**Отчет по проекту**

**1 Описание проекта**

Проект представляет собой текстовую многопользовательскую игру с сервером, написанным на Clojure, и ботом для автоматизации взаимодействия, реализованным на Prolog. Игровой сервер позволяет игрокам подключаться через TCP, исследовать комнаты, собирать предметы (оружие, броню, деньги), взаимодействовать с другими игроками и выполнять действия через текстовые команды. Бот подключается к серверу, анализирует ответы, полученные от него, и выполняет команды автоматически.

**1.1 Дополнительные загрузки перед началом работы**

Для работы проекта необходимо установить следующие компоненты:

**Для серверной части**

* Java Development Kit (JDK): версия 8 или выше.
* Leiningen: инструмент для сборки проектов Clojure. Устанавливается через Homebrew:

brew install leiningen

* Зависимости Clojure (автоматически подтягиваются Leiningen):

org.clojure/clojure:1.4.0

server-socket:1.0.0

**Для части работы с ботом**

* SWI-Prolog: устанавливается через Homebrew:

brew install swi-prolog

**1.2 Запуск проекта в консоли**

Запуск сервера осуществляется с помощью команд

Lein deps – установление зависимостей

Lein run – запуск сервера

По умолчанию сервер запускается на порту 3333:

Launching Mire server on port 3333

После запуска сервера с другой консоли можно подключиться к игре с помощью telnet

telnet localhost 3333

Запуск бота осуществляется в отдельной консоли, где с помощью команды swipl нужно открыть среду Prolog.

Далее загружаем файл с ботом:

[mirebot2].

И выполняем команду запуска:

main.

Для корректной работы сервер должен быть запущен перед подключением бота.

**2 Реализация проекта**

**2.1 Серверная часть**

Реализована на языке Clojure и представляет из себя текстовый многопользовательский сервер, обеспечивающий управление игровым процессом.

Рассмотрим основные компоненты серверной части:

**2.1.1 Управление комнатами**:

Все комнаты в проекте описаны в текстовых файлах, содержащих информацию о названии, описании, доступных выходах, предметах и обитателях. При запуске сервера они загружаются и преобразуются в структурированные данные с использованием функций load-room и add-rooms.

Функция load-room считывает данные из файла комнаты, преобразует их в карту данных (map) и добавляет в глобальную переменную rooms. Комнаты имеют следующие ключевые атрибуты:

* :name — имя комнаты, совпадающее с названием файла.
* :desc — текстовое описание комнаты.
* :exits — доступные направления для перемещения, представленные в виде ссылок (ref) на набор направлений.
* :items — список доступных предметов в комнате (например, оружие, броня).
* :inhabitants — обитатели комнаты, которые могут быть другими игроками или ботами.

После загрузки все данные объединяются в глобальную карту rooms, доступную для игрового процесса. Это позволяет динамически обновлять комнаты, добавлять или удалять предметы и обновлять состояние обитателей.

* + 1. **Инвентарь игрока**

Каждый игрок имеет динамический инвентарь, где хранятся собранные предметы. Инвентарь реализован с использованием динамической переменной \*inventory\*, которая обновляется в режиме реального времени в зависимости от действий игрока.

Игроки могут взаимодействовать с предметами в комнате с помощью следующих команд:

grab <предмет> — для взятия предмета из комнаты и добавления его в инвентарь.

drop <предмет> — для удаления предмета из инвентаря и возвращения его в комнату.

Эти команды обрабатываются с использованием функций, определённых в файле player.clj. В них используются транзакции (dosync), чтобы гарантировать целостность данных, особенно в многопользовательской среде.

* + 1. **Сетевая связь**

Сетевая связь реализована с использованием библиотеки server-socket, которая позволяет серверу принимать входящие TCP-соединения. Сервер по умолчанию запускается на порту 3333.

Каждое подключение клиента обрабатывается в отдельном потоке с использованием функции mire-handle-client. Эта функция связывает потоки ввода и вывода клиента с игровой логикой сервера, обеспечивая интерактивность.

Процесс обработки выглядит следующим образом: сервер получает данные от клиента (команду). Далее данные передаются обработчику команд и результат (например, описание комнаты, ответ на команду) возвращается клиенту через поток вывода.

* + 1. **Механика взаимодействия**

Механика взаимодействия строится вокруг текстовых команд, которые игроки вводят для управления своим персонажем. Основные команды включают:

look — осмотреть текущую комнату, включая её описание, доступные выходы и предметы.

move <направление> — переместиться в указанное направление (например, move north).

inventory — показать содержимое инвентаря игрока.

grab <предмет> и drop <предмет> — взаимодействие с предметами в комнате и инвентаре.

Эти команды обрабатываются с помощью функции execute, определённой в файле commands.clj. Она маршрутизирует команды на соответствующие функции-обработчики, обеспечивая выполнение действий в зависимости от текущего состояния игрока и комнаты.

