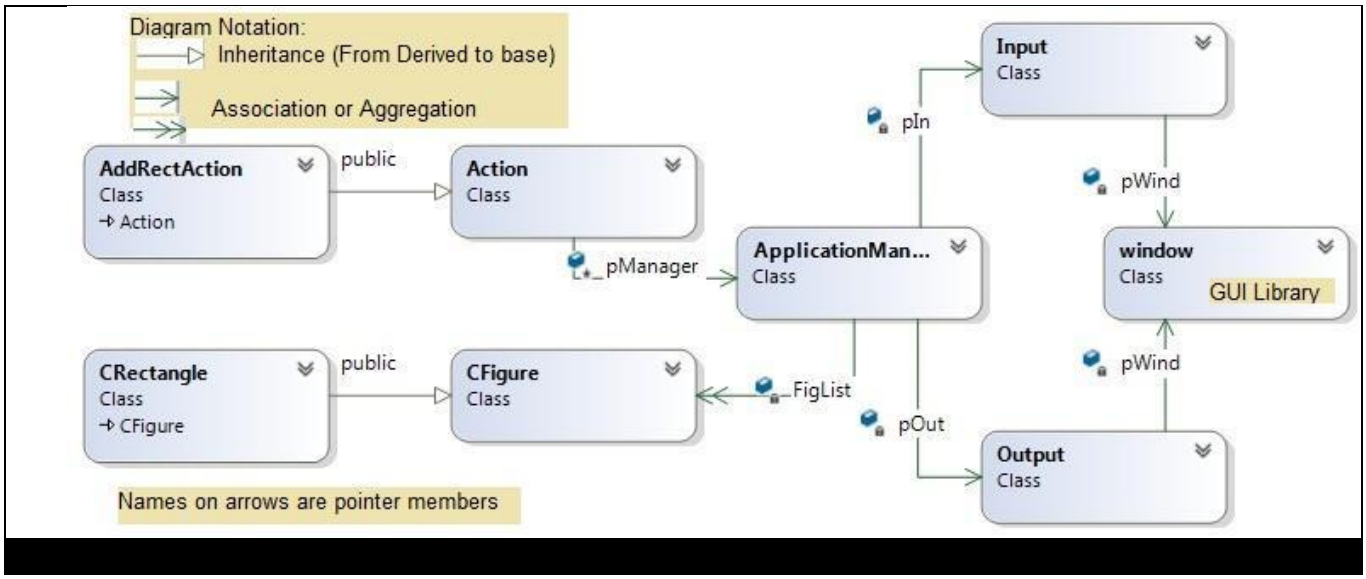
Main Classes

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).

We want you to **stick** to the given design because it is your **first OOP project** and we want to give you an example of a fairly-good designed code to work in it. In most of the projects of next years, you will be free to design your OOP classes the way you want.

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



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 through the console. All must be done through 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 manage or 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 (Hexagon class for example), you must **inherit** it from this class. Then you should override virtual functions of class **CFigure (**e.g., Draw, Save, 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**. 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 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 current mode).

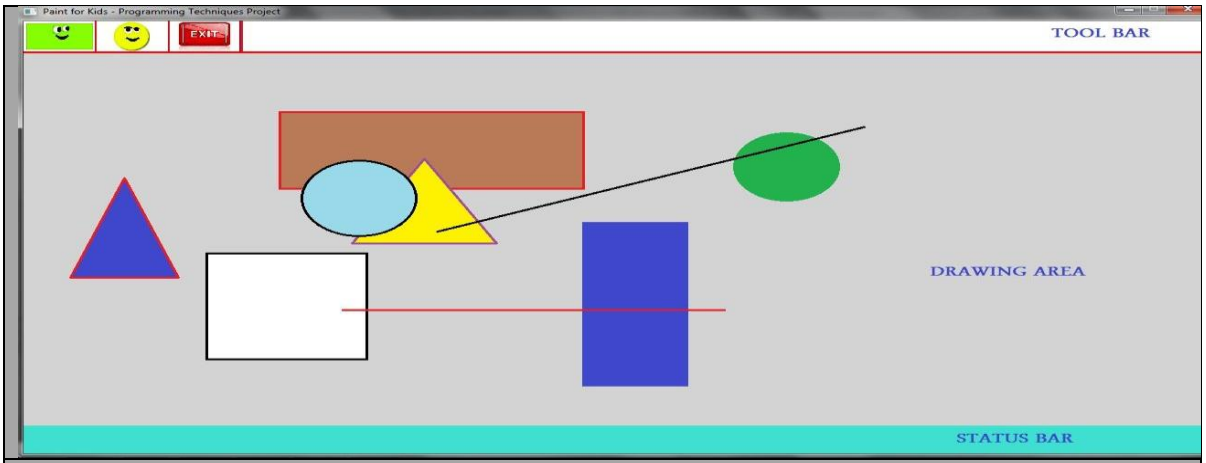


Figure 1 – Application screenshot

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 "Add 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`** (an **"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 **AddRectAction::ReadActionParameters( )** which calls the ***Input*** class to get rectangle parameters (i.e. the 2 corner points of the rectangle) from the user. **Notice** that when ***AddRectAction*** wants to print messages to the user on the status bar, it calls some functions from the **Output** class.
   b. Creates (allocates) 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 only its job or responsibility.

