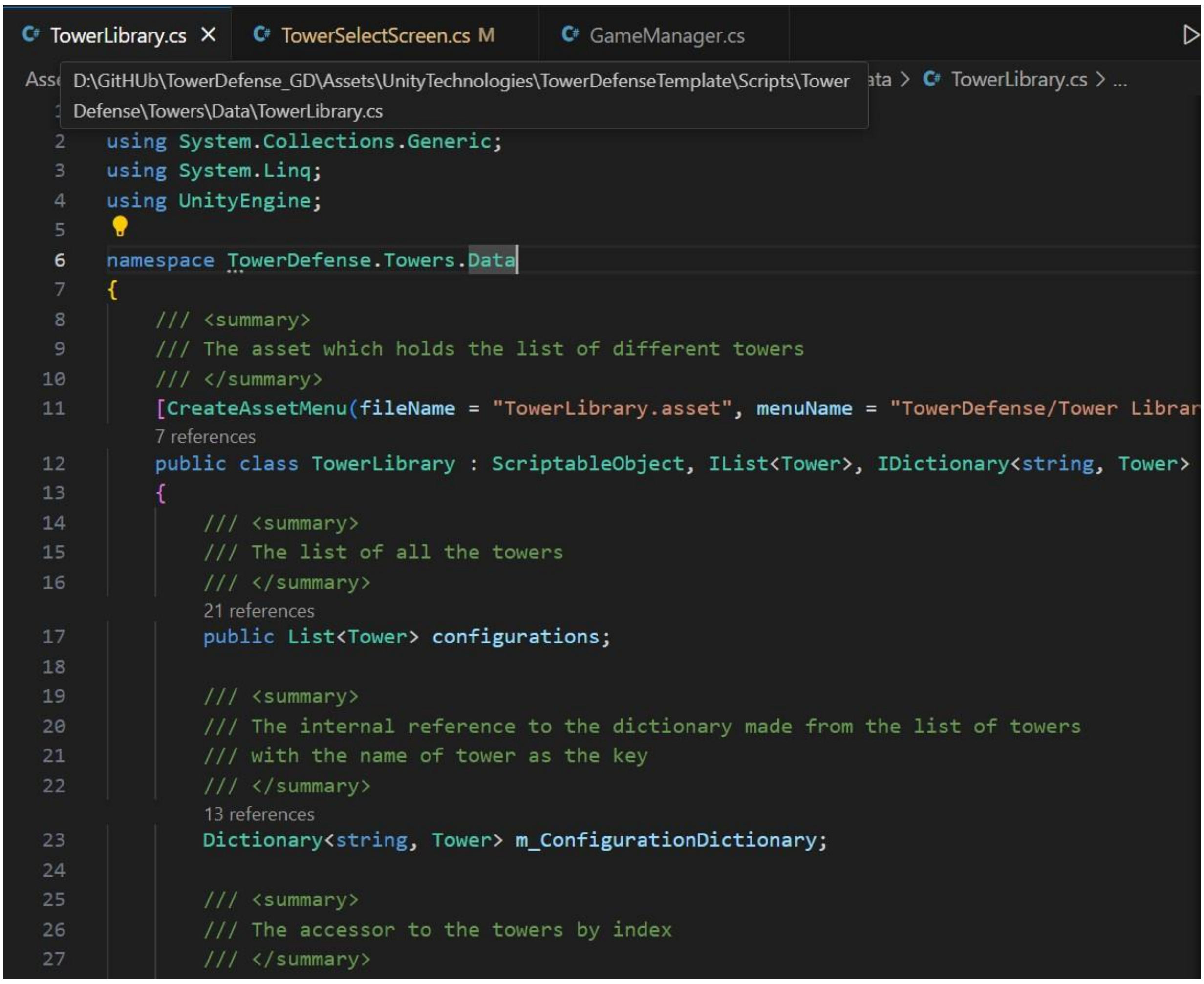
Chapter 3

Some Important Code Screenshots

3.1 TowerLibrary.cs



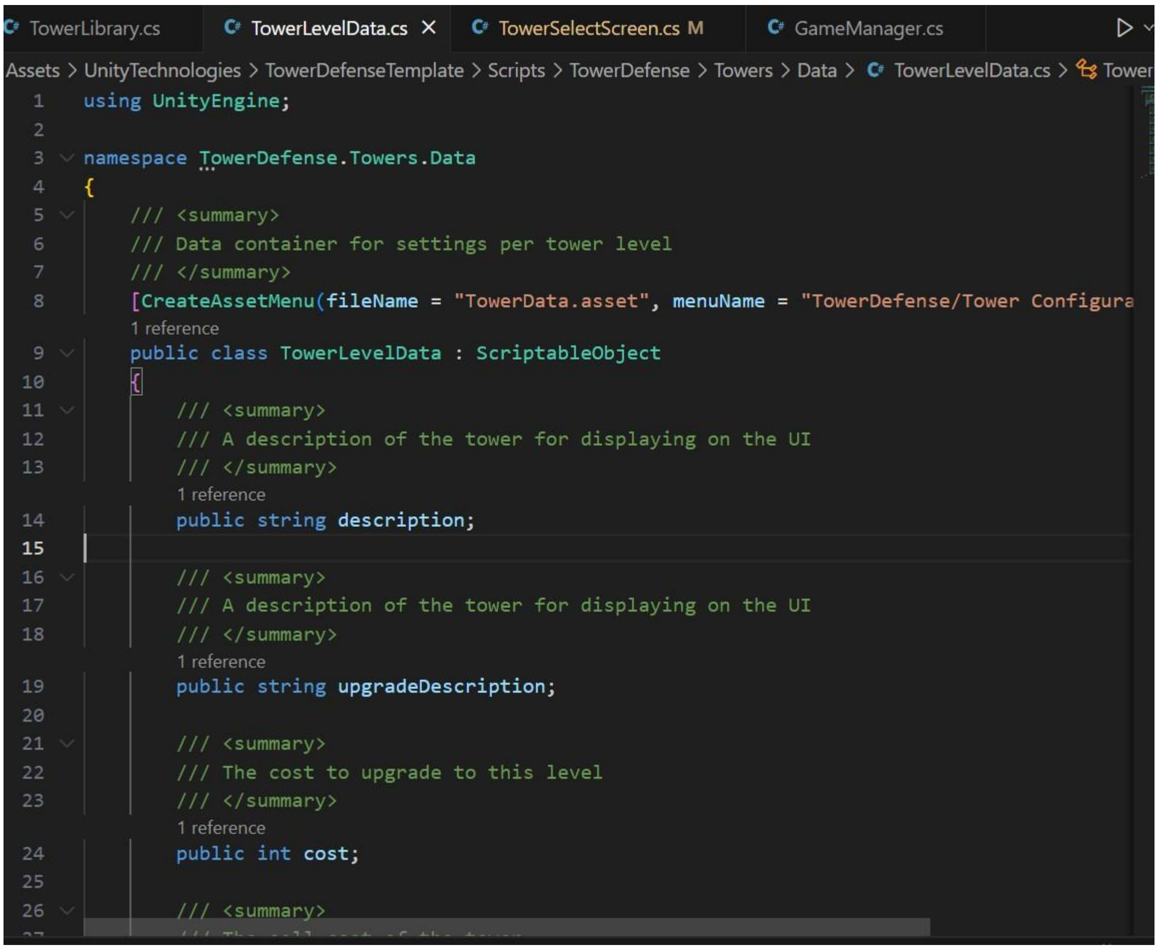
TowerLibrary.cs
TowerSelectScreen.cs M
GameManager.cs
D:\GitHUb\TowerDefense_GD\Assets\UnityTechnologies\TowerDefenseTemplate\Scripts\Tower Defense\Towers\Data\TowerLibrary.cs
TowerLibrary.cs > ...

```
using System.Collections.Generic;
using System.Linq;
using UnityEngine;

namespace TowerDefense.Towers.Data
{
    /// <summary>
    /// The asset which holds the list of different towers
    /// </summary>
    [CreateAssetMenu(fileName = "TowerLibrary.asset", menuName = "TowerDefense/Tower Librar
    7 references
    public class TowerLibrary : ScriptableObject, IList<Tower>, IDictionary<string, Tower>
    {
        /// <summary>
        /// The list of all the towers
        /// </summary>
        21 references
        public List<Tower> configurations;

        /// <summary>
        /// The internal reference to the dictionary made from the list of towers
        /// with the name of tower as the key
        /// </summary>
        13 references
        Dictionary<string, Tower> m_ConfigurationDictionary;

        /// <summary>
        /// The accessor to the towers by index
        /// </summary>
```

Figure 3.1: TowerLibrary Script

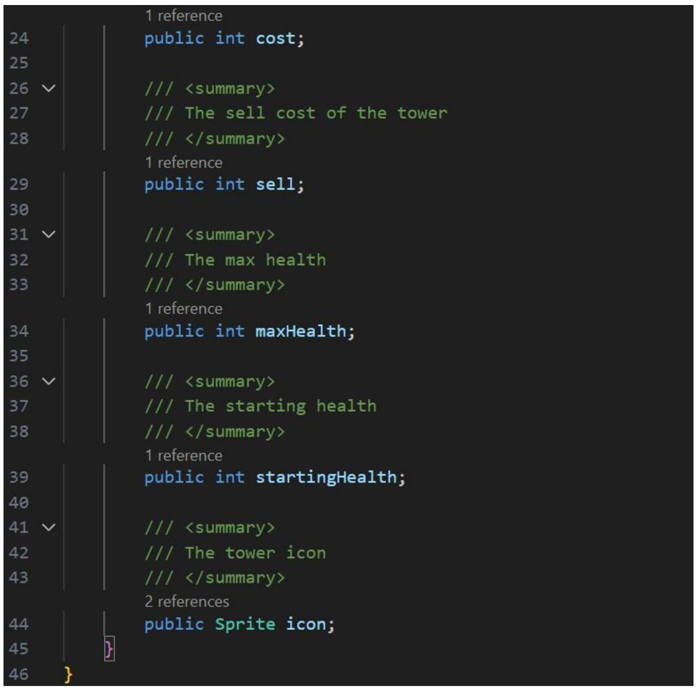
The Script in Figure 3.1 is a ScriptableObject that manages a collection of Tower objects, providing both list-based and dictionary-based access. It implements IList and IDictionary to enable retrieval by index and by tower name.

