Game Engine Design & implementation Assignment 1

Video Report Link: https://youtu.be/qZQ2rIbJ5IY

Team and Responsibilities

Manvir Punglia (100828507):

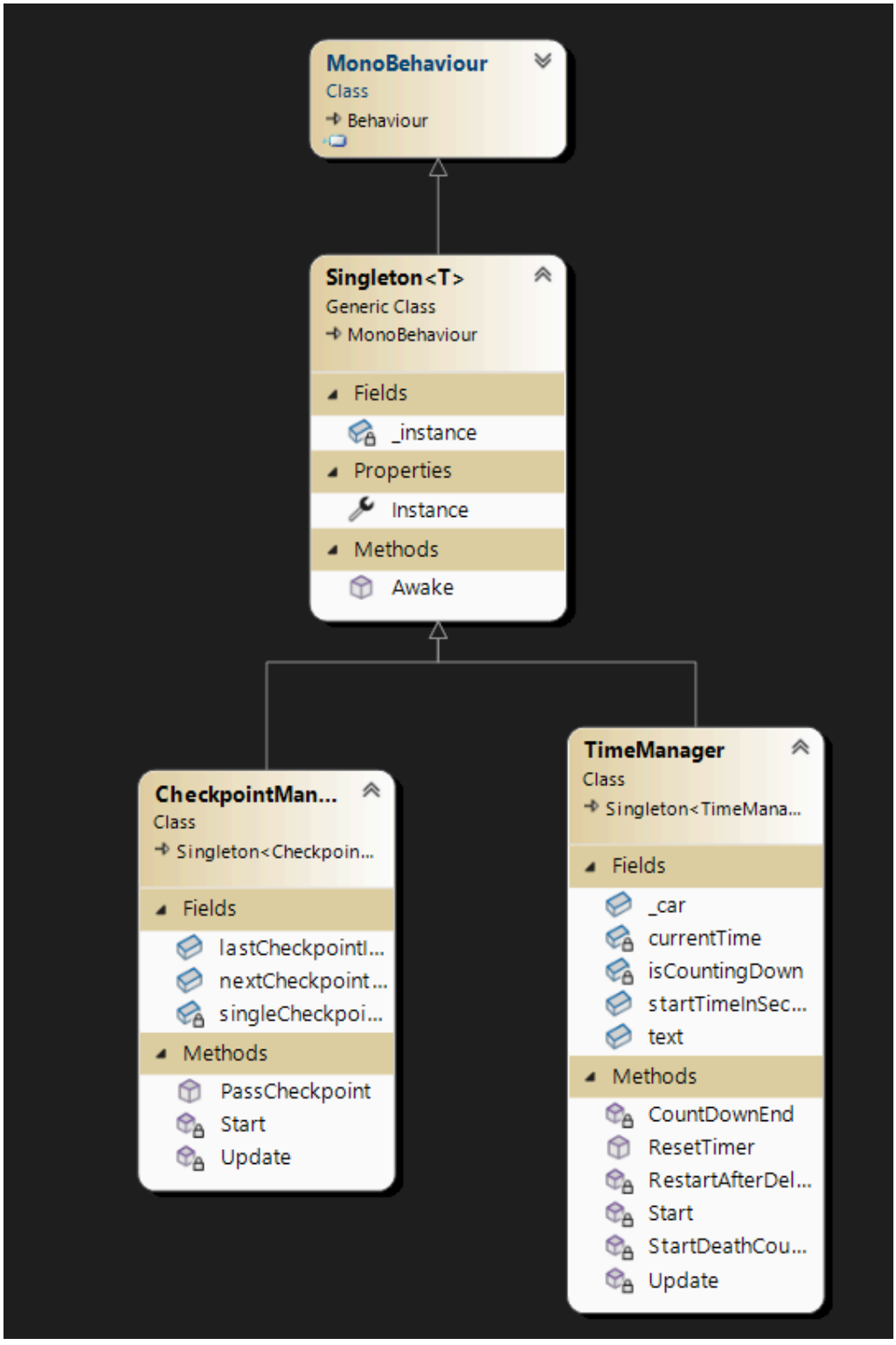
- Singleton
  - Time Manager
- Command design pattern
  - Custom button mapping
- Factory pattern
  - Scene population (Buildings)
- Observer
  - Speedometer
- The UMLs for each design pattern done

