بسم الله الرحمن الرحيم

"نرفع درجاتٍ من نشاء وفوق كل ذي علم عليم"

Fundamentals of Computer Programming Project Requirements

Paint for Kids

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***) you 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.

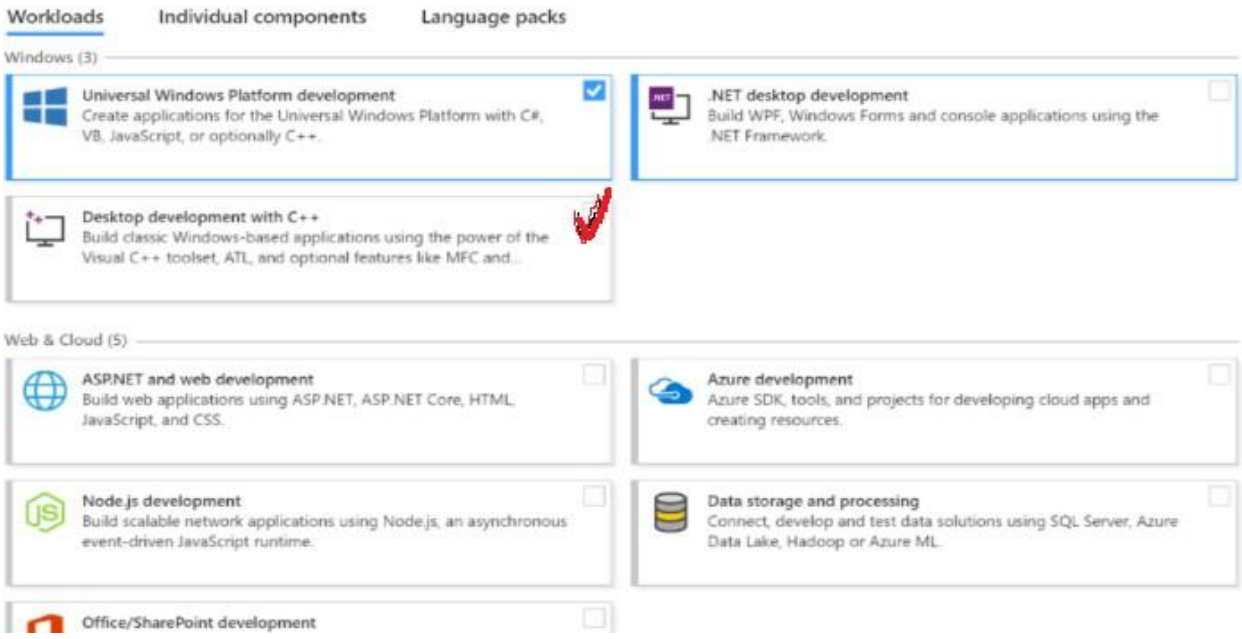
Given Code:

As 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.).

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.

NOTE:

- Number of students per team = **6 students.**
- **Project Duration: form Wednesday 5/1/2022 to Friday 28/1/2022**
- Use visual studio 2022 Community edition as IDE **download from here https://visualstudio.microsoft.com/vs/community/ when you install . select desktop development with c++.**



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 some 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. The main operations supported by the application are:

[I] Draw Mode: [60%]

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

1- [5%] Add Figure: adding a new figure to the list of figures. This includes:
- Adding a new **square**, a new **ellipse**, or a new **hexagon.**

2- [5%] Change Current Colors: changing the current ***drawing*** color, the current ***filling*** color, or the window ***background*** color. 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/Unselect a Figure:
- To select a figure, user should click inside the figure or on its border (***same for filled and unfilled figures***)
- The selected 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 re-clicks on a selected figure, this will un-select it

Note: the color used for highlighting must not appear in the palette of colors.

4- [5%] Change Figure Colors: changing the ***drawing***, or ***filling*** colors for the selected figure. This affects the selected figure only.

5- [10%] Delete a Figure: deleting the selected figure.

6- [5%] Resize a Figure: resize the selected figure by a factor of 1/4, 1/2, 2, 4

7- [5%] Send to back/Bring to Front: for the selected figure.

8- [7.5%] 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).

9- [7.5%] 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.

10- [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.

11- [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- [30%] 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 ellipses.
- **Figure Fill Color:** e.g. pick all red figures, all non-filled figures …etc.
- **Figure Type and Fill Color:** e.g. pick all blue hexagons.

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.

2- [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:

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- [2.5%] Move a Figure by Dragging
3- [2.5%] Delete Multiple Figures: user selects multiple figures and deletes them in one click.

General Operation Constraints

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

2. **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.

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

4. 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

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.