3.2 TowerLevelData.cs


TowerLibrary.cs
TowerLevelData.cs
TowerSelectScreen.cs M
GameManager.cs
Assets > UnityTechnologies > TowerDefenseTemplate > Scripts > TowerDefense > Towers > Data > TowerLevelData.cs > Tower

```
using UnityEngine;

namespace TowerDefense.Towers.Data
{
    /// <summary>
    /// Data container for settings per tower level
    /// </summary>
    [CreateAssetMenu(fileName = "TowerData.asset", menuName = "TowerDefense/Tower Configura
    public class TowerLevelData : ScriptableObject
    {
        /// <summary>
        /// A description of the tower for displaying on the UI
        /// </summary>
        public string description;

        /// <summary>
        /// A description of the tower for displaying on the UI
        /// </summary>
        public string upgradeDescription;

        /// <summary>
        /// The cost to upgrade to this level
        /// </summary>
        public int cost;

        /// <summary>
```

Figure 3.2: TowerLevelData Script Part 1

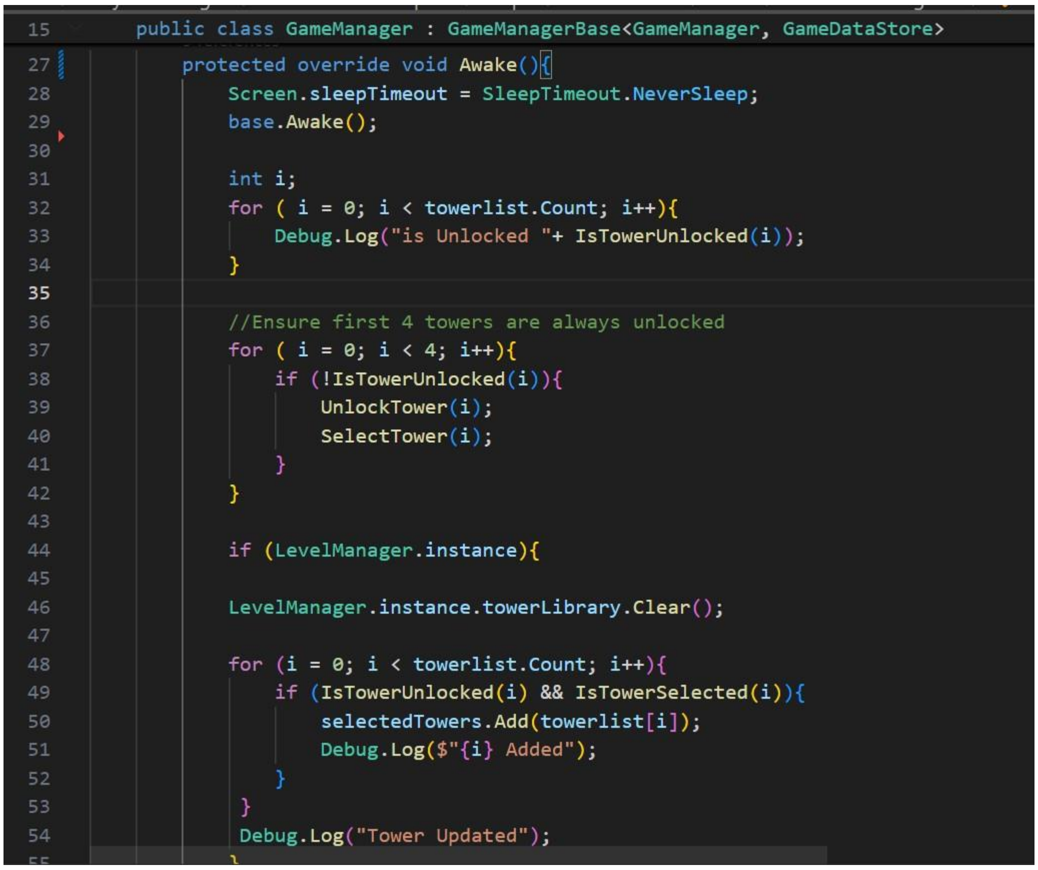


```
        public int cost;

        /// <summary>
        /// The sell cost of the tower
        /// </summary>
        public int sell;

        /// <summary>
        /// The max health
        /// </summary>
        public int maxHealth;

        /// <summary>
        /// The starting health
        /// </summary>
        public int startingHealth;

        /// <summary>
        /// The tower icon
        /// </summary>
        public Sprite icon;
    }
}
```

Figure 3.3: TowerLevelData Script Part 2

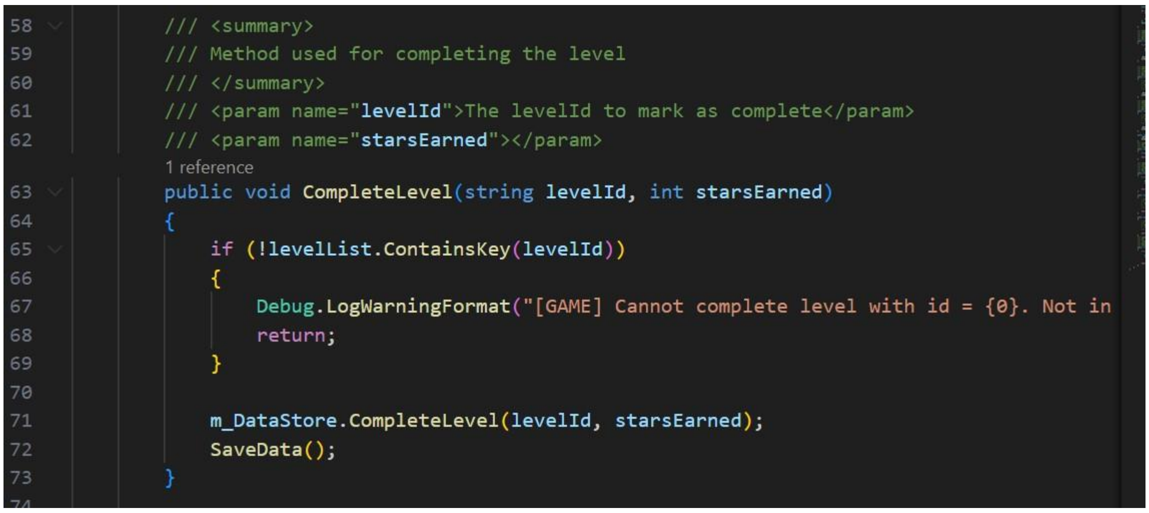
The script in figure [3.2,3.3] is a ‘ScriptableObject‘ that stores settings for each tower level, including descriptions, cost, health, and an icon for UI display. It helps manage tower upgrades and attributes in a Tower Defense game.

3.3 GameManager.cs



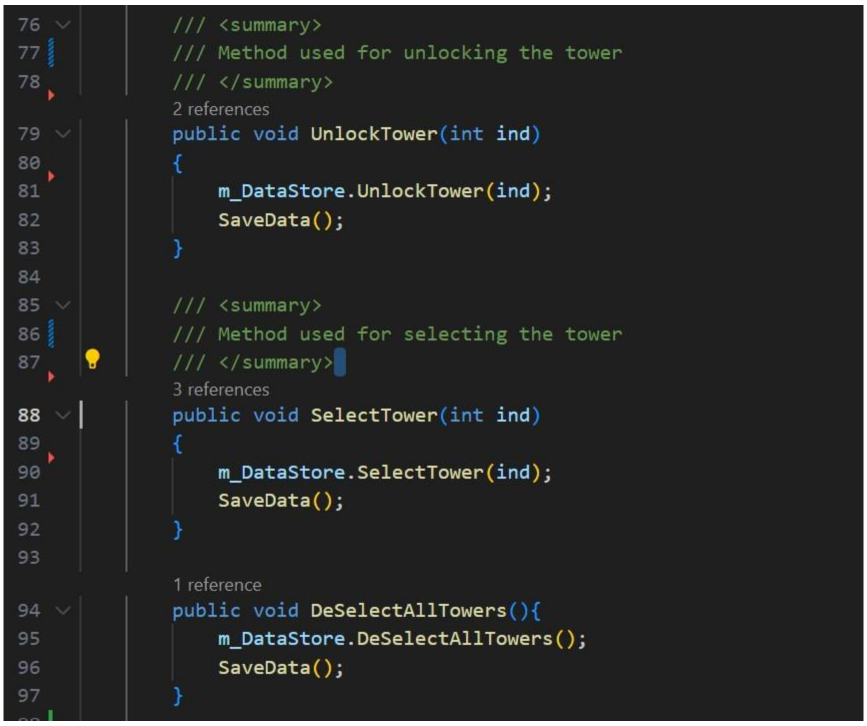
```csharp
    public class GameManager : GameManagerBase<GameManager, GameDataStore>
        protected override void Awake(){
            Screen.sleepTimeout = SleepTimeout.NeverSleep;
            base.Awake();

            int i;
            for ( i = 0; i < towerlist.Count; i++){
                Debug.Log("is Unlocked "+ IsTowerUnlocked(i));
            }

            //Ensure first 4 towers are always unlocked
            for ( i = 0; i < 4; i++){
                if (!IsTowerUnlocked(i)){
                    UnlockTower(i);
                    SelectTower(i);
                }
            }

            if (LevelManager.instance){

            LevelManager.instance.towerLibrary.Clear();

            for (i = 0; i < towerlist.Count; i++){
                if (IsTowerUnlocked(i) && IsTowerSelected(i)){
                    selectedTowers.Add(towerlist[i]);
                    Debug.Log($"{i} Added");
                }
             }
             Debug.Log("Tower Updated");
```

