МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ УКРАИНЫ

Таврический национальный университет

имени В.И.Вернадского

Факультет математики и информатики

Кафедра прикладной математики

Специализация ………………………………………………….

**Система электронного документооборота**

**Система електронного документообігу**

Магистерская работа

**Абоимов Сергей Александрович,**

Студент 5 курса специальности прикладная математика

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Научный руководитель:

кандидат физ.-мат.наук **Козлов А.И.**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Работа допущена к защите:**

Заведующий кафедрой

доктор физ.-мат.наук,профессор **Чехов В.Н.**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Симферополь, 2010г.

Оглавление

[1. Введение 4](#_Toc262641511)

[2. Работа приложения 6](#_Toc262641512)

[2.1. Архитектура программы qMessenger. 9](#_Toc262641513)

[2.2. Архитектура Серверной части приложения. 11](#_Toc262641514)

[3. ПОТОКИ 15](#_Toc262641515)

[3.1. Методы synchronized 16](#_Toc262641516)

[3.1.1. Операторы synchronized 18](#_Toc262641517)

[3.2. Методы wait и notify 19](#_Toc262641518)

[3.3. Подробности, касающиеся wait и notify 21](#_Toc262641519)

[3.4. Взаимная блокировка 22](#_Toc262641520)

[3.5. Приостановка потоков 24](#_Toc262641521)

[3.6. Прерывание потока 24](#_Toc262641522)

[3.7. Завершение работы потока 25](#_Toc262641523)

[3.8. Завершение приложения 26](#_Toc262641524)

[3.9. Использование Runnable 27](#_Toc262641525)

[3.10. Ключевое слово volatile 27](#_Toc262641526)

[3.11. Отладка потоков 28](#_Toc262641527)

[4. Архитектура модель вид контроллер(MVC model-view-controller) 29](#_Toc262641528)

[4.1. Класс Observable и интерфейс Observer 30](#_Toc262641529)

[4.2. Выводы о модели MVC 31](#_Toc262641530)

[5. Базы данных Java Database Connectivity(JDBC) 33](#_Toc262641531)

[5.1. Структура базы в Серверной части приложения. 34](#_Toc262641532)

[5.2. Хранимые процедуры 35](#_Toc262641533)

[5.3. Использование хранимых процедур в qMessenger 37](#_Toc262641534)

[6. Заключение 41](#_Toc262641535)

[7. Приложение 42](#_Toc262641536)

[7.1. Проблема с сокетами. 42](#_Toc262641537)

[7.2. Подключение пользователей. 45](#_Toc262641538)

[7.3. Обработка Xml. 47](#_Toc262641539)

[8. Список использованной литературы 49](#_Toc262641540)

# Введение

В наше время, всё чаще и чаще используются различные электронные системы. Независимо от степени сложности функционала программы, будь то почтовая программа или программа управления космической станцией, разработчики программного обеспечения стремятся сделать свои программы как можно более простые в обращении, максимально эффективными, что бы программа была переносима и имела возможность запуска на разных операционных системах и работали максимально быстро.

Программное обеспечение разработанное в этой работе было направленно на то что бы максимально упростить «документооборот». Конечно же, оборот может быть не только документов, а и любых файлов фильмов, музыки игр. При всём при этом нужно было ориентироваться на обыкновенного пользователя, который не желает проходить бесконечные регистрации или мучатся в настройках программы или же для отправки одного документа 10 пользователям должен сделать больше 10 кликов мышью.

В виду того что на сегодняшний день созданного много операционных систем, хотя большая часть пользователей всё же пользуются OC Windows, всё же существует хоть и не столь большой процент людей которые пользуются другими операционными системами. Обычно такие пользователи находятся в различного рода учреждениях, которые не в состоянии позволить купить большое число копий OC Windows. Либо только часть учреждения имеет одну операционную систему а другая часть другую. Вот на таких пользователей и рассчитана программа разработанная в данной работе. Поскольку реализация клиентской части и серверной части написана полностью на Java это позволяет сделать возможным запуск программы практически под все известные платформы. Стоит лишь зайти на официальный сайт Sun (www.sun.com)и скачать последнюю версию Java Runtime Environment для запуска приложения.

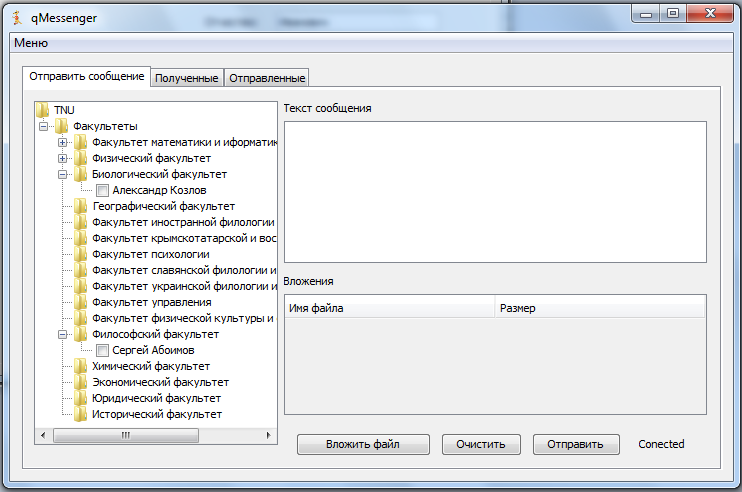
На сегодняшний день существуют и более менее поддерживаются две технологии написания кросс платформенных приложений. Это Java и Qt. Разница в том, что программы написанные на Java работают по принципу, что для работы программы нужно скомпилировать её и потом запускать под любой платформой. Приложения написанные на Qt работают несколько иначе, для каждой ОС Qt поставляет свой набор классов, и что бы сделать возможным запуск приложения нам нужно скомпилировать приложение под определенной ОС. Недостаток из за которого моя работа написана на Java, это то что под разными OC приложение выглядит по разному. То есть, нету гарантии того что под другой операционной системой какой ни будь шрифт или кнопка перекроет другую кнопку и весь интерфейс станет не пригодным для использования. И не мене важно то что Java содержит в себе намного больше компонентов чем Qt.

Поэтому целью работы стало написать приложение клиент – сервер, который может работать на большинстве операционных систем, и не иметь дефектов как в графической части так и в серверной из за специфики операционной системы.

# Работа приложения

Программа qMessenger совмещает в себе почтового клиента и программу быстрого обмена сообщениями. В зависимости от ситуации программа либо сохраняет сообщения на сервере либо пересылает их напрямую получателю.

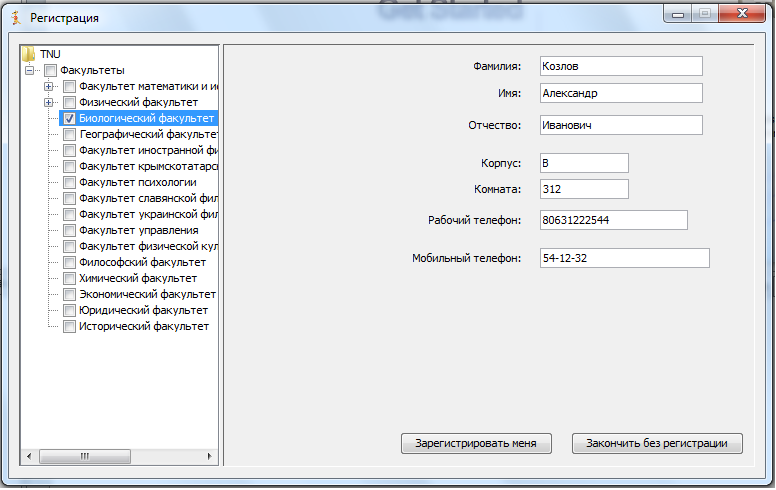
Программа qMessenger имеет простой и удобный интерфейс. Вы можете легко отправить сообщение одному или многим пользователям. Даже если пользователи в момент отправки сообщений недоступны, то они получат их как только запустят программу. Тоже самое касается файлов. Файлы при отправке тоже сохраняются на сервере, и в любой удобный момент пользователь, которому их отправили, всегда сможет их загрузить.



В программе qMessenger имеется возможность сохранять все отправленные и полученные сообщения на сервере, что бы в дальнейшем была возможность их загрузить и просмотреть.

Как видно на рисунке выше, список пользователей имеет древовидную структуру, что облегчает процесс поиска необходимого пользователя, для отправки ему сообщений. Также можно заметить, что в отличии от программ быстрого обмена сообщениями, в программе qMessenger можно отправлять сообщения и файлы сразу многим пользователям, что очень удобно. И в отличии от почтовых программ, вам не нужно помнить email всех пользователей что бы отправить им сообщения, а также это избавляет пользователей от весьма сложной для многих регистрации и настройки почтовых программ, а также ограничений на размер файлов при отправке.

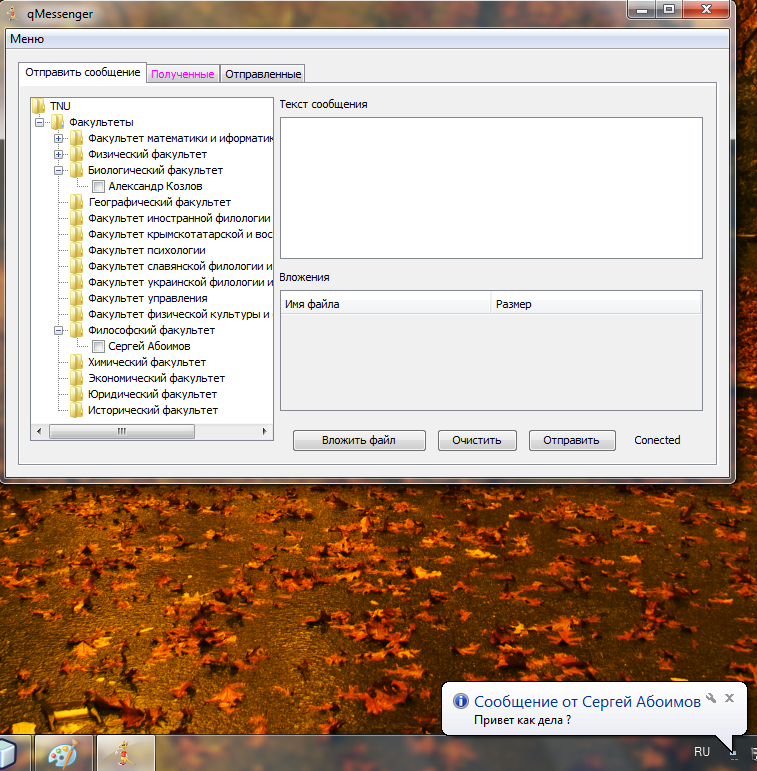
На рисунке ниже изображена простая регистрационная форма qMessenger.

