**Oral Exam Prework**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Name | Jillian Jabbes | Team | BrainStew | TL | 2 | Date | 04/13/2025 | Time | Midnight |

Fill in the underlined areas (and the boxes above), now but don’t write on the remainder of this form.

|  |  |
| --- | --- |
| **Contribution:** Briefly describe what your feature(s) is/are:  The feature I created in BrainStew’s “Techdown” 2D platformer game is the player system, which involved implementing the core playable character mechanics for movement, combat (stats), and physics interactions (player object) with the environment. The movement included the ability for users to send inputs for the character to idle, run, jump, double jump, and be able to utilize further scripts to add powerups or additional combat features like shooting. Lastly applied to the character is sprites, states, and animation tied to the core mechanics.  **------------------------------------------------------------------------------------------------------------------**  Walk me through your Gantt chart. How long did this take? How long did you estimate it would take? What did you learn about your skill as an estimator?  One of BrainStew’s first major meetings was deciding on who is responsible for what features, and we all set on one major feature as well as a secondary one whenever help was needed there. So, I ended up organizing my Gantt chart around that plan where it covers mostly my primary feature and for overlaps I’d interact and work with that team member whenever feature overlaps happened to get things working then help out with the secondary feature if required. So in the middle here it states “Work with Weapon Specialist” as the weapon our player uses should be able to interact effectively so I often found myself talking with Bidhi and playtesting with her. In a way this made sense for me as a TL2 too. My secondary feature in case was helping Andrew with bosses, which at this moment we haven’t found much need to besides bouncing ideas.  Personally, I found that my feature up to this point was able to get me to approximately 53 hours when I estimated 57 hours total for everything. I assume that the final stretch of finishing up the game I will definitely reach that. I think my estimation skills aren’t far off, I probably didn’t count all the hours honestly especially when it came to research and figuring out Unity as well as creating graphics. I even taught myself how to use Aseprite, a popular pixel art and animation software.  Run your game and point out places where your code is called and run. (I will cycle through asking you this question and the next one until you either run out of interesting things to talk about or it is clear that you have made an above average contribution.)  (Likely things I’ll point out from the game’s start to finish)   * Main menu: Logo and menu designed by me * Options button: On this page I programmed the script responsible for BC-MODE and the toggle * The script responsible for changing scenes whenever buttons are pressed (not complex, but reused in a lot of places) * Pressing Play…immediately we see the player object the focus * Inputting common movement keys WASD will move the player around, calls my movement scripts, input controllers, ground check, and jump capability – show in the inspector the customizability of many of these values and how they impact movement. * Whenever the player is hit by an enemy, talk about Player Stats script, the player healthbar updating, PTakeDamage script, and how the enemy is sending damage * The lever (not complex, but something quick I made) – was a plan for the level designer to expand on more but this was an MVP basis   Show the C++/C# code that was run. Walk me through the methods called from the time it enters your section of code.  Code showcase found under Assets > src > TL2 > JJ-Scripts and opened on code editor… | /10 |
| **Technical:**  Walk me through your test plan. Give an example where a test case later found a bug in your code by things a teammate added later. (Or explain why you chose a test case specifically because you wanted to ensure that a teammate would know if they broke your code.)  My test plan for the player mainly focuses on verifying that the individual systems work correctly – as in ranging from player instantiation and movement to jump dynamics and health management – as well as how the systems interact with each other and how it responds to external influences like enemy attacks or level design changes. Most of what I have implemented are boundary tests verifying that values stay within defined ranges or values, stress testing to understand how well the feature performs under stressful conditions and what the failure points are (e.g. when many enemies attack simultaneously can the health updates correctly keep track of changes…). Implemented tests are specifically:   * Boundary test for minimum speed * Boundary test for maximum speed * Stress test for repeated simultaneous damage to player health from multiple enemies * Instantiation test to make sure player is constructed with all required components * Jump test to verify input is indeed translated into upward movement * Healing test to confirm health remains within bounds and doesn’t go beyond max * Knockback test to make sure external forces, like enemy impacts, result in correct velocity changes * Damage bypass test to make sure player damage processing respects BC cheat mode   Other ideas I have in my plan is I could expand to doing integration and regression tests which would be useful to run specifically when changes have been introduced by teammates.  