The Task Output Framework: An Object Oriented Tool For Application Development

John E. Ellis, Trilogy Consulting Corp.

Abstract

This paper discusses the limitations and drawbacks of some traditional approaches used when developing applications, specifically applications that automate tasks and generate output in text or graphical form. An alternative approach is laid out that utilizes object oriented principles to transcend these limitations. Also, this paper will discuss some additional functionality gained from using these techniques.

Introduction

One common type of application is the automation of some task or series of tasks, generating text or graphical output in a catalog or external file, or a data set. Usually the user wants the ability to change various parameters so that the task will perform differently based on the information they provide. Once the task is defined, the user may want to run the same task periodically.

The traditional approach to this problem would be some type of code generation. In SAS you could use macro and/or SCL to generate the code to a file and have the user run that program as needed. This approach has some built-in limitations and drawbacks.

Once the task has been run and the output generated, some systems stop, while others provide the capability to view or print the output. The problems that arise here center around handling output in its different formats.

The focus of this paper is not on the graphical user interface portion of the process. Rather, this paper deals with the development of the code that lies behind the interface. More specifically the code used to perform the requested task.

Sample Application

The best way to look at the new approach, and the benefits it has, is to examine a situation that could occur and contrast the traditional approach with a new technique.

The process that we are going to automate, involves the selection of variables from a data set that will then be run through various analyses. The initial analysis will consist of the generation of simple summary statistics, and graphical representations of the data. Once these processes have been run, the output will be available for viewing and printing.

Traditional approach: Task

Let us first look at the process of developing the code that will perform the task of generating the simple summary statistics. The information that will be collected from the user is the following:

1. Data set of interest
2. Variables to be analyzed
3. Statistics to be calculated
4. Class variables to be used

One mechanism that is often used to generate the code is SAS Macro. The approach here is to take the parameters and simply pass them into the macro. There are several problems that arise when this approach is taken. When using the macro facility in this manner for application development, you are not taking advantage of the tools that are native to SAS/AF® and SCL. It is not a good idea to switch between languages when there is no advantage. Everything that can be done in macro can be accomplished in SCL. By using SCL you eliminate the potential for confusion that arises in a program when changing the thought processes of two languages.

Even though this is a simple example, one key limitation of this approach is its lack of flexibility. This macro is set up to run analysis on two variables and produce two statistics. What if you wanted to perform the analysis on more variables or calculate additional statistics? The problem with adding more parameters to the macro is that it will just make the macro more complicated and will not really solve the problem. Someone might suggest that the macro could be called multiple times for each set of variables. That technique would not be very efficient, because it causes the data to be read multiple times. You want to design the code to be extensible and efficient. As the needs of the end user change, your code needs to be able to change along with them.

Let us now take a look at how this task is generally accomplished using SCL.

```
submit;
   proc means data = &data
endsubmit;

do I = 1 to listlen(stat_l);
   stat = getitemc(stat_l,I);
   submit;
      &stat
   endsubmit;
end;

submit;
   ;
   var
endsubmit;

do I = 1 to listlen(var_l);
   var = getitemc(var_l,I);
   submit;
      &var
   endsubmit;
end;

submit;
   ;
   class
endsubmit;

do I = 1 to listlen(classv_l);
   class_var = getitemc(classv_l,I);
   submit;
      &class_var
   endsubmit;
end;

submit;
   ;
   output out=stats
endsubmit;

do I = 1 to listlen(stat_l);
   stat = getitemc(stat_l,I);
   submit;
      &stat=
   endsubmit;
   do j = 1 to listlen(var_l);
```

```
      var = getitemc(var_1,I);
      submit;
         &stat.&var
      endsubmit;
   end;
end;

submit;
   ;
   by regimen;
   run;

   proc print data = stats;
   run;
endsubmit;

ignored = filename('statcode', 'c:\temp\stat.sas');
ignored = preview('file', 'statcode');

submit continue;
endsubmit;
```

Using SCL allows the utilization of SCL lists to store the information collected from the user. By doing this I solve the problem of flexibility that was evident in the macro technique, and I am staying within one language.

