ATDD Getting Started Guide

This is a very simple, very crude quick start guide to help ASD-200-CMP participants that are new to SpecFlow. This is juuuuust enough to get you started. For real help with syntax and structure, refer to the SpecFlow documentation at https://docs.specflow.org/projects/specflow/

None of this code is perfect or ideal. It likely still needs cleanup or refactoring before you're "done". It is, hopefully, enough to get you started.

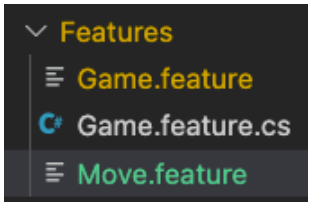
C# with SpecFlow

SpecFlow, like Cucumber, uses two types of files: 1) Feature and 2) Step. The feature file is a simple definition of what scenarios are being tested and what the test data is. The Step file works like glue - taking the inputs from the Feature file and passing them along to your system operations, and calling assertions to check if the tests pass.

We already have a simple Feature file and Step file in our project. We COULD add more scenarios to it. But, let's create a separate set of files for move, since it's a distinct system operation.

Feature File

Navigate to LevelUpGame.Tests/Features/ and create a new file named "Move.feature"



Open the file so we can add some content! Let's start with a description of the Feature. Add the following text. Feel free to customize your description.

```
Feature: Move in a direction

    I want to move my character. If they attempt to
    move past a boundary, the move results in no change in position.
```

Now let's add a scenario outline. This is where we will setup our test data, using what we learned in Specification by Example (SbE). We'll name the Scenario Outline. Then, we'll express the Spec by Example table columns in Given/When/Then syntax. Then, we'll add the examples to the scenario.

Add the Scenario outline. Be sure to indent it under the Feature heading.
IMPORTANT: *Because we want to run our acceptance tests and unit tests as separate goals, we're going to use a tag to tell test-acceptance to run only acceptance tests. You MUST add the @acceptance annotation to your Scenario Outlines, otherwise they will not be run as part of the goal.*

```
Feature: Move in a direction

    I want to move my character. If they attempt to
    move past a boundary, the move results in no change in position.

    @acceptance
    Scenario Outline: Move in a direction
```

Now, turn your SbE table into Given When Then syntax. The preconditions are "Givens". The System Operation is the "When". The Post conditions are "Thens". You might need to simplify your SbE to transform it into tests. For instance, I made the decision below to separate the X and Y coordinates into separate columns rather than express them as (x,y).

Add the details to the Scenario Outline. Notice the indentation below. The <variables> represent the columns of data from your SbE table.

```
Scenario Outline: Move in a direction
    Given the character starts at position with XCoordinates <startingPositionX>
    And starts at YCoordinates <startingPositionY>      Erin Perry, 5 days ago •
    And the player choses to move in <direction>
    When the character moves
    Then the character is now at position with XCoordinates <endingPositionX>
    And YCoordinates <endingPositionY>
```

Now, add your data from the table to the "Examples" section. Use vertical bars as the column separators. Make sure the names in your columns match the <variableNames in your Scenario Outline exactly. Your values might be different, depending on how you're laying out your Game Map.

```
And YCoordinates <endingPositionY>
Examples:
    | startingPositionX | startingPositionY | direction | endingPositionX | endingPositionY |
    | 0 | 0 | NORTH | 0 | 1 |
    | 0 | 0 | SOUTH | 0 | 0 |
```

That's all we need for our Feature file! To reach your Definition of Done, you'll want to go back later and add images and context so this becomes living documentation. (See game.feature for an example), but this is good enough to get started. If you run `make test-acceptance` now, you should see Move.feature.cs auto-generated, and your two rows Skipped (since we haven't created the step file for them yet).

Step File

The feature file is in human readable syntax and lets you express your test cases. The Step File is there to glue those test cases to your code. We'll be coding the step file in the language we're using for our application, in this case C#.

Let's give it a shot!

Create a new class in LevelUpGame.Tests/Steps/MoveSteps.cs

We'll need 4 fields to help us store data and call system operations in the GameController class. Inside the MoveSteps class, add the following:

```
        GameController testObj = new GameController();
        int startX, startY, endX, endY;
        GameController.DIRECTION direction;
        Point currentPosition;
```

Add any using statements you need so the code compiles.

Given

Let's start with our Givens. Step files use annotations to tell the SpecFlow runner which methods should map to Givens, Whens, and Thens.

