Cairo University
Faculty of Engineering
CMPS102 **Fall 2023**

Programming Techniques

Project Requirements (Phase 2)

Main Classes

Because this is your first object-oriented application, you are given a ***code framework*** where we have ***partially*** implemented 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 shapes on the screen and reading the coordinates of mouse clicks …etc.).

You must **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. In most of the projects of next years, you will be free to design your OOP classes the way you want.

Below are the classes of phase 1.

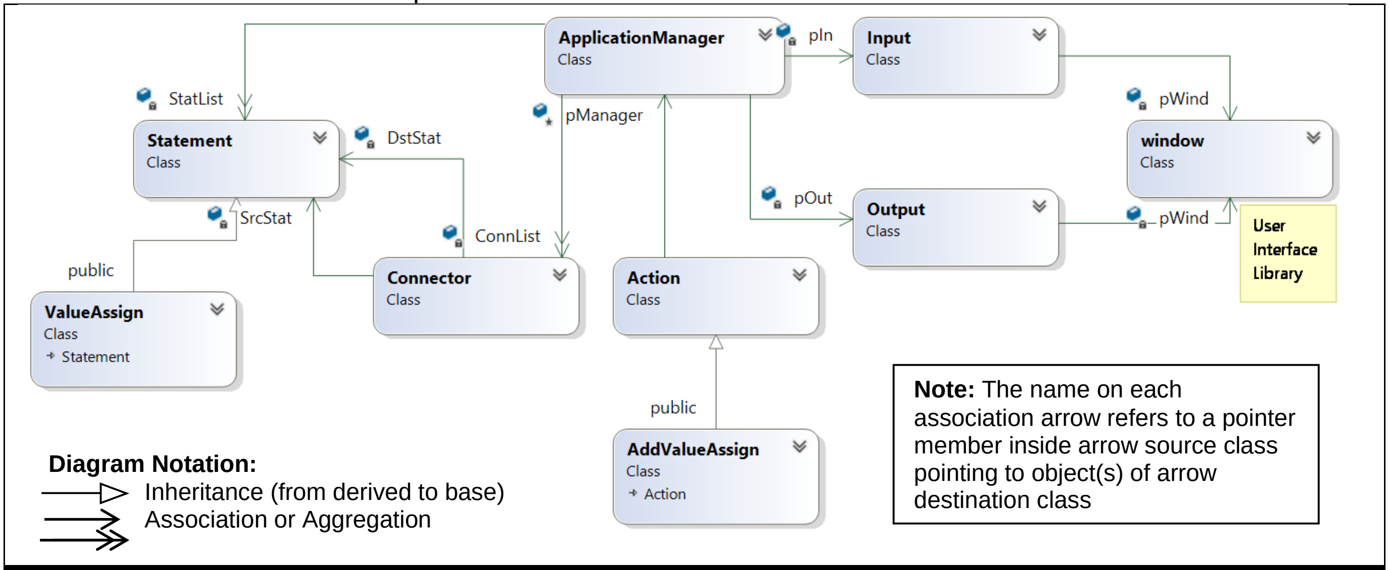


Figure 1 – Class Diagram of the application

Input Class:
ALL user inputs must come through this class. If any other class wants 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, flowchart 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 the application. As its name shows, its job is to ***manage or instruct*** other classes to do their job (**NOT** to do other classes' jobs). It has pointers to objects of almost all other classes in the application. In addition, this class maintains the list of statements and connectors inside the application. This is the **ONLY** class that can operate directly on the application statement list (**StatList**) or connector list (**ConnList**).

