# Sprint 0

|  |  |  |
| --- | --- | --- |
| Student | Score | Details |
| Ray, 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 Additional Features Bonus -- 0  **Implementation: 57.8/60** IController – 5/5 Controllers -- 15/15 IAnimated Sprite – 4.8/5 Sprite Classes – 14/15  Overall – 19/20  Bonuses:  4 – ICommand/Command Pattern  Total: 101.8/100 | * No unexpected variation in “initial state” (e.g. sprite already displayed despite no key/button having been pressed) * Initial window color Is NOT Cornflower Blue ☺ * Change in initial Size * No unexpected variation in “behavior” (e.g. toggling of visibility) * Smooth animation (frame cycling) * R/T bounce occurs but not a screen edge (screen edges were NOT a requirement) * Only one sprite is “displayed”; each key/button press replaces any prior display action   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  Inside the Sprint0 requirements it was stated:  *Finally, In the Game (main) class, only use the interfaces in the Update and Draw methods.*  Inside Game1 I see  private UnmovingAnimatedSprite animatedSprite;  private UnmovingUnanimatedSprite unMunASprite;  private MovingUnanimatedSprite movingSprite;  private MovingAnimatedSprite fullSprite;  What I expected to see is  private ISprite animatedSprite;  private ISprite unMunASprite;  private ISprite movingSprite;  private ISprite fullSprite;  Which DOES NOT Compile  Your Command implementation is requiring concrete class types versus ISprite in each constructor and as the receiver – we fix this!  Now however CenterStage isn’t part of the interface but accesses inside every Command .Execute  Here’s a revised ISprite  public interface ISprite  {  void Update(GameTime gameTime);  void Draw(SpriteBatch spriteBatch);  bool Activated { get; set; }  void CenterStage();  }  Now everything finally compiles and your interface and concrete Sprite classes adhere to a polymorphic implementation  Game1 for the most part is clean – especially Update and Draw  As far as the implementation for a simple sprite (non-moving non animated) this isn’t too bad  class UnmovingUnanimatedSprite : ISprite  {  public Texture2D Texture { get; set; }  public Vector2 StartingLocation { get; set; }  public bool Activated { get; set; }  public UnmovingUnanimatedSprite(Texture2D texture, Vector2 start)  {  Texture = texture;  StartingLocation = start;  Activated = false;  }  public void CenterStage()  {  Activated = true;  }  public void Update(GameTime gameTime)  {  }  public void Draw(SpriteBatch spriteBatch)  {  spriteBatch.Draw(Texture, StartingLocation, Color.White);  }  }  I prefer NOT to see the public data mamebers (those should be Proprty get and set and protected/private versus public  What’s next for the 4 concrete Sprites – Now that you’ve written 4 independent Sprite classes, do you see any opportunities for refactoring those 4 classes into 1 class that supports conditional animation and conditional movement? Look at the 4 classes side x side x side and see if any duplicate functionality jumps out at you. 4 independent classes was the requirement to get you to begin to look for refactoring and optimization opportunities. Parameterized “state” and behavior driven by “state” will get you to the ONE Sprite class upon which all other sprites (actor, enemy, obstacle, item, scenery) will derive/inherit from.  You’ve got a reasonably well built Moving Animated variant for Sprite. What could you do for this class to extend its capabilities to conditionally eliminate motion and/or animation of frames?  public interface IController  {  void Update();  }  Legit!  public class KeyboardController : IController  {  private KeyboardState previousKeyboardState;  ICommand command;  Dictionary<Keys, ICommand> commandDict;  public KeyboardController(Dictionary<Keys, ICommand> dict)  {  previousKeyboardState = Keyboard.GetState();  commandDict = dict;  }  public void Update()  {  KeyboardState currentState = Keyboard.GetState();  Keys[] keysPresed = currentState.GetPressedKeys();  foreach (Keys key in keysPresed)  {  if(!previousKeyboardState.IsKeyDown(key) && commandDict.ContainsKey(key))  {  command = commandDict[key];  command.Execute();  }  }  previousKeyboardState = currentState;  }  }  My one point of constructive criticism is the expectation on the client  Here’s what that code looks like  kbDict.Add(Keys.Q, command = new ExitCommand());  kbDict.Add(Keys.W, command = new UnmovingUnanimatedCommand(unMunASprite));  kbDict.Add(Keys.E, command = new UnmovingAnimatedCommand(animatedSprite));  kbDict.Add(Keys.R, command = new MovingUnanimatedCommand(movingSprite));  kbDict.Add(Keys.T, command = new MovingAnimatedCommand(fullSprite));  keyboardController = new KeyboardController(kbDict);  The client has to create a Dictionary and assign keys to Commands and then pass the Dictionary to the Constructor of the Controller. Why does the client have to pass as a collection? Is the client expected to change if the internal storage of commands within the Controller changes? What are you going to do to map new/replacement commands? Right now you have to instantiate a new controller since you pass during construction.  RE: Command Pattern  public interface ICommand  {  void Execute();  }  Yep!  class ExitCommand : ICommand  {  public ExitCommand()  {  }  public void Execute()  {    }  }  Um…. There’s nothing here!  class UnmovingUnanimatedCommand : ICommand  {  ISprite reciever;  public UnmovingUnanimatedCommand(ISprite reciever)  {  this.reciever = reciever;  }  public void Execute()  {  reciever.CenterStage();    }  }  Much Better!  And Because I changed from the Concrete Sprite type to ISprite, Guess What – You only need ONE of the FOUR SpriteTypeCommand classes!! Much Cleaner! |