All of the following code is written in C# and using the Unity game engine. Unity invites developers to use an object-oriented design approach via an Entity Component System. Every class derives from the base class `MonoBehaviour`.

Task 1: Following a design principle — Dependency Inversion Principle

Some years ago I developed a multiplayer game. In the game, players received different missions (with different ways to accomplish the required goal(s)). Each of the missions have some common functionality, which was implemented in the following abstract class called `MissionBase`. Each mission class inherits from this base class. Also, they have to implement the abstract method `MissionAccomplished()`.

```
1  public abstract class MissionBase : MonoBehaviour
2  {
3      public int Points;
4
5      protected bool _missionIsActive;
6      public bool MissionIsActive { get { return _missionIsActive; } }
7
8      public MissionType MissionType;
9
10  // Omitted methods:
11  public virtual void InitializeMission(GameObject player, MissionBase ↩
        Template) {...}
12  public virtual void UpdateSpecificMissionStuff() {...}
13
14     public abstract bool MissionAccomplished();
15
16 }
```

Then the singleton class `MissionManager` loops through all active missions and checks if any of them have been accomplished. If so, certain things are triggered, such as audio cues and GUI text popups. Also, each time a mission has been accomplished, a random door will either open or close. This is triggered by the use of delegates and events, as shown in the `MissionManager` (lines 9-11 and 42-57).

```
1  public class MissionManager : MonoBehaviour
2  {
3      // Singleton itself
4      private static MissionManager _instance;
5
6      public List<MissionBase> AllAvailableMissionsTotal;
7      public List <GameObject> Players;
8
9      public delegate void MissionCompleted();
10     public static event MissionCompleted OnMissionCompletedDoorsUpper;
11     public static event MissionCompleted OnMissionCompletedDoorsLower;
12
13     void Update()
14     {
```

```
15    // only check if game is playing
16          if (GameManager.Instance.PlayingState != PlayingState.Playing)
17              return;
18
19          for (int i = InstantiatedMissions.Count - 1; i >= 0; i--)
20          {
21              MissionBase m = InstantiatedMissions[i];
22              if (m != null)
23              {
24               // dont look into inactive missions
25                  if (!m.MissionIsActive)
26
27                      return;
28      // look if mission has been accomplished
29                  if (m.MissionAccomplished())
30                  {
31                      GameObject g = (GameObject)Instantiate(↩
                            MissionIsCompletedPrefab);
32
33                      g.CreateMissionAccomplishedText();
34                      m.GivePointsToPlayer();
35
36                      DoorGoUpDown();
37                  }
38              }
39          }
40      }
41
42      public void DoorGoUpDown()
43      {
44          int random = Random.Range(0, 2);
45
46          switch (random)
47          {
48              case 0:
49                  if (OnMissionCompletedDoorsLower != null)
50                      OnMissionCompletedDoorsUpper();
51                  break;
52              case 1:
53                  if (OnMissionCompletedDoorsLower != null)
54                      OnMissionCompletedDoorsLower();
55                  break;
56          }
57      }
58  }
```

Each door then subscribes to these events, as shown in the `Door` class.

```
1   public enum DoorLocation
2   {
3       Upper,
4       Lower
5   }
6
7   public class Door : MonoBehaviour
8   {
```

```
9     private Vector3 doorOpenScale;
10    private Vector3 doorOpenPos;
11    private goingUp, goingDown;
12
13    public DoorLocation DoorLocation;
14
15    void OnEnable()
16    {
17        if (this.DoorLocation == DoorLocation.Upper)
18            MissionManager.OnMissionCompletedDoorsUpper += DoorGoDown;
19        else if (this.DoorLocation == DoorLocation.Lower)
20            MissionManager.OnMissionCompletedDoorsLower += DoorGoDown;
21    }
22
23    void OnDisable()
24    {
25        if (this.DoorLocation == DoorLocation.Upper)
26            MissionManager.OnMissionCompletedDoorsUpper -= DoorGoDown;
27        else if (this.DoorLocation == DoorLocation.Lower)
28            MissionManager.OnMissionCompletedDoorsLower -= DoorGoDown;
29
30        if (this.DoorLocation == DoorLocation.Upper)
31            MissionManager.OnMissionCompletedDoorsUpper -= DoorGoUp;
32        else if (this.DoorLocation == DoorLocation.Lower)
33            MissionManager.OnMissionCompletedDoorsLower -= DoorGoUp;
34    }
35
36    void DoorGoUp()
37    {
38        StartCoroutine(OpenDoor());
39    }
40
41    void DoorGoDown()
42    {
43        StartCoroutine(CloseDoor());
44    }
45
46    // Coroutines in Unity make it easier to time events
47    IEnumerator CloseDoor()
48    {
49        // Different things happens here to open the door
50        // eg. enabling the renderer and changing local scale via Vector3.↩
              Lerp()
51    }
52
53    // Coroutines in Unity make it easier to time events
54    IEnumerator OpenDoor()
55    {
56        // Different things happens here to open the door
57        // eg. enabling the renderer and changing local scale via Vector3.↩
              Lerp()
58    }
59 }
```

