

1. 项目概述

1.1 项目背景

本项目选自 GitHub 开源仓库 `jonasgeiler/sudoku`，该项目实现了一个基于 Svelte + TailwindCSS 的 Web 数独小游戏。系统支持选择难度、填写数字、候选笔记、提示、计时以及通关判定等基础功能，整体界面简洁，交互清晰。从软件工程视角来看，该项目规模适中且功能边界明确，同时作者在 README 中明确指出系统仍存在功能缺失，如撤销/重做（Undo/Redo）与自定义题目创建等。

1.2 技术栈

系统采用现代前端技术栈实现，其主要技术组成如下：

- 前端框架：Svelte
- UI 样式：TailwindCSS
- 构建工具：Rollup
- 状态管理：Svelte Store（writable / derived）
- 第三方库：
 - `fake-sudoku-puzzle-generator`（题目生成）
 - `@mattflow/sudoku-solver`（数独求解）

2. 面向对象分析

2.1 系统愿景

从需求层面出发，系统的总体愿景可以概括为：构建一个轻量易上手且具有良好交互体验的 Web 数独游戏系统，使用户能够顺畅地完成数独练习过程，并在设计层面为后续功能扩展预留充足空间。

2.2 参与者

- 用户：系统的唯一外部参与者，负责进行数独游戏操作。

2.3 用例分析

1. 开始游戏并选择难度

参与者：用户

前置条件：用户已进入数独游戏主界面

操作流程：

- 用户点击“开始游戏”或“新游戏”按钮
- 系统展示可选的游戏难度选项
- 用户选择目标难度
- 系统根据所选难度生成对应的数独题目
- 系统初始化棋盘状态并开始游戏

结果：成功生成并显示符合所选难度的数独棋盘，游戏进入进行状态

2. 选择单元格并进行交互

参与者：用户

前置条件：数独游戏已开始，棋盘已正确显示

操作流程：

- ◆ 用户点击棋盘上的某一单元格
- ◆ 系统记录该单元格为当前选中位置
- ◆ 系统高亮显示被选中的单元格

结果：当前选中单元格被明确标识，为后续操作提供上下文

3 填写数字与清空单元格

参与者：用户

前置条件：已选中目标单元格，且该单元格允许编辑

操作流程：

- ◆ 用户在选中单元格后输入一个数字
- ◆ 系统接收输入并更新棋盘显示
- ◆ 用户可选择删除操作以清空该单元格内容
- ◆ 系统同步更新棋盘状态

结果：单元格内容被正确填写或清空，棋盘状态得到更新

4 使用候选笔记辅助解题

参与者：用户

前置条件：数独游戏进行中，目标单元格尚未填写确定数字

操作流程：

- ◆ 用户选中目标单元格
- ◆ 用户进入候选笔记模式
- ◆ 用户选择一个或多个候选数字
- ◆ 系统将候选数字记录为该单元格的笔记内容

结果：候选笔记成功记录，用于辅助后续解题分析

5 冲突检测与错误提示

参与者：用户

前置条件：用户已在棋盘中填写数字

操作流程：

- ◆ 系统在用户填写或修改数字后触发规则检测
- ◆ 系统检测是否存在同行、同列或同宫冲突
- ◆ 若发现冲突，系统以高亮等方式提示相关单元格

结果：玩家能够直观地识别违反数独规则的填写结果

6 计时、暂停与继续游戏

参与者：用户

前置条件：游戏已开始，计时功能已启用

操作流程：

- ◆ 系统在游戏开始时自动启动计时
- ◆ 用户点击暂停按钮
- ◆ 系统暂停游戏并停止计时
- ◆ 用户点击继续按钮
- ◆ 系统恢复游戏并继续计时

结果：计时状态与游戏状态保持一致，解题时间被准确记录

7. 完成数独并获得反馈

参与者：用户

前置条件：棋盘中所有单元格均已填写

操作流程：

- ◆ 系统检测当前棋盘是否满足数独规则
- ◆ 若检测通过，系统判定游戏完成
- ◆ 系统向用户展示完成反馈信息

结果：游戏被成功判定为完成状态，用户获得通关反馈

8. 游戏设置的管理

参与者：用户

前置条件：用户进入数独游戏页面

操作流程：

- ◆ 用户点击“设置”按钮
- ◆ 系统弹出设置窗口
- ◆ 用户对设置项进行修改
- ◆ 用户点击“保存”按钮
- ◆ 系统验证设置项合法性
- ◆ 配置生效并返回游戏界面