Figure 3.4: GameManager Script Part 1



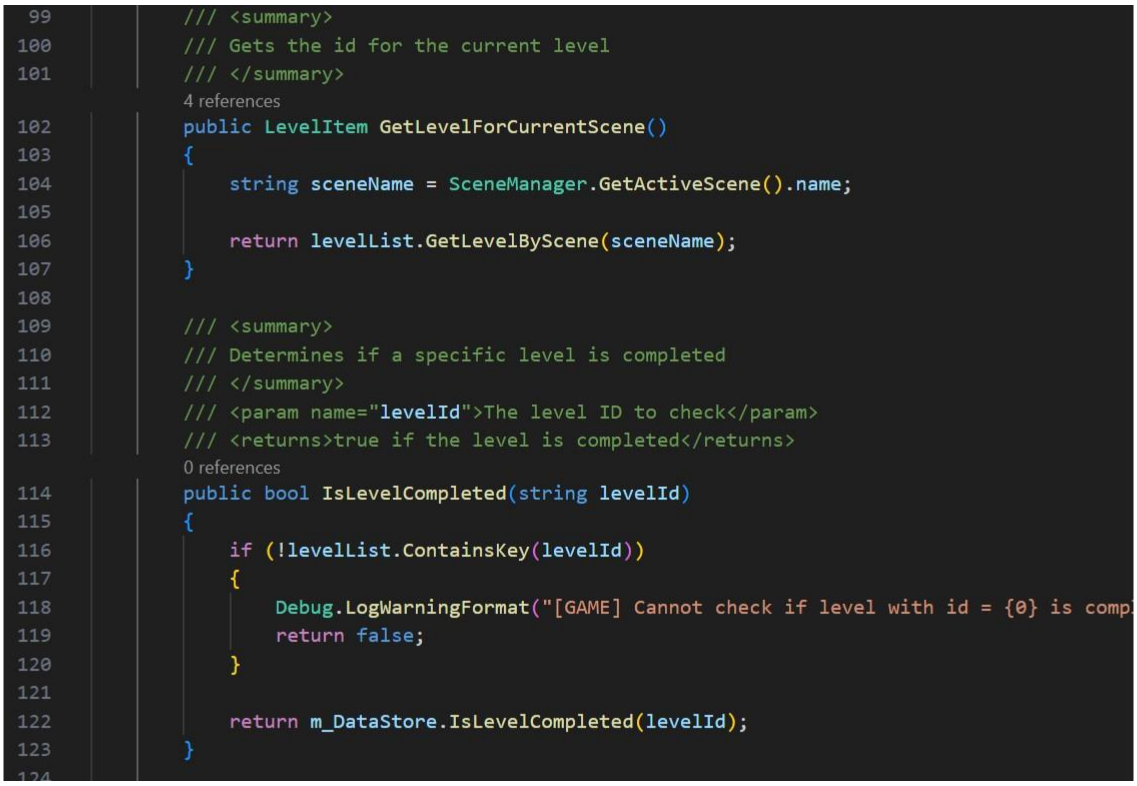
```csharp
        /// <summary>
        /// Method used for completing the level
        /// </summary>
        /// <param name="levelId">The levelId to mark as complete</param>
        /// <param name="starsEarned"></param>
        1 reference
        public void CompleteLevel(string levelId, int starsEarned)
        {
            if (!levelList.ContainsKey(levelId))
            {
                Debug.LogWarningFormat("[GAME] Cannot complete level with id = {0}. Not in
                return;
            }

            m_DataStore.CompleteLevel(levelId, starsEarned);
            SaveData();
        }
```

Figure 3.5: GameManager Script Part 2



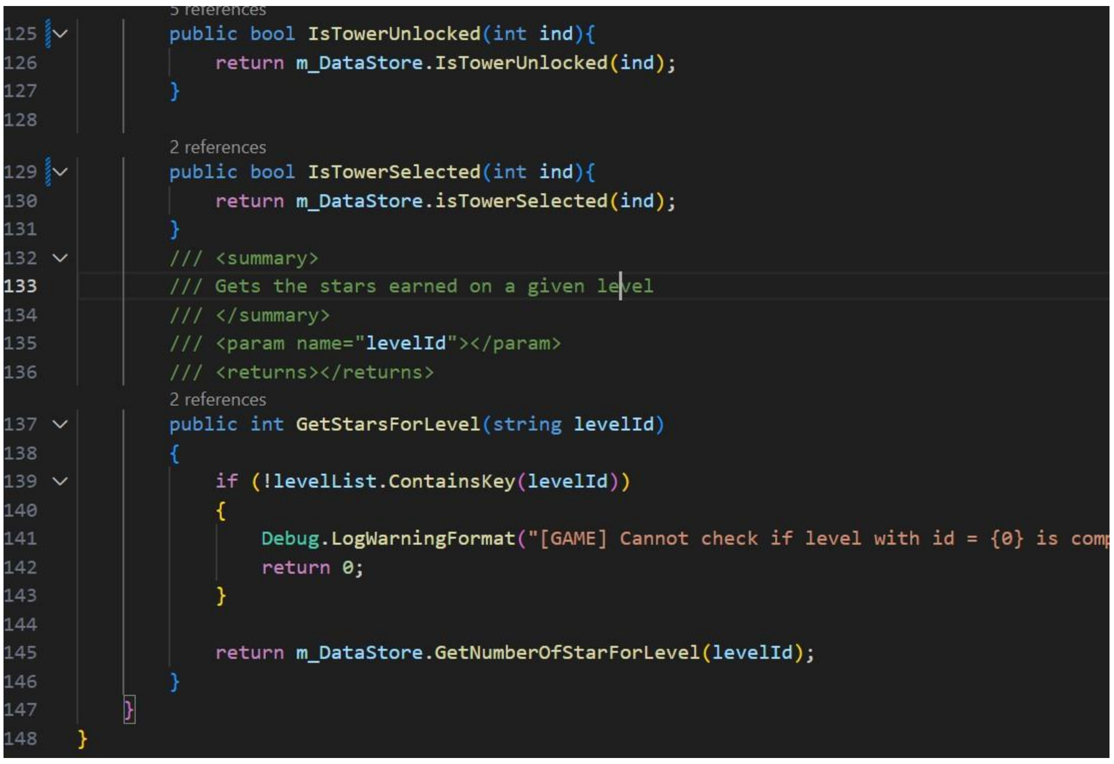
```
/// <summary>
/// Method used for unlocking the tower
/// </summary>
2 references
public void UnlockTower(int ind)
{
    m_DataStore.UnlockTower(ind);
    SaveData();
}

/// <summary>
/// Method used for selecting the tower
/// </summary>
3 references
public void SelectTower(int ind)
{
    m_DataStore.SelectTower(ind);
    SaveData();
}

1 reference
public void DeSelectAllTowers(){
    m_DataStore.DeSelectAllTowers();
    SaveData();
}
```

Figure 3.6: GameManager Script Part 3



```
/// <summary>
/// Gets the id for the current level
/// </summary>
4 references
public LevelItem GetLevelForCurrentScene()
{
    string sceneName = SceneManager.GetActiveScene().name;

    return levelList.GetLevelByScene(sceneName);
}

/// <summary>
/// Determines if a specific level is completed
/// </summary>
/// <param name="levelId">The level ID to check</param>
/// <returns>true if the level is completed</returns>
0 references
public bool IsLevelCompleted(string levelId)
{
    if (!levelList.ContainsKey(levelId))
    {
        Debug.LogWarningFormat("[GAME] Cannot check if level with id = {0} is comp
        return false;
    }

    return m_DataStore.IsLevelCompleted(levelId);
}
```

Figure 3.7: GameManager Script Part 4

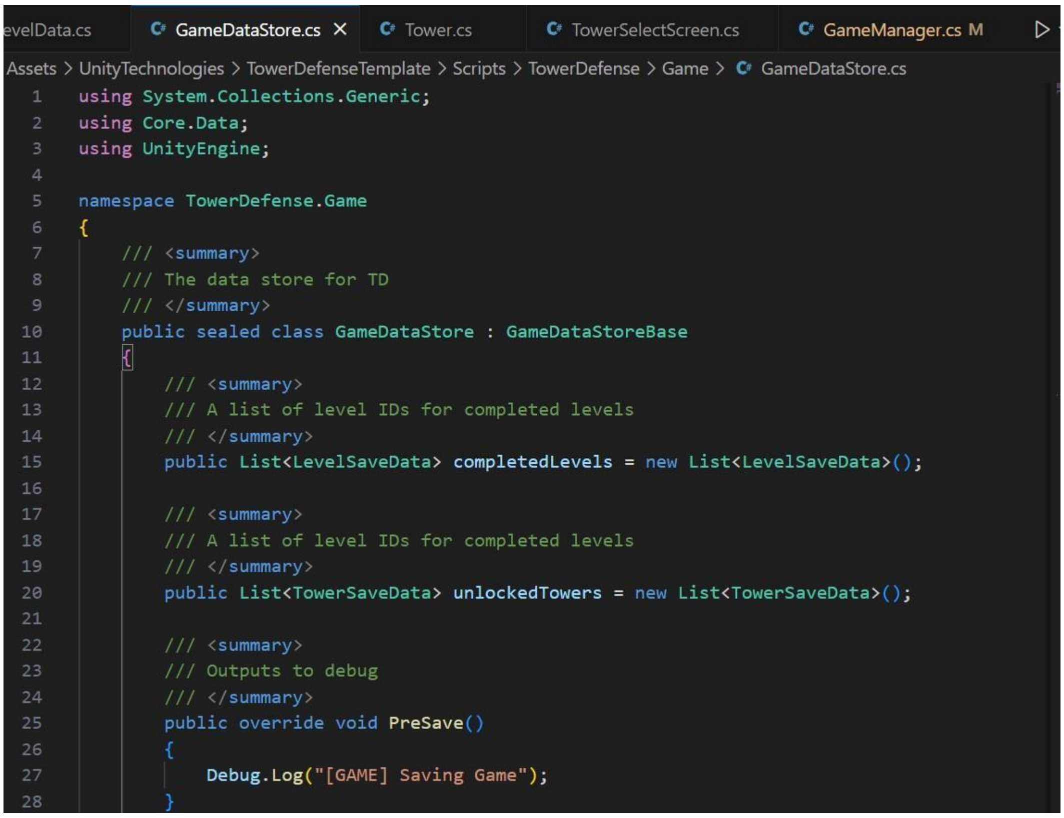


```
        5 references
        public bool IsTowerUnlocked(int ind){
            return m_DataStore.IsTowerUnlocked(ind);
        }

        2 references
        public bool IsTowerSelected(int ind){
            return m_DataStore.isTowerSelected(ind);
        }
        /// <summary>
        /// Gets the stars earned on a given level
        /// </summary>
        /// <param name="levelId"></param>
        /// <returns></returns>
        2 references
        public int GetStarsForLevel(string levelId)
        {
            if (!levelList.ContainsKey(levelId))
            {
                Debug.LogWarningFormat("[GAME] Cannot check if level with id = {0} is comp
                return 0;
            }

            return m_DataStore.GetNumberOfStarForLevel(levelId);
        }
    }
}
```

Figure 3.8: GameManager Script Part 5

The script in figure [3.4,3.8] implements the GameManager for whole game and every level. This class contain code wrappers for implementing the core methods for towers and levels unlocking and selecting logic.

