Clean Coding

Presentation by **Binh Lai**

Improve code quality

Meaningful variable name	Avoid confusing abbreviations and choose names that clearly reflect the intention of the variable or function.
Short and concise functions	Long and complex functions can make code reading and maintenance challenging
Avoid unnecessary comments	Clean code should be clear and self-explanatory, reducing the need for explanatory comments.
Avoid code duplication	Code duplication can increase the likelihood of errors and make maintenance difficult.
Keep function and method small	Using small functions and methods improves code readability and understanding
Proper use of comments	Comments should be used to clarify the purpose or intention of the code
Apply SOLID principles	The SOLID principles promote modularity and flexibility in the code.

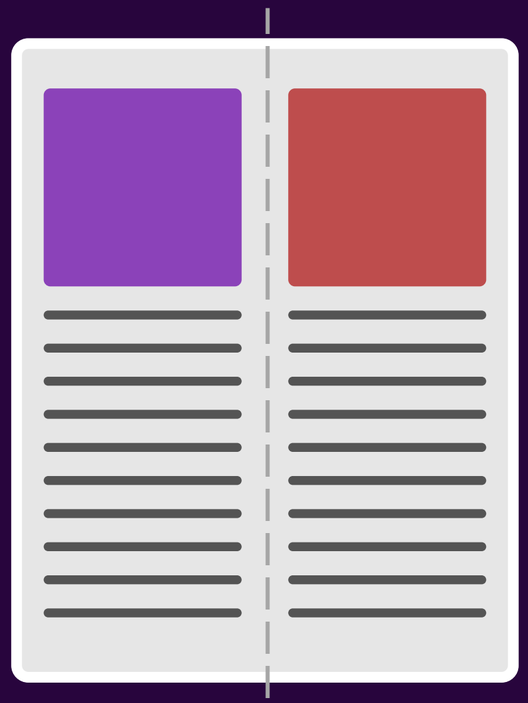
SOLID Principles

& How it change my code

Presentation by **Binh Lai**

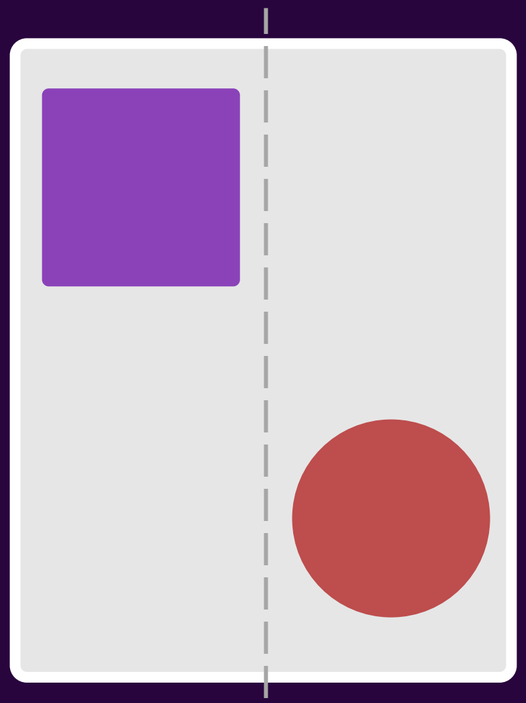
Single responsibility principle

Every software component should have one and only one responsibility



Cohesion

Cohesion is the degree to which the various parts of software component are related

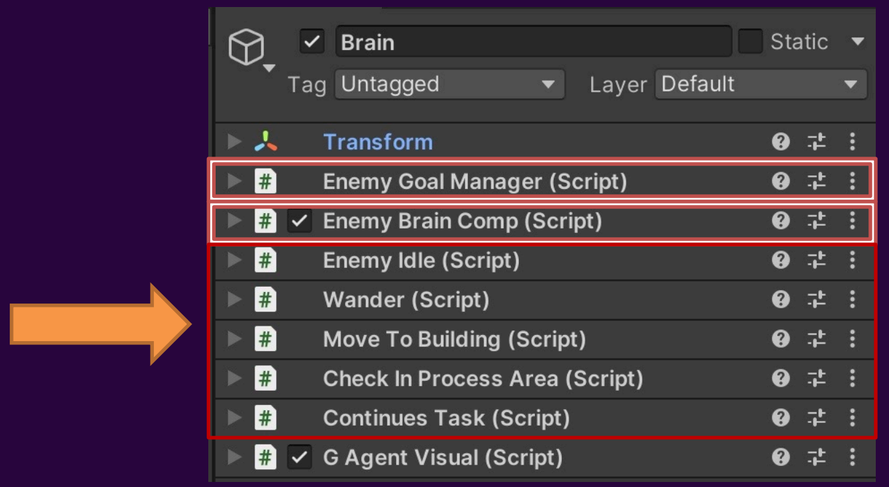


Coupling

Coupling is defined as the level of inter dependency between various software components