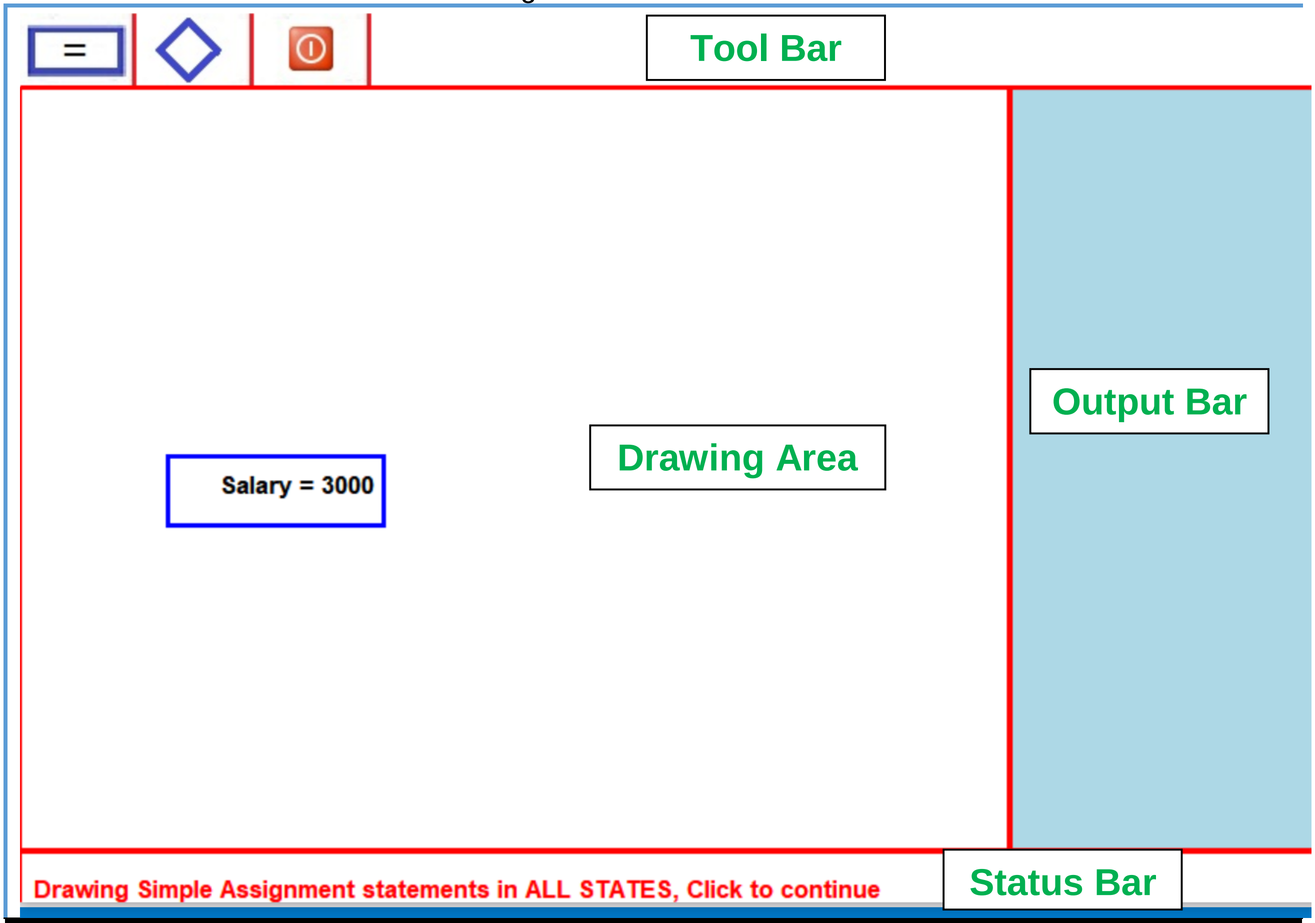
Statement Class:
This is the base class for all types of statements supported by the application. To add a new type of statements, you must **inherit** it from this class. Then you should override virtual functions found in the class **Statement (**e.g. Draw, Save, etc.).

Connector Class:
This is the class for the connector that connects between two statements.

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.

Example Scenarios

The application window in design mode **may** look like the window in the following figure. The window is divided to **tool bar**, **drawing area**, **status bar** and **output bar**. The tool bar of any mode contains icons for all the actions in this mode (***Note***: the tool bar in the figure below is not completed and you should extend it to include all actions of the current mode). The status bar is to print messages to the user. The output bar will be used in the simulation mode to output the running result to the user. It should exist from the design mode too in order not to draw in its area.



Drawing Simple Assignment statements in ALL STATES, Click to continue

Figure 2 – Framework screenshot (here you can only add Value Assignment)

Example Scenario 1: AddValueAssign

Here is an example scenario for **drawing a value assignment statement** on the output window. It is performed through the four main 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 "Value Assignment Statement" icon in the tool bar to draw a value assignment.
3- The ***Input*** class checks the area where the user clicked and recognizes that it is a "add value assignment" operation. It returns **ADD_VALUE_ASSIGN** (an **enum** value indicating the required action) to the manager.