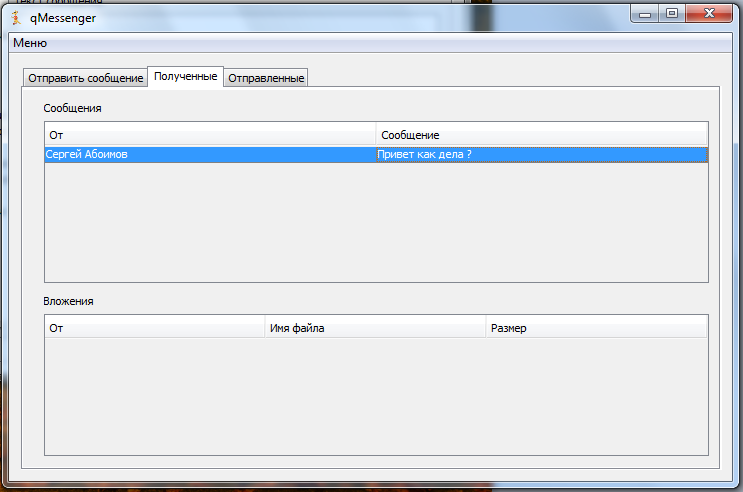


Достаточно выбрать факультет из дерева слева и нажать кнопку «зарегистрировать меня». После чего вы сразу имеете возможность общаться с другими зарегистрированными пользователями.

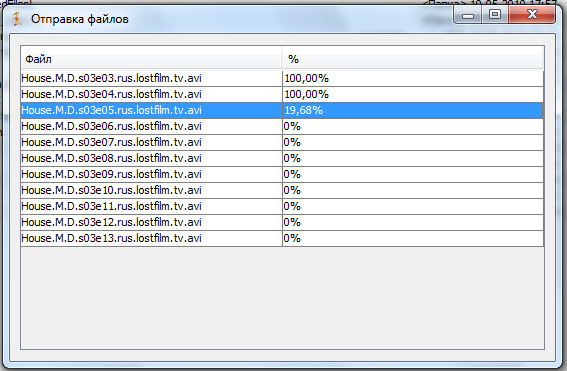
Важно заметить, что с одного компьютера не будет возможности запуска 2х программ с разными пользователями, поскольку при регистрации пользователя используется как идентификатор Ip адрес компьютера.

После регистрации, вы можете начинать посылать сообщения. Как только вам кто то пришлет сообщения то вы увидите как цвет вкладки «полученные» изменился а также можете заметить, подсказку из системного трея, что вам кто то написал. Это изображено на рисунке ниже.





Прием и отправка файлов выглядит следующим образом



Как видно из рисунка, мы может контролировать какие файлы были переданы/полученны.

Если закрыть это окно то передача файлов прекратится, а уже переданные файлы будут доступны клиенту. Только если вы принимаете файлы и случайно закроете окно, то вы всё равно сможете позже их загрузить.

Программа qMessenger имеет опцию сворачивания в трей, что бывает очень полезным, поскольку так она меньше всего отвлекает пользователя от работы. А из трея при приходе сообщений она будет уведомлять пользователя.

## Архитектура программы qMessenger.

Для отображения графических форм используется библиотека Swing. Более детально об этой библиотеке рассказывается в другой главе, и приводится простой пример для понимания.

Преимущество Swing перед ранее используемой библиотекой в программе qMessneger, SWT состоит в том, что компоненты Swing в разных операционных системах выглядят одинаково и имеют одинаковый размер. А также очень существенно то что, в Swing графический поток работает независимо от потока который вополняет логику приложения, что существенно облегчает разработку и возникновение блокирующих ситуаций при синхронизации потоков. (подробно о блокирующих ситациях и потоках в другой главе)

Вся бизнес логика qMessenger находится в package clientapp.

Как только запускается программа, сразуже создается экземпляр класса User, который является наследником класса Thread, и когда вызывается метод Connect автоматически создаётся отдельный поток в котором программа подключается к серверу и посылает команды серверу.

Класс User это основной класс программы, через который вся программа работает. Он служит связующим звеном между отправкой сообщений на сервер, а также принятие сообщений с сервера, логикой программы и графической частью. И для удобства, это класс доступен из глобального контекста. Этот класс содержит все необходимые методы для работы клиента с сервером. А поскольку он является связующим звеном между всеми компонентами программы, то можно легко контролировать логику программы, либо находить ошибки.

Поток который запускает класс User делает только одно : слушает сообщения от сервера. Как только сервер что то посылает клиенту, этот поток реагирует и запускает соответсвующие механизмы обработки данной команды и в случае необходимости синхронизирует действия с графическим потоком.

Связь с сервером реализовано с помощью сокетов. Для этого был реализован класс Messages в конструктор которого передаются два сокета SocketIn, SocketOut. Первый для сообщений от сервера, а второй для сервера. Таким образом имея два подключения, мы можем асинхронно общаться с сервером. Работает этот класс следующим образом. Сначала вычисляется длина отправляемого сообщения, и записывается в виде десятичного числа, далее следует разделительный символ (FormatCharacters.marker) а далее само сообщение( в нашем случае это просто строка). Таким образом, на сервере мы очень просто можем прочитать наше сообщение. Мы просто считываем символы до разделительного (FormatCharacters.marker), а потом читаем ровно то количество символов которое было послано. Именно так реализованы в программе метаданные.

Для удобства работы, и скрытия в дальнейшем методов отсылки метаданных и преобразования строк для отправки, был создан класс наследник FormatedMessages. Этот класс уже содержит необходимый набор методов для общения с сервером. Для того что бы вызвать соответствующее поведение сервера нужно сделать следующее. Сначала сделать запрос функционала( например мы говорим серверу что хотим отправить сообщение, для этого все зарезервированные строки сохранены в классе FormatCharacters, и мы посылаем серверу строку FormatCharacters. TextMessege), а дальше вторым запросом отправляем сообщение. В дальнейшем с развитием программы, стала необходимость отправлять не просто сообщение как строку, а еще дополнительный набор атрибутов, например время сообщения, списко пользователей для получения и так далее. Поэтому была использована сериализация. Сериализация это преобразование объекта в какой либо вид( в нашем случа это бинарное представление ) с последующим восстановлением. Поэтому всё что нам нужно сделать это сериализовать объект в бинарный массив отправить этот массив на сервер, а на сервере восстановить объект. Таким образом это избавляет проект от лишних строчек кода, которые нужно писать для того что бы анализировать метаданные для каждого объекта. А так мы имеет общий алгоритм который преобразует класс в бинарный вид и потом восстанавливает его. Эти методы находятся в классе CommunicationBase. В томже packag'е находятся классы которые служат связующими между сервом и клиентом, т.е те которые используются для сериализации и десериализации.

Класс GLobal представляет собой глобальный контекст приложения, в котором доступны различные настройки приложение, например адрес сервера, иконка программы и тд, а также в нем содержится экземпляр класса User, который очень часто используется в разных частях программы.

Класс ApplicationSetting используется для чтения и сохранения настроек программы в файл.

Класс ScreenView это связующее звено между логикой, сервером и графическим интерфейсом. Все изменения которые приходят с сервера, или делаются логикой используют этот класс для синхронизации логики и интерфейса пользователя.

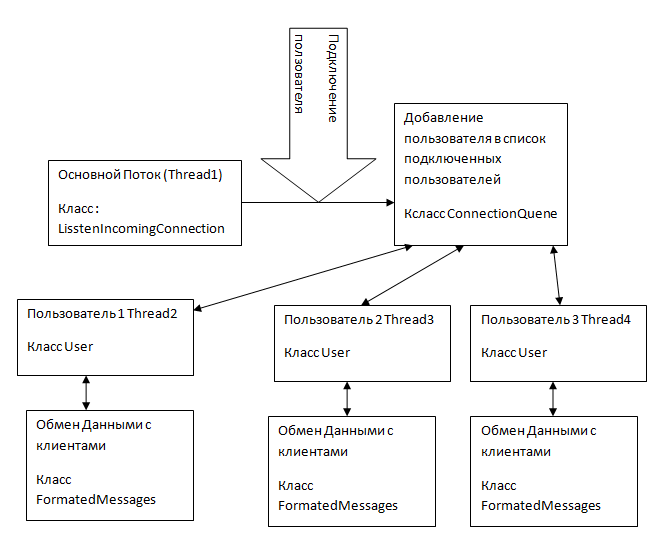
## Архитектура Серверной части приложения.

В отличии от клиента, сервер не имеет графической части. Он работает как скрытый процесс. Аналогично клиенту, в сервере основным классом, с помощью которого взаимодействует сервер является User. Только с тем отличием, что созданных объектов такого класса будет равно количеству пользователей присоединённых к серверу. Каждый класс User является наследником класса Thread, и следовательно при подключении нового пользователя будет создаваться новый объект класса User и поток.

Для взаимодействия пользоватлей между собой, разработан класс ConnectionQuene который содержит все созданные объекты класса User. Этот класс предоставляет механизмы взаимодействия пользователей между собой, например рассылка сообщений, файлов и так далее. Что бы каждый объект класса User мог взаимодействовать с другими объектами User, в конструктор класса User необходимо передать ссылку на объект ConnectionQuene .

Каждый объект класса User создает объект FormatedMessages который как и на клиентской части служит классом связи между клиентом и сервером. Таким образом, при подключении нового пользователя, создаётся новый поток, в нем создается объект класса User, и объект класса FormatedMessages для обмена данных с каждым подключенным клиентом.

На рисунке ниже приведена схема взаимодействия классов.



Для каждого пользователя, открывается два сокетных соединения, для возможности асинхронного обмена данных. Синхронная возможность также присуцтвует и часто используется когда это необходимо.

Асинхронная часть реализована так: у каждого пользователя есть два класса SocketIn, SocketOut. SocketIn постоянно слушается сервером, что бы среагировать на запрос клиента. SocketOut служит для того что бы сервер когда необходимо, мог послать команду клиенту.

Синхронная часть работает так: если запрос приходит из SocketIn то ответ записывается тоже в SocketIn, и наоборот. На сервере и клиенте реализованы 4 метода для работы синхронно и асинхронно. Это SendMessage, SendMessageSync, ReceiveMessage, ReceiveMessageSync. Тут SendMessage и ReceiveMessageSync используют один сокет, ровно как и ReceiveMessage ,SendMessageSync. Поэтому если вам необходимо послать сообщение и получить ответ синхронно то нужно использовать первую пару, соответственно что бы получить сообщение и ответить синхронно то нужно использовать вторую пару.

