APD Communications

C# class to execute a number of pieces of work

Ieaun Roberts

Contents

Goals:

1. The actions must be executed sequentially – one at a time
    a. The actions are not necessarily all executed on the same thread
2. The actions must be executed in the order that they were added to the class
    a. The actions are not necessarily added all at the same time

How to use the system

Included along with the requested class (or in this case classes [Action_Executer.cs and Threaded_Action.cs]) is a win form testing system. There are 8 methods available to play around with:

Method	Parameters	Description
Simple_ConsoleWrite 1-4	None	Prints "I have written :" + 1 | 2 | 3 | 4
Count_to_1000	None	Prints 1 - 1000
Simple_ConsoleWrite	String	Prints whatever you supply as an argument in Tb 1
Increment	Int	Prints the number supplied and number supplied +1
Ass_Two_Ints	Int,Int	Prints the sum of the two integers supplied in Tb1 and Tb2

Steps:

1. Select the method you would like to add to the stack to be executed.
    a. Fill in any parameters if required.
2. Enable either of the radio buttons:
    a. "On same thread", if you would like to continue this instruction on the current thread.
    b. "Start new thread", if you would like to create a new thread for the following action to execute on.
3. Click "Add to stack" to add the action to the stack (adds it to the instance of the class).
4. Once you have added all your Actions, either check or uncheck "Force thread await"
    a. Unchecked: threads will not wait for each other to finish.
    b. Checked: threads will wait for each other to finish.

