软件测试与质量保障实验报告

1 项目中质量保障措施分析

1.1 已采取的质量保障措施

以下基于 **Unity 卡牌合成 + 自走棋策略游戏项目** 实际情况，以 `CardManager`脚本为例子，详细阐述已实施的质量保障措施：

措施类别	措施内容	具体体现	效果
代码规范与可维护性	C#/Unity 脚本严格使用命名规范；拆分管理类（CardManager、BattleManager 等）	`CardManager` 类中使用单例模式管理卡牌，`allCards` 字典按 `CardType` 分类存储，避免单一职责违反	提高可读性与可维护性，降低 Bug 率
函数式编程与安全	使用泛型方法 `GetCardAttribute<T>()` 实现类型安全的属性访问	支持 `CreatureCardAttribute`、`ResourceCardAttribute` 等多种类型，避免类型转换错误	编译期类型检查，防止运行时异常
防御性编程	对象创建前进行有效性检查；使用 `TryGetXxx()` 模式	`CreateCard()` 中先检查 `cardDescription.IsValid()`；`TryGetCardAttribute<T>()` 返回 bool 避免 null 异常	防止无效数据导致的运行时崩溃

措施类别	措施内容	具体体现	效果
事件驱动与解耦	定义 `onCardCreated` 和 `onCardDeleted` 事件，实现模块间松耦合	CardManager 创建/删除卡牌时触发事件，其他系统订阅事件响应	提高系统灵活性和可测试性
功能测试	自测游戏核心系统（卡牌合成、战斗流程、背包拖拽、资源系统）	验证 `CreateCard()`、`DeleteCard()`、`CreateCardSlot()` 等核心方法的逻辑正确性	发现逻辑与交互错误
日志与调试	Unity Console 日志分类输出，验证战斗状态机、Buff 系统流程	使用 `Debug.Log()` 追踪卡牌生命周期：卡牌创建、槽位分配、属性修改	加速定位错误来源，便于问题复现
数据持久化与恢复	使用 `SaveDataManager` 保存游戏状态，支持断点续玩	`InitProductionScene()` 从存档文件加载卡牌数据，`LoadProductionScene()` 在场景切换时恢复状态	确保数据一致性，防止进度丢失
版本控制	Git 管理 Unity 项目，保留可回滚记录	每个功能完成后提交，记录详细提交说明	避免因错误提交导致严重后果，支持快速回滚

1.2 可以补充的质量保障措施

建议类别	可添加措施	实现方法	意义
单元测试	使用 Unity Test Framework (UTF) 对 CardManager 核心方法进行单元测试	编写测试用例验证 `CreateCard()`、`DeleteCard()`、`GetCardAttribute<T>()` 的正确性	防止修改导致回归缺陷，确保代码质量
代码静态分析	使用 SonarQube 或 Roslyn 分析器进行代码质量评估	检测代码中的坏味道、复杂度过高、安全漏洞等问题	提前发现潜在缺陷
性能压测	大规模卡牌场景下的性能测试（如 1000+ 卡牌）	监测帧率、内存占用、GC 暂停时间，优化 Dictionary 查询效率	提前暴露性能瓶颈
异常处理补强	对异常输入进行更完善的错误处理	在 `CreateCard()` 失败时返回详细错误信息而非 null；捕获 Dictionary 访问异常	防止静默失败，提高系统稳定性

1.3 应该使用的开发流程

```
代码开发
    ↓
[代码审查] → 检查规范性、逻辑正确性
    ↓
[单元测试] → 验证核心方法
    ↓
[集成测试] → 验证模块交互
    ↓
[功能测试] → 玩家角度验证
```

```
    ↓
[性能压测] → 大场景性能验证
    ↓
[CI/CD] → 自动化流程
    ↓
发布
```

---

2 用等价类划分法设计测试用例

2.1 规约条件拆解 → 有效等价类 / 无效等价类

条件来源	有效等价类	无效等价类
a. 首字符为字母，后续为字母或数字	首字符为大小写字母，后续任意组合字母/数字，如 `A`、`abc123`、`Var2`	首字符为数字如 `1a` / 含特殊字符如 `_var`、`var-name` / 全数字 `123`
b. 长度 1～8 字符	字符数范围 1～8，如 `a`（1字符）、`abcdefgh`（8字符）	空字符串（0字符） / 超长如 `abcdefghi`（9字符）或更长
c. 必须先说明后使用	变量声明在使用之前，如先 `int x;` 再 `x = 5;`	未声明直接使用，如使用变量 `y` 但从未声明
d. 一个说明语句至少有一个标识符	声明语句中包含至少一个标识符，如 `int a;` 或 `int a, b, c;`	说明语句中没有任何标识符，如单纯的 `int ;`
e. 保留字不能使用	使用合法的非保留标识符，如 `myVar`、`count`、`data`	使用 C 语言保留字如 `int`、`while`、`for`、`if`、`return` 等

