1. 用例分析

1.1加载数独游戏用例描述表

|  |  |
| --- | --- |
| 用例名 | 加载数独游戏 |
| 简要描述 | 用户从数据文件中选择并加载一个数独问题。在用户确认后，棋盘将显示部分预填的数字，并开始计时。 |
| 参与者 | 主参与者：用户  系统：负责加载数独问题并初始化棋盘。 |
| 涉众 | 用户：希望游戏尽快加载并显示初始状态。  开发团队：需要确保加载逻辑高效且文件格式兼容。 |
| 相关用例 | 输入解答：加载完成后，用户开始填写答案。  计时功能：加载完成后系统开始计时。 |
| 前置条件 | 系统已启动并进入主界面。  文件选择路径可用，并存在有效的数独文件。 |
| 后置条件 | 系统显示初始棋盘，并开始计时。 |
| 基本事件流  1. 用户选择“加载数独”选项。  2. 系统弹出文件选择窗口。  3. 用户选择一个数独数据文件并点击“确认”。  4. 系统读取文件并解析初始数独棋盘。  5. 系统将部分数字填入棋盘，并开始计时。  6. 用户确认进入游戏。 | |
| 备选事件流  A-1：文件无效或损坏  1. 用户选择了一个无效或格式不正确的文件。  2. 系统提示错误信息，并返回文件选择窗口。  A-2：用户取消加载  1. 用户在文件选择窗口中点击“取消”。  2. 系统返回主界面，不加载任何数独任务。 | |

1.2输入解答用例描述表

|  |  |
| --- | --- |
| 用例名 | 加载数独游戏 |
| 简要描述 | 用户向棋盘的某个空白单元格输入解答，并由系统验证其合法性。 |
| 参与者 | 主参与者：用户  系统：负责检测用户输入的合法性。 |
| 涉众 | 用户：希望快速知道输入是否符合规则。 |
| 相关用例 | 提示候选数字：如果用户请求提示，系统将提供候选数字。  撤销与恢复：错误输入可以通过撤销功能回退。 |
| 前置条件 | 数独棋盘已加载。  用户可以进行操作。 |
| 后置条件 | 系统更新棋盘状态。 |
| **基本事件流**  1. 用户点击某个空白单元格。  2. 用户输入一个数字（1-9）。  3. 系统检查该数字在当前行、列和 3x3 宫格中是否重复。  4. 如果合法，则更新棋盘状态；否则提示错误。 | |
| 备选事件流  输入非法数字  1. 用户输入了 1-9 之外的数字或字符。  2. 系统弹出提示：“输入无效，请输入1到9之间的数字。” | |

1.3提示候选数字用例描述表

|  |  |
| --- | --- |
| 用例名 | 提示候选数字 |
| 简要描述 | 用户选择某个空白单元格并请求系统提供候选数字。 |
| 参与者 | 主参与者：用户  系统：负责计算候选数字并返回。 |
| 涉众 | 无 |
| 相关用例 | 输入解答：用户根据提示填写数字。 |
| 前置条件 | 数独棋盘已加载。  存在空白单元格。 |
| 后置条件 | 系统更新候选数字列表。 |
| **基本事件流**  1. 用户点击某个空白单元格。  2. 用户选择“提示候选数字”按钮。  3. 系统计算该单元格的所有合法候选数字，并返回给用户。 | |
| 备选事件流  无候选数字  1. 如果当前状态下没有合法候选数字。  2. 系统弹出提示：无可用候选数字。 | |