Работать асинхронно тоже весьма просто. Вы посылаете сообщение например SendMessage. И добавляете функцию которая будет вызываться когда сервер ответит, вызвав вашу функцию на клиенте.

Реализация приема и отправкли файлов выглядтит следующим образом. Клиен, через равные промежутки времени делает запрос на сервер, о наличии файлов доступных для скачивания. Если таковые существуют, то сервер передаёт список уникальных идентификаторов клиенту, по которым клиент может в дальнейшем запросить эти файлы у сервера. Для згрузки файлов, клиенту нужно сделать обращение на определенный порт сервера, и запросить файл. Для этого клиент создает отдельный поток, что бы загрузка файлов не обмену сообщениями и другой информацией, и запрашивает по уникальному идентификатору (который был ранее получен от сервера) файл. Сервер в это время слушает всех клиентов подключенных на определенный порт, и при наличии подключения, высылает клиенту файл, соответствующий запрошенному идентификатору.

Все файлы хранятся на сервере, а также, записи о файлах и идентификаторы хранятся в базе данных. Информация хранящаяся в базе представляет собой уникальный идентификатор файла, время отсылки файла, контрольная сумма файла(для проверки правильности передачи) а также бит, который говорит о том что были ли доставлены файлы клиенту и идентификатор пользователя пославшего файл. Таким образом очень просто получить список всех файлов которые необходимо отправить заданному пользователю. Реализация приема файлов на сервере содержится в класса ReceiveFiles, а отправки в SendFiles.

Для работы с базой данных реализован класс dbFunc, каждый метод которого принимает экземпляр класса dbConnection. dbConnection это класс который создает независимое подключение к базе и использует его для получения данных. Это сделано для того что бы можно было использовать в нужных местах много подключений или наоборот одно подключение для всех действий. Все функции в dbFunc используют хранимые процедуры. Это обосновывается тем, что повышается скорость работы, а также мы избавляемся от того что бы делать много запросов подряд, которые нацелены выполнить 1 функцию и хранимые процедуры обеспечивают дополнительную бизнес «прослойку» между логикой сервера и базой данных.

Дерево факультетов хранится в Xml файле. При запуске сервера загружается этот файл и анализируется. Этот файл дожжен иметь для каждого xml тэга уникальный идентификатор(id) а также, не обязательный атрибут fullName в который можно написать название факультета или структурного подразделения на любом языке. При загрузке сервер анализирует каждый тэг и смотрит наличие таких в базе используя идентификатор. Если такой идентификатор существует в базе, то имя тэга обновляется в базе, если не существует то создается новая запись. Также в базу сохраняется список ребер дерева, для того что бы можно было определить полный путь пользователя. Ребра сохраняются просто номерами идентификаторв. Например ели корневая вершина имеет идентификатор 1 и у корневой вершины будет 3 потомка с идентификаторами 2,3,4, то в базу запишется 3 записи это (1,2) (1,3) (1,4). Таким образом имея список ребер очень просто будет найти полный путь пользователя от корневой вершины до того места в дереве где он находится. Функциональность создающая такие записи в базе и находящая путь реализована в классе BuildTreePath.

Класс GenerateUserTree отвечает за динамическое построение дерева, в узлах которого отображаются зарегистрированные пользователи. Это реализовано достаточно просто. Мы рекурсивно обходим дерево, и в каждом узле дерева делаем запрос в базу данных. Если в базе существует пользователь, которому назначен данный идентификатор вершины, то мы создаем вершину в Xml дереве с информации о пользователе(ях). После чего этот сгенерированный xml файл отправляется клиенту. Примечание: поскольку новый клиент может зарегистрироваться в тот момент когда уже другие клиенты подключены к серверу, то сервер сам автоматически рассылает вновь созданное дерево всем пользователям, что бы подключенные пользователи смогли увидеть вновь зарегистрированного пользователя.

# ПОТОКИ

Большинство программистов привыкло писать программы, которые выполняются шаг за шагом, в определенной последовательности.

Без использования потоков невозможно написать программу подобную qMessenger. Особенно это касается серверной части. Серверу приходится обслуживать много клиентов одновременно, и поэтому написать серверную часть без концепции потоков невозможно. Тем более с развитием многопроцессорных систем работа с потоками стала еще более актуальна.

Поток может работать независимо от других потоков. Потоки также осуществляют совместный доступ к объектам.

Совместный доступ одновременно является и одним из самых полезных свойств многопоточности, и источников самых больших проблем. При использовании приведенной схемы “выборка-изменение-запись” возникает потенциальная опасность того, что при одновременной работе двух потоков с одним и тем же объектом произойдет наложение, приводящее к разрушению объекта.

Многие реальные задачи программирования лучше всего решаются с применением нескольких потоков. Например, интерактивные программы, предназначенные для графического отображения данных, нередко разрешают пользователю изменять параметры отображения в реальном времени. Оптимальное динамическое поведение интерактивных программ достигается благодаря использованию потоков. Если рассмотреть как пример qMessnger, его клиентскую часть, то не получилось реализовать отправку файла и при этом возможность общения с другими пользователями одновременно.

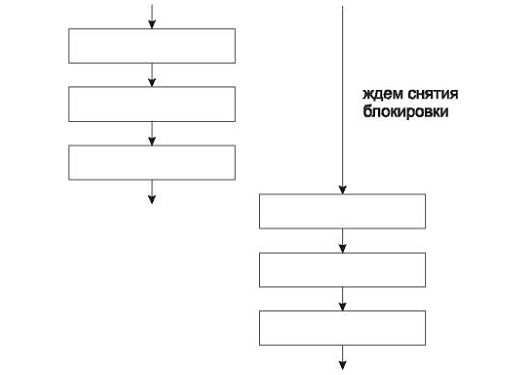
В однопоточных системах иллюзия работы с несколькими потоками обычно достигается за счет использования прерываний или программных запросов (polling). Программные запросы служат для объединения частей приложения, управляющих отображением информации и вводом данных. Особенно тщательно должна быть написана программа отображения — запросы от нее должны поступать достаточно часто, чтобы реагировать на ввод информации пользователем в течение долей секунды. Эта программа либо должна позаботиться о том, чтобы операции графического вывода занимали минимальное время, либо прерывать свою собственную работу для выполнения запросов. Например, в библиотеке Swing автоматически при активации любого графического компонента создаётся поток, и поэтому в независимости от того сколько потребует времени выполнения та или иная задача, интерфейс будет всегда доступен. Это одна из важных отличительных особенностей Swing перед Swt.

Один поток обновляет изображение на основе текущих данных, а другой — обрабатывает ввод со стороны пользователя. Если ввод оказывается сложным (например, пользователь заполняет экранную форму), первый поток (вывод данных) может работать независимо, вплоть до получения новой информации. В модели с применением программных запросов приходится либо приостанавливать обновление изображения, чтобы дождаться завершения нетривиального ввода, либо производить сложную синхронизацию, чтобы изображение могло обновляться во время заполнения формы пользователем. Модель с разделением процессов ввода и отображения может поддерживаться в многопоточной системе непосредственно, вместо того чтобы заново подгонять ее для реализации очередной задачи.

## Методы synchronized

В qMessenger эти методы используются для синхронизиции графического интерфейса и выполняющих логику потоков. Например, прикопировании файлов, процесс копирования можно приостановить – тут фактически происходит вмешательство одного потока в другой. Для это и используются методы synchronized.

Чтобы класс мог использоваться в многопоточной среде, необходимо объявить соответствующие методы с атрибутом synchronized (позднее мы узнаем, что же входит в понятие “соответствующие”). Если некоторый поток вызывает метод synchronized, то происходит блокировка объекта. Вызов метода synchronized того же объекта другим потоком будет приостановлен до снятия блокировки.



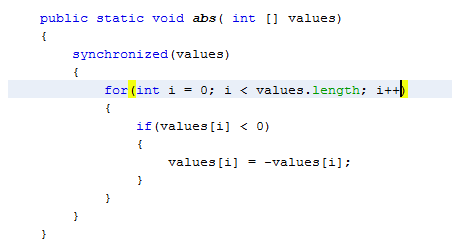
Синхронизация приводит к тому, что выполнение двух потоков становится взаимно исключающим по времени. Проблема вложенных вызовов решается очевидным образом : если синхронизированный метод вызывается для объекта, который ранее был заблокирован тем же самым потоком, то метод выполняется, однако блокировка не снимается вплоть до выхода из самого внешнего синхронизированного метода.

Синхронизация решает проблему, возникающую в нашем примере: если действия выполняются в синхронизированном методе, то при попытке обращения к объекту со стороны второго потока в тот момент, когда с объектом работает первый поток, доступ будет отложен до снятия блокировки.

Методы класса также могут синхронизироваться с использованием блокировки на уровне класса. Два потока не могут одновременно выполнять синхронизированные статические методы одного класса. Блокировка статического метода на уровне класса не отражается на объектах последнего — вы можете вызвать синхронизированный метод для объекта, пока другой поток заблокировал весь класс в синхронизированном статическом методе. В последнем случае блокируются только синхронизированные статические методы. Если синхронизированный метод переопределяется в расширенном классе, то новый метод не обязан быть синхронизированным. Метод суперкласса при этом остается синхронизированным, так что несинхронность метода в расширенном классе не отменяет его синхронизированного поведения в суперклассе. Если в несинхронизированном методе используется конструкция super.method() для обращения к методу суперкласса, то объект блокируется на время вызова до выхода из метода суперкласса.

### Операторы synchronized

Оператор synchronized позволяет выполнить синхронизированный фрагмент программы, который осуществляет блокировку объекта, не требуя от программиста вызова синхронизированного метода для данного объекта. Оператор synchronized состоит из двух частей: указания блокируемого объекта и оператора, выполняемого после получения блокировки. Общая форма оператора synchronized выглядит следующим образом: synchronized (выражение) оператор Взятое в скобки выражение должно указывать на блокируемый объект — обычно оно является ссылкой на объект. После блокировки выполняется оператор — так, словно для данного объекта выполняется синхронизированный метод. Чаще всего при блокировке объекта необходимо выполнить сразу несколько операторов, так что оператор, как правило, представляет собой блок. Приведенный ниже метод заменяет каждый элемент числового массива его модулем, причем доступ к массиву регулируется оператором synchronized:



Массив values содержит изменяемые элементы. Мы синхронизируем доступ к нему, указывая его в качестве объекта в операторе synchronized. После этого можно выполнять цикл и быть уверенным в том, что массив не будет изменен каким-либо другим фрагментом программы, в котором тоже установлена синхронизация для массива values.