结果：系统成功更新并应用新的设置项

2.4 领域模型

领域模型用于描述系统在业务层面所涉及的对象及其职责，而不关心这些对象在具体实现中如何被编码。本项目领域模型具体包括：

(1) **Game**: 用于表示一次完整的数独游戏过程。它描述了当前游戏的整体状态，包括所选难度、游戏是否处于进行或暂停状态、已用时间以及提示的使用情况。该概念贯穿玩家从开始游戏到完成游戏的整个流程，是系统中最核心的对局级对象。

(2) **Puzzle**: 表示一份具体的数独题面，包含题目中预先给定的数字布局。题目决定了玩家需要解答的具体内容，并通常与难度等级相关联，是数独游戏能够反复进行的重要基础。

(3) **SudokuBoard** 描述数独的 9×9 网格结构，是题目在游戏过程中的具体承载形式。棋盘不仅包含题面中固定的数字，还需要动态反映玩家在解题过程中填写和修改的内容。

(4) **Cell** 是棋盘中的最小组成单元，每个单元格对应一个确定的位置，并可能包含一个数字或处于未填写状态。单元格还承担候选笔记、冲突标记等信息，是玩家与系统交互最频繁的对象。

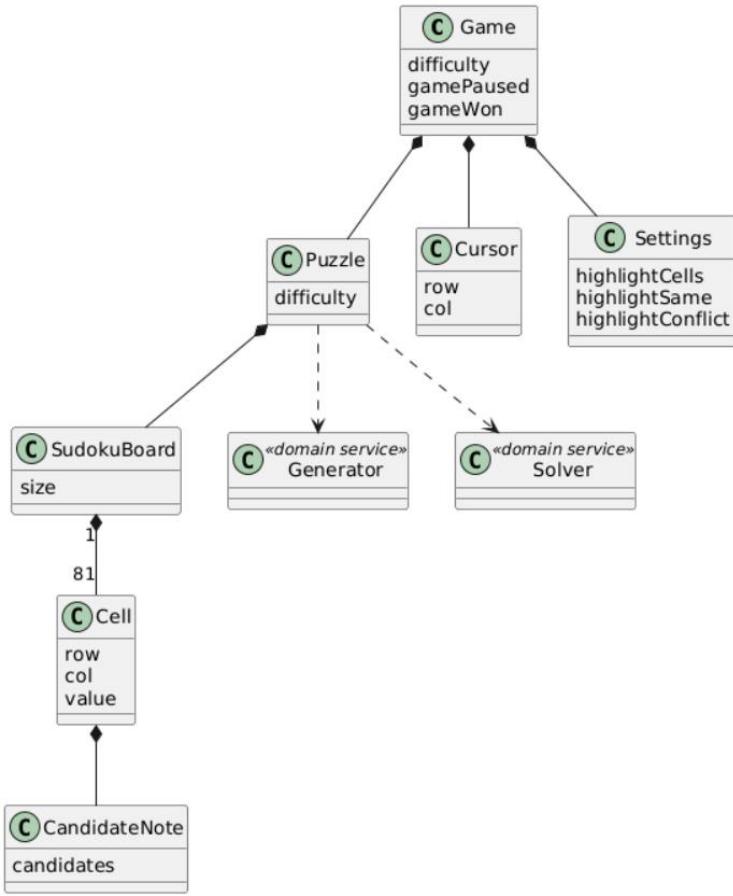
(5) **CandidateNote** 表示玩家为某个单元格记录的候选数字集合，用于辅助逻辑推理。该概念不会直接影响数独的最终解，但在解题过程中具有重要的辅助作用。

