SWE621 – HW4

Group:
Venkata Krishna Chaitanya Chirravuri **(G01336659)**
Sai Prashanth Reddy Kethiri **(G01322333)**

Reference System – VisTrails
Additional Systems – Visualization Tool Kit (VTK), matplotlib

1. VisTrails

a) The design pattern identified is the **Bridge Structural Design Pattern**. Bridge is a structural design pattern that lets us split a large class or a set of closely related classes into few separate hierarchies—abstraction and implementation—which can be developed independently of each other.

The **problem** is while working with the workflow, the user intends to create, modify and update the workflow according to the requirements and dependencies from the VisTrails GUI i.e.., Abstraction layer. With all the available options in the Vistrails GUI it's not possible to have dedicated classes and their extended classes in the code base, as it results in very complex class hierarchies—abstractions as a whole together layer code base. As they can't have dedicated class-code written for each combination of the instantiated objects available.

So instead the developers **tackled/solved** this problem by using Bridge Structural Design Pattern, rather than creating pipeline/workflow code for all the available scenarios, they have splitted and defined classes which are related to each other and have referenced them later in the pipeline creation. Following this design pattern, it gives leverage to the developer to avoid the explosion of the class-hierarchies by transforming it into several related class hierarchies. This design pattern achieves this by separating the abstraction layer from the implementation layer.

Here the pipeline execution in the abstraction layer is available in the ***VisTrails/vistrails/gui/vistrail_view.py*** : The file describes a container widget consisting of a pipeline view and a version tree for each opened Vistrail.

class QVistrailView(QtGui.QWidget) : QVistrailView is a widget containing four stacked widgets: Pipeline View, Version Tree View, Query View and Parameter Exploration view for manipulating vistrails.
The class and the file have few dependencies on other defined modules like:
vistrails.gui.collection.vis_log : This modules builds a widget to visualize workflow execution logs
vistrails.gui.common_widgets : These common widgets use the interface of VisTrails. These are only simple widgets in terms of coding and additional features. It should have no interaction with VisTrail core
vistrails.gui.mashups.mashup_view : This module offers all the methods for updation view
vistrails.gui.module_info : This module defines the information of the module
vistrails.gui.paramexplore.pe_view : This module gui operations for has ParameterExploration
vistrails.gui.pipeline_view : This module has gui front end operations for pipeline view
vistrails.gui.ports_pane : This module has gui front end operations for ports_pane
vistrails.gui.query_view : This module has gui front end operations for query_view
vistrails.gui.version_view : This module has gui front end operations for version_view
vistrails.gui.vis_diff : This module has gui front end operations for vis_diff
vistrails.gui.vistrail_controller : This module has gui front end operations for vistrail_controller

The implementation layer:
VisTrails/vistrails/core/vistrail/vistrail.py where all the required classes and their methods are defined for all the implementations.
class Vistrail(DBVistrail): Vistrail is the base class for storing versioned pipelines.
This file and the class have dependencies on other modules like:
vistrails.core.vistrail.action.Action & **vistrails.core.vistrail.action_annotation**: These define the constructors and copy,
vistrails.core.vistrail.module : Represents a module from a Pipeline ,
vistrails.core.data_structures : This has graph, point, queuerect and stack ,

vistrails.core.paramexplore.paramexplore : This file contains the definition of the class ParameterExploration

The **constraints** here are that some of the modules defined in the dependencies are prohibited in communicating among vistrails.core and vistrails.gui. **(Goal)** By splitting a monolithic class into several classes, so you can change the classes in each hierarchy independently of the classes in the others. This brings easy maintainability and minimizes the risk of breaking existing code.

The main **advantage** of using the bridge structural design pattern between GUI and API is, it gives the option of extensibility, where you can introduce new abstractions and implementations independently from each other. The client code works with high-level abstractions. It isn't exposed to the platform details. Since a huge class is split into several class hierarchies, you can change the classes in each hierarchy independently of the classes in the others.

The **disadvantage** of using Bridge Design Pattern is it divides a big class that contains both the abstraction and the implementation into two separate class hierarchies. Hence, we need to manage two different classes along with their subclasses instead of one big class, thereby increasing the code complexity. The performance of the system also gets affected because of the indirection of the request from the Abstraction to the Implementor.

b) The **design pattern** identified is the **Memento Behavioural Design Pattern**. Memento is a behavioral design pattern that lets you save and restore the previous state of an object without revealing the details of its implementation.