2.2 等价类与测试覆盖分析

根据规约条件，设计覆盖有效和无效等价类的测试用例：

测试类别	对应规约	代表等价类	测试目的
有效等价类	a, b	格式合法、长度合法	验证合法标识符能通过检查
无效等价类	a	首字符非字母、含非法字符	验证格式错误的标识符被拒绝
无效等价类	b	空字符串、长度 ≥ 9	验证长度限制被正确执行
无效等价类	c	声明前使用	验证前向引用检查
无效等价类	d	说明语句无标识符	验证声明语句必须包含标识符
无效等价类	e	保留字	验证保留字被拒绝

2.3 测试用例设计与执行

条件 a：标识符格式检查

编号	测试输入	规约检查	预期结果	实际结果	覆盖等价类
TC01	a	首字符为字母，无后续	接受	接受	单字母
TC02	A	首字符为大写字母	接受	接受	大写字母
TC03	abc123	首字符为字母，后续混合字母数字	接受	接受	字母+数字混合
TC04	Var2Name	首字符为大写，后续混合	接受	接受	驼峰命名
TC05	1abc	**首字符为数字**	拒绝	拒绝	数字开头
TC06	_var	**首字符为下划线（特殊字符）**	拒绝	拒绝	特殊字符开头
TC07	var-name	**中间含连字符（特殊字符）**	拒绝	拒绝	含非法字符
TC08	var name	**中间含空格（特殊字符）**	拒绝	拒绝	含空格

伪代码：

```
def is_valid_identifier(identifier):
    """
    验证标识符是否符合规约
    规约:
    - 首字符必须是字母
    - 后续字符可以是字母或数字
    - 长度必须在 1~8 之间
    - 不能是保留字
    """
    # 保留字集合
    reserved_words = {
        'int', 'float', 'double', 'char', 'void',
        'if', 'else', 'while', 'for', 'do',
        'switch', 'case', 'break', 'continue',
        'return', 'struct', 'union', 'enum'
    }

    # 检查条件 b: 长度 1~8
    if len(identifier) < 1 or len(identifier) > 8:
        return False, "长度不在 1~8 范围内"

    # 检查条件 a: 首字符为字母
    if not identifier[0].isalpha():
        return False, "首字符必须是字母"

    # 检查条件 a: 后续字符为字母或数字
    for char in identifier[1:]:
        if not (char.isalnum()):
            return False, "包含非法字符（只允许字母和数字）"

    # 检查条件 e: 保留字检查
    if identifier.lower() in reserved_words:
        return False, "不能使用保留字"
```

```
    return True, "有效的标识符"

# 测试用例执行
test_cases = [
    ('a', True),
    ('abc123', True),
    ('Var2Name', True),
    ('1abc', False),
    ('_var', False),
    ('var-name', False),
    ('abcdefgh', True),
    ('abcdefghi', False),
    ('int', False),
    ('while', False),
    ('myVar', True),
]

for identifier, expected in test_cases:
    result, msg = is_valid_identifier(identifier)
    status = "PASS" if result == expected else "FAIL"
    print(f"{status}: {identifier} → {msg}")
```

条件 b：标识符长度检查

编号	测试输入	字符数	规约检查	预期结果	实际结果	覆盖等价类
TC09	a	1	最小边界	接受	接受	最小长度
TC10	ab	2	边界 +1	接受	接受	短标识符
TC11	abcdefgh	8	最大边界	接受	接受	最大长度
TC12	`` (空)	0	最小边界 -1	拒绝	拒绝	空字符串
TC13	abcdefghi	9	最大边界 +1	拒绝	拒绝	超长
TC14	abcdefghij	10	严重超长	拒绝	拒绝	大幅超长

条件 c：声明与使用顺序检查

编号	代码片段	规约检查	预期结果	实际结果	覆盖等价类
TC15	int x; x = 5;	先声明后使用	接受	接受	正确顺序
TC16	x = 5;	未声明直接使用	报错（未定义）	报错	前向引用
TC17	y = y + 1;	使用后声明	报错（使用未定义变量）	报错	前向引用