Cohesion

```
public void Update()
{
    StaminaIncreaser();
    if (isHaveTarget)
    {
        if (isResetAll)
        {
            isResetAll = false;
            //Turnoff Coroutine in Wander
            StopAllCoroutines();
            //To set RunOrWalk at instance
            isStateDecide = false;
            //print("have Enemy "+ isWander);
        }

        //Create attackerEnemy to solve for attackerController only
        if (!isAttacked)
        {
            if (!isAttacking) RunOrWalk();
            RunToTarget();

        }
        else
        {
            if (!isAttacking) RunOrWalk();
            RunToTarget();
        }
    }
    else
    {
        if (!isResetAll)
        {
            isResetAll = true;
            ResetAllisState();
        }
        else
        {
            Wander();
        }
    }
}
```



Responsibility: Set goals

Responsibility: Decide actions

Responsibility: Conduct actions

Responsibility: Set goals, decide actions, conduct action

Coupling

```
private void APlusAlgorithm()
{
    if (CurrentAction != null && CurrentAction.running)
    {
        float distanceToTarget = Vector3.Distance( a: destination, b: transform.position);
        if (distanceToTarget <= finishDistance)
        {
            // Debug.Log("Distance to Goal: " + distanceToTarget);
            if (!isInvoke)
            {
                Invoke( methodName: "CompleteAction", CurrentAction.Duration);
                isInvoke = true;
            }
        }

        return;
    }

    if (_planner == null || _actionQueue == null)
    {
        _planner = new GPlanner();
```

Responsibility: Select action, detect when the agent reaches the destination

```
public override void APlusAlgorithm()
{
    if (CurrentAction != null && CurrentAction.running)
    {
        if (_isDone)
        {
            _isDone = false;
            _actionQueue = null;
            if (!_isInvoke)
            {
                _isInvoke = true;
                CompleteAction();
            }
        }

        return;
    }

    if (_planner == null || _actionQueue == null)
    {
        _planner = new GPlanner();
```

Responsibility: Select action

```
private void StartMove()
{
    // Check if entity close enough from destination
    // If not, calculate path
    // Move to the closet conner
    // Wait for entity reach the conner

    var currentPos :Vector3 = m_Transform.position;
    NavMesh.CalculatePath( sourcePosition: currentPos, targetPosition: _currentDestination, areaMask: NavMesh.AllAreas, _path);
    _path.corners.OrderBy(t :Vector3 => Vector3.Distance( a: currentPos, b: t));
    _currentIndex = 0;
    SetCurrentConner();
}

Frequently called  2 usages  Nanyka
private void SetCurrentConner()
{
    _currentIndex++;
    if (_currentIndex >= _path.corners.Length)
    {
        _currentProcessUpdate.StopProcess();
        return;
    }

    var currentPos :Vector3 = m_Transform.position;
    _currentConner = _path.corners.ElementAt(_currentIndex);
    _currentConner = new Vector3(_currentConner.x, currentPos.y, _currentConner.z);

    // Rotate object to the conner
    Vector3 currentVelocity = _rigidbody.velocity;
    Quaternion offsetRotation = Quaternion.identity;
    if (currentVelocity != Vector3.zero)
        offsetRotation = Quaternion.FromToRotation(currentVelocity, Vector3.zero);
    m_Transform.rotation = Quaternion.LookRotation(_currentConner - m_Transform.position) * offsetRotation;
}

// Walking animation move the object 1 meter and check the distance to the target at the end of the animation
0+ asset usages  Nanyka
public void NewMovingLoop()
{
    if (Vector3.Distance( a: m_Transform.position,
            b: new Vector3(_currentConner.x, m_Transform.position.y, _currentConner.z)) < _stopDistance)
        SetCurrentConner();
}
```