E.g "On same thread" turned on at 4, "Force thread await" checked

Thread 1	Thread 2
1	
2	
3	
	4
	5

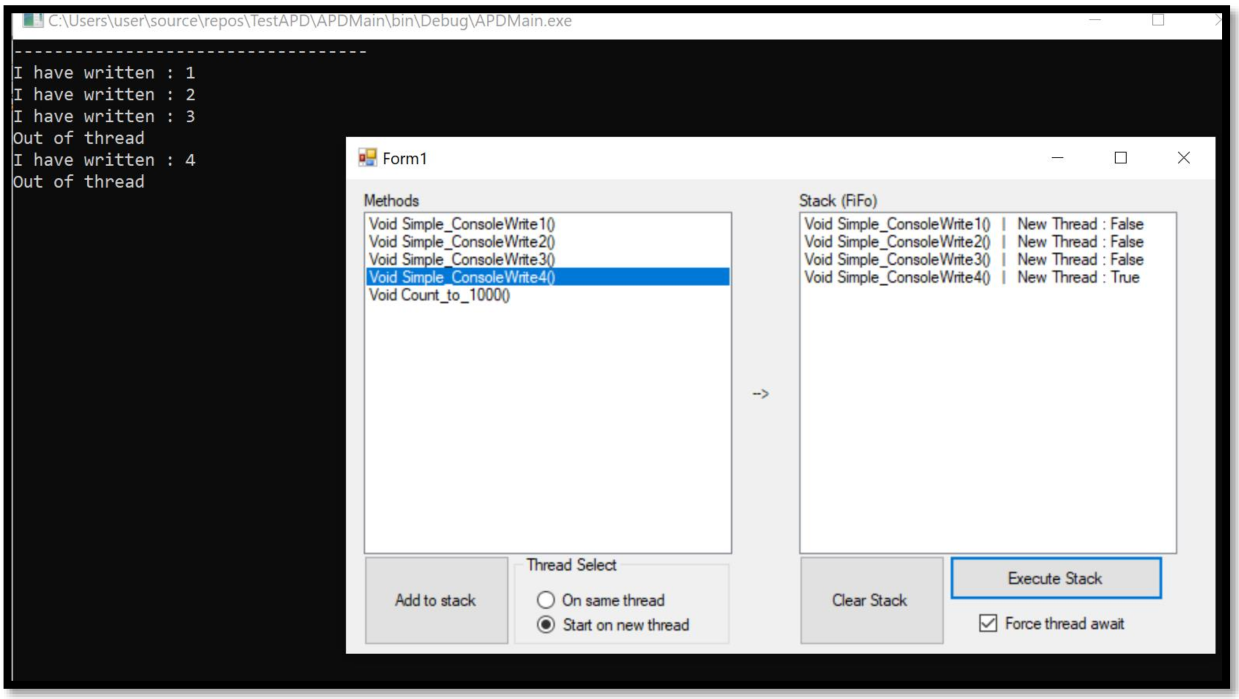
Result: 1,2,3,4,5

E.g "On same thread" turned on at 4, "Force thread await" unchecked

Thread 1	Thread 2
1	▨
▨	4
2	▨
▨	5
3	

Result: 1,4,2,5,3

Initial version



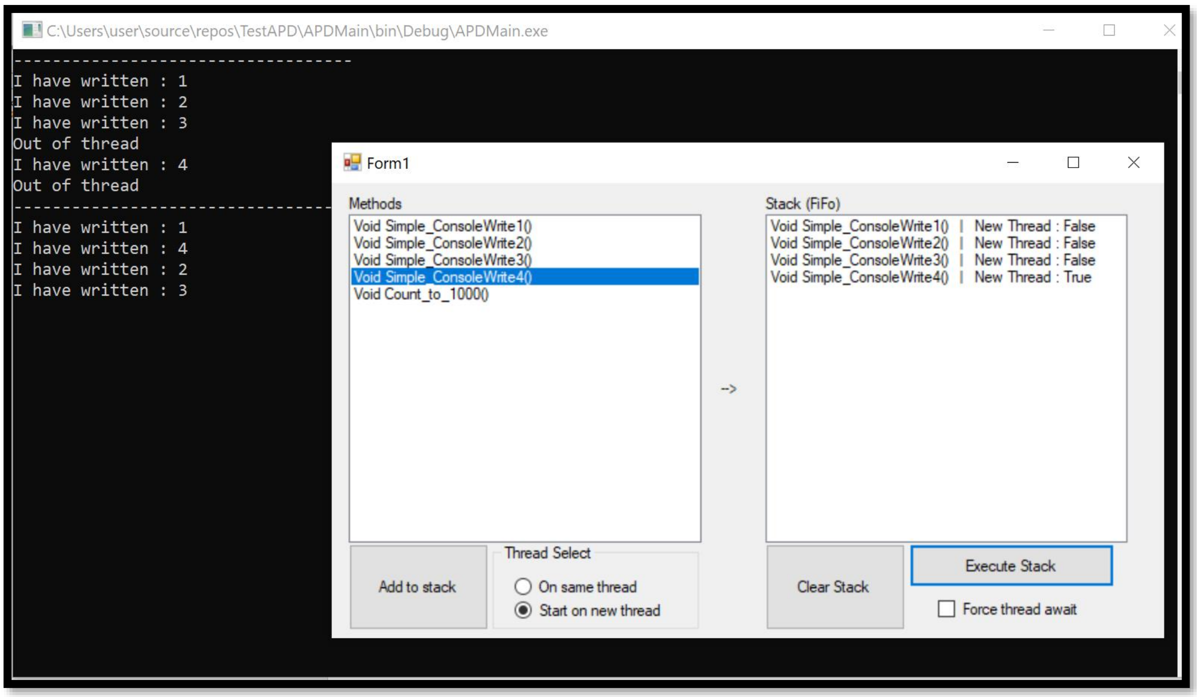
```
C:\Users\user\source\repos\TestAPD\APDMain\bin\Debug\APDMain.exe
----------------------------------
I have written : 1
I have written : 2
I have written : 3
Out of thread
I have written : 4
Out of thread
```

```
Void Simple_ConsoleWrite1()  |  New Thread : False
Void Simple_ConsoleWrite2()  |  New Thread : False
Void Simple_ConsoleWrite3()  |  New Thread : False
Void Simple_ConsoleWrite4()  |  New Thread : True
```

Thread Select: ( ) On same thread [✓ Start on new thread]

[✓ Force thread await]

In the above image the instructions are called to write a respective number, they are executed sequentially in the order added to the class satisfying **Goal 1a, 1b, 2a.** Additionally, included is the "Force thread await" checkbox to view how the operation would occur if the threads were allowed to operate without the "wait your turn" directive (Shown below).



```
C:\Users\user\source\repos\TestAPD\APDMain\bin\Debug\APDMain.exe
----------------------------------
I have written : 1
I have written : 2
I have written : 3
Out of thread
I have written : 4
Out of thread
----------------------------------
I have written : 1
I have written : 4
I have written : 2
I have written : 3
```