Saad Khan (100829159):

- Singleton
  - Checkpoint manager
- Command design pattern
  - Car customization
- Factory pattern
  - Obstacle creation
- Observer
  - Checkpoint tracker
- The UMLs for each design pattern done

UMLs and Explanations

Singleton



Time Manager

The Time Manager inherits from the Singleton Abstract Class we were given in class, It is a singleton because we want to ensure that only one version of it is going at a time since it will kill the player when the time increments to 0. It uses a Coroutine to count down but the timer will reset if the car hits a checkpoint.

```
1 reference
IEnumerator StartDeathCountdown()
{
    isCountingDown = true;

    while (currentTime > 0)
    {
        currentTime -= Time.deltaTime;

        text.text = Mathf.Ceil(currentTime).ToString();

        yield return null;
    }

    currentTime = 0;
    text.text = "0";
    isCountingDown = false;

    CountDownEnd();
}
```

```
public void ResetTimer()
{
    currentTime = startTimeInSeconds;

    if (!isCountingDown)
    {
        StartCoroutine(StartDeathCountdown());
    }
}
```

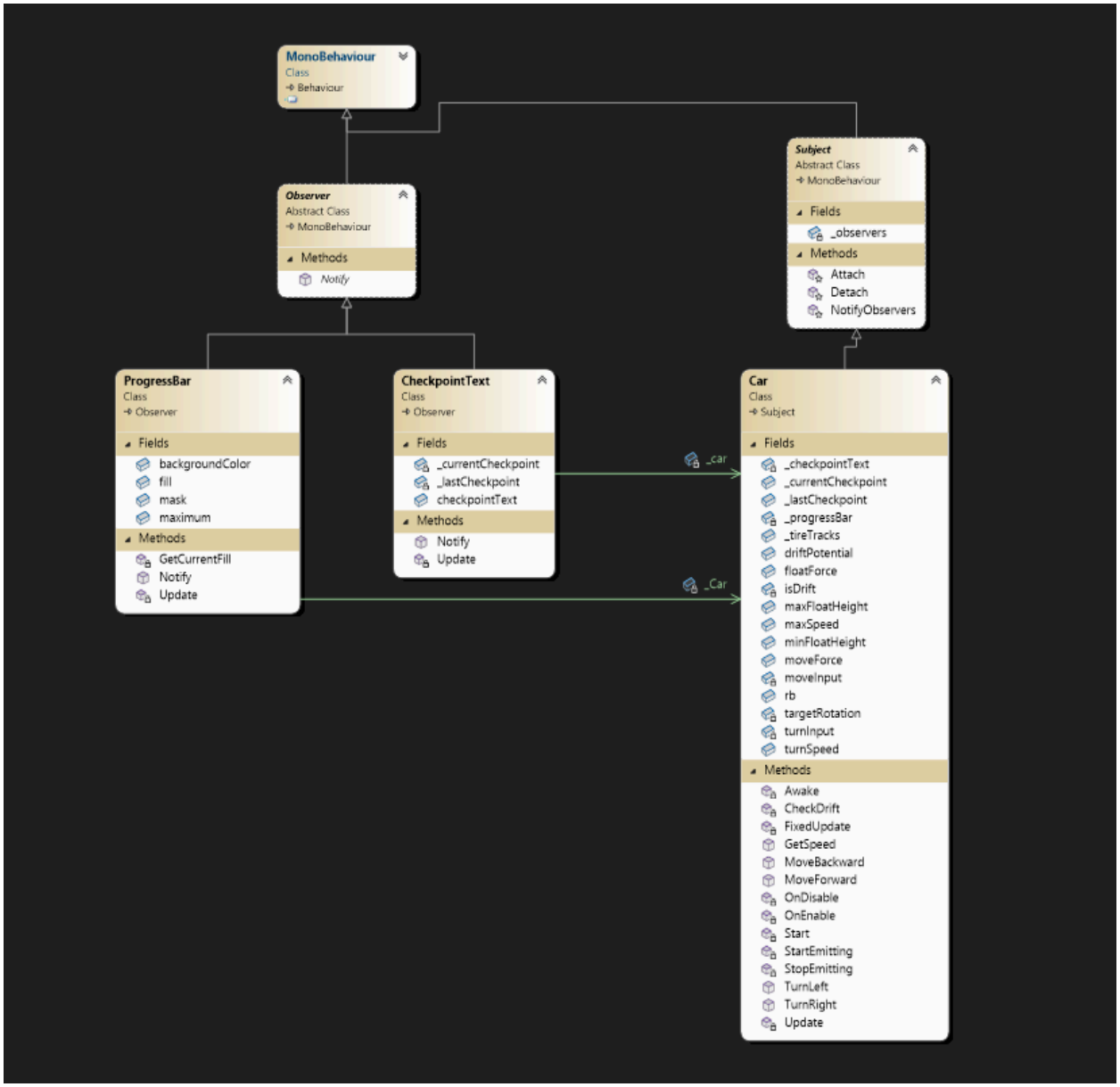
Checkpoint Manager

The Checkpoint Manager is responsible for keeping track of the individual checkpoints on the track. It creates a list, finds all checkpoints in the level, and stores them in it. Additionally, it works in tandem with a separate script to handle when the player crosses a checkpoint. In the backend, every time the player crosses a checkpoint, a value that tracks which checkpoint the player is currently on will increment.

```
Unity Script (1 asset reference) | 6 references
public class CheckpointManager : Singleton<CheckpointManager>
{
    private List<SingleCheckpoint> singleCheckpointList;
    [HideInInspector] public int nextCheckpointIndex;
    [HideInInspector] public int lastCheckpointIndex;

    Unity Message | 0 references
    void Start()
    {
        Transform checkpointsTransform = transform.Find("Checkpoints");
