Министерство образования и науки РФ ФГАОУ ВО «Нижегородский государственный университет им. Н.И. Лобачевского» (ННГУ)

Факультет вычислительной математики и кибернетики

Кафедра ИИСГео

Игра «Коридорчики»

Реализация на Java RMI

Выполнил:

студент ф-та ВМК гр. 84-11

Аратский А.В.

Проверил:

ассистент каф. МО ЭВМ, ВМК

Свистунов А.Н.

Нижний Новгород

2014 г.

Содержание

[Содержание 2](#_Toc404802636)

[Введение 3](#_Toc404802637)

[Технология Java RMI 4](#_Toc404802638)

[Архитектура RMI 4](#_Toc404802639)

[Использование RMI 6](#_Toc404802640)

[Постановка задачи 8](#_Toc404802641)

[Руководство пользователя 9](#_Toc404802642)

[Запуск сервера 9](#_Toc404802643)

[Запуск клиента. Игра 10](#_Toc404802644)

[Руководство программиста 13](#_Toc404802645)

[Серверное приложение 13](#_Toc404802646)

[Структура программы 13](#_Toc404802647)

[Описание алгоритмов 14](#_Toc404802648)

[UML диаграмма классов 16](#_Toc404802649)

[Клиентское приложение 17](#_Toc404802650)

[Заключение 19](#_Toc404802651)

[Список литературы 20](#_Toc404802652)

Введение

Сегодняшние реалии таковы, что разработка практически любого программного обеспечения требует хороших знаний параллельного и распределенного программирования. Понятия параллельной и распределенной обработки данных не являются эквивалентными. Параллелизм подразумевает одновременное существование и выполнение задач. Распределение же обозначает территориальную удаленность процессов друг от друга. При этом задачи, связанные с распределенной обработкой данных, могут выполняться несколькими последовательными этапами в различные периоды времени.

Профессор вычислительной техники Эндрю С. Таненбаум определяет распределенную систему как набор независимых компьютеров, представляющийся их пользователям единой объединенной системой. Однако сами по себе независимые компьютеры не могут представляться пользователю единой системой. Обеспечить это можно только с помощью дополнительного специального программного обеспечения, называемого программным обеспечением промежуточного уровня (англ. middleware).

Промежуточный уровень обязательным не является, но его наличие крайне желательно. Его задача - скрыть гетерогенность платформы и обеспечить удобную модель программирования. В качестве примера такого рода программного обеспечения можно привести следующие продукты:

* **CORBA** (OMG) - разрабатываемая консорциумом OMG технология, позволяющая вызывать методы удаленных объектов.
* **Java** (Sun) - технология выполнения промежуточного байт-кода на виртуальной машине. Откомпилированный байт-код может выполняться на любой платформе, где есть JVM (Java Virtual Machine), без перекомпиляции;
* **.NET** (Microsoft) - технология, основанная на выполнении предварительно откомпилированных в промежуточный код компонентов. Такой компонент может выполняться на любой платформе, для которой реализована поддержка библиотеки времени исполнения . NET;
* **Java Remote Method Invocation** (Sun) — программный интерфейс вызова удаленных методов в языке Java.

Последнее из перечисленных средств является реализацией такой технологии, как удалённый вызов процедур (англ. Remote Procedure Call). Технология позволяет вызывать компьютерам процедуры или функции в другом адресном пространстве (как правило, на удалённых машинах). Средства удалённого вызова процедур предназначены для облегчения организации распределённых вычислений и создания распределенных клиент-серверных информационных систем.

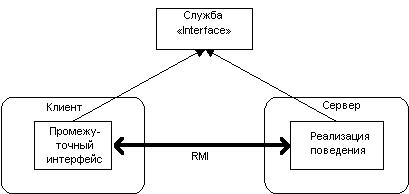
Технология Java RMI

Главной целью разработчиков RMI было предоставление возможности программистам разрабатывать распределенные Java-программы, используя такие же синтаксис и семантику, как и при разработке обычных нераспределенных программ. Для этого они должны были преобразовать модель работы классов и объектов в одной виртуальной машине JVM в новую модель работы классов и объектов в распределенной (несколько JVM) вычислительной среде. Разработчики RMI стремились сделать использование распределенных Java-объектов таким же, как и использование локальных объектов. В итоге была создана система, которая переносит безопасность и устойчивость архитектуры Java в мир распределенных вычислений. Технология Remote Method Invocation (RMI), впервые была представленна в JDK 1.1.

