一、整体概述

本次对 `Stardew_Valley` 项目的重构，核心目标是：

- **将原本硬编码的业务逻辑拆分为可扩展的策略 / 状态对象**，降低类复杂度；
- **通过多种设计模式解耦模块之间的依赖关系**，方便多人协作与后续扩展；
- **为季节、天气、好感度、地图与作物生长等玩法预留统一扩展点**，支撑课程设计和加分项。

本报告主要围绕以下几个方面展开：

- 商店与定价系统：**策略模式（Strategy）+ 观察者模式（Observer）**；
- 时间系统与全局驱动：**观察者模式（Observer）**；
- 农作物生长系统：**状态模式（State）**；
- 地图子系统：**桥接模式（Bridge）+ 适配器模式（Adapter）+ 工厂模式（Factory）**；
- NPC 管理系统：**空对象模式（Null Object Pattern）**；
- 工程配置与协作规范的辅助性改动。

二、商店模块与价格策略（Strategy 模式）

2.1 涉及文件

- `Classes/Store/Store.cpp`
- `Classes/Store/Store.h`
- `Classes/Store/Strategy/PricingStrategy.h`
- `Classes/Store/Strategy/SeasonalPricingStrategy.h`
- `Classes/Store/Strategy/SeasonalPricingStrategy.cpp`
- `Classes/Store/Strategy/WeatherPricingStrategy.h`
- `Classes/Store/Strategy/WeatherPricingStrategy.cpp`
- `Classes/Store/Strategy/AffectionPricingStrategy.h`
- `Classes/Store/Strategy/AffectionPricingStrategy.cpp`
- 依赖：`Control/TimeManager.h`，`proj.win32/Constant.h` 等

2.2 修改前的典型问题（背景）

在原始设计中，商店的价格逻辑通常直接写在 `Store.cpp` 里：

- 根据季节、天气、星期几等条件进行 `if-else` 或 `switch` 判断；
- 若未来想加“好感度折扣”“节日活动折扣”等新逻辑，只能不断往 `Store` 里追加条件分支；
- 导致 `Store` 职责过重，既管库存、又管 UI 回调、又管所有定价策略，**严重违背单一职责和开闭原则。**

2.3 抽象价格策略接口 `PricingStrategy`

文件：`Classes/Store/Strategy/PricingStrategy.h`

- 定义了统一的价格策略接口：

```
class PricingStrategy {
public:
    virtual ~PricingStrategy() = default;
```

```
    // 对单个商品应用价格策略（原地修改 totalPrice）
    virtual void apply(ProductNode& product, const TimeManager& timeManager) = 0;
};
```

- 其中：
  - `ProductNode` 表示商店中的单个商品（包含类型、数量、总价、适用季节等）；
  - `TimeManager` 提供当前季节、天气、星期等环境信息。

设计模式说明：`PricingStrategy` 是经典的 **策略模式（Strategy）** 抽象接口，后续所有价格规则都以"策略"的形式实现。

2.4 季节价格策略 `SeasonalPricingStrategy`

文件：`SeasonalPricingStrategy.h / SeasonalPricingStrategy.cpp`

- 功能：根据**当前季节**与商品的 `discountSeason` / `increaseSeason` 调整价格。实现大致逻辑：
  - 若 `product.discountSeason == 当前季节`，按 `DISCOUNT_RATE_BY_SEASON` 打折；
  - 若 `product.increaseSeason == 当前季节`，按 `INCREASE_RATE_BY_SEASON` 涨价；
  - 否则不改价。

价值：将"季节折扣 / 涨价"的逻辑从 `Store` 中抽离，集中封装在一个策略类中，方便调整和测试。

2.5 天气价格策略 `WeatherPricingStrategy`

文件：`WeatherPricingStrategy.h / WeatherPricingStrategy.cpp`

- 功能：根据**天气**调整农作物相关商品的价格：
  - 通过 `TimeManager` 获取当前 `Weather`；
  - 若商品类型为 `Seed` 或 `Base`：
    - 雨天（`Rainy`）略微涨价（示例为 ×1.05）；
    - 干燥（`Dry`）略微降价（示例为 ×0.95）。