От вас не требуется, чтобы объект, указанный как аргумент оператора synchronized, использовался в теле оператора. Можно представить себе ситуацию, при которой единственное назначение объекта заключается в том, чтобы служить для блокировки большего набора объектов. В этом случае объект-представитель может и не обладать собственными функциями, но использоваться во всех операторах synchronized, желающих выполнить действия с некоторыми или всеми объектами из этого набора.

В подобных ситуациях существует и другой подход — спроектировать класс представитель с несколькими синхронизированными методами, служащими для выполнения операций с другими объектами. При таком варианте не только достигается более четкая инкапсуляция операций, но и исчезает возможный источник ошибок — доступ к объектам вне операторов synchronized, вызванный забывчивостью программиста.

Тем не менее иногда с защищаемыми объектами выполняется слишком много операций, чтобы их все можно было инкапсулировать в виде методов класса, и для защиты многопоточного доступа приходится пользоваться оператором synchronized.

Иногда разработчик класса не принимает во внимание его возможное использование в многопоточной среде и не синхронизирует никакие методы. Чтобы применить такой класс в многопоточной среде, у вас имеется две возможности:

* Создать расширенный класс, в котором вы переопределяете нужные методы, объявляете их synchronized и перенаправляете вызовы этих методов при помощи ссылки super.
* Воспользоваться оператором synchronized для обеспечения доступа к объекту, с которым могут работать несколько потоков.

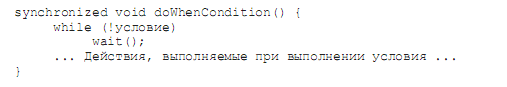
В общем случае расширение класса является более удачным решением — оно устраняет последствия возможной ошибки программиста, забывающего внести доступ к объекту в оператор synchronized. Тем не менее, если синхронизация необходима лишь в одном-двух фрагментах программы, то оператор synchronized предоставляет более простое решение.

## Методы wait и notify

Механизм блокировки решает проблему с наложением потоков, однако хотелось бы, чтобы потоки могли обмениваться информацией друг с другом. Для этого существует два метода: wait и notify. Метод wait позволяет потоку дождаться выполнения определенного условия, а метод notify извещает все ожидающие потоки о наступлении некоторого события.

Методы wait и notify определены в классе Object и наследуются всеми классами. Они, подобно блокировке, относятся к конкретным объектам. При выполнении wait вы ожидаете, что некоторый поток известит (notify) о наступлении события тот самый объект, в котором происходит ожидание.

Существует стандартная конструкция, которой следует пользоваться в работе с wait и notify. Поток, ожидающий события, должен включать что-нибудь похожее на следующий фрагмент:



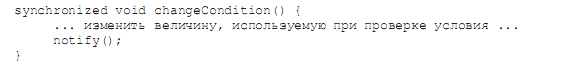
Здесь следует обратить внимание на несколько аспектов:

* Все действия выполняются внутри синхронизированного метода. Это необходимо — в противном случае нельзя быть уверенным в содержимом объекта. Например, если метод не синхронизирован, то после выполнения оператора while нет гарантии, что условие окажется истинным — ситуация могла быть изменена другим потоком.
* Одно из важных свойств определения wait заключается в том, что во время приостановки потока происходит атомарное (atomic) снятие блокировки с объекта. Когда говорят об атомарной приостановке потока и снятии блокировки, имеется в виду, что эти операции происходят вместе и не могут отделяться друг от друга. В противном случае снова возникла бы “гонка”: извещение могло бы придти после снятия блокировки, но перед приостановкой потока. В этом случае извещение никак не влияет на работу потока и фактически теряется. Когда поток возобновляет работу после получения извещения, происходит повторная блокировка.
* Условие всегда должно проверяться внутри цикла. Никогда не следует полагать,

что возобновление работы потока означает выполнение условия. Другими

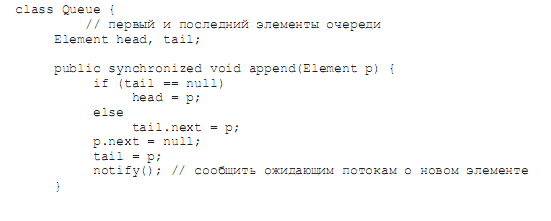
словами, не заменяйте while на if.

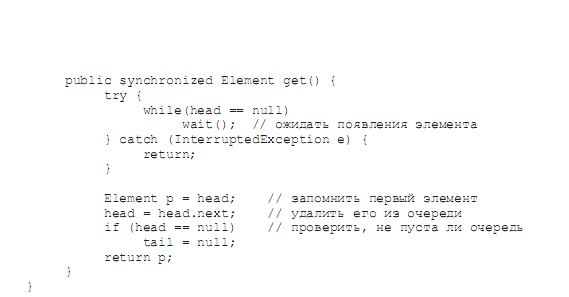
С другой стороны, метод notify вызывается методами, изменяющими данные, которые могут ожидаться другим потоком.



Несколько потоков могут ждать один и тот же объект. Извещение notify возобновляет тот поток, который ждет дольше всех. Если необходимо возобновить все ожидающие потоки, используйте метод notifyAll.

Приводимый ниже класс реализует концепцию очереди. Он содержит методы, которые используются для помещения элементов в очередь и их удаления:





Такая реализация очереди во многом напоминает ее воплощение в однопоточной системе. Отличий не так уж много: методы синхронизированы; при занесении нового элемента в очередь происходит извещение ожидающих потоков; вместо того чтобы возвращать null для пустой очереди, метод get ждет, пока какой-нибудь другой поток занесет элемент в очередь. Как занесение, так и извлечение элементов очереди может осуществляться несколькими потоками (а не обязательно одним).

## Подробности, касающиеся wait и notify

Существует три формы wait и две формы notify. Все они входят в класс Object и выполняются для текущего потока:

**public final void wait(long timeout) throws InterruptedException**

Выполнение текущего потока приостанавливается до получения извещения или до истечения заданного интервала времени timeout. Значение timeout задается в миллисекундах. Если оно равно нулю, то ожидание не прерывается по тайм-ауту, а продолжается до получения извещения.

**public final void wait(long timeout, int nanos) throws InterruptedException**

Аналог предыдущего метода с возможностью более точного контроля времени; интервал тайм-аута представляет собой сумму двух параметров: timeout (в миллисекундах) и nanos (в наносекундах, значение в диапазоне 0–999999).

**public final void wait() throws InterruptedException**

Эквивалентно wait(0).

**public final void notify()**

Посылает извещение ровно одному потоку, ожидающему выполнения некоторого условия. Потоки, которые возобновляются лишь после выполнения данного условия, могут вызвать одну из разновидностей wait. При этом выбрать извещаемый поток невозможно, поэтому данная форма notify используется лишь в тех случаях, когда вы точно знаете, какие потоки ожидают событий, какие это события и сколько длится ожидание. Если вы не уверены в каком-либо из этих факторов, вероятно, следует воспользоваться методом notifyAll.

public final void notifyAll()

Посылает извещения всем потокам, ожидающим выполнения некоторого условия. Обычно потоки стоят, пока какой-то другой поток не изменит некоторое условие. Используя этот метод, управляющий условием поток извещает все ожидающие потоки об изменении условия. Потоки, которые возобновляются лишь после выполнения данного условия, могут вызывать одну из разновидностей wait.

Все эти методы реализованы в классе Object. Тем не менее они могут вызываться только из синхронизированных фрагментов, с использованием блокировки объекта, в котором они применяются. Вызов может осуществляться или непосредственно из такого фрагмента, или косвенно — из метода, вызываемого в фрагменте. Любая попытка обращения к этим методам для объектов за пределами синхронизированных фрагментов, для которых действует блокировка, приведет к возбуждению исключения IllegalMonitorState Exception.

## Взаимная блокировка

Если вы имеете дело с двумя потоками и с двумя блокируемыми объектами, может возникнуть ситуация взаимной блокировки (deadlock), при которой каждый объект дожидается снятия блокировки с другого объекта. Представим себе, что объект X содержит синхронизированный метод, внутри которого вызывается синхронизированный метод объекта Y, который, в свою очередь, также содержит синхронизированный метод для вызова синхронизированного метода объекта X. Каждый объект ждет, пока с другого объекта не будет снята блокировка, и в результате ни один из них не работает. Подобная ситуация иногда называется “смертельными объятиями” (deadly embrace). Рассмотрим сценарий, в соответствии с которым объекты jareth и cory относятся к некоторому классу Friendly:

1. Поток 1 вызывает синхронизированный метод jareth.hug. С этого момента поток 1 осуществляет блокировку объекта jareth.

2. Поток 2 вызывает синхронизированный метод cory.hug. С этого момента поток 2 осуществляет блокировку объекта cory.

3. Теперь cory.hug вызывает синхронизированный метод jareth.hugBack. Поток 1 блокируется, поскольку он ожидает снятия блокировки с cory (в настоящее время осуществляемой потоком 2).

4. Наконец, jareth.hug вызывает синхронизированный метод cory.hugBack. Поток 2 также блокируется, поскольку он ожидает снятия блокировки с jareth (в настоящее время осуществляемой потоком 1).

Возникает взаимная блокировка — cory не работает, пока не снята блокировка с jareth, и наоборот, и два потока навечно застряли в тупике.

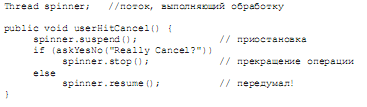
Конечно, может повезти, и один из потоков завершит весь метод hug без участия второго. Если бы этапы 3 и 4 следовали бы в другом порядке, то объект jareth выполнил бы hug и hugBack еще до того, как cory понадобилось бы заблокировать jareth. Однако в будущих запусках того же приложения планировщик потоков мог бы сработать иначе, приводя к взаимной блокировке. Самое простое решение заключается в том, чтобы объявить методы hug и hugBack несинхронизированными и синхронизировать их работу по одному объекту, совместно используемому всеми объектами Friendly. Это означает, что в любой момент времени во всех потоках может выполняться ровно один метод hug — опасность взаимной блокировки при этом исчезает. Благодаря другим, более хитроумным приемам удается одновременно выполнять несколько hug без опасности взаимной блокировки.

Java не умеет ни обнаруживать такую ситуацию, ни предотвращать ее. Подобные проблемы с трудом поддаются отладке, так что предотвращать их надо на стадии проектирования.