Add the following code directly after the fields above.

```
[Given(@"the character starts at position with XCoordinates
(.*)")]
public void givenTheCharacterStartsAtX(int startX)
{
    this.startX = startX;
}

[Given(@"starts at YCoordinates (.*)")]
public void givenTheCharacterStartsAtY(int startY)
{
    this.startY = startY;
}
```

That's easy enough. The text in the Given annotation maps directly to the text in the Feature file. We've replaced the {variable} with (.*). In the methods, we'll take in integers for the X and Y coordinates. Inside the methods, we're storing the inputs so we can use them in a moment.

Let's add the last "Given" —> for the movement direction.

```
[Given(@"the player chooses to move in (.*)")]
public void givenPlayerChoosesDirection(String direction)
{
    this.direction = (GameController.DIRECTION)
Enum.Parse(typeof(GameController.DIRECTION) , direction);
}
```

In this method, we're going to parse that String to the ENUM for direction in the GameController.

When

Time to add our When

```
[When(@"the character moves")]
public void whenTheCharacterMoves()
{
    testObj.SetCharacterPosition(new Point(this.startX,
this.startY));
    testObj.Move(this.direction);
    GameController.GameStatus status = testObj.GetStatus();
    this.currentPosition = status.currentPosition;
}
```

The When() is in charge of calling our system operation. It calls a method called "setCharacterPosition". This method isn't a system operation, but it IS a new method we're going to add to allow us to make the GameController testable. With this method, we can now set state, providing a position for the player to start on before a move. Then, we call the move system operation, get the updated game status, and parse out a position from that games status. If you've just added this code, it won't compile. We need to add the setCharacterPosition method and the currentPosition to the gameStatus class. Let's do that now!

Updating GameController so we can compile

Open the GameController at LevelUpGame/levelup/GameController.cs
First, add a new field to the GameStatus stuct. This will allow us to return the player's current position at any point in the game. You'll need to import the Point class from System.Drawing as well.

```
public record struct GameStatus(
    // TODO: Add other status data
    String playerName,      You, ye
    Point currentPosition
);
```

In the constructor for this class, we'll set the currentPosition to an invalid coordinate, so we can be sure our tests only pass if the coordinates are properly set.

```
5 references
public GameController()
{
    status.playerName = DEFAULT_PLAYER_NAME;
    //Set current position to a nonsense place
    status.currentPosition = new Point(-1,-1);
}
```

Now, let's add the setCharacterPosition method:

```
public void SetCharacterPosition(Point coordinates)
{
    //TODO: IMPLEMENT THIS TO SET CHARACTERS CURRENT POSITION -- exists to be testable
}
```

Wait…why is this just an empty method? Why can't we just add the implementation now? Well…because we're doing the OUTER loop of ATDD right now. We're doing just enough to get the test cases setup and compiling. THEN (later in the course), we'll start implementing methods using Unit Test Driven Development. So, create the empty method, which is just enough to get the code to compile, and leave a TODO to implement later. Head back to your MoveSteps class and see if you've resolved the compilation errors.

Then

What good is a test without some assertions? The Thens are where we put the actual tests for our scenarios. Add the two Thens, as shown below.

```
[Then(@"the character is now at position with XCoordinates (.*)")]
public void checkXCoordinates(int endX)
{
    Assert.NotNull(this.currentPosition, "Expected position not null" );
    Assert.AreEqual(endX, this.currentPosition.X);
}

[Then(@"YCoordinates  (.*)")]
public void checkYCoordinates(int endY)
{
    Assert.NotNull(this.currentPosition, "Expected position not null");
    Assert.AreEqual(endY, this.currentPosition.Y);
}
```

Yay! Failing Tests!

Make sure everything is saved, then rerun `make test-acceptance`

You should see test failures. That's good! We want failing tests, but ones that are wired correctly. If things are working, you should see

Failed! - Failed: 2, Passed: 2, Skipped: 0, Total: 4,

Now it's time to finish up by adding all your test cases to this scenario, as well as create new Features and scenarios for any other acceptance tests you need to add. Don't forget to add images, write-ups, etc to make this fantastic documentation too!

A quick tip: If you add or change scenarios, and see "Tests Skipped" in the build output, it normally means there's some sort of syntax error in your Feature or Step file (or that the Step File doesn't exist yet).