I would argue that this way of using delegates and events is related to the *Dependency Inversion Principle*, following the Hollywood principle of "Don't call us, we'll call you!" The

abstraction (mission accomplished) should not depend on the details (doors open/close), but vice versa. This is done via the `MissionManager` class that acts as a link between the missions and the doors. The `MissionManager` and the `Mission` classes don't care what happens when `DoorGoUpDown()` gets called; it's up to the `Door` class to put in functionality to the events.

Task 2: Violation of a design principle — Single Responsibility Principle

As an example of violating the *Single Responsibility Principle*, I have chosen the class `GameManager`. We were three guys working on a game, and we had a tight deadline, meaning that we didn't spend a lot of time planning on how to implement the classes. The initial idea behind the `GameManager` class was to have a singleton-like class that stored references to the different game objects, almost like a middle-man (it is very difficult to use static classes in Unity, so it's often preferred to use a singleton).

As the name suggests, the `GameManager` class is responsible for a lot of system-like things that were needed to be configured in order for the game to work. However, the class turned out to be a go-to place for everything that was not directly related with a game entity (e.g., the player characters). Said in another way: when unsure where to put code, the programmers used the `GameManager` as a "garbage bin". The `GameManager` class ended up being very big and containing a lot of responsibilities, including: storing references to all player characters; handling input; handling the camera movement; keeping track of player scores; as well as scene handling. Needless to say, this made the class very fragile, which often created a cascading domino-effect whenever we needed to make a tiny change in the code. Instead, we should have delegated each of the responsibilities out to a separate class, say, for input handling or keeping track of the players.

The class became very big and difficult to navigate in, with more than 1000 lines of code and no real structure. At the time of developing the game, most things made sense to us, but looking back at it now, it's a gigantic mess with lots of convoluted code. One sign of this is the extensive use of comments, which was/is necessary to keep track of what's happening.

```
1  public class GameManager : MonoBehaviour
2  {
3   // prefabs
4   public Transform mainCamera;
5   public GameObject VikingPrefab;
6   public GameObject HammerPrefab;
7
8   // input
9   private bool TopLeftPressed, TopRightPressed, BottomLeftPressed, ↩
        BottomRightPressed;
10  public bool UseKeyboardInsteadOfTouch = true;
11  private Vector2 ScreenCenter;
12
13  // powerups
14  private float powerUpSpawnCounter = 1;
15
16  // characters and sprites
17  public Transform TopLeftSpawn, TopRightSpawn, BottomLeftSpawn, ↩
        BottomRightSpawn;
18  int NumberOfPlayers;
19  private List<GameObject> PlayerObjects;
20
21  public Sprite DonutBody, DonutArm, DonutPortrait;
```

```
22  public Sprite EyeBody, EyeArm, EyePortrait;
23
24  private int[] spritesUsed;
25  public VikingPlayer TopLeftPlayer, TopRightPlayer, BottomLeftPlayer, ↩
        BottomRightPlayer;
26  public Transform DeadPosition;
27  public float selectCooldownTopLeft, selectCooldownTopRight, ↩
        selectCooldownBottomLeft, selectCooldownBottomRight;
28  public GameObject StartButton;
29  public Text ChooseUniqueCharacter;
30
31  // scores
32  public int PlayersAlive;
33  private bool canPlayNextRound = false;
34  public GameObject WinnerButton;
35
36  // camera
37  private Camera MainCamera;
38  private float[] xPositions;
39  private float[] yPositions;
40  private float shakeAmount;
41
42  // collision
43  public List<Collider2D> boundaryColliders = new List<Collider2D>();
44
45  // sounds
46  public MusicPlayer MusicPlayer;
47  private AudioSource audioSource;
48     public AlertSound AlertSound;
49     public GameObject SmackSound;
50
51
52     //Round timer
53     public bool UseShrinkingLevel = true;
54     public float RoundTimer = 30f;
55
56  // ... a lot of code has been omitted, but here are examples of some of ↩
        the methods:
57  Start() {...}
58  Update(){...}
59  StartGame(){...}
60  FixedUpdate(){...}
61  GetInputs(){...}
62  AddPlayer(){...}
63  ShowScoreBoard(){...}
64  PlayNextRound{...}
```