1.4撤销与恢复用例描述表

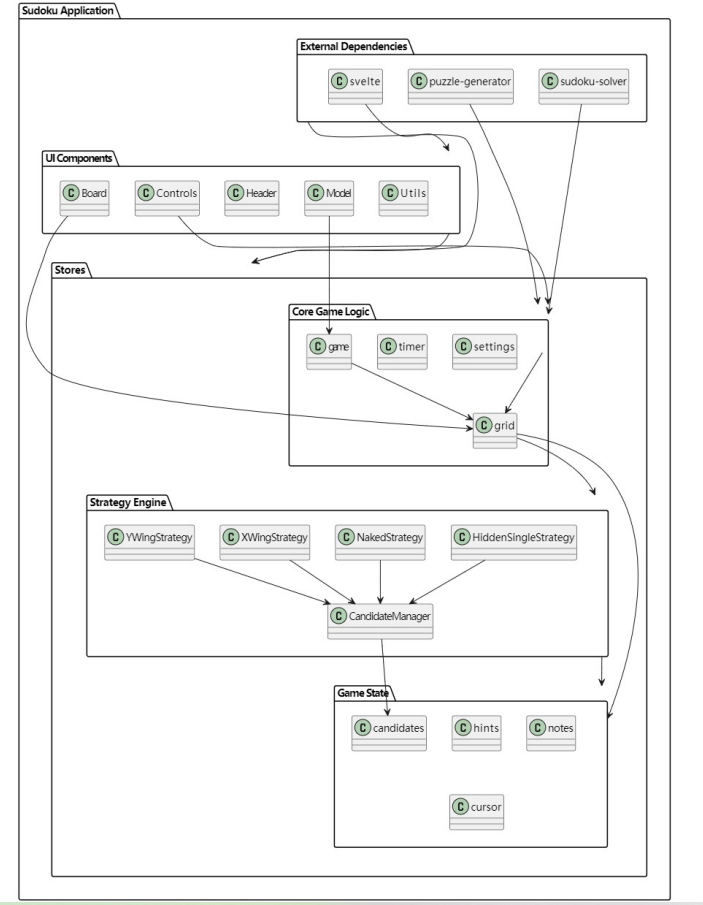
|  |  |
| --- | --- |
| 用例名 | 撤销与恢复 |
| 简要描述 | 用户可以撤销最近一次输入的操作，或恢复到之前撤销的状态。 |
| 参与者 | 主参与者：用户  系统：负责维护操作历史并执行撤销或恢复。 |
| 涉众 | 无 |
| 相关用例 | 输入解答：用户错误输入后可以撤销。  重置游戏：撤销与恢复功能不会影响重置后的状态。 |
| 前置条件 | 数独棋盘已加载并开始游戏。 |
| 后置条件 | 棋盘状态更新为撤销/恢复后的状态。 |
| **基本事件流**  1. 用户点击“撤销”按钮。  2. 系统将棋盘恢复到上一步状态。  3. 用户点击“恢复”按钮。  4. 系统前进到下一步状态。 | |
| 备选事件流  无更多状态可撤销  1. 用户点击“撤销”按钮，但没有更多历史状态。  2. 系统提示：“无法再撤销。” | |

1.5 重置游戏用例描述表

|  |  |
| --- | --- |
| 用例名 | 重置游戏 |
| 简要描述 | 用户可以将棋盘恢复到初始状态，所有输入的解答将被清除。 |
| 参与者 | 主参与者：用户  系统：负责重置棋盘。 |
| 涉众 | 无 |
| 相关用例 | 加载数独游戏：重置后状态与初始加载状态一致。 |
| 前置条件 | 数独游戏已加载。 |
| 后置条件 | 棋盘恢复到初始状态。 |
| **基本事件流**  1. 用户点击“重置”按钮。  2. 系统将棋盘恢复到初始状态，并清除所有输入。  3. 用户可以重新开始游戏。 | |
| 备选事件流  用户取消重置  1. 用户选择重置时，系统弹出确认窗口。  2. 用户选择“取消”，系统保持当前状态。 | |

2.领域模型

主要分为几个模块：UI组件、核心游戏逻辑、策略引擎、游戏状态和外部依赖。UI组件包括工具类 (Utils)、控件(Controls)、头部组件(Header)、游戏板(Board)和数据模型(Model)，负责用户交互和界面 展示。核心游戏逻辑模块管理游戏的基本操作，如设置(settings)、计时器(timer)、游戏主逻辑(game)和 网格(grid)。策略引擎模块包含不同的求解策略，如HiddenSingle、Naked-pairs、 XWing和YWing，以及候选数字管理器(CandidateManager)。游戏状态模块管理候选 数字(candidates)、提示(hints)、用户笔记(notes)和光标状态(cursor)。外部依赖模块包括Svelte框架、 数独谜题生成器(puzzle-generator)和数独求解器(sudoku-solver)，为应用程序提供必要的功能支持。 这些模块通过明确的依赖关系连接在一起，构成了整个应用的核心逻辑和交互流程