        if (checkpointsTransform == null)
        {
            Debug.Log("Couldn't find the checkpoints");
        }

        singleCheckpointList = new List<SingleCheckpoint> ();

        foreach (Transform Checkpoint in checkpointsTransform)
        {
            SingleCheckpoint singleCheckpoint = Checkpoint.GetComponent<SingleCheckpoint>();
            singleCheckpoint.setCheckpoints(this);
            singleCheckpointList.Add(singleCheckpoint);
        }

        nextCheckpointIndex = 0;
    }
```

```
1 reference
public void PassCheckpoint(SingleCheckpoint singleCheckpoint)
{
    if (singleCheckpointList.IndexOf(singleCheckpoint) == nextCheckpointIndex)
    {
        Debug.Log("Correct Checkpoint");
        nextCheckpointIndex++;
    }
    else
    {
        Debug.Log("Wrong Checkpoint");
    }
}
```

Observer



Speedometer

The Car class which is the controller for the player object is the Subject and the Speedometer is the Observer. The Car attaches all _progressBar observers to itself OnEnable and sends out a Notify in its Update function. This Notify is received by the ProgressBar object which then sets its current progress value to the speed for the car. This is then updated to the Speedometer object in the Scene.

```
Unity Message | 0 references
private void OnEnable()
{
    Attach(_progressBar);
    Attach(_checkpointText);
}
```

```
void Update()
{
    //Debug.Log(rb.velocity.magnitude);
    NotifyObservers();
    _currentCheckpoint = CheckpointManager.In
    _lastCheckpoint = CheckpointManager.Insta

    CheckDrift();
}
```

```
1 reference
void GetCurrentFill()
{
    float fillAmount = (float)current/(float)maximum;
    mask.fillAmount = fillAmount;
    fill.color = backgroundColor;
}

Unity Message | 0 references
private void Update()
{
    GetCurrentFill();
}

2 references
public override void Notify(Subject subject)
{

    _Car = subject.GetComponent<Car>();
    current = (int)_Car.GetSpeed();
}
```