Responsibility: Detect when the agent reaches the destination

Open-Closed principle

Software components should be closed for modification, but open for extension

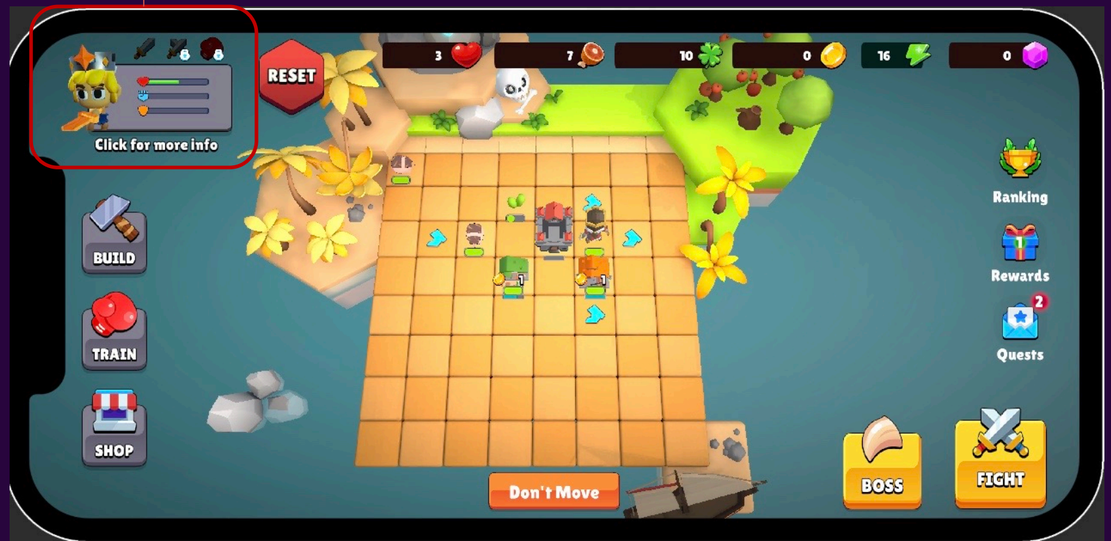
The benefits of the Open-Closed Principle:

- Ease of adding new features
- This leads to the minimal cost of developing and testing software
- OCP often requires decoupling which, in turn, automatically follows the Single responsibility principle

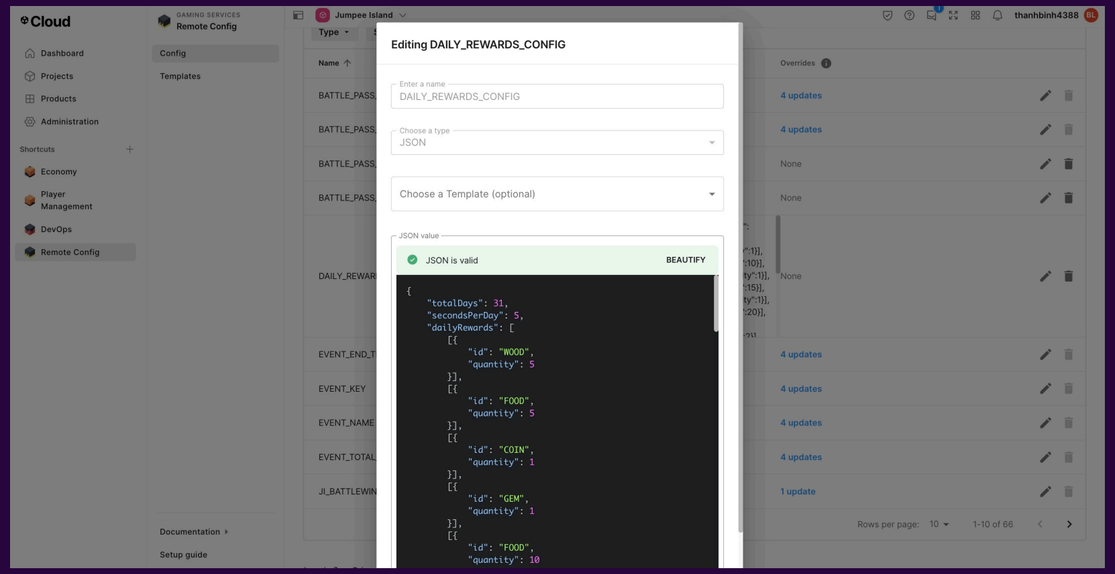
```
public class BuildingInGame : MonoBehaviour, IShowInfo, IConfirmFunction, IRemoveEntity, IGetEntityInfo,
    IAttackResponse
{
```

```
public class CreatureInGame : MonoBehaviour, IGetEntityInfo, IShowInfo, IRemoveEntity, ICreatureMove,
    IAttackResponse, ICreatureType
{
```