One feature that users often want is the ability to define a task and then run that task periodically. Under the current scenario, the application would need to be set up to recall the code written to the "stat.sas" file and then submit that code. If the user wants to run that code with some slight modifications, they would be required either to manually modify the code or recreate the task from scratch. Neither of these solutions is desirable. If either of these solutions were implemented, it would require an additional interface to perform the modification. Later in this paper I will present a solution to all of these problems. Now that I have addressed the area of flexibility in dealing with additional parameters, the next area concern is the graphical representation of the analysis that was performed.

Normally the code used to perform the analysis resides behind the frame that is used to collect the information. Now that it is time to add the graphing functionality to the system, it would seem reasonable to add the code that generates the graph to the end of the code that does the analysis. The problem with doing this centers on expansion. When the time arises to add additional analysis or graphical representation, the code now begins to become cumbersome. The more code that is added, the greater the possibility of corrupting the existing code. The logical answer to this problem is to separate the code. This raises the question of how to achieve this separation. This is a topic I will discuss later.

Traditional approach: Output

Ultimately, the result of running the various analyses is the generation of output. This output could be in various formats. Generally, the output is either text-based or graphical. Traditionally, dealing with these two formats causes us to develop two separate interfaces to interact with the output. One frame would be developed to view and interact with the text-based output and a second frame for the graphic output. Having two frames is not the problem. The problem is that the functionality of these two frames is essentially the same. Generally there are only a few functions that you will perform on any output. The code behind the frame is based on what those actions are being performed on, rather than the actions you want to perform.

A Better Way

Now we will begin to explore the utilization of object oriented techniques to deal with these problems. One of the key concepts in object oriented programming is the separation of the visual portion of a system from the non-visual. Employing this principle allows the visual and non-visual portions of the system to vary independently. Additionally, this will help alleviate the problems that arise if we added the code to perform the graphical analysis to the frame code.

The first step when applying this principle is to identity the logical areas, or packages, of the system. As you go through this process, visual and non-visual portions of the system will begin to separate themselves. The diagram in Figure 1 illustrates this separation.

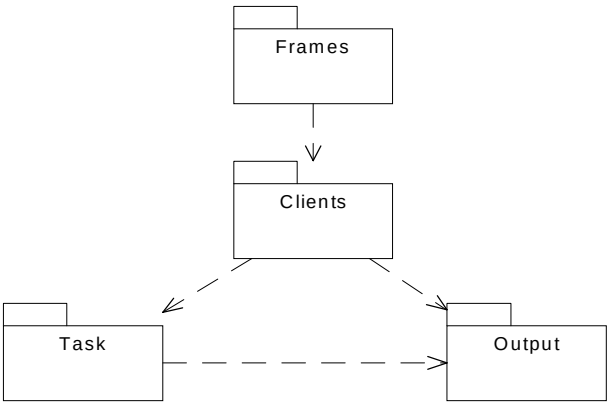

Figure 1

The Frames package will hold all of the screens used to collect the information from the user and the screens used to view the resulting output. This separation occurs because all of the processes that are involved in collecting the parameters used to generate our analysis have nothing to do with how we perform the analysis. The Clients package essentially acts as an intermediary. The classes in this package will store the information collected from the frames and pass the necessary elements to the task and initiate the running of the task. These classes will also help to organize and arrange the output that is generated. The Task and Output packages serve as the containers for the main elements of the framework.

The Framework

A framework outlines a series of classes that are designed to work together to provide a reusable structure and guideline for solving problems within an application. The use of abstraction with the framework allows it to become customizable for the specific needs of a particular application. Figure 3 diagrams the structure of the Task Output framework.