So far none of my currently implemented tests have found a bug in my code due to a teammate adding code or game objects. Though some tests I have could let me or a teammate know if that happens – for example my stress test RepeatDmg.cs can be ran when my teammate adds new enemy behaviors or modes of attack that send damage to the player. In the case that damage is done extremely rapidly in succession or something is miscalculated causing the player’s health to drop below zero or skip the player death sequence the test would detect that because player’s health should never go negative and the game-over scene should be triggered correctly without crashing.  Pick a Prefab you have created that is documented well in a separate readme file.  Prefab Name: Player  View PrefabReadMe.html  (I will point to several places in your code documentation and ask)  What question where you trying to answer here?  A: (example) So whoever wants to use the prefab knows what the player components do in a general sense  Who do you anticipate would be asking that question?  Techdown developers mostly who create a new scene and need to add a player or for testing reasons. Or perhaps other developers going through our game to learn or to help.  What other questions might this person need the answers to?  View my Prefab doc page, which are some questions those asking may have. Like understanding the player scripts, how to adjust things, how do add new abilities to the player prefab if needed, etc.  -----------------------------------------------------------------------------------------------------------------  Show me a class in your code where there could be either static or dynamic binding.  So another small feature I implemented in the game on the fly to move to the next level was a lever object that the player collides into. The class there was currently static, but this mechanism could be easily experimented with to have a dynamic version. I renamed the original class to have the keyword ‘Static’  using UnityEngine;  using UnityEngine.SceneManagement;  using UnityEngine.Events;  public class StaticLeverCollision : MonoBehaviour  {      //[SerializeField] private UnityEvent \_collisionEvent;      [SerializeField] private string collideTag = "Player";      void OnCollisionEnter2D(Collision2D col)      {          if (col.collider.CompareTag(collideTag))          {              ChangeSceneStatic(2); //static bound method call              gameObject.SetActive(false);          }      }      void ChangeSceneStatic(int sceneID)      {          Debug.Log("Static method called. Changing to scene ID: " + sceneID);          SceneManager.LoadScene(sceneID);      }  }  Moved to here:  BrainStew’s GitHub  Assets > src > tl2 > JJ-Scripts > DynamicBinding  **To view before and after mp4, view attached staticVSdynamic.mp4 on Canvas**  Write some mock code on this paper showing how you would set the static type and dynamic type of a variable.  Here is what I did to get this feature into dynamic. Since I still need some script to be able to attach to a unity object, I would have to use a monobehaviour script to call another script that doesn’t inherit from monobehaviour for dynamic binding…  using UnityEngine;  using UnityEngine.SceneManagement;  using UnityEngine.Events;  public class DynamicLeverCollision : MonoBehaviour  {      private LeverBase lever;      //[SerializeField] private UnityEvent \_collisionEvent;      [SerializeField] private string collideTag = "Player";      void Start()      {          lever = new LeverDynamic(2); //init with scene ID 2      }      void OnCollisionEnter2D(Collision2D col)      {          if (col.collider.CompareTag(collideTag))          {              lever.ChangeScene(); //dynamically bound method call              //SceneManager.LoadScene(2);              gameObject.SetActive(false);          }      }  }  Which calls….  using UnityEngine;  using UnityEngine.SceneManagement;  public class LeverBase  {      public virtual void ChangeScene()      {          Debug.Log("Base class method called.");      }  }  public class LeverDynamic : LeverBase  {      private int sceneID;      public LeverDynamic(int sceneID)      {          this.sceneID = sceneID;      }      public override void ChangeScene()      {          Debug.Log("Dynamic class method called. Changing to scene ID: " + sceneID);          MockChangeScene(sceneID);      }      private void MockChangeScene(int sceneID)      {          Debug.Log($"Mock changing scene to ID: {sceneID}");          UnityEngine.SceneManagement.SceneManager.LoadScene(sceneID);      }  }  Super Class: \_\_\_\_\_\_\_Lever Base\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  Sub Class: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_LeverDynamic\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  Virtual Function: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ChangeScene()\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  Choose a dynamically bound method. What method gets called now?  When I instantiate my variable with:  Lever = new LeverDynamic(2);  And then call  Lever.ChangeScene()  The dynamically bound method called is the overridden method in LeverDynamic. That is, LeverDynamic.ChangeScene() gets called, and prints:  “Dynamic class method called. Changing to scene ID: 2”  Then calls MockChangeScene(2) which loaded scene 2 via SceneManager.LoadScene(2)  Change the dynamic type. What method gets called now?  If I change the dynamic type of the same variable by doing:  Lever = new LeverBase();  And then call  Lever.ChangeScene();  Now the base class version is invoked and prints:  “Base class method is called.”  Which shows that when the variable’s actual (runtime) type is LeverDYnamic, the overridden dynamic method is invoked. And when its type is simply LeverBase, the base (non-overriden) method is used instead.  Pick a statically bound method. Which one would be called in each of the two previous cases?  In my static script (StaticLeverCollision.cs), I just have the method:  ChangeSceneStatic(int sceneID)  Which is called directly from the collision event. No matter what dynamic type might be at work in the dynamic one, the static method call does not use virtual. It always calls that method as defined in the script at compile time vs at runtime. The static one its behavior is not determined by the objects runtime type.  -------------------------------------------------------------------------------------------------------------------  Show me an example of reuse in your code where you violate copyright law.  I used a lever image found on google for the lever object’s sprite in two of our scenes – used as the transition interactable to get to next levels.  <https://www.freepik.com/icon/control-lever_416142>  How does it violate copyright?  The source site with this lever image has a Legal section and at the bottom it says Copyright © 2010-2025 Freepik Company S.L. All rights reserved.  So I could argue going off of what’s described in the legal section, that the assets, like graphics or icons, are still intellectual property. This company’s assets and collaborator content they sponsor on the site are protected by copyright and some form of intellectual property rights, plus belong to their owners.   The images found on there specify what kind of licenses they have and what usage is permitted for graphic assets. Using the lever asset in BrainStew’s game and claiming that asset as part of our own creation for the game then later selling the game with no credit to the source could pose violation. Also Freepik prohibits selling derived works based on their assets. So selling it without proper licensing or competing with their services in any way is also violation.  What did you have to do to integrate it with the code you wrote?  It’s simply a transparent png image dragged to Unity’s sprite renderer component – it’s for lever object that has Box Collider 2D with a player-lever collision event to change scenes to the next level’s scene ID  What are the legal implications if you market your code with the re-used portion?  The main implication for this is a copyright violation strike or claim and having to go in and remove it (takedown notice) or face legal fines. Freepik could pursue legal action. If financial penalties or strikes happen, that also implies reputational damage to BrainStew, making it harder to secure partnerships or funding in the future.  Use fair use argue that you can use this anyway.  Despite the earlier argument though, I found that this image specifically does has a free use license and I pixelated the lever icon substantially to state fair use. In other words, it’s transformative enough that it adds value beyond the original work. And we are not selling the asset itself as it’s part of our overarching creative work. It will not harm Freepik’s market either because our game doesn’t compete with their services or similar graphic assets.  -------------------------------------------------------------------------------------------------------------------  4. **One big** or two small, well-chosen patterns.  Small Patterns = {Singleton, Private Class Data}  Which patterns did you choose?  1.\_\_\_\_\_\_\_\_\_\_\_\_\_Strategy Pattern\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  Why did you choose each pattern? (Justify your use of it).  The pattern I found my code to employ is the strategy pattern.  I selected this pattern because this is what my code happens to have currently – when I created the logic for my player controller feature, I was not aware we were required to have code to reflect some pattern until TL3s introduced us to the concept. By then, most of my core player mechanics were already done and was being expanded on.  I have no reason to change the logic or make new code to abide by any of the other patterns, and thus, this is the closest pattern that it aligns with.  <https://sourcemaking.com/design_patterns/strategy>  <https://refactoring.guru/design-patterns/strategy>  From my understanding, the strategy pattern is a behavioral design pattern that lets you define a family of interchangeable algorithms, encapsulating each one and making them independently interchangeable at runtime. This approach adheres to the open/closed principle by enabling new behaviors to be added without modifying existing client code. In practice, it replaces extensive conditional logic with polymorphism, making the system more modular and easier to test and extend…(sources above)  My code has this strategy pattern because of the way it handles input from the user:  So it has abstract input behavior…the abstract class InputController.cs is the strategy interface by defining methods like RetrieveMoveInput(), RetrieveJumpInput(), and RetrieveFastFallInput(). This abstraction is key because it hides the details on how input is captured.  using UnityEngine;  public abstract class InputController : ScriptableObject  {      public abstract float RetrieveMoveInput();      public abstract bool RetrieveJumpInput();      public abstract bool RetrieveFastFallInput();  }  The PlayerController.cs, is the concrete implementation derived from the InputController.cs, encapsulates the specific behavior for handling input from the keyboard. This is nice because ti allows me to swap the implementation with another (for different control schemes or types of input) without changing the input data.  using UnityEngine;  [CreateAssetMenu(fileName = "PlayerController", menuName = "InputController/PlayerController")]  public class PlayerController : InputController  {      public override bool RetrieveJumpInput()      {          return Input.GetKeyDown(KeyCode.W) ||                 Input.GetKeyDown(KeyCode.Space) ||                 Input.GetKeyDown(KeyCode.UpArrow);      }      public override float RetrieveMoveInput()      {          return Input.GetAxisRaw("Horizontal");      }      public override bool RetrieveFastFallInput()      {          return Input.GetKey(KeyCode.S) ||                 Input.GetKey(KeyCode.DownArrow);      }  }  Then the CONTROLLER class holds a refence to my InputController.cs (scriptable object), acting as the context that delegates input retrieval. My player capabilities like Move.cs and Jump.cs get their input behavior through reference, so behavior can be switched at runtime or extended if needed.  