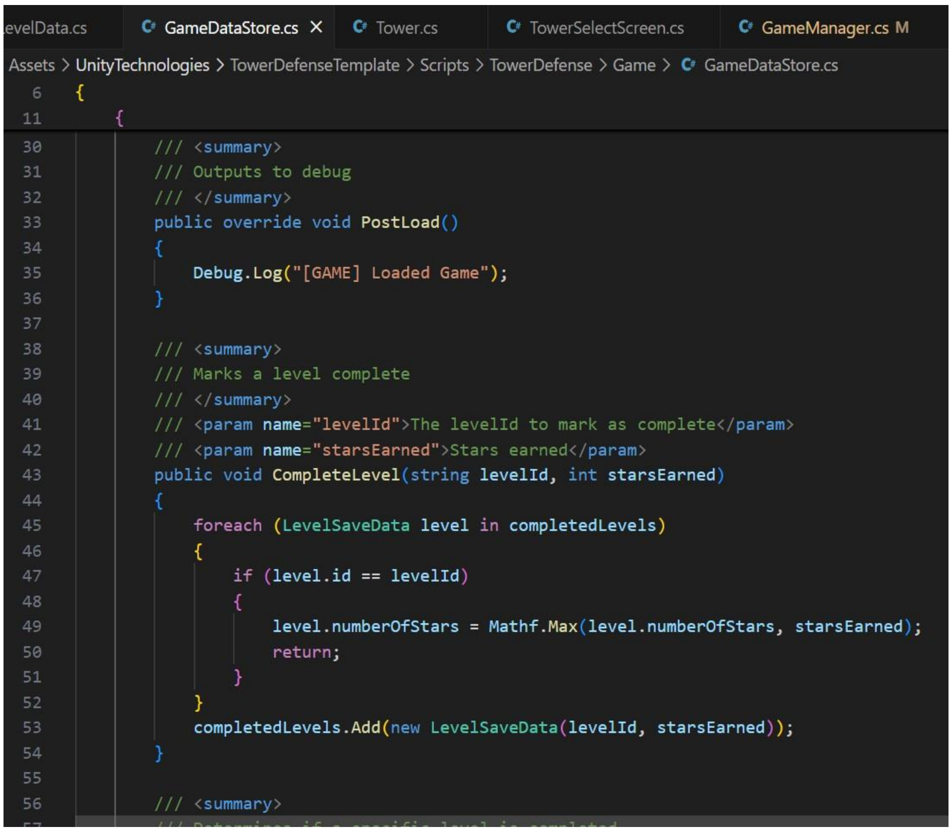
3.4 GameDataStore.cs



evelData.cs
GameDataStore.cs
Tower.cs
TowerSelectScreen.cs
GameManager.cs M
Assets > UnityTechnologies > TowerDefenseTemplate > Scripts > TowerDefense > Game > GameDataStore.cs

```csharp
using System.Collections.Generic;
using Core.Data;
using UnityEngine;

namespace TowerDefense.Game
{
    /// <summary>
    /// The data store for TD
    /// </summary>
    public sealed class GameDataStore : GameDataStoreBase
    {
        /// <summary>
        /// A list of level IDs for completed levels
        /// </summary>
        public List<LevelSaveData> completedLevels = new List<LevelSaveData>();

        /// <summary>
        /// A list of level IDs for completed levels
        /// </summary>
        public List<TowerSaveData> unlockedTowers = new List<TowerSaveData>();

        /// <summary>
        /// Outputs to debug
        /// </summary>
        public override void PreSave()
        {
            Debug.Log("[GAME] Saving Game");
        }
```

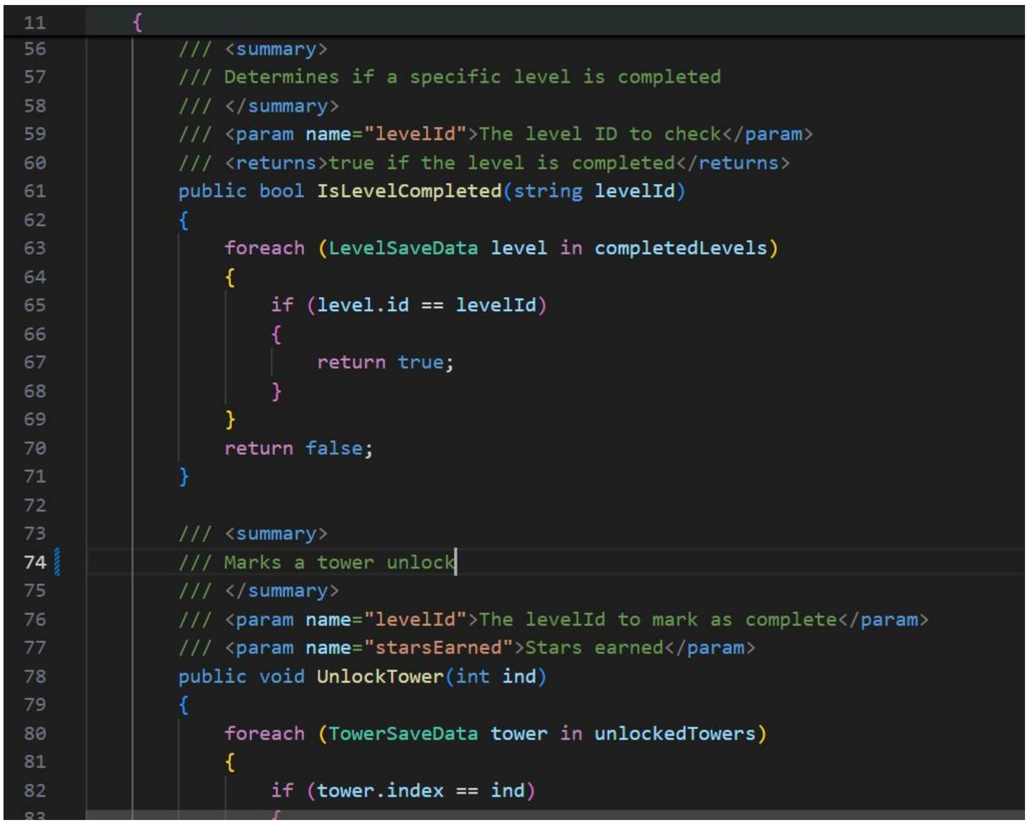
Figure 3.9: GameDataStore Script Part 1



evelData.cs
GameDataStore.cs
Tower.cs
TowerSelectScreen.cs
GameManager.cs M
Assets > UnityTechnologies > TowerDefenseTemplate > Scripts > TowerDefense > Game > GameDataStore.cs

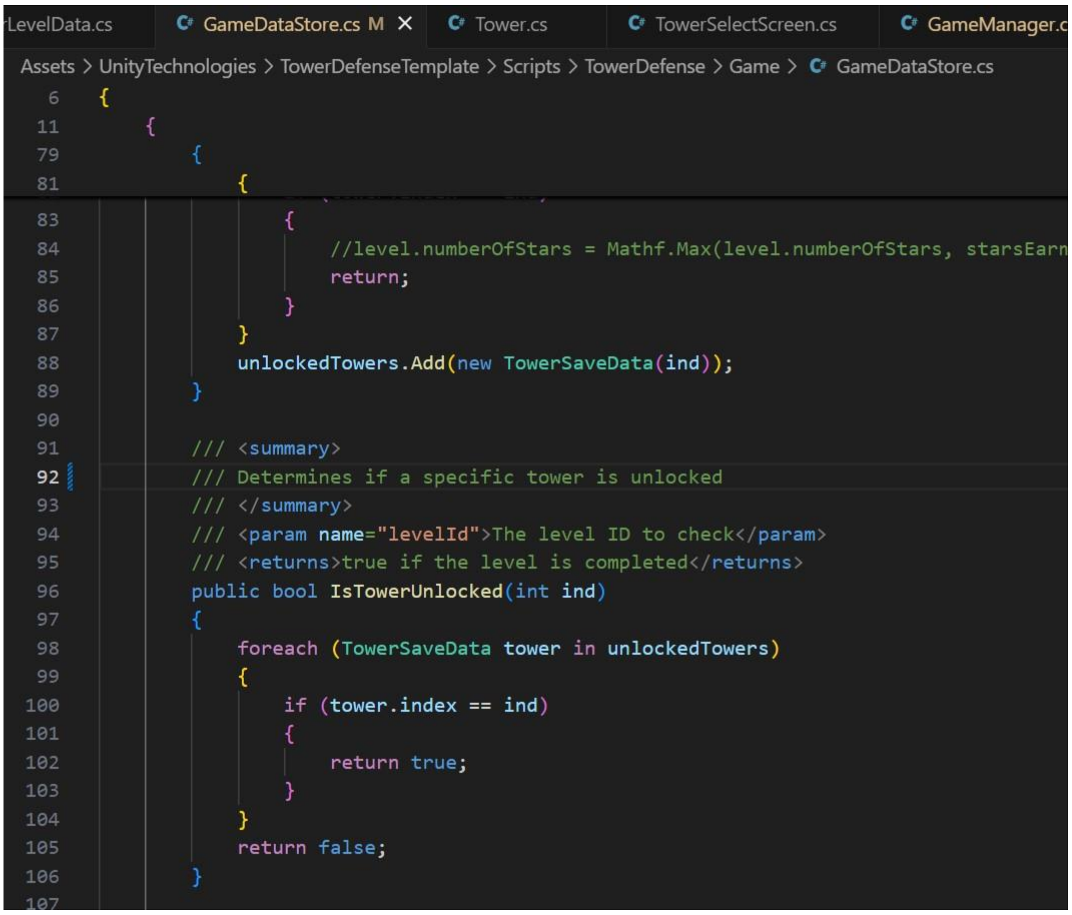
```csharp
{
    {
        /// <summary>
        /// Outputs to debug
        /// </summary>
        public override void PostLoad()
        {
            Debug.Log("[GAME] Loaded Game");
        }

        /// <summary>
        /// Marks a level complete
        /// </summary>
        /// <param name="levelId">The levelId to mark as complete</param>
        /// <param name="starsEarned">Stars earned</param>
        public void CompleteLevel(string levelId, int starsEarned)
        {
            foreach (LevelSaveData level in completedLevels)
            {
                if (level.id == levelId)
                {
                    level.numberOfStars = Mathf.Max(level.numberOfStars, starsEarned);
                    return;
                }
            }
            completedLevels.Add(new LevelSaveData(levelId, starsEarned));
        }

        /// <summary>
```

Figure 3.10: GameDataStore Script Part 2



```
    {
        /// <summary>
        /// Determines if a specific level is completed
        /// </summary>
        /// <param name="levelId">The level ID to check</param>
        /// <returns>true if the level is completed</returns>
        public bool IsLevelCompleted(string levelId)
        {
            foreach (LevelSaveData level in completedLevels)
            {
                if (level.id == levelId)
                {
                    return true;
                }
            }
            return false;
        }

        /// <summary>
        /// Marks a tower unlock
        /// </summary>
        /// <param name="levelId">The levelId to mark as complete</param>
        /// <param name="starsEarned">Stars earned</param>
        public void UnlockTower(int ind)
        {
            foreach (TowerSaveData tower in unlockedTowers)
            {
                if (tower.index == ind)
```

Figure 3.11: GameDataStore Script Part 3

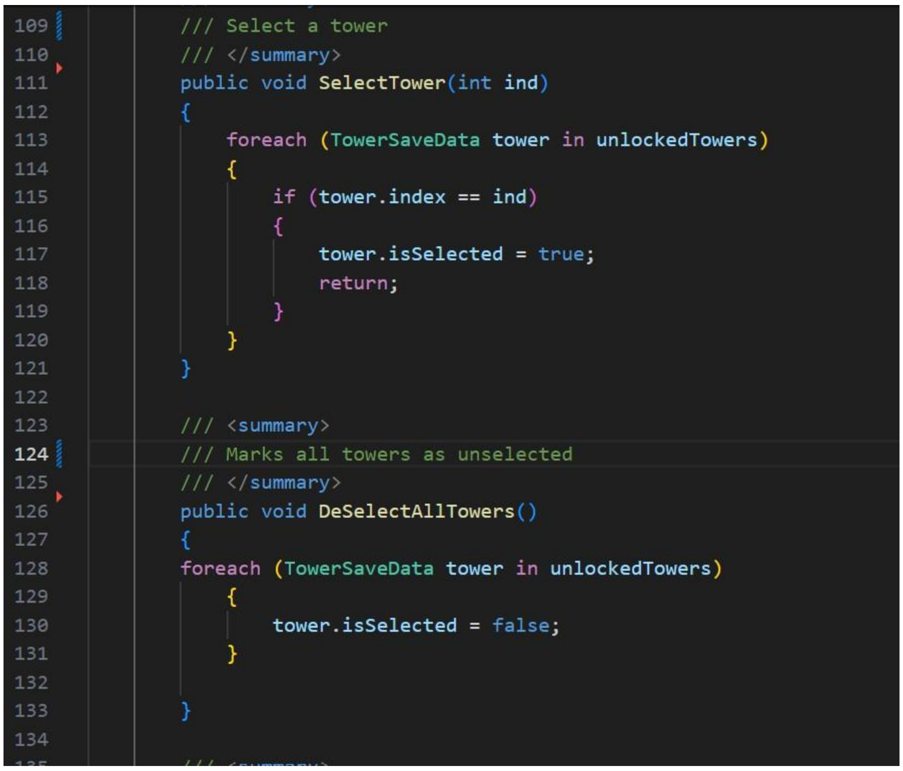


LevelData.cs | GameDataStore.cs M | Tower.cs | TowerSelectScreen.cs | GameManager.c

Assets > UnityTechnologies > TowerDefenseTemplate > Scripts > TowerDefense > Game > GameDataStore.cs

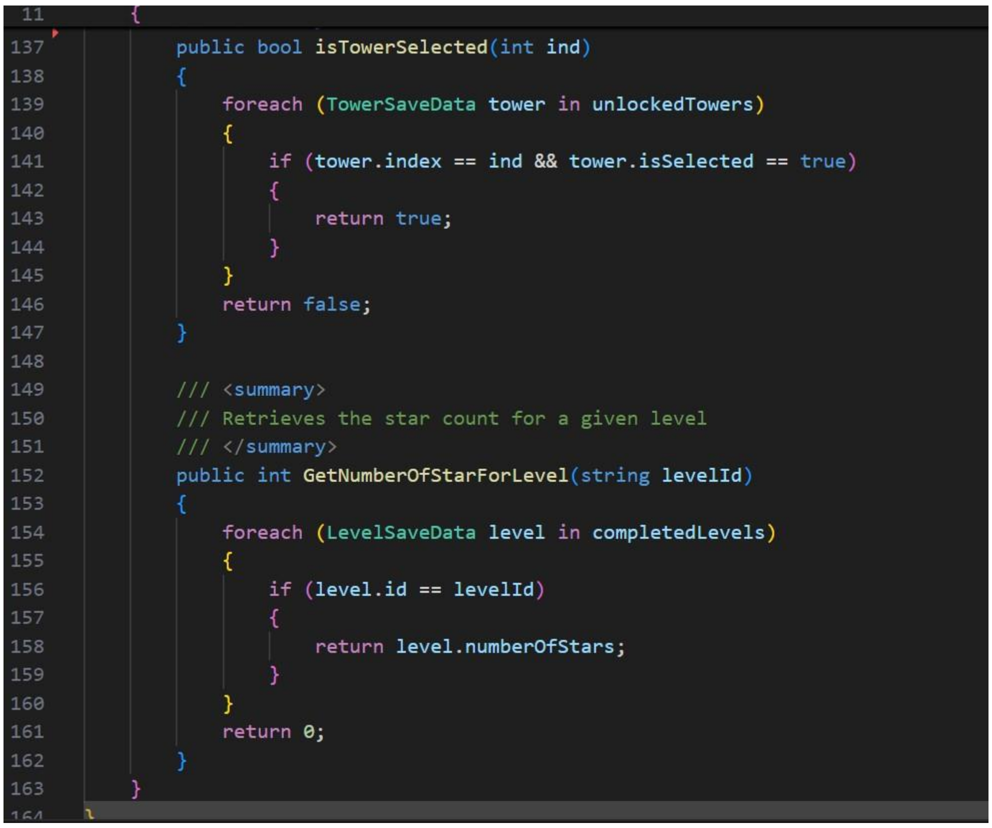
```
{
    {
        {
            {
                {
                    //level.numberOfStars = Mathf.Max(level.numberOfStars, starsEarn
                    return;
                }
            }
            unlockedTowers.Add(new TowerSaveData(ind));
        }

        /// <summary>
        /// Determines if a specific tower is unlocked
        /// </summary>
        /// <param name="levelId">The level ID to check</param>
        /// <returns>true if the level is completed</returns>
        public bool IsTowerUnlocked(int ind)
        {
            foreach (TowerSaveData tower in unlockedTowers)
            {
                if (tower.index == ind)
                {
                    return true;
                }
            }
            return false;
        }
```

Figure 3.12: GameDataStore Script Part 4



```
        /// Select a tower
        /// </summary>
        public void SelectTower(int ind)
        {
            foreach (TowerSaveData tower in unlockedTowers)
            {
                if (tower.index == ind)
                {
                    tower.isSelected = true;
                    return;
                }
            }
        }

        /// <summary>
        /// Marks all towers as unselected
        /// </summary>
        public void DeSelectAllTowers()
        {
        foreach (TowerSaveData tower in unlockedTowers)
            {
                tower.isSelected = false;
            }

        }

