数独游戏 OOAD 逆向分析

一、项目概述

1.1 项目背景

本项目是一个基于Web的数独游戏应用，源自GitHub开源项目 jonasgeiler/sudoku。系统使用Svelte框架和TailwindCSS构建，提供了完整的数独游戏功能，包括多难度谜题生成、填写验证、候选数标记、提示功能、计时器、游戏分享等核心功能。

项目地址：https://sudoku.jonasgeiler.com

1.2 技术栈概览

- **前端框架**：Svelte
- **样式框架**：TailwindCSS
- **构建工具**：Rollup
- **核心算法库**：
  - fake-sudoku-puzzle-generator (谜题生成)
  - @mattflow/sudoku-solver (数独求解)

二、面向对象分析 (OOA)

2.1 系统愿景

构建一个轻量级、易用、功能完整的在线数独游戏平台，为不同水平的玩家提供流畅的游戏体验。系统支持多难度级别、智能提示、冲突检测、游戏分享等功能，并通过模块化设计为未来扩展（如撤销/重做、自定义谜题等）预留空间。

核心价值：

- **便捷性**：基于Web，无需安装，随时随地可玩
- **挑战性**：提供从"非常简单"到"困难"的四个难度级别
- **辅助性**：提供提示、候选数、冲突高亮等辅助功能
- **可分享性**：支持通过编码分享游戏状态

2.2 用例分析

2.2.1 核心用例

用例1：开始新游戏

- **参与者**：玩家
- **前置条件**：无
- **主要流程**：
  1. 玩家选择难度级别（Very Easy/Easy/Medium/Hard）
  2. 系统根据难度生成数独谜题

3. 系统初始化游戏状态（重置计时器、提示次数、光标位置）
4. 系统显示数独棋盘

- **后置条件**：新游戏开始，计时器启动

用例2：填写数字

- **参与者**：玩家
- **前置条件**：游戏已开始，玩家已选中单元格
- **主要流程**：
  1. 玩家选择空白单元格
  2. 玩家输入1-9的数字
  3. 系统验证输入是否与行/列/宫格存在冲突
  4. 系统显示数字并标记冲突（如有）
- **后置条件**：单元格被填写，可能触发冲突提示

用例3：使用提示功能

- **参与者**：玩家
- **前置条件**：游戏已开始，提示次数未用尽
- **主要流程**：
  1. 玩家选择需要提示的单元格
  2. 玩家点击提示按钮
  3. 系统求解当前数独
  4. 系统填入该位置的正确答案
  5. 系统减少可用提示次数
- **后置条件**：单元格被填入正确答案

用例4：标记候选数

- **参与者**：玩家
- **前置条件**：游戏已开始，玩家选中空白单元格
- **主要流程**：
  1. 玩家进入候选数标记模式
  2. 玩家选择候选数字
  3. 系统在单元格中显示/隐藏候选数
- **后置条件**：候选数被更新

用例5：完成游戏

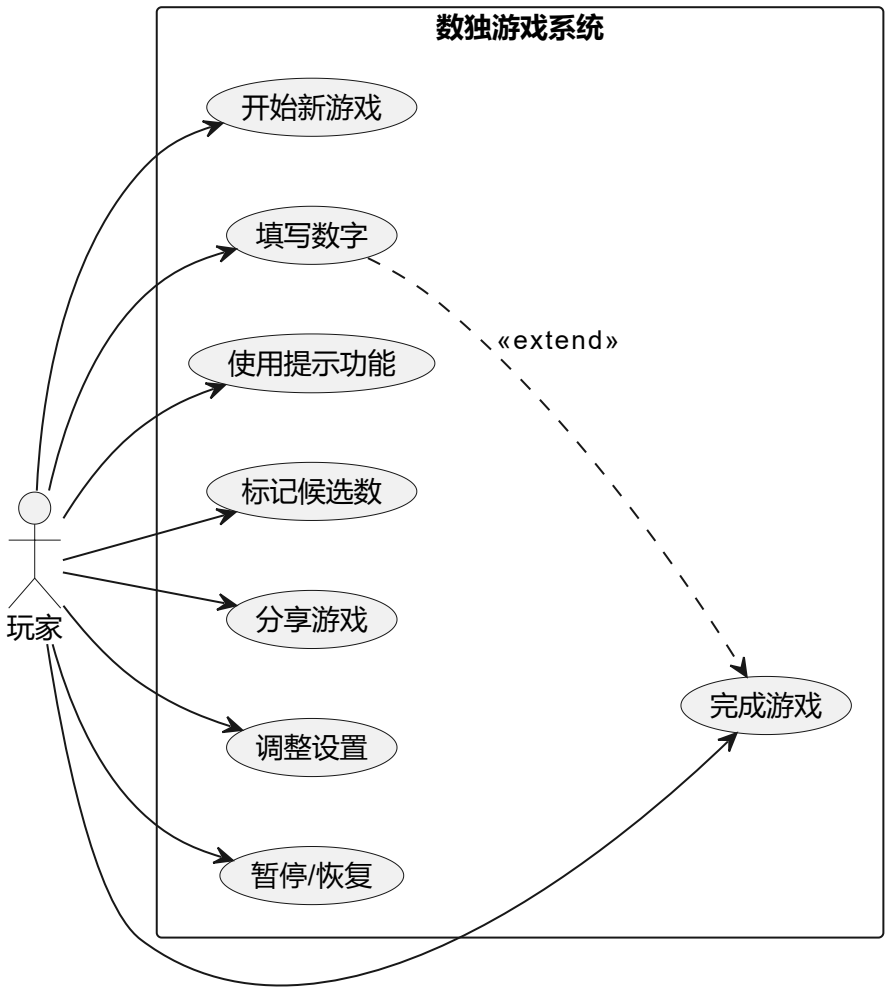
- **参与者**：系统（自动触发）
- **前置条件**：所有单元格已填写
- **主要流程**：
  1. 系统检测所有单元格无冲突
  2. 系统停止计时器
  3. 系统显示完成提示和用时
- **后置条件**：游戏结束