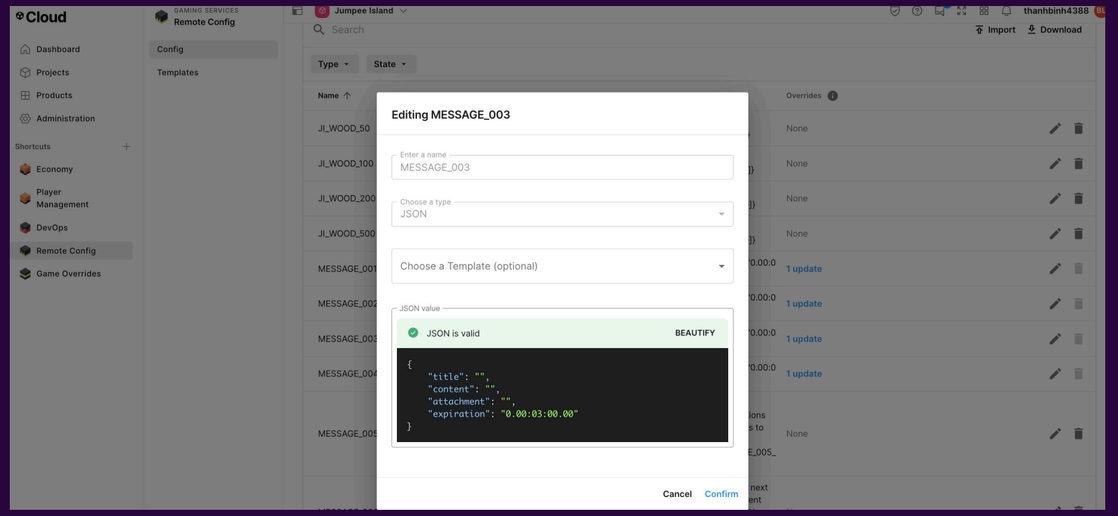
```
public interface IShowInfo
{
    1 usage  2 implementations  NanykaLab
    public (Entity entity,int jump) ShowInfo();
}
```



