**Workflow Foundation Internals (I)**

**Andrew Au**

Inspired by the book [Essential Windows Workflow Foundation](http://www.amazon.com/Essential-Windows-Workflow-Foundation-Dharma/dp/0321399838) that describes the last version of Workflow Foundation, I can’t stop myself trying to write an equivalent piece for WF4. While the basic working principle is fundamentally the same, the programming model is quite different. We will start from the same principle of using serialization of delegate, and we will develop our ‘home-made’ workflow runtime, and we will see how WF4 made its design decisions.

First of all, let’s us review the underlying CLR technology for continuation. Continuation is a point that can be saved for resuming execution, and therefore it needs to contain pointer to executable code. Delegate is used for this purpose. The great thing about delegate can be round-tripped to a binary store and back remain executable. Here is a code sample to show the roundtrip of that delegate works.

*Code Sample 1: SerializableDelegate*

Serializable delegate provided us with a mechanism to suspend a running managed thread, and resume in another process (perhaps on another machine). Doing so has a lot of advantages. The most important one is that we remove the ‘affinities’. The code is no longer stuck to original process or even the original machine. This allows us to scale the application by simply adding more machines. Moreover, we now have control. For example, we could delete the serialized delegate instead of resuming it. By doing so, we essentially canceled the execution. Similarly, we could suspend the execution for days without worrying main memory are being used. Let us take a look at this code sample to see how serializable delegate allow us to break a program into several processes.

*Code Sample: Version 1*

Run the program three times, you will see “Hello world to homemade workflow foundation!” is displayed on the console. With the code above, we have just engineered our first most primitive workflow application using the code above. Needless to say, this is rough and there are a lot of things we can improve upon. We will take our first step to separate the concern of scalability (i.e. the fact that we are serializing delegates from the business logic). For that, we create a new class named ReadReadWrite, move the business logic method there, and that leads us to the version 2.



*Code Sample: Version 2*

First, I put a while loop in the main program to avoid run the program three times. This is really just for convenience. It is good enough for us to know we CAN break them into different processes, but we don’t have to. The refactoring is mostly straightforward, but there is one thing that worth notice here. I made NextDelegate a property of ReadReadWrite. This is to allow all delegates to share a uniform signature.

The hosting program and the business logic are separated now. At this point, we have the still got two coupling between the host and the business logic. The host need to know the starting point is RunStep1, and the host need to know NextDelegate is the property storing the continuation. These coupling make the host not a generic one. We could remove these coupling by having a base class for ReadReadWrite. Let’s call it Activity.



*Code Sample: Version 3*

The refactoring is straightforward. Looking at ReadReadWrite, the code is now pretty easy to write. One thing that we don’t like is that the Execute method and the RunStep2 method are essentially the same piece of logic to read a file, it is best to share the logic into reusable components. To tackle this problem, we realize there are really two hurdles. One is that they update different states, and that they return different delegates. The fact that they are returning different delegate is a problem, because these delegates are really control logic, and is orthogonal to reading a file. We made them together and therefore the logic not cohesive. Now we further optimize the structure by separating them.



*Code Sample: Version 4*

It is non-trivial to write the Sequence activity, and it is far from optimal for now. We will postpone the discussion of optimizing Sequence to the next post. For now I want to focus on removing the duplication of Read1 and Read2. With the code above, Read1 and Read2 are really just reading file now. The last step to merge these two classes together is to make them access the state by name instead of a static field reference.



*Code Sample: Version 5*

Now we have reached a state where we can write reusable activity. These activities don’t have to concern themselves with the serialization. Without reading the code of Main, one does not even know serialization is happening. Looking at Read or Write, does it look like our activities API? In the next post of this series, we will continue to work on this example and show why there is a parameter to the Execute method, and how that separates program from data.