```

Figure 3.13: GameDataStore Script Part 5

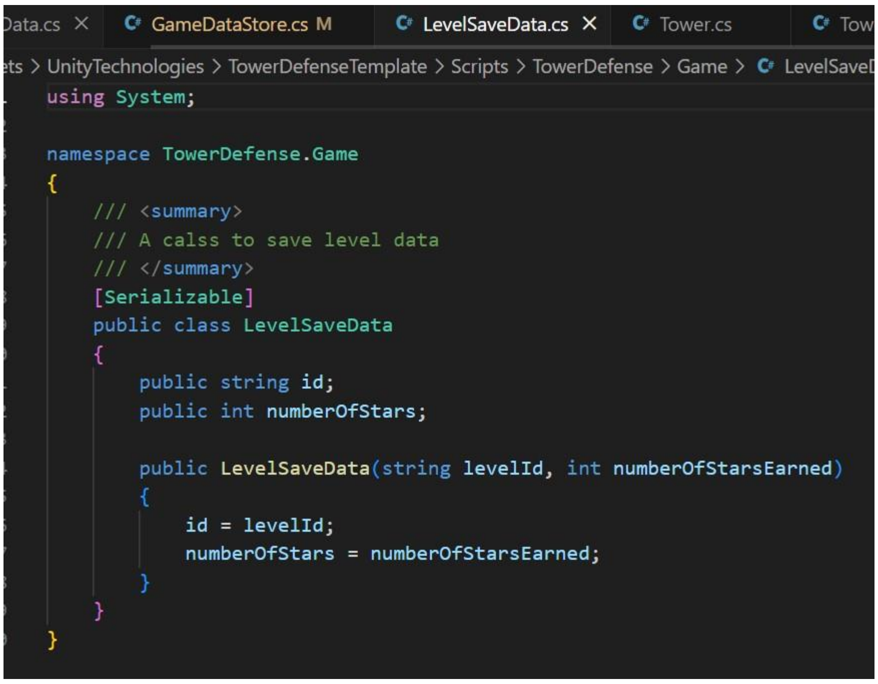


```
    {
        public bool isTowerSelected(int ind)
        {
            foreach (TowerSaveData tower in unlockedTowers)
            {
                if (tower.index == ind && tower.isSelected == true)
                {
                    return true;
                }
            }
            return false;
        }

        /// <summary>
        /// Retrieves the star count for a given level
        /// </summary>
        public int GetNumberOfStarForLevel(string levelId)
        {
            foreach (LevelSaveData level in completedLevels)
            {
                if (level.id == levelId)
                {
                    return level.numberOfStars;
                }
            }
            return 0;
        }
    }
}
```

Figure 3.14: GameDataStore Script Part 6

The script in figure [3.9 - 3.14] is a Data Storage Container and contains the implementation of the levels and towers unlocking and selecting. The data is saved in a file in persistent path of the device.

3.5 LevelSaveData.cs

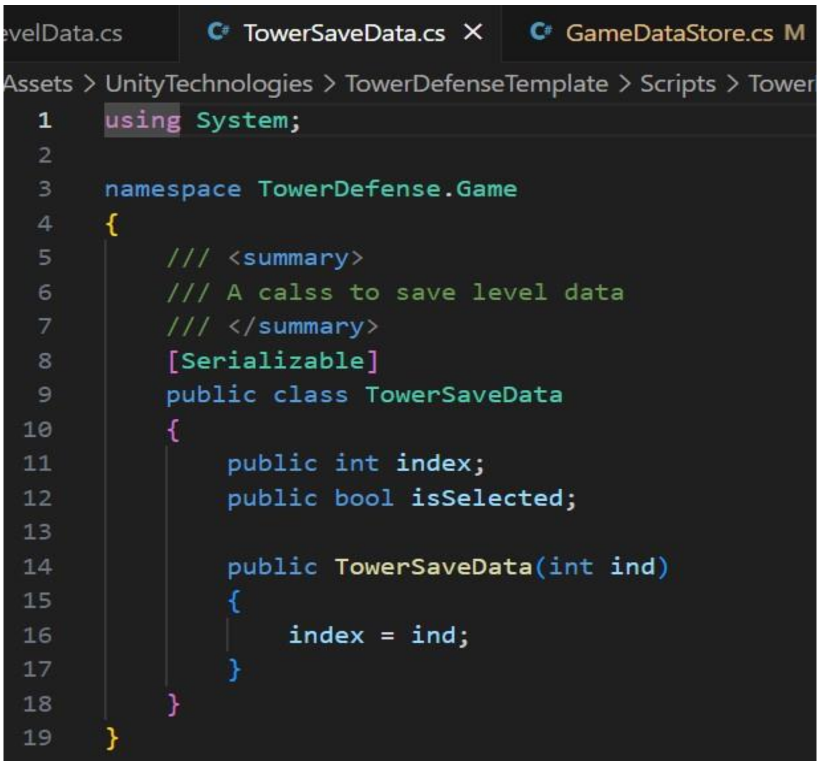

Data.cs
GameDataStore.cs M
LevelSaveData.cs
Tower.cs
Tow
ts > UnityTechnologies > TowerDefenseTemplate > Scripts > TowerDefense > Game > LevelSave

```
using System;

namespace TowerDefense.Game
{
    /// <summary>
    /// A calss to save level data
    /// </summary>
    [Serializable]
    public class LevelSaveData
    {
        public string id;
        public int numberOfStars;

        public LevelSaveData(string levelId, int numberOfStarsEarned)
        {
            id = levelId;
            numberOfStars = numberOfStarsEarned;
        }
    }
}
```

Figure 3.15: LevelSaveData Script

The script in figure [3.15] is a serialized class for saving level data.

3.6 TowerSaveData.cs



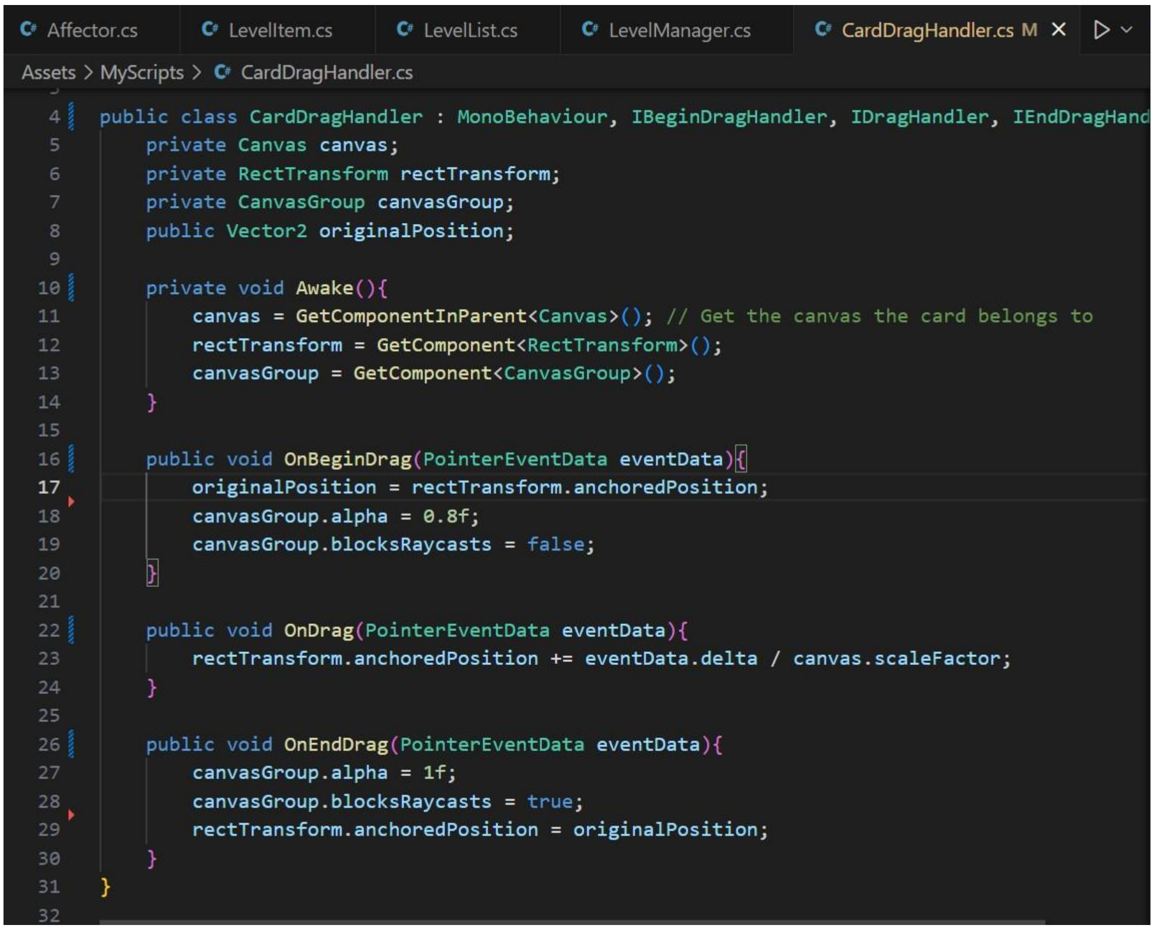
evelData.cs
TowerSaveData.cs
GameDataStore.cs M
Assets > UnityTechnologies > TowerDefenseTemplate > Scripts > Tower

```csharp
using System;

namespace TowerDefense.Game
{
    /// <summary>
    /// A calss to save level data
    /// </summary>
    [Serializable]
    public class TowerSaveData
    {
        public int index;
        public bool isSelected;

        public TowerSaveData(int ind)
        {
            index = ind;
        }
    }
}
```

Figure 3.16: TowerSaveData Script

The script in figure [3.16] is a serialized class for saving tower data.