(6) **Cursor** 用于表示当前被玩家选中的单元格位置，从而明确后续操作的作用对象。

(7) **Settings** 描述玩家对游戏行为的个性化配置，例如是否开启高亮提示、是否限制提示次数等。这些设置会影响系统在交互过程中的表现方式。

(8) **Timer** 用于记录游戏的时间信息，描述游戏开始、暂停与恢复过程中的时间变化。它为游戏统计与完成判定提供基础支持。

(9) **Solver** 与 **Generator** 分别用于描述系统中数独求解与题目生成相关的功能模块，为题目初始化和提示功能提供支持。



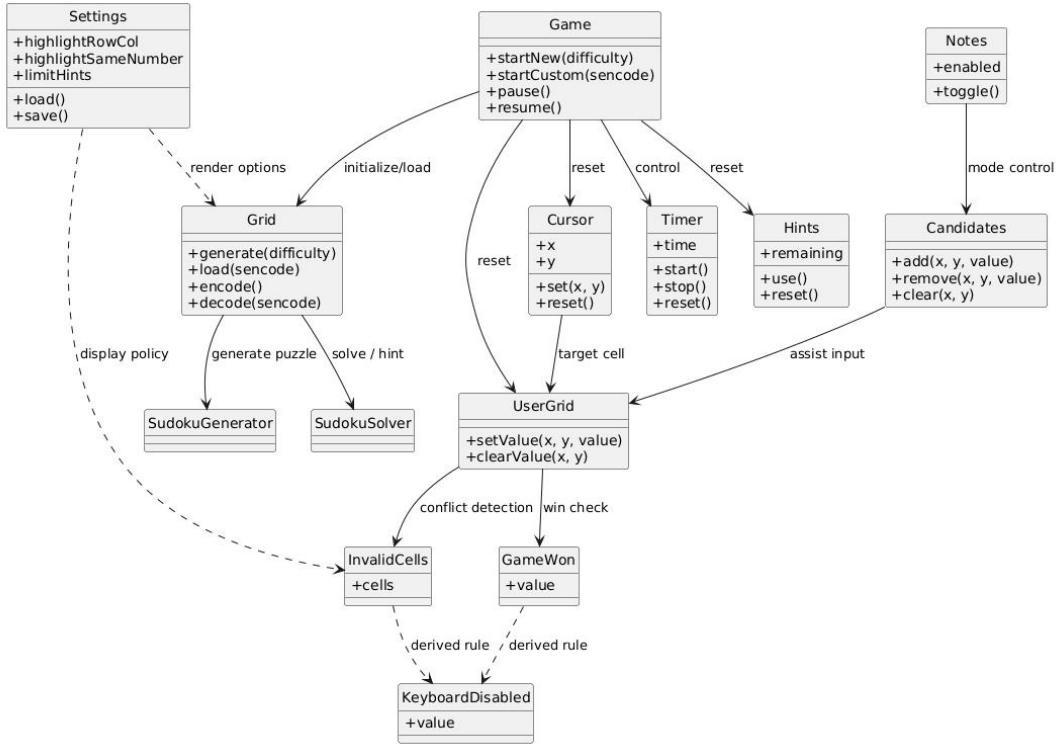
3. 面向对象设计

3.1 技术架构设计

从整体设计角度来看，该系统采用以前端为核心的分层式架构。虽然在实现层面并未严格遵循传统软件工程中的三层或四层架构划分，但从职责分离的角度出发，仍然可以清晰地识别出不同层次的功能定位。

- (1) 表示层主要负责与玩家进行直接交互，承担界面展示和用户操作响应的职责。通过棋盘、控制区域和弹窗等界面元素，系统将内部状态直观地呈现给玩家，并将玩家的操作转化为系统可以理解的交互请求。
- (2) 状态层用于集中管理游戏过程中产生的各种状态信息，包括棋盘内容、当前选中位置、候选笔记、计时信息以及游戏设置等。通过统一的状态管理机制，系统能够保证界面展示与内部数据的一致性，实现状态变化驱动界面更新的效果。
- (3) 领域与用例层负责组织和协调游戏过程中的核心业务流程，例如开始新游戏、加载题目以及暂停和继续游戏等操作。该层在逻辑上位于界面层与底层算法之间，使业务规则不会直接分散在界面逻辑中。
- (4) 算法适配层主要承担数独题目生成与求解相关的职责，为系统提供必要的算法支持。通过将具体算法实现与上层逻辑隔离，可以在不影响整体架构的前提下对算法进行替换或优化。
- (5) 基础设施层用于处理与运行环境相关的技术细节，例如本地数据存储等功能，从而避免这些细节对业务逻辑产生直接影响。

3.2 对象模型



4. 设计模式

1. 观察者模式

该项目以“状态驱动界面”为设计理念：当游戏状态发生变化时，界面能够自动同步更新，而无需在每个交互事件后手动刷新 UI。从设计模式角度看，这种“状态发布—界面订阅”的机制符合观察者模式的基本结构：状态对象在内部数据改变时通知订阅者，订阅者据此更新自身表现。由于数独棋盘、候选笔记、计时、提示等都属于频繁变化的状态，采用观察者模式能够显著降低界面与状态之间的耦合，使系统在交互密集的情况下依然保持逻辑清晰与响应及时。