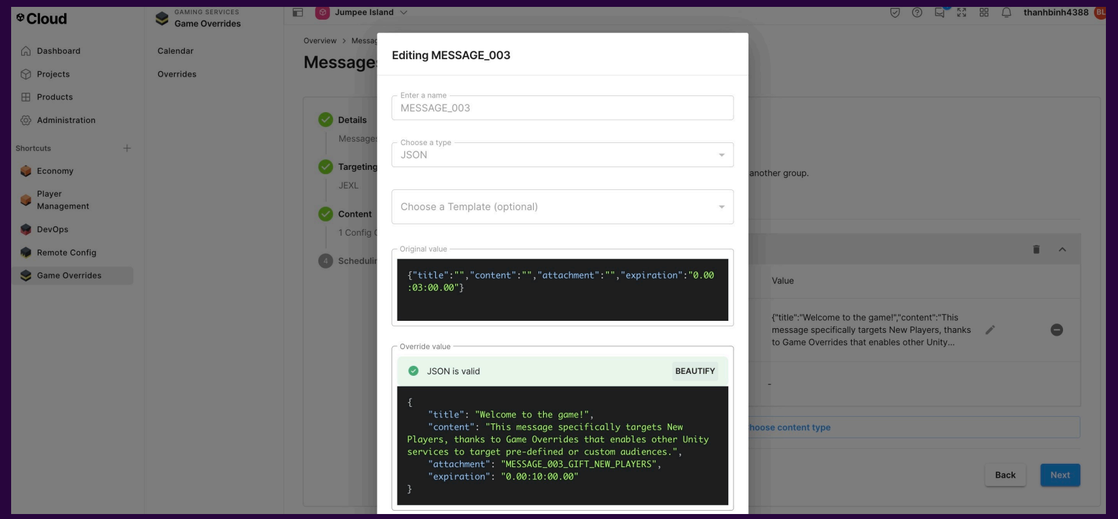
Use remote config to set daily rewards



Use remote config to define messages send to players

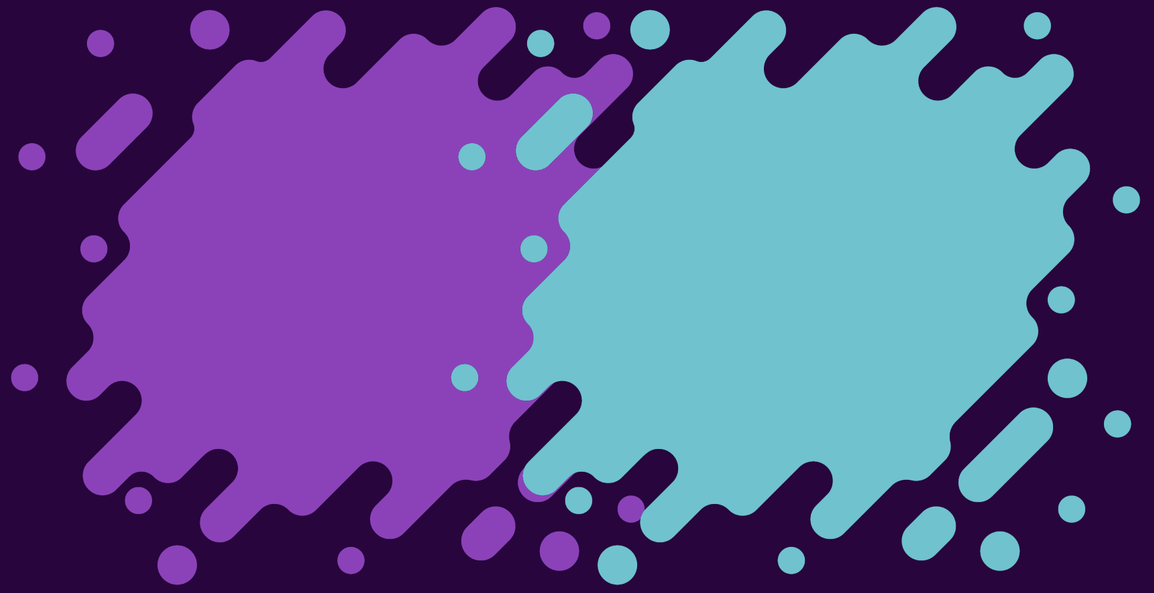


…and use Game Override to define the content of message for each type of player



Liskov substitution principle

Object should be replaceable with their subtypes without affecting the correctness of the program



Buildings lose their health and resources stocked in it

```cs
public override void TakeDamage(int damage, Entity fromEntity)
{
    if (GameFlowManager.Instance.GameMode == GameMode.ECONOMY)
    {
        if (fromEntity.GetFaction() != FactionType.Player)
            EnemyRopeCurrency(damage);

        if (m_BuildingData.BuildingType != BuildingType.MAINHALL)
            m_HealthComp.TakeDamage(damage, m_BuildingData, fromEntity);
    }
    else if (GameFlowManager.Instance.GameMode == GameMode.BATTLE)
    {
        // If player's creatures attack enemy building, they also seize loot from this storage
        if (fromEntity.GetFaction() == FactionType.Player && m_BuildingData.FactionType == FactionType.Enemy)
        {
            var seizedAmount:int = EnemyRopeCurrency(damage);

            MainUI.Instance.OnShowCurrencyVfx.Invoke(m_BuildingData.StorageCurrency.ToString(), seizedAmount,
                fromEntity.GetData().Position);
        }

        m_HealthComp.TakeDamage(damage, m_BuildingData, fromEntity);
    }
    else
        m_HealthComp.TakeDamage(damage, m_BuildingData, fromEntity);

    SavingSystemManager.Instance.OnSavePlayerEnvData.Invoke();
}
```

```cs
public abstract class Entity: MonoBehaviour
{
    #region HEALTH

    4 usages  5 overrides  NanykaLab
    public abstract void TakeDamage(int damage, Entity fromEntity);

    #endregion
```

