```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using static PlayerTurn;

[System.Serializable]
#region PlayerStats
public class PlayerStats
{
    // Propriedades públicas
    public Dictionary<SkillType, int> SkillImprovements { get; private
set; }
    public int Health { get; set; }
    public int MaxHealth { get; set; }
    public int Sanity { get; set; }
    public int MaxSanity { get; set; }
    public int Lore { get; set; }
    public int Influence { get; set; }
    public int Observation { get; set; }
    public int Strength { get; set; }
    public int Will { get; set; }
    public int Movement { get; set; }
    public int CombatTestModifier { get; set; }
    public int InfluenceTestModifier { get; set; }
    public int LoreTestModifier { get; set; }
    public int ObservationTestModifier { get; set; }
    public int StrengthTestModifier { get; set; }
    public int WillTestModifier { get; set; }

    // Construtor
    public PlayerStats()
    {
        SkillImprovements = new Dictionary<SkillType, int>
        {
            { SkillType.Lore, 0 },
            { SkillType.Influence, 0 },
            { SkillType.Observation, 0 },
            { SkillType.Strength, 0 },
            { SkillType.Will, 0 },
        };
    }
```

```
    // Melhora uma habilidade específica
    public void ImproveSkill(SkillType skillType)
    {
        if (SkillImprovements.ContainsKey(skillType))
        {
            int currentImprovement = SkillImprovements[skillType];
            if (currentImprovement < 2)
            {
                SkillImprovements[skillType]++;
                Debug.Log("Skill " + skillType.ToString() + " improved
to: " + (currentImprovement + 1));
            }
            else
            {
                Debug.Log("Skill " + skillType.ToString() + " already
at maximum improvement.");
            }
        }
        else
        {
            Debug.LogError("Invalid skill type: " +
skillType.ToString());
        }
    }

    // Diminui uma habilidade específica
    public void DiminishSkill(SkillType skillType)
    {
        if (SkillImprovements.ContainsKey(skillType))
        {
            int currentImprovement = SkillImprovements[skillType];
            if (currentImprovement > 0)
            {
                SkillImprovements[skillType]--;
                Debug.Log("Skill " + skillType.ToString() + "
diminished to: " + (currentImprovement - 1));
            }
            else
            {
                Debug.Log("Skill " + skillType.ToString() + " already
at minimum improvement.");
            }
        }
```

```
        else
        {
            Debug.LogError("Invalid skill type: " +
skillType.ToString());
        }
    }

    // Obtém o modificador de teste de uma habilidade específica
    public int GetSkillTestModifier(SkillType skillType)
    {
        switch (skillType)
        {
            case SkillType.Combat:
                return CombatTestModifier;
            case SkillType.Influence:
                return InfluenceTestModifier;
            case SkillType.Lore:
                return LoreTestModifier;
            case SkillType.Observation:
                return ObservationTestModifier;
            case SkillType.Strength:
                return StrengthTestModifier;
            case SkillType.Will:
                return WillTestModifier;
            default:
                Debug.LogError("Invalid skill type: " +
skillType.ToString());
                return 0;
        }
    }

    // Obtém o modificador de melhoria de habilidade de uma habilidade
específica
    public int GetSkillImprovementModifier(SkillType skillType)
    {
        if (SkillImprovements.ContainsKey(skillType))
        {
            return SkillImprovements[skillType];
        }
        else
        {
            Debug.LogError("Invalid skill type: " +
skillType.ToString());
```

```
            return 0;
        }
    }
}
#endregion
public class Player
{
    public List<Token> TokenPool { get; set; }

    public string InvestigatorName { get;  set; }
    public string Occupation { get; set; }
    public string StartingLocation { get; set; }
    public InvestigatorData investigator { get; set; }

     public PlayerStats stats { get; set; }
    public List<Card> Inventory { get; set; }
    public List<Asset> InventoryAsset { get; set; }
    public Player()
    {
        Inventory = new List<Card>();
        InventoryAsset = new List<Asset>();
        TokenPool = new List<Token>();

    }

    public bool DiscardSpecificAssetCard(Asset card)
    {
        // Itera sobre o invent◆rio do jogador