## Приостановка потоков

Поток может быть приостановлен (suspended), если необходимо быть уверенным в том, что он возобновится лишь с вашего разрешения. Для примера допустим, что пользователь нажал кнопку CANCEL во время выполнения длительной операции. Работу следует приостановить до того момента, когда пользователь подтвердит (или нет) свое решение.

Фрагмент программы может выглядеть следующим образом:



Метод userHitCancel сначала вызывает suspend для потока, выполняющего операцию, чтобы остановить его вплоть до вашего распоряжения. Затем пользователь должен ответить, действительно ли он хочет отменить операцию. Если да, то метод stop снимает поток; в противном случае метод resume возобновляет работу потока.

Приостановка ранее остановленного потока, а также возобновление работы потока, который не был приостановлен, не приводит ни к каким нежелательным последствиям.

## Прерывание потока

В некоторых методах класса Thread упоминается прерывание (interrupting) потока. Соответствующие методы зарезервированы для возможности, которая вскоре будет включена в Java. Попытка их вызова приводит к возбуждению исключения NoSuchMethodError и уничтожению вызывающего потока. Концепция “прерывания” оказывается полезной, если выполняемому потоку необходимо предоставить некоторую степень контроля над моментом обработки события. Например, в цикле вывода может понадобиться информация из базы данных, извлекаемая посредством транзакции; если при этом поступает запрос на прекращение работы, желательно дождаться нормального завершения транзакции. Поток пользовательского интерфейса может реализовать такой запрос, прерывая поток вывода и давая ему возможность дождаться конца транзакции. Подобная схема будет хорошо работать лишь в том случае, если поток вывода “хорошо себя ведет” и в конце каждой транзакции проверяет, не поступил ли запрос на прерывание (и прекращает работу в этом случае).

Прерывание потока в общем случае не должно влиять на его работу, однако некоторые методы (такие, как sleep или wait) возбуждают исключение InterruptedException. Если в вашем потоке во время прерывания выполнялся один из таких методов, то будет возбуждено прерывание Interrupted Exception.

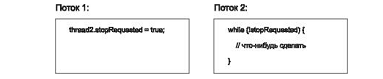
Для работы с прерываниями используются несколько методов. Метод interrupt посылает прерывание в поток; метод isInterrupted проверяет факт прерывания потока; статический метод interrupted проверяет, прерывался ли текущий поток.

## Завершение работы потока

Работа потока прекращается, когда происходит выход из его метода run. Так происходит нормальное завершение потока, но вы можете остановить поток и по-другому.

Желательно использовать самый “чистый” способ: вместо того чтобы насильственно прекращать существование потока, лучше дать ему завершиться добровольно. Чаще всего для этого используют логическую переменную, значение которой опрашивается потоком.

Например:



Самый прямолинейный способ завершить поток — вызвать его метод stop, который запустит объект ThreadDeath, указав ему в качестве цели нужный поток. ThreadDeath является подклассом класса Error, а не Exception. Лучше не перехватывать ThreadDeath, если только не нужно выполнить какие-нибудь чрезвычайно неординарные завершающие действия, с которыми не справится finally. Если уж вы перехватываете ThreadDeath, обязательно нужно объект-исключение заново, чтобы поток мог “умереть”. Если же ThreadDeath не перехватывается, то обработчик ошибок верхнего уровня просто уничтожает поток, не выводя никаких сообщений.

Поток также может возбудить ThreadDeath для самого себя, чтобы завершить свою собственную работу. Это может пригодиться, если поток углубился на несколько уровней ниже метода run и не удается легко сообщить run о том, что пора заканчивать.

Другой форме метода stop можно вместо ThreadDeath передать какое-то другое исключение. Хотя обычно возбуждение исключений оказывается не самым лучшим способом для обмена информацией между потоками, вы можете использовать эту форму общения для того, чтобы послать потоку какое-то сообщение. Например, если некоторый поток выполняет длительные вычисления для определенных входных значений, то интерфейсный поток может разрешить пользователю изменить эти значения прямо во время вычислений. Конечно, можно просто завершить поток и начать новый. Тем не менее, если промежуточные результаты вычислений могут использоваться повторно, то вместо завершения потока можно создать новый тип исключения Restart Calculation и воспользоваться методом stop, чтобы запустить новое исключение в поток. При этом поток должен перехватить исключение, рассмотреть новые входные значения, по возможности сохранить результаты и возобновить вычисления.

Вызов метода destroy для потока — самая решительная мера. Этот метод уничтожает поток без выполнения нормальных завершающих действий, к которым относится и снятие блокировки со всех объектов потока, так что применение destroy может навечно заблокировать другие потоки. По возможности нужно стараться избегать вызова destroy.

## Завершение приложения

Работа каждого приложения начинается с запуска одного потока — того, в котором выполняется метод main. Если ваше приложение не создает других потоков, то после выхода из main оно завершается. Но давайте предположим, что в приложении возникают другие потоки.

Существует две разновидности потоков: потоки-пользователи (users) и потоки-демоны (daemons). Наличие оставшихся потоков-пользователей приводит к продолжению работы приложения, тогда как потоки-демоны могут уничтожаться. После снятия последнего потока-пользователя происходит закрытие всех потоков-демонов, и работа приложения на этом заканчивается. Для пометки потока-демона применяется метод setDaemon(true), а метод getDaemon проверяет значение соответствующего флага. По умолчанию “демонический” статус потока совпадает со статусом его потока-создателя. После того как поток начнет выполняться, изменить данное свойство невозможно; при попытке сделать это возбуждается исключение IllegalThreadState Exception.

Если новый поток порождается в методе main, то он наследует от своего создателя статус потока-пользователя. После завершения main приложение будет выполняться до того, как завершится и этот порожденный поток. В исходном потоке нет ничего особенного — просто он оказывается первым при конкретном запуске приложения. После этого такой поток ничем не отличается от всех остальных потоков. Приложение работает до тех пор, пока не будут завершены все его потоки-пользователи. С точки зрения runtime-системы, исходный поток создается лишь для того, чтобы породить другой поток и умереть, предоставляя порожденному потоку выполнять всю работу. Если вы хотите, чтобы ваше приложение завершалось вместе с завершением исходного потока, необходимо помечать все создаваемые потоки как потоки-демоны.

## Использование Runnable

В интерфейсе Runnable абстрагируется концепция некой сущности, выполняющей программу во время своей активности. Интерфейс Runnable объявляет всего один метод:

**public void run();**

Класс Thread реализует интерфейс Runnable, поскольку поток как раз и является такой сущностью — во время его активности выполняется программа. Прежде всего, расширение классов производится на основе одиночного наследования — если некоторый класс расширяется для того, чтобы он мог выполняться в потоке, то одновременно расширить и его, и Thread не удастся. Кроме того, если нужна только возможность выполнения, то вряд ли нужно наследовать и все накладные расходы, связанные с Thread.

Во многих случаях проще реализовать Runnable. Объект Runnable может выполняться в отдельном потоке — для этого следует передать его конструктору Thread. Если объект Thread конструируется с объектом Runnable, то реализация Thread.run вызывает метод run переданного объекта.

## Ключевое слово volatile

Механизм синхронизации помогает в решении многих проблем, однако, если вы откажетесь от его использования, сразу несколько потоков смогут одновременно изменять значение некоторого поля. Если это делается намеренно (может быть, для синхронизации доступа используются другие средства), следует объявить поле с ключевым словом volatile. Например, если у вас имеется переменная, значение которой постоянно отображается потоком графического вывода и может изменяться несинхронизированными методами, то фрагмент вывода может выглядеть следующим образом:



Если бы значение currentValue не могло изменяться внутри метода ShowValue, то компилятор мог бы предположить, что величина current Value остается в цикле постоянной, и просто использовать константу 5 вместо вызова showValue.

Однако, если во время выполнения цикла значение currentValue может быть изменено другим потоком, то предположение компилятора будет неверным. Объявление поля currentValue с ключевым словом volatile не позволяет компилятору делать подобные предположения.

## Отладка потоков

Несколько методов класса Thread предназначены для облегчения отладки многопоточных приложений. Эти средства используются для вывода информации о состоянии программы. Ниже приведен список методов класса Thread, помогающих в процессе отладки:

**public String toString()**

Возвращает строковое описание потока, включающее его имя, приоритет и имя группы.

**public String countStackFrames()**

Возвращает количество кадров стека в потоке.

**public static void dumpStack()**

Выводит в System.out содержание стека для текущего потока.

Также существует ряд отладочных средств для отслеживания состояния группы потоков. Следующие методы вызываются для объектов ThreadGroup и выдают информацию об их состоянии:

**public String toString()**

Возвращает строковое описание группы, включающее ее имя и приоритет.

**public synchronized void list()**

Выводит в System.out список содержимого группы и ее подгрупп.

# Архитектура модель вид контроллер(MVC model-view-controller)



Архи­тектура MVC использует принципы объектно-ориентированного проектирования для разбиения приложений на компоненты данных, компоненты представления и компоненты обработки ввода. Компоненты данных хранят исходные данные при­ложения, такие как текст документа в текстовом процессоре или расположения фигур на шахматной доске. Компоненты представления чаще всего дают визуаль­ное представление данных приложения, например, трехмерное изображение фи­гур на шахматной доске. Компоненты обработки ввода обслуживают ввод поль­зователя, например, движения мышью для перемещения фигур на шахматной доске.

MVC широко используется в приложениях, использующих метафору рабочего стола, имитаторах и других видах программ.

Архитектура модель-вид-контроллер (MVC) отделяет прикладные данные (со­держащиеся в модели) от компонентов графического представления и логики об­работки ввода (контроллера). Архитектура MVC изначально была разработана в Smalltalk-80 в качестве способа для отделения пользовательского интерфейса от основных данных приложения. На рис. представлены взаимоотношения меж­ду компонентами MVC:



Контроллер реализует управляющую часть приложения для обработки пользо­вательского ввода. Модель содержит прикладные данные, а вид генерирует визу­альное отображение данных, хранящихся в модели. Когда пользователь вводит данные (например, набирая текст в текстовом процессоре), контроллер модифици­рует модель в соответствии с этим вводом. Важно заметить, что модель содержит только исходные (необработанные) прикладные данные. В простом текстовом ре­дакторе модель может содержать только символы, которые образуют документ. Когда модель изменяется, она уведомляет вид об изменении, чтобы он обновился в соответствии с измененными данными. Вид в текстовом процессоре может ото­бражать символы на экране с использованием заданных размеров, шрифтового оформления и т.д.