Task

The Task package contains a series of classes that are used to set up and implement any functionality that would be performed on a task. Obviously the main action that is performed on any task is running it. This action seems fairly straightforward. However there are some issues that need to be considered. Let's examine the analysis we were attempting to perform earlier.

The analysis consisted of two distinct yet related task. First we wished to generate some simple summary statistics and then generate a graphical representation of the results. I stated earlier that the code for these two tasks should be separated so that we can avoid the problems that arise when code is added to an existing system. Even though these are separate processes, we want to run them together. This could potentially pose a problem to the client. The client represents the entity that is responsible for initiating the analysis process. If the client were set up to initiate each process separately, this would once again create a problem with expansion. As the system expands, classes from both the Client and the Task packages would undergo changes. The client would have to be

modified for each new process and each new task. It would be preferable if the client did not need to know how many processes where involved in running a task. By utilizing abstraction, inheritance, and polymorphism, a class structure can be set up that allows us to achieve this goal. Figure 2 illustrates this class structure.

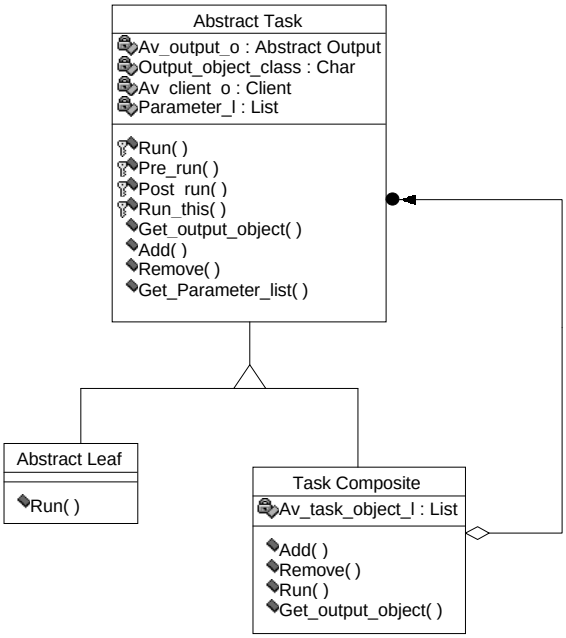

Figure 2

This structure represents an implementation of the Composite pattern (Gamma 1995, page 163). Utilization of this pattern allows us to treat Leafs and Composites equally. The Abstract Task class declares an interface for objects in the composition and implements any default behavior. The Abstract Leaf class acts as the parent for individual processes. This class also provides the base implementation of the Run method for all subsequent processes. The Task Composite class implements the functionality of any task that is composed of other objects from this hierarchy.

The key element in this hierarchy is the interface defined in the Abstract Task class. This singular interface will allow the client to remain unaware of the number of processes that are involved in a task. The majority of the methods in this class have no implementation. This is either because their functionality varies depending on whether or not the object can have children, and/or they are the methods that will implement the analysis. It may seem strange that the Add and Remove methods are part of the interface of the Abstract Task class. This was done because I chose to stress transparency. By doing this, it is possible for the client to be unaware of the type of the task object with which it is dealing.

Let us now examine the four methods that will be used to run the analysis. The purpose of the **Pre_run** method is to implement any functionality prior to running the core of the task. The **Run_this** method will implement the core functionality of the task. The **Post_run** method is used to wrap-up the analysis process. The functionality of the **Run** method will vary depending on the subclass.

Abstract Leaf Run Method

```
RUN:
   method
   ;
   call super(_self_, 'run');
   call send(_self_, 'pre_run');
   call send(_self_, 'run_this');
   call send(_self_, 'post_run');
   endmethod;
```

Task Composite Run Method

```
RUN:
   method ;
   call super(_self_, 'run');
   call send(_self_, 'pre_run');

   /* Call the run method for all task
      objects contained by the composite */
   do i = 1 to listlen(av_task_object_l);
      call send(getitemn(av_task_object_l,i),
                'run');
   end;

   call send(_self_, 'post_run');
   endmethod;
```

The **Run** method for the Abstract Leaf class steps through the process of running the other three methods that run the analysis. Setting up the **Run** method in this fashion allows subclasses to implement the functionality of the three remaining methods that make-up the run process.