Архитектура RMI

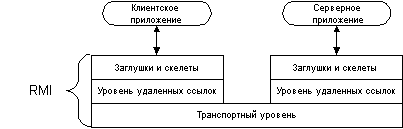
Архитектура RMI основана на одном важном принципе: определение поведения и реализация этого поведения считаются разными понятиями. Конкретно в RMI определение удаленной службы кодируется при помощи интерфейса Java. Реализация удаленной службы кодируется в классе. RMI поддерживает два класса, реализующих один и тот же интерфейс. Первый класс является реализацией поведения и исполняется на сервере. Второй класс работает как промежуточный интерфейс для удаленной службы и исполняется на клиентской машине. Это показано на следующей диаграмме.

Клиентская программа вызывает методы прокси-объекта, RMI передает запрос на удаленную JVM и направляет его в реализацию объекта. Любые возвращаемые из реализации значения передаются назад в прокси-объект и затем в клиентскую программу.



Клиентская программа вызывает методы прокси-объекта, RMI передает запрос на удаленную JVM и направляет его в реализацию объекта. Любые возвращаемые из реализации значения передаются назад в прокси-объект и затем в клиентскую программу.

Реализация RMI, по существу, состоит из трех абстрактных уровней.



Первый - это уровень заглушки и скелета, расположенный непосредственно перед разработчиком. Этот уровень перехватывает вызовы методов, произведенные клиентом при помощи переменной-ссылки на интерфейс, и переадресует их в удаленную службу RMI. В реализации RMI Java 2 SDK новый протокол связи сделал классы скелетов не нужными. RMI использует отражение для установления соединения с объектом удаленной службы. Использовать классы и объекты скелетов следует только в JDK 1.1 и совместимых с ним реализациях систем. Следующий уровень - уровень удаленной ссылки. Этот уровень понимает, как интерпретировать и управлять ссылками на удаленные объекты служб. В JDK 1.1 этот уровень соединяет клиентов с удаленными объектами служб, которые исполняются на сервере. Это соединение является связью типа один к одному (однонаправленное соединение). В Java 2 SDK этот уровень был расширен поддержкой активации пассивных удаленных объектов при помощи технологии Remote Object Activation. Транспортный уровень основан на соединениях TCP/IP между сетевыми машинами. Он обеспечивает основные возможности соединения и некоторые стратегии защиты от несанкционированного доступа.

Клиенты находят удаленные службы, используя службу имен или каталогов. Служба имен или каталогов исполняется на хорошо известном хосте и имеет известный номер порта. RMI может использовать много различных служб каталогов, включая Java Naming and Directory Interface (JNDI). RMI и сама включает в себя простую службу, называемую реестром RMI, rmiregistry. Реестр RMI работает на каждой машине, содержащей объекты удаленных служб и принимающей запросы на обслуживание, по умолчанию используя порт 1099. На хосте программа сервера создает удаленную службу, предварительно создавая локальный объект, реализующий эту службу. Затем она экспортирует этот объект в RMI. Как только объект экспортирован, RMI создает службу прослушивания, ожидающую соединения с клиентом и запроса службы. После экспорта, сервер регистрирует объект в реестре RMI, используя общедоступное имя. На стороне клиента к реестру RMI доступ обеспечивается через статический класс Naming. Он предоставляет метод lookup(), который клиент использует для запросов к реестру.