MVC не ограничивает приложение одним видом и одним контроллером. В тек­стовом процессоре, например, может быть два вида одной модели документа. Один вид может отображать структуру документа, а другой — документ в режиме пред­варительного просмотра для печати. Текстовый процессор также может иметь раз­личные контроллеры, например, для обработки ввода с клавиатуры и для обработ­ки манипуляций мышью. Если один из контроллеров вносит изменения в модель, то и вид в режиме структуры, и окно предварительного просмотра должны немед­ленно отобразить внесенные изменения. Разработчик может предоставить допол­нительные виды и контроллеры для модели без внесения изменений в имеющиеся компоненты.

Компоненты Swing Java реализуют вариант MVC, в котором вид и контроллер объединены в один объект, называемый делегатом



Делегат обеспечива­ет как графическое представление модели, так и интерфейс для модификации мо­дели. Например, каждый компонент JButton имеет соответствующий компонент ButtonModel, для которого объект JBntton является делегатом. Компонент ButtonModel содержит информацию о состоянии, например, была ли кнопка JBntton нажата и доступна ли кнопка, а также список слушателей действий ActionListe­ner. Объект JButton обеспечивает графическое представление (например, прямо­угольник на экране с надписью и рамкой) и модифицирует состояние компонента JButtonModel (например, когда пользователь нажимает кнопку JButton).

## Класс Observable и интерфейс Observer

Паттерн проектирования Observer (Наблюдатель) позволяет установить слабую связь (loose coupling) между объектом и зависимыми от него объектами. Слабо свя­занные объекты взаимодействуют путем вызова методов, объявленных в извест­ных интерфейсах, а не вызова методов, объявленных в определенных классах. Ис­пользование методов интерфейса делает каждый из объектов независимым от дру­гих объектов класса. Например, механизм обработки событий использует слабое связывание для уведомления объектов о событиях. Если объект должен обрабаты­вать определенные события, он реализует соответствующий интерфейс слушателя (например, ActionListener). Объекты, которые генерируют события, вызывают ме­тоды интерфейса слушателя для уведомления объектов-слушателей о событиях. Такое слабое связывание дает, например, возможность компоненту JButton посы­лать уведомление о событии типа ActionEvent подклассу JFrame, который реали­зует интерфейс ActionListener. Компонент JButton взаимодействует с подклассом JFrame только через метод ActionPerfomed интерфейса ActionListener, а не через какой-либо другой метод, который является специфичным для подкласса JFrame. Объект JButton может посылать уведомления о событиях ActionEvent другим объектам, которые также реализуют интерфейс ActionListener (например, класс, определяемый программистом, или внутренний класс).

Класс java.util.Observable представляет модель в архитектуре MVC, или субъект в паттерне проектирования Observer (Наблюдатель). Класс Observable предос­тавляет метод addObserver, который принимает параметр java.util.Observable, Интерфейс Observer соответствует виду в архитектуре MVC и обеспечивает слабое связывание между объектом Observable и его наблюдателями Observer. Если объ­ект Observable изменяется, он уведомляет каждого зарегистрированного наблюда­теля Observer об изменении. Наблюдатель может представлять собой экземпляр любого класса, который реализует интерфейс Observer; поскольку объект Obser­vable вызывает методы, определенные в интерфейсе Observer, объекты остаются слабо связанными.

## Выводы о модели MVC

* Архитектура модель-вид-контроллер (MVC) отделяет данные приложения (содержащие­ся в модели) от компонентов графического интерфейса (вида) и логики обработки ввода (контроллера).
* Базовые классы Java Foundation Classes (их чаще называют ком по вентам и Swing) реали­зуют вариант архитектуры MVC, в котором вид и контроллер объединены в один объект, называемый делегатом. Делегат обеспечивает и графическое представление модели, и интерфейс для модификации модели.
* Каждый объект JButton имеет соответствующую модель ButtonModel, для которой объ­ект JBntton является делегатом. Модель ButtonModel хранит информацию о состоянии, например, была ли нажата кнопка JButton или активна ли кнопка, а также список слу­шателей ActionListener. Объект JButton обеспечивает графическое представление (на­пример, прямоугольник на экране с надписью и рамкой) и изменение состояния модели ButtonModel (например, когда пользователь щелкает на кнопке JButton).
* Паттерн проектирования Observer (Наблюдатель) — наиболее общий способ примене­ния архитектуры МVС, который обеспечивает слабое связывание данного объекта и зави­сящих от него объектов.
* Класс java.utiLObservable представляет модель в МУС или субъект в паттерне проекти­рования Observer (Наблюдатель}. Класс Observable предоставляет метод addObserver, ко­торый принимает параметр типа java.util.Observer.
* Интерфейс Observer представляет вид в архитектуре МУС или наблюдатель в паттерне проектирования Observer (Наблюдатель). Когда объект Observable изменяется, он уве­домляет об изменении каждого зарегистрированного наблюдателя.
* Архитектура модель-вид-кон роллер требует, чтобы модель уведомляла свои виды ори изменениях модели. Метод setChanged класса Observable устанавливает флаг changed модели. Метод notifyObservers класса Observable уведомляет всех наблюдателей Obser­ver (т.е. виды) об изменении.
* Объект Observable должен вызывать метод setChanged до вызова метода notifyObservers. Метод notifyObservers вызывает метод update интерфейса Observer для каждого зареги­стрированного наблюдателя.

# Базы данных Java Database Connectivity(JDBC)

База данных — это коллекция данных. Существует миого различных страте¬гий для организации данных таким образом, чтобы облегчить доступ и манипули¬рование данными. Система управления базой данных (DataBase Management System — DBMS) предоставляет механизмы для хранения и организации данных в соответствии с форматами базы данных. Системы управления базами данных дают возможность осуществлять доступ и хранить данные, не заботясь об их внутреннем представлении.

Наиболее популярными на сегодняшний день базами данных являются реляци¬онные базы данных. Язык Structured Query Language (SQL) практически повсеме¬стно используется в системах реляционных баз данных для выполнения запросов (т.е. для получения информации, которая удовлетворяет заданному условию) и для манипулирования данными.

К популярным системам управления реляционными базами данных относятся Microsoft SQL Server, Oracle, Sybase, DB2, Informix и MySQL.

Приложения Java взаимодействуют с базами данных и манипулируют их дан¬ными с помощью интерфейса прикладного программирования Java Database Connectivity (JDBC). Драйвер JDBC реализует интерфейс с конкретной базой дан¬ных. Такое отделение интерфейса прикладного программирования от конкретных драйверов дает возможность разработчикам переходить к другой системе управле¬ния базами данных, не модифицируя код Java, который осуществляет доступ к базе данных. Большинство современных систем управления базами данных со¬держат драйверы JDBC. Имеется также множество драйверов JDBC от сторонних поставщиков.

java.sun.com/products/jdbc/ Этот сайт содержит информацию, связанную с JDBC, включая спецификацию JDBC, FAQ (часто задаваемые вопросы) по JDBC, загружаемое программное обес¬печение и другую важную информацию.

## Структура базы в Серверной части приложения.

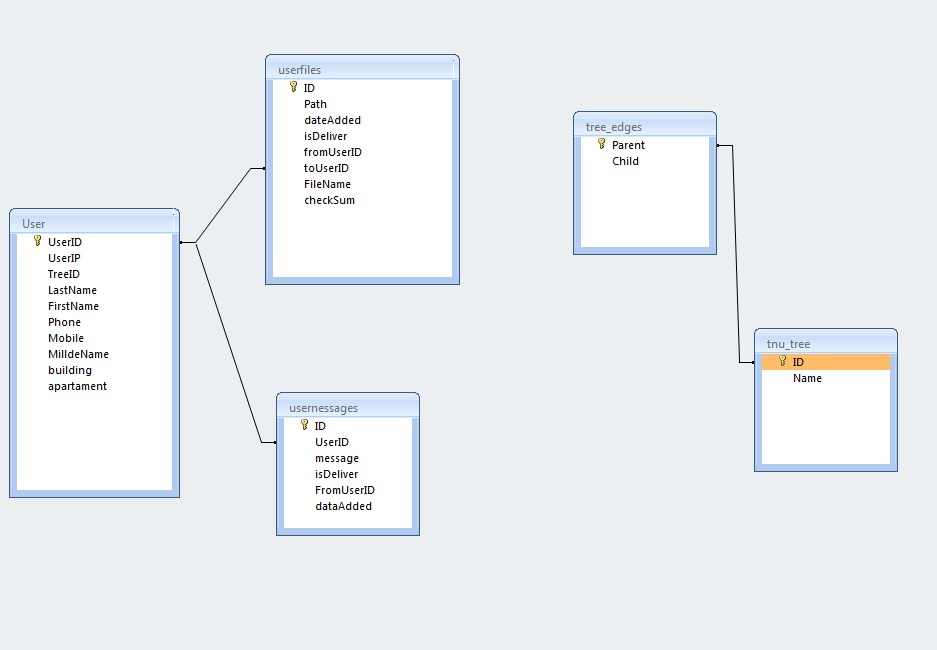


Таблица User – имеет два уникальных ключа это UserID и UserIP. Второй из них сделан что бы предотвратить запись пользователей с одинаковым Ip. Также в этой таблице хранится информация о пользователе – имя, фамилия, телфон и так далее.

Таблица Userfiles – хранит в себе информацию о переданных или ожидающих передачу файлов. Если файл передан то поле IsDeliver становится равным true, если файл ожидает передачи то false. В этой таблице присутствуют два поля, которые относятся к таблице User – это toUserId и fromUserId. Специфика базы не позволяет создавать два внешних ключа ссылающихся на одно и тоже место в другой таблице. Поэтому создан только 1 внешний ключ соединяющие toUserID и UserID. Поле Path – содержит физический путь к файлу на сервере. Поле FileName содержит оригинальное название файла. Это поле нужно для того, что бы можно было восстановить исходное имя файла, поскольку оригинальное имя файла на диске сервера может иметь другое имя, что бы обеспечить гарантию того что не будет совпадений имён файлов. Поле dateAdded говорит само за себя – время отсылки файла. Поле checksum – контрольная сумма файла переданная с клиента. Потом сервер её передаёт клиенту который закачивает файл. И конечный клиент может определить корректно ли файл был передан или нет.

Таблица tnu\_tree содержит два поля. ID – идентификатор звена в дереве пользователей и его название – Name.

Таблица tnu\_edges содержит рёбра дерева. В поле Parent записан родительское звено вершины которая содержится в Children. Соответственно по этим данным легко восстановить путь от вершины к конкретному пользователю.

Таблица usermessages устроена аналогично таблице userfiles. Она содержит поля fromUserId и userId. Первое это идентификатор пользователя от которого пришло сообщение. А второе поле это идентификатор пользователя которому это сообщение адресовано. Поле message – это посланное сообщение.