价值：天气影响逻辑与商店本身解耦，未来可以很方便地修改涨跌比例，或增加对其他天气（如暴雨、下雪）的处理。

2.6 好感度价格策略 `AffectionPricingStrategy`

文件：`AffectionPricingStrategy.h / AffectionPricingStrategy.cpp`

- 目标：为"**按 NPC 好感度影响价格**"提供扩展点；
- 当前实现：
  - `apply` 已经实现，参数签名与接口一致；
  - 由于目前项目中尚未有统一的好感度系统，内部暂时不修改 `product.totalPrice`，仅占位；
  - 后续如果增加"玩家与店主好感度"数据，只需在该类中补全逻辑即可。

价值：从架构层面预留了好感度定价策略的插槽，为加分项或后续扩展留足空间，而不用再改动 Store 主逻辑。

2.7 `Store` 如何使用这些策略

文件：`Classes/Store/Store.cpp`

- 在构造函数中初始化策略列表：

```
_pricingStrategies.push_back(new SeasonalPricingStrategy());
_pricingStrategies.push_back(new WeatherPricingStrategy());
_pricingStrategies.push_back(new AffectionPricingStrategy());
```

- 在 `updatePrices()` 中统一应用所有策略：

```
for (auto& product : _product) {
    for (auto* strategy : _pricingStrategies) {
        if (strategy) {
            strategy->apply(product, *_timeManager);
        }
    }
}

// 保留原有的星期四统一折扣逻辑
std::string weekDay = _timeManager->getWeekDay();
if (weekDay == "Thursday") {
    for (auto& product : _product) {
        product.totalPrice = static_cast<int>(product.totalPrice *
DISCOUNT_RATE_BY_SEASON);
    }
}
```

总结（本模块的设计模式应用）：

- **策略模式（Strategy）**：
  - 抽象策略：`PricingStrategy`；
  - 具体策略：`SeasonalPricingStrategy`、`WeatherPricingStrategy`、`AffectionPricingStrategy`；
  - 上下文：`Store`，只负责持有策略列表并调用 `apply`。
- **优点**：
  - 新增任意定价规则 → 新增一个策略类即可，基本无需修改 `Store`；
  - `Store` 类从复杂的“价格细则”中解放，职责更聚焦，**符合开闭原则**；
  - 可以在最终报告中作为 Strategy 模式的典型案例。

三、时间系统与观察者模式（Observer）

3.1 涉及文件

- `Classes/Control/TimeManager.h`
- `Classes/Control/TimeManager.cpp`
- `Classes/Control/Observer/TimeObserver.h`
- 实现 `TimeObserver` 接口的类（例如 `Store` 等）

3.2 设计结构

- `TimeManager`：负责维护游戏内的**日期、季节、天气、星期**等时间相关信息，是**被观察者（Subject）**；
- `TimeObserver`：定义了时间相关事件的观察者接口，包括：
  - `onDayChanged(int day)`：新的一天开始；
  - `onSeasonChanged(Season season)`：季节变化；
  - `onTimeChanged(int hour, int minute)`：更细粒度的时间变动等。

以商店为例，`Store` 同时扮演 `TimeObserver`：

- 构造函数中注册为观察者：

```
if (_timeManager) {
    _timeManager->attach(this);
}
```

- 析构时解除注册，防止悬空指针：

```
if (_timeManager) {
    _timeManager->detach(this);
}
```

- 实现时间回调：

```
// TimeObserver: 新的一天
void Store::onDayChanged(int /*day*/) {
    refreshStock();
}

// TimeObserver: 季节变化
void Store::onSeasonChanged(Season /*season*/) {
    updatePrices();
}

// TimeObserver: 时间变化（当前未使用，可根据需求扩展）
void Store::onTimeChanged(int /*hour*/, int /*minute*/) {
    // 暂不根据小时/分钟动态调整价格
}
```