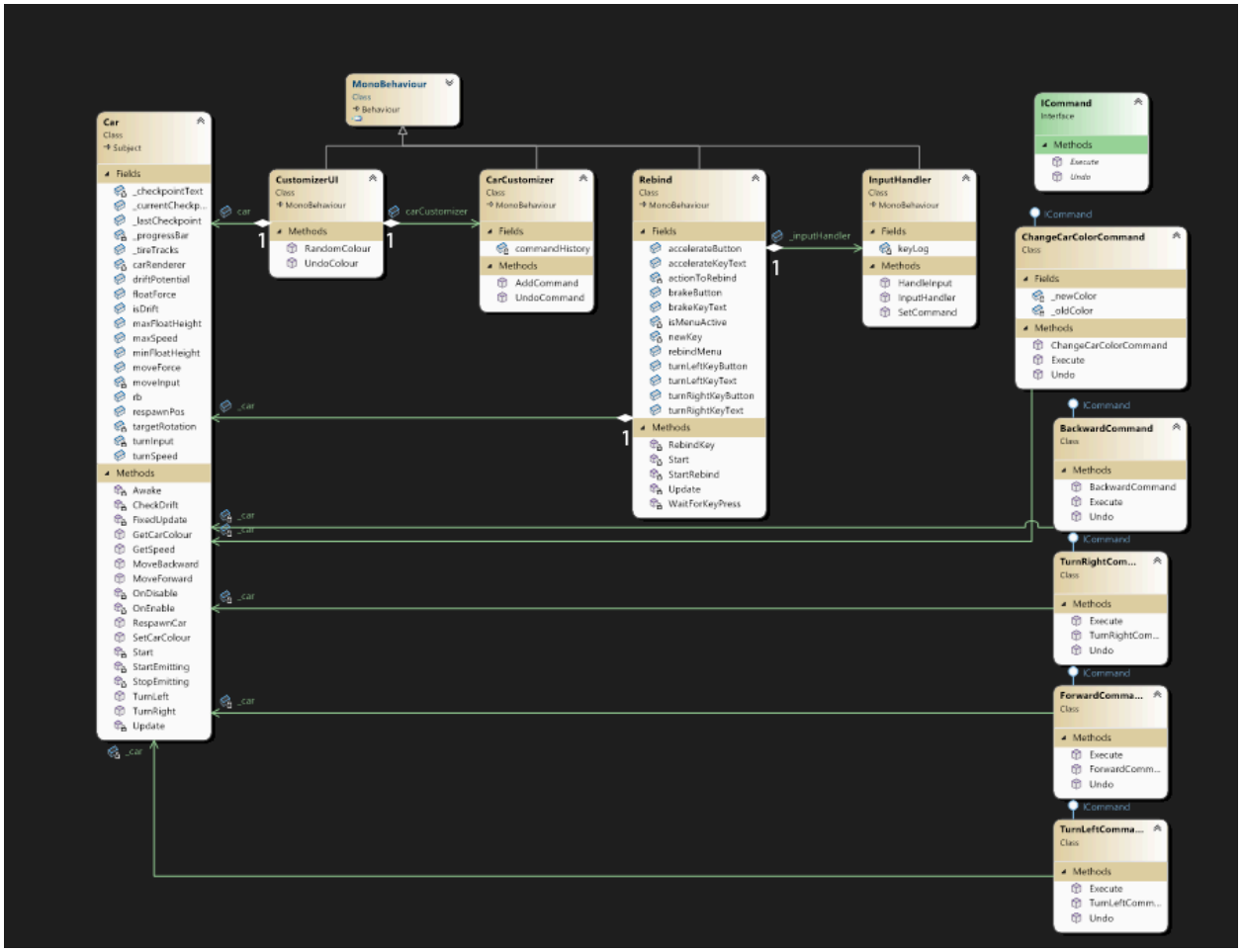
Checkpoint Tracker

The Checkpoint Tracker works with the Checkpoint Manager to handle and keep track of the checkpoints on the track. More specifically, the Checkpoint Tracker (the observer) will update the checkpoint UI element to increment the checkpoint based on when the car (the subject) moves past a checkpoint.

```
2 references
public override void Notify(Subject subject)
{
    _car = subject.GetComponent<Car>();
    _currentCheckpoint = _car._currentCheckpoint;
    _lastCheckpoint = _car._lastCheckpoint;
}
```

```
Unity Message | 0 references
void Update()
{
    if (checkpointTextObject != null)
    {
        checkpointTextObject.GetComponent<Text>().text = (_currentCheckpoint.ToString()) + "/" + _lastCheckpoint;
    }

}
```