Task 3: Two design smells

1) Rigidity

I once developed a 2D platformer game in C#. The game had a lot of tweakable parameters. The goal was to measure the influence on the player's perception of the game when these parameters were changed. Therefore, it was important to be able to save the parameters to a format that could then be send to a MySQL database. All of the parameters were stored in the class `TweakableParameters`. However, since there were so many parameters, adding/removing a single parameter required changes in multiple places in the same file. This would often result in errors, because I sometimes forgot to add new parameter code in all of the appropriate places.

It was very cumbersome to change any of the parameters. First of all, there are static fields for each parameter that define the range of a given value. This is used so that an outside class can choose a random value within a given range. Then there is the constructor class itself, which takes in a lot of parameters. For debug purpose, I decided that it would be possible to pass in a NULL parameter, which would result in the system using the default parameters defined in the fields themselves. The most problematic method was `ToStringDatabaseFormat()`, which has the purpose of printing the parameters out in the correct format and order. It was very hard to make any changes here, since the order has to be the exact same as defined in a Javascript file on a server. Just a single mistake in this method would result in unusable data. I dread every time I had to make a tiny adjustment in the code — it was very rigid and fragile. Especially `string.Format()` in the method `ToStringDatabaseFormat()` was painful to use, since it was impossible to spot if I wrote a small mistake in the long string (line 124).

```
1
2  public class TweakableParameters
3  {
4      // Ranges (used for random choosing)
5      static public Vector2 GravityRange = new Vector2(-5f,-30.1f);
6      static public Vector2 TerminalVelocityRange = new Vector2(-5,-60.1f);
7      static public Vector2 JumpPowerRange = new Vector2(2f,30.1f);
8      static public Vector2 AirFrictionHorizontalPercentageRange = new ↩
           Vector2(0, 99.1f);
9      static public Vector2 GhostJumpTimeRange = new Vector2(0f,2.1f);
10     static public Vector2 MinimumJumpHeightRange = new Vector2(0.1f, 5.1f)↩
           ;
11     static public Vector2 ReleaseEarlyJumpVelocityRange = new Vector2(0f, ↩
           3.1f);
12     static public Vector2 ApexGravityMultiplierRange = new Vector2(1f, ↩
           15.1f);
13     static public Vector2 MaxVelocityXRange = new Vector2(1, 20.1f);
14     static public Vector2 GroundFrictionPercentageRange = new Vector2(0f, ↩
           99.1f);
15     static public Vector2 ReleaseTimeRange = new Vector2(0.001f, 3.1f);
16     static public Vector2 AttackTimeRange = new Vector2(0.001f, 3.1f);
17     static public Vector2 TurnAroundBoostPercentRange = new Vector2(100f, ↩
           400.1f);
18     static public Vector2 AnimationMaxSpeedRange = new Vector2(50f, 150f);
```

```
19
20      // Constructor to set new parameters
21      public TweakableParameters(float? gravity, float? jumpPower, bool? ↩
            useAirFriction, bool? keepGroundMomentumAfterJump, float? ↩
            airFrictionHorizontal,
22          float? terminalVelocity, float? ghostJumpTime, float? ↩
                minimumJumpHeight, float? releaseEarlyJumpVelocity,
23          float? apexGravityMultiplier, float? maxVelocityX, bool? ↩
                useGroundFriction, float? groundFrictionPercentage,
24          float? releaseTime, float? attackTime, float? ↩
                turnAroundBoostPercent,
25          bool? useAnimation, float? animationMaxSpeed, int? isDuplicate)
26      {
27
28          if (gravity.HasValue)
29              Gravity = new Vector3(0, gravity.Value, 0);
30          if (jumpPower.HasValue)
31              JumpPower = jumpPower.Value;
32
33          if (useAirFriction.HasValue)
34              UseAirFriction = useAirFriction.Value;
35