## Хранимые процедуры

Поскольку для разработки нашего серверного приложения были использованы хранимые процедуры, я рассмотрю их основные положения.

Хранимые процедуры представляют собой набор команд SQL, которые могут компилироваться и храниться на сервере. Таким образом, вместо того, чтобы хранить часто используемый запрос, клиенты могут ссылаться на соответствующую хранимую процедуру. Это обеспечивает лучшую производительность, поскольку данный запрос должен анализироваться только однажды и уменьшается трафик между сервером и клиентом. Концептуальный уровень можно также повысить за счет создания на сервере библиотеки функций.

Хранимые программы (процедуры и функции) поддерживаются в MySQL 5.0. Хранимые процедуры – набор SQL-выражений, который может быть сохранен на сервере. Как только это сделано, клиенту уже не нужно повторно передавать запрос, а требуется просто вызвать хранимую программу.

Хранимые процедуры и функции (подпрограммы) могут обеспечить лучшую производительность потому, что меньше информации требуется для пересылки между клиентом и сервером. Выбор увеличивает нагрузку на сервер БД, но снижает затраты на стороне клиента.

Хранимые подпрограммы также позволяют вам использовать библиотеки функций, хранимые в БД сервера. Эта возможность представлена для многих современных языков программирования, которые позволяют вызывать их непосредственно (например, используя классы).

Хранимые процедуры требуют наличия таблицы proc в базе mysql. Эта таблица обычно создается во время установки сервера БД

При создании, модификации, удалении хранимых подпрограмм сервер манипулирует с таблицей mysql.proc

Начиная с MySQL 5.0.3 требуются следующие привилегии:

* **CREATE ROUTINE** для создания хранимых процедур
* **ALTER ROUTINE** необходимы для изменения или удаления процедур. Эта привилегия автоматически назначается создателю  процедуры (функции)
* **EXECUTE** привилегия потребуется для выполнения подпрограммы. Тем не менее, автоматически назначается создателю процедуры (функции). Также, по умолчанию, SQL SECURITY параметр для подпрограммы DEFINER , который разрешает пользователям, имеющим доступ к БД вызывать подпрограммы, ассоциированные с этой БД.

Начиная с MySQL 5.0.1, загруженная процедура или функция связана с конкретной базой данных. Это имеет несколько смыслов:

Когда подпрограмма вызывается, то подразумевается, что надо произвести вызов USE db\_name (и отменить использование базы, когда подпрограмма завершилась, и база больше не потребуется)

Можно квалифицировать обычные имена с именем базы данных. Это может быть использовано, чтобы ссылаться на подпрограмму, которая - не в текущей базе данных. Например, для выполнения хранимой процедуры p или функции f которые связаны с БД test, вы можете сказать интерпретатору команд так: CALL test.p() или test.f().

Когда база данных удалена, все загруженные подпрограммы связанные с ней тоже удаляются. В MySQL 5.0.0, загруженные подпрограммы - глобальные и не связанны с базой данных. Они наследуют по умолчанию базу данных из вызывающего оператора. Если USE db\_name выполнено в пределах подпрограммы, оригинальная текущая БД будет восстановлена после выхода из подпрограммы (Например текущая БД db\_11, делаем вызов подпрограммы, использующей db\_22, после выхода из подпрограммы остается текущей db\_11 )

MySQL поддерживает полностью расширения, которые разрешают использовать обычные SELECT выражения (без использования курсоров или локальных переменных) внутри хранимых процедур. Результирующий набор, возвращенный от запроса, а просто отправляется напрямую клиенту. Множественный SELECT запрос генерирует множество результирующих наборов, поэтому клиент должен использовать библиотеку, поддерживающую множественные результирующие наборы.

**CREATE PROCEDURE** – создать хранимую процедуру.

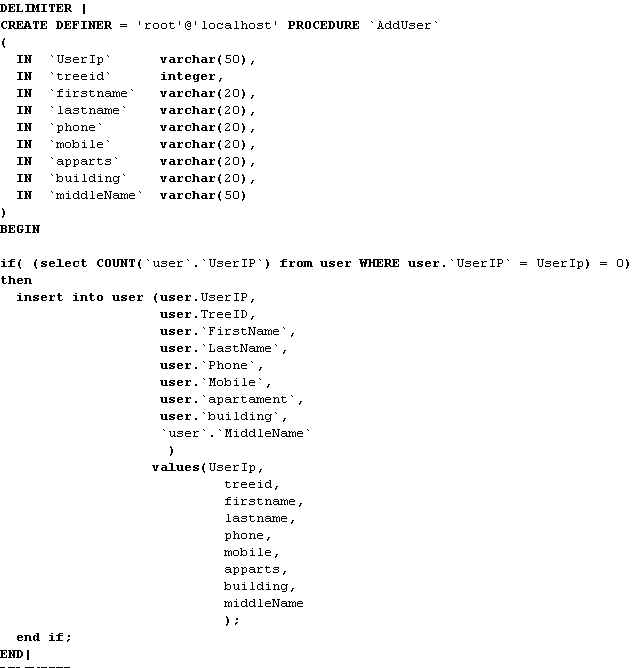
**CREATE FUNCTION** – создать хранимую функцию.

**Синтаксис:**

**CREATE PROCEDURE** имя\_процедуры ([параметр\_процедуры[,...]])  
[характеристёика ...] тело\_подпрограммы  
  
**CREATE FUNCTION** имя\_функции ([параметр\_функции[,...]])  
**RETURNS** тип  
[характеристика ...] тело\_подпрограммы  
  
параметр\_процедуры:  
 [ **IN** | **OUT** | **INOUT** ] имя\_параметра тип  
параметр\_функции:  
 имя\_параметра тип  
  
тип:  
 Любой тип данных MySQL  
  
характеристика:  
 LANGUAGE SQL  
 | [NOT] DETERMINISTIC  
 | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }  
 | SQL SECURITY { DEFINER | INVOKER }  
 | COMMENT 'string'  
  
тело\_подпрограммы:  
 Правильное SQL выражение.

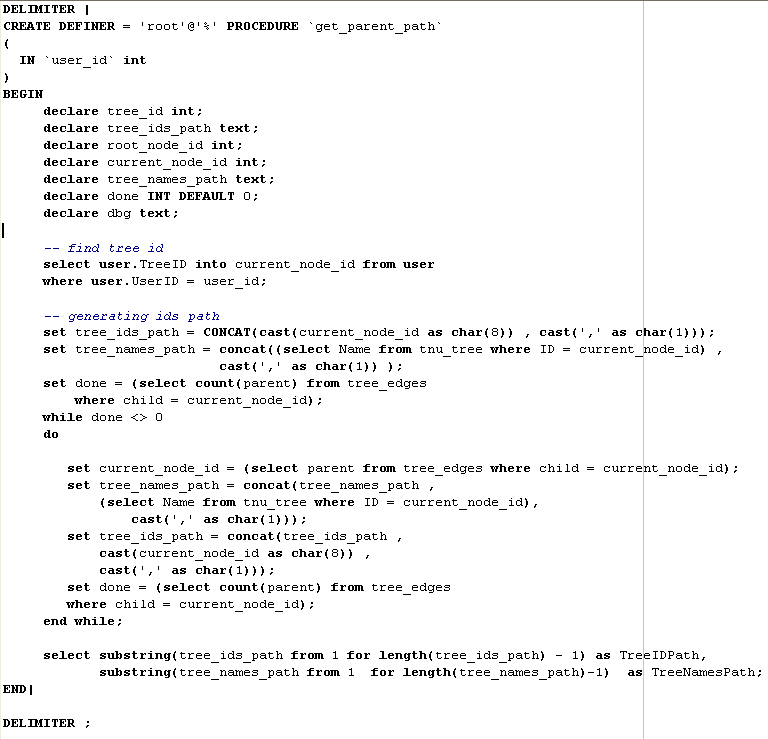
## Использование хранимых процедур в qMessenger

В качес тве примеров приводятся некоторые хранимые процедуры с подробным описанием.



Процедура AddUser – одна из самых простых процедур написанных на SQL.

В качестве входных параметров используется информация о пользователе, для внесения в базу. В начале мы проверяем существует ли в базе пользователь с новым Ip. Если существует то тогда мы не добавляем иначе, создаётся новая запись в базе с информацией о пользователе. Даже в этой простой процедуре уже видно преимущество – вместо двух запросов к базе, тут это выполняется за один проход.



Вот еще один пример. Процедура get\_parent\_path. Это немного сложнее процедура для поиска пути на пользователя на дереве. Работает она следующим образом. В качестве входящего параметра используется идентификатор пользователя user\_id. Далее по этому идентификатору достаётся идентфикатор узла дерева и записывается в current\_node\_id.

Далее я использую две переменных tree\_ids\_path и tree\_names\_path. В первой из них, в конце процедуры, будет содержаться список идентификаторов узлов дерева пути, во второй – список названий соответствующих идентификаторов. Для их инициализации используется – текущий идентификатор дерева.

Данные о дереве у нас хранятся в таблице tree\_edges. Поле tree\_edges.Parent это родитель, tree\_edges.Child – потомок. Поэтому имея такую таблицу, мы можем в цикле, просматривая все рёбра, находить ребро потомок которого является текущий узел дерева. Потом родитель становится текущим узлом, и так далее, пока не достигнем корневой вершины. Это всё достаточно просто реализуется одним лишь оператором Select который выбирает нужные нам рёбра из базы.

Ну и в конце мы просто возвращаем результат.

В этой процедуре производительность уже существенная, за счет использования хранимых процедур. Поскольку операций выборки из базы тут очень много, она будет работать существенно быстрее, хотя бы экономя время на постоянных подключениях к базе и последующую обработку данных.

# Заключение

И так, в этой работе было разработано кросс платформенное приложение, которое по своей функциональности совмещает в себе почтовую программу и программу быстрого обмена сообщениями.