Command



Button Remapping

The Button Remaper uses the Abstract Command class to create 4 concrete classes that represent the movement of the car. These commands access the car object passed to them from the InputHandler(Invoker) through the Interface and execute the movement functions. The InputHandler(Invoker) stores a Dictionary that has a Keycode(Key) and a ICommand(entry), and has a function which allows new vales to be added to the dictionary and detects if any of the inputs from the dictionary have been trigger and if they have it Executes the related Command. The Rebinder connects with the UI in the Scene, it initially sets the base buttons and then listens for buttonpresses and uses SetCommand function to add he new input keys to the Dictonary.

```
3 references
public class ForwardCommand :ICommand
{
    private Car _car;

    2 references
    public ForwardCommand(Car car)
    {
        _car = car;
    }

    3 references
    public  void Execute()
    {
        _car.MoveForward();
    }

    2 references
    public void Undo()
    {

    }
}
```

```
public class BackwardCommand : ICommand
{
    private Car _car;

    2 references
    public BackwardCommand(Car car)
    {
        _car = car;
    }

    3 references
    public  void Execute()
    {
        _car.MoveBackward();
    }

    2 references
    public void Undo()
    {

    }
}
```

```
public class InputHandler:MonoBehaviour
{
    private Dictionary<KeyCode, ICommand> keyLog;

    0 references
    public InputHandler()
    {
        keyLog = new Dictionary<KeyCode, ICommand>();
    }

    8 references
    public void SetCommand(KeyCode key, ICommand command)
    {
        keyLog[key] = command;
    }

    1 reference
    public void HandleInput()
    {
        foreach (var entry in keyLog)
        {
            if (Input.GetKey(entry.Key))
            {
                entry.Value.Execute();
                Debug.Log(entry.Value);
            }
        }
    }
}
```

```
// Default
_inputHandler.SetCommand(KeyCode.W, new ForwardCommand(_car));
_inputHandler.SetCommand(KeyCode.S, new BackwardCommand(_car));
_inputHandler.SetCommand(KeyCode.A, new TurnLeftCommand(_car));
_inputHandler.SetCommand(KeyCode.D, new TurnRightCommand(_car));
```

```
accelerateButton.onClick.AddListener(() => StartRebind("Accelerate"));
brakeButton.onClick.AddListener(() => StartRebind("Brake"));
turnLeftKeyButton.onClick.AddListener(() => StartRebind("Left"));
turnRightKeyButton.onClick.AddListener(() => StartRebind("Right"));
```

```
void RebindKey(string action, KeyCode key)
{
    switch (action)
    {
        case "Accelerate":
            _inputHandler.SetCommand(key, new ForwardCommand(_car));
            accelerateKeyText.text = key.ToString();
            Debug.Log("Accelerate bind: " + key.ToString());
            break;
        case "Brake":
            _inputHandler.SetCommand(key, new BackwardCommand(_car));
            brakeKeyText.text = key.ToString();
            break;
        case "Left":
            _inputHandler.SetCommand(key, new TurnLeftCommand(_car));
            turnLeftKeyText.text = key.ToString();
            break;
        case "Right":
            _inputHandler.SetCommand(key, new TurnRightCommand(_car));
            turnRightKeyText.text = key.ToString();
            break;
    }
}
```