The **Run** method for the Task Composite class follows a similar path to that of the Abstract Task Leaf, with two exceptions. This particular implementation does not initiate the **Run_this** method on itself. The main purpose of the composite class is to initiate the **Run** method on its children. The contained Task objects are maintained in the **Av_task_object_l** list instance variable. Looping through this list allows us to initiate operations on child objects.

The **Parameter_l** instance variable is used to store the parameters needed to run the analysis. There are several different techniques that can be used to pass this information to the task. One implementation would involve passing the list in as a parameter of the Run method. Another possible solution would be to pass the list in as a parameter when the task object is created.

Earlier, I mentioned that users often wish to recall a previously-run task and submit that task again. By storing the information required to run the task with the task, we can now utilize the same interface to run the task again. Simply calling the Run method on the task object will rerun the task. If the user wishes to modify some of the parameters used to run the task, the **Get_parameter_list** method can be used to retrieve that information. This allows the utilization of the same GUI that collects the parameters to modify the parameters and rerun the task. Since objects reside in memory, we need to store the necessary information to re-run the task before leaving the SAS session. The answer is persistent storage (Andrew Norton, 1996). I am not going to discuss the details of persistent storage. However, the basic concept lets us store the state of an object and then recreate that object with its old state intact.

Output

The Output package contains the classes that will set up and implement any functionality that is performed on an output object. There are three main actions that are taken on any output, no matter the format. Those actions are printing, viewing, and saving to an external file. How can all of these functions be implemented when you are dealing with two distinct formats (graph and text) of output? The solution to this problem, once again, is in the interface. Designing one interface will allow us to switch between the two formats at run time. This will also allow one GUI to work with both

formats. Figure 4 illustrates a class hierarchy that will help achieve this goal.

The Abstract Output class declares an interface for objects in the composition and implements any default behavior. The Abstract Output Leaf class acts as the parent for the two output formats. This class provides the base implementation of the **Get_output_location** method for all subsequent formats. The Output Composite class implements the functionality of any output that is composed of other objects from this hierarchy. The Graph class implements the core functionality defined by the Abstract Output class for graphic output. The Text class implements the core functionality defined by the Abstract Output class for text-based output.

Once again the Abstract Output class does not implement the core functionality of the hierarchy. The only implementation in this class is that of the **Get** and **Set** methods for the instance variables. One thing that might appear odd is that the **Output_type** variable is set to "Catalog" and yet we have other instance variables that seem to refer to external files. If you think about it, even a SAS catalog entry can be viewed has having a path, name and extension. Essentially the path for a catalog is the library and catalog. The filename is simply the name of the catalog entry while the extension is the entry type. The **Get_output_location** method of the Abstract Output Leaf class is used to build name of the location.

Abstract Output Leaf:
Get_output_location Method

```
GETOLOC:
   method
      out_output_location $
   ;
   call super(_self_,'get_output_location',
              out_output_location);
   call send(_self_, 'get_path', path);
   call send(_self_, 'get_file_name',
             file_name);
   call send(_self_, 'get_extension',
             extension);
   out_output_location = path ||'.'||
      file_name ||'.'|| extension;
   endmethod;
```

One function that is an integral component of output objects is the redirection of output to the appropriate location. Traditionally, when a task is run, that task redirects the output to a location of its choice. However, the actual process of redirecting output is not the responsibility of the task. The task can tell the output object where it wants the output relocated, but it should not perform that redirection. The specification of the location of the output can be set on the output object from the **Pre_run** method of the task class. The redirection could then be implemented in the **Pre_process** method of the output object.

The augmented **Pre_run** method would look as follows.