        for (int i = 0; i < InventoryAsset.Count; i++)
        {
            // Se o nome da carta corresponder ao nome da carta
especificada
            if (InventoryAsset[i] == card)
            {
                // Remove a carta do invent◆rio
                Inventory.RemoveAt(i);
                return true; // Retorna verdadeiro para indicar que a
carta foi descartada com sucesso
            }
        }

        // Se nenhuma carta corresponder ao nome da carta especificada,
retorna falso
```

```
        return false;
    }
    public Card DiscardRandomCard()
    {
        // Cria uma inst◆ncia da classe Random
        System.Random random = new System.Random();

        // Gera um ◆ndice aleat◆rio
        int randomIndex = random.Next(Inventory.Count);

        // Armazena a carta que ser◆ descartada
        Card cardToDiscard = Inventory[randomIndex];

        // Remove a carta do invent◆rio
        Inventory.RemoveAt(randomIndex);

        // Retorna a carta que foi descartada
        return cardToDiscard;
    }
}

public class PlayerController : MonoBehaviour
{
    public PlayerActionPhase actionPhase;

    [Header("Components")]
    public GameController gameController;
    public DiceRoller diceRoller;
    public Button endTurnButton;

    [Header("Character Stats")]
    public int currentPosition = 0;
    public List<string> eventList;
    public int skillValue;
    public int testModifier;
    public int improvementModifier;
    public int impairmentModifier;
    public int bonus;
    public int additionalDice;
    public int clueTokens;
    public int focusTokens;
    public int ResourcesTokens;
    public bool isBlessed;
```

```
    public bool isCursed;
    public int movPoints = 1;
    public CombatEncounter currentCombatEncounter;

    [Header("Investigator Data")]
    public Investigator choosed;
    public InvestigatorData currentInvestigator;
    public InvestigatorData investigatorData;

    [Header("Buttons")]
    Button restButton;
    Button confirmButtonRest;
    Button cancelButtonRest;
    bool restActionPending;

    public Button travelButton;
    public Button tradeButton;
    public Button prepareButton;
    public Button acquireButton;
    public Button componentButton;
    public Button focusButton;
    public Button localButton;
    public Button gatherButton;

    public PlayerStats playerStats;

    public Player player;
    private Vector2 targetPosition;
    public Tile currentTile;
    public TokenManagement tokenManagement;
    [SerializeField] private PlayerActionPhase _playerAction;
    [SerializeField] private Button movementButton;
    public bool _movementAction;

    void Awake()
    {
        Debug.Log("Iniciando o m◆todo Awake...");
        player = new Player();

        movementButton =
GameObject.FindGameObjectWithTag("movementButton").GetComponent<Button>
();
        gameController = FindObjectOfType<GameController>();
```

```
        _playerAction = FindObjectOfType<PlayerActionPhase>();
        diceRoller = gameController.GetComponent<DiceRoller>();
        gameController.prayer = player;

#region Buttons

        restButton =
GameObject.FindGameObjectWithTag("restButton").GetComponent<Button>();

        // Obtem os componentes Button de todos os objetos filhos do
botão de descanso
        Button[] childButtons =
restButton.GetComponentsInChildren<Button>();

        // Verifica se existem dois botões filhos (para confirmar e
cancelar)
        if (childButtons.Length >= 2)
        {
            // Atribui os botões de confirmar e cancelar.
            // Isso pressupõe que o botão de confirmação seja o
primeiro filho e o de cancelamento seja o segundo.
            confirmButtonRest = childButtons[0];
            cancelButtonRest = childButtons[1];
        }
        else
        {
            Debug.LogError("Rest button does not have enough child
buttons.");
        }

        restButton.onClick.AddListener(Rest);
        confirmButtonRest.onClick.AddListener(ConfirmRest);
        cancelButtonRest.onClick.AddListener(CancelRest);

        restActionPending = false;

        travelButton =
GameObject.FindGameObjectWithTag("movementButton").GetComponent<Button>
();
        travelButton.onClick.AddListener(ActionMovement);

        tradeButton =
GameObject.FindGameObjectWithTag("tradeButton").GetComponent<Button>();
```

```
        tradeButton.onClick.AddListener(Trade);

        prepareButton =
GameObject.FindGameObjectWithTag("prepareButton").GetComponent<Button>(
);
        prepareButton.onClick.AddListener(PrepareForTravel);

        acquireButton =
GameObject.FindGameObjectWithTag("acquireButton").GetComponent<Button>(
