软件实现具体分析

1. 考虑你擅长的编程语言，那个是最危险的特性？如果你正在编写编程团队的编码指南，是否会完全禁用该特性？为什么？

1.1 我认为最危险的 C# 特性：`static` 全局可变状态（特别是单例模式）

在 C# 中，最危险的语言特性不是语法层面的“指针”或“反射”，而是 “**可全局访问的可变状态**”，例如通过：

- `public static CardManager Instance;`
- `DontDestroyOnLoad(gameObject);`

这种 Singleton（单例） + 全局可变状态的组合是许多游戏架构中 Bug 的来源。

其生命周期难以预测，且容易引发隐式依赖。多场景下，经常出现多次生成 / 销毁导致：

- 旧实例仍然存在
- 新实例未初始化
- Awake / Start 执行顺序混乱

且其剧透**隐藏的数据依赖性**，在我的游戏软件中卡牌、卡槽、UI、数据库都依赖 `CardManager.Instance`，这些依赖在编译期无法检查，往往导致运行时难以定位的 NullReferenceException。
对于测试而言，单例模式也极其不友好，无法 mock 或替换，导致测试代码耦合度过高，难以进行单元测试，不利于扩展与重构。

1.2 是否应该完全禁止

不应该完全禁止，但必须加入严格的编码规范与替代方案。
因为在游戏开发（尤其 Unity）中，全局管理器是常见架构模式，有如方便跨场景访问核心服务（如音频管理、场景管理），简化小型项目的架构设计之类的需求与优点。
替代方案（如依赖注入 DI、ServiceLocator）虽然更安全，但增加复杂度，团队未必适合。

在团队中使用时，必须设立严格规范，例如：

- Manager 类数量必须严格受控
- 明确生命周期、初始化顺序
- 单例禁止存储大量可变状态
- 状态必须通过封装的 API 访问

---

2. 找到你编写的一个程序，分析它的风格。它是否已经正确缩进？名字是否选择良好？它可以从重构中受益吗？

2.1 缩进与格式

大部分代码保持了 统一的 4 空格缩进，但代码结构非常庞大，某些函数由于过多嵌套导致视觉阅读困难。
例如：

```
private void RestoreCardsToSlot(CardSlotData cardSlotData, Dictionary<long, Card>
createdCards)
{
    foreach (var cardID in cardSlotData.cardIDs)
    {
        if (createdCards.TryGetValue(cardID, out var card))
            CardSlot.ChangeCardToSlot(card.cardSlot, slot, card, null, true);
        else
            Debug.LogError($"Unknown cardID: {cardID}");
    }
}
```

虽然缩进正确，但多层嵌套使得阅读时需要频繁跳动视线，影响理解。

2.2 命名风格

采用了 Unity 社区常见的 PascalCase/CamelCase 规范，但仍有部分名称语义不够清晰。
风格如：

```
InitProductionScene()
LoadProductionScene()
GetWorkEfficiencyValue()
TryGetCardIconAttribute()
```

2.3 整体风格总结

CardManager 是一个典型的“上帝对象（God Class）”，其承担了过多职责：

- 卡牌生成
- 卡槽管理
- UI 创建
- 存档系统
- 生产场景加载
- 战斗奖励
- 声音初始化
- 时间管理

这种过度集中的设计导致难以维护与扩展，且违反单一职责原则（SRP）。在合作中使得新成员难以理解项目结构与设计意图。

2.4 该代码是否可从重构中受益?

可以从重构中受益良多。重构可包含提取子模块、拆解过长函数、引入设计模式等，通过重构可提升代码的可读性、可维护性与测试性。

如将 CardManager 拆分为：

- **CardFactory**：卡牌生成
- **CardSlotManager**：卡槽管理
- **CardAttributeStore**：属性读写
- **SceneCardLoader**：场景加载/存档
- **EventUICoordinator**：事件卡 UI

---

3. 分析你编写的程序，简述你能找到的圈复杂度最大的方法（函数）是什么？它的圈复杂度是多少？结合你编写程序的经验，谈谈代码圈复杂度过大的缺点，并给出几种降低代码圈复杂度的方法。

3.1 圈复杂度分析

圈复杂度是衡量代码复杂度的指标，表示程序中独立路径的数量。圈复杂度越高，代码越复杂，维护难度越大。在我的程序中，圈复杂度最高的函数是：

```
LoadProductionScene()
{
    foreach (var cardData in saveData.allCardData)
    {
        object attribute = cardData.cardDescription.cardType switch
        {
            CardType.Creatures => GetCardAttribute<CreatureCardAttribute>
(cardData.cardID),
            CardType.Resources => GetCardAttribute<ResourceCardAttribute>
(cardData.cardID),
            _ => null,
        };

        if (attribute == null && cardData.cardDescription.cardType !=
CardType.Events)
            continue;

        Card card = CreateCard(...);
    }
    ...
    foreach (var (cardID, (index, progress)) in eventCardProgress)
    {
        if (tmpCardsDict.TryGetValue(cardID, out var card))
        {
            CardSlot cardSlot = card.cardSlot;
            card.StartEvent(index, progress);
        }
    }
```

```
    ...
}
```

其包含多重嵌套循环和条件分支，导致其圈复杂度大大提高。

决策点	来源	数量
foreach	5 处	+5
if / else if / continue	6–7 处	+7
switch	1 个（3 分支）	+3
其他逻辑判断	~3	+3

最终其近似圈复杂度约 15 – 18。

3.2 圈复杂度过大的缺点

1. **可读性差**：新人无法快速理解逻辑
2. **测试难度大**：每个分支都要验证
3. **Bug 容易潜伏**：尤其是多层嵌套的条件分支
4. **重构困难**：依赖耦合过多
5. **逻辑扩展困难**：新增一个功能就可能破坏已有流程
6. **测试和Debug困难**：复杂路径导致难以覆盖所有场景，增加调试难度。
7.

3.3 降低圈复杂度的方法

方法 1：提取子函数

例如：

从：

```
LoadProductionScene()
```

拆分成：

```
public void LoadProductionScene()
{
    CreateCardsFromSave();
    RestoreCardSlots();
    CreateBattleRewards();
    RestoreEventCardProgress();
    InitUI();
}
```

每个函数职责单一，易于理解与测试。

方法 2：策略模式 / 工厂模式处理不同卡牌类型

将

```
switch (card.cardDescription.cardType) { … }
```

替换为：

- `ICardAttributeHandler` 接口
- `CreatureAttributeHandler`
- `ResourceAttributeHandler`

减少 switch 链。

方法 3：提前返回（Guard Clause）减少嵌套

例如：

```
if (attribute == null) return;
```

可消除深嵌套的 else。