3.3 设计模式分析

- **观察者模式（Observer）**：
  - Subject：`TimeManager`；
  - Observer 接口：`TimeObserver`；
  - 具体观察者：`Store` 以及未来可能的作物系统、节日活动系统、UI 系统等。
- **优点**：

- 时间系统与各业务模块实现解耦，“谁关心时间变化，谁就实现 `TimeObserver` 接口”；
- 新增一个“随时间变化”的模块（例如节日活动管理器）时，只需实现观察者接口并注册，无需修改 `TimeManager` 内部代码；
- 整个架构从“主动轮询”转变为“事件驱动”，代码更清晰，也更符合游戏逻辑。

四、农作物生长系统与状态模式（State）

4.1 涉及文件

- `Classes/Crops/Crops.cpp`
- `Classes/Crops/Crops.h`
- `Classes/Crops/State/CropState.h`
- `Classes/Crops/State/SeedState.h`
- `Classes/Crops/State/SproutingState.h`
- `Classes/Crops/State/GrowingState.h`
- `Classes/Crops/State/MatureState.h`

4.2 重构动机

农作物的生长具有明显的阶段性：

- 种子阶段 → 发芽阶段 → 生长期 → 成熟阶段；

若直接在 `Crops` 中用 `switch` 或 `if-else` 判断当前“生长天数 / 生长阶段”，会导致：

- 生长逻辑集中在一个巨大的函数中，可读性差；
- 新增“枯萎”“病虫害”等特殊阶段时，必须修改原有的公共逻辑，风险较大。

4.3 状态模式设计

- `CropState`：抽象状态基类，定义各个阶段共有的接口，例如：
  - `update(Crops& crop, const TimeManager& timeManager)`：在一天或一个时间片内更新作物状态；
  - 以及按需定义的 `onEnter()`、`onExit()` 等钩子函数；
- 具体状态类：
  - `SeedState`：种子阶段；
  - `SproutingState`：发芽阶段；
  - `GrowingState`：生长期；
  - `MatureState`：成熟期。
- `Crops`：作为“环境类（Context）”，内部持有一个指向当前 `CropState` 的指针：
  - 在每个时间步调用 `state->update(*this, timeManager)`；
  - 当满足一定条件时，由状态类内部控制阶段迁移（例如从 `SeedState` 切换到 `SproutingState`）。

4.4 设计模式分析

- **状态模式（State）**：
  - 不再由 `Crops` 自己通过一堆 `if-else` 判断当前处于哪个生长阶段；
  - 每个状态封装自己阶段的逻辑和向下一阶段的迁移条件；
  - `Crops` 只需持有并委托当前状态对象即可。

- **优点**：
  - 各阶段逻辑高度聚合，代码更易维护和阅读；
  - 新增特殊阶段（如“凋谢”“病害”）仅需增加新的 `XXXState`，并在合适位置切换状态即可，**降低对已有代码的侵入性**；
  - 非常适合作为报告中 State 模式的教学示例。

五、地图子系统：Bridge + Adapter + Factory

5.1 涉及文件（根据项目结构）

- 抽象地图与实现：
  - `Classes/Maps/GameMap.h`
  - `Classes/Maps/GameMap.cpp`
  - `Classes/Maps/Bridge/MapImplementation.h`
  - `Classes/Maps/Bridge/DefaultMapImplementation.h`
  - `Classes/Maps/Bridge/DefaultMapImplementation.cpp`
- 适配器：
  - `Classes/Maps/Adapter/MapAdapter.h`
  - `Classes/Maps/Adapter/MapAdapter.cpp`
- 工厂：
  - `Classes/Maps/Factory/MapFactory.h`
  - `Classes/Maps/Factory/MapFactory.cpp`
- 各具体地图：
  - `Classes/Maps/FarmMap.h / .cpp`
  - `Classes/Maps/IndoorMap.h / .cpp`
  - `Classes/Maps/MineMap.h / .cpp`
  - `Classes/Maps/TownMap.h / .cpp`