3.7 CardDragHandler.cs

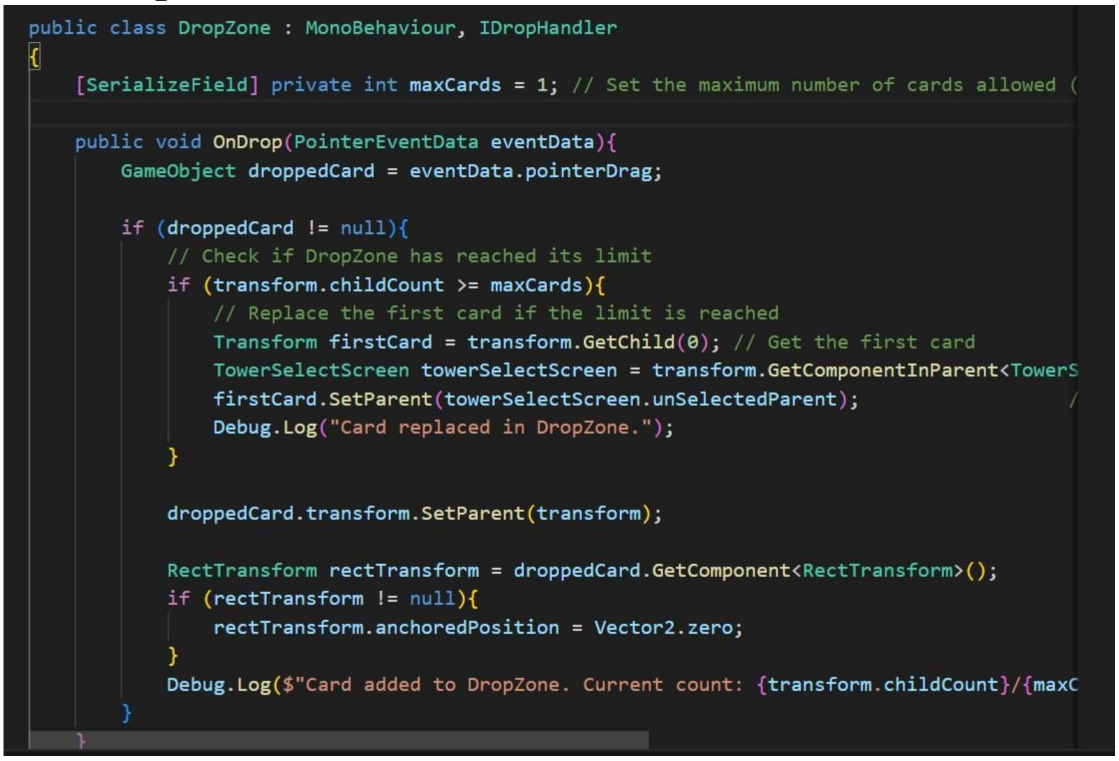


```
public class CardDragHandler : MonoBehaviour, IBeginDragHandler, IDragHandler, IEndDragHand
    private Canvas canvas;
    private RectTransform rectTransform;
    private CanvasGroup canvasGroup;
    public Vector2 originalPosition;

    private void Awake(){
        canvas = GetComponentInParent<Canvas>(); // Get the canvas the card belongs to
        rectTransform = GetComponent<RectTransform>();
        canvasGroup = GetComponent<CanvasGroup>();
    }

    public void OnBeginDrag(PointerEventData eventData){
        originalPosition = rectTransform.anchoredPosition;
        canvasGroup.alpha = 0.8f;
        canvasGroup.blocksRaycasts = false;
    }

    public void OnDrag(PointerEventData eventData){
        rectTransform.anchoredPosition += eventData.delta / canvas.scaleFactor;
    }

    public void OnEndDrag(PointerEventData eventData){
        canvasGroup.alpha = 1f;
        canvasGroup.blocksRaycasts = true;
        rectTransform.anchoredPosition = originalPosition;
    }
}
```

Figure 3.18: CardDragHandler Script

The script in figure [3.18] contains the implementation of unity built-in methods for drag and drop.

3.8 DropZone.cs

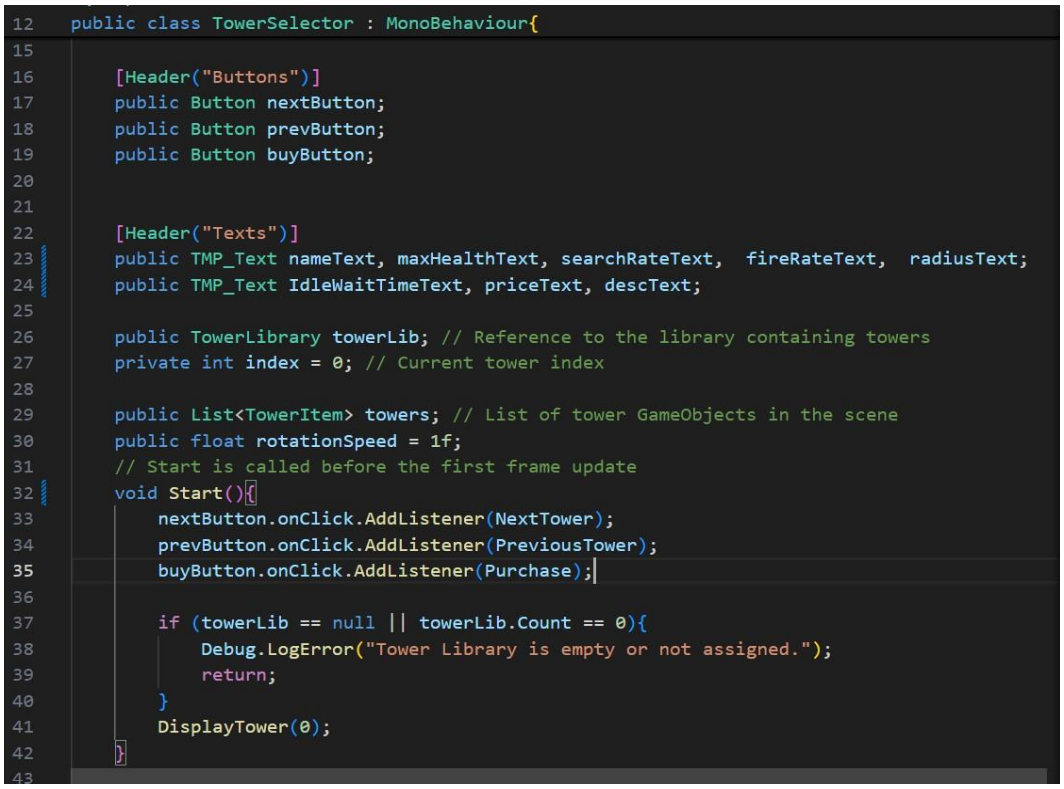


```
public class DropZone : MonoBehaviour, IDropHandler
{
    [SerializeField] private int maxCards = 1; // Set the maximum number of cards allowed (

    public void OnDrop(PointerEventData eventData){
        GameObject droppedCard = eventData.pointerDrag;

        if (droppedCard != null){
            // Check if DropZone has reached its limit
            if (transform.childCount >= maxCards){
                // Replace the first card if the limit is reached
                Transform firstCard = transform.GetChild(0); // Get the first card
                TowerSelectScreen towerSelectScreen = transform.GetComponentInParent<TowerS
                firstCard.SetParent(towerSelectScreen.unSelectedParent);                   /
                Debug.Log("Card replaced in DropZone.");
            }

            droppedCard.transform.SetParent(transform);

            RectTransform rectTransform = droppedCard.GetComponent<RectTransform>();
            if (rectTransform != null){
                rectTransform.anchoredPosition = Vector2.zero;
            }
            Debug.Log($"Card added to DropZone. Current count: {transform.childCount}/{maxC
        }
    }
```

Figure 3.19: DropZone Script

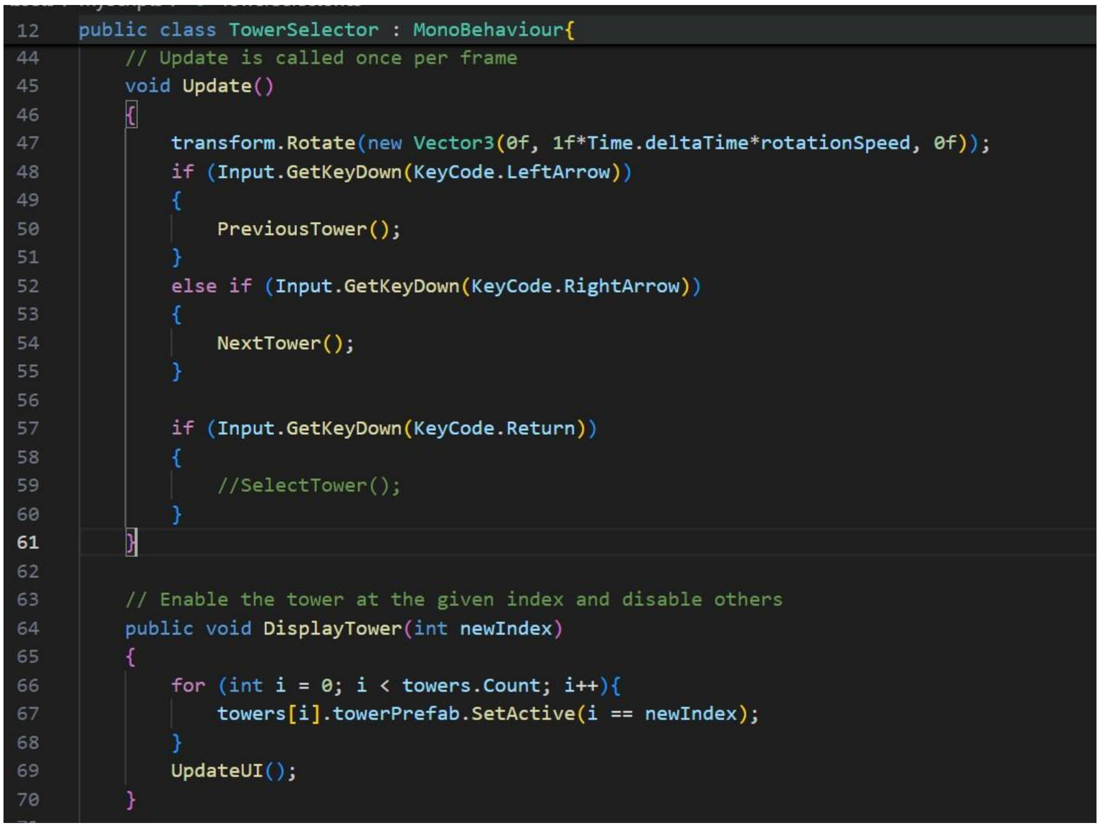
The script in figure [3.19] creates a drop zone for the draggable object.