);
        acquireButton.onClick.AddListener(AcquireAssets);

        componentButton =
GameObject.FindGameObjectWithTag("componentButton").GetComponent<Button
>();
        componentButton.onClick.AddListener(ComponentAction);

        focusButton =
GameObject.FindGameObjectWithTag("focusButton").GetComponent<Button>();
        focusButton.onClick.AddListener(FocusAction);

        localButton =
GameObject.FindGameObjectWithTag("localButton").GetComponent<Button>();
        localButton.onClick.AddListener(LocalAction);

        gatherButton =
GameObject.FindGameObjectWithTag("gatherButton").GetComponent<Button>()
;
        gatherButton.onClick.AddListener(GatherResources);
        #endregion

        endTurnButton = gameController.endTurnButton;
        endTurnButton.onClick.AddListener(EndTurn);
        Debug.Log("Metodo Awake concluido.");
        // Add the OnClick listener
    }

    public void SetInvestigator(Investigator investigator)
    {
        playerStats = new PlayerStats();

        choosed = investigator;
        investigatorData = new InvestigatorData();
```

```
        investigatorData.SetInvestigatorData(investigator);
        playerStats = SetPlayerStats(investigatorData);
        player.stats = playerStats;
        playerStats.Movement = 1;

        movPoints = playerStats.Movement;
    }

    public int GetBaseSkillValue(SkillType skillName)
    {
        switch (skillName)
        {
            case SkillType.Lore:
                return player.stats.Lore;
            case SkillType.Influence:
                return player.stats.Influence;
            case SkillType.Observation:
                return player.stats.Observation;
            case SkillType.Strength:
                return player.stats.Strength;
            case SkillType.Will:
                return player.stats.Will;
            default:
                Debug.LogError("Invalid skill name: " + skillName);
                return -1;
        }
    }

    public PlayerStats SetPlayerStats(InvestigatorData
currentInvestigator)
    {
        Debug.Log("Definindo as estat♦sticas do jogador...");
        playerStats.Health = currentInvestigator.health;
        playerStats.MaxHealth = playerStats.Health;
        playerStats.Sanity = currentInvestigator.sanity;
        playerStats.MaxSanity = playerStats.Sanity;
        playerStats.Lore = currentInvestigator.lore;
        playerStats.Influence = currentInvestigator.influence;
        playerStats.Observation = currentInvestigator.observation;
        playerStats.Strength = currentInvestigator.strength;
        playerStats.Will = currentInvestigator.will;
        playerStats.CombatTestModifier = 0;
        playerStats.InfluenceTestModifier = 0;
```

```
        playerStats.LoreTestModifier = 0;
        playerStats.ObservationTestModifier = 0;
        playerStats.StrengthTestModifier = 0;
        playerStats.WillTestModifier = 0;
        /*
                Debug.Log("Name:" + player.InvestigatorName);
                Debug.Log("Ocuppation:" + player.Occupation);
                Debug.Log("Health: " + playerStats.Health);
                Debug.Log("Sanity: " + playerStats.Sanity);
                Debug.Log("Lore: " + playerStats.Lore);
                Debug.Log("Influence: " + playerStats.Influence);
                Debug.Log("Observation: " + playerStats.Observation);
                Debug.Log("Strength: " + playerStats.Strength);
                Debug.Log("Will: " + playerStats.Will);

                Debug.Log("Estat◆sticas do jogador definidas.");
        */
        return playerStats;
    }

    public void SetTargetPosition(Vector2 target, int tileID)
    {
        Debug.Log("Definindo a posicao alvo...");

        targetPosition = target;
        //targetTileID = tileID;

        Debug.Log("Posi◆◆o alvo definida.");
    }



    public void EndTurn()
    {
        Debug.Log("Turno encerrado.");

        endTurnButton.interactable = false;
        gameController.currentPhase = GamePhase.EncounterPhase;

            StartCoroutine(ExecuteTurn());

    }
```

```
    IEnumerator ExecuteTurn()
    {
        Debug.Log("Executando turno...");


        yield return new WaitForSeconds(0);


        endTurnButton.interactable = true;

        movPoints = 1;
        Button buttonaction =
GameObject.FindGameObjectWithTag("movementButton").GetComponent<Button>
();
        buttonaction.interactable = true;
        buttonaction.image.color = Color.white;
        CombatSystem combatSystem = CombatSystem.Instance;
        if (combatSystem != null )
        {