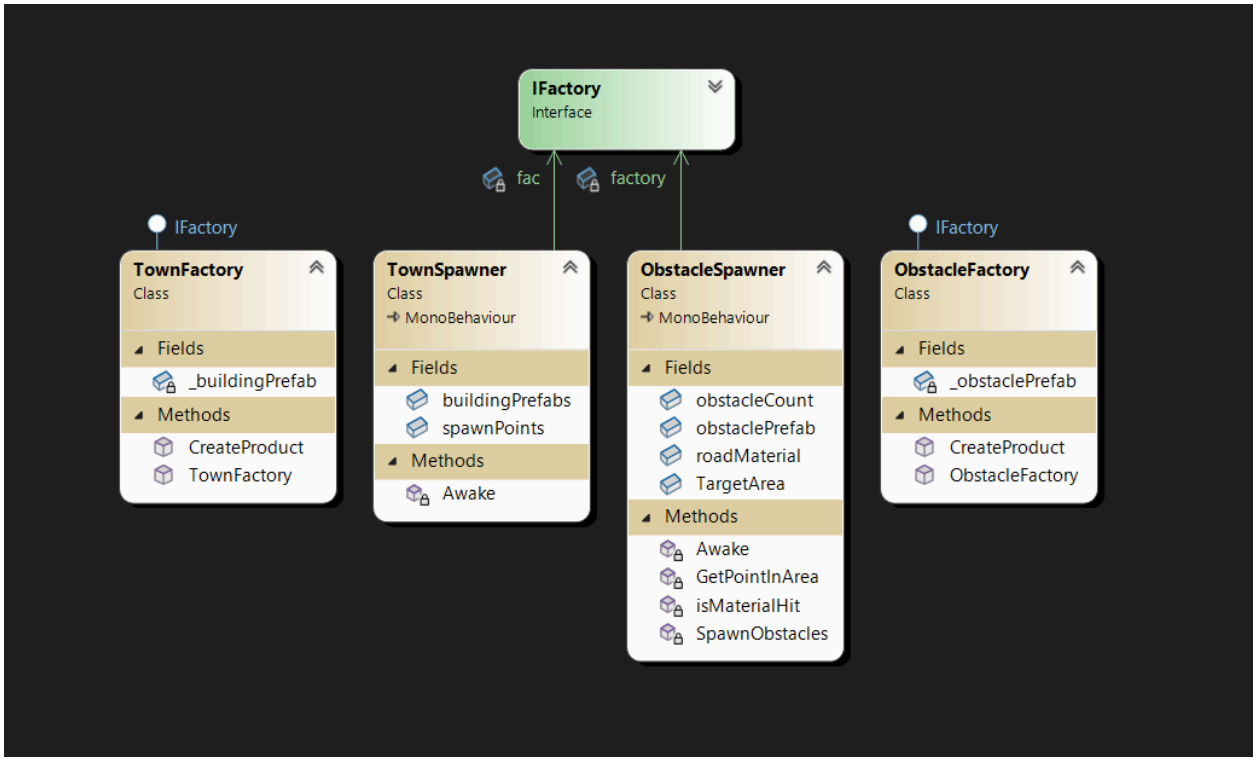
Car Customizer

The Car Customizer is responsible for changing the appearance of the car. There is one concrete command that currently changes the look of the car, the ChangeCarColourCommand will randomly select a colour for the car and apply it through the invoker, the Car Customizer. Additionally, The Car Customizer can undo the colour change command; by utilizing a Stack, we can push and pop commands with the add and undo commands respectively. Lastly, two buttons in the UI allow you to make these changes, one for randomly changing the colour and another for undoing the change.

```
Unity Script (1 asset reference) | 1 reference
public class CarCustomizer : MonoBehaviour
{
    private Stack<ICommand> commandHistory = new Stack<ICommand>();

    1 reference
    public void AddCommand(ICommand command)
    {
        command.Execute();
        commandHistory.Push(command);
    }

    1 reference
    public void UndoCommand()
    {
        if (commandHistory.Count > 0)
        {
            ICommand command = commandHistory.Pop();
            command.Undo();
        }
    }
}
```

```
2 references
public class ChangeCarColorCommand : ICommand
{
    private Car _car;
    private Color _newColor;
    private Color _oldColor;

    1 reference
    public ChangeCarColorCommand(Car car)
    {
        this._car = car;
        this._oldColor = _car.GetCarColour();
        this._newColor = new Color(Random.value, Random.value, Random.value);
    }

    3 references
    public void Execute()
    {
        _car.SetCarColour(_newColor);
    }

    2 references
    public void Undo()
    {
        _car.SetCarColour(_oldColor);
    }
}
```

Factory



Scene population (Buildings)

The Town Factory creates a concrete factory which uses the IFactory interface, it has Constructor which sets the prefab it is supposed to spawn and a CreateProduct function which Instantiates the object in the world. The TownSpawner holds a list of prefabs which it will spawn and the location the prefabs need to go. OnAwake it places the prefabs randomly on all the locations listed in the spawnPoints array.

```
public class TownFactory : IFactory
{
    private GameObject _buildingPrefab;

    1 reference
    public TownFactory(GameObject buildingPrefab)
    {
        _buildingPrefab = buildingPrefab;
    }

    3 references
    public GameObject CreateProduct()
    {
        return GameObject.Instantiate(_buildingPrefab);
    }
}
```