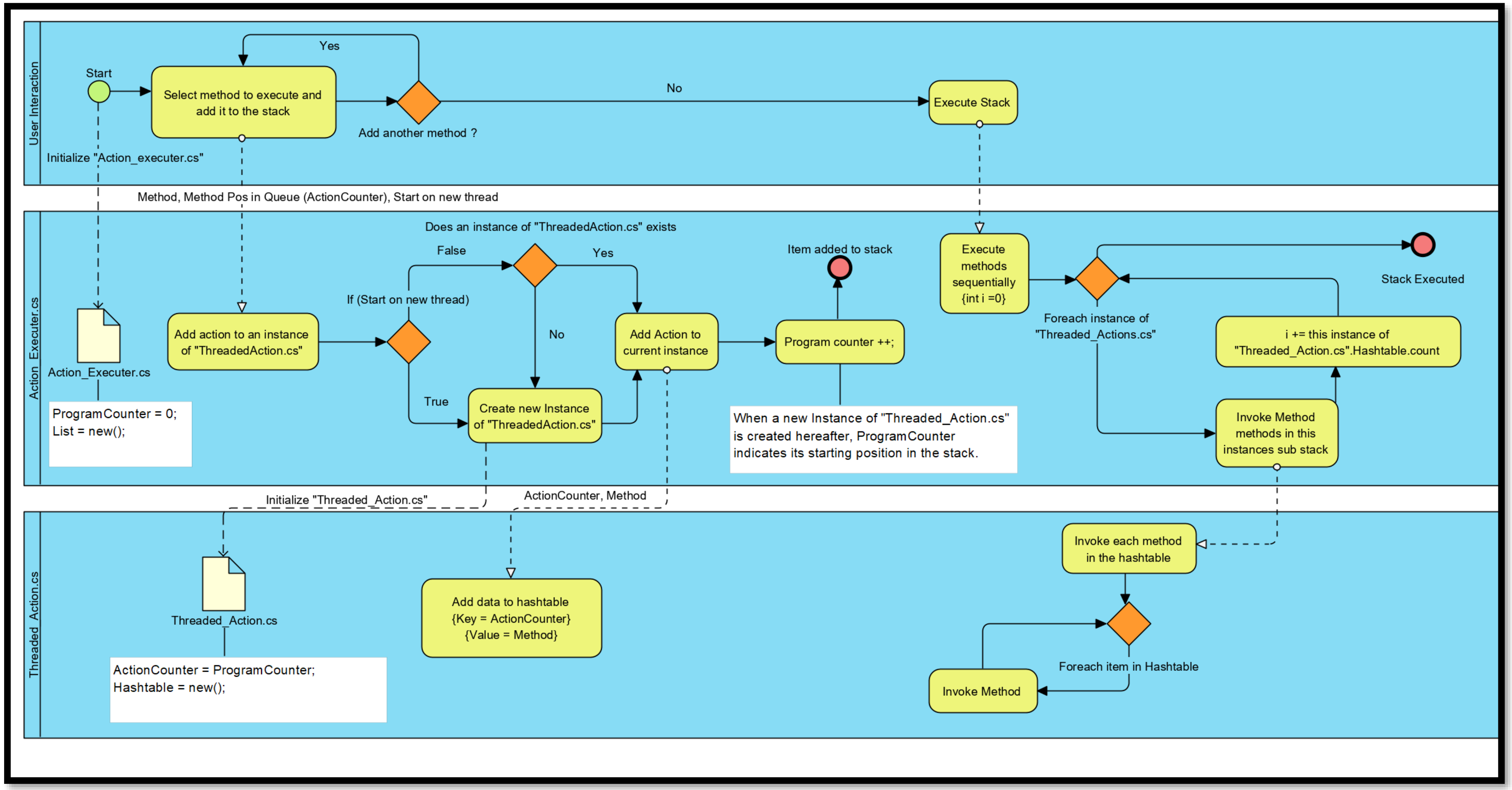
```
Void Simple_ConsoleWrite1()  |  New Thread : False
Void Simple_ConsoleWrite2()  |  New Thread : False
Void Simple_ConsoleWrite3()  |  New Thread : False
Void Simple_ConsoleWrite4()  |  New Thread : True
```