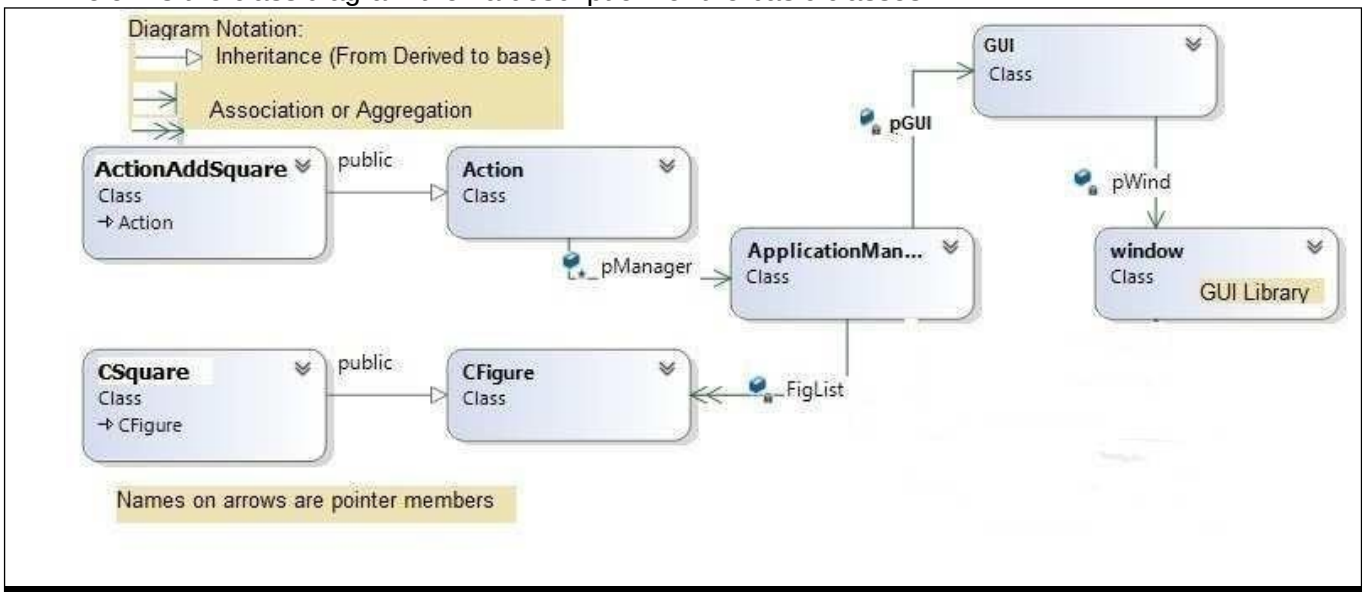


Figure 1 – Class Diagram of the Application

GUI Class:

This class is responsible for user interface. It is the only class that has a direct access to the graphics library. User inputs must come through this class. It is also responsible for toolbar and status bar creation, figures drawing, and for messages printing to the user.

If any other class needs to read input or make output, it must call a member function of GUI class. You should extend this class and add suitable member functions.

Note: - No input or output is done thought the console. All must be done thought the GUI window.

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 (Ellipse 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 (operation), 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).