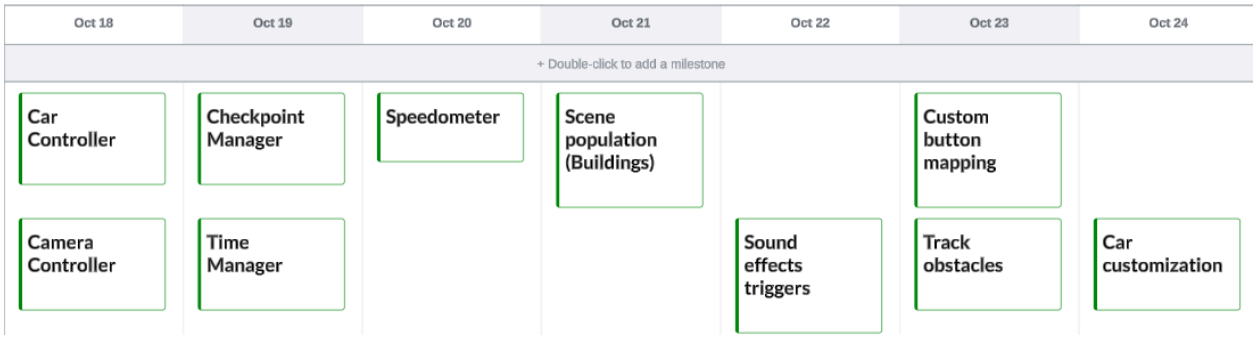
```
public class TownSpawner : MonoBehaviour
{
    public GameObject[] buildingPrefabs;
    public Transform[] spawnPoints;
    IFactory fac;

    Unity Message | 0 references
    void Awake()
    {
        for (int i = 0; i < spawnPoints.Length; i++)
        {

            int rand = Random.Range(0, buildingPrefabs.Length);


            IFactory fac = new TownFactory(buildingPrefabs[rand]);

            GameObject building = fac.CreateProduct();


            building.transform.position = spawnPoints[i].position;
            building.transform.rotation = spawnPoints[i].rotation;
        }
    }
}
```

Obstacle creation

The Obstacle Factory is responsible for creating random obstacles on the track. It will collect the obstacle prefabs, and randomly select one for every instantiation, however, there is one condition put in place before it can instantiate. The objects will only spawn if the material is a predefined target material, the logic behind this involves checking the potential spawn position within a large area, if the spawn position contains the target material, the obstacle will be placed. In the case it tries to spawn in a position where the material is on and the target material is not the same, it will loop through a for loop until it does for every obstacle.

```
void SpawnObstacles()
{
    for (int i = 0; i < obstacleCount; i++)
    {
        Vector3 spawnPos = GetPointInArea(TargetArea);

        if (isMaterialHit(spawnPos))
        {
            int rand = Random.Range(0, obstaclePrefab.Length);

            IFactory factory = new ObstacleFactory(obstaclePrefab[rand]);

            GameObject product = factory.CreateProduct();

            product.transform.position = spawnPos;
            product.transform.rotation = Quaternion.AngleAxis(Random.Range(0f, 360f), Vector3.up);

        }
        else
        {
            i--;
        }

    }
}
```

Timeline:



Use of third-party resources

EasyRoads3D: https://www.easyroads3d.com/
It was used as a tool to create the road structure.

Checkpoint System: https://www.youtube.com/watch?v=IOYNg6v9sfc&t=531s
The reason behind using this system is that it creates a robust experience that fulfills the needs of what we're trying to achieve. The contributions I made were implementing a function that resets the checkpoints, this function will be invoked every time the player explodes and respawns. I also created a flag that will check whether the checkpoint objects are null, this way, the game won't throw an error and will notify me if the checkpoint objects don't fill the list automatically for any reason.

CineMachine: https://unity.com/unity/features/editor/art-and-design/cinemachine
It was used to provide a smooth camera when turning and moving.

Visual Studio Class Designer:
https://learn.microsoft.com/en-us/visualstudio/ide/class-designer/designing-and-viewing-classes-and-types?view=vs-2022
It was used to create the UML diagrams for each pattern created and discussed.

Unity. (2021). *Level Up Your Code With Game Programming Patterns* (LTS EDITION).

https://unity.com/resources/level-up-your-code-with-game-programming-patterns

Used to help understand design patterns, UMLs and their implementation into unity