Sprint 0

Student	Score	Details
Moore, Matthew	**Functionality: 40/40** Non-Moving Non-Animated Sprite -- 5/5 Non-Moving Animated Sprite -- 5/5 Moving Non-Animated Sprite -- 5/5 Moving Animated Sprite -- 5/5 Legend (Sprite Text) – 3/3 Keyboard Input-- 10/10 Gamepad Input-- 7/7 **Implementation: 54/60** IController – 5/5 Controllers -- 14/15 IAnimated Sprite – 5/5 Sprite Classes – 15/15 Overall – 15/20 **Total: 94/100**	***Programming Activities*** Your assignment is to implement a quite simple interactive program. At run-time, the user should be able to interactively select between display of a motionless and non-animated sprite (ex: Mario standing still), a motionless and animated sprite (ex: Mario running in place), a moving and non-animated sprite (ex: dead Mario floating up and down), and a moving animated sprite (ex: Mario running right and left). The user should also be able to quit the program with a key press. Supported Inputs (keys and buttons) will be enumerated in an on-screen legend. • No unexpected variation in "initial state" (e.g., sprite already displayed despite no key/button having been pressed) • An interesting Interpretation of the requirements. The user while attempting to PRESS (and release) the key/button to display the sprite on the screen – will see the sprite added to the screen WHILE (during) the key is detected as pressed in different (x,y) positions. If I wanted/expected a single sprite, I have to be able to press and release the key/button within 16.667 ms. I wonder if that is intentional or how you coded the controller. I'll see when I dig into the code. • Smooth animation (frame cycling) • Supports 'N' different sprites simultaneously "displayed." As for initial playtesting, all of the keyboard keys function as expected. The same can also be said for the gamepad controller buttons. **Implementation Review** Good use of independent files separating interfaces and classes. Next step would be to make use of folders to organize files into collections of themes. I will always start my review with Game.cs But in your case I'm not sure what I'm seeing

I see contained within the Project of the solution two sets of code for Sprint0. One buried in a folder sprint0v2 and the second as a series of files under the project named sprint0v2. And it's NOT the namespaces that will dictate which files are being referenced – Oh Boy!
The differences are subtle – naming conventions ('_' for example)
The .sln WILL NOT BUILD!!

SECOND Attempt
This is an encouraging sight

```
        public List<Sprite> spriteList;
```

but then there's this

```
        private KeyboardController keyboard;
        private GamepadController gamePad;
```

Sprite – not ISprite – hmm

```
    public class Sprite : ISprite
```

okay – so there's a base class (?) – but does the collection List<> have to be the class or an interface? Well

```
        public List<ISprite> spriteList;
```

does compile and run.

I wonder if IController can be substituted?

```
        private IController keyboard;
        private IController gamePad;
```

YES it does compile and run

```
        protected override void Update(GameTime gameTime)
        {
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back ==
ButtonState.Pressed || Keyboard.GetState().IsKeyDown(Keys.Escape))
                Exit();
```

You should remove this from Update

```
    public interface ISprite
    {
        void Update(GameTime gameTime, GraphicsDeviceManager graphics);
        void Draw(SpriteBatch spriteBatch);
    }
```

Plain and simple interface

Then I see this inside game – look at all of those publics and that STATIC for Game

```
        public static Game1 Instance { get; private set; }

        //Create Keyboard and GamePad states
        public Texture2D movingAnimatedSprite;
        public Texture2D statioinaryAnimatedSprite;
        public Texture2D movingNonAnimatedSprite;
        public Texture2D stationarySpriteTexture;
        public Texture2D controls;
        public Vector2 marioSpeed = new(-1, 0);
        public Vector2 cloudSpeed = new(0, -1);
        public SpriteEffects flip = SpriteEffects.None;
        public List<ISprite> spriteList;
```

something tells me that static is used to gain unfettered access to Game's internals
there are 25 (TWENTY FIVE) usage of that static – YUCK!!

Let's find some of that garbage access to the "global" game

```
public interface IController
{
    void Update();

}
```

Another clean interface

```
    public class KeyboardController : IController
    {
        private Dictionary<Keys, ICommand> keyCommands = new();

        public KeyboardController()
        {
            keyCommands = new Dictionary<Keys,ICommand> {
            { Keys.Q, new Quit()},
            { Keys.W, new StationarySprite() },
            { Keys.E, new AnimatedStationarySprite()},
            { Keys.R, new MovingSprite()},
            { Keys.T, new MovingAnimatedSprite()},
            };
        }

        public void Update()
        {
            var keyboardState = Keyboard.GetState();
            foreach (var key in keyCommands.Keys)
            {
                if (keyboardState.IsKeyDown(key))
                {
                    keyCommands[key].Execute();
                    //add a cool down TODO. Im not smart enough yet I guess. Plus
Im pretty sure this is just wrong.
```

```
                    }
                }

            }
        }
```

That’s not a bad Controller – using a dictionary to map Keys to ICommand – nice touch
Hardcoding the ICommands inside the Constructor – okay but then this controller isn’t reusable
And finally, for any Key in this dictionary that’s down, execute the command
Listen to that again – every 16.667 ms, traverse the dictionary checking every key to see if it’s down…

So every 16.6667 ms, 5 keys are checked – BTW your game is running right now on my PC, doing a CRAZY number of checks waiting on me to return to the game to press something
Versus using KeysPressed from the monogame API which will tell you IF something is pressed, then you check the pressed key as a key in the dictionary THEN you Execute the command

Okay – with going depper I can tell you need to review the Command Pattern – the Pattern would have you passing the RECEIVER in the constructor of the Command. I can tell from looking that because you didn’t PASS a Receiver tot eh Command, internally to Execute in the Command you’re using that ugly static Instance to manipulate Game state

```
        class StationarySprite : ICommand
        {
            private Sprite pipe;

            public void Execute()
            {
                //create a stationary sprite
                pipe = new Sprite(Game1.Instance.stationarySpriteTexture, 1, 1,
Sprite.RandomPositionPipe(), Vector2.Zero, SpriteEffects.None);
                Game1.Instance.spriteList.Add(pipe);
            }
        }
```

And there it is sports fans – a Home Run!
Or a foul ball

NOW consider this – Game as a client of the Controller, INSTANTIATES the Commands passing *this* as an argument to each of the Commands and passes references of the commands to the controller along with a KEY (‘W’ for example) to map to the Controllers dictionary. Inside the Command it uses the reference to the game receiver INSTEAD of that ugly static and it invokes a method of Game to perform the action – Command.Execute is simply receiver.DoSomething() – Your Execute is doing work (functioning almost as a Sprite Factory of sorts).

On the topic of Sprites, you managed to create a single Sprite class, that with the removal of the 4 statics for determining random positions could be reused in future sprints

I am concerned about these publics from Sprite NOT in the interface. Why do they need to be public?

```
        public Texture2D Texture { get; set; }
        public int Rows { get; set; }
        public int Columns { get; set; }
```