- There is a save function in **ApplicationManager** and in each figure class but they perform different jobs:

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**. Note that it only calls function save of each figure; ONLY calling without making the save logic itself (not the responsibility of **ApplicationManager** but the responsibility of each figure class). This note is important and has a huge grade percentage.

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

```
Current_Draw_Color    Current_Fill_Color
Number_of_Figures_n
Figure_1_Type  Figure_ID    Figure Parameters (coordinates, draw color, fill color ...etc.)
Figure_2_Type  Figure_ID    Figure Parameters (coordinates, draw color, fill color ...etc.)
Figure_3_Type  Figure_ID    Figure Parameters (coordinates, draw color, fill color ...etc.)
……..
……..
Figure_n_Type  Figure_ID    Figure Parameters (coordinates, draw color, fill color ...etc.)
```

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

```
BLUE    GREEN
5
RECT    1    100    200    17    30    BLUE    RED
RECT    2    20     30     154   200   RED     NO_FILL
TRIANG  3    10     20     70    30    220     190    BLACK    RED
RECT    4    10     200    45    90    BLACK   BLUE
TRIANG  5    20     30     80    90    220     190    BLUE     RED
```

- **Explanation of the above example**

```
   BLUE     GREEN //current draw/fill that will be used to draw any new figures.
   5  //Total number of figures is 5

   RECT     1      100    200    17     30     BLUE  RED
//Figure1: Rectangle,ID=1,corner1(100,200),corner2(17,30),draw=blue,fill=red

   RECT     2      20     30     154    200    RED   NO_FILL
//Figure2: Rectangle,ID=2,corner1(20,30),corner2(154,200),draw=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),
//draw=black, fill=red

   RECT     4      10     200    45     90     BLACK  BLUE
//Figure4: Rectangle,ID=4,corner1(10,200),corner2(45,90),draw=black,fill=blue

   TRIANG   5      20     30     80     90     220    190    BLUE  RED
//Figure3: Triangle,ID=5,corner1(20,30),corner2(80,90),corner3(220,190),
//draw=blue,fill=red
```

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 can be done using ***enum*** statement in C.
- ❑ **The LoadAction**:
  For lines in the above file, the **LoadAction** loops on each line in the input and for each line it **reads** only the figure type then **creates** (allocates) an object of that figure class. Then, it **calls CFigure::Load** virtual function that is overridden in the class of each figure type to make the figure load its data from the opened file by itself (its job). Then, it **calls ApplicationManager::AddFigure** to add the created figure to **FigList**.

Project Phases

Phase 2 (Project Delivery) [75% 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** (given for phase 2) and the remaining classes should be implemented. Start by implementing the base classes then move to derived classes.

Phase 2 Deliverables*:*

- One zip file containing 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 (images, saved files, …etc.)**.
  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

Note that Each project phase should be submitted first **online** (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 day of the week of submitting the project online and the discussion schedule of each phase will be announced later.

Phase 2 Evaluation Criteria

Draw Mode [75%]

- ❑ Each operation percentage is mentioned beside its description.

Play Mode [20%]

- ❑ Each operation percentage is mentioned beside its description.

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 operation percentage is mentioned 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 → MINUS 30%** of Operation Grade
   - ❑ **Separate class** for each figure 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 (Very Important).**

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 operations) 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 with their problems in their code parts (helping does NOT mean implementing their parts).

✓ You should **inform the TAs** before the deadline **with a sufficient time (some weeks before)** if any problems occurred between members to be able to give warnings and take actions.

✓ If one or more members didn't make their work, the other members will NOT be affected **as long as:** i) the students informed the TAs about this problem with a sufficient time before the deadline (to give warnings and take actions), ii) the students who finished their part integrated all their parts together AND their part can be tested easily to see the output.

✓ If a student couldn't finish his part, try to still integrate it to at least take the integration grade.

✓ If a student ignores other team members and does their parts without a permission, a penalty also will be applied to him. Complain to the involved TA if your team members do not work, but do not ignore them.

APPENDEX A

[I] 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 **'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 implemented action.

- It is recommended to give similar actions to the same member because they have similar implementation. For example: 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) then integrate before dividing the rest of the actions.

[II] 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. 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**. Similarly, class **ApplicationManager** must not contain any logic. It only should call functions (the **maestro**). 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 **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** (or similar implementations like **class type** data member or **typeid**) to check the object type outside the class and perform the class job there (not inside the class in a virtual member function).
  - 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.
  - In addition, virtual functions check on the object type automatically, so no need to add dynamic_cast and check on the object type when you use virtual functions.

- **Not all the actions** need to add a corresponding function inside **ApplicationManager**. This will make **ApplicationManager** perform the responsibility of these actions. However, some actions need to loop on **FigList** (e.g., ***SaveAction*** …etc.). In this case only (looping on FigList), you can add functions for them in **ApplicationManager** that loop on the lists 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 the software engineering point of view. Search for the reasons of that or ask your TA for more information.

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