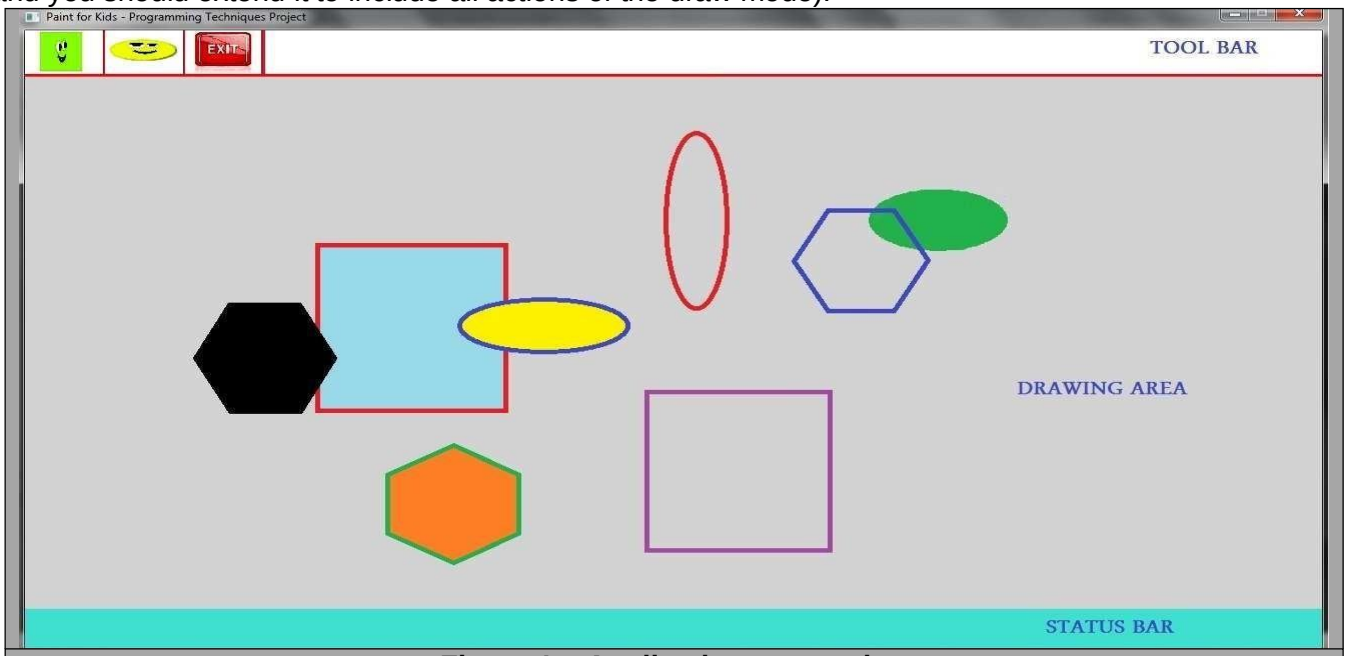


Figure 1 – Application screenshot

Example Scenario 1: ActionAddSqaure

Here is an example scenario for **drawing a square** on the output window. It is performed through the four steps mentioned in 'Appendix A - implementation guidelines' section. These four steps are inside "ApplicationManager::Run( )" function in the given code.

Step 1: Detect user clicks and map to an action type

1- The ***ApplicationManager*** calls the ***GUI***::***MapInputToActionType*** ( ) function.
2- The user clicks on the Square icon in the tool bar.
3- The ***GUI*** class checks the area of the click and recognizes that it is a "draw square" operation. It returns **`DRAW_SQUARE`** (enum value representing the action: ActionType) to the manager.

Step 2: Create a suitable action

1- ***ApplicationManager::CreateAction(ActionType)*** is called to create an object of type ***ActionAddSqaure*** *class* and returns it.

Step 3: Execute the action

1- ApplicationManager::ExecuteAction(…) calls ***ActionAddSqaure::Execute( )***
2- ActionAddSqaure*::Execute( )*

a. Calls the ***GUI*** class to get square parameters (i.e. 2 points) form the user.
b. From the data points, prepare square data (top left corner and side length)
c. Creates a figure object of type ***CSquare*** 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.

ApplicationManager::UpdateInterface( ) is called to draw the updated list of figures as follows

1- It calls the virtual function ***CFigure::DrawMe( )*** for each figure in **FigList**. (in this example, function **CSquare::DrawMe( )** is called)

2- CSquare::DrawMe( ) calls ***GUI::DrawSquare*(…)** to draw a square on the output window.

Note:
GUI should provide a draw function each figure. GUI::DrawSquare is already given. You should create GUI::DrawEllipse and GUI::DrawHexagon

Example Scenario 2: SaveAction

- **Note: Save/Load** has NO relation to the GUI class. 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. **ActionSave::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

The Load Action

For lines in the above file, the Load Action (ActionLoad class) should first **read** the figure type then **create** an object of that figure. Then, it should **call** CFigure::Load virtual function that is overridden in each figure type to make the figure load its data from the opened file. Then, it should **calls** ApplicationManager::AddFigure to add the created figure to FigList.

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
2
SQR     1    100    200    17    BLUE      RED
SQR     12   20     30     54    RED       NO_FILL
```

- **Explanation of the above example**

```
BLUE    GREEN    YELLOW  //Draw, fill, and background colors respectively
2  //Total number of figures is 2

SQR     1    100    200    17    BLUE      RED
//Figure1: Square, ID=1, topLeftcorner(100,200), length(17), color=blue, fill=red

SQR     12   20     30     54    200   RED       NO_FILL
//Figure2: Square,ID=2, topLeftcorner(20,30), length(54),color = red, not filled
```

Notes:

- There should be lines for Ellipse and Hexagon figures as well. You should decide whatare the parameters needed to be saved for each of them.
- 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's 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.
- Color can be stored as (R,G,B) components instead of color name.

Project Deliverables & Evaluation Criteria

Deliverables:

(1) **Workload division**: a **printed page** containing team information and a table that contains members' names and the classes each member has implemented.
(2) A folder 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. Three different graph files

Name the folder Fundamental Programming - project, then team number

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, …etc.

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 each 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 (**PG**) from the group grade (**GG**) according to this evaluation.
The overall grade for each student will be the product of GG and PG.

Note: we will reduce the PG 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 (see function ApplicationManager::Run( )):**
  - Read user input and map to action type
  - Create suitable action object for that action type.
  - Execute the action
  - Reflect the action to the Interface

- **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 CSquare needs to draw itself on the GUI, it calls function ***GUI::DrawSquare*** because dealing with the graphics window is the responsibly of class GUI.

  Abusing Getters: Do not 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 as 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.