tl;dr

I made an extended state machine.
It has a current state and a dictionary of values (key: string, value:any).
States have key-value-pairs that map an event to a transition.
Transitions have target states, a condition (lambda) and an action (lambda). If the condition is true, the state machines current state is changed to the target state and the action is executed.

To run the project, open the StateMachineDSL.sln solution in Visual Studio, select either the CoffeeMaker or the CookingHood projects and run either the CoffeeMakerProgram or the CookingHoodProgram.

I have also included .exe files for the two programs. You can also just run those.

Then a command line window opens and prints the current state. You can then enter commands for the coffee machine :"ON", "OFF", "PAID", "WATER", "COFFEE", "COCOA", "DONE"

or for the cooking hood: "PLUS", "MINUS"

State Machine DSL

For this assignment I have created an internal DSL for creating State Machines. The primary goal of the DSL has been to offer as much IDE support as possible. It is written in C# which is (mostly) a statically typed language.

Programs

In order to prove the abilities of the language, I have also implemented two programs using the language. The first program models a Coffee Machine like those deloyed at TEK at SDU. The other program is a Cooking hood with 6 power levels. Both programs have been implemented both with the traditional "Command-Query" API and with the Fluent Interface DSL.

The Coffee machine should operate as follows:
It should begin in the "Off" state.
From there it can be turned on.
When it is on it will provide coffee, water and cocoa depending on user selection.
Before selecting coffee or cocoa, the user must have paid.
Before selecting any drink, the user must have placed a cup in the machine.
The machine can be turned off.

The cooking hood should operate as the one used as an example in the course.

Semantic Model

The semantic model used in this project is an *Extended State Machine*. The State Machine has a current *State* and some *variables* that can have values of any type.

The State Machine processes *Events*. When an event is processed, the current State might change, depending on the *Transitions* that the current State holds.

Each State holds Transitions to other States, which hold transitions to more States and so on.

Each Transition has a target State, a *Condition* and an *Action*. The Condition is a boolean expression that must hold in order to perform the Transition. This is most often a check that some variable has some value. The Action is an effect the transition has. This is most often a change to the value of some variable.

Language Design

This section first describes the Internal Domain Specific Language created in this project, and then describes some of the considerations and decisions made in the project.

The Language

As the primary goal of this project was strong IDE support, I have used a Class Symbol Table (as decribed by Martin Fowler in chapter 44 of Domain Specific Languages). This can be seen in the following code snippet.

```
class CoffeeMachineBuilder : StateMachineBuilder
{
    /// Class symbol table. (abbreviated)
    public States off, waitingForPayment, brewingHotWater [...];
    public Events ON, PAID, WATER, CUP [...];
    public Variables<bool> cupIsPlaced;
```

The class symbol table above enables type-checking in the IDE. Instead of using strings for all the identifiers, I am now able to use instances of specific classes. Therefore the IDE forces me to use the correct type. I am, for example, forced to transition to a States object. The code snippet below shows how the objects in the symbol table are used

```
    public override StateMachine BuildStateMachine()
    {

    var stateMachine =
    InitialState(off)
        .OnEvent(ON)
            .TransitionTo(waitingForPayment)
```

The above snippet shows how the initial state "off" is created and that it should transition to "waitingForPayment" when it recieves the event "ON". The snippet below shows how variables can be accessed and used in a Condition.

```
.State(waitingForPayment)
    .OnEvent(PAID)
        .TransitionTo(waitingForDrinkSelection)
    .OnEvent(WATER) // Water is free. Does not need payment.
        .CheckThat(cupIsPlaced.IsEqualTo(true))
        .TransitionTo(brewingHotWater)
//...
```

The snippet below shows how variables are assigned values. Note that no transition is added to events "CUP" and "NOCUP". These are self transitions that only perform an action.

```
.EveryState() // Add the following transitions to all states
    .OnEvent(OFF)
        .TransitionTo(off)
    .OnEvent(CUP)
        .ModifyVariable(cupIsPlaced).SetValue(true)
    .OnEvent(NOCUP)
        .ModifyVariable(cupIsPlaced).SetValue(false)
.Build();
```

The below snippet shows how both a transition and an action can be added to an event.

```
InitialState(PowerOff)
    .OnEvent(PLUS)
        .ModifyVariable(power).SetValue(MIN_POWER)
            .And().TransitionTo(PowerOn)
```

Design decisions

Method Chaining

For my internal DSL i have used Method Chaining. I have done this for two reasons:

1. It reads fluently, and things happen in the order you expect (as opposed to function nesting)
2. It increases the IDE support. My StateMachineBuilder implements several interfaces, and each method returns only one of these. It can therefore define which methods are legal after each method call.

Types

A major hassle in this project has been the types of variables. I want the DSL to support variables of any type, but i also want strong type safety.

For example would it be bad to have a boolean variable and trying to assign and integer value. I have used Generics to avoid this, but my solution is not perfect. I have made sure that you cannot assign integer values to boolean variables, but my type checking does not prevent method calls like: `ModifyVariable<bool>(cupIsPlaced).Add(true)`, which is bad, because adding booleans does not make sense. However, this would yield an ArgumentException when building the State Machine, so at least it "fails fast".