Thread Select: ( ) On same thread [✓ Start on new thread]

☐ Force thread await

Users are also able to add additional Actions at any time before executing the stack satisfying **Goal 2b**

Initial design of the system

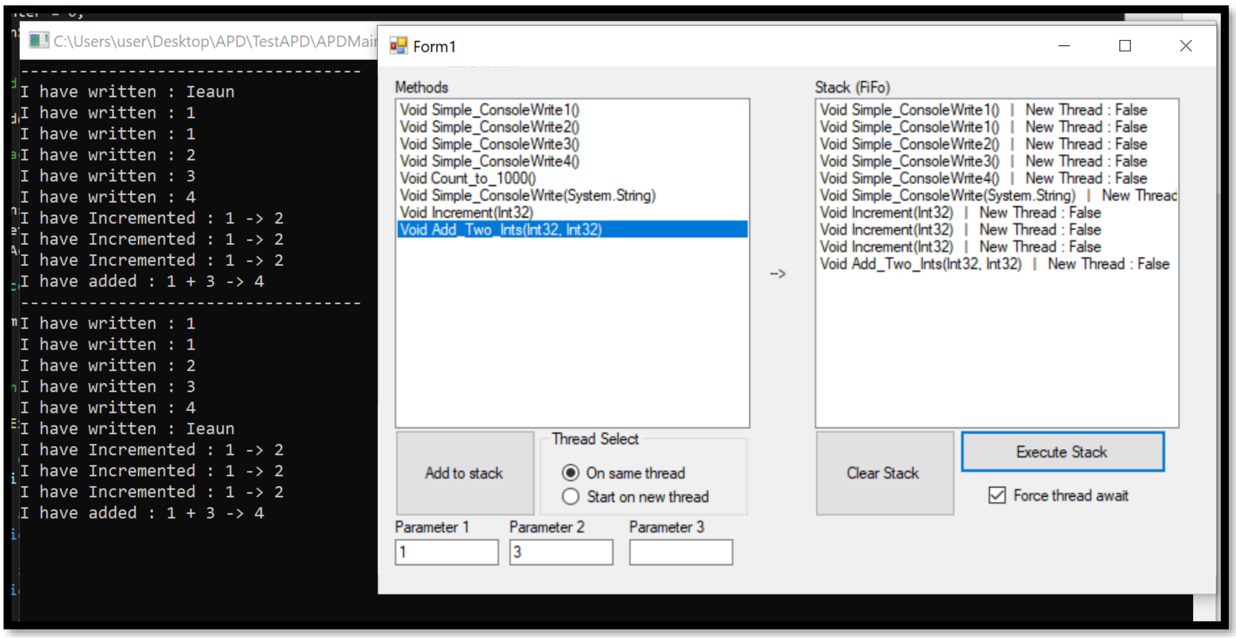

User Interaction
Start
Select method to execute and add it to the stack
Yes
Add another method ?
No
Execute Stack
Initialize "Action_executer.cs"
Method, Method Pos in Queue (ActionCounter), Start on new thread
Action_Executer.cs
Action_Executer.cs
ProgramCounter = 0;
List = new();
Add action to an instance of "ThreadedAction.cs"
If (Start on new thread)
Does an instance of "ThreadedAction.cs" exists
False
Yes
No
True
Create new Instance of "ThreadedAction.cs"
Add Action to current instance
Item added to stack
Program counter ++;
When a new Instance of "Threaded_Action.cs" is created hereafter, ProgramCounter indicates its starting position in the stack.
Execute methods sequentially {int i =0}
Foreach instance of "Threaded_Actions.cs"
Stack Executed
i += this instance of "Threaded_Action.cs".Hashtable.count
Invoke Method methods in this instances sub stack
Initialize "Threaded_Action.cs"
ActionCounter, Method
Threaded_Action.cs
Threaded_Action.cs
ActionCounter = ProgramCounter;
Hashtable = new();
Add data to hashtable {Key = ActionCounter} {Value = Method}
Invoke each method in the hashtable
Foreach item in Hashtable
Invoke Method

It is worth noting that the designs on the previous page were made in the planning stages and I was unable to update them as my trail membership to Visual Paradigms expired. The system was implemented similarly however there are some deviations.