伪代码

```python
class IdentifierSymbolTable:
    """符号表实现，用于追踪标识符的声明与使用"""

    def __init__(self):
        self.declared_vars = set()  # 已声明变量集合
        self.used_vars = []  # 使用记录

    def declare(self, identifier):
        """声明一个标识符"""
        if identifier in self.declared_vars:
            raise Exception(f"变量 {identifier} 已重复声明")
        self.declared_vars.add(identifier)
        print(f"声明: {identifier}")

    def use(self, identifier):
        """使用一个标识符"""
        if identifier not in self.declared_vars:
            raise Exception(f"错误: 变量 {identifier} 未声明，不能使用")
        self.used_vars.append(identifier)
        print(f"使用: {identifier}")

    def parse_code(self, code_lines):
        """解析代码行并检查"""
        for line in code_lines:
            if 'int' in line:  # 声明语句
                var_name = line.split()[-1].rstrip(';')
                self.declare(var_name)
            elif '=' in line:  # 赋值语句
                var_name = line.split('=')[0].strip()
                self.use(var_name)

# 测试用例
print("=== 测试用例 TC15：正确顺序 ===")
table = IdentifierSymbolTable()
try:
    table.parse_code(['int x;', 'x = 5;'])
    print("TC15 PASS\n")
except Exception as e:
    print(f"TC15 FAIL: {e}\n")

print("=== 测试用例 TC16：未声明直接使用 ===")
table = IdentifierSymbolTable()
try:
    table.parse_code(['x = 5;'])
    print("TC16 FAIL: 应该报错\n")
except Exception as e:
    print(f"TC16 PASS: 正确捕获错误 - {e}\n")
```

条件 d：说明语句中必须有标识符

编号	代码片段	规约检查	预期结果	实际结果	覆盖等价类
TC18	`int a;`	一个标识符	接受	接受	单标识符
TC19	`int a, b, c;`	多个标识符	接受	接受	多标识符
TC20	`int ;`	无标识符的声明	语法错误	语法错误	无标识符

条件 e：保留字不能使用

编号	测试输入	保留字判断	预期结果	实际结果	覆盖等价类
TC21	`myVar`	不是保留字	接受	接受	普通标识符
TC22	`count`	不是保留字	接受	接受	普通标识符
TC23	`int`	是 C 语言保留字	拒绝	拒绝	保留字
TC24	`while`	是 C 语言保留字	拒绝	拒绝	保留字
TC25	`for`	是 C 语言保留字	拒绝	拒绝	保留字
TC26	`return`	是 C 语言保留字	拒绝	拒绝	保留字
TC27	`if`	是 C 语言保留字	拒绝	拒绝	保留字

3 边界值分析：NextDate 程序测试用例

3.1 问题规约分析

计算第二天日期的 NextDate 程序规约：

- **输入参数**：`year`、`month`、`day`
- **定义域**：`year ∈ [1950, 2050]`、`month ∈ [1, 12]`、`day ∈ [1, 31]`
- **功能**：输入一个有效的日期，返回其后一天的日期
- **约束条件**：
  - 日期必须有效（考虑闰年、大小月等）
  - 跨越月份和年份的日期计算
  - 输入超出范围时应返回错误

3.2 边界值划分

对每个变量，取：

类型	下边界 -1	下边界	下边界+1	上边界-1	上边界	上边界+1
year	1949	1950	1951	2049	2050	2051
month	0	1	2	11	12	13
day	0	1	2	30	31	32

3.3 边界值组合策略

使用**边界值分析法**的三种覆盖策略：

策略 1：逐一边界法（One Boundary）

对每个变量的边界值与其他变量的中间值组合

策略 2：边界对覆盖法（Boundary Pair）

两个变量同时取边界值，第三个变量取中间值

策略 3：全边界法（Full Boundary）

所有变量都取边界值

3.4 关键组合测试用例

3.4.1 Year 边界值测试

编号	year	month	day	预期结果	覆盖点	说明
ND01	1949	6	15	错误：year 下越界	year 下越界-1	年份低于最小值
ND02	1950	1	1	1950/1/2	year 下边界	年份最小值正常
ND03	1951	6	15	1951/6/16	year 下边界+1	年份最小值正常
ND04	2049	12	31	2050/1/1	year 上边界-1	年底跨越年份
ND05	2050	6	15	2050/6/16	year 上边界	年份最大值正常
ND06	2051	6	15	错误：year 上越界	year 上越界+1	年份高于最大值