Step 2: Create a suitable action

1- ***ApplicationManager::ExecuteAction(ActionType)*** is called. It creates an object of type ***AddValueAssign*** action and calls ***AddValueAssign::Execute*** to execute the action

Step 3: Execute the action

1- ***ApplicationManager::ExecuteAction(…)*** calls ***AddValueAssign::Execute( )***
2- ***AddValueAssign::Execute* ( )**
   a. Calls ***AddValueAssign*::ReadActionParameters( )** which calls the ***Input*** class to get the action parameters (i.e. the position point of the rectangle, LHS and RHS) from the user. **Notice** that when ***AddValueAssign*** wants to print messages to the user on the status bar, it calls a function from the **Output** class (**Output::PrintMessage** function). **Note:** Deciding the parameters of each action and reading them inside function **ReadActionParameters( )** are graded.
   b. Creates (allocates) a statement object of type ***ValueAssign*** class and asks the ***ApplicationManager*** to add it to the current list of Statements **StatList** (by calling ***ApplicationManager::AddStatement(…)*** function).

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

Step 4: Reflect the action to the Interface

1- The ***ApplicationManager::UpdateInterface( )*** is called to draw the updated list of statements (and connectors).
2- ***ApplicationManager::UpdateInterface( )*** calls the virtual function ***Statement::Draw( )*** for each statement. (in this example, function ***ValueAssign::Draw( )*** is called)
3- ***ValueAssign::Draw( )*** calls ***Output::DrawAssign( )*** to draw assignment statement.

This means that there is a draw function in **Output** class for each type of statement which takes the statement parameters (e.g. the position point …etc.) and draw it on the window. The draw in each **Statement** class calls the draw function of the Output class that draws this type of statements. Then the update interface function of **ApplicationManager** loops on **StatList** (and **ConnList**) and calls function **Draw** of each statement (and connector).

Example Scenario 2: SaveAction

Save/Load has NO relation with the **Input** or **Output** classes. They save/load actions write/read 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**, each **Statement** class and the **Connector** class but each save function performs a different job:

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

2. **Connector::Save(…)**
   It is similar to **Statement::Save(…)** but to save the information of connectors.

3. **ApplicationManager::SaveAll(…)**
   It is the responsible for **calling** Save function for each statement and connector because **ApplicationManager** is the only class that has **StatList** and **ConnList**. Note that it only calls function save of each statement/connector; ONLY calling without making the save logic itself (not the responsibility of **ApplicationManager** but the responsibility of each statement/connector class). This note is important and has a huge grade percentage.

4. **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 and load a flowchart from a simple text file. In this section, the file format is described together with an example and an explanation for that example. The application should enable the user to create a new flowchart or to load an existing one and edit it.

- **Flowchart File Format**

```
Number_of_Statements
Statement_1_Type      Statement_ID     Stat_Graphics_info     Stat_Code_Info
Statement_2_Type      Statement_ID     Stat_Graphics_info     Stat_Code_Info
…………………………………………………………..
Statement_n_Type      Statement_ID     Stat_Graphics_info     Stat_Code_Info
Number of Connectors
Source_Statement_ID   Target_Statement_ID       Outlet_branch
…………………………………………………………..
Source_Statement_ID   Target_Statement_ID       Outlet_branch
```

Notes:

- ❑ **Outlet_branch** is only applicable for connections going out of the **conditional statements** because conditional statement has two output branches **(YES branch: 1, NO branch: 2)**. For other connections it is **set to 0.**

- **Example:** The flowchart shown in figure 3.a below is represented by the file in figure 3.b

