Создание модели предприятия с использованием средств языка программирования Java

(https://github.com/Dew25/Library.git)

Язык программирования Java, как, впрочем, и другие языки программирования, предоставляет необходимый инструментарий для создания модели любого в действительности существующего предприятия. Модель, это всегда упрощенная схема взаимодействий неких сущностей, которые и представляют предприятие. Сама программная модель создается для автоматизации, удобного хранения данных и всевозможных подсчетов, которые необходимы предприятию для его функционирования.
Итак, задача, которая стоит перед нами, выражается следующим техническим заданием.

Создать программную модель библиотеки, которая может удобно добавлять новые книги в свой архив, выдавать книги читателю для чтения.

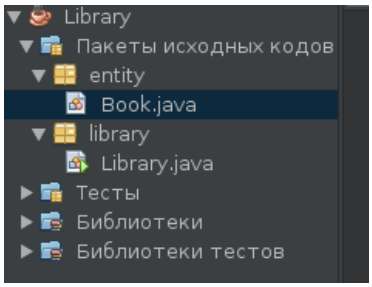
Вначале нам нужно осознать какие сущности взаимодействуют в предприятии под названием «библиотека». Подумав и поразмышляв на эту тему, мы можем прийти к следующей схеме.
В библиотеке взаимодействуют три сущности. Это «Книга», «Читатель» и еще одна сущность, которая связывает читателя со взятой им книгой, при этом эта сущность еще будет хранить время выдачи и время возврата книги читателем. Назовем эту сущность «История»
Взаимодействие этих сущностей и будет отажать модель предприятия под названием «Библиотека».

Хранить наш код мы будем на github.com, поэтому нам надо создать там репозиторий с именем нашего проекта «**Library**».
Затем создать проект Java с главным классом.
Внутри папки с проектом создайте файл с именем **.gitignore** (это надо сделать с помощью тектового редактора типа Notepad++, SublimeText или подобному им). Добавьте в этот файл следующие строки:
build.xml
/build
/dist
/nbproject

После этого надо открыть в папке проекта программу gitbash и набрать следующие команды:
git init
git add .
git commit -m "инициация репозитория"
git remote add origin https://github.com/<имя вашего репозитория>/Library.git (эту строку можно скопировать в подсказках, которые дает github.com после создания нового репозитория)
git push origin master

Приступим к созданию сущностей.

В качестве сущности хорошо подходит такая структура как класс. С помощью IDE NetBeans в пакете entity создадим классы описывающие выше перечисленные сущности.



В классах создадим поля для хранения состояния сущности, конструкторы и геттеры — сеттеры. Не забудем переопределить equals и hashcode, а также напишем удобный метод toString().

```java
package entity;

import java.util.Objects;

/**
 *
 * @author jvm
 */
public class Book {
    private Long id;
    private String bookName;
    private String isbn;
    private String author;
    private Integer yearPublishing;

    public Book() {
    }

    public Book(Long id, String bookName, Stri
        this.id = id;
        this.bookName = bookName;
        this.isbn = isbn;
        this.author = author;
        this.yearPublishing = yearPublishing;
    }
```

```java
package entity;

import java.util.Objects;

/**
 *
 * @author jvm
 */
public class Reader {
    private Long id;
    private String name;
    private String surname;
    private String phone;
    private String city;

    public Reader() {
    }

    public Reader(Long id,
        this.id = id;
        this.name = name;
        this.surname = surn
        this.phone = phone;
        this.city = city;
    }
```

```java
package entity;

import java.util.Date;
import java.util.Objects;

/**
 *
 * @author jvm
 */
public class LibHistory {
    private Long id;
    private Book book;
    private Reader reader;
    private Date bookIssued;
    private Date bookReturn;

    public LibHistory() {
    }

    public LibHistory(Long id,
        this.id = id;
        this.book = book;
        this.reader = reader;
        this.bookIssued = bookI
        this.bookReturn = bookR
    }
```

После создания сущностей, мы можем инициировать экземпляры и представить их взаимодействие.
Сделаем это в методе main() нашего приложения.

```
package library;

import entity.Book;
import entity.LibHistory;
import entity.Reader;
import java.util.Calendar;
import java.util.GregorianCalendar;

/**
 *
 * @author jvm
 */
public class Library {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        Book book1 = new Book(1L, "Voina i Mir", "123-123123", "Lev Tolstoy", 2010);
       Reader reader1 = new Reader(1L, "Ivan", "Ivanov", "56565656", "Johvi");

       Calendar c = new GregorianCalendar();

       LibHistory libHistory1 = new LibHistory(1L, book1, reader1, c.getTime(), null);

        System.out.println(book1.toString());
        System.out.println(reader1.toString());
        System.out.println(libHistory1.toString());

    }

}
```