The **problem** here can be described as, while creating and working on the workflows, the user would like to backtrack some changes and keep the record of some other things. This also being one of the important architectural goals in building Vistrails i.e.., to provide the provenance infrastructure where every execution and the details of every step can be tracked back. VisTrails transparently tracks changes made to workflows, including all the steps followed in the exploration. This would be possible to know about the state and details of all the objects involved in the workflow. We might not anticipate having access to all the objects involved where the exact details can be replicated at some random stage when needed. This restriction in accessing the state and details of the objects makes it difficult to achieve the provenance architecture.

As some workflows avoid concurrency usage and restrict the access until a workflow is completed. So, avoiding all the complexities, the developers **tackled** this problem by using Memento Behavioural Design Pattern, as it would be merely impossible to keep track of all updated information on the objects involved in the operations. They have come up with logging the state, properties, and the details of the objects in every operation and every stage at workflow enabling them to replicate the same values, state, and properties at whatever the step they want to retrieve. Keeping the logs of all the states, properties and the objects is a wiser decision to achieve complete provenance infrastructure.

Following this design pattern mitigated all problem of concurrent access of the objects/workflows etc. the historical information on the objects, workflows are accessed through the provenance browser in the abstraction layer:

VisTrails/vistrails/gui/collection/vis_log.py: The GUI for viewing the logs are present here. can

VisTrails/VisTrails/blob/v2.2/doc/usersguide/provenance.rst : This document snippet at extends the discussion about the historical information of the steps involved on a higher-level. Below we have discussed all the lower-level implementations from the front-end scripts to the corresponding backend-scripts where the implementations are defined.

VisTrails/vistrails/core/collection/vistrail.py: The backend part is executed here, where classes and methods for creating, updating the workflow and all the changes done to an existing workflow are recorded with all the minute details and can be reloaded at any given stage of the workflow building process.

VisTrails/vistrails/core/vistrail/controller.py : Is where the saving part of the workflow/pipeline are defined

class VistrailController : This class is defined where all the operations to the workflow like creation, deletion, updating and autosave are defined using this as the super-class. Many methods for retrieving the versioning of pipelines, checking aliases for the modules, existing pipelines and workflows are defined here. The connections to the database and their related port connectivity information are also defined as part of the class.

Cache_pipelines are also specifically defined to cache and speed up the versioning process of the pipelines and workflows.

VisTrails/vistrails/core/collection/vistrail.py : Is where the reload methods to retrieve the state properties of the workflows are defined.

class VistrailEntity(Entity): This class is created in a generic way for all the VisTrails entities and their related operations. Several methods for adding workflows, revoking changes to them, and adding a few parameters are defined here.

The **constraint** that is imposed on the elements is that contents of the memento aren't accessible to any other object except the one that produced it. Other objects must communicate with mementos using a limited interface which may allow fetching the snapshot's metadata (creation time, the name of the performed operation, etc.), but not the original object's state contained in the snapshot. **(Goal)** By this, it's not able to tamper with the state stored inside the memento. At the same time, the originator has access to all fields inside the memento, allowing it to restore its previous state at will.

The **advantage** of using memento behavioral design pattern in VisTrails is, it makes the restoration of object state to its previous state quicker.

But, since it keeps on having checkpoints of object state so that it would be easier to restore to a previous state, the **downside** is that it consumes a lot of RAM. This inturn reduces the overall performance of the system.

c) Third design pattern that we identified in our system is the **Singleton Creational Pattern.** This pattern ensures that a class has only one instance, while providing a global access point to this instance.

   The **problem** here is, when we want to have multiple instances of VisTrails to be used at the same time, imagine creating one workflow and later we want to create a new workflow. In a system that does not follow a singleton design pattern, when we create a new workflow with an already existing one, it will return the existing one instead of returning a new workflow object.

This is **solved** by using a singleton design pattern, where the constructor is made private, restricting the other objects using the new operator. In the application, the static creation method acts as a constructor under which all the calls made by the private constructor creates the object in a static field and all the corresponding calls revoke the object instantiated from the cache. So with this pattern creation, all the other object creation methods are disabled.

VisTrails/vistrails/gui/application.py : This is the main application of vistrail, it will calls for initializations to the theme, packages and the builder class

VistrailsApplicationSingleton(VistrailsApplicationInterface,QtGui.QApplication): ***VistrailsApplicationSingleton*** is the singleton of the application, there will be only one instance of the application during VisTrails.