36          if (airFrictionHorizontal.HasValue)
37              AirFrictionHorizontalPercentage = airFrictionHorizontal.Value;
38
39          if (keepGroundMomentumAfterJump.HasValue)
40              KeepGroundMomentumAfterJump = keepGroundMomentumAfterJump.↩
                    Value;
41
42          if (terminalVelocity.HasValue)
43              TerminalVelocity = terminalVelocity.Value;
44
45          if (ghostJumpTime.HasValue)
46              GhostJumpTime = ghostJumpTime.Value;
47
48          if (minimumJumpHeight.HasValue)
49              MinimumJumpHeight = minimumJumpHeight.Value;
50
51          if (releaseEarlyJumpVelocity.HasValue)
52              ReleaseEarlyJumpVelocity = releaseEarlyJumpVelocity.Value;
53
54          if (apexGravityMultiplier.HasValue)
55              ApexGravityMultiplier = apexGravityMultiplier.Value;
56
57          if (maxVelocityX.HasValue)
58              MaxVelocityX = maxVelocityX.Value;
59
60          if (useGroundFriction.HasValue)
61              UseGroundFriction = useGroundFriction.Value;
62
63          if (groundFrictionPercentage.HasValue)
64              GroundFrictionPercentage = groundFrictionPercentage.Value;
65
66          if (releaseTime.HasValue)
67              ReleaseTime = releaseTime.Value;
68
69          if (attackTime.HasValue)
```

```
70              AttackTime = attackTime.Value;
71
72          if (turnAroundBoostPercent.HasValue)
73              TurnAroundBoostPercent = turnAroundBoostPercent.Value;
74
75          if (useAnimation.HasValue)
76              UseAnimation = useAnimation.Value;
77
78          if (animationMaxSpeed.HasValue)
79              AnimationMaxSpeed = animationMaxSpeed.Value;
80
81          if (isDuplicate.HasValue)
82              IsDuplicate = 1;
83          else
84              IsDuplicate = 0;
85      }
86
87      // FIELDS
88
89      // air
90      public Vector3 Gravity = new Vector3(0, -30f, 0);
91      public float JumpPower = 20;
92      public bool UseAirFriction = false;
93      public float AirFrictionHorizontalPercentage = 90f;
94      public bool KeepGroundMomentumAfterJump = false;
95      public float TerminalVelocity = -30f;
96      public float GhostJumpTime = 0.2f;
97      public float MinimumJumpHeight = 2f;
98      public float ReleaseEarlyJumpVelocity = 0.5f;
99      public float ApexGravityMultiplier = 3;
100
101     // ground
102     public float MaxVelocityX = 15f;
103     public bool UseGroundFriction = false;
104     public float GroundFrictionPercentage = 50f;
105     public float ReleaseTime = 0.4f;
106     public float AttackTime = 0.4f;
107     public float TurnAroundBoostPercent = 0f;
108
109     // animation
110     public bool UseAnimation = true;
111     public float AnimationMaxSpeed = 100f;
112
113     // extra
114     public int IsDuplicate;
115
116     // stage
117     public int Level = 0;
118     public int Deaths = 0;
119     public float TimeSpentOnLevel = 0;
120
121     // Print to MySQL database
122     public string ToStringDatabaseFormat()
123     {
124             return string.Format("&Gravity={0}&JumpPower={1}&↩
                    AirFrictionHorizontalPercentage={2}&TerminalVelocity={3}&↩
                    GhostJumpTime={4}&MinimumJumpHeight={5}&↩
```

```
                ReleaseEarlyJumpVelocity={6}&ApexGravityMultiplier={7}&↩
                MaxVelocityX={8}&ReleaseTime={9}&AttackTime={10}&↩
                AnimationMaxSpeed={11}&Level={12}&Deaths={13}&↩
                TimeSpentOnLevel={14}",
125             Gravity.y,
126             JumpPower,
127             AirFrictionHorizontalPercentage,
128             TerminalVelocity,
129             GhostJumpTime,
130             MinimumJumpHeight,
131             ReleaseEarlyJumpVelocity,
132             ApexGravityMultiplier,
133             MaxVelocityX,
134             ReleaseTime,
135             AttackTime,
136             AnimationMaxSpeed,
137             ParameterManager.Instance.Level + 1,
138             StateManager.Instance.DeathsOnThisLevel,
139             StateManager.Instance.TimeSpentOnLevel);
140     }
141 }
```

