数独项目逆向分析

1、设计思想、设计原则和设计模式

1.1、设计思想

该项目采用了 组件化 和 模块化 的设计思想，结合 Svelte 框架 的响应式特性，简化了 UI 渲染和状态管理。首先，组件化将界面划分为多个独立的组件（例如 `Board`、 `Controls`、`Header`、`Modal`）。每个组件负责特定的功能，提高了代码的复用性和可维护性。此外，该项目还使用Svelte Store 管理全局状态（例如 `grid`、`candidates`、`timer`、 `hints`），通过 Store 实现组件之间的状态共享和同步。另外项目还使用了分层结构，整个项目被分为UI 层（组件）、逻辑层（Store 和核心逻辑文件）和 数据层（`constants.js`）。清晰的分层有助于分离关注点，便于维护和扩展。

1.2、 设计原则

a、项目满足单一职责原则（SRP），每个组件和Store负责单一功能，例如Board 组件负责显示数独网格，candidates Store 负责管理候选数字。

b、开闭原则（OCP），系统易于扩展，但不易于修改。例如，可以轻松添加新的提示功能或设置选项，而无需修改现有代码。

c、依赖倒置原则（DIP），高层模块（如 UI 组件）依赖于抽象（Store），而不是具体的实现。这有助于保持灵活性和可测试性。

d、DRY 原则，公共逻辑被抽象为 Store 和模块函数，避免了代码重复。

1.3、设计模式

a、**观察者模式：**Svelte Store 本身类似观察者模式，视图层订阅 store变化。

b、**模块化与函数式风格**：更多是函数集与简单对象的模块化，而非严格面向对象的设计模式。例如 `game.js` 提供统一的游戏控制接口。

项目没有明显采用经典的 GoF 面向对象设计模式（如策略、工厂、单例）进行抽象，但通过 store 的订阅机制实现界面与数据的解耦，使用 store 类似于观察者/发布-订阅模式。

2、愿景、用例分析、领域模型、技术架构与对象模型

2.1 愿景

愿景：开发一个交互性强、功能完整的在线数独游戏，支持自定义数独、提示功能、候选数字标记、游戏设置和分享功能，为用户提供良好的游戏体验。

2.2 用例分析

1. **开始新游戏**：选择难度并生成新的数独网格。
2. **输入数字**：在单元格中输入数字或候选数字。
3. **使用提示**：在非笔记模式下，自动填充正确的数字。
4. **暂停/继续游戏**：暂停或恢复游戏计时。
5. **分享数独**：生成分享链接或二维码，分享给他人。
6. **游戏设置**：调整游戏选项，如定时器显示、候选数字高亮。
7. **检测获胜状态**：检查当前数独是否已完成且无错误。

2.3 领域模型

SudokuGame（隐性存在）：

- 属性：难度、游戏状态(暂停/进行中)、用户网格状态、提示次数、计时器状态、光标位置、候选数字集合。
- 行为：开始新游戏、加载自定义游戏、暂停、恢复、判断胜利、应用提示、生成和求解数独等。