5.2 桥接模式（Bridge）

- `GameMap`：对上层（如 `GameViewController`、`MapSwitchManager`）暴露统一的地图接口，例如：
  - 加载地图资源；
  - 处理玩家在地图上的移动与碰撞；
  - 提供传送点、NPC 刷新点等高层功能。
- `MapImplementation` / `DefaultMapImplementation`：封装具体的地图加载与绘制细节：
  - 例如加载不同 TMX 地图文件；
  - 设置室内 / 室外光照；
  - 配置背景音乐等。
- `GameMap` 内部持有一个 `MapImplementation*` 指针：
  - 所有地图操作最终通过该实现类落地。

设计模式分析：

- 这是典型的 **桥接模式（Bridge）**：
  - 抽象（`GameMap`）与实现（`MapImplementation`）分离，通过组合而非继承来解耦；
  - 可以独立扩展“地图抽象接口”和“地图底层实现”，相互影响最小。

5.3 适配器模式（Adapter）

- `MapAdapter`：作为"旧地图实现"和"新统一接口"之间的适配层：
  - 对上暴露统一方法（例如 `load`, `changeMap`, `getCurrentMap` 等）；
  - 对下调用 `FarmMap` / `MineMap` / `TownMap` 等具体实现。
- 使用场景：
  - 原有代码中可能直接操作 `FarmMap` 等具体类；
  - 通过 `MapAdapter`，让上层逻辑（如 `InteractionManager`、`LoginLayer` 页面）只依赖统一接口，不感知具体地图类型。

设计模式分析：

- **适配器模式（Adapter）**：
  - 统一对外的地图操作接口，屏蔽具体地图实现细节；
  - 为兼容旧代码、平滑过渡到新架构提供过渡层。

5.4 工厂模式（Factory）

- `MapFactory`：集中管理各种地图的创建逻辑：
  - 根据地图类型枚举或 ID，返回相应的地图实例（农场、城镇、矿洞、室内等）；
  - 未来新增地图类型，只需在 `MapFactory` 中注册一次。
- 优点：
  - 避免在各处随意 `new FarmMap` / `new TownMap`，创建逻辑统一管理；
  - 可以在工厂内部实现缓存、懒加载或者对象池。

设计模式分析：

- **工厂模式（Factory）**：
  - 封装对象创建过程，调用方只关心"要什么地图"，不关心"怎么创建"；
  - 符合单一职责和开闭原则，便于扩展新地图类型。

六、NPC 管理系统与空对象模式（Null Object Pattern）

6.1 涉及文件

- `Classes/Character/NullNPC.h`
- `Classes/Character/NullNPC.cpp`
- `Classes/Control/NpcManager.h`
- `Classes/Control/NpcManager.cpp`
- `Classes/Character/NPC.h`（基类）

6.2 重构动机

在原始设计中，`NpcManager::getNPCByName()` 在找不到指定 NPC 时会返回 `nullptr`：

```
NPC* NpcManager::getNPCByName(const std::string& name) {
    for (auto npc : _npcs) {
        if (npc->getName() == name) {
            return npc;
        }
    }
```

```
    return nullptr;  // 找不到时返回空指针
}
```

调用方必须进行空指针检查：

```
NPC* abigail = NpcManager::getInstance()->getNPCByName("Abigail");
if (abigail) {  // 必须检查空指针
    abigail->showDialog();
} else {
    CCLOG("NPC not found!");
}
```

问题：

- 调用方需要频繁进行 `if (npc != nullptr)` 检查，代码冗余且容易遗漏；
- 一旦遗漏空指针检查，程序可能崩溃（空指针解引用）；
- 不符合"防御性编程"原则，代码安全性较低。