Task: Pre_run Method

```
PRE_RUN:
   method
   ;
   if av_output_o not in(.,_blank_) then do;
      call send(av_output_o, 'set_path',
                'work.sugi');
      call send(av_output_o, 'set_file_name',
                'descstat');
      call send(av_output_o, 'pre_process');

    endmethod;
```

Text: Pre_process Method

```
PREPROC:
   method
   ;
   call super(_self_, 'pre_process');
   call send(_self_, 'get_output_location',
             output_location);
   submit continue;
      proc printto print = &output_location
           new;
      run;
   endsubmit;
   endmethod;
```

Let us now examine the implementation of the other core functionality for the Text class.

Text: Print Method

```
PRINT:
   method
   ;
   call super(_self_, 'print');
   call send(_self_, 'get_output_location',
             output_location);
   ignored = preview('copy',
                     output_location);
   ignored = preview('print');
   ignored = preview('clear');
   endmethod;
```

Text: Output Method

```
OUTPUT:
   method
   optional = in_dialog 8 in_fname  $ in_rc 8
   ;
   call super(_self_, 'output', in_dialog,
              in_fname, in_rc);
   if in_dialog in(_blank_, .)
      then in_dialog = 1;
   if in_dialog then
      filedlg_rc=filedialog('saveas',fname,'
        ',' ','*.lst','*.txt','*.*');
   else do;
      filedlg_rc = in_rc;
      fname = in_fname;
   end;

   call send(_self_, 'get_output_location',
             output_location);
   if filedlg_rc in(0,1,2) then do;
      ignored = preview('clear');
      ignored = preview('copy',
                output_location);
      ignored = filename('myfile',fname);
      select;
         when(filedlg_rc in(0,1))
            ignored =
               preview('file','myfile');
         when(filedlg_rc = 2)
            ignored =
               preview('file','myfile',
                       'append');
         otherwise;
      end;
      ignored = preview('clear');
      ignored = filename('myfile','');
   end;
   endmethod;
```

Text: View Method

```
VIEW:
   method in_parent_o 8
   optional = in_frame 8
   ;
   call super(_self_, 'view', in_parent_o,
              in_frame
   region_l = makelist();
   call send(in_parent_o, '_is_swapped_',
             swapped);
   if swapped then
      call send(in_parent_o, '_swap_in_');
   call send(in_parent_o, '_get_region_',
             region_l,'P');
   call send(in_parent_o, '_swap_out_');
*******************************************
   This section build the list of default
   widget attributes
*******************************************;
   default_l = makelist();
   /* Add the frame to the default list */
   dummy = setnitemn(default_l, in_frame,
           '_frame_');
   /* Add the region list to default list */
   dummy = setniteml(default_l,region_l,
           '_region_'
   /* Load and create the viewer */
   viewer_c=loadclass('sashelp.fsp.catview');
   call send(viewer_c, '_new_', viewer_o,
             default_l);
   /* Set the instance variable */
   dummy = setnitemn(_self_, viewer_o,
           'viewer_o');
   /* Set the catalog entry on the viewer */
   call send(_self_, 'get_output_location',
             output_location);
   call send(viewer_o, '_set_entry_',
             output_location);
   ignored = dellist(default_l,'y');
   endmethod;
```

The design of the View method allows the use of one frame for working with both graphical and text based output. The key to this design is the dynamic loading of the appropriate viewer into an area of a frame. This same technique can be used by the Graph class. The Graph class would load the SAS/GRAPH® Output class as the viewer.

Additional Benefits

The utilization of the Composite pattern yields additional benefits for each half of the framework. One of the key principles at work in the Composite pattern is polymorphism. This principal allows the run-time switching of objects. This allows the developer to provide the user with the capability to build or chain a sequence of tasks at runtime. Remember, composites can not only contain leafs but also other composites. Therefore it does not matter how the individual task was set up at design time. Any combination of leafs and composites can be chained together at run time to build one large task.