Использование RMI

Рабочая RMI-система состоит из нескольких частей:

* Определение интерфейсов для удаленных служб
* Реализация удаленных служб
* Файлы заглушки и скелета *(начиная с версии J2SE 5.0 реализована динамическая компиляция файлов заглушки и скелета)*
* Сервер, предоставляющий удаленные службы
* Служба имен RMI, дающая возможность клиентам найти удаленные службы
* Поставщик файла классов (HTTP или FTP-сервер)
* Клиентская программа, которая нуждается в удаленных службах

Рассмотрим их реализацию на конкретном примере.

|  |
| --- |
| **public** **interface** Calculator **extends** java.rmi.Remote {  **public** long add(long a, long b) **throws** java.rmi.**RemoteException**;  **public** long sub(long a, long b) **throws** java.rmi.**RemoteException**;  } |

Интерфейс Calculator определяет все удаленные возможности, предлагаемые службой. Этот интерфейс расширяет интерфейс Remote, и в сигнатуре каждого метода определяется, что он может генерировать объект RemoteException.

|  |
| --- |
| **public** **class** CalculatorImpl  **extends** java.rmi.server.UnicastRemoteObject **implements** Calculator {  // Реализации должны иметь явный конструктор для  // того, чтобы объявить исключительную ситуацию RemoteException  **public** CalculatorImpl() **throws** java.rmi.**RemoteException** { }  **public** long add(long a, long b) **throws** java.rmi.**RemoteException** {  **return** a + b;  }  **public** long sub(long a, long b) **throws** java.rmi.**RemoteException** {  **return** a - b;  }  } |

Класс реализации использует UnicastRemoteObject для присоединения к системе RMI. В данном примере класс реализации непосредственно расширяет UnicastRemoteObject. Это не является обязательным требованием. Класс, не расширяющий UnicastRemoteObject, может использовать свой метод exportObject() для присоединения к RMI.

Для старых версий также необходимо скомпилировать файл заглушки и скелета. Для это используется компилятор RMI, rmic. Компилятор запускается с указанием файла класса, реализующего удаленную службу. В новых версиях эти файлы компилируются динамически при запуске реализации удалённых служб.

|  |
| --- |
| **import** java.rmi.Naming;  **public** **class** CalculatorServer {  **public** CalculatorServer() {  **try** {  Calculator c = **new** CalculatorImpl();  **Naming**.rebind("rmi://localhost:1099/CalculatorService", c);  } **catch** (**Exception** e) { e.printStackTrace(); }  }  **public** **static** void main(**String** args[]) {  **new** CalculatorServer();  }  } |

Удаленные службы RMI должны быть помещены в процесс сервера. Класс CalculatorServer является очень простым сервером, предоставляющим простые элементы для размещения.

Исходный код клиента следующий:

|  |
| --- |
| **import** java.rmi.Naming;  **import** java.rmi.RemoteException;  **import** java.net.MalformedURLException;  **import** java.rmi.NotBoundException;  **public** **class** CalculatorClient {  **public** **static** void main(**String**[] args) {  **try** {  Calculator c =  (Calculator) **Naming**.lookup("rmi://remotehost/CalculatorService");  **System**.out.println( c.sub(4, 3) );  **System**.out.println( c.add(4, 5) );  }  **catch** (**Exception** e) { e.printStackTrace(); }  }  } |

Постановка задачи

Требуется создать распределённую систему, использующую технологию Java RMI. Система представляет собой игру «Коридорчики».

Правила игры:

* Игровое поле - клеточная доска заданных заранее размеров
* Количество игроков неограниченно
* Игроки по очереди проводят горизонтальные или вертикальные линии в одну клетку
* Игрок, которому удалось замкнуть линиями клетку, ставит в ней свой знак и получает еще один ход.
* Когда все клетки окажутся занятыми, подсчитывают, кто «захватил» больше клеток тот и победитель

Распределённая система представляет собой:

Приложение-сервер

* На сервере хранится непосредственно игровое поле, с которым взаимодействуют все игроки (приложения-клиенты).
* Так же сервер хранит всю информацию об игроках и производит управление игрой (начинает игру, определяет конец игры).
* Является консольным приложением.

Приложение-клиент

* Является приложение с визуальным интерфейсом, где отображается игровое поле, таблица с текущими результатами игроков, и дополнительные возможности (настройка, чат).
* Через приложение-клиент игрок (пользователь) получает доступ к удалённому игровому полю и взаимодействует с ним (играет).

Руководство пользователя

Запуск сервера

Приложения RMI для своей работу требует запущенную удалённую службу реестра RMI Registry. Реестр запускается командой **rmiregistry**.