用例6：分享游戏

- **参与者**：玩家

- **前置条件**：游戏已开始
- **主要流程**：
    1. 玩家点击分享按钮
    2. 系统将当前游戏状态编码为字符串
    3. 系统生成分享链接或二维码
    4. 玩家通过链接分享给他人
- **后置条件**：分享链接生成

2.2.2 用例图



2.3 领域模型

领域模型描述数独游戏的核心业务概念、业务属性及其之间的逻辑关系。

2.3.1 核心领域概念

1. 数独游戏 (SudokuGame)

- 表示一局完整的数独游戏会话
- 核心属性：
    - difficulty: 游戏的难度（非常简单/简单/中等/困难/自定义）

    - state: 当前状态（进行中/已暂停/已完成）
    - startTime: 游戏开始的时间点
    - elapsedTime: 玩家已花费的时间
    - isPaused: 标识游戏是否处于暂停状态

2. 数独谜题 (Puzzle)

- 表示一个数独题目的初始状态
- 核心属性：
    - difficulty: 该谜题的难度
    - initialLayout: 题目给定的固定数字及其位置
    - solution: 该谜题的标准答案

3. 数独棋盘 (Board)

- 表示数独游戏的当前状态
- 核心属性：
    - size: 9×9的网格
    - cells: 包含的所有单元格
    - filledCount: 已填写的单元格数量

4. 单元格 (Cell)

- 表示棋盘上的一个格子
- 核心属性：
    - row: 所在行（0-8）
    - column: 所在列（0-8）
    - box: 所属的3×3宫格（0-8）
    - value: 填写的数字（0表示空，1-9表示数字）
    - isFixed: 是否为题目给定的固定数字
    - candidates: 玩家标记的可能数字列表

5. 候选数 (Candidate)

- 表示单元格中标记的候选数字
- 核心属性：
    - value: 1-9之间的数字
    - cell: 该候选数所在的单元格

6. 提示记录 (HintUsage)

- 表示提示功能的使用情况
- 核心属性：
    - remaining: 剩余可使用的提示次数
    - used: 已经使用的提示次数
    - limited: 是否限制提示次数

7. 游戏设置 (GameSettings)

