Using Workflow Technology to Maximize Software Re-Use	Revision: A Revision Date: 06/11/2020

Introduction

In this article, I would like to discuss a design pattern that I have employed on several occasions for several different clients. I will be discussing design choices I made that worked well, and a few that did not work so well. Hopefully, when you encounter a situation where this design pattern may be appropriate, you will not make the mistakes I made, or, if you do, you will make them more quickly.

As a shorthand name, I refer to the design pattern solution as a workflow engine, which solves a workflow problem.

When is a Workflow Engine Appropriate?

I think of a workflow engine as a tool that is part of something bigger. The immediate problem solved by a workflow engine is to take a collection of data, analyze it using a set of instructions, and return a set of results, without any need for outside interaction during the running of the workflow engine. The workflow engine starts does its work and ends without having to be managed by a user or outside the program.

The key part of the above description is the “set of instructions” that control what the engine does. By allowing for different sets of instructions, the same workflow engine can handle multiple problems and be used differently in an unknown future, without having to be re-written. If the set of instructions exists in a language that is understandable by the end-user, then the end-user can potentially modify or enhance the workflow now or in the future. Alternatively, an “upgrade” to the system that includes the workflow engine can be performed with minimal effort by just providing a new set of instructions that can be passed to the workflow engine.

Workflow Engine Examples

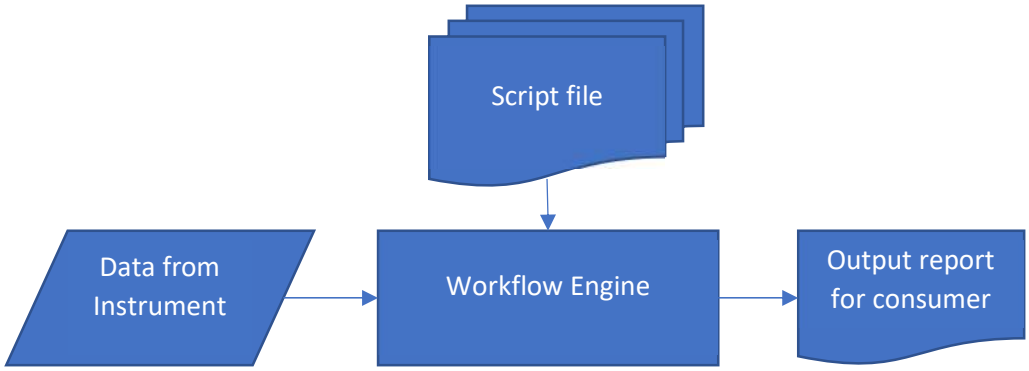
In my own experience, one of my clients had a medical instrument that measured characteristics of a patient sample to determine if the patient had disease X. By putting different reagents on the instrument, different diseases (U, V, Y, and Z) could be detected. The raw data measured by the instrument required a lot of post-processing in software before patient results were delivered. However, the post-processing for disease Y detection was similar but slightly different from, disease X post-processing. Also, it was known that in the next few years, the client would be adding many new tests with each software release, building the instrument’s testing menu. The client could not predict exactly what these future tests would require for post processing but be reasonably confident the post processing steps would be close to the steps currently in use. A workflow engine, where each test’s post-processing was implemented using a different script, was an ideal solution.

In a different situation, a client had a process that generated many different files (different file formats, different data content) at various points during the overall process. From each of these files, a small

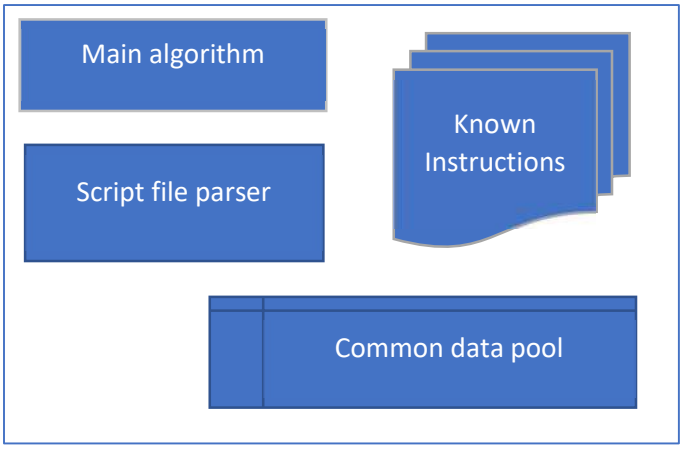
amount of data had to be extracted to form an overall final report for the process. We were near the end of the initial development cycle, and various other stakeholders were finally getting to see the final report and they all were going to want changes to what it contained. Since no schedule delay to accommodate these changes was allowed, we used a workflow engine where the script contained the instructions for extracting data from the files and instructions on how to store it in the final report. This approach was perfect for allowing rapid, flexible responses to stakeholder requests.