2) Needless Repetition

I wrote the following code as one of my first bigger game projects in Unity. It's a game where you control one or more monsters. Depending on what mode the game is in, different events are triggered. The `MonsterHighlightSound` class is responsible for playing an audio clip depending on the monster's distance to a record player. In this class, there are three given monster objects. In the `Update()` method, the system checks each of the three monsters with some very basic if-else statements. The problem here is that the code is basically repeated three times, which on a larger scale (more than three monsters) could create some significant maintenance problems. A more clever approach would be to generalize the functionality into a method. Then every monster object could be added to a list and iterated through in a foreach loop.

```
1  public class MonsterHighlightSound : MonoBehaviour {
2
3      public GameObject monster1_origin;
4      public GameObject monster2_origin;
5      public GameObject monster3_origin;
6
7      private float minVolumeDistance = 0.5f;
8
9      public int distanceToMove = 10;
10
11     private bool monster1_activate;
12     private bool monster2_activate;
13     private bool monster3_activate;
14
15  void Update ()
16     {
17         if (GameMode.MyGameMode == Mode.Zooming)
18         {
```

```
19              //////// MONSTER 1 ////////////////////////
20
21              float volumeMonster1 = 1 - (Mathf.Abs(transform.position.x - ↩
                    monster1_origin.transform.position.x));
22              monster1_origin.GetComponent<AudioSource>().volume = ↩
                    volumeMonster1;
23
24              if (volumeMonster1 >= minVolumeDistance && monster1_activate ↩
                    == false)
25              {
26                  monster1_activate = true;
27                  monster1_origin.GetComponent<AudioSource>().Play();
28              }
29              else if (volumeMonster1 < minVolumeDistance && ↩
                    monster1_activate == true)
30              {
31                  monster1_activate = false;
32                  monster1_origin.GetComponent<AudioSource>().Stop();
33              }
34
35              //////// MONSTER 2 ////////////////////////
36
37              float volumeMonster2 = 1 - (Mathf.Abs(transform.position.x - ↩
                    monster2_origin.transform.position.x));
38              monster2_origin.GetComponent<AudioSource>().volume = ↩
                    volumeMonster2;
39
40              if (volumeMonster2 >= minVolumeDistance && monster2_activate ↩
                    == false)
41              {
42                  monster2_activate = true;
43                  monster2_origin.GetComponent<AudioSource>().Play();
44              }
45              else if (volumeMonster2 < minVolumeDistance && ↩
                    monster2_activate == true)
46              {
47                  monster2_activate = false;
48                  monster2_origin.GetComponent<AudioSource>().Stop();
49              }
50
51              //////// MONSTER 3 ////////////////////////
52
53              float volumeMonster3 = 1 - (Mathf.Abs(transform.position.x - ↩
                    monster3_origin.transform.position.x));
54              monster3_origin.GetComponent<AudioSource>().volume = ↩
                    volumeMonster3;
55
56              if (volumeMonster3 >= minVolumeDistance && monster3_activate ↩
                    == false)
57              {
58                  monster3_activate = true;
59                  monster3_origin.GetComponent<AudioSource>().Play();
60              }
61              else if (volumeMonster3 < minVolumeDistance && ↩
                    monster3_activate == true)
62              {
63                  monster3_activate = false;
```

```
64                  monster3_origin.GetComponent<AudioSource>().Stop();
65              }
66          }
67      }
68  }
```