Запустим приложение и получим вывод данных в консоль.

```
Вывод - Library (run)
run:
Book{id=1, bookName=Voina i Mir, isbn=123-123123, author=Lev Tolstoy, yearPublishing=2010}
Reader{id=1, name=Ivan, surname=Ivanov, phone=56565656, city=Johvi}
LibHistory{id=1, book=Voina i Mir, reader=Ivan Ivanov, bookIssued=Thu Sep 06 23:47:40 EEST 2018, bookReturn= }
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 1 секунда)
```

Мы можем прочитать состояние сущностей и вывести его в текстовом виде.
Можно понять, что читатель Ivan Ivanov взял книгу Voina i Mir шестого сентября 2018 года в 23 часа, 47 минут и на момент вывода данных её еще не вернул (поле bookReturn пусто).

Добавим наши файлы и изменения файлов с кодом в git.
Для этого в консоле наберите следующие комманды:
git add .
git commit -m"Шаг 1. Выдача книги пользователю"
git push origin master

Таким образом, мы сделали слепок наших файлов и скопировали (протолкнули) наш код в репозиторий на сайте github.com.

Теперь можно использовать код на другом компьютере, для этого можно клонировать репозиторий. Чтобы это сделать выполните команду:
git clone https://github.com/<ваш репозиторий>/Library.git

Если вы уже раньше клонировамли этот репозиторий на этот компьютер, то вам придется набрать следующую команду:
git pull origin master
Она «вытянет» изменения с удаленного репозитория и сделает ваш код на этом компьютере

актуальным. Теперь вы можете работать с этим кодом. Не забудьте только в конце работы сделать команды:
git add .
git commit -m"Шаг 1. Выдача книги пользователю"
git push origin master

Продолжаем работать с кодом программы.
Предположим что свершился еще один акт библиотечного дела — возврат книги.
В результате изменится состояние сущности, которая хранит эту информацию.

```
    public static void main(String[] args) {

        Book book1 = new Book(1L, "Voina i Mir", "123-123123", "Lev Tolstoy", 2010);
        Reader reader1 = new Reader(1L, "Ivan", "Ivanov", "56565656", "Johvi");

        Calendar c = new GregorianCalendar();

        LibHistory libHistory1 = new LibHistory(1L, book1, reader1, c.getTime(), null);

        System.out.println(book1.toString());
        System.out.println(reader1.toString());
        System.out.println(libHistory1.toString());

        c.add(Calendar.DATE,2);
        libHistory1.setBookReturn(c.getTime());
        System.out.println(libHistory1.toString());

    }

}
```

```
Вывод
Library (run) × | Library - /home/jvm/Library ×
run:
Book{id=1, bookName=Voina i Mir, isbn=123-123123, author=Lev Tolstoy, yearPublishing=2010}
Reader{id=1, name=Ivan, surname=Ivanov, phone=56565656, city=Johvi}
LibHistory{id=1, book=Voina i Mir, reader=Ivan Ivanov,
bookIssued=Fri Sep 07 00:24:22 EEST 2018, bookReturn= }
LibHistory{id=1, book=Voina i Mir, reader=Ivan Ivanov,
bookIssued=Fri Sep 07 00:24:22 EEST 2018, bookReturn=Sun Sep 09 00:24:22 EEST 2018}
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Мы добавили к дате два дня и изменили состояние объекта libHistory1, инициировав пустое поле bookReturn с помощью сет-метода значением - дата возврата книги.
Обратите внимание, поле bookReturn теперь имеет данные, сообщающие дату возврата книги.

(выполните команды
git add .
git commit -m"Шаг 2. Возврат книги читателю")

Таким образом, мы можем моделировать работу библиотеки на компьютере. Хотя эта модель еще очень примитивна, ее можно усложнять, все более приближая к оригиналу.

Приступим.

Пока наша модель может получать данные через консольный ввод и только от программиста, а требуется, чтобы данные в программу мог вводить любой пользователь.
Чтобы решить эту задачу нам надо сделать дружелюбный интерфейс для пользователя. Создадим класс, который будет общаться с пользователем и направлять его действия. Назовем его App, и добавим ему метод run(), в котором будет «бегать» наша библиотека в цикле do{}while(<boolean>). Для выбора задач будем использовать конструкцию switch.

Листинг класса App