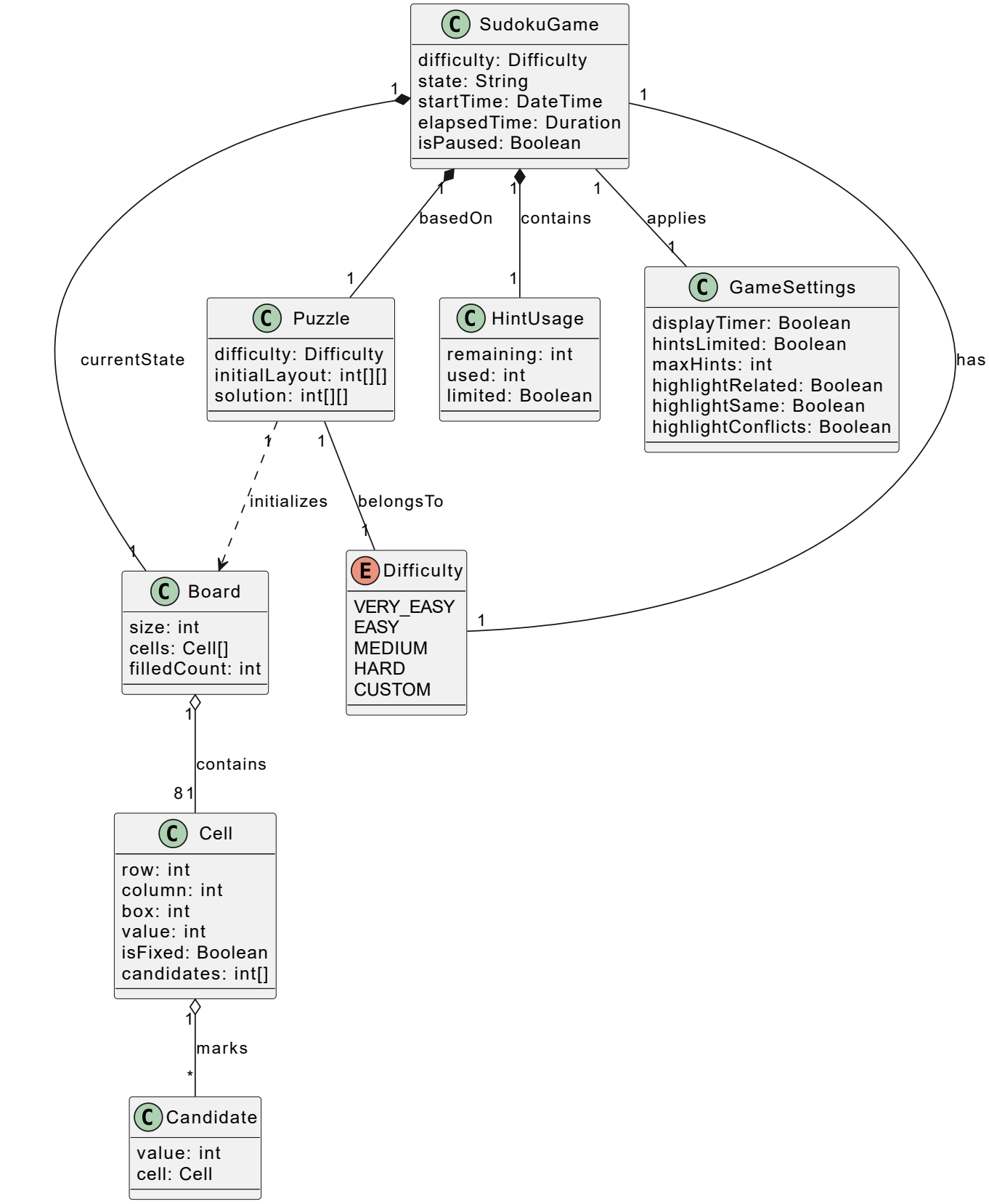
- 表示玩家的个性化配置
- 核心属性：

- displayTimer: 是否显示计时器
- hintsLimited: 是否限制提示次数
- maxHints: 提示次数上限
- highlightRelated: 是否高亮相关单元格
- highlightSame: 是否高亮相同数字
- highlightConflicts: 是否高亮冲突单元格

8. 难度级别 (Difficulty)

- 表示数独的难度等级
- 可选值：非常简单(VERY_EASY)、简单(EASY)、中等(MEDIUM)、困难(HARD)、自定义(CUSTOM)

2.3.2 领域模型类图



三、面向对象设计 (OOD)

3.1 系统技术架构

从业务逻辑角度，系统采用分层架构，核心分为以下几层：

1. 核心算法层

- **数独生成**：调用第三方库fake-sudoku-puzzle-generator，根据难度生成谜题
- **数独求解**：调用第三方库@mattflow/sudoku-solver，提供求解能力
- **规则验证**：实现数独规则检测逻辑，识别冲突单元格

2. 业务逻辑层

- **游戏控制**：管理游戏生命周期（开始、暂停、恢复、完成）
- **棋盘管理**：维护初始谜题和用户填写状态
- **提示管理**：控制提示次数和提示应用
- **编码服务**：实现棋盘的编码和解码