При запуске приложения сервера в аргументах вызова можно указать размеры клеточной доски. Иначе будут выбраны значения по умолчанию.

При удачном запуске сервера будет выведена следующая информация:

**-> Start server ...**

**-> Gameboard created**

**Width 4**

**Heigth 5**

**-> Gameboard listener added**

**-> Server ready ...**

**-> Waiting for players ...**

После чего сервер будет находится в ожидании игроков. Как только количество игроков достигнет двух, сервер предложит начать игру. Однако количество игроков не ограниченно, и можно продолжать подключать новых игроков пока не наберётся нужное вам количество. Для начала игры просто введите y или Y, тогда сервер больше не будет ждать новый клиентов и начнёт игру.

**-> Waiting for players ...**

**-> Player Alexandr joined ...**

**-> Player Svetlana joined ...**

**-> There are 2 players. Start game? [y/n]**

**-> Player Dmitry joined ...**

**y**

**-> Next: Alexandr**

**-> Next: Svetlana**

**-> Update score:**

**1 Alexandr $0**

**2 Svetlana $5**

**3 Dmitry $0**

**-> Next: Dmitry**

**-> Update score:**

**1 Alexandr $0**

**2 Svetlana $5**

**3 Dmitry $5**

**-> Next: Alexandr**

Во время игры сервер выводит информацию о текущем игроке и обновляет результаты. В конце игры сервер объявляет победителя.

**-> Game Over**

**-> Winner: Dmitry**

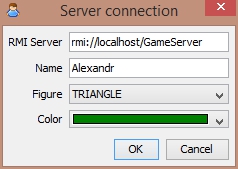
**3 Dmitry $40**

**2 Svetlana $35**

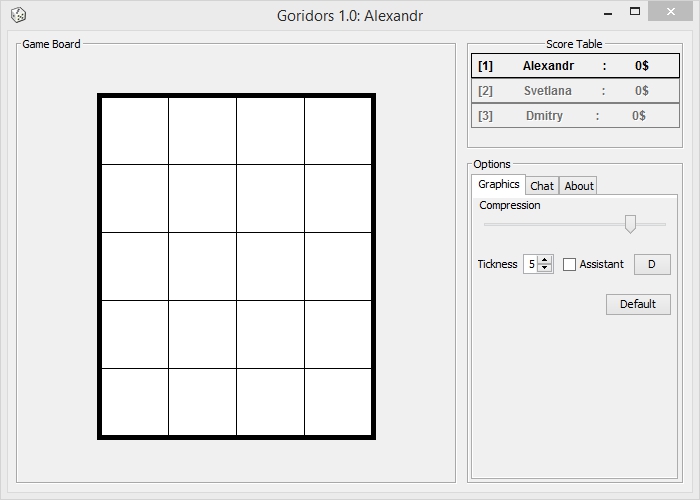
**1 Alexandr $25**

Запуск клиента. Игра

При запуске приложения клиента, пользователю в первую очередь необходимо подключиться к удалённому серверу и создать нового игрока.

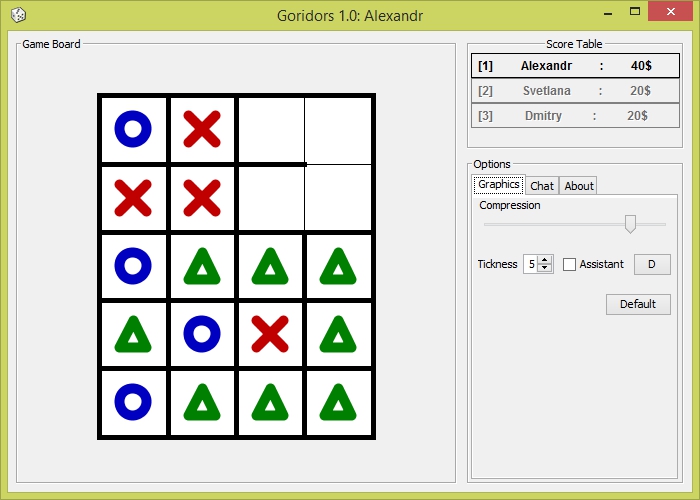
Для этого в появившемся диалоговом окне необходимо выбрать rmi-адрес сервера, имя, а также цвет и форму фигуры которой вы будете отмечать захваченные клетки. Всего в приложении определено четыре фигуры: крестик, круг, треугольник и квадрат.

В случае успешного соединения с удалённым сервером будет выведено сообщение. Далее будет открыто основное игровое окно, но все компоненты на нём будут отключены, пока сервер не начнёт игру.

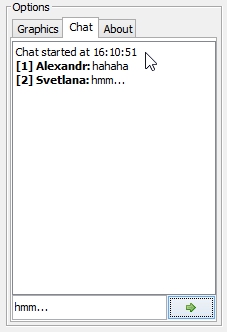
Как только игра начнётся в главном окне появиться три компоненты.

**Игровое поле (Game Board)** - непосредственно, сама доска на которой захватываются клетки. Когда приходит очередь делать ход, у пользователя появляется возможность выбрать (выделить) одну из невыделенных стен клетки. При наведении курсора на тонкую стену клетки – стена выделяется и курсор меняется. Клик по стене оставляет её выделенной. В случае, если по правилам игрок захватил клетку, даётся дополнительный ход, а на его счёт записывается 5 очков.

**Таблица результатов (Score Board)** – в таблице отображаются все принимающие в игре участие игроки, и их количество очков. Игрок делающий текущий ход подсвечивается чёрным цветов.

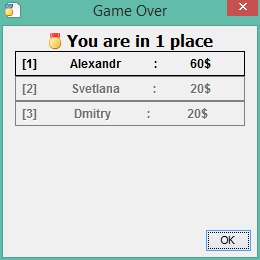
Пример работы клиента:-

У пользователя также есть некоторые дополнительные возможности во вкладках **Опции (Options).** Во вкладке *Графика (Graphics)* можно настроить некоторые параметры рисования игрового поля.

* Compression – позволяет регулировать размеры поля относительно компоненты Game Board.
* Tickness – ширина линий выделенных (выбранных) стен клеток.
* Assistant – включает подсветку информации о выделенной клетке (игрок которому она принадлежит, номер клетки и т.д.)
* Кнопка D – включает альтернативный режим отображения.
* Кнопкой Default – позволяет сбросить все настройки до изначальных.

Во вкладке *Чат (Chat)* у игроков есть возможность пообщаться друг с другом.

И во вкладке *О\_программе* (About) описаны правила игры.

По завершению игры пользователю будет представлено диалоговое окно с результатами игры.

После чего приложение будет закрыто.

Руководство программиста

Для реализации данной задачи использовались технологии Java RMI и Swing (для графического интерфейса).

Полные исходные коды последней версии можно найти в репозитории <https://github.com/AlexandrAratsky/java_rmi_game.git>

Поскольку распределённая система представляет собой две независиме друг от друга программы, рассмотрим их по отдельности.

Серверное приложение

### Структура программы

Программный код сервера условно можно разбить на две части.

Первая часть реализует непосредственно механизм игры. Именно в ней содержаться классы отвечающие за игровое поле, игроков и т.д. Включает пакеты:

* **game.enigne**  - содержит классы реализующие механику игры;
* **game.enigne.callbacks –** содержит интерфейс вызовов для обратной связи;

Вторая – реализует удалённый сервер. Включает пакеты:

* **game.server –** содержит удалённые интерфейсы;
* **game.server.impl –** содержит реализации удалённых интерфейсов;

Наиболее содержательний по классам пакет - **game.enigne.**  Такие классы:

* *BoardWall* – объекты, реализуемые этим классом, хранят в себе одно из двух состояний у стены игровой клетки (есть выделенная стена, нет стены).
* *BoardCell* – класс реализации, непосредственно игровой клетки. Содержит в себе четыре ссылки на объекты BoardWall (стены клетки) и ссылку на игрока (объект класса Player), которому принадлежит данная клетка (изначально клетка свободна, то есть ссылка содержит нулевой указатель).
* *Side* – вспомогательный касс перечисление стен (правая, левая, верхняя, нижняя).
* *Player* – класс содержащий различную информацию об игроке (индикационный номер, имя, фвет и форма фигуры, количество очков).
* *PlayerComp* – вспомогательный клас-компаратор для сортировки игроков по очкам.
* *GameBoard* – непосредственно главный класс в котором реализован механизм игры. Игровое поле представляет собой двумерный массив из BoardCell (клеток). Каждая клеток ссылается на созданные для неё стены (смежные клетки ссылаются на одну и ту же граничащую стену). Содержит внутреннюю очередь игроков. Внутри класса реализованы все методы по необходимой обработке игрового поля во время игры. Такие как выбор следующего игрока, обработка хода, проверка поля на конец игры и т.д.

Для реализации обратной связи с GameBoard в пакете **game.enigne.callbacks** определён интерфейс с методами обратного вызова

* *BoardCallbacks* – интерфейс содержащий callback’и для класса GameBoard.

Подробнее о реализации коллбэков и UML Диаграмма классов пакета чуть ниже.

Часть, содержащая реализацию удалённого сервера, состоит из четырёх классов.

* *GameServer* – непосредственно удалённый интерфейс предосталяющий методы приложению клиенту.
* *ServerCallbacks* – интерфейс определяющий набор функция обратного вызова (callback’ов) для обратной связи с сервером.
* GameServerImpl – основной класс, реализация удалённого интерфейса. Содержит в себе метод main для запуска сервера (себя же).
* *ServerCallbacksAdapter* – вспомогательный класс для реализации callback’ов.

Следует заметить, что все классы объекту, которых передаются по сети наследуют интерфейс сериализации. Подробнее рассмотрим структуру сервера.

### Описание алгоритмов

Разделение программы на часть, отвечающую за механизмы игры и сам сервер позволила уменьшить взаимосвязи между реализацией механики и реализацией связи с клиентом. Принцип работы сервера заключается в том, что внутри его класса определена локальная переменна *GameBoard*, с которой и взаимодействуют клиенту (для отрисовки, для хода и т.д.). При таком подходе часть методов класса *GameBoard* серверу приходится делегировать, но это позволяет свободно менять остальную часть поведения.

Интерфейс удалённого объекта повторяет некоторые методы класса *GameBoard:*

|  |
| --- |
| **public** **interface** GameServer **extends** Remote {    **public** **static** **String** RMI\_SERVER = "rmi://localhost/";  **public** **static** **String** RMI\_NAME = "GameServer";    **public** int getWidht() **throws** **RemoteException**;  **public** int getHeight() **throws** **RemoteException**;    **public** Player getCellPlayer(int i, int j) **throws** **RemoteException**;  **public** BoardWall getCellWall(int i, int j, Side side) **throws** **…**  **public** **String** getCellInfo(int i, int j) **throws** **RemoteException**;  **public** **String** getCellPlayerInfo(int i, int j) **throws** **RemoteException**;  **public** int getCurrentPlayerID() **throws** **RemoteException**;  **public** **List**<Player> getPlayerslist() **throws** **RemoteException**;  **public** void cellClick(int i, int j,Side side) **throws** **RemoteException**;  **public** void addCallbackListener(ServerCallbacks listener) …  **public** int addPlayer(**String** name, int figure, **String** colorS) **throws**…  **public** void newMessage(**String** name, **String** text) **throws…**  } |

Такие методы как getWidht или getHeight имеют реализацию:

|  |
| --- |
| @Override  **public** int getWidht() **throws** RemoteException  { **return** board.getWidht(); }  @Override  **public** int getHeight() **throws** RemoteException  { **return** board.getHeight(); } |

Но большая часть методов реализует алгоритмы непосредственно сервера. Например, метод addCallbackListener(ServerCallbacks listener) наобходим для обратной связи с сервером. Помимо игральной доски сервер содержит внитри себя список из интерфейсов ServerCallbacks, непосредственно реализации которых передают ему его клиенты (используя для этого класс ServerCallbacksAdapter). При наступлении некоего события сервер используюя методы интерфейса ServerCallbacks отправляет запросы всем его слушателям-клиентам (то есть исполняет код реализованный клиентами). Это есть классическая реализация паттерна Наблюдатель (Observer).

|  |
| --- |
| **public** **interface** ServerCallbacks **extends** Remote, Serializable {  **public** void boardChanege() **throws** **RemoteException**;  **public** void nextPlayer(Player next) **throws** **RemoteException**;  **public** void updateScore(int playerID, int score) **throws** **…**  **public** void startGame() **throws** **RemoteException**;  **public** void newMessageForChat(**String** name, **String** text) **throws** **…**  **public** void gameOver(int idWinner) **throws** **RemoteException**;  }  //абстрактный класс адаптер с пустыми реализациями методов для использование клиентом  **public** **abstract** **class** ServerCallbacksAdapter **extends** UnicastRemoteObject **implements** ServerCallbacks { … |

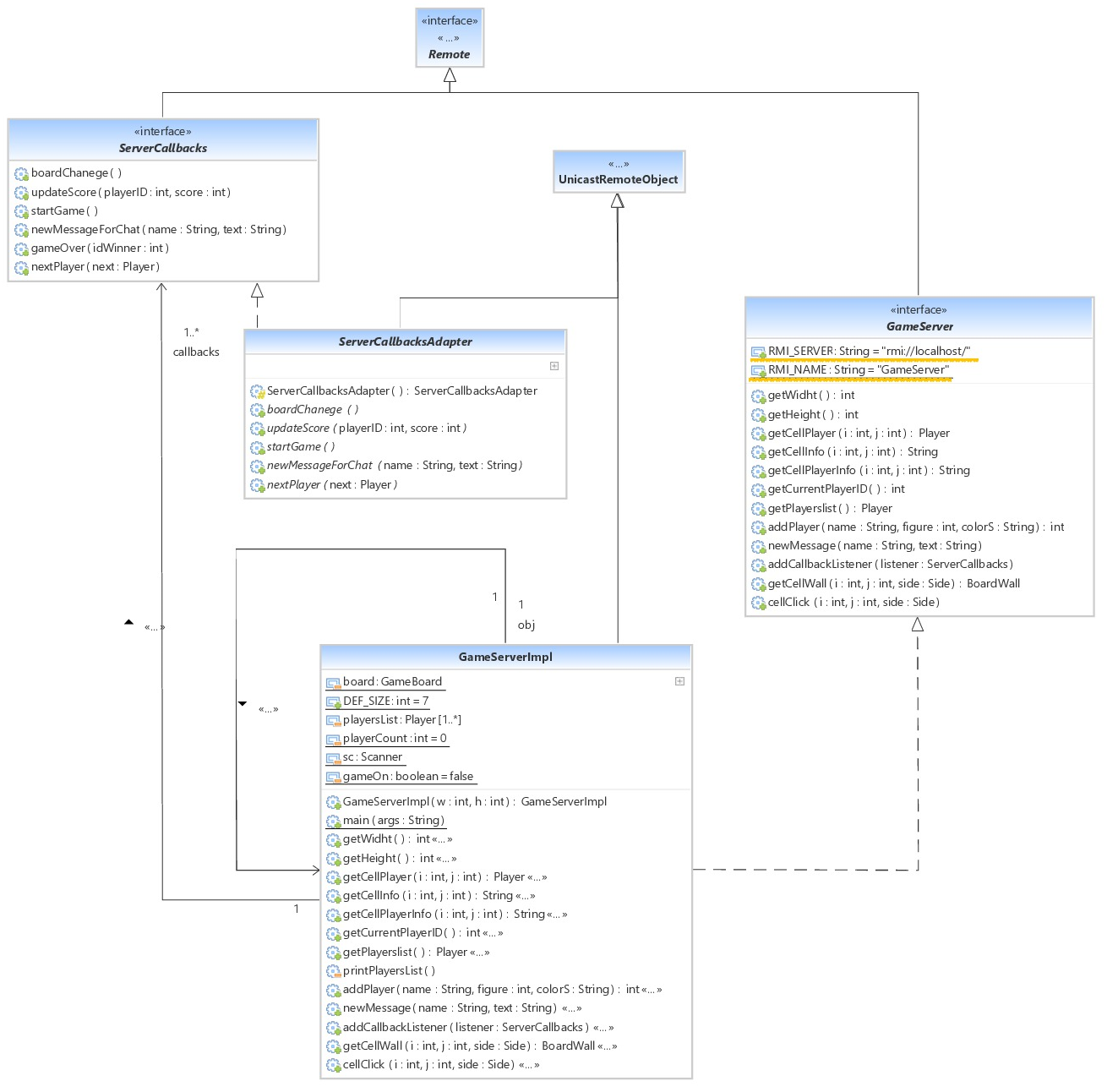
Код отправляемый на сервер клиентом:

|  |
| --- |
| server.addCallbackListener(new ServerCallbacksAdapter() {  @Override  **public** void boardChanege() **throws** **RemoteException** {…}  @Override  **public** void nextPlayer(Player next) **throws** **RemoteException** {…}  @Override  **public** void updateScore(int ID, int score) **throws** **RemoteException** {…}  @Override  **public** void startGame() **throws** **RemoteException** {…}  @Override  **public** void gameOver(int idWinner) **throws** **RemoteException** {…} }); |

Создание сервера идентично тому, что было описано в примере об RMI.

### UML диаграмма классов





Остальные классы связей друг с другом не имеют.

Клиентское приложение

Клиентское приложение для отображения библиотеку Swing. Для удобства все классы разделены на следующие пакеты:

* **game.client** - содержит класс точку входа клиентского приложения;
* **game.client.gui –** содержит два основных классасодержащих форму основного окна приложения;
* **game.client.gui.elements** - содержит компоненты для основного окна игры;
* **game.client.gui.drawing –** вспомогательные классы для отрисовки доски;

Запуск клиентского приложения начинается из класса *GameClient* пакета **game.client**. В ним происходит создания диалогового окна *JClientDialog*, где пользователь задаёт свои параметры. После чего происходит получение удалённого сервера и, в случае успешного подключения, создаётся главная форма *JMainWindow*, содержащая в себе все необходимые компоненты для отображения. В конструктор формы передаётся полученный сервер и далее клиент взаимодействует с сервером через форму. *JClientDialog* и *JMainWindow* содержатся в пакете **game.client.gui.**

Опишем другие классы. Пакет **game.client.gui.elements**:

* *JBoardArea* – компонента JPanel, отрисовывающая игровую доску, и реализующая взаимодействие с ней;
* *JChatPanel* – компонента JPanel, реализующая интерфейс чата;
* *JColorComboBox* – компонента JComboBox с возможностью выбора цвета;
* *JEndDialog* – форма JDialog выводящаяся при конце игры;
* *JOptionsPanel*– компонента JPanel, содержащая панель настройки графики;
* *JPanelAbout* – компонента JPanel с правилами игры;
* *JScoreTable* – компонента JPanel с результатами игроков;
* *JPlayerNote* – запись игрока в *JScoreTable*.

Пакет **game.client.gui.drawing** содержит вспомогательные классы для рисование игрового поля в классе *JBoardArea*;

Заключение

Для полноценной реализации распределённых приложений требуется достаточно глубокое проектирование. И от качества выбранной архитектуры программы зависят выполнения многие требования. Полученный вариант архитектуры игры, является масштабируемой и может выполняться параллельно. Поскольку распределяемый ресурс (в данном случае игровой движок) не является критичным ресурсом, то есть доступ к нему могут осуществлять в любой момент времени, а изменение контролируется сервером система обладает некой базовой управляемостью. Безусловно многие требование были реализованы ещё в технологии Java RMI (напр. Прозрачность).

Сейчас, технология RMI уже не в тренде, поскольку существует множество других альтернатив, включая различные самодельные реализации. Однако она по прежнему хорошо подходит для решаемых задач и, может облегчить для начинающего Java программиста вход в мир распределённого программирования.

Список литературы

1. “Введение в распределенные вычисления” М.С. Косяков 2014
2. Основы RMI – JavaGuru.ru <http://javagu.ru/portal/dt?last=false&provider=javaguru&ArticleId=GURU_ARTICLE_81127&SecID=GURU_SECTION_80694>
3. Threads and Callbacks in RMI

<http://letrungthang.blogspot.ru/2010/08/threads-and-callbacks-in-rmi.html>