3.9 TowerSelector.cs



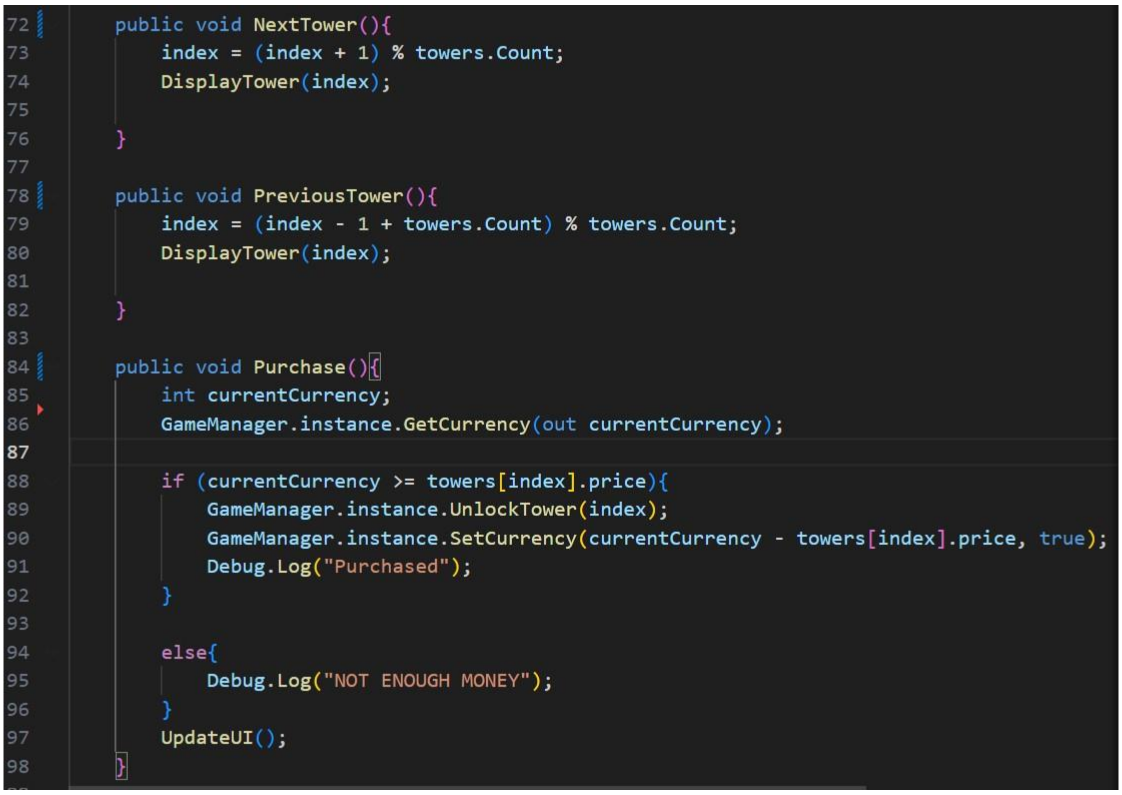
```
public class TowerSelector : MonoBehaviour{

    [Header("Buttons")]
    public Button nextButton;
    public Button prevButton;
    public Button buyButton;


    [Header("Texts")]
    public TMP_Text nameText, maxHealthText, searchRateText,  fireRateText,  radiusText;
    public TMP_Text IdleWaitTimeText, priceText, descText;

    public TowerLibrary towerLib; // Reference to the library containing towers
    private int index = 0; // Current tower index

    public List<TowerItem> towers; // List of tower GameObjects in the scene
    public float rotationSpeed = 1f;
    // Start is called before the first frame update
    void Start(){
        nextButton.onClick.AddListener(NextTower);
        prevButton.onClick.AddListener(PreviousTower);
        buyButton.onClick.AddListener(Purchase);

        if (towerLib == null || towerLib.Count == 0){
            Debug.LogError("Tower Library is empty or not assigned.");
            return;
        }
        DisplayTower(0);
    }
```

Figure 3.20: TowerSelector Script Part 1



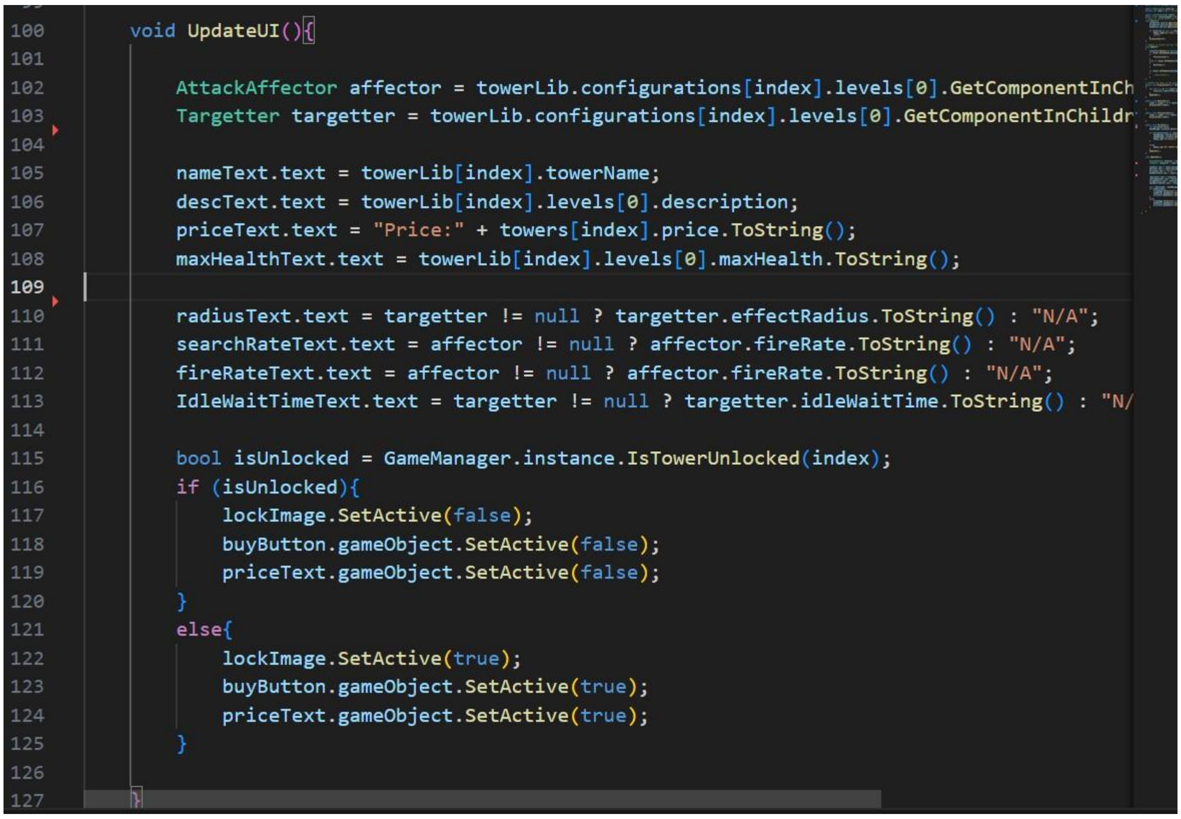
```
public class TowerSelector : MonoBehaviour{
    // Update is called once per frame
    void Update()
    {
        transform.Rotate(new Vector3(0f, 1f*Time.deltaTime*rotationSpeed, 0f));
        if (Input.GetKeyDown(KeyCode.LeftArrow))
        {
            PreviousTower();
        }
        else if (Input.GetKeyDown(KeyCode.RightArrow))
        {
            NextTower();
        }

        if (Input.GetKeyDown(KeyCode.Return))
        {
            //SelectTower();
        }
    }

    // Enable the tower at the given index and disable others
    public void DisplayTower(int newIndex)
    {
        for (int i = 0; i < towers.Count; i++){
            towers[i].towerPrefab.SetActive(i == newIndex);
        }
        UpdateUI();
    }
```

Figure 3.21: TowerSelector Script Part 2



```
    public void NextTower(){
        index = (index + 1) % towers.Count;
        DisplayTower(index);

    }

    public void PreviousTower(){
        index = (index - 1 + towers.Count) % towers.Count;
        DisplayTower(index);

    }

    public void Purchase(){
        int currentCurrency;
        GameManager.instance.GetCurrency(out currentCurrency);

        if (currentCurrency >= towers[index].price){
            GameManager.instance.UnlockTower(index);
            GameManager.instance.SetCurrency(currentCurrency - towers[index].price, true);
            Debug.Log("Purchased");
        }

        else{
            Debug.Log("NOT ENOUGH MONEY");
        }
        UpdateUI();
    }
```

Figure 3.22: TowerSelector Script Part 3



```
    void UpdateUI(){

        AttackAffector affector = towerLib.configurations[index].levels[0].GetComponentInCh
        Targetter targetter = towerLib.configurations[index].levels[0].GetComponentInChildr

        nameText.text = towerLib[index].towerName;
        descText.text = towerLib[index].levels[0].description;
        priceText.text = "Price:" + towers[index].price.ToString();
        maxHealthText.text = towerLib[index].levels[0].maxHealth.ToString();

        radiusText.text = targetter != null ? targetter.effectRadius.ToString() : "N/A";
        searchRateText.text = affector != null ? affector.fireRate.ToString() : "N/A";
        fireRateText.text = affector != null ? affector.fireRate.ToString() : "N/A";
        IdleWaitTimeText.text = targetter != null ? targetter.idleWaitTime.ToString() : "N/

        bool isUnlocked = GameManager.instance.IsTowerUnlocked(index);
        if (isUnlocked){
            lockImage.SetActive(false);
            buyButton.gameObject.SetActive(false);
            priceText.gameObject.SetActive(false);
        }
        else{
            lockImage.SetActive(true);
            buyButton.gameObject.SetActive(true);
            priceText.gameObject.SetActive(true);
        }

    }
```

Figure 3.23: TowerSelector Script Part 4

The script in figure [3.20 – 3.23] implements the tower shop. It contain methods for selecting and buying different towers.