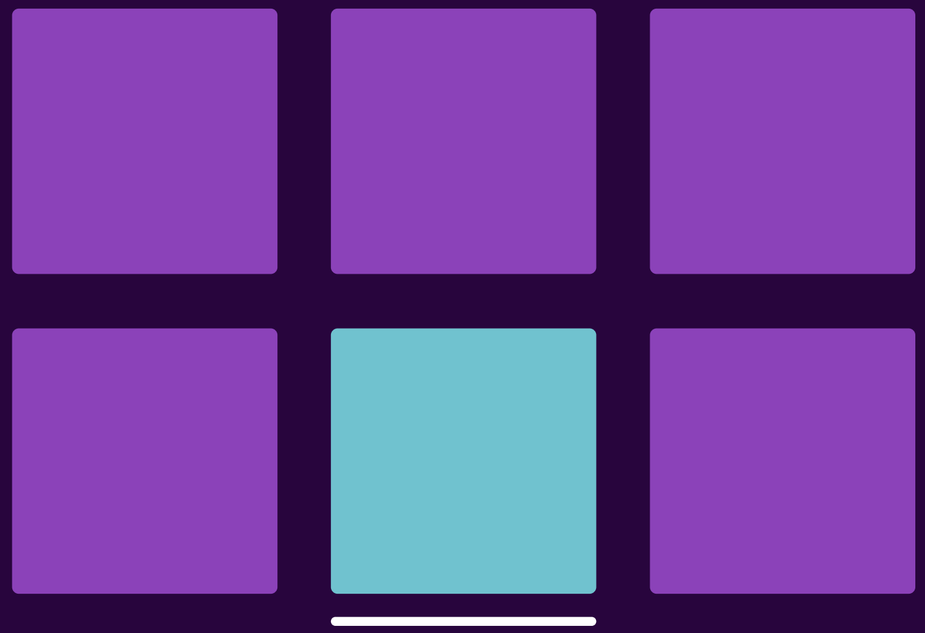
```cs
public override void TakeDamage(int damage, Entity fromEntity)
{
    if (m_CreatureData.EntityName.Equals("King") && GameFlowManager.Instance.GameMode == GameMode.ECONOMY)
        return;

    m_AnimateComp.SetAnimation(AnimateType.TakeDamage);
    m_HealthComp.TakeDamage(damage, m_CreatureData, fromEntity);
    SavingSystemManager.Instance.OnSavePlayerEnvData.Invoke();
}
```

Creature lose health only

```cs
public void Attack(Vector3 attackPoint, Entity mEntity, int jumpStep)
{
    var mEnvironment = GameFlowManager.Instance.GetEnvManager();
    var attackFaction = mEnvironment.CheckFaction(attackPoint);
    var target:GameObject = mEnvironment.GetObjectByPosition(attackPoint, attackFaction);
    if (target == null)
        return;

    if (target.TryGetComponent(out Entity targetEntity))
    {
        var selectedSkill = mEntity.GetSkills().ElementAt(jumpStep - 1);
        var skillEffect = selectedSkill.GetSkillEffect();
        if (skillEffect != null)
            skillEffect.TakeEffectOn(attackEntity:mEntity, sufferEntity:targetEntity);

        if (targetEntity.GetFaction() != mEntity.GetFaction())
        {
            targetEntity.TakeDamage(mEntity.GetAttackDamage(), mEntity);
            mEntity.GainGoldValue();
        }
    }
}
```

