Архитектурный Документ: Шахматы: Дополнение

Раздел 1: Общее Описание Системы

Основная цель игры "Шахматы: Дополнение" – предоставить двух игрокам уникальный игровой опыт, сочетающий классические шахматы с элементами карточной игры и фэнтезийными механиками. Игра расширяет базовые правила шахмат, предлагая новые способы взаимодействия с игровым полем и фигурами.

Система представляет собой распределенное клиент-серверное приложение.

Godot-клиент (Frontend): Отвечает за визуализацию игрового поля, фигур, интерфейса пользователя, отправку действий игрока на сервер и отображение обновлений, полученных от сервера.

Python-сервер (Backend): Отвечает за всю игровую логику, управление состоянием игры, обработку ходов, проверку их валидности, генерацию случайных событий, управление колодой и рукой игрока, а также синхронизацию данных между игроками.

База данных (будущее расширение): Планируется для хранения персистентных данных, таких как профили игроков, статистика и достижения.

Высокоуровневая схема взаимодействия:

Блок ввода данных пользователем (на клиенте) -> Блок обработки информации сервером -> Блок отображения изменений (на клиенте).

Основные технологии компонентов:

- **Godot-клиент:** Godot Engine (GDScript).
- **Python-сервер:** Python.

- **Сетевое взаимодействие**: WebSockets.
- **Формат данных:** JSON.

База данных требует дополнительной консультации и выбора.

Используемые принципы проектирования:

Модульность (Modularity): Разделение системы на независимые, логически связанные компоненты или модули. Это упрощает разработку, тестирование, отладку и сопровождение, так как изменения в одном модуле меньше влияют на другие. Например, модуль игровой логики на сервере должен быть отделен от модуля управления матчами.

Разделение ответственности (Separation of Concerns): Каждый компонент или модуль должен иметь четко определенную и одну единственную ответственность. Клиент отвечает за отображение и ввод, сервер — за логику и состояние. Модуль ГСЧ отвечает только за генерацию чисел, а не за их применение.

Единый Источник Истины (Single Source of Truth - SSOT): Сервер всегда является единственным и авторитетным источником текущего состояния игры. Клиенты только отображают это состояние и отправляют запросы на его изменение. Это предотвращает рассинхронизацию и непредусмотренное вмешательство.

Идемпотентность (Idempotency): Не зависящие от вероятности операции, если они повторяются несколько раз, должны приводить к тому же результату, что и однократное выполнение. Это может быть полезно для сетевых запросов в случае потери пакетов.

Отказоустойчивость (Fault Tolerance): Система должна быть способна продолжать работать, даже если часть ее компонентов выйдет из строя или возникнут ошибки. Например, обработка отключения игрока.

Расширяемость (Extensibility): Архитектура должна позволять легко добавлять новые функции, фигуры, карты, типы фигур и карт, ауры, состояния и т.д., а также новые механики без значительных переработок существующего кода.

Тестируемость (Testability): Проектирование компонентов таким образом, чтобы их было легко тестировать изолированно и в комплексе.

Раздел 2: Детализация Компонентов

2.1 Godot-клиент (Frontend)

Основные подсистемы/модули:

Модуль графического интерфейса (GUI): Отвечает за отображение всех элементов пользовательского интерфейса (кнопки, панели, текст, всплывающие подсказки, лог).

Модуль рендеринга доски и фигур: Отвечает за визуализацию шахматной доски, фигур на ней, карт на руке игрока, аур и состояний.

Модуль управления пользовательским вводом: Обрабатывает действия игрока (клики мышью, перетаскивание фигур, выбор карт).

Модуль сетевого взаимодействия: Отвечает за отправку действий игрока на сервер и прием обновлений состояния игры от сервера через WebSockets.

Модуль отображения информации: Выводит всплывающие подсказки и игровой лог.

Основная ответственность каждого подсистемы/модуля:

GUI: Управление интерактивными элементами.

Рендеринг: Визуализация игрового мира.

Ввод: Преобразование действий пользователя в команды.

Сетевое взаимодействие: Коммуникация с сервером.

Отображение информации: Предоставление обратной связи игроку.

Взаимодействие подсистем/модулей внутри клиента:

Модуль ввода данных пользователя отправляет запросы к Модулю сетевого взаимодействия.

Модуль сетевого взаимодействия получает обновления от сервера и передает их Модулю рендеринга доски и фигур, а также Модулю графического интерфейса для отображения изменений.

Внешние зависимости: Godot-клиент зависит от Python-сервера для получения актуального состояния игры и обработки игровой логики.

2.2 Python-сервер (Backend)

Основные подсистемы/модули:

Модуль управления матчами: Создание, поиск и присоединение к матчам.