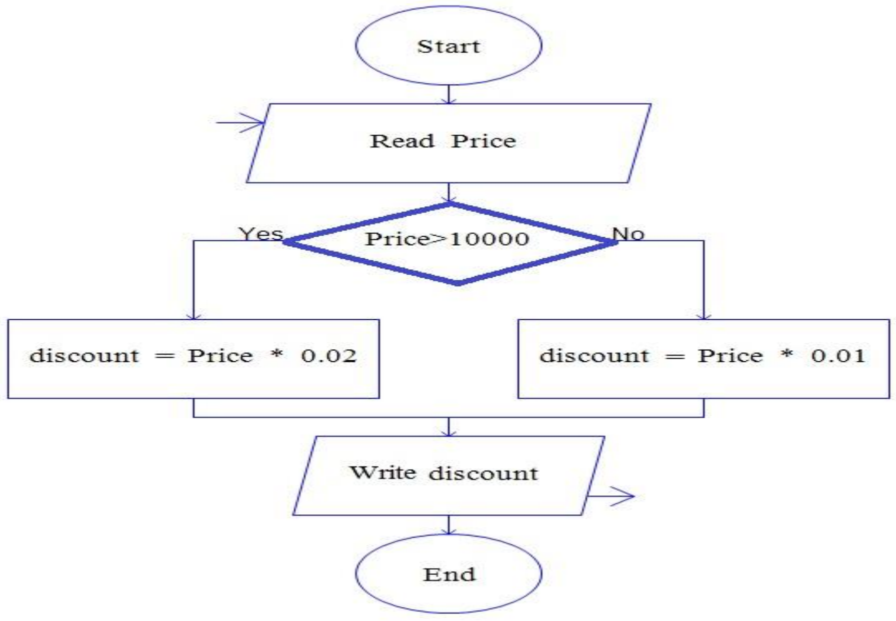


Figure 3.a – Flow Chart Diagram

```
7
STRT        1       200     90
READ        2       150     145     Price
COND        3       200     190     Price   GRT     10000
OP_ASSIGN           4       100     250     discount        Price   MUL     0.02
OP_ASSIGN           5       230     250     discount        Price   MUL     0.01
WRITE       6       160     312     discount
END         7       200     376
7
1           2       0
2           3       0
3           4       1
3           5       2
4           6       0
5           6       0
6           7       0
```

Figure 3.b - File that represents the flow chart

Figure 3.b represents the flowchart in a text file. However, the order of the statements in the text file is not always the same as the order of the statements in the flowchart. The user usually creates the statements and the connections in any order and then saves the chart at any time.

Note: **Statement IDs can be not consecutive but they must be unique.**

- **Explanation of the above example**

Here is the explanation of figure 3.b file

```
7    //The flowchart has 7 statements, so 7 lines will be followed for statements

STRT        1      200    90
//Start statement, ID→1, Center Point (200,90)

READ        2      150    145    Price
//Read statement, ID→2, Left Corner (150,145), Var to read→ Price

COND        3      200    190    Price   GRT   10000
//Conditional statement, ID→3, Diamond upper Point (200,190), Operand1→ Price, Operator→ greater_than,
Operand2→ 10000  (i.e. Price > 10000)

OP_ASSIGN        4      100    250    discount      Price   MUL   0.02
//Operator Assignment, ID→4, Left Corner (100,250), LHS→ discount, RHS→ (Operand1→Price, Operator→
multiplication, Operand2→  0.02) (i.e. discount = Price * 0.02)

OP_ASSIGN        5      230    250    discount      Price   MUL   0.01
//Operator Assignment, ID→5, Left Corner (230,250), LHS→ discount, RHS→ (Operand1: Price, Operator→
multiplication, Operand2:  0.01) (i.e. discount = Price * 0.01)

WRITE       6      160    312    discount
//Write statement, ID→6, Left Corner (160,312), Var to write→discount

END         7      200    376
//End statement, ID→7, Center Point (200,376)


7    //Chart has 7 connectors, so 7 lines will be followed for connectors
1         2       0       //Statement 1 (Start) is connected to Statement 2(Read)
2         3       0       //Stat 2 (Read) is connected to Stat 3(Conditional)
3         4       1       //Stat 3 (Conditional) is connected to Stat 4(OP_ASSIGN), branch 1 (YES_branch)
3         5       2       // Stat 3 (Conditional) is connected to Stat 5(OP_ASSIGN), branch 2 (NO_branch)
4         6       0       // Stat 4 (OP_ASSIGN) is connected to Stat 6(Write)
5         6       0       // Stat 5 (OP_ASSIGN) is connected to Stat 6(Write)
6         7       0       // Stat 6 (Write) is connected to Stat 7(End)
```

Notes:

- ❑ You can select any IDs for the statements. Just make sure ID is **unique** for each statement.
- ❑ You are allowed to add some modifications to this file format if necessary but get **approval from your instructor** first.
- ❑ For statement types, you can use numbers instead of text to simplify the "**load**" operation. For example, you can give each statement type a number.
- ❑ Note that each statement type has different number of information items to read from the file. You can determine this number as soon as you know the type of the statement.
- ❑ **Loading Statement in the LoadAction**:
  For each statement in the statement lines above, the **LoadAction** first **reads** the statement type then **creates** (allocates) an object of that type. Then, it **calls Statement::Load** virtual function that is overridden in each derived class of class **Statement** to make each statement type loads its data from the opened file by itself (its job). Then, it **calls ApplicationManager::AddStatement** to add the created statement to **StatList**.

- **[BONUS] <u>Code File</u>**
  As mentioned in the **<u>bonus part</u>**, your application should be able to create a C++ code file that is equivalent to the flowchart. Here is an equivalent code file of the above example

```
#include <iostream>
using namespace std;
int main()
{
   double Price, discount;
   cin >> Price;
   if (Price>10000)
   {
      discount =Price*0.02;
   }
   else
   {
      discount =Price*0.01;
   }
   cout << discount;
   return 0;
}
```

<u>Notes</u>:

- ❑ It is required to support only "**double**" variables in your application.
- ❑ Any other equivalent valid C++ code is acceptable.
- ❑ Each statement should be able to write its code to the output code file through the virtual function "***Statement::GenerateCode***(…)"

Project Phases

Phase 2 (Project Delivery) [75% of total project grade]

In this phase, the completed I/O classes, Defs.h, UI_Info.h and HelperFn (.h an .cpp) (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. Work must be divided between team members.

Phase 2 Deliverables*:*

(1) **Workload division**: a **printed page** containing team information and a table containing members' names and the classes each member has implemented.

(2) A zip folder that contains the following

a. ID.txt file. (Information about the team: name, IDs, team email)
b. The workload division document.
c. The project code and resources files (images, saved files, …etc.).
d. Sample flowcharts: Three different charts. For each chart, provide:
e. [1] chart text file (created by save operation), [2] chart screenshot for the graph generated by the application, [3] screenshot of chart simulation showing final output, and [4] [BONUS] code file generated by the application.

The zip folder will be uploaded on Google classroom.

Note that Each project phase should be uploaded on Google classroom (same time for all groups). **No modifications are allowed after the upload time.** After that the face-to-face **discussion** of the project will be held. The day of the week of sending each project phase and the discussion schedule of each phase will be announced later.

Phase 2 Evaluation Criteria

Design Mode [65%]

- ❑ Each operation percentage is mentioned beside its description.

Simulation Mode [30%]

- ❑ Each operation percentage is mentioned beside its description.

Code Organization & Style [5%]

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

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 statement and action
   - ❑ **Encapsulation**
   - ❑ 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 instructor 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.
- ❑ There is no bonus on anything other than the operations mentioned in the bonus section.
- ❑ **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.

You should **inform the TAs** before the deadline **with a sufficient time (some weeks before it)** if any of your team members does not contribute in the project work and does not make his/her tasks. The TAs should warn him/her first before taking the appropriate grading action.

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. **StatList** is an array of **Statement pointers** to be able to point to statements of any type and use 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 **ValueAssign** needs to draw itself on the GUI, it calls function ***Output::DrawAssign*** 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 **GetStatList()** that gets the array of statements to other classes to loop on it there. **StatList** (and **ConnList**) and looping on them 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 statement type, you should make it virtual function in class **Statement** 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 **class type** data member) to check the object type outside the class and perform the class job there (not inside the class using 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.

- **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 **StatList** or **ConnList** (e.g. ***SaveAction*** …etc.). In this case only, 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 the approach of passing variables in 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.**

Note: you should read the comments mentioned in the code carefully because they mention some implementation guidelines and give you hints on class responsibilities and how classes should communicate with each other.

[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 **'save'** should create **'SaveAction'** and write the code related to **SaveAction** inside **'ApplicationManager'**, **'Connector'** and **'Statement'** hierarchy classes. Then run and check if the statements and connectors 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: 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 statements and connectors …etc.) then integrate before dividing the rest of the actions.