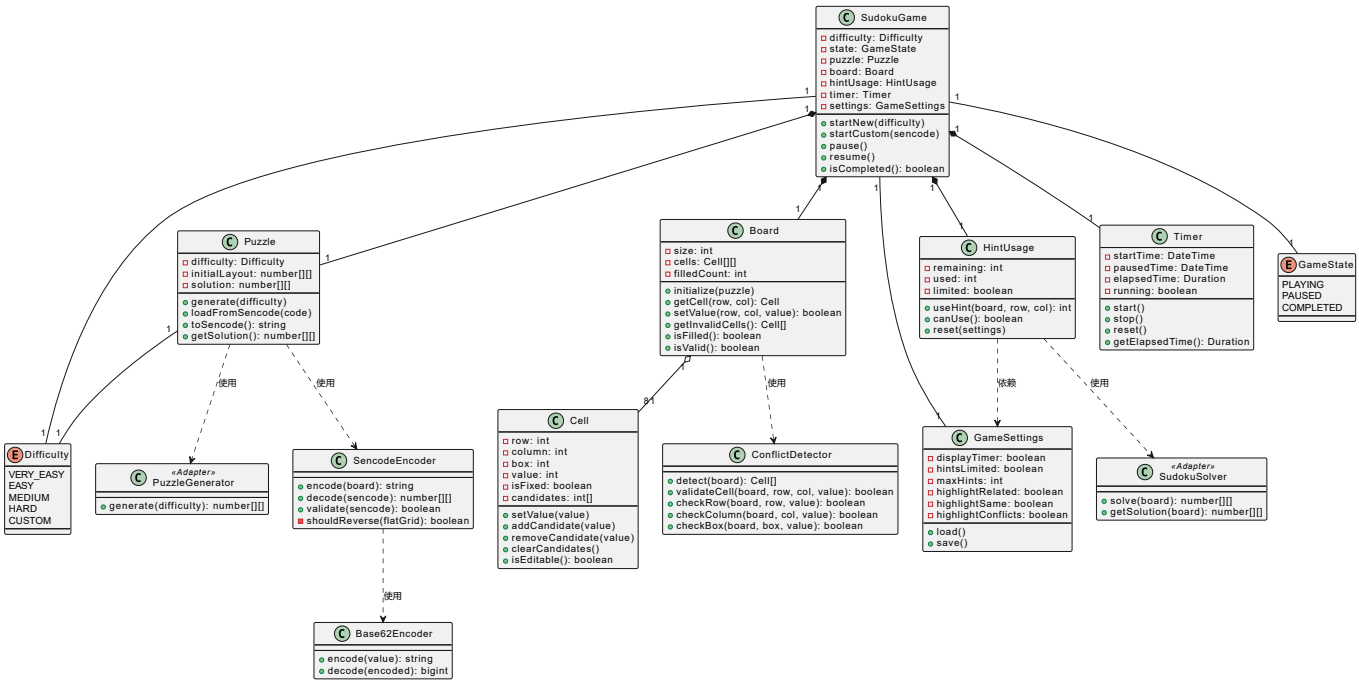
3. 工具服务层

- **计时服务**：处理游戏计时逻辑
- **设置管理**：管理用户个性化配置
- **候选数管理**：维护候选数标记

架构特点：

- **算法解耦**：核心算法依赖第三方库，便于替换
- **职责清晰**：每个模块负责明确的业务功能
- **易于扩展**：分层设计便于添加新功能

3.2 对象模型



3.3 设计评价

3.3.1 设计原则分析

1. 单一职责原则 (Single Responsibility Principle)

- SudokuGame专注于游戏会话管理，Board专注于棋盘状态，Cell专注于单元格数据，职责划分清晰
- ConflictDetector单独负责规则验证，PuzzleGenerator专注谜题生成，SencodeEncoder负责编解码

优点：类的职责清晰，易于理解和维护

2. 开闭原则 (Open-Closed Principle)

- 难度级别通过Difficulty枚举定义，新增难度无需修改现有代码，符合开闭原则
- PuzzleGenerator和SudokuSolver使用适配器模式，可以替换不同的算法实现而无需修改客户端代码

不足：

- Board的验证逻辑直接调用ConflictDetector，如果需要支持不同的验证策略（如宽松模式、严格模式），需要修改Board类
- 缺少对验证规则的抽象，扩展性有限

3. 接口隔离原则 (Interface Segregation Principle)

- 各服务类（ConflictDetector、PuzzleGenerator、SudokuSolver、SencodeEncoder）提供的接口简洁明确
- 客户端只依赖其需要的方法，如HintUsage只使用SudokuSolver的solve方法