Interface segregation principle

No client should be forced to depend on method it does not use



```
public abstract class Entity: MonoBehaviour
{
    [NonSerialized] public UnityEvent<Entity> OnUnitDie = new();

    [Header("Default components")]
    [SerializeField] protected Transform m_Transform;  Changed in 0+ assets

    ENTITY DATA

    HEALTH

    ATTACK

    SKIN

    #region SKILL

    Frequently called  3 usages  5 overrides  NanykaLab
    public abstract IEnumerable<Skill_SO> GetSkills();

    #endregion

    #region EFFECT

    Frequently called  3 usages  5 overrides  NanykaLab
    public abstract EffectComp GetEffectComp();

    #endregion
```

```
public class CollectableEntity : Entity
{
    #region SKILL

    Frequently called  0+3 usages  new *
    public override IEnumerable<Skill_SO> GetSkills()
    {
        throw new System.NotImplementedException();
    }

    #endregion

    #region EFFECT

    Frequently called  0+3 usages  NanykaLab
    public override EffectComp GetEffectComp()
    {
        throw new System.NotImplementedException();
    }
```

```
public class CreatureEntity : Entity, IStatsProvider<CreatureStats>
{
    public override IEnumerable<Skill_SO> GetSkills()
    {
        return m_SkillComp.GetSkills();
    }

    Frequently called  1 usage  Nanyka
    public SkillComp GetSkillComp()
    {
        return m_SkillComp;
    }

    #endregion

    #region EFFECT

    Frequently called  1+3 usages  NanykaLab
    public override EffectComp GetEffectComp()
    {
        return m_EffectComp;
    }

    Frequently called  3 usages  NanykaLab
    public int GetJumpBoost()
    {
        return m_EffectComp.GetJumpBoost();
    }

    #endregion
```

Both creature and collectable object inherit Entity
The creature can suffer effects from the enemy's attack or use its skill, but not the collectible object
As a result, these override functions are **Empty method implementations** on CollectableEntity and those make the class low cohesion, but still tightly coupling

Segmentation them into interface instead of abstract functions of Entity class

```
47 usages  4 inheritors  1+1 exposing APIs
public interface IAttackRelated
{
    1 usage  4 implementations
    public Vector3 GetPosition();
    2 usages  4 implementations
    public void GainGoldValue();
    Frequently called  19 usages  4 implementations
    public FactionType GetFaction();
    3 usages  4 implementations
    public int GetAttackDamage();
    4 usages  4 implementations
    public void TakeDamage(int damage, IAttackRelated fromEntity);
    Frequently called  3 usages  4 implementations
    public IEnumerable<Skill_SO> GetSkills();
    Frequently called  3 usages  4 implementations
    public EffectComp GetEffectComp();
    1 usage  4 implementations
    public void AccumulateKills();
    Frequently called  1 usage  4 implementations
    public void UpdateTransform(Vector3 pos, Vector3 dir);
}
```

```
public class BuildingEntity : Entity, IAttackRelated
{
    public void GainGoldValue()
    {
        throw new NotImplementedException();
    }
```

```
public class CollectableEntity : Entity, IAttackRelated
{
public Vector3 GetPosition()
{
    throw new System.NotImplementedException();
}

    public override void UpdateTransform(Vector3 position, Vector3 rotation)
    {
        throw new System.NotImplementedException();
    }
public IEnumerable<Skill_SO> GetSkills()
{
    throw new System.NotImplementedException();
}

Frequently called  0+3 usages  new *
public EffectComp GetEffectComp()
{
    throw new System.NotImplementedException();
}

0+1 usages  new *
public void AccumulateKills()
{
    throw new System.NotImplementedException();
}
```

While the new interfaces help to make the coding lose coupling, they remain low cohesion with many Empty method implementations