Overview of Workflow Engine

Pictorially, a design solution that we at RND Group have used for multiple clients looks something like this:



The workflow engine itself has these significant pieces:



Main Algorithm

The main algorithm of the workflow engine is conceptually simple in what it does. In pseudocode:

```
PopulateKnownInstructions // Known Instructions now populated
while (instructions remain in script)
{
        GetNextInstructionThatCanRun(Known Instructions)
        ProcessInstruction(Common data pool)
}
```

That's it! Of course, some assumptions are going on here, such as:

- There is an instruction very early in the script that "acquires" the raw data that is being processed and moves it into the common data pool so that the remaining instructions have something to act on.
- There is an instruction very late in the script that "generates" the desired output of the workflow engine from the contents of the common data pool.

Script File Parser

The script file parser is self-explanatory. Its role is to take in a script file and parse out the individual instructions, along with any arguments associated with that instruction. How you build this piece is dependent on the format of the script file, which will be discussed later.

Known Instructions

This is a collection containing each instruction that the workflow engine can handle in any script file that it processes. The actual type of the collection depends to some extent on the design choices you make for future growth, i.e. how do you want to add new instructions in the future. It could be as simple as a dictionary of classes that all inherit from a base Instruction class. The "name" part of the dictionary could be the instruction name, and the "value" part of the dictionary could be a class that contains the code that should execute when the instruction appears in the script.

Another option for populating the collection would be to use the Managed Extensibility Framework (MEF) that is a part of the .NET Framework from Microsoft (https://docs.microsoft.com/en-us/dotnet/framework/mef/). Essentially, this allows for new services (think "instructions") to be discovered at run-time, rather than at compile time. Thus, future versions of the workflow engine could support new instructions without having to change a line of code in the core workflow engine or any existing instructions, and, most importantly, without having to re-test any existing code. This can result in substantial savings in time and effort when doing future releases or upgrades. Other frameworks and operating systems have somewhat similar ways of discovering services at run-time.

Common Data Pool

This part of the workflow engine is the heart of the design. The general idea is that each instruction is responsible for retrieving the data it needs to do something meaningful, such as performing a calculation and then puts the result back into the common data pool. Instructions should make as few assumptions as possible about what has happened before in the script, and likewise, make as few assumptions as possible about what will happen later in the script. In other words, the contents of the common data pool at any point in time should NOT be dependent on exactly what instructions have been executed so far in the script. That is up to the scriptwriter to manage.

For example, assume instruction A uses pieces of data X and Y and creates from that a piece of data Z. Likewise, instruction B uses X and Y to create Z, but perhaps with a different algorithm. Now, we have a situation where the end-user can use either instruction A or B to do their overall processing, just by changing the script. Conceivably, nothing prior to the instruction needs to change in the script, nor anything after the instruction needs to change in the script. Script writers (end users) will appreciate that kind of flexibility.

Design Options

In this section, I review some possible workflow engine design options. I will review the pros and cons that I have experienced with some of these choices as well.

Common Data Pool Choices

The common data pool is a key component of a workflow engine, so some good thought needs to go into its design. The biggest problem to keep in mind is figuring out how to deal with result type incompatibilities between instructions. For all types of workflow engines, there is a subtle form of "linkage" between different instructions that the end-user (the scriptwriter) must be aware of. In particular, if instruction A generates and stores something called "IntermediateResult", and then later in the script, instruction B wants to use "IntermediateResult" to do more work, then the type of object that instruction A stores needs to be compatible (if not identical) to what instruction B wants to use. While there are ways to work around this limitation, they do increase the amount of contextual understanding required on the part of the scriptwriter, so be careful. Hopefully, the design of the common data pool will help to reduce this problem as much as practical.

If your workflow engine fits nicely within the memory and processor constraints of the system as a whole, a simple choice for the common data pool is a *Dictionary<string, object>()* collection (using .Net notation). The pros of this type of common data pool are that it supports different types of raw data for different types of instructions, thus giving your instructions lots of power to do a variety of tasks. Access speed to the data is as close to instantaneous as you can get for both retrieving and storing data. One major con to this type of data pool is that whenever an instruction retrieves something from the common data pool, it may have to "convert" or cast the retrieved object into a type that the instruction can work with better. Another con of this type of structure is that you (the instruction writer) or you (the scriptwriter) must manage namespace collision issues for the intermediate results that are created. For example, if the scriptwriter uses instruction #3 to create an intermediate result called "Rslt_a", and then uses instruction #7 to create another intermediate result called "Rslt_a", did the scriptwriter intend to overwrite what was done in instruction #3, or was it a mistake?

If the size of the intermediate results is somewhat large, then a database can be used to manage the common data pool. Depending upon the environment where the workflow engine will live, there may already be a database that can be used. In general, a "key-value" database or store would likely be the best choice, if no other considerations are relevant.

Another choice, if the intermediate results are still larger, is to use the file system itself as a common data pool. This is especially useful if the instructions already exist as command-line executables. In this scenario, each intermediate result is a file stored somewhere meaningful in the file system. Each instruction gets its data from one or more files and creates its result as another set of files.

Script File Format Choices

The next major choice that needs to be thought out is "what does an instruction script look like?"

In my workflow engine development experience, I have made script files in the form of XML files, Json files, and straight text files. There are both advantages and disadvantages to each format. Essentially, it becomes a trade-off between what the script writers are comfortable with, and how much effort you want to put into the coding of the script file parser. If you use a formatted structure like XML or Json, then you can potentially do less development work on the script file parser, because the instructions are already in a known format, and you can provide a schema to ensure that the script file is valid before you even start parsing it. However, using a more structured format like XML or Json puts a larger burden on the scriptwriter(s), because they not only have to know the purpose of what they are trying to accomplish with the script, but they also have to know the language of the script (e.g. XML or Json) and what is legal in that language.

In general, the script file will want to support the following:

- An instruction name
- A set of parameters that give more details to the instruction on what data to act on, or what to store
- An order that the instruction should be executed. Note that some script file formats, this is obvious to the scriptwriter, others, not so much. A text file will generally be assumed to execute from the first line to the last line, whereas an XML or Json file where each instruction is in a different node does not necessarily have a specific absolute order to nodes.
- A commenting mechanism so that the scriptwriter can document what they did, why, etc. Always think of the future maintainers!

I will say that based on my experiences so far, even software developers can be frustrated when trying to write a script file in Xml or Json, so I would recommend a text file format without hesitation. This is especially true if you build in an obvious commenting mechanism into the script file, and then provide the scriptwriters with lots of well-commented example script files they can copy/paste from using their favorite text editor.

Instruction Design – Best Practices

Common Needs of all Instructions

If each instruction available to the scriptwriter is implemented as a piece of code that the workflow engine main process can call, then there are several things you can do as a workflow engine designer to make your life easier. Here is a list of ideas that have worked out well for me.

1. Isolate the code for each instruction as much as possible from all other instructions. Assuming you are using an object-oriented language, this means each instruction is in a separate class at the very least. Consider putting each instruction in a different assembly also. This gives you a lot of flexibility in the future to upgrade an instruction without breaking other parts of your system.
2. Each instruction should derive from a common base class, if possible. The base class will contain the code to retrieve previous intermediate results and store new intermediate results. This gives you a lot of flexibility during initial development to fine-tune the common data pool design as well, without having to re-do each instruction's code.

3. The instruction base class should also provide the logging services for instructions (you will want to include logging the workflow execution process).
4. Most instructions should have a list of parameters that appear in the script file. Usually, these parameters are mundane, e.g. the name of the intermediate value to use as input #1, the name of the intermediate value to create, etc. Some instructions will have more complex input parameters. This can also influence your choice of script file format.
5. Leave a place for the scriptwriter to put comments in the script file. They are just as forgetful about the details of the instruction as anyone after time passes.

Dealing with Errors During Instruction Execution

One big challenge that you will face as a workflow engine designer is dealing with errors that occur during the execution of a script. In particular, the raw data that is given to the workflow engine system is probably not guaranteed to always produce correct results. Often, at various points in the execution of a script, the intermediate results are checked to see if they are still "in-bounds" or "valid". If the intermediate result is out of bounds, the final output is known to be invalid, or not reportable, or some equivalent. Continuing with the normal script execution is pointless, or even counterproductive. How do you deal with this in your design of instructions?

One paradigm that I have found useful in designing workflow engines is to make it known to everyone upfront (instruction coders, scriptwriters, etc.) that the entire script must ALWAYS execute to completion. This has several benefits and expectations, as follows:

> **Benefit**: Even if something bad happens early during the script execution, there exists the possibility that a later instruction can either correct the error, or (more likely) generate an output that gives details about what went wrong during the process. It is always better from a customer's point of view to fail with a report about what went wrong than to fail without generating a description of what went wrong.
>
> **Expectation**: Every instruction must deal with missing or invalid data in its input - any input. I cannot stress this enough; any input could be potentially corrupt. Paranoia is your friend when you are writing code to implement instructions. Make NO assumptions about what the scriptwriter has already done before calling this instruction. Usually, this implies a lot of error checking of the intermediate values that are retrieved to determine if they can be used by the instruction's algorithm.
>
> **Expectation**: Every instruction must not crash in such a way as to bring down the entire workflow. If instructions are implemented in code, this usually means each instruction is fully wrapped in a try/catch block to deal with any unforeseen errors. The catch block can a) log the exception and b) set the overall process validity to false, (see below).

Useful tip: I usually designate a special intermediate Boolean value just for the overall validity of the script process. Most instructions will, as their first action, check the status of this intermediate. Typical instructions, if they see this value as false, can exit immediately. Generally, this should be done without logging anything at all, as this will keep the log files meaningful when debugging what went wrong with a script. Immediately after this check, add a log statement indicating that Instruction X (with parameters y, z) is beginning. This will help everyone later when debugging script file (or workflow engine) issues.

Not all instructions will follow this paradigm, as some instructions are specifically designed to run when overall validity is false, such as the final report, or a final cleanup step.

Support Tools

One client of RND Group requested a specific support tool to help with the creation of script files. Some of the features of this tool were:

- The user could select an instruction to add to a specific location (execution order). The tool would tell the user what parameters the instruction needed, types, description of what the instruction did, etc.
- The tool could analyze a script to ensure that when each instruction was executed in the specified order, that there was a prior instruction that created the intermediate result the instruction expected. This prevented an obviously flawed script from getting past the script developer's realm and into more general use.
- If instructions could be executed "in-parallel", i.e. two or more instructions could execute in either order (or at the same time) without affecting the outcome of the script, this was shown visually to the user.

Another RND Group client needed a workflow engine that required lots of parallelism and large, file-based intermediate results. Each instruction in the script was a separate program (command-line interface-driven). In this case, there is an off-the-shelf product called Cromwell that executes script files written in the WDL language. Some useful links about Cromwell and WDL are below:

- https://github.com/openwdl/wdl/blob/master/versions/draft-2/SPEC.md
- https://cromwell.readthedocs.io/en/stable/

Note that using WDL and Cromwell is not for the faint of heart, and the script development process in WDL requires a lot of comfort with programming techniques.

Conclusion

A workflow engine solves a common problem of getting a series of processing steps (instructions) organized into a format that a technologist and even an end-user can modify using their own domain knowledge. Future problems can be solved with a workflow engine without having to write a significant amount of new code, (and possibly none), avoiding a lot of the re-testing effort that occurs whenever new functionality is introduced into a system.

Hopefully, this paper has given you some ideas on what to do, and possibly a few things to avoid, if you are considering or currently using a workflow engine. RND Group has experience in implementing workflow solutions in a variety of different problem domains.

Please reach out to us if you would like assistance in addressing your specific workflow engine or other regulated software needs. We have been helping clients deliver complex, high-quality software for medical devices for over 20 years and would appreciate an opportunity to speak with you about your specific needs.