6.3 空对象模式设计

Null Object Pattern（空对象模式） 是一种行为型设计模式，用于消除空指针检查，提供"什么都不做"的默认行为。

参考：Wikipedia - Null Object Pattern

6.3.1 `NullNPC` 类实现

文件：`Classes/Character/NullNPC.h/.cpp`

- `NullNPC` 继承自 `NPC`，实现所有 `NPC` 的方法，但所有方法都是"空操作"：

```
class NullNPC : public NPC {
public:
    static NullNPC* getInstance();  // 单例模式

    // 重写所有 NPC 方法，提供空操作
    void showDialog() override;
    void showDialogue(const std::string& dialogueText) override;
    void increaseAffection(int value) override;
    // ... 其他方法均为空操作
};
```

- 实现示例：

```
void NullNPC::showDialog() {
    // 空操作：不显示任何对话框，但不会崩溃
```

```
    CCLOG("NullNPC::showDialog() - No NPC found, operation ignored.");
}
```

- `NullNPC` 使用单例模式，确保全局只有一个空对象实例。

6.3.2 `NpcManager` 修改

修改 `NpcManager::getNPCByName()`，找不到 NPC 时返回 `NullNPC` 实例：

```
NPC* NpcManager::getNPCByName(const std::string& name) {
    for (auto npc : _npcs) {
        if (npc->getName() == name) {
            return npc;
        }
    }
    // 返回 Null Object，消除调用方的空指针检查
    return NullNPC::getInstance();
}
```

同时，`NpcManager::showDialog()` 不再需要空指针检查：

```
void NpcManager::showDialog(NPC* npc) {
    npc->showDialog();  // 即使 npc 是 NullNPC，也会安全执行（空操作）
}
```

6.4 设计模式分析

- **空对象模式（Null Object Pattern）**：
  - `NullNPC` 是“空对象”，提供与 `NPC` 相同的接口，但所有方法都是空操作；
  - `NpcManager` 在找不到 NPC 时返回 `NullNPC` 而非 `nullptr`，消除调用方的空指针检查。
- **优点**：
  - **代码更简洁**：调用方无需频繁检查 `if (npc != nullptr)`；
  - **更安全**：即使调用方忘记检查，也不会因空指针解引用而崩溃；
  - **符合多态原则**：`NullNPC` 与真实 `NPC` 都继承自同一基类，可以无缝替换；
  - **便于调试**：`NullNPC` 的方法可以输出日志，帮助定位“找不到 NPC”的问题。
- **应用场景**：
  - 适用于“查找对象可能失败，但调用方希望安全地继续执行”的场景；
  - 在游戏开发中，NPC、物品、地图对象等都可能找不到，空对象模式能有效提升代码健壮性。

6.5 与其他模式的关系

- **与单例模式结合**：`NullNPC` 使用单例模式，确保全局只有一个空对象实例，节省内存；
- **与策略模式类似**：都通过多态提供不同的行为实现，但 Null Object 专注于“空操作”场景。

七、工程配置与协作相关改动

6.1 工程文件 `Stardew_Valley.vcxproj` 更新

文件：`proj.win32/Stardew_Valley.vcxproj`

- 为了解决“无法解析的外部符号（undefined reference）”问题，并让策略类真正参与编译链接，本次在工程文件中新增了以下编译项：

```
<ClCompile Include="..\Classes\Store\Store.cpp" />
<ClCompile Include="..\Classes\Store\Strategy\SeasonalPricingStrategy.cpp" />
<ClCompile Include="..\Classes\Store\Strategy\WeatherPricingStrategy.cpp" />
<ClCompile Include="..\Classes\Store\Strategy\AffectionPricingStrategy.cpp" />
```