The initialization must be explicitly signaled. Otherwise, any modules importing vis_application will try to initialize the entire app. Some of the important methods defined here are :
start_application(optionsDict=None, args=[]): Initializes the application singleton.

stop_application(): Stop and finalize the application singleton.
interactiveMode(self): interactiveMode() -> None, Instantiate the GUI for interactive mode

send_notification(self, notification_id, *args): does global notifications

run_single_instance(self, args): code for single instance of the application

The **Constraint** that is imposed by this design pattern on this system is that the class VistrailsApplicationSingleton should have only a single instance, while providing a global access point to this instance. The only way to get its object is to call the getInstance method. This method caches the first created object and returns it in all subsequent calls.

The **Pros** of using this design pattern is that we can be sure that there is only a single instance for this class for which we can have global access. This singleton object is initialized only when it is requested for the first time.

Coming to the **disadvantages** of this design pattern are, it may be difficult to unit test the client code of the Singleton because many test frameworks rely on inheritance when producing mock objects. Since the constructor of the singleton class is private and overriding static methods is impossible in most languages.

2. Visualization Tool Kit (VTK)

a) The **closest equivalent problem** that we found to our main system is to create 2D charts that scale well to large data sets.

There have been several **techniques to accomplish** this goal, mainly centered around passing arrays to drawing functions to draw multiple 2D primitives using only one function call. Most rendering backends expect coordinates to be packed in memory in certain ways. One of the most common is as a 1D array with a length tuple times number of points. The vtkPoints2D and vtkPoints3D classes achieve this packing in their underlying data structure. Using this data structure will lead to the best performance, otherwise the context must repack the arrays (such as data coming from multiple columns in a table).

The 2D API is currently composed of two levels, a concrete class called vtkContext2D that is called by the paint functions of components operating within the 2D API, and a vtkContextDevice2D which is called by the vtkContext2D to actually draw to a context. The vtkContext2D contains a pointer to the derived class of the vtkContextDevice2D to do low level painting. This is the class that must be implemented for a new backend to be supported. The vtkContext2D builds up more complex 2D constructs on top of the basic constructs implemented in the device. This a **bridge design pattern** where Abstraction is vtkContext2D, Implementer is vtkContextDevice2D and Concrete Implementers are vtkOpenGLContextDevice2D and vtkQtContextDevice2D. There are **no constraints** imposed on the system by making use of this design pattern.

Bridge Pattern divides the code of an app that manages devices and their remote controls. The vtkContextDevice2D classes act as the implementation, whereas the vtkContextArea acts as the abstraction.

Abstraction
vtkContext2D is an abstract class to draw 2D primitives. In this sense a ContextDevice is a class used to paint 2D primitives onto a device, such as an OpenGL context or a QGraphicsView.

vtkAbstractContextBufferId::Superclass - An 2D array where each element is the id of an entity drawn at the given pixel. The effective/concrete subclass vtkContextBufferId stores the whole buffer in RAM.

vtkAbstractContextItem - This class is the common base for all context scene items.

vtkBlockItem - It draws a block of the given dimensions, and reacts to mouse events.

vtkBrush - provides a brush that fills shapes drawn by vtkContext2D.

vtkContextActor - provides a vtkProp derived object. This object provides the entry point for the vtkContextScene to be rendered in a vtkRenderer.

vtkContextClip - This class can be used to clip the rendering of an item inside a rectangular area.

vtkContextDevice2D - Abstract class for drawing 2D primitives.

vtkContextDevice3D - Abstract class for drawing 3D primitives.

vtkCpntextKeyEvent - Provides a convenient data structure to represent key events in the vtkContextScene. Passed to vtkAbstractContextItem objects.

vtkContextMapper2D - This class provides an abstract base for 2D context mappers. They only accept vtkTable objects as input.

vtkContextMouseEvent - Provides a convenient data structure to represent mouse events in the vtkContextScene.

vtkContextScene - Provides a 2D scene that vtkContextItem objects can be added to. Manages the items, ensures that they are rendered at the right times and passes on mouse events.

vtkContextTransform - Transforms a point from the parent coordinate system.