Модуль игровой логики: Содержит все правила шахмат, правила карт, аур, состояний, движения фигур, правила взятия, стоимости фигур и условий победы/поражения.

Модуль генерации случайных чисел (ГСЧ): Отвечает за все случайные события в игре (например, шанс Некроманта, нестабильные карты). Передает случайные числа в блоки, которым это необходимо.

Модуль синхронизации состояния игры: Отвечает за поддержание единого состояния игры и его рассылку подключенным клиентам.

Модуль сетевого взаимодействия (WebSockets): Обрабатывает входящие запросы от клиентов и отправляет исходящие обновления.

Блок генерации игрового пространства: Используется в начале партии для формирования доски и колоды.

Блок валидации хода: Проверяет возможность совершения игрового хода (может ли фигура так ходить, соблюдаются ли правила шахмат, не оказывается ли король под шахом и т.п.).

Блок изменения шахматной позиции: Применяет ход и все связанные с ним вероятностные расчеты и эффекты.

Блок расчета изменений: Расчитывает, какие изменения повлечет за собой сделанный ход (применяя вероятностные события, ауры, состояния).

Блок обработки условий завершения игры: Проверяет условия мата/пата и другие условия победы/поражения.

Блок постобработки: Проверяет и применяет эффекты, происходящие после основного хода (например, отмена хода забывчивой фигурой).

Основная ответственность каждого подсистемы/модуля:

Управление матчами: Создание и управление игровыми сессиями.

Игровая логика: Обеспечение корректности игрового процесса по правилам.

ГСЧ: Генерация случайности.

Синхронизация: Актуальное состояние для всех клиентов.

Сетевое взаимодействие: Обмен данными с клиентами.

Генерация игрового пространства: Инициализация игры.

Валидация хода: Проверка легальности хода.

Изменение позиции: Применение хода и его эффектов.

Расчет изменений: Детализация последствий хода.

Обработка завершения игры: Определение победителя/ничьей.

Постобработка: Финальные проверки после хода.

Взаимодействие подсистем/модулей внутри сервера:

Модуль сетевого взаимодействия получает ход от клиента.

Запрос передается в Блок валидации хода.

Если ход валиден, он передается в Блок изменения шахматной позиции.

Блок изменения шахматной позиции взаимодействует с Блоком расчета изменений (для применения вероятностных событий) и с Блоком валидации (для проверки шахматных правил после хода).

Информация о ходе переходит в Блок изменения шахматной позиции, где к ходу применяются все вероятностные расчеты.

Блок изменения шахматной позиции вызывает Блок обработки условий завершения игры, который в свою очередь вызывает Блок расчета изменений и Блок валидации для проверки возможных ходов противника.

После этого информация проходит через Блок постобработки (для эффекта Забывчивости).

Модуль ГСЧ предоставляет случайные числа всем модулям, которым это необходимо (например, Модулю игровой логики для Некроманта или нестабильных карт).

После обработки хода, Модуль синхронизации состояния игры рассылает обновления клиентам через Модуль сетевого взаимодействия.

Внешние зависимости: Python-сервер может зависеть от библиотеки WebSockets, а в будущем от выбранной СУБД.

Раздел 3: Взаимодействие Компонентов и Потоки Данных

Организация сетевого взаимодействия между Godot-клиентом и Python-сервером:

Используется протокол WebSockets для постоянного двунаправленного соединения.

Обмен данными происходит в формате JSON.

Типы сообщений от клиента к серверу:

Запрос на создание/поиск/присоединение к матчу.

Запрос на "Взять одну карту из колоды".

Запрос на "Обменять одну карту" (с указанием сбрасываемой карты).

Запрос на "Разыграть карту" (с указанием карты и, при необходимости, целевой фигуры/клетки).

Запрос на "Совершить ход имеющейся на доске фигурой" (с указанием фигуры и координат перемещения).

Запрос на использование "Двойного добора" (с указанием типа действия: взять 2 или обменять 1 на 2).

Запрос на полную синхронизацию состояния игры.

Типы сообщений от сервера к клиенту:

Обновление состояния игры после хода (изменения позиции фигур, наложение/снятие аур/состояний, изменения в составе карт на руке и колоде). Сервер передает только произошедшие изменения.

Полное состояние игры (по запросу клиента для восстановления синхронизации).

Уведомления о завершении игры (мат, пат).

Уведомления об ошибках (некорректный ход, невозможность действия).

Основные сценарии использования и соответствующие потоки данных:

Сценарий: *Игрок делает ход фигурой*.

Игрок на Godot-клиенте перетаскивает фигуру (Модуль управления пользовательским вводом).