            combatSystem.StartCombat(currentTile.TileID);

        }
        // Agora você pode chamar métodos na instância


        Debug.Log("Move points recarregados"+movPoints);
        _playerAction.ResetActions();


        Debug.Log("Turno concluido.");
    }


    #region REST
    public void Rest()
    {
        // Verifica se o investigador está compartilhando o espaço com
um monstro
        if (CheckForMonsters())
        {
            Debug.Log("Cannot perform Rest action. There are monsters
in the same space.");
```

```
            return;
        }

        restActionPending = true; // A ação de descanso está pendente
até que seja confirmada ou cancelada
    }

    public void ConfirmRest()
    {
        if (!restActionPending) // Verifica se uma ação de descanso
está pendente
        {
            Debug.Log("No rest action to confirm.");
            return;
        }

        {
        // Verifica se o investigador está compartilhando o espaço com
um monstro
        if (CheckForMonsters())
        {
            Debug.Log("Cannot perform Rest action. There are monsters
in the same space.");
            return;
        }

        // Recupera a quantidade de saúde e sanidade a serem
recuperadas durante o descanso
        int healthToRecover = 1;
        int sanityToRecover = 1;

        // Gasta recursos para recuperar saúde e sanidade adicionais
        int additionalHealth = SpendResourcesForHealthRecovery();
        int additionalSanity = SpendResourcesForSanityRecovery();

        // Calcula a nova quantidade de saúde e sanidade após o
descanso
        int newHealth = Mathf.Min(player.stats.Health + healthToRecover
+ additionalHealth, player.stats.MaxHealth);
        int newSanity = Mathf.Min(player.stats.Sanity + sanityToRecover
+ additionalSanity, player.stats.MaxSanity);
```

```
        // Atualiza as estatísticas do jogador com a saúde e sanidade
recuperadas
        player.stats.Health = newHealth;
        player.stats.Sanity = newSanity;

        Debug.Log("Resting... Recovered Health: " + healthToRecover + "
+ " + additionalHealth + ", Recovered Sanity: " + sanityToRecover + " +
" + additionalSanity);
    }

        restActionPending = false; // A ação de descanso foi concluída
    }

    public void CancelRest()
    {
        if (!restActionPending) // Verifica se uma ação de descanso
está pendente
        {
            Debug.Log("No rest action to cancel.");
            return;
        }

        restActionPending = false; // A ação de descanso foi cancelada
        Debug.Log("Rest action cancelled.");
    }

    private bool CheckForMonsters()
    {
        Token monsterToken =
tokenManagement.FindActiveTokenOnTile(TokenType.Monster,
currentTile.TileID);
        // Se o token do monstro for diferente de null, significa que
há um monstro no tile
        if (monsterToken != null)
        {
            return true;
        }
        return false;
    }

    private int SpendResourcesForHealthRecovery()
    {
        int additionalHealth = 0;
```

```
        // Implemente a lógica para gastar recursos a fim de recuperar
saúde adicional durante o descanso
        // Exemplo de implementação:
        // additionalHealth = player.GetResourceCount() / 2;

        return additionalHealth;
    }

    private int SpendResourcesForSanityRecovery()
    {
        int additionalSanity = 0;

        // Implemente a lógica para gastar recursos a fim de recuperar
sanidade adicional durante o descanso
        // Exemplo de implementação:
        // additionalSanity = player.GetResourceCount() / 2;

        return additionalSanity;
    }

    public void ActionMovement()
    {

        _playerAction.PerformAction(Action.Travel);

        //Debug.Log(_movementAction);

    }
        public void Travel()
    {
        _movementAction = true;
        Button buttonaction =
GameObject.FindGameObjectWithTag("movementButton").GetComponent<Button>
();
        buttonaction.interactable = false;
    }
    #endregion
    public void Trade()
    {
        // Implement trade logic here
    }
```

```
    public void AcquireAssets()
    {
        // Implement acquire assets logic here
    }

    public void PrepareForTravel()
    {
        // Implement prepare for travel logic here
    }

    public void ComponentAction()
    {
        // Implement component action logic here
    }

    public void FocusAction()
    {
        // Implement focus action logic here
    }

    public void LocalAction()
    {
        // Implement local action logic here
    }

    public void GatherResources()
    {
        // Implement gather resources logic here
    }

    // ...