using UnityEngine;  public class CONTROLLER : MonoBehaviour  {      public InputController input = null;  }  I thought when coding this would be nice, because it’s very readable for whoever is porting controls beyond keyboard – so encapsulating input behavior makes it easy to introduce new input ways without altering core logic of player or game mechanics. According to the website that explains this pattern, that’s a core benefit to using strategy. More described in pros and cons section…  Overall,  I use the Strategy pattern to decouple the way input is retrieved from the behavior of my player object. SO the abstract class InputController.cs defines the contract for obtaining input, and the concrete class PlayerController.cs implements this contract. The CONTROLLER.cs holds a reference to an InputController—allowing us to plug in different input strategies if needed. This separation encapsulates the varying algorithm (how input is detected) away from the component that uses it (game movement or jumping).  Move.cs  using/referencing the controllers  using UnityEngine;  [RequireComponent(typeof(CONTROLLER))]  public class Move : MonoBehaviour  {      [SerializeField,Range(0f,100f)] private float \_maxSpeed = 4f;      [SerializeField,Range(0f,100f)] private float \_maxAcceleration = 35f;      [SerializeField,Range(0f,100f)] private float \_maxAirAcceleration = 20f;      private CONTROLLER \_controller;      private Vector2 \_direction,\_desiredVelocity,\_velocity;      private Rigidbody2D \_body;      private Ground \_ground;      private float \_maxSpeedChange,\_acceleration;      private bool \_onGround;      public float \_KBForce;      public float \_KBCounter;      public float \_KBTotalTime;      public bool \_HitFromRight;      void Awake()      {          \_body = GetComponent<Rigidbody2D>();          \_ground = GetComponent<Ground>();          \_controller = GetComponent<CONTROLLER>();      }      void Update()      {          \_direction.x = \_controller.input.RetrieveMoveInput();          \_desiredVelocity = new Vector2(\_direction.x,0f) \* Mathf.Max(\_maxSpeed - \_ground.Friction,0f);      }      private void FixedUpdate()      {          \_onGround = \_ground.OnGround;          \_velocity = \_body.linearVelocity;          \_acceleration = \_onGround ? \_maxAcceleration : \_maxAirAcceleration;          \_maxSpeedChange = \_acceleration \* Time.deltaTime;          \_velocity.x = Mathf.MoveTowards(\_velocity.x, \_desiredVelocity.x, \_maxSpeedChange);          if (\_KBCounter <= 0)          {          \_body.linearVelocity = \_velocity;          }          else          {              if (\_HitFromRight)              {                  \_body.linearVelocity = new Vector2(-\_KBForce\*2, \_KBForce);              }              else              {                  \_body.linearVelocity = new Vector2(\_KBForce\*2, \_KBForce);              }              \_KBCounter -= Time.deltaTime;          }      }  }  Pros and Cons  Pros   * Flexibility & Extensibility   It lets me or other members swap out input behaviors without modifying core game logic. Like if later I want to add support for touch input (though our TL6 didn’t see this it’s fine), you can just add that to inherit from InputController and update the reference in CONTROLLER   * Decoupling & Maintainability   Delegating input logic to independent component, the code is modular and not cluttered. So it’s easier to find stuff actually even if you have to look at all files. Then to test and maintain each part of the system independently (my vertical and horizontal movement is separate too). So it sticks to the requirement of the open/closed principle of this pattern since new input methods can be added without modifying code   * Simplified Conditional Logic   Less multiple conditional statements like in a single player controller script to handle different input sources.  Cons   * Overhead with just one implementation   When there’s only one strategy (PlayerController.cs) used, the additional abstraction is kind of extra complexity. If I needed to swap behaviors at runtime, maybe its over-engineered? But it still works well in the end   * Additional Class Abstractions   The extra layer of abstraction increases the number of classes/file management in my project. So you’d have to look at all files to see what’s going on sometimes   * Runtime Indirection   Even though flexibility is nice here, the indirection by selecting and managing strategies might be adding a slight performance overhead in some scenarios but its not bad  Draw the class diagram for your pattern(s).  Original class diagram:    With my class names:    Would something else have worked as well or better than this pattern? When would be a bad time to use this pattern?  I mean, maybe? This is what I ended up making and happened to resemble the strategy pattern. I like how I wrote it and the controls are fun to use in-game so I don’t see myself changing anything.  Since I only have one instance of the player at a time per scene, maybe the observer pattern would work well for player feature. The state pattern is a given for managing player states and animations. Singleton pattern would be easy to setup for Player Stats as well to be globally accessible.  The observer one would work because it’s useful to broadcast health changes to UI like the healthbar. The player object acts as a subject to notify healthbar UI to update in real-time. But I think strategy remains the best for the input mechanics itself.  When would be a bad time to use the strategy pattern though?  If our project only ever uses one input method (just keyboard and nothing else) then the extra abstraction adds unnecessary complexity, I suppose. But our project does indeed require porting controls. | /4  /3  /3  /4  /4 |