```
public class App {
    private List<Book> books = new ArrayList<>();
    private List<Reader> readers = new ArrayList<>();
    public void run(){
        String repeat = "r";
        Scanner scanner = new Scanner(System.in);
        int task;
        do{
            System.out.println("Выберите действие:");
            System.out.println("0 - выход из программы");
            System.out.println("1 - добавить книгу в библиотеку");
            System.out.println("2 - добавить нового читателя");
            System.out.println("3 - выдать книгу читателю");
            System.out.println("4 - возврат книги");
            System.out.println("5 - история работы библиотеки");

            task = scanner.nextInt();
            switch (task) {
                case 0:
                    repeat="q";
                    break;
                case 1:
                    BookCreator bookCreator = new BookCreator();
                    books.add(bookCreator.returnNewBook());
                    break;
                case 2:
                    ReaderCreator readerCreator = new ReaderCreator();
                    readers.add(readerCreator.returnNewReader());
                    break;
                default:
                    System.out.println("Выберите одно из действий!");
            }


        }while("r".equals(repeat));

    }
}
```

Для реализации задач 1 и 2 нам пришлось создать два вспомогательных класса BookCreator и ReaderCreator. У этих классов есть методы, которые возвращают экземпляр инициированной книги или читателя. Чтобы хранить экземпляры книг и читателей, были созданы переменные books и readers, являющиеся массивом типа List

Листинг класса BookCreator

```
15  public class BookCreator {
16      public Book returnNewBook(){
17          Scanner scanner = new Scanner(System.in);
18          System.out.println("-----Добавление новой книги-----");
19          Book book = new Book();
20          System.out.println("Название книги:");
21          book.setBookName(scanner.nextLine());
22          System.out.println("ISBN книги:");
23          book.setIsbn(scanner.nextLine());
24          System.out.println("Автор книги:");
25          book.setAuthor(scanner.nextLine());
26          System.out.println("Год издания книги:");
27          book.setYearPublishing(scanner.nextInt());
28          return book;
29      }
30  }
```

Класс ReaderCreator устроен так же, но только он создает нового читателя. Я не стал приводить здесь листинг этого класса в надежде, что вы напишете его самостоятельно.

Теперь, чтобы наше приложение можно было использовать, надо реализовать задачи 3, 4 и 5. Для этого добавим массив, для хранения экземпляров сущности, LibHistory. Затем создадим новый класс LibHistoryCreator, в котором реализуем функционал задачи.

Реализация третьей задачи.
Подумаем, что нам надо для того чтобы реализовать возможность выдать конкретную книгу, из списка, конкретному читателю, тоже хранимого в списке. Представляется, что для этого надо написать класс с методом, в который придется передать список книг и список читателей. Чтобы иметь возможность выбора, придется напечатать список книг и список читателей. После этого надо получить экземпляр выбранной книги из списка и экземпляр выбранного читателя. И наконец создать экземпляр класса LibHistory и инициализировать этот объект, добавив время инициализации.
Придумали, теперь реализуем.

```
public class LibHistoryCreator {
    public LibHistory returnNewLibHistory(List<Book> books, List<Reader> readers){
        System.out.println("-----Выдача книги читателю------");
        Scanner scanner = new Scanner(System.in);
        System.out.println("Список книг: ");
        int countBooks = books.size();
        for(int i = 0; i<countBooks; i++){
            System.out.println(i+1+". "+books.get(i).getBookName());
        }
        System.out.println("Список читателей: ");
        int countReaders = readers.size();
        for(int i = 0; i<countReaders; i++){
            System.out.println(i+1+". "+readers.get(i).getName()+" "+readers.get(i).getSurname());
        }
        System.out.println("Выберите номер книги:");
        int numberBook = scanner.nextInt();
        Book book = books.get(numberBook-1);
        System.out.println("Выберите номер читателя:");
        int numberReader = scanner.nextInt();
        Reader reader = readers.get(numberReader-1);
        Calendar c = new GregorianCalendar();
        LibHistory libHistory = new LibHistory(null, book, reader, c.getTime(), null);
        return libHistory;
    }
}
```

Теперь надо добавить решение задачи в класс App.
Сначала создадим список для историй:

```
private List<LibHistory> libHistories = new ArrayList<>();
```

Затем добавим решение задачи в оператор switch:

```
case 3:
    LibHistoryCreator libHistoryCreator = new LibHistoryCreator();
    libHistories.add(libHistoryCreator.returnNewLibHistory(books, readers));
    break;
```

Реализация четвертой задачи.

Четвертая задача будет решена, если мы к существующему экземпляру класса LibHistory добавим значение в поле bookReturn. Нам придется написать метод в новом классе, который будет получать параметр - список libHistories.
Реализовать придется вывод всего списка читателей, которые взяли книги и возможность выбора строки с нужным читателем и возвращаемой им книгой. Предусмотрим также случай, когда мы захотим прервать текущую операцию.
Делаем.

```java
public class BookReturner {
    public boolean returnLibHistory(List<LibHistory> libHistories){
        try{
            System.out.println("-----Возврат книги-----");
            LibHistory libHistory = new LibHistory();
            Scanner scanner = new Scanner(System.in);
            int countLibHistories = libHistories.size();
            for (int i = 0; i < countLibHistories; i++) {
                LibHistory history = libHistories.get(i);
                System.out.println(i+1+". "+history.getReader().getName()
                        +" "+history.getReader().getSurname()
                        +": "+history.getBook().getBookName());
            }
                System.out.println("Выберите номер строки с возвращаемой книгой: ");
                System.out.println("Чтобы ничего не делать наберите -1");
                int numHistory = scanner.nextInt();
                if(numHistory < 0) return false;
                libHistory = libHistories.get(numHistory-1);
                libHistories.remove(libHistory);
                Calendar c = new GregorianCalendar();
                libHistory.setBookReturn(c.getTime());
            return true;
        }catch(Exception e){
            return false;
        }
    }
}
```

Сохраним наши изменения в репозитории git.
Выполним следующие действия в консоле:
git add .
git commit -m "Шаг 4. Решение третьей и четвертой задачи."

Осталась последняя задача, которая заложена в нашу программу на этом, начальном, этапе.

Реализация пятой задачи.
Нам требуется вернуть историю работы библиотеки. Подумав над этим, мы можем прийти к выводу, что не вся история будет нам полезна. Интерес будут представлять только те записи, где есть пользователи не вернувшие еще книги в библиотеку. Вот их то мы и выведем, чтобы иметь возможность видеть, у кого находятся выданные на чтение книги.
Осознать что надо делать — уже пол дела.
Реализуем задумку с помощью класса, который, подумав, назовем HistoryReturner. В этом классе создадим метод printListWhoTookBooks, который ничего не будет возвращать, а будет только выводить нам информацию о читателях имеющих на руках книги. Понятно, что ему через параметры придется предать список всех историй библиотеки libHistories.

Листинг класса HistoryReturner

```
public class HistoryReturner {
    public void printListWhoTookBooks(List<LibHistory> libHistories){
        System.out.println("-----Список читателей взявших книги----");
        SimpleDateFormat sdfDate = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        int countHistories = libHistories.size();
        for(int i=0; i<countHistories; i++){
            LibHistory h = libHistories.get(i);
            if(h.getBookReturn()==null){
                System.out.println(i + 1 + ". "+h.getBook().getBookName() +
                        ". Читает: " + h.getReader().getName()+
                        " " + h.getReader().getSurname()+
                        ". Взято: " + sdfDate.format(h.getBookIssued()));
            }
        }
    }
}
```

Осталось только внести некоторые дополнения в класс App.

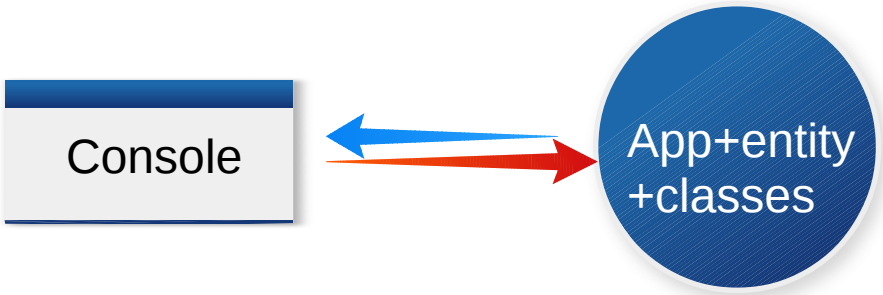
```
case 5:
    HistoryReturner historyReturner = new HistoryReturner();
    historyReturner.printListWhoTookBooks(libHistