|  |
| --- |
| **《面向对象程序设计》课程设计报告** |
| **各组员班级姓名：**  **人23级1班刘承晧**  **人23级1班林伟煌**  **人23级1班常艺豪** |
| **分工及贡献比例**  **刘承晧（组长）设计类间关系，定义主要接口，数独游戏的内部逻辑、数独控制类、一部分工具类的设计与实现 50%**  **林伟煌 数独游戏的输入输出部分，一部分工具类的设计与实现25%**  **常艺豪 数独游戏的文件相关部分的设计与实现，包括文件处理接口类25%** |
| **需求分析（分析系统的使用者，从使用者角度确定系统的外部接口和用例模型，描述系统的结构、功能、活动、性能等，确定系统的软件需求）**  **一、系统使用者分析**  **在设计和开发本数独游戏系统之前，首先需要明确系统的目标用户，以确保系统能够满足他们的需求。系统的使用者主要包括：**   * **数独玩家：对数独感兴趣的玩家，希望通过游戏放松娱乐，提升逻辑思维能力；或是希望挑战不同难度的数独谜题，提高解题技巧** * **开发者和扩展者：希望在现有系统基础上进行二次开发，添加新功能或移植到其他平台**   **二、系统外部接口和用例模型**  **从使用者的角度出发，我们确定了系统的外部接口和用例模型，以满足不同用户的需求。**  **1.外部接口**   * **用户界面接口：**   1. **控制台界面：通过命令行进行交互，适用于喜欢简洁界面的用户**   2. **图形用户界面（未来扩展）：通过窗口、按钮等进行交互，提供更直观的游戏体验** * **文件接口：**   1. **题库加载接口：支持从外部文件加载数独谜题，方便用户选择不同的谜题**   2. **游戏存档接口：支持保存和加载游戏进度，方便用户中途退出后继续游戏**   **2.用例模型**  **用例1：开始新游戏**   * **参与者：玩家** * **描述：玩家选择开始新游戏，系统提供一个新的数独谜题** * **流程：**   1. **玩家启动.exe文件**   2. **玩家选择一个题目编号**   3. **系统加载并显示数独棋盘**   4. **玩家与系统经行进一步的交互**   **用例2：输入数字**   * **参与者：玩家** * **描述：玩家在棋盘上填写数字** * **流程：**   1. **玩家输入填写数字对应的命令**   2. **输入行、列、填入数**   3. **系统验证输入的合法性（如是否与已有数字冲突）。**   4. **提示是否成功并更新棋盘显示**   **用例3：去除候选数**   * **参与者：玩家** * **描述：玩家在单元格中去除已有的候选数** * **流程：**   1. **玩家输入去除候选数对应的命令**   2. **输入行、列、候选数**   3. **系统内部进行对应操作**   4. **更新棋盘显示**   **用例4：撤销/重做操作**   * **参与者：玩家** * **描述：玩家希望撤销或重做之前的操作** * **流程：**   1. **输入选择“撤销”或“重做”对应的命令**   2. **系统检查操作是否合法**   3. **系统更新棋盘到相应的状态。**   4. **更新棋盘显示**   **用例5：保存游戏**   * **参与者：玩家** * **描述：玩家保存当前游戏进度** * **流程：**   1. **玩家选择“保存游戏”对应的命令**   2. **系统执行相应的文件操作，保存或加载游戏数据。**   **三、系统结构、功能、活动和性能描述**   1. **系统结构**   **系统采用分层架构，主要由以下模块组成：**  **1.模型层（Model）：实现数独游戏的核心逻辑和数据结构**   * + **Sudoku 类及其相关的 Cell、Row、Column、Block 等类**   **2.视图层（View）：负责与用户的交互，显示游戏界面和获取用户输入**   * + **IOInterface 接口及其实现类 ConsoleIO**   **3.控制器层（Controller）：协调模型和视图，处理用户输入并更新模型状态**   * + **SudokuController 类**   **4.应用程序层：封装整个游戏，初始化和启动游戏应用**   * + **GameApplication 类**   **5.工厂模式组件：创建系统的各个组件，支持扩展**   * + **GameComponentsFactory 接口及其实现类**  1. **系统功能**  * **游戏逻辑功能：**   + **初始化和生成数独棋盘**   + **检查数字输入的合法性和冲突检测**   + **判断游戏是否完成**   + **提供候选数支持**   + **支持撤销和重做操作** * **用户交互功能：**   + **显示数独棋盘和相关信息**   + **获取用户输入（数字填入、候选数、命令等）**   + **提供菜单选项，供用户选择操作** * **数据持久化功能：**   + **从文件加载数独谜题**   + **保存和加载游戏进度** * **辅助功能：**   + **提供自动补全候选数。**   + **记录和显示游戏时间和步长**  1. **系统活动**  * **游戏初始化活动：**   + **系统启动时，初始化各个组件**   + **加载指定的数独谜题** * **游戏进行活动：**   + **循环等待用户输入**   + **根据用户操作更新内部游戏状态**   + **实时更新显示，反馈用户操作结果** * **游戏结束活动：**   + **当数独被正确解出时，显示胜利信息**   + **提供重启游戏或退出选项**  1. **系统性能**  * **响应速度：系统应在用户输入后即时响应，更新棋盘显示，提供流畅的游戏体验** * **稳定性：系统应能稳定运行，避免崩溃或未处理的异常** * **资源消耗：由于系统主要运行在控制台环境，资源消耗较低，应能在大多数计算机上运行** |
| **系统设计（确定出系统的主要组成元素，各组成元素的职责和边界定义，以及各组成元素间的交互和协作关系）**  **一、总体架构**  **本系统采用了面向对象的设计思想，结合了模型-视图-控制器（MVC）模式、抽象工厂模式和命令模式，构建了一个高内聚、低耦合、易于扩展的数独游戏程序。系统主要由以下五大部分组成：**   1. **数独内部逻辑（模型）** 2. **输入输出（视图）** 3. **控制器（控制器）** 4. **文件处理** 5. **工厂与应用程序初始化**   **下面是这五大部分下的系统的主要组成元素介绍。**  **二. 数独内部逻辑（模型）**  **负责数独游戏的核心逻辑，实现游戏的状态管理和业务规则。**   * **Cell 类：表示数独中的一个单元格，包含以下属性：**   + **value：当前数值**   + **candidates：候选值集合**   + **fixed：标识是否为初始提示的固定单元格**   + **提供查看和修改单元格状态的接口** * **CellContainer 类：用于管理 9 个 Cell 对象的容器类，提供以下功能：**   + **添加、获取 Cell**   + **检查容器是否合法（无重复值）**   + **判断容器是否已完成**   + **重置容器中每个 Cell 的候选值** * **Row、Column、Block 类：继承自 CellContainer 类，分别表示数独中的行、列和九宫格。功能相同，但语义不同** * **Sudoku 类：作为数独游戏的模型层，聚合了 Cell、Row、Column、Block 等类，负责数独游戏的核心逻辑，包括：**   + **初始化棋盘**   + **设置单元格值**   + **添加/删除候选值**   + **检查游戏是否完成**   + **加载和保存游戏数据**   **三. 输入输出（视图）**  **负责与用户的交互，展示游戏界面并获取用户输入。**   * **IOInterface 类：抽象接口类，定义了输入输出的规范化接口，支持不同的输入输出方式。** * **ConsoleIO 类：继承自 IOInterface，实现了控制台的输入输出接口，负责：**   + **在控制台显示游戏信息（棋盘、提示等）**   + **获取用户的输入操作** * **职责和边界：视图层只负责游戏与用户的信息交互，不包含业务逻辑或游戏状态的管理**   **四. 控制器**  **负责协调模型和视图，处理用户输入，更新模型状态，并通知视图更新。**   * **SudokuController 类：作为控制器层，聚合了 Sudoku（模型）和 IOInterface（视图）对象，职责包括：**   + **游戏的初始化和启动**   + **处理用户输入，解析并执行相应的操作**   + **更新模型状态（Sudoku 对象）**   + **通知视图（IOInterface）更新显示**   + **采用了 MVC 模式，将模型、视图和控制器分离，增强了系统的可维护性和可扩展性** * **Command 类：抽象命令接口，定义了执行和撤销操作的纯虚函数 execute()，用于：**   + **将用户的操作封装为命令对象** * **具体命令类：继承自 Command 类，实现具体的游戏操作，如：**   + **设置单元格值命令**   + **添加候选值命令**   + **撤销操作命令等** * **OperationRecorder 类：用于记录用户的操作历史，提供：**   + **记录操作的功能**   + **支持撤销和重做操作**   **五、文件处理**  **负责数独游戏的数据加载和保存。**   * **PuzzleLoader 类：抽象接口类，定义了加载和保存数独棋盘的规范化接口，支持不同的存储格式。** * **PuzzleLoaderDAT 类：继承自 PuzzleLoader，实现了 DAT 格式的题目加载和保存，职责包括：**   + **从 DAT 文件中加载数独题目**   + **将当前游戏状态保存到 DAT 文件** * **职责和边界：文件处理类只负责游戏数据的加载和保存，与游戏逻辑和用户交互解耦。**   **5. 工厂与应用程序初始化**  **为了提高系统的灵活性和封装性，采用了抽象工厂模式，统一创建系统的各个组件。**   * **GameComponentsFactory 类：抽象工厂接口，定义了创建以下组件的抽象方法：**   + **输入输出接口（IOInterface）**   + **题目加载器（PuzzleLoader）**   + **数独游戏实例（Sudoku）**   + **控制器（SudokuController）** * **ConsoleGameComponentsFactory 类：继承自 GameComponentsFactory，实现了创建控制台版本组件的方法。** * **GameApplication 类：负责应用程序的初始化，职责包括：**   + **使用工厂创建所有必要的组件**   + **提供 startGame() 方法启动游戏**   + **封装组件的创建和初始化逻辑，简化 main 函数，增强封装性**   **六、各组成元素的职责和边界定义**   * **模型（Sudoku 及相关类）：负责游戏的核心逻辑和状态管理，不直接与用户交互** * **视图（IOInterface 及其实现）：负责与用户的交互，展示游戏界面和获取用户输入，不包含业务逻辑** * **控制器（SudokuController）：协调模型和视图，处理用户输入，更新模型状态，通知视图更新** * **文件处理（PuzzleLoader 及其实现）：负责游戏数据的加载和保存，与模型交互，不涉及用户交互** * **工厂（GameComponentsFactory 及其实现）：负责创建系统的各个组件，封装创建细节，提高系统的可扩展性和灵活性** * **应用程序（GameApplication）：负责应用程序的初始化和启动，封装组件的创建和初始化逻辑**   **七、各组成元素间的交互和协作关系**   1. **应用程序启动**    * **在 main 函数中，创建 GameApplication 对象并调用 startGame() 方法**    * **GameApplication在初始化时，使用工厂创建所需的组件，初始化游戏环境** 2. **组件创建**    * **GameApplication 使用 GameComponentsFactory 创建：**      + **输入输出接口（IOInterface）**      + **题目加载器（PuzzleLoader）**      + **数独实例（Sudoku）**      + **数独控制器（SudokuController）** 3. **游戏主循环**    * **GameApplication 的启动函数调用SudokuController的startGame函数启动游戏后，进入主循环，流程如下：**      + **获取用户输入：从 IOInterface 获取用户的输入操作**      + **解析用户输入：SudokuController 解析输入，对应到先前创建菜单中的相应 Command 对象**      + **执行命令：调用 Command 的 execute() 方法，更新 Sudoku（模型）的状态**      + **记录操作：使用 OperationRecorder 记录对值的修改操作，支持撤销和重做**      + **更新视图：SudokuController 通知 IOInterface 更新显示，展示最新的游戏状态** 4. **文件加载和保存**    * **Sudoku（模型）通过 PuzzleLoader（文件处理）加载和保存游戏数据**    * **PuzzleLoader 负责具体的文件操作，与模型逻辑解耦** |
| **系统设计优化（考虑系统的可维护、可复用、可扩展和灵活性等质量要素，说明你的设计如何适应未来需求变化）**  **为了提高系统的可维护性、可复用性、可扩展性和灵活性，我们在设计过程中深入考虑了未来可能的需求变化，采用了多种设计模式和原则，优化了系统的结构。以下是具体的优化措施：**  **1. 接口抽象与文件处理和输入输出的优化**  **针对可能的文件格式变化，我们抽象了文件处理和输入输出接口。**   * **实现方式：**   + **定义了 PuzzleLoader 接口类，提供加载和保存数独棋盘的规范化接口。**   + **PuzzleLoaderDAT 类实现了 DAT 格式的加载和保存。**   + **如果未来需要支持其他格式（如 XML、JSON），只需实现新的 PuzzleLoader 子类。**   + **定义了 IOInterface 接口类，提供涉及输入输出的规范化接口。**   + **Console类实现了控制台形式的输入输出交互。**   + **如果未来需要支持其他格式（如GUI、SFML），只需实现新的 IOInterface子类。** * **优势：**   + **可复用性：文件处理和输入输出逻辑独立，可在其他项目中复用。**   + **可扩展性：支持多种文件格式和交互方式，满足不同需求。**   + **灵活性：可以根据需要选择或切换不同的文件处理实现和交互方式。**   **2. MVC 模式的引入**  **为了清晰地分离关注点，增强系统的结构化，我们采用了模型-视图-控制器（MVC）模式。**   * **实现方式：**   + **模型（Model）：Sudoku 类及相关的内部逻辑类，负责游戏的核心逻辑和数据处理。**   + **视图（View）：IOInterface 及其实现类 ConsoleIO，负责与用户的交互和界面显示。**   + **控制器（Controller）：SudokuController 类，负责协调模型和视图，处理用户输入和更新模型状态。** * **优势：**   + **可维护性：各部分职责明确，修改一部分不会影响其他部分。**   + **可扩展性：可以独立地扩展视图或模型层，而无需修改控制器。**   + **灵活性：支持多种视图实现，如控制台、GUI 等。**   **3. 控制器类的职责明确化**  **SudokuController 类专注于控制游戏的进行，与 Sudoku（模型）和 IOInterface（视图）进行交互。**   * **实现方式：**   + **SudokuController 处理用户输入，解析并执行相应的命令。**   + **更新模型状态后，通知视图更新界面。**   + **不直接参与模型的逻辑实现或视图的具体呈现，保持职责的单一性。** * **优势：**   + **可维护性：控制器的逻辑清晰，修改控制流程不会影响模型或视图。**   + **可扩展性：可以在控制器中添加新的控制逻辑或流程，如多玩家模式等。**   + **灵活性：控制器可以根据需要协调不同的模型和视图，实现多种玩法。**   **4. 游戏整体的优化与封装**  **我们设计了 GameApplication 类，进一步封装了系统的初始化和启动过程。**   * **实现方式：**   + **GameApplication 类负责：**     - **使用工厂创建所有必要的组件。**     - **初始化游戏环境。**     - **提供 startGame() 方法，供客户端调用。**   + **在 main 函数中，只需实例化 GameApplication 并调用 startGame()。** * **优势：**   + **可维护性：初始化逻辑集中在 GameApplication，易于管理和修改。**   + **封装性：隐藏了组件的创建细节，客户端代码更简洁。**   + **灵活性：可以在 GameApplication 中添加更多的初始化配置或参数支持。**   **5. 职责分离与高内聚低耦合**  **为了提高代码的可维护性，我们遵循了单一职责原则，确保每个类的职责清晰且边界明确。**   * **实现方式：**   + **Sudoku 类：只负责数独游戏的核心逻辑和规则实现。**   + **IO 类：只负责与用户的交互，展示游戏界面和获取输入。**   + **文件处理类：只负责游戏数据的加载和保存。**   + **控制器类：负责协调模型和视图，控制游戏流程。**   + **菜单类：负责菜单构建和其中的指令（聚合Command类）。**   + **其它工具类等：负责一些细小的功能，如计时（Timer类）** * **优势：**   + **可维护性：修改或扩展某个功能，只需修改对应的类，不影响其他部分。**   + **可复用性：各类功能独立，可以在其他项目或模块中复用。比如若想新增加菜单，复用原有的菜单类即可。**   + **可扩展性：添加新功能时，只需添加新的类或方法，符合开闭原则。**   **6. 抽象工厂模式的应用**  **考虑到游戏未来可能需要支持不同的平台（如图形界面、网络版等），我们引入了抽象工厂模式，将系统的组件创建过程封装起来。**   * **实现方式：**   + **定义了 GameComponentsFactory 抽象工厂接口，提供创建系统核心组件的方法，如 IOInterface、PuzzleLoader、Sudoku、SudokuController 等。**   + **实现了具体的工厂类 ConsoleGameComponentsFactory，用于创建控制台版本的组件。**   + **如果未来需要支持图形界面，只需新增一个 GUIGameComponentsFactory，实现相应的创建方法。** * **优势：**   + **可复用性：工厂接口和抽象产品的使用，使得组件创建逻辑可以在不同的上下文中复用。**   + **可扩展性：添加新的平台支持时，无需修改现有代码，只需实现新的工厂类。**   + **灵活性：客户端代码（如 main 函数）与具体的组件实现解耦，只依赖于抽象工厂接口。**   **7. 命令模式的应用**  **为了支持未来可能增加的菜单项（如提示等），我们采用了命令模式。**   * **实现方式：**   + **定义了 Command 抽象类，所有具体命令类继承自该类，实现 execute()方法。**   + **MenuManager 类聚合了 Command 对象，管理和执行用户选择的命令。**   + **新的命令可以通过创建新的命令类并在菜单中注册来添加。** * **优势：**   + **可复用性：命令对象可以在不同的上下文中复用。**   + **可扩展性：添加新命令不影响现有代码，符合开闭原则。**   + **灵活性：支持可以多种功能，提升用户体验。**   **8. 数据封装与扩展性的提升**  **为了方便未来在数据库中存储更多的内容，我们设计了 PuzzleData结构体。**   * **实现方式：**   + **PuzzleData结构体封装了数独题目的相关信息，如难度、题目 ID、棋盘数据等。**   + **在函数调用时，传递 PuzzleData 对象，方便在需要时扩展数据内容，而无需修改函数签名。** * **优势：**   + **灵活性：可以方便地添加新的数据字段，满足扩展需求。**   + **可维护性：数据管理集中化，修改更容易。**   + **可复用性：PuzzleData 可以在不同的模块中使用，避免数据冗余。**   **9. 操作记录与撤销功能的优化**  **为了支持撤销和重做操作，我们设计了 OperationRecorder 类，记录用户的历史操作。**   * **实现方式：**   + **OperationRecorder 类管理一个操作栈，记录每个执行的命令。**   + **在执行命令时，SudokuController 将命令对象传递给 OperationRecorder 进行记录。** * **优势：**   + **可维护性：将操作记录和撤销逻辑封装在专门的类中，职责明确。**   + **灵活性：可以方便地调整撤销和重做的策略，如限制次数、清空记录等。**   + **用户体验提升：提供撤销和重做功能，提高游戏的可玩性。**   **10.** **单例模式与状态管理的封装**  **我们对游戏状态的管理使用了单例模式，确保状态管理器 StateManager 在整个程序运行期间只有一个实例存在，并集中管理游戏的状态数据、计时器、计数器等信息。**   * **单例模式的实现** * **实现方式：**   + **通过 getInstance() 方法返回唯一的 StateManager 实例，利用了静态局部变量 instance 确保实例只被创建一次。**   + **私有化构造函数，禁止通过 new 创建多个 StateManager 实例。**   + **禁止复制和赋值操作，防止通过复制构造或赋值操作创建多个实例。** * **优势：**   + **全局唯一性：StateManager 的单例模式保证了全局只存在一个状态管理器实例，避免了多个实例可能带来的状态不一致问题。**   + **易于访问：通过 getInstance() 静态方法，可以在任意位置轻松获取状态管理器，简化了代码的管理。**   + **资源节约：避免了重复创建状态管理器实例，提高了系统的资源利用效率。** * **状态管理的封装** * **实现方式：**   + **StateManager 类中包含了 PuzzleData（封装了数独棋盘的相关信息）、Timer（管理游戏计时）、Counter（管理游戏中的统计信息）。**   + **通过集中管理这些组件，StateManager 提供了一个统一的接口用于访问游戏的核心状态信息。** * **优势：**   + **高内聚：将与游戏状态相关的功能（如计时、计数、棋盘数据等）集中在一个类中，增强了类的内聚性。**   + **低耦合：其他模块只需与 StateManager 交互，减少对内部组件（如 Timer、PuzzleData）的直接依赖，实现了模块间的松耦合。**   + **可维护性：状态管理器集中管理状态，使得修改游戏状态的逻辑时，只需调整 StateManager，不会影响其他模块。**   + **扩展性：未来如果需要添加新的状态信息，只需在 StateManager 中新增相应的成员变量，并进行必要的封装，符合开闭原则。**   **11**. **智能指针的使用与资源管理**  **为了提高资源管理的安全性和代码的健壮性，我们在合适的地方使用了智能指针（如 std::unique\_ptr）。**   * **实现方式：**   + **在 GameApplication 和其他需要管理动态资源的地方，使用智能指针代替手动的内存管理。**   + **智能指针自动管理对象的生命周期，防止内存泄漏。** * **优势：**   + **可维护性：减少了内存管理的负担，降低了错误的可能性。**   + **可靠性：自动资源管理，提高了程序的稳定性。**   + **简洁性：代码更加简洁，减少了 delete 等操作。**   **总结**  **通过以上的设计优化，我们在系统中充分体现了面向对象设计的原则和设计模式的优势。系统具备了良好的可维护性、可复用性、可扩展性和灵活性，能够适应未来的需求变化。具体体现在：**   * **使用抽象工厂模式和 MVC 模式，增强了系统的结构性和模块化，方便添加新的功能和平台支持。** * **引入命令模式，支持扩展新的游戏指令，提升了系统的灵活性和可维护性。** * **引入单例模式，避免同一类多个实例可能导致的冲突或数据不一致问题。** * **职责分离，各个类和模块边界清晰，降低了耦合度，提高了代码的质量。** * **封装性，通过 GameApplication 等类，隐藏了复杂的初始化和管理逻辑，提供了简洁的接口。** * **资源管理优化，使用智能指针等现代 C++ 特性，提升了代码的可靠性和安全性。**   **这些设计优化不仅满足了当前的功能需求，还为未来的功能扩展和维护提供了坚实的基础。** |
| **编程实现（仅给出关键代码和注释说明即可，不必粘贴全部源程序）**  SudokuController::SudokuController(Sudoku\* sudokuModel, IOInterface\* ioInterface)  : sudoku(sudokuModel), io(ioInterface), archieve(1),  isSudokuRunning(true), isRunning(true), operationRecorder(sudokuModel) {  // 初始化游戏菜单选项  gameMenuManager.addOption("输入一个数", new InputNumberCommand(sudoku, io, &operationRecorder));  gameMenuManager.addOption("擦去一个数", new EraseNumberCommand(sudoku, io, &operationRecorder));  gameMenuManager.addOption("输入候选数", new AddCandidateCommand(sudoku, io));  gameMenuManager.addOption("删除候选数", new RemoveCandidateCommand(sudoku, io));  gameMenuManager.addOption("自动更新候选数", new AutoUpdateCandidatesCommand(sudoku, io));  gameMenuManager.addOption("保存游戏", new SaveGameCommand(sudoku, io, sudoku->getID()));  gameMenuManager.addOption("重置游戏", new ResetGameCommand(sudoku, io));  gameMenuManager.addOption("自动填入唯一候选数", new AutoSetNumberCommand(sudoku));  gameMenuManager.addOption("返回上一步对值的修改", new BackCommand(sudoku, io, &operationRecorder));  gameMenuManager.addOption("撤销返回上一步", new RevokeBackCommand(sudoku, io, &operationRecorder));  gameMenuManager.addOption("退出游戏", new ExitCommand(io, &isSudokuRunning));    // 初始化主菜单选项（未实现，可以后拓展）  mainMenuManager.addOption("选择游戏存档", new GetNumber(io, &archieve));  mainMenuManager.addOption("退出程序", new ExitCommand(io, &isRunning));  }  // 抽象工厂，创建一个抽象的工厂接口，用于创建相关的对象  class GameComponentsFactory {  public:  virtual ~GameComponentsFactory() = default;  // 创建输入输出接口  virtual IOInterface\* createIOInterface() = 0;  // 创建题目加载器  virtual PuzzleLoader\* createPuzzleLoader() = 0;  // 创建数独游戏实例  virtual Sudoku\* createSudoku(PuzzleLoader\* loader) = 0;  // 创建控制器实例  virtual SudokuController\* createController(Sudoku\* sudoku, IOInterface\* io) = 0;  };  // 具体工厂类  class ConsoleGameComponentsFactory : public GameComponentsFactory {  public:  IOInterface\* createIOInterface() override {  return new ConsoleIO();  }  PuzzleLoader\* createPuzzleLoader() override {  return new PuzzleLoaderDAT();  }  // 实现创建数独游戏实例的方法  Sudoku\* createSudoku(PuzzleLoader\* loader) override {  return new Sudoku(loader);  }  // 实现创建控制器实例的方法  SudokuController\* createController(Sudoku\* sudoku, IOInterface\* io) override {  return new SudokuController(sudoku, io);  }  };  // 应用类：封装整个游戏，使用智能指针，使得资源管理更加安全，避免了内存泄漏的风险  class GameApplication {  public:  GameApplication(std::unique\_ptr<GameComponentsFactory> factory);  GameApplication(); // 提供缺省值的构造函数  void startGame();  private:  std::unique\_ptr<GameComponentsFactory> factory;  std::unique\_ptr<IOInterface> io;  std::unique\_ptr<PuzzleLoader> puzzleLoader;  std::unique\_ptr<Sudoku> sudoku;  std::unique\_ptr<SudokuController> controller;  };  // 主函数  #include "GameApplication.h"  int main() {  // 创建游戏应用程序  GameApplication app;  // 启动游戏  app.startGame();  // 智能指针会自动清理资源  return 0;  } |
| **运行测试（给出测试方法、测试数据与测试结果）**  **1.测试方法**  **使用集成测试，验证各模块之间的交互是否正确，确保系统整体功能正常；模拟用户操作，测试完整的游戏流程，包括游戏启动、数字填入、撤销操作、游戏保存和加载等功能；检查模块之间的数据传递和接口调用是否正确。**  **初始化棋盘**  **5 3 0 0 7 0 0 0 0**  **6 0 0 1 9 5 0 0 0**  **0 9 8 0 0 0 0 6 0**  **8 0 0 0 6 0 0 0 3**  **4 0 0 8 0 3 0 0 1**  **7 0 0 0 2 0 0 0 6**  **0 6 0 0 0 0 2 8 0**  **0 0 0 4 1 9 0 0 5**  **0 0 0 0 8 0 0 7 9** ****测试数据 1****：对值和候选数的部分操作  1. **用户操作**：用户填入 4 在第 1 行第 3 列。（输入1 1 3 4，以回车间隔）    * 预期结果：数字正确填入棋盘。 2. **候选数更新**：用户输入自动更新候选数对应的命令（数字5），未填入的Cell的候选数是否合理更新。    * 预期结果：棋盘上的Cell的候选数被合理更新。    * 预期结果：棋盘状态与保存前一致。   下面为测试1的运行截图：     ****测试数据 2****：错误输入与撤销  1. **用户操作**：用户尝试填入一个非法的数字，如在第 2 行第 1 列填入 6（该行已有 6）。（输入1 2 1 6，以回车间隔）    * 预期结果：系统显示红色提示“无效操作。请重新输入。” 2. **撤销操作**：用户再次进行填数操作，并撤销操作。    * 预期结果：数字恢复为原状态，操作步数增加。   下面为测试2的运行截图：     ****测试数据 3****：完成游戏  1. **完成游戏**：用户将剩余的空格填满，完成游戏。    * 预期结果：系统提示“恭喜你完成了数独！请按回车键退出”并显示最终的时间和步数统计。再次点击回车键，结束游戏，控制台退出。   测试3截图：    （此处直接打开了一个已经完成了的数独，故步数和时间显示为0）  **测试结果预期**   1. **模块间交互**：所有模块如 ConsoleIO、SudokuController、MenuManager 等能够正常交互，输入输出流畅。 2. **用户操作反馈**：用户在填入数字、撤销、保存和加载等操作时，系统能及时且准确地反馈信息。 3. **错误处理**：非法输入操作应能及时提示错误，撤销操作恢复原状。 4. **游戏保存和加载**：保存的游戏数据能够正确加载，棋盘状态与步数能够完全恢复。 5. **候选数处理**：手动或自动更新候选数时，候选数字能够正确显示和删除。 6. **计时和步数统计**：计时和步数统计在整个游戏过程中正确累计，并在游戏结束时正确显示。 |
| **使用说明（给出程序运行及使用中需要注意的问题）**   1. **启动程序：**启动程序后，您将会看到数独游戏的控制台界面。控制台将显示游戏的棋盘和相关信息，帮助您开始游戏。   **注意：**建议调整控制台窗口的大小，使其宽度和高度足够，以便正确显示数独棋盘和游戏相关信息。如果控制台窗口过小，可能导致棋盘和游戏信息的排布出现错乱，影响游戏的正常进行。   1. **数独题目文件：**数独题目保存在puzzles.dat中，需要和sudoku.exe放在同一路径下。 2. **导入数独题目：**程序中包含一些预置的数独题目，但数量有限。为了增加游戏的多样性，您可以通过手动编辑 puzzles.dat 文件来导入新的数独题目。   **注意：**添加新题目时，请严格按照已有的题目格式进行，以确保游戏可以正确加载这些题目。例如，每个题目需要包含题号（ID）、难度（Difficulty）、棋盘数据（Board）等信息。   1. **提示输入错误：**请检查输入的指令是否符合规定的格式，例如，输入的行列位置和数值是否在合理范围内。若出现指令提示错误的情况，重新输入并确保输入格式正确。 2. **文件编码：**本程序的所有文件都应保存为带标签的UTF-8编码，以避免加载文件时出现字符编码问题。 |
| **总结展望（包括小组总结和每位组员针对自己分工的总结，对程序进行分析评价，总结遇到的问题及解决办法，展望可以完善或提高的方面）**  **小组总结**  在这次数独游戏项目开发中，我们团队紧密协作，成功实现了一个具有良好的灵活性、可拓展性、可复用新的能充分体现面向对象的思想的数独游戏。整个项目的开展展示了小组成员优秀的团队精神和实践能力。我们通过合理的分工合作高效推进项目进度。其圆满完成得益于每个组员的辛勤努力和积极沟通。遇到问题时，我们共同探讨解决方案，这一过程不仅锻炼了每位成员的编程技能，也加强了我们的协作能力。我们为其付出的时间与汗水让我们切身体会到了面向对象的程序设计思想。  **组员个人总结**  刘承晧：  作为项目的负责人，我完成了代码的大部分工作，包括核心功能设计与实现等。但项目的完成脱离不开我与其它组员的合理分工与配合。这个项目到现在总共有104次提交、创建过16个分支，是我们的心血与精益求精的精神体现。我们的程序除了基本功能外，还包含了自动更新候选数、撤销和重做等能给用户带来更好的使用体验的操作；更重要的是，我们的程序背后的代码处处体现着面向对象的程序设计思想，使得其有优秀的灵活性、可复用性、可拓展性。  然而，优化是永无止境的，出于时间精力的限制，有些想法以及可以改善的地方并没有去做，比如我们的设计都是基于“经典数独游戏”做的，也就是说规则是定死的。存在一些“变体数独”，比如它的九宫格形状发生了变化、或者添加了新的规则（如对角线应该满足1到9不重复），这些改动对于我们目前的代码来讲都不好去适应；需要对游戏规则部分做重构，或是设计新的规则接口来适应这些变化。  在开发过程中我们遇到了许多问题，并不断探寻更好的设计方案：比如针对数独类是否应该负责游戏进行的问题我们使用了MVC模式、针对命令可拓展性的问题我们使用了命令模式、针对代码各个部分的灵活性和开闭原则我们使用了抽象工厂模式；通过不断地修改、重构我们的代码，我们才得到了今天这般集多种设计模式于一体的优秀代码。如小组总结中提到的，我在项目上付出的时间让我切身体会到了面向对象的程序设计思想，我认为这些时间与汗水是值得的，更因为它让我深刻领悟到了“面向对象程序设计”的艺术。  林伟煌：  在本次项目中，我主要负责了IO模块的设计与实现，涵盖了数独棋盘的显示、用户输入的处理、以及时间和步数的统计功能。  通过一系列输入函数捕获用户操作，验证合法性后传递给游戏逻辑进行处理。通过采用面向对象的模块化设计，按照职责单一原则将功能划分为不同的类，如Timer负责计时，Counter负责步数统计。各个模块之间的职责明确，增强了代码的可维护性和可复用性。易于维护。为了提升用户体验，我在控制台输出中加入了颜色提示功能，通过不同颜色区分固定数字和用户输入值，最初的数独棋盘显示存在对齐问题，候选数字的长度不一致导致显示不整齐。为此，我使用了C++的setw()函数确保输出的数字宽度一致，最终实现了良好的视觉效果。同时完善了错误输入的处理机制，确保用户输入的有效性并提高了交互的友好性。  尽管目前程序已实现了基本功能，但未来可以继续优化：如引入图形化用户界面（GUI）以提升交互美观性，支持更多变体数独游戏，以及增加快捷键或鼠标操作来简化用户输入，从而进一步提升程序的灵活性和用户的操作体验。这次的设计让我深刻领悟到了“面向对象程序设计”的艺术。  常艺豪：  作为项目的成员之一，我主要负责数独游戏的文件加载和保存模块的设计与实现。，我实现了PuzzleLoaderDAT类，该类继承自PuzzleLoader接口，具体实现了loadPuzzle和savePuzzle两个核心函数，负责从文件加载数独数据和将数独数据保存到文件中。实现这一过程的代码涉及到文件的正确打开、内容读取和保存、以及在边界条件下的稳健性。  我将id，难度等信息改进为puzzledata结构体，以便之后数独增加其他信息，提高了程序的拓展性与复用性。除了基本的加载与保存功能外，我还致力于提高文件读写的效率，优化了文件管理的方式，以减少加载时间并增强用户体验。在开发过程中，我还遇到了文件操作失败的问题，例如文件无法打开或读取内容不完整。这些问题通异常处理机制得到了改善，并进一步增强了代码的鲁棒性。通过不断地修改和重构，我们不断寻求更加优雅和高效的实现方式，以适应不断变化的需求和挑战。  通过这次项目的开发，我切身体会到了文件系统操作的复杂性以及面向对象设计思想的力量。这些时间和努力是值得的，因为它让我深刻领悟到了“面向对象程序设计”的艺术。  **展望与提高**  基于我们在本次数独项目开发中的经验和反思，未来可以从以下几个方面继续提高和完善：   1. 增加用户界面： 目前数独游戏基于命令行，我们可以通过引入图形用户界面（GUI）来进一步提升用户体验。例如，使用工具如 Qt 或 SFML 来开发图形界面，增加按钮、输入框等元素，使交互更加直观。我们还可以设计更为美观的界面，并引入棋盘更新的动画效果，以提高游戏的视觉体验。 2. 引入不同难度级别的自动生成器： 目前游戏依赖从文件加载数独棋盘，未来可以实现自动生成不同难度的数独功能。通过设计算法生成从简单到困难的数独题目，不仅丰富了游戏的内容，还能吸引更多类型的玩家。 3. 支持更多棋盘尺寸和变种玩法： 当前版本仅支持传统的 9x9 数独，未来可以增加对 4x4、16x16 等不同尺寸棋盘的支持。此外，还可以引入变种玩法，比如对角线数独、六角数独等，提升游戏的趣味性和挑战性。 4. 提示与解答分析功能： 可以为游戏添加提示功能，帮助玩家分析棋盘并提供合理的建议，尤其是在玩家遇到困难时。通过实现解答算法，系统可以为玩家提供解题思路，甚至在用户需要时一步步展示解题过程。 |