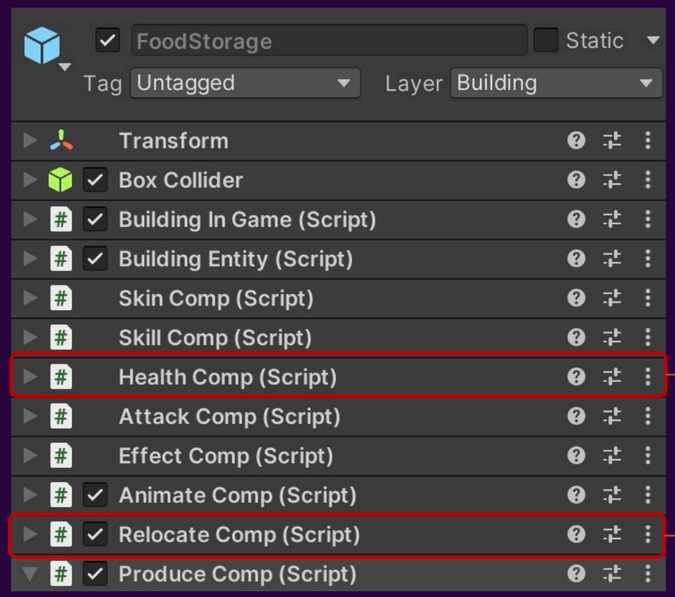
Split IAttackRelated interface into some smaller but Higher cohesive interfaces

```
public interface IAttackRelated
{
    // public Vector3 GetPosition();
    2 usages  4 implementations
    public void GainGoldValue();
    Frequently called  16 usages  4 implementations
    public FactionType GetFaction();
    3 usages  4 implementations
    public int GetAttackDamage();
    4 usages  4 implementations
    public void TakeDamage(int damage, IAttackRelated fromEntity);
    Frequently called  3 usages  4 implementations
    public IEnumerable<Skill_SO> GetSkills();
    Frequently called  3 usages  4 implementations
    public EffectComp GetEffectComp();
    1 usage  4 implementations
    public void AccumulateKills();
}

11 usages  3 inheritors  0+1 exposing APIs
public interface ISkillRelated
{
    Frequently called  1 usage  3 implementations
    public void UpdateTransform(Vector3 pos, Vector3 dir);
    Frequently called  3 usages  3 implementations
    public FactionType GetFaction();
}

public interface IGetTransformData
{
    public Vector3 GetPosition();
}
```

Dependency inversion principle

Abstractions should not depend on details. Details should depend on abstractions

Dependency Inversion is the strategy of depending upon interfaces or abstract functions and classes rather than upon concrete functions and classes.




```
public void TakeDamage(int damage, EntityData entityData, IAttackRelated killedBy)
{
    if (_isDeath)
        return;

    entityData.CurrentHp -= damage;
    var healthPortion float = entityData.CurrentHp * 1f / m_MAXHp;
    entityUI.UpdateHealthSlider(healthPortion);
    TakeDamageEvent.Invoke(healthPortion);

    if (entityData.CurrentHp <= 0)
    {
        // if (killedBy.TryGetComponent(out CreatureEntity creatureEntity))
        //     creatureEntity.AccumulateKills();

        killedBy.AccumulateKills();

        Die(killedBy);
    }
}
```

```
public class RelocateComp : MonoBehaviour
{
    private Entity m_Entity;
    private Camera _camera;
    private int _buildingLayer = 1 << 9;
    private int _tileLayer = 1 << 6;
    private bool _isSelectEntity;

    Event function  NanykaLab
    private void Start()
    {
        _camera = Camera.main;
        m_Entity = GetComponent<Entity>();
    }
```

Concrete classes depend on abstract interfaces and abstract class rather than directly connecting with other concrete classes

```
public class CreatureEntity : Entity, IStatsProvider<CreatureStats>, IAttackRelated, ISkillRelated
{
    [SerializeField] private Transform m_RotatePart;   ☢ Changed in 19 assets
    [SerializeField] private SkinComp m_SkinComp;   ☢ Changed in 19 assets
    [SerializeField] private HealthComp m_HealthComp;   ☢ Changed in 19 assets
    [SerializeField] private AttackComp m_AttackComp;   ☢ Changed in 19 assets
    [SerializeField] private SkillComp m_SkillComp;   ☢ Changed in 19 assets
    [SerializeField] private EffectComp m_EffectComp;   ☢ Changed in 19 assets
    [SerializeField] private AnimateComp m_AnimateComp;   ☢ Changed in 19 assets
    [SerializeField] private MovementComp m_MovementComp;   ☢ Unchanged
```

```
public class CharacterEntity : Entity, ISpecialSkillReceiver, ITroopAssembly, IGetEntityData<CreatureStats>
{
    Frequently called  2 usages
    public Vector3 _assemblyPoint { get; set; }

    // control components
    [SerializeField] private SkinComp m_SkinComp;   ☢ Changed in 3 assets
    [SerializeField] private MovementComp m_MovementComp;   ☢ Changed in 5 assets
    [SerializeField] private EffectComp m_EffectComp;   ☢ Changed in 3 assets
    [SerializeField] private EnemyBrainComp m_Brain;   ☢ Changed in 4 assets

    private ISkillMonitor m_SkillMonitor;
    private IHealthComp m_HealthComp;
    private IAnimateComp m_AnimateComp;
    private IMover m_Mover;
```

Thank You!

Presentation by **Binh Lai**