1. 系统技术架构
2. **探索回溯**
   * 在中期汇报之前，使用栈结构实现了undo、redo和reset功能。此前的回溯功能只能回到最初始未填入数字的状态，缺少分支。
   * 当前版本改为 DAG 结构，并补充之前缺少的分支回溯功能。
   * 具体实现已更新至strategy-chain分支。
3. **下一步提示**
4. **题目导入**
   * 支持两种 URL 格式：
     1. 不含候选值，仅包含 0~9 的数字。
     2. 包含候选值状态的位图编码形式。
   * URL 的导入支持两种位置：
     1. 在welcome界面直接导入 URL。
     2. 在左上角选择难易程度时，通过Enter Code导入 URL。
5. **资源集成，策略实现**

图示

描述已自动生成

策略调度流程

* + 策略按照难度划分为简单策略、中级策略和高级策略：
    1. 简单策略：Last\_remaining，hidden\_single
    2. 中级策略：naked\_pairs
    3. 高级策略：X-Wing，Y-Wing
  + Note: 在资源调度器的设计中，尝试重新设计策略管理器的架构为类似于推理链的形式，包含行、列、九宫格三个迭代。但当前无法使用到最终项目中。
  + 资源调度器思路如下:
    1. StrategyPoint.js：负责表示数独中的单元格位置和候选数；提供点位的字符串表示和比较功能
    2. StrategyGroup.js：存储策略分析的结果，管理受影响的点位和需要删除的候选数
    3. StrategyScanner.js：提供遍历数独盘面的功能；支持扫描行、列、宫和自定义区域
    4. StrategyScheduler.js：管理和调度不同的解题策略；提供策略注册和查找功能；维护全局调度器实例
    5. BaseStrategy.js：所有具体策略的基类；提供通用的策略功能和接口
    6. 在Action.svelte中对所有策略先进行注册，再调用

4.对象模型

#### 优点

**(1) 模块化与高内聚低耦合**

**模块化清晰：**从图中可以看出，各个类（如Game、Grid、Keyboard、Solver等）职责明确，各自负责不同的功能模块。例如，Grid类专注于网格相关的操作，如subscribe、unsubscribe、dissolveCandidates等方法，这使得每个类的功能相对独立，便于理解和维护。

**低耦合：**类之间的依赖关系相对明确且较少。例如，Game类与其他类（如Difficulty、Setting、Grid等）存在关联，但这种关联是通过特定的接口和方法来实现的，而不是紧密的耦合。这样当某个类的内部实现发生变化时，对其他类的影响较小，提高了系统的可维护性和可扩展性。

1. **可扩展性**

**易于添加新功能：**如果需要添加新的功能或模块，例如增加一种新的游戏模式或一种新的输入设备支持，由于现有的类已经有了明确的职责划分，可以相对容易地添加新的类或者在现有类的基础上进行扩展，而不会对整个架构造成太大的冲击。例如，如果要添加新的难度级别，可以在Difficulty类中进行扩展，而不会影响到Game类和其他类的核心功能。

策略模式的应用（如StrategyScheduler和Strategy）：StrategyScheduler类与Strategy类的组合允许灵活地切换不同的策略。这意味着可以在不修改现有代码的基础上，通过添加新的Strategy子类来实现不同的游戏策略或算法，例如不同的解谜策略或游戏流程控制策略，提高了系统的灵活性和可扩展性。

1. **可维护性**

**封装性良好：**各个类的属性大多被设置为私有（图中未明确显示，但通常类的属性会进行封装），并通过公有方法来提供对外的接口。这种封装性可以有效地隐藏类的内部实现细节，防止外部代码对类的内部状态进行随意修改，从而降低了代码出错的可能性，提高了系统的稳定性和可维护性。