The same principal can be used with output objects to group and arrange output into larger reports. Because of the singular interface these reports can be made up of either text or graphs.

Conclusion

Take advantage of the tools that are provided. SCL, when used in conjunction with base SAS, can provide a powerful tool for applications development. SAS/AF and SCL provide a good foundation for applying object oriented principles in the design of applications.

The Task-Output framework takes advantage of object oriented principles to provide the applications developer with a tool that can be used in a variety of applications. Ultimately, the framework provides the user of the application with a system that is both flexible and extensible.

All of the method implementations that I have provided only serve as a guide. Depending on your application and situation, your implementation may vary.

References

Gamma, Erich, Richard Helm, Ralph Johnson, John Vlissides (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley Publishing Company.

Norton, Andrew A. (1996), "Persistent Storage of SCL Data Objects," in *Proceedings of the Twenty-First Annual SAS Users Group International Conference*. Cary, NC: SAS Institute, Inc.

Acknowledgments

I would like to thank Jack Fuller and Steve St. Peter for their assistance in reviewing this paper and in the development of the concepts that have been discussed.

The author may be contacted at:

John E. Ellis
Lead Software Engineer
Trilogy Consulting Corp.
5278 Lovers Lane
Kalamazoo Mi, 49002
616 344-4100
Email: Jeellis@Trilogyusa.com

