Framework: Getting Started

State: July 18th , 2019

Note:

This is not a tutorial that tells you how to get started with Unity and C#.
Some basics will be told but we assume that you already have most of this knowledge and are able to work with it properly.

1. Framework Startup Behaviour

As you may know, a Scene in Unity mainly consists of *GameObjects*.
To bring logical behaviour to them, it is possible to add so called Components to them that are scripts we wrote.
As we only work with C# in this application, all of the scripts we talk about are C# scripts.
When you add scripts to *GameObjects*, Unity will call them in a specific way, as they usually derive from a base class called *MonoBehaviour*. With "in a specific way" we mean, that there are a bunch of methods that will be called at specific events.
For instance, there is the *Monobehaviour.Start()* method which is called when a script is enabled and before every *Monobehaviour.Update()* method. The ***Update()*** methods of scripts are called every frame.
Now imagine if you have a script that is based on the result of another script that should run previously.
How would you accomplish this?
You could add boolean variables to store the state and "synchronize" both ***Update()*** methods.
This will probably work for easy scenarios but as we do not know which script is called at first, we may skip some frames.
A case like this is not ideal so that we have to find another solution.
To get to the point, the framework is currently loading all the data on application startup.
Therefore, it is required that some scripts run after the loading process of the according data finished.
There is currently no "hub" or initial room that lets us configure which workspace to load and how.
This is already a major point on our "ToDo" list and should be considered when adding new features or modifying existing ones.
Loading the data and "managing" the step-by-step call of the according scripts is done by the class ***ApplicationLoader***.
It uses a singleton pattern to allow "global" access to its unique instance.
So in case a script requires specific data or wants to have access to a specific visualization, it can get a reference from this class.
Unity also offers a feature called *Script Execution Order* that we plan to make more use of in future versions as well,
but for now, the *ApplicationLoader* exists and does its job for you.

2. Loaders and Spawners

The framework provides a consistent way for loading data and showing visualizations in the virtual environment.
Loading data is performed by "Loader" classes. For example, such are the ***AppConfigLoader*** (loads the app_config.conf file from the workspace), the ***StructureLoader*** (loads the software system directory structure), the ***VariabilityModelLoader*** (loads the feature model) or the ***RegionLoader*** (loads code regions of files). Of course, there are a lot more loaders available.
Something they have in common is their purpose and based on that, the base class they derive from.
This is the reason why there is an abstract loader class that is called ***FileLoader***. It tells that a loader deriving from it loads data from a file on the disk. This way, loaders can be grouped into categories. Some of them need to be executed before others are.
That is the case if they depend on data from another loader. As an example, we can only know which directory to load as the software system when we have loaded the data from the *app_config.conf* file previously.
The ***ApplicationLoader*** takes care of executing the loaders in the desired order in its ***Awake()*** method.
After the loading process finished, we can start to create and show the visualizations. This is where ***Spawners*** come into play.
A spawner is a class that takes previously loaded data and uses it to create an according visualization.
Some of might be executed on startup and some on runtime. This depends on its purpose.
For instance, we may want to show the structure of the software system that we have loaded as soon as the application is started. In contrast, we only want to show the content of files and visualizations related to it when the user desires it.
Unlike loaders, spawners can already be configured in Unity's editor *inspector window*. This way, we can quickly add and remove spawners as well as configure the way they behave (run on startup, be enabled/disabled, …). We have already planned to change the way we use loaders in a similar fashion. As this may change suddenly in the future, consider it in the development process.
The ***ApplicationLoader*** takes care of executing spawners on startup in the desired order in its ***Start()*** method.

To be continued...