Клиент проверяет локально базовую валидацию (например, выбранная фигура принадлежит игроку, конечная клетка находится на доске).

Модуль сетевого взаимодействия клиента формирует JSON-сообщение о типе хода (перемещение фигуры) и деталях (исходная/конечная координаты).

Сообщение отправляется на Python-сервер.

Модуль сетевого взаимодействия сервера принимает сообщение и передает его в Модуль игровой логики.

Модуль игровой логики (через Блок валидации хода) проверяет легальность хода (согласно правилам фигуры, аур, состояний, а также шахматным правилам).

Если ход валиден, Блок изменения шахматной позиции обрабатывает ход, применяет все вероятностные события (используя Модуль ГСЧ) и эффекты (ауры, состояния). Также происходит проверка на взятие фигуры.

Блок постобработки проверяет условия (например, эффект Забывчивости) и в случае срабатывания откатывает ход.

Модуль синхронизации состояния игры формирует JSON-сообщение, содержащее новую доску.

Модуль сетевого взаимодействия сервера отправляет сообщение обоим клиентам.

Модуль сетевого взаимодействия клиента принимает сообщение и передает его Модулю рендеринга доски и фигур и Модулю графического интерфейса для обновления визуального состояния игры.

Сценарий: *Игрок разыгрывает карту "Аура".*

Игрок на Godot-клиенте выбирает карту "Аура" и целевую фигуру.

Клиент отправляет JSON-сообщение о типе хода (разыгрывание карты), названии карты и ID целевой фигуры.

Сервер принимает сообщение, проверяет валидность (наличие карты, возможность наложения ауры на фигуру, правила приоритета аур).

Модуль игровой логики применяет ауру к фигуре.

Сервер отправляет клиентам обновление об изменении состояния фигуры (наложенная аура, снятая карта).

Клиенты обновляют отображение.

Обеспечение синхронизации.

Сервер является Единым Источником Истины.

После каждого действия игрока (или автоматического события на сервере), сервер обрабатывает изменения и отправляет новое состояние доски.

Клиент всегда может запросить полное состояние игры у сервера для восстановления синхронизации в случае рассинхронизации или после переподключения.

Генерация случайных чисел (ГСЧ) происходит исключительно на сервере. Клиентам не передаются сами результаты генерации чисел, а только результаты произошедших событий, которые были определены ГСЧ.

Пример: для Некроманта, сервер генерирует случайное число и определяет, будет ли воскрешена фигура. Клиентам отправляется сообщение "Фигура X была воскрешена на клетке Y", а не "ГСЧ сгенерировал Z, поэтому фигура X была воскрешена". Это предотвращает непредусмотренное вмешательство и обеспечивает честность игры.

Раздел 4: Персистентность Данных

На данном этапе персистентность данных игры не требуется, однако имеет смысл рассмотреть возможность реализации в будущем.

Выбор СУБД:

SQLite.

Плюсы: Очень проста в настройке и использовании (не требует отдельного сервера), легко переносится, хороша для небольших объемов данных и прототипирования.

Минусы: Не предназначена для высокой конкурентной нагрузки (много одновременных записей), не масштабируется на несколько серверов.

PostgreSQL.

Плюсы: Надежная, поддерживает сложные запросы, хорошо масштабируется, большое сообщество и множество инструментов.

Минусы: Требует отдельной установки и настройки сервера базы данных, сложнее в освоении для новичков.

Redis.

Плюсы: Очень высокая скорость чтения/записи, идеально для быстрого доступа к данным.

Минусы: Не предназначена для хранения сложных реляционных данных, обычно не является основной базой данных для профилей пользователей, если не используется в комбинации с другой СУБД.

Данные для хранения: Статистика игроков (победы/поражения), достижения,, возможно, "артефакты" или кастомизация, если такие механики будут введены.

Для взаимодействия с реляционной СУБД из Python обычно используются ORM (Object-Relational Mapper), такие как SQLAlchemy. ORM позволяют работать с базой данных, используя объекты Python, что упрощает код и делает его более читаемым, а также обеспечивает некоторую

независимость от конкретной СУБД. Прямые SQL-запросы также возможны, но менее удобны для больших проектов.

Раздел 5: Нефункциональные Требования в контексте Архитектуры

Поддержка требований производительности.

Асинхронный сервер: Использование асинхронного фреймворка в Python (например, asyncio или FastAPI/Sanic на его основе) позволит серверу эффективно обрабатывать множество одновременных клиентских подключений и запросов, не блокируясь в ожидании ввода/вывода.

Вынесение ресурсоемких операций: Если в будущем появятся очень сложные расчеты или ИИ, их можно будет вынести в отдельные микросервисы или воркеры.

Централизация игровой логики на сервере: Клиент не выполняет ресурсоемких расчетов игровой логики, что позволяет поддерживать производительность даже на менее мощных клиентских устройствах.

Поддержка требований надежности.

Единый Источник Истины (сервер): Гарантирует, что состояние игры всегда корректно, даже если у клиента возникли временные сбои или рассинхронизация. Клиент может запросить полное состояние для восстановления.

Механизмы восстановления сессии: Сервер должен иметь возможность переподключить игрока к прерванной игре и восстановить его состояние.

Обработка некорректных действий клиентов: Сервер должен строго валидировать все входящие запросы от клиентов, отбрасывая нелегальные

ходы или действия. Это предотвращает сбои и непредусмотренное вмешательство.

Логирование: Подробное логирование событий на сервере позволит отслеживать ошибки и восстанавливать хронологию событий.

Поддержка требований безопасности.

Вся критически важная игровая логика и состояние игры находятся исключительно на сервере. Это предотвращает модификацию правил или состояний игры на стороне клиента.

Использование WebSockets Secure (WSS): Для шифрования трафика между клиентом и сервером. Это защитит данные игроков от перехвата.

Валидация на сервере: Все входные данные от клиентов должны быть тщательно проверены на сервере, чтобы предотвратить инъекции или другие атаки.

Отсутствие прямого доступа клиентов к БД: Клиенты не должны иметь прямого доступа к базе данных (когда она появится). Все операции с данными должны проходить через сервер.

Поддержка требований Маштабируемости.

Горизонтальное масштабирование сервера: Асинхронный Python-сервер может быть разработан таким образом, чтобы его можно было запускать на нескольких инстансах (серверах) и распределять нагрузку между ними (например, с помощью балансировщика нагрузки). Это позволяет обрабатывать большее количество одновременно играющих пользователей.

Микросервисная архитектура (перспектива): В будущем, если проект сильно вырастет, можно выделить отдельные части серверной логики (например, модуль ГСЧ, модуль управления матчами) в отдельные микросервисы.

Выбор СУБД: Изначальный выбор SQLite ограничивает масштабируемость БД, но при переходе на PostgreSQL или другие масштабируемые решения можно обеспечить масштабируемость и на уровне данных.

Поддержка требований удобства сопровождения.

Модульность кода и разделение ответственности: Четкое разделение компонентов и модулей внутри них упрощает понимание кода, внесение изменений и исправление ошибок, так как каждый модуль имеет свою четкую область ответственности.

Стандарты кодирования и документация: Принятие общих стандартов кодирования для Python и GDScript, а также комментирование кода и ведение актуальной документации (включая этот Архитектурный Документ и спецификации API) значительно упростят поддержку проекта.

Использование версионного контроля (Git): Для отслеживания изменений, совместной работы и возможности отката к предыдущим версиям.

Раздел 6: Допущения, Ограничения и Открытые Вопросы

Стабильное интернет-соединение: Предполагается, что оба игрока будут иметь достаточно стабильное интернет-соединение для поддержания WebSocket-соединения.

Наличие актуального Godot-клиента: Пользователи будут использовать последнюю версию клиентского приложения для совместимости с серверной частью.

Ограничение на 2 игрока: Текущая архитектура ориентирована строго на матчи между двумя игроками.

Отсутствие сложной авторизации/профилей на старте: Профили игроков и полноценная система авторизации будут реализованы на более поздних этапах.

Односерверная конфигурация (на старте): Изначально сервер будет развернут на одном инстансе, что может стать узким местом при очень большом количестве одновременных игроков.

Отсутствие полноценной системы матчмейкинга: На старте предполагается простая система создания/присоединения к матчам, без сложного подбора по рейтингу или другим параметрам.

Зависимость от WebSockets: Все взаимодействие основано на WebSockets; альтернативные протоколы для специфических нужд (например, UDP для очень быстрых, но не гарантированных обновлений) не рассматриваются.

Открытые вопросы:

Детализация системы авторизации и профилей игроков: После принятия решения о СУБД, необходимо будет детально проработать схему БД и механизмы аутентификации/авторизации.

Система логирования и мониторинга: Разработка эффективных методов сбора логов и мониторинга состояния сервера для быстрого обнаружения и устранения проблем.

Стратегия развертывания (CI/CD): Как будет автоматизирован процесс сборки, тестирования и развертывания клиентского и серверного приложений.

Оптимизация производительности рендеринга на клиенте: Хотя Godot производителен, сложные эффекты или большое количество объектов могут потребовать оптимизации.