Implementation
vtkContextArea provides a clipped drawing area surrounded by four axes. The drawing area is transformed to map the 2D area described by DrawAreaBounds into pixel coordinates. DrawAreaBounds is also used to configure the axes. Item to be rendered in the draw area should be added to the context item returned by GetDrawAreaItem().

The size and shape of the draw area is configured by the following member variables:

- Geometry: The rect (pixel coordinates) defining the location of the context area in the scene. This includes the draw area and axis ticks/labels.
- FillViewport: If true (default), Geometry is set to span the size returned by vtkContextDevice2D::GetViewportSize().
- DrawAreaResizeBehavior: Controls how the draw area should be shaped. Available options: Expand (default), FixedAspect, FixedRect, FixedMargins.
- FixedAspect: Aspect ratio to enforce for FixedAspect resize behavior.
- FixedRect: Rect used to enforce FixedRect resize behavior.
- FixedMargins: Margins to enforce for FixedMargins resize behavior.

Both my main system (VisTrails) and the reference system (VTK) considered the same design pattern for the similar problems they faced. So, comparing both of them, they do not have an **advantage** of one system over the other. Implementing bridge design patterns in both the systems resulted in the same **disadvantages** like increase in code complexity and decrease in the performance of the system.

b) Another **closest problem** that we found on our additional system compared to our main system is backwards compatibility. A user while creating a workflow might want to make few changes to their prior decisions or might make few changes to the existing workflow or might revert the changes they made. (**Note -** This feature is only available to VTK4. In the later versions of VTK this feature has been removed to simplify data object classes and to allow for the separation of data and execution model libraries as part of the ongoing modularization effort. This change causes a few incompatibilities.)

So, the **solution** for this problem in VTK V4 is having metadata of the workflow. Older versions of VTK used to have classes like vtkProcessObject, vtkSource and a few other subclasses that kept track of the versions and the workflow. They achieved this by adapting the **factory method** of creational design pattern, where an interface for creating objects in a superclass is provided, but allows subclasses to alter the type of objects that will be created.

In VTK V4 it used to have following classes and subclasses to backtrack or make changes to an existing workflow,

vtkDataSetSource - an abstract class that specifies an interface for dataset objects. It also provides methods to provide information about the data, such as center, bounding box, and representative length.

vtkDataObjectSource - an abstract object that specifies behavior and interface of field source objects. Field source objects are source objects that create vtkFieldData (field data) on output.

vtkImageSource **-** Source of data for the imaging pipeline.

vtkPointSetSource - abstract class whose subclasses generate point data

vtkPolyDataSource **-** vtkPolyData is a data object that is a concrete implementation of vtkDataSet. vtkPolyData represents a geometric structure consisting of vertices, lines, polygons, and/or triangle strips. Point and cell attribute values (e.g., scalars, vectors, etc.) also are represented.

vtkProcessObject - vtkProcessObject is an abstract object that specifies behavior and interface of visualization network process objects (sources, filters, mappers). Source objects are creators of visualization data; filters input, process, and output visualization data; and mappers transform data into another form (like rendering primitives or write data to a file). vtkProcessObject fires events for Start and End events before and after object execution (via Execute()). These events can be used for any purpose (e.g., debugging info, highlighting/notifying user interface, etc.)

vtkSource **-** abstract class specifies interface for visualization network source. Source objects are objects that begin the visualization pipeline. Sources include readers (read data from file or communications port) and procedural sources (generate data programmatically). vtkSource objects are also objects that generate output data. In this sense vtkSource is used as a superclass to vtkFilter.

The only **constraint** imposed by this design pattern on the system is to save the system resources by reusing the existing objects rather than building one each time.

By using this design pattern you can only make changes to your workflow in VTK V4 but the later versions of VTK do not have the option of backward compatibility. This is considered a major **disadvantage** in the system. So, VisTrails with the Memento design pattern is far more **advantageous** compared to the present VTK system.

c) Another closest **problem** that we found in our additional system is executing multiple pipelines in VTK. Sometimes, we need to run the same pipeline with different parameters or two different pipelines and compare their outputs.

   This problem can be **solved** by making use of vtkSMPTools (Shared Memory Parallelism). The vtkSMPTools framework implements a thin wrapper around existing tools such as TBB and OpenMP, and it provides a simple application programming interface (API) for writing parallel code.