优点：接口设计精简，避免了接口污染

4. 依赖倒置原则 (Dependency Inversion Principle)

不足：

- Puzzle、Board、HintUsage等高层模块直接依赖具体的服务类（PuzzleGenerator、ConflictDetector、SudokuSolver）
- 缺少抽象接口层，导致高层模块与低层实现紧耦合
- 如果需要更换算法实现，需要修改多处代码

改进空间：应该定义抽象接口（如IGenerator、ISolver、IValidator），让高层模块依赖抽象而非具体实现

3.3.2 设计模式分析

1. **外观模式**：SudokuGame作为外观，统一协调各子对象，简化了客户端调用，隐藏了内部复杂性，应用效果良好
2. **适配器模式**：PuzzleGenerator和SudokuSolver成功隔离了第三方库依赖，统一了接口，便于替换实现
3. **策略模式**：难度级别的处理体现了策略模式思想，不同难度对应不同的生成策略

3.4 改进建议

3.4.1 功能改进

1. 实现撤销/重做功能

- 引入命令模式，将每个用户操作封装为命令对象
- 设计Command接口，包含execute()和undo()方法
- 实现具体命令类：SetCellCommand、ClearCellCommand等
- 维护命令历史栈（undoStack和redoStack）
- 支持undo()和redo()操作，增强用户体验

2. 添加游戏进度保存

- 引入备忘录模式（Memento Pattern）保存游戏状态
- 设计Memento类封装游戏快照
- Caretaker负责管理备忘录的存储和恢复
- 支持自动保存和手动保存
- 实现"继续上次游戏"功能

3. 支持自定义谜题编辑

- 引入建造者模式（Builder Pattern）构建自定义谜题
- 设计PuzzleBuilder提供流畅的构建接口
- 实现验证逻辑确保谜题的有效性和唯一解
- 支持从编辑模式直接开始游戏

3.4.2 设计原则改进

1. 依赖倒置原则

问题：高层模块直接依赖具体实现

改进方案：

- 定义抽象接口层：
  - IGenerator接口：定义谜题生成契约
  - ISolver接口：定义求解器契约
  - IValidator接口：定义验证器契约
- 高层模块（Puzzle、Board、HintUsage）依赖抽象接口
- 具体实现类实现对应接口
- 通过依赖注入或工厂模式创建具体实例

优点：

- 降低模块间耦合
- 便于替换算法实现
- 提高可测试性

2. 开闭原则

问题：扩展验证规则需要修改Board类

改进方案：

- 定义IValidationRule接口
- 实现多种验证规则：RowRule、ColumnRule、BoxRule、DiagonalRule等
- Board维护验证规则列表
- 通过配置添加或移除验证规则，无需修改Board代码

优点：

- 符合开闭原则（对扩展开放，对修改封闭）
- 支持不同数独变体（标准数独、对角线数独、不规则数独等）

- 提高系统灵活性

3. 单一职责原则

问题：SudokuGame承担过多职责

改进方案：

- 引入GameStateManager专门管理游戏状态转换
- 引入GameTimer专门处理计时逻辑
- SudokuGame作为纯粹的协调者（外观）
- 进一步细化职责，每个类只做一件事

优点：

- 提高类的内聚性
- 便于单独测试和维护
- 符合单一职责原则

3.4.3 设计模式改进

1. 引入状态模式管理游戏状态

问题：游戏状态转换逻辑分散，缺少集中管理

改进方案：

- 定义GameState抽象类或接口
- 实现具体状态类：PlayingState、PausedState、CompletedState
- 每个状态类封装该状态下的行为
- 状态转换逻辑封装在状态类内部

优点：

- 消除大量条件判断语句
- 状态转换规则集中管理
- 易于添加新状态
- 符合开闭原则

2. 引入组合模式处理单元格组

问题：处理行、列、宫格时代码重复

改进方案：

- 定义CellComponent接口
- Cell实现为叶子节点
- CellGroup（行、列、宫格）实现为组合节点
- 统一的接口处理单个单元格和单元格组

优点：

- 简化客户端代码

- 消除重复逻辑
- 统一处理部分-整体层次结构

3. 引入模板方法模式优化验证流程

问题：ConflictDetector的检查流程存在重复

改进方案：

- 定义抽象的ValidationTemplate
- 定义模板方法：validate() = preCheck() + checkRule() + postCheck()
- 子类实现具体的检查规则（checkRow、checkColumn、checkBox）
- 固定算法骨架，灵活变化细节

优点：

- 代码复用性高
- 算法结构清晰
- 易于维护和扩展