Each action is added to the "Action_executer.cs" class which in turn adds it to an instance of "Threaded_Action.cs". If you request a new thread to be used, then a new instance of "Threaded_Action.cs" is created and the actions are added to that instance thereafter. Once the user executes the stack, the instances of "Threaded_Action.cs" are executed sequentially.

Threaded_Action instance 1	Threaded_Action instance 2	Threaded_Action instance 3
1	▨	▨
1	▨	▨
▨	2	▨
▨	2	▨
▨	▨	3

Final version:



```
----------------------------------
I have written : Ieaun
I have written : 1
I have written : 1
I have written : 2
I have written : 3
I have written : 4
I have Incremented : 1 -> 2
I have Incremented : 1 -> 2
I have Incremented : 1 -> 2
I have added : 1 + 3 -> 4
----------------------------------
I have written : 1
I have written : 1
I have written : 2
I have written : 3
I have written : 4
I have written : Ieaun
I have Incremented : 1 -> 2
I have Incremented : 1 -> 2
I have Incremented : 1 -> 2
I have added : 1 + 3 -> 4
```

Methods:
```
Void Simple_ConsoleWrite1()
Void Simple_ConsoleWrite2()
Void Simple_ConsoleWrite3()
Void Simple_ConsoleWrite4()
Void Count_to_1000()
Void Simple_ConsoleWrite(System.String)
Void Increment(Int32)
Void Add_Two_Ints(Int32, Int32)
```

Stack (FiFo):
```
Void Simple_ConsoleWrite1()  |  New Thread : False
Void Simple_ConsoleWrite1()  |  New Thread : False
Void Simple_ConsoleWrite2()  |  New Thread : False
Void Simple_ConsoleWrite3()  |  New Thread : False
Void Simple_ConsoleWrite4()  |  New Thread : False
Void Simple_ConsoleWrite(System.String)  |  New Thread
Void Increment(Int32)  |  New Thread : False
Void Increment(Int32)  |  New Thread : False
Void Increment(Int32)  |  New Thread : False
Void Add_Two_Ints(Int32, Int32)  |  New Thread : False
```

Thread Select: [✓ On same thread] Start on new thread

Force thread await ✓

Parameter 1	Parameter 2	Parameter 3
1	3	

The final version of the system includes the ability to add parametrized actions, the UI is limited to a maximum of 3 parameters, the class is limited to 10.

In the above figure, a new thread is started at instruction 6, "I have written: Ieaun".

The first block on the console was printed with "Force thread await" unchecked, the second block was printed with "Force thread await" checked.

Reflection:

Before starting this program, I had no prior knowledge of the “Action” type however I was familiar with delegates and event handlers. My first implementation of this system made heavy use of reflection which was later altered to align with the emphasis on the “Action” type.

A relevant project I have completed that is slightly similar to this would be my coursework for the MSc module “Component based Architecture” on my portfolio site (leaun.github.io) where I built a very simple virtual machine. I learnt a lot about threading and program execution in the .net environment completing that piece of coursework.

I knew how to do this using a pure-blooded reflection method by just finding the references in assemblies and invoking instances. This is how I initially built up the system. The actual use of actions is where I fell a little short on the first 2 days of developing this system (in terms of parameter inclusion).

I was familiar with the strongly typed nature of .net and dynamic invocation of methods however I was not familiar with Actions and generic types in the form: invokeActionMethod<T> (Action<T> action).

```
public void invokeActionMethod<T> (Action<T> action, object [] parameters)
```

Completing this test was a valuable learning experience for me and I feel I am walking away from this having gained a bit more knowledge of the .net environment and a greater understanding of delegates.

I made extensive use of the Microsoft documentation on Actions and Generics, and also many blog sites such as stack overflow, code project and geeks for geeks.

References

dasblinkenlight, 2011. *codeproject.com/Questions/164911/Pass-a-method-with-N-params-as-parameter-in-C.* [Online]
Available at: https://www.codeproject.com/Questions/164911/Pass-a-method-with-N-params-as-parameter-in-C
[Accessed 2020].

Microsoft, 2015. *docs.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/.* [Online]
Available at: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/
[Accessed 26 04 2020].

Microsoft, 2020. *docs.microsoft.com.* [Online]
Available at: https://docs.microsoft.com/en-us/dotnet/api/system.action-1?view=netcore-3.1
[Accessed 2020].