# Приложение

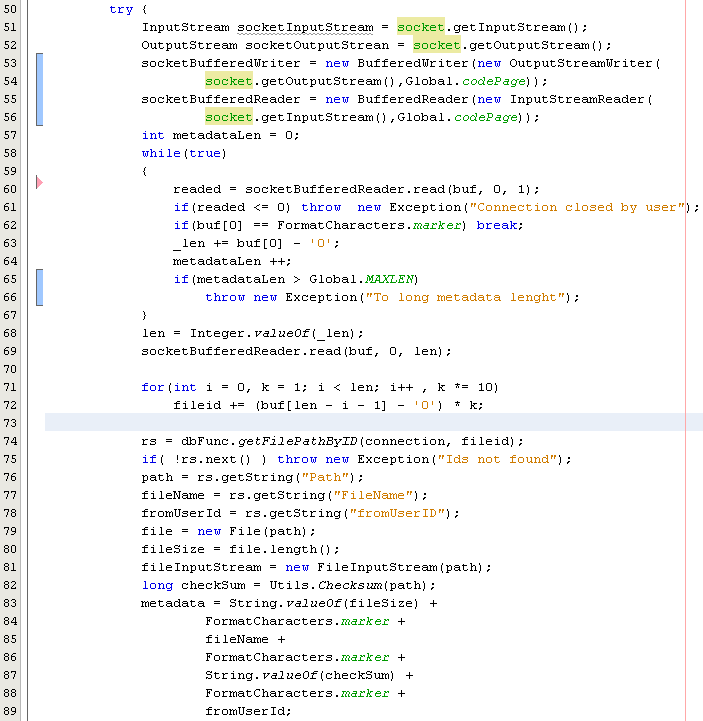
В этой главе я рассмотрю наиболее тонкие и весьма важные моменты в реализации как клиентской так и серверной части.

## Проблема с сокетами.

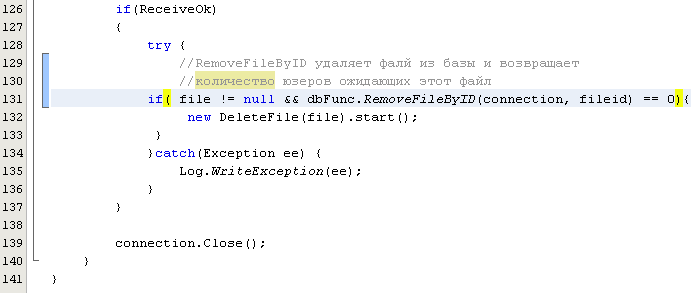
На этапе разработки я столкнулся с проблемой передачи имён файлов с русскими именами. Решение было найдено – использовать буфферезированные потоки BufferedReader/BufferedWriter. Эти классы работают в отличии от InputStramReader/InputStreamWriter с 4х байтовыми массивами. Это позволяет передавать все символы Юникода. Таким образом дописав в класс Messages использование таких классов я решил прблему с передачей русских имен файлов. Но как выяснилось позже, эти потоки конфликтуют с обычными InputStreamReader/Writer. InputStreamReaeder в приложении служат для передачи файлов. Тоесть когда я сначала передаю данные которые содержат информацию о файле – имя файла, размер и тд, а потом сам файл, выяснилось что такая схема не работает. Решение было найдено следующее. После того как мы отправили данные буферезированным потоком, я присваиваю объекту это класса null, и вызываю через определенное время сборщик мусора. Казалось бы сборщик мусора должен сам вызывается в нужное время, но в моей программе он должен видимо это делать чаще. После вызова сборщика мусора объект BufferedReader/Writer уничтожается, и больше не конфликтует с другими классами использующими этот сокет. На листинге ниже представлен код серверной части. В строке 95-96 мы присваиваем null объектам BufferedReader /Writer что бы гарантировать что сборщик мусора удалит эти объекты. В строке 97 вызывается сборщик мусора. Далее в 98 строке мы принудительно останавливаем поток на 500 миллисекунд. Эта цифра подобранна экспериментально. Нужна для того что бы сборщику было больше времени на удаление объектов.

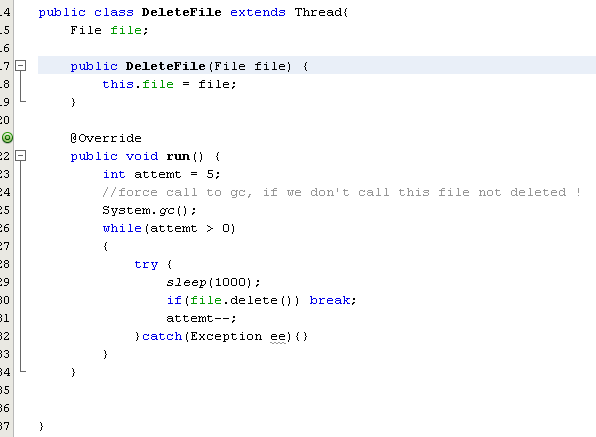
Есть еще одна небольшая проблема с которой я столкнулся при разработке. После того как файл был отправлен, его необходимо удалить с сервера. Но поскольку бывают очень часто случаи когда в памяти остаётся не удалённый объект который ссылается на этот файл, и система не может его удалить. Для этого, примерно также как было с сокетами, я принудительно вызывал сборщик мусора. Но после этого, я создаю дополнительный поток, который несколько раз с определенным интервалом пытается удалить этот файл. И после каждой неудавшейся попытки вызывает сборщик мусора. Это гарантирует что файл удалится. Эта функциональность реализована в классе DeleteFile. Как видно в строчке 132 вызывается эта функциональность.

Там же приведен листинг класса удаляющего файл. Реализован он достаточно просто в цикле поток пытается удалить файл. Если это не получается то ждем 1 секунду и пытаемся еще раз. Поскольку это выполняется в потоке, сервер не тратит на это время и переключается на работу с другими клиентами.





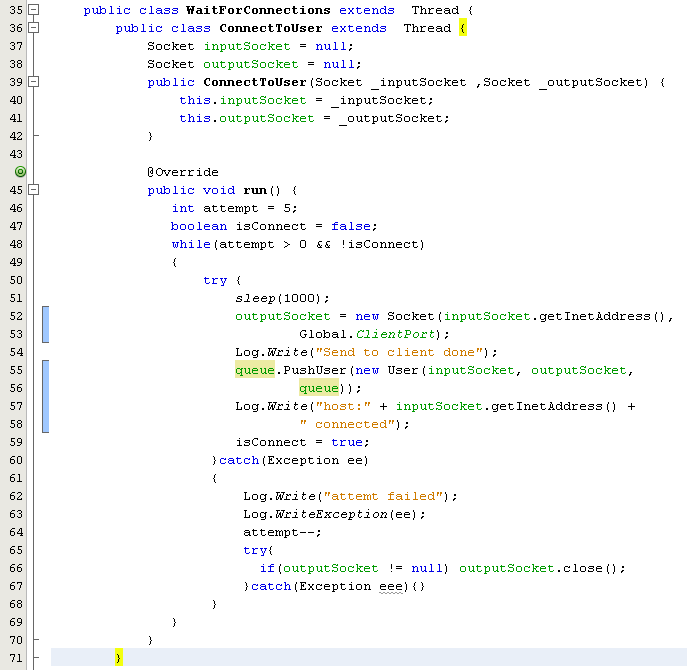




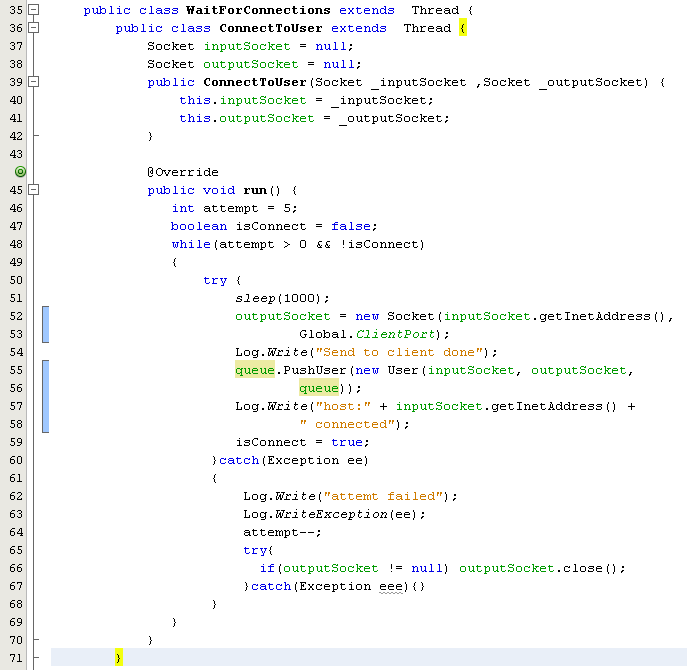
Класс DeleteFile для удаления файла.

## Подключение пользователей.

Подключение пользователей реализовано следующим образом. Пользователь создает серверный сокет, и клиентский сокерт. Серверный сокет выносится в отдельный поток и слушается. Сразу же после этого, пользователь использует клиентский сокет подключается к серверу. В этот момент как сервер зафиксировал подключение к себе, делает обратное подключение к пользователю на его серверный сокет. Код серверного подключение представлен на листинге ниже.



Обратите внимаение. Сервер выполняет подключение к пользователю уже в отдельном потоке. Причем не один раз. Поскольку время между тем как пользователь установит серверный сокет у себя, может быть большим чем когда сервер будет делать обратный вызов. Именно для этого сервер пытается подключится несколько раз. Это реализовано в цикле – строки 51-59. Если по какой то причине не получается подключится, сервер пытается еще раз. Таких попыток всего 5. Но это число легко изменить. Мое тестирование показало, что подключение в 99% случаях происходит с первого раза. В очень редких случаях со второго.



## Обработка Xml.

При обработке xml файла, в Java некоторые вершины помечаются как '#text'. Поэтому такие вершины нужно как то дополнительно обрабатывать.

При анализе Xml используется Dom обработчик. Это обосновывается тем, что нам всегда нужно знать из какой вершины мы пришли. А это не возможно реализовать используя Sax обработчик. Используя Dom я легко, поиском в глубину анализирую Xml , добавляю ребра в таблицу tree\_edges. А также Dom хорошо подходит, когда нам нужно сгенерировать дерево и зарегистрированных пользователей. Код это класса представлен на листинге ниже.



Поиском в ширину, мы добавляем вершины Xml в очередь. Очередь представлена объектом LinkedList<int>Q. В строчке 41 создается этот объект. Затем добавляется главная вершина в очередь. А в цикле мы достём очередную вершину из очереди(строка 49), и смотрим в базе нет ли таких пользователей, которые соответствуют данной вершине. Это определяется в строчке 66. Функция базы данных возвращает нам список пользователей, представленных объектом UserInfo. Если этот список не пуст, то мы добавляем всех пользователей в Xml и в дальнейшем отправляем клиенту.

# Список использованной литературы

1. <http://java.sun.com/products/jfc/tsc/articles/architecture/>
2. <http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html>
3. <http://java.sun.com/docs/books/tutorial/networking/sockets/>
4. <http://dev.mysql.com/usingmysql/java/>
5. http://sql-ex.ru/