3.4.2 Month 边界值测试

编号	year	month	day	预期结果	覆盖点	说明
ND07	1950	0	15	错误：month 下越界	month 下越界-1	月份低于最小值
ND08	1950	1	15	1950/1/16	month 下边界	一月中间日期正常
ND09	1950	2	15	1950/2/16	month 下边界+1	二月中间日期正常
ND10	1950	11	15	1950/11/16	month 上边界-1	十一月中间日期正常
ND11	1950	12	15	1950/12/16	month 上边界	十二月中间日期正常
ND12	1950	13	15	错误：month 上越界	month 上越界+1	月份高于最大值

3.4.3 Day 边界值测试

编号	year	month	day	预期结果	覆盖点	说明
ND13	1950	6	0	错误：day 下越界	day 下越界-1	日期低于最小值
ND14	1950	6	1	1950/6/2	day 下边界	月初正常
ND15	1950	6	2	1950/6/3	day 下边界+1	月初正常
ND16	1950	6	30	1950/7/1	day 上边界-1	六月末跨月
ND17	1950	6	31	错误：day 无效	day 上越界	六月无31日
ND18	1950	7	32	错误：day 上越界	day 严重越界	日期高于最大值

伪代码

```
def is_leap_year(year):
    """判断是否为闰年"""
    return (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0)

def days_in_month(year, month):
    """返回该月的天数"""
    if month in [1, 3, 5, 7, 8, 10, 12]:
        return 31  # 大月
    elif month in [4, 6, 9, 11]:
        return 30  # 小月
    elif month == 2:
        return 29 if is_leap_year(year) else 28

def is_valid_date(year, month, day):
    """检查日期是否有效"""
    if year < 1950 or year > 2050:
        return False, "年份超出范围 [1950, 2050]"
    if month < 1 or month > 12:
        return False, "月份超出范围 [1, 12]"
    if day < 1 or day > 31:
        return False, "日期超出范围 [1, 31]"

    # 检查该月是否有该天数
    max_day = days_in_month(year, month)
    if day > max_day:
        return False, f"该月只有 {max_day} 天，不能是 {day} 日"

    return True, "有效日期"

def next_date(year, month, day):
    """计算下一天的日期"""
    # 验证输入
    valid, msg = is_valid_date(year, month, day)
    if not valid:
        return None, f"错误: {msg}"

    # 简单情况：月中的日期
```

```
    max_day = days_in_month(year, month)
    if day < max_day:
        return (year, month, day + 1), "正常"

    # 月末情况：跨月
    if month < 12:
        return (year, month + 1, 1), "跨月"

    # 年末情况：跨年
    return (year + 1, 1, 1), "跨年"

# 测试用例执行
print("=== Year 边界值测试 ===")
test_cases_year = [
    (1949, 6, 15, False, "ND01: year 下越界"),
    (1950, 1, 1, True, "ND02: year 下边界"),
    (1951, 6, 15, True, "ND03: year 下边界+1"),
    (2049, 12, 31, True, "ND04: year 上边界-1"),
    (2050, 6, 15, True, "ND05: year 上边界"),
    (2051, 6, 15, False, "ND06: year 上越界"),
]

for year, month, day, should_pass, desc in test_cases_year:
    result, msg = next_date(year, month, day)
    status = "PASS" if (result is not None) == should_pass else "FAIL"
    print(f"{status} {desc}: {msg}")

print("\n=== 闰年特殊情况测试 ===")
test_cases_leap = [
    (2000, 2, 28, True, "ND24: 闰年二月28日→29日"),
    (2000, 2, 29, True, "ND25: 闰年二月29日→3月1日"),
    (1900, 2, 28, True, "ND26: 百年非闰年→3月1日"),
    (2001, 2, 28, True, "ND27: 平年二月28日→3月1日"),
    (2001, 2, 29, False, "ND28: 平年无29日"),
]

for year, month, day, should_pass, desc in test_cases_leap:
    result, msg = next_date(year, month, day)
    status = "PASS" if (result is not None) == should_pass else "FAIL"
    output = f"{result}" if result else "Invalid"
    print(f"{status} {desc}: {output}")
```

3.4.4 月份特殊情况测试

编号	year	month	day	预期结果	覆盖点	说明
ND19	1950	1	31	1950/2/1	大月末	一月末跨月
ND20	1950	4	30	1950/5/1	小月末	四月末跨月（小月）
ND21	1950	4	31	错误：day 无效	小月越界	四月无31日

编号	year	month	day	预期结果	覆盖点	说明
ND22	1950	9	30	1950/10/1	小月末	九月末跨月（小月）
ND23	1950	11	30	1950/12/1	小月末	十一月末跨月（小月）

3.4.5 闰年特殊情况测试

编号	year	month	day	预期结果	覆盖点	说明
ND24	2000	2	28	2000/2/29	闰年二月28日	闰年二月有29日
ND25	2000	2	29	2000/3/1	闰年二月29日	闰年二月末跨月
ND26	1900	2	28	1900/3/1	百年非闰年	1900非闰年，二月只有28日
ND27	2001	2	28	2001/3/1	平年二月28日	平年二月末跨月
ND28	2001	2	29	错误：day 无效	平年无29日	平年二月无29日
ND29	2000	1	1	2000/1/2	闰年元旦	闰年正常工作

3.4 闰年判断规则说明

```
闰年定义：
- 年份能被 4 整除 且 （不能被 100 整除 或 能被 400 整除）
- 即：(year % 4 == 0 && year % 100 != 0) || (year % 400 == 0)

例外：
- 2000 年：2000 % 400 == 0 → 闰年
- 1900 年：1900 % 100 == 0 && 1900 % 400 != 0 → 平年
```

4 单元测试、集成测试、确认测试的理解与应用

4. 对单元测试、集成测试、确认测试的理解

4.1 单元测试

单元测试是对软件中最小可测试的逻辑单元进行独立验证的测试方法。在面向对象中，单元通常指类或类中的方法。
主要目标是验证每一个功能单元的输入、输出和边界条件均符合预期。

4.2 集成测试

集成测试是把多个模块组合起来，检查模块之间的交互关系是否正确。
当模块单独运行正确时，它们组合到一起可能产生接口不匹配、数据传递错误等问题。

常见策略有：

- 自顶向下
- 自底向上
- 增量式集成测试

4.3 确认测试

确认测试是从用户需求角度验证整个系统是否符合需求规格说明，是面向用户的最终测试阶段，通常由测试人员或用户参与完成，是交付前的重要环节。

4.2 在项目中的应用

4.2.1 单元测试应用（Unity Test Framework）

测试对象：CardManager 核心方法 使用到的单元测试如下：

```
[Test]
public void TestCreateCard_ValidInput_ReturnsCard()
{
    CardDescription desc = new CardDescription {
        cardType = CardType.Creatures, cardName = "TestCard"
    };
    Card card = CardManager.Instance.CreateCard(desc, Vector2.zero);

    Assert.IsNotNull(card);
    Assert.AreEqual("TestCard", card.cardName);
}

[Test]
public void TestDeleteCard_ExistingCard_RemovesFromDictionary()
{
    Card card = CreateTestCard();
    CardManager.Instance.DeleteCard(card.cardID);

    bool exists = CardManager.Instance.TryGetCardAttribute<CreatureCardAttribute>(
        card.cardID, out _);
    Assert.IsFalse(exists);
}
```

测试覆盖：

- `CreateCard()` - 有效/无效输入
- `DeleteCard()` - 存在/不存在的卡牌
- `GetCardAttribute<T>()` - 类型转换正确性
- `TryGetCardAttribute<T>()` - 返回值验证

4.2.2 集成测试应用

测试场景：模块间交互

```
[UnityTest]
public IEnumerator TestCardCreation_TriggersUIUpdate()
{
    int initialCount = MainUIManager.Instance.GetCardCount();
    CardManager.Instance.CreateCard(CreateTestCardDescription(), Vector2.zero);
    yield return null;

    Assert.AreEqual(initialCount + 1, MainUIManager.Instance.GetCardCount());
}
```

集成点测试：

- CardManager ↔ UI 系统（卡牌创建触发界面更新）
- CardManager ↔ SaveDataManager（存档加载数据一致性）
- CardManager ↔ 事件系统（事件触发与订阅）

4.2.3 确认测试应用

测试场景：端到端用户流程

测试用例	测试步骤	验收标准
卡牌合成	1. 拖拽 3 张同类卡牌到合成槽 2. 点击合成按钮 3. 查看结果	合成成功获得 2 星卡牌 消耗的卡牌消失 UI 显示正确
战斗流程	1. 布置 5 张战斗卡牌 2. 开始战斗 3. 观察战斗过程 4. 查看结算	卡牌正确加载 AI 正常工作 奖励正确发放
存档恢复	1. 游玩并执行各种操作 2. 保存退出 3. 重新加载	所有进度保留 卡牌状态一致 资源数量正确
性能测试	1. 创建 100+ 卡牌 2. 频繁合成操作 3. 监控性能	FPS ≥ 50 内存 < 2GB 无卡顿崩溃