1. **灵活性**

**配置与设置分离（如Setting类）：**Setting类的存在使得系统的配置和设置可以独立管理。这意味着可以方便地对游戏的各种参数（如难度设置、默认值等）进行配置和修改，而不需要修改其他核心类的代码。这种分离使得系统能够更好地适应不同的运行环境和用户需求，提高了系统的灵活性。

**事件驱动机制（如subscribe和unsubscribe方法）：**在多个类中（如Grid、Game等）出现的subscribe和unsubscribe方法，系统采用了事件驱动的架构。这种机制使得各个模块之间可以通过发布和订阅事件来进行通信，而不是直接调用彼此的方法。这样可以降低模块之间的直接依赖，使得系统更加灵活，易于扩展和维护。例如，当Grid类中的某些状态发生变化时，可以通过事件通知其他相关的类（如Game类），不需要Grid类知道具体有哪些类会对这些事件感兴趣。

#### 缺点

**(1) 缺少设计模式支持**

**问题：**在当前的设计中，没有看到明显的设计模式支持。例如，Timer 和 Game 的耦合性可能过高，Hints 缺乏对提示逻辑的进一步封装。

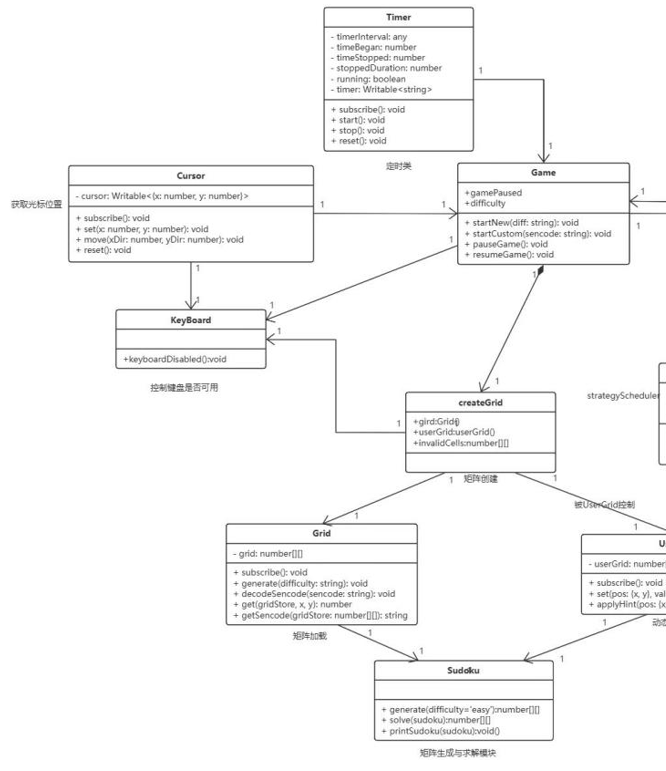
**改进建议：**使用观察者模式将 Timer 的状态变化通知给其他相关对象。

引入工厂模式为 Grid 生成逻辑服务。

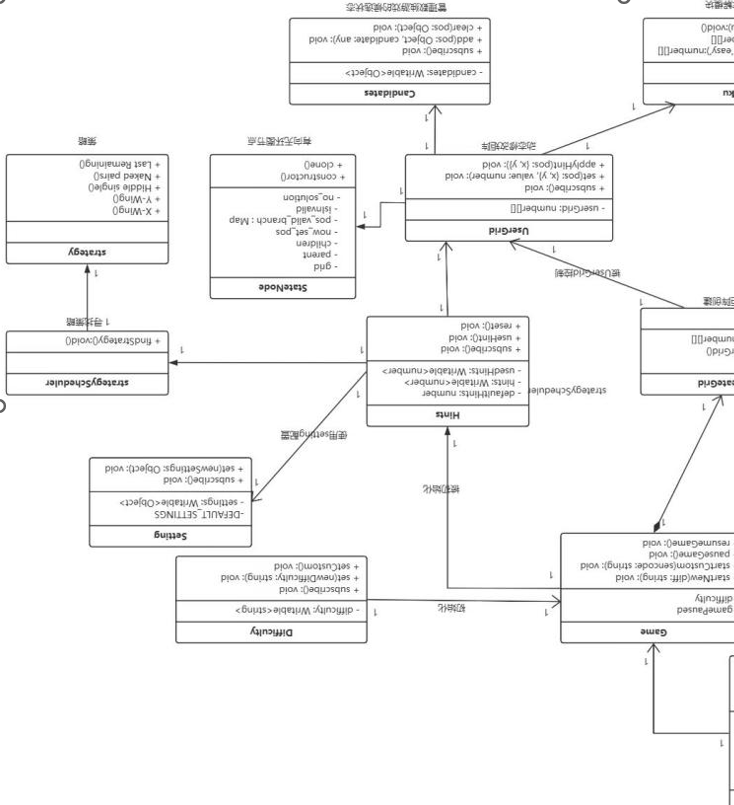
**(2) 数据一致性问题**

**问题：**多个类（如 UserGrid 和 Grid）可能会维护相似的数据，从而增加数据不一致的风险。

**改进建议：**统一数据来源，将核心网格逻辑集中到一个类中，通过状态切换或注入机制区分用户输入和系统生成的数据。



UML图(1)



UML图(2)

5.设计说明

**设计原则**

**开放封闭原则**

通过继承或实现接口来添加新功能，而不修改现有类的源代码

各种策略如**HiddenSingle**、**X-Wing**，**Y-Wing**等封装为独立的对象，通过`StrategyScheduler`统一调度，在调度时形成策略链，这些策略可以在不修改现有代码的情况下进行扩展或替换。