    public bool SkillCheck(SkillType skill, int targetValue)
    {
        int testModifier = 0;

        switch (skill)
        {
            case SkillType.Combat:
                testModifier = playerStats.CombatTestModifier;
                break;
            case SkillType.Influence:
                testModifier = playerStats.InfluenceTestModifier;
```

```
                break;
            case SkillType.Lore:
                testModifier = playerStats.LoreTestModifier;
                break;
            case SkillType.Observation:
                testModifier = playerStats.ObservationTestModifier;
                break;
            case SkillType.Strength:
                testModifier = playerStats.StrengthTestModifier;
                break;
            case SkillType.Will:
                testModifier = playerStats.WillTestModifier;
                break;
        }


        // Obtém o valor base da habilidade
        int skillValue = GetBaseSkillValue(skill);

        // Obtém qualquer melhoria na habilidade
        int improvement = playerStats.SkillImprovements[skill];

        // Calcula o total de dados a serem rolados
        int totalDice = skillValue + improvement + testModifier; ;

        // Realiza a rolagem dos dados
        int rollResult = diceRoller.RollDice(totalDice);


        // Determina se a rolagem foi bem-sucedida
        bool success = rollResult >= targetValue;

        // Registra o resultado no console de depuração
        Debug.Log($"Skill Check - Skill: {skill}, Target Value:
{targetValue}, Roll Result: {rollResult}, Success: {success}");

        // Retorna o resultado
        return success;
    }


    public void AddAsset(Card card)
    {
```

```
        Debug.Log("Adicionando asset...");
        player.Inventory.Add(card);
        Debug.Log("Asset adicionado.");
    }

    private void HandleDefeat()
    {
        Debug.Log("Checking for defeat...");

        if (playerStats.Health <= 0 || playerStats.Sanity <= 0)
        {
            Debug.Log("Investigator is defeated");
        }

        Debug.Log("Defeat check completed.");
    }

    public void LoseHealth(int amount, bool canPreventLoss = true)
    {
        Debug.Log("Losing " + amount + " health...");

        if (canPreventLoss)
        {
            // Implement logic to prevent loss with effects here
        }

        playerStats.Health = Mathf.Max(0, playerStats.Health - amount);
        HandleDefeat();

        Debug.Log("Current Health: " + playerStats.Health);
    }

    public void LoseSanity(int amount, bool canPreventLoss = true)
    {
        Debug.Log("Losing " + amount + " sanity...");

        if (canPreventLoss)
        {
            // Implement logic to prevent loss with effects here
        }

        playerStats.Sanity = Mathf.Max(0, playerStats.Sanity - amount);
        HandleDefeat();
```

```
        Debug.Log("Current Sanity: " + playerStats.Sanity);
    }

    public void RecoverHealth(int amount)
    {
        Debug.Log("Recuperando " + amount + " de sa♦de...");

        playerStats.Health = Mathf.Min(playerStats.MaxHealth,
playerStats.Health + amount);

        Debug.Log("Sa♦de atual: " + playerStats.Health);
    }

    public void RecoverSanity(int amount)
    {
        Debug.Log("Recuperando " + amount + " de sanidade...");

        playerStats.Sanity = Mathf.Min(playerStats.MaxSanity,
playerStats.Sanity + amount);

        Debug.Log("Sanidade atual: " + playerStats.Sanity);
    }
    private void CheckForDefeat()
    {
        Debug.Log("Verificando a derrota...");

        if (playerStats.Health <= 0 || playerStats.Sanity <= 0)
        {

        }

        Debug.Log("Verificação de derrota concluída.");
    }
    public void StandUp()
{
    // ...
}


    public int GetDiceCount()
    {
        int baseDiceCount = 0;
```

```
        // Determine base dice count based on the skill being tested
        SkillType skill = SkillType.Combat; // Change this to the
appropriate skill
        switch (skill)
        {
            case SkillType.Combat:
                baseDiceCount = player.stats.Strength; // Use the
appropriate stat for combat
                break;
            case SkillType.Influence:
                baseDiceCount = player.stats.Influence;
                break;
            case SkillType.Lore:
                baseDiceCount = player.stats.Lore;
                break;
            case SkillType.Observation:
                baseDiceCount = player.stats.Observation;
                break;
            case SkillType.Strength:
                baseDiceCount = player.stats.Strength;
                break;
            case SkillType.Will:
                baseDiceCount = player.stats.Will;
                break;
            default:
                Debug.LogError("Invalid skill type: " + skill);
                break;
        }
        return baseDiceCount;
    }

        public int GetTestModifier()
    {
        SkillType skill = SkillType.Strength; // Change this to the
appropriate skill for combat tests
        return player.stats.GetSkillTestModifier(skill);
    }

    public int GetImprovementModifier()
    {
        SkillType skill = SkillType.Combat; // Change this to the
appropriate skill for combat tests
```

```
        return player.stats.GetSkillImprovementModifier(skill);
    }

    public int GetImpairmentModifier()
    {
        return impairmentModifier;
    }

    public int GetBonus()
    {
        return bonus;
    }

    public int GetAdditionalDice()
    {
        return additionalDice;
    }

    public bool IsBlessed()
    {
        return isBlessed;
    }

    public bool IsCursed()
    {
        return isCursed;
    }

    public List<int> RollDice(int count)
    {
        List<int> diceResults = new List<int>();

        for (int i = 0; i < count; i++)
        {
            int rollResult = Random.Range(1, 7); // Assuming 6-sided
dice
            diceResults.Add(rollResult);
        }

        return diceResults;
    }
```

```
}
```