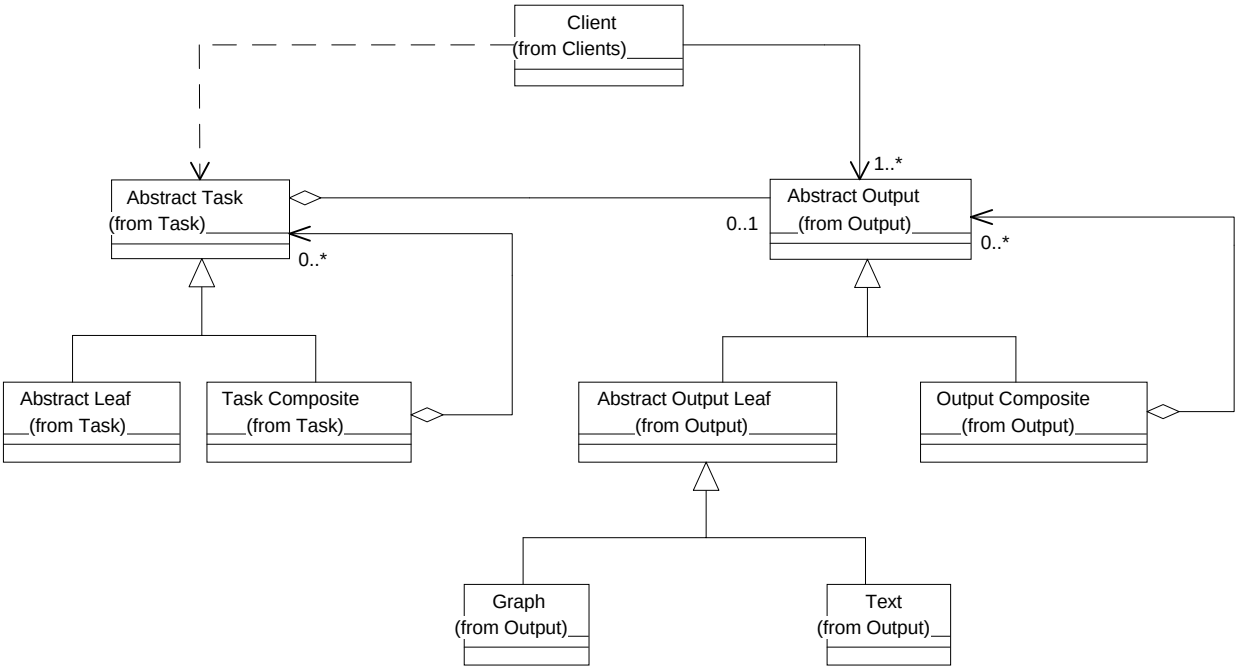

Figure 3

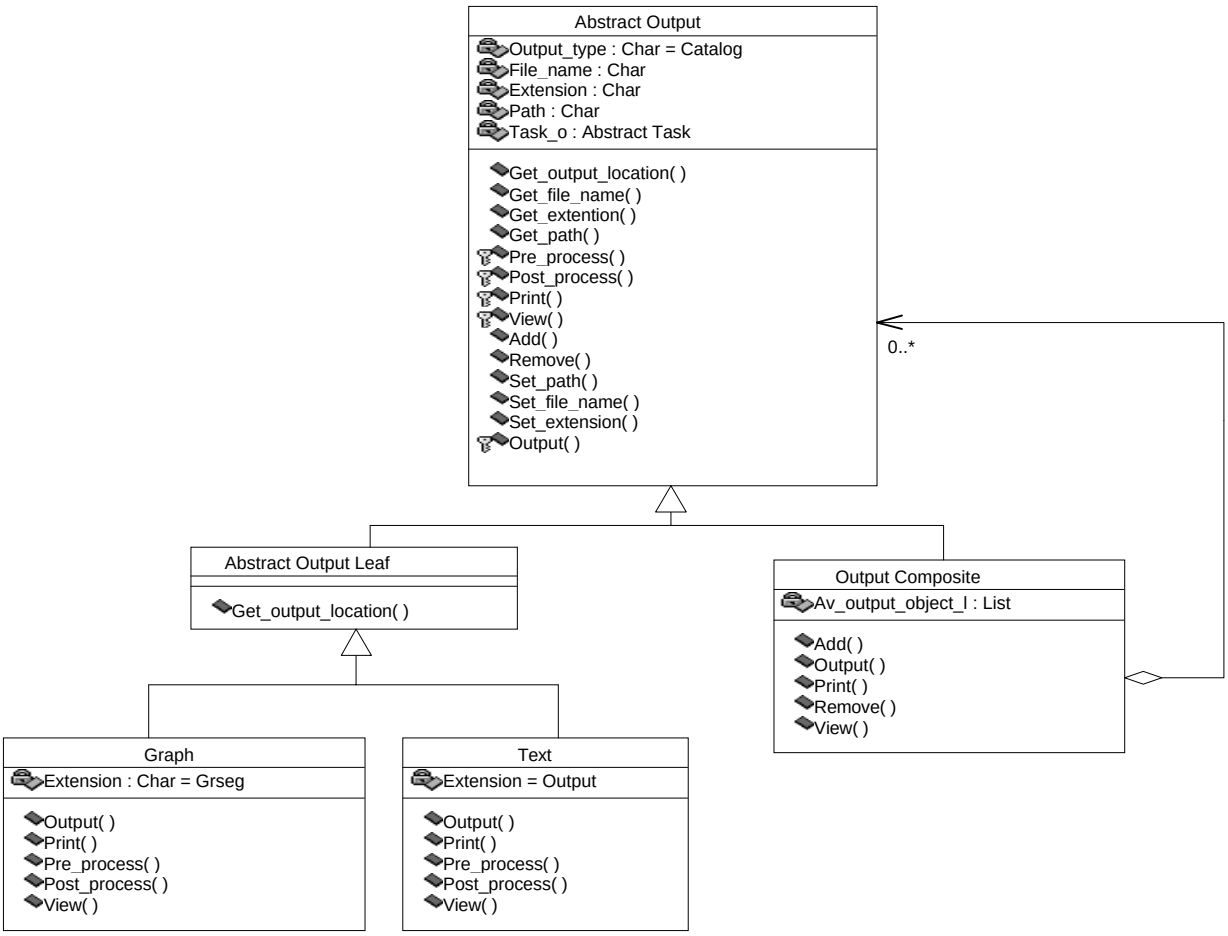

Figure 4