**依赖倒置原则**

高层模块不直接依赖低层模块，而是通过抽象进行依赖。如**Hints**依赖于抽象的解决方法类，而不是具体的策略。

**单一职责原则**

每个组件只负责一个功能。如`Board`组件负责显示数独棋盘， `Controls`组件负责用户操作，`keyboard`组件负责键盘行为

**设计模式**

**观察者模式**

定义对象间的一种一对多依赖关系，当一个对象状态发生改变时，其依赖者会自动收到通知。

在stores中使用观察者模式实现响应式更新，利用 `Svelte` 的 `store` 特性，如通过 `hintsAvailable`订阅状态变化，用 `setting` 通知状态变更，保证组件响应数据变动，维护界面与数据一致性。

**状态模式**

允许对象在内部状态改变时改变其行为。

采用状态模式管理游戏状态，将游戏状态作为对象，并通过**DAG**储存状态，以实现回溯和探索，避免条件判断堆砌，增强代码可读性和可维护性。

**组合模式**

将对象组合成树形结构，以表示“部分-整体”的层次结构。

运用组合模式构建组件树，将多个 Cell 组件组合成 Board 组件，不同组件负责不同功能，通过属性传递数据，提高组件复用性和可扩展性。

**单例模式**

确保一个类只有一个实例，并提供全局访问点。在Game和Settings类上应用单例模式。Game 用于管理游戏状态，避免在多个地方重复创建游戏实例；Settings 集中管理游戏设置，确保设置一致性。

**策略模式**

定义一系列算法，将每个算法封装起来，使它们可以互换。将每一种实现的策略，如**HiddenSingle**、**X-Wing**，**Y-Wing**等封装为独立的对象，统一通过`StragetySchduler`调度，可拓展性强。

**命令模式**

将请求封装为对象，以便使用不同的请求、队列或日志来参数化对象。

运用命令模式处理键盘输入，将键盘操作封装为命令对象，实现统一执行接口，根据键值选择相应命令执行，提高系统可扩展性和可维护性。  
**OOT设计特点**

1. **封装**

从代码可见良好的数据封装，通过封装保护了游戏核心数据和逻辑；提供了清晰的接口供外部使用；便于维护和扩展

1. **继承和多态**

复用基类的通用功能；子类专注于特定功能；继承与多态结合提高了代码的可维护性和可读性