**2.2 Бот**

Реализован на языке Prolog и представляет из себя автономный скрипт, предназначенный для взаимодействия с сервером текстовой игры. Основной задачей бота является автоматизация ввода команд и обработки ответов сервера.

Для подключения к серверу используется библиотека socket, которая обеспечивает взаимодействие через TCP. Подключение реализовано в функции main с использованием tcp\_connect. После установления соединения бот инициирует процесс взаимодействия, вызывая главный цикл работы.

Бот анализирует текстовые ответы сервера с помощью грамматик DCG, определённых в файле mirebot2.pl. Эти грамматики позволяют распознавать ключевые элементы ответа, такие как направления, оружие, броня и выходы. Примеры правил:

Парсинг направлений:

e(north) --> [north].

e(south) --> [south].

Парсинг оружия:

w(blade) --> [blade].

После парсинга информация сохраняется в динамических фактах (exit/1, weapon/1, armor/1), которые используются для принятия решений.

На основе данных, полученных от сервера, бот выполняет команды. Например:

Команда move <направление> используется для перемещения в указанную сторону. Логика реализована в функции process, которая формирует текст команды и отправляет её на сервер.

Команда grab <предмет> обрабатывается функцией process\_2, которая анализирует доступные предметы и выбирает первый из них для добавления в инвентарь.

Аналогично, функция process\_3 отвечает за сбор брони.

Для обеспечения непрерывного взаимодействия с сервером бот использует цикл loop. Этот цикл постоянно читает данные с сервера с помощью функции read\_loop и обрабатывает полученные данные, обновляя внутренние факты и принимая решения. После этого отправляет команды обратно на сервер для выполнения.Таким образом, бот функционирует в режиме реального времени, реагируя на изменения состояния игры и выполняя команды автоматически.

**3 Основные принципы работы на примере функций**

**3.1 Сервер**

1. **Функция load-room (загрузка комнаты):**

(defn load-room [rooms file]

(let [room (read-string (slurp (.getAbsolutePath file)))]

(conj rooms

{(keyword (.getName file))

{:name (keyword (.getName file))

:desc (:desc room)

:exits (ref (:exits room))

:items (ref (or (:items room) #{}))

:inhabitants (ref #{})}})))

Функция load-room реализована на языке Clojure и отвечает за загрузку данных об игровых комнатах из текстовых файлов в глобальную переменную rooms, которая используется для управления игровым процессом. Она является ключевым элементом механизма инициализации игрового мира и обработки данных, связанных с комнатами.

Функция возвращает обновлённую коллекцию rooms, включающую новую комнату, готовую для использования в игровом процессе. Эти данные становятся доступными для других частей программы, таких как обработка взаимодействия игроков с комнатами.

1. **Функция mire-handle-client (обработка подключения клиента):**

(defn mire-handle-client [in out]

(binding [\*in\* (reader in) \*out\* (writer out)]

(print "What is your name?\n")

(flush)

…

))

Предназначена для обработки подключения каждого клиента к серверу текстовой игры. Она связывает потоки ввода-вывода клиента с логикой игрового сервера, обеспечивая взаимодействие между игроком и игровым миром.

**3.2 Бот:**

1. **Функция process (движение бота):**

process(Stream) :-

exit([Direction|\_]),

format(atom(Command), 'move ~w~n', [Direction]),

write(Stream, Command),

flush\_output(Stream),

retractall(exit(\_)),

read\_loop(Stream).

Эта функция реализует выполнение команды перемещения бота в указанное направление. Это достигается путём анализа текущих данных, создания текстовой команды и отправки её на сервер через установленное соединение.

Функция начинает свою работу с проверки динамического факта exit/1, который содержит список доступных направлений из текущей комнаты. Этот факт ранее был сохранён ботом на основе данных, полученных от сервера. Используя запись exit([Direction|\_]), функция извлекает первое направление из списка и сохраняет его в переменной Direction.  
После определения направления перемещения создаёт текстовую команду, понятную серверу. Это делается с использованием format, который генерирует строку вида: move <направление>  
Сформированная команда записывается в поток Stream, связанный с сервером, с помощью write(Stream, Command). После этого используется flush\_output(Stream), чтобы гарантировать немедленную отправку команды серверу.

После отправки команды факт exit/1, содержащий информацию о доступных направлениях, удаляется с помощью retractall(exit(\_)) (предотвращения повторного использования старых данных, которые могут быть уже неактуальными после перемещения).  
В конце вызывается read\_loop(Stream), который инициирует следующий цикл работы бота.

1. **Функция loop (главный цикл):**

loop(Stream) :-

read\_loop(Stream),

command\_in\_room(Stream),

loop(Stream).

Представляет собой главный цикл работы бота. Она отвечает за поддержание непрерывного взаимодействия с сервером, обеспечивая чтение входящих данных, их обработку и выполнение действий. Это основа автоматизации работы бота, она обеспечивает его способность реагировать на изменения в игровом мире в режиме реального времени.