2. 工厂模式

在实现层面，该项目并未采用大量传统面向对象的“类—实例”结构，而是通过工厂函数的方式创建并封装状态对象。这种写法通常表现为由一个创建函数返回对外接口，对外只暴露订阅能力与少量必要操作，而将内部细节（例如计时器的内部计时逻辑、状态更新细节等）隐藏在闭包中，从而形成模块化封装。其优势在于：既保留了“对象拥有状态与行为”的面向对象特征，又避免了在前端小型项目中引入复杂的类继承体系，使代码更加轻量、可读，并便于拆分与复用。

5. 评价

结合课程中关于高内聚、低耦合以及可扩展设计的相关原则，可以对该系统现有的面向对象设计进行评价。

5.1 优点

从整体上看，该系统在当前功能规模下具备较好的结构合理性。系统通过将不同类型的状态划分为独立的状态对象，使各类数据各司其职，职责边界相对清晰。例如，棋盘数据、光标位置、候选笔记、计时信息以及用户设置等均由不同

的状态模块分别管理，这种划分方式有助于提高系统的可理解性与可维护性。同时，系统利用派生状态来表达业务规则，例如冲突检测与通关判定等逻辑由已有状态关系自动推导得出，使规则表达更加直观，避免了大量显式判断逻辑散落在界面代码中。此外，系统对设置持久化等基础设施细节进行了封装，使界面层无需关心具体的存储实现，从而在一定程度上实现了界面逻辑与运行环境的解耦。

5.2 不足

系统仍然存在一些结构性不足。首先，与棋盘相关的数据与规则被分散在多个状态对象中，缺乏一个统一的核心领域对象来承担聚合职责，这在功能扩展或规则复杂化时可能导致理解成本和维护成本的上升。其次，部分用例级别的业务流程并未完全集中管理，而是需要界面组件直接协调多个状态对象完成操作，这在一定程度上提高了组件与状态之间的耦合度，不利于系统规模扩大后的演化。此外，当前的状态组织方式并未为操作历史保留清晰抽象，这使得撤销、重做等高级交互功能的实现难度较高。最后，由于部分领域规则分布在状态更新逻辑与界面交互中，系统在进行单元测试时难以将核心逻辑完全剥离为纯函数，从而限制了测试覆盖范围。

6. 改进建议

首先，在领域建模层面，建议引入更加明确的核心领域对象，对棋盘及其相关规则进行统一聚合。通过将当前分散在多个状态对象中的棋盘数据、候选信息和规则校验逻辑集中到一个领域对象中，可以更好地维护系统不变量，提高数据一致性，并降低跨模块协作的复杂度。这种做法有助于提升系统的内聚性，使棋盘相关逻辑在概念和结构上更加清晰。

其次，为了支持更复杂的交互需求，例如撤销与重做操作，可以引入命令模式对用户行为进行抽象。通过将“填写数字”“清空单元格”“编辑候选笔记”等操作封装为独立的命令对象，并为其提供可逆操作接口，系统可以在不破坏现有结构的前提下实现操作历史管理，从而显著增强交互灵活性。

在用例组织层面，建议进一步强化用例层的职责，将“开始游戏、输入数字、使用提示、暂停与继续”等完整业务流程集中由统一的服务模块进行管理。界面层只负责触发用例请求并展示结果，而不直接操作多个状态对象，从而降低界面层与状态层之间的耦合度，使系统结构更符合“面向用例设计”的思想。

此外，在数据结构与类型设计方面，也存在一定的优化空间。例如，将候选数字集合从数组形式替换为集合结构，可以提高操作语义的清晰度并减少潜在错误；使用更明确的位置表示方式替代字符串形式的坐标，也有助于提升数据一致性与可读性。在条件允许的情况下，引入更严格的类型约束机制，有助于在编译或开发阶段提前发现潜在问题。

最后，从软件质量保障的角度出发，建议逐步引入针对核心领域逻辑的单元测试。通过对规则校验、通关判定以及操作回滚等关键逻辑进行测试，可以在不显著增加系统复杂度的前提下提升整体可靠性，为后续功能扩展提供更稳定的的基础。