   A thread local object helps to safely store variables in a parallel context. It maintains a copy of an object of the template type for each thread that processes data, it creates a storage for all threads. The actual objects are created the first time Local() is called. When using the thread local storage object, a **common design pattern is to write/accumulate data (Active Object Design Pattern)** to local objects when executing in parallel. Then in a sequential reduce step, iterates over the whole storage to do a final accumulation.

   vtkSMPTools **-** provides a set of utility functions that can be used to parallelize parts of VTK code using multiple threads.

   vtkSMPThreadLocalObject - creates storage for all threads but the actual objects are created the first time Local() is called.

   vtkMultithreader - is a class that provides support for multithreaded execution using pthreads.This class can be used to execute a single method on multiple threads, or to specify a method per thread.

vtkMultiProcessController - used to control multiple processes in a distributed computing environment.

vtkMPIController **-** concrete class which implements the abstract multi-process control methods defined in vtkMultiProcessController using MPI (Message Passing Interface).

vtkParallelRenderManager **-** operates in multiple processes. It provides proper renderers and render windows for performing the parallel rendering correctly.

The main **advantage** of using active object design pattern is it allows one or more independent threads of execution to interleave their access to data modeled as a single object. With this design pattern, VTK achieved data parallelism.

(Disadvantage) VTK's data is not simple. There is a lot of extra stuff added (reference counting, vtkInformation support) that could talk to a lot of other objects. Reference counting alone looks very thread-unsafe and to make things more complicated, there seems to be extra code to side-step garbage collection when the objects are created outside of the "main thread" - possibly to support threaded algorithms that need to create their own working space? Therefore, for larger datasets you wouldn't want or need multiple pipelines executing in the same process. Instead you would be distributing the workload across a large cluster or an N-way machine, and the main thread is no longer the limiting factor. Hence, VTK's current implementation of multi threads is considered to be non-trivial and pointless.

So, the design pattern used in VisTrails for working with multiple workflows at the same time is far more efficient and useful.

3. MATPLOTLIB

a) The **equivalent problem** we found in matplotlib is for plotting a graph with selected parameters either by importing matplotlib as a library, or through interactive GUI like choosing a 2D plot like histogram, bar plot etc, then choosing the axes and stuff. The code cannot be explicitly extended for each of the combinations as it's not always defined whether it is imported as a library or being used as a GUI for plotting the graphs. This complexity arises from many alarming errors while integrating into other 3rd party tools, and we might also encounter many other compatibility and versioning issues as well.

This has been **tackled** using the **Abstract Factory Method** where the objects that are related are defined as single classes and are referenced to each other and the **Bridge Design** method for linking the GUI and the matplotlib implementation. This way the complex and explosive coding can be avoided, and code reusability is leveraged enabling the users to easily handle the compatibility issues as well. All the related objects are grouped and defined in a single class and later the implementation layer is separated from the abstraction layer. This makes the abstraction and the implementation layers independent of each other giving enough flexibility of interchanging the way to interact with the implementation layer in many ways.

matplotlib/lib/matplotlib/backends/qt_editor/figureoptions.py : This file has all the options for choosing different styles for plotting the figures onto the figure canvas.

LINESTYLES = {'-': 'Solid','--': 'Dashed','-.': 'DashDot',':': 'Dotted','None': 'None',}

DRAWSTYLES = {'default': 'Default','steps-pre': 'Steps (Pre)', 'steps': 'Steps (Pre)','steps-mid': 'Steps (Mid)','steps-post': 'Steps (Post)'}

figure_edit(axes, parent=None):Edit matplotlib figure options

prepare_data(d, init):Prepare entry for Form Layout.

matplotlib/lib/matplotlib/widgets.py : GUI neutral widgets, Widgets that are designed to work for any of the GUI backends. All of these widgets required are predefined a `matplotlib.axes.Axes`instance and pass that as the first parameter.

class LockDraw: Some widgets, like the cursor, draw onto the canvas, and this is not desirable under all circumstances, like when the toolbar is in zoom-to-rect mode and drawing a rectangle. To avoid this, a widget can acquire a canvas' lock with``canvas.widgetlock(widget)`` before drawing on the canvas; this will prevent other widgets from doing so at the same time.

class Widget: Abstract base class for GUI neutral widgets.

class AxesWidget(Widget):Widget connected to a single `~matplotlib.axes.Axes`. To guarantee that the widget remains responsive and not garbage-collected, a reference to the object should be maintained by the user.

class Button (AxesWidget): A GUI neutral button.