Grid：代表初始给出的数独题面。

UserGrid：表示用户在初始题面基础上填写的数字状态。

Candidates：每个单元格的候选数字集。

Cursor：当前选中单元格的位置。

Difficulty：当前游戏的难度级别。

Hints：提示次数管理。

Timer：计时器管理游戏用时。

Modal：模态窗口，用于显示设置、分享、结束提示等。

2.4技术架构

前端框架：Svelte

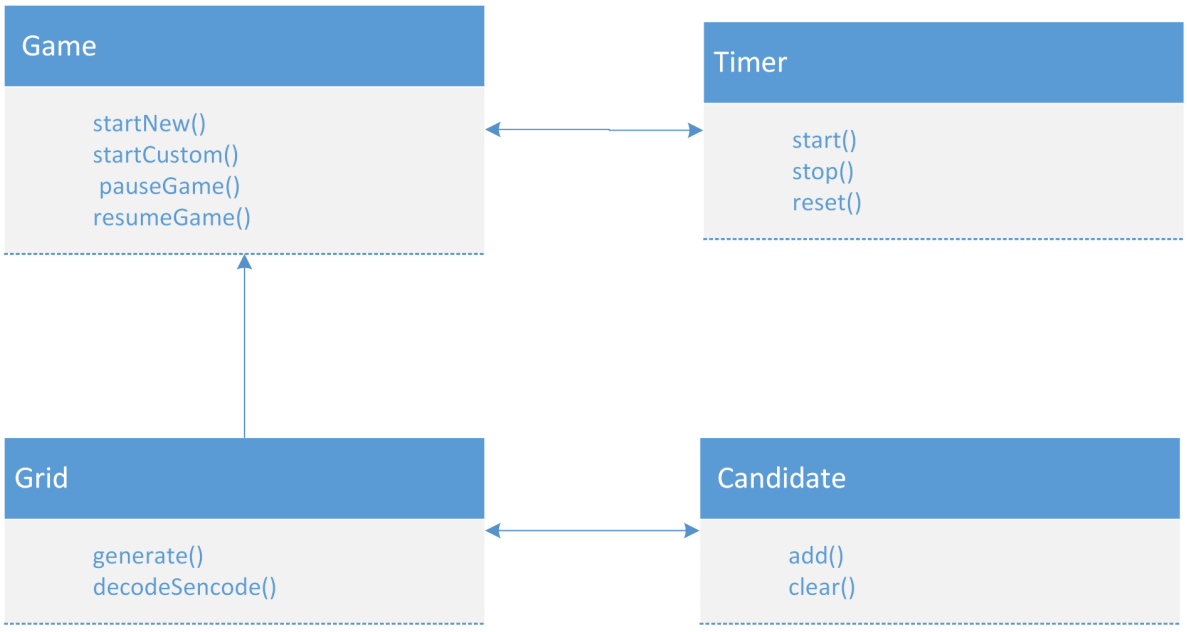
状态管理：Svelte Store

UI 组件：基于 Svelte 的组件化架构

核心逻辑：JavaScript 模块（如 `sudoku.js`、`game.js`）

第三方库：`@mattflow/sudoku-solver`：求解数独；`fake-sudoku-puzzle-generator`：生成数独。

2.5对象模型



3、现有OOD架构的优劣与改进建议

优点：

- **简单直观**：代码逻辑相对简洁，易于理解。每个 store 管理一个特定的领域概念状态，无过度复杂的类继承关系。
- **灵活扩展**：通过新增 store 和组件文件，可以快速添加新功能或修改现有逻辑。
- **数据驱动UI**：Svelte Store与组件双向绑定，UI自动随数据变化更新，降低耦合。

缺点：

- **缺乏明显的面向对象抽象**：没有类与接口，难以通过多态或继承来拓展功能。对新需求的扩展可能要求在全局范围搜索替换逻辑。
- **业务逻辑分散**：逻辑分布在多个 store、组件与函数文件中，没有统一的领域模型类，导致逻辑理解需要在多文件间跳转。
- **可测试性与可维护性欠佳**：缺少清晰的分层架构和抽象接口，不易为各独立模块进行单元测试与 Mock。

改进建议：

引入更面向对象的抽象层：

- 可考虑将与数独逻辑（如求解、生成、验证）的代码封装成独立的类，例如 `SudokuPuzzle` 类，包含 `generate()`, `solve()`, `validate()` 等方法。
- 将候选数字、用户输入、提示和计时器抽象为服务类，通过接口与实现分离，方便日后更换实现或增加策略。

使用依赖注入或更清晰的模块边界：

- 将游戏逻辑（game.js）中的函数改造成类的实例方法，构造时传入 `GridService`、`TimerService`、`HintService` 等依赖，有助于测试与拓展。

采用更丰富的设计模式：

- 对提示逻辑引入 **策略模式(Strategy)**：不同难度下的自动建议策略不同。
- 对求解、生成数独可以有不同算法实现，通过接口注入，以 **抽象工厂(Factory)** 或 **策略模式** 实现可替换的求解引擎。

分层架构：

- 将UI、逻辑与数据存储层分层更清晰。
- Store可作为数据观察层，逻辑进一步封装在服务对象中，从而降低 Store 中的业务逻辑负担。