- 这样，三种价格策略的实现文件会被正确编译并链接到最终的 `Stardew_Valley.exe` 中。

6.2 `.ipch` 等 VS 缓存文件不纳入版本控制

项目中出现的 `.vs/.../ipch/*.ipch` 文件（例如 `DEFAULTMAPIMPLEMENTATION.ipch`）：

- 作用：Visual Studio 的 IntelliSense 预编译头缓存，用于提高智能提示速度；
- 特点：体积大、与本地环境高度相关、可自动重新生成；
- 结论：**不应提交到 Git 仓库**，应在 `.gitignore` 中忽略。

建议在 `.gitignore` 中添加：

```
/.vs/
*.ipch
*.sdf
*.opensdf
*.VC.db
*.VC.VC.opendb
```

6.3 编码警告（C4819）与 UTF-8 统一

编译日志中多次出现：

- `warning C4819: 该文件包含不能在当前代码页(936)中表示的字符。请将该文件保存为 Unicode 格式以防止数据丢失`

原因：

- 源文件中包含中文注释（包含本次重构新增的中文头部注释）；
- 当前 VS 代码页为 936（GBK），对部分 UTF-8 字符支持不完整。

建议：

- 将工程中含中文注释的 `.h/.cpp` 文件统一保存为 **UTF-8（带 BOM）**；
- 在团队规范中明确“统一使用 UTF-8 编码”，避免乱码与潜在数据丢失。

八、设计模式应用总结（可用于最终报告小结）

综合本次重构，可以在最终设计报告中这样总结：

- **商店与价格策略**：
  - 通过 **策略模式（Strategy）** 抽象出 `PricingStrategy` 接口，具体实现包括按季节定价的 `SeasonalPricingStrategy`、按天气定价的 `WeatherPricingStrategy`、预留好感度定价的 `AffectionPricingStrategy` 等；
  - 商店 `Store` 只负责维护策略列表并调用 `apply`，极大减轻了自身的逻辑复杂度，实现了价格逻辑的"可插拔化"和**开闭原则**。
- **时间子系统**：
  - 使用 **观察者模式（Observer）** 设计了 `TimeManager` 与 `TimeObserver`，由时间管理器在日期、季节、天气等发生变化时，自动通知商店、作物等模块；
  - 各模块只需实现 `TimeObserver` 接口并注册/注销，即可接入时间系统，形成统一的"时间驱动架构"。
- **农作物生长系统**：
  - 利用 **状态模式（State）** 将作物生长过程拆分为 `SeedState`、`SproutingState`、`GrowingState`、`MatureState` 等多个状态类；
  - `Crops` 只需持有当前状态并委托其执行更新逻辑，新阶段（如枯萎、病害）可以在不修改原有核心逻辑的前提下平滑扩展。
- **地图子系统**：
  - 通过 **桥接模式（Bridge）** 将 `GameMap` 抽象接口与 `MapImplementation` 具体实现分离，使高层逻辑与底层 TMX 加载、光照、资源管理解耦；
  - 借助 **适配器模式（Adapter）** 的 `MapAdapter` 封装多种地图具体实现，对上层暴露统一地图接口，提升可维护性；
  - 使用 **工厂模式（Factory）** 的 `MapFactory` 集中管理不同地图对象的创建逻辑，为新增地图类型提供统一入口。
- **NPC 管理系统**：
  - 使用 **空对象模式（Null Object Pattern）** 实现 `NullNPC` 类，当 `NpcManager::getNPCByName()` 找不到指定 NPC 时返回 `NullNPC` 实例而非 `nullptr`；
  - 消除调用方的空指针检查，提升代码安全性与简洁性，符合防御性编程原则。

通过以上多种设计模式的综合应用（包括 GoF 23 种经典模式中的 Strategy、Observer、State、Bridge、Adapter、Factory，以及额外的 Null Object Pattern），本次重构在不改变原有游戏玩法的前提下，大幅提升了代码的可读性、可扩展性和可维护性，为后续功能扩展与课程报告撰写打下了良好基础。