Constraint imposed on the system is: Use the Abstract Factory when your code needs to work with various families of related products, but you don't want it to depend on the concrete classes of those products—they might be unknown beforehand or you simply want to allow for future extensibility.

The **advantage** of this system over the main system is that the pairing of the both abstract factory and bridge pattern becomes useful when some abstractions defined by *Bridge* can only work with specific implementations. In this case, *Abstract Factory* can encapsulate these relations and hide the complexity from the client code.

b) Matplotlib can be used in two ways, either by importing matplotlib as an external library or using any third-party GUI and using matplotlib as an external backend source. Matplotlib does not require creating workflows or creating pipelines. The most **equivalent problem** we found for backtracking any changes, steps and reloading the state properties in VisTrails is similar to the reloading or retrieving the previous plot/figure using matplotlib. We intend to discuss here about retrieving the plot/figure or state of the plot or caching in similar terms.

Similar to the memento design pattern in VisTrails, matplotlib doesn't have any explicit ways of storing all the state, properties and values of the object for plotting the graphs/figures. But, the **solution** to this problem can be achieved using the caching can be achieved using external libraries in addition to matplotlib. Though matplotlib still offers api methods for retrieving the recent plots, it cannot be scaled for backtracking all the other steps and retrieving other properties. We have to clear the current figure or clear the axes for deleting it permanently.

matplotlib/lib/matplotlib/pyplot.py : pyplot.py has all the methods defined here for plotting the figure to make changes to them.
The interactive part of matplotlib designs are described in this script.

show(*args, **kwargs): Display all open figures.

close(fig=None):Close a figure window.

clf():Clear the current figure.

draw():Redraw the current figure.

axes(arg=None, **kwargs):Add an Axes to the current figure and make it the current Axes.

delaxes(ax=None):Remove an `~.axes.Axes` (defaulting to the current axes) from its figure.

sca(ax):Set the current Axes to *ax* and the current Figure to the parent of *ax*.

cla():Clear the current axes.

Since, we cannot backtrack our results in matplotlib, it is considered as one of the **disadvantages** in our additional system when compared with VisTrails. So, it is clear that the main system (VisTrails) chose a better decision pattern.

c) The closest **problem** that we identified in matplotlib is building multiple graphs in a single plot. Multiple graphs are useful if you want to visualize the same variable but from different angles (e.g. side-by-side histogram and boxplot for a numerical variable).

(Solution) In Matplotlib, we can draw multiple graphs in a single plot by using subplot() function and other by superimposition of the second graph on the first i.e, all graphs will appear on the same plot.

subplot(m,n,p) divides the current figure into an m-by-n grid and creates axes in the position specified by p. The first subplot is the first column of the first row, the second subplot is the second column of the first row, and so on. If axes exist in the specified position, then this command makes the axes the current axes.
This is a wrapper of **Figure.add_subplot**

matplotlib.pyplot.subplots - A subplot () function is a wrapper function which allows the programmer to plot more than one graph in a single figure by just calling it once.

matplotlib/tutorials/intermediate/arranging_axes.py :

- We can create a basic 2-by-2 grid of Axes using `~matplotlib.pyplot.subplots`. It returns a `~matplotlib.figure.Figure` instance and an array of `~matplotlib.axes.Axes` objects. The Axes objects can be used to access methods to place artists on the Axes; here we use `~.Axes.annotate`.

matplotlib/examples/subplots_axes_and_figures/gridspec_and_subplots.py:

- Sometimes we want to combine two subplots in an axes layout created with `~.Figure.subplots`. We can get the `~.gridspec.GridSpec` from the axes and then remove the covered axes and fill the gap with a new bigger axes.

Figure.add_subplot - It consists of a wrapper, that provides additional behavior when working with the implicit API

There will be **constraints** on the size of the graph in a subplot based on the use case. Using subplots is **advantageous**, as we can get different views on the data, making it more handy for the data analysis. Both design patterns of VisTrails and subplots have their own set of advantages, stating that both have equal importance in their own scenarios.There is **no disadvantage** of using subplot.

References

- **https://refactoring.guru/**
- **https://www.vistrails.org//index.php/Main_Page**
- **https://github.com/VisTrails/VisTrails**
- **https://vtk.org/doc/nightly/html/**
- **https://github.com/Kitware/VTK**
- **https://matplotlib.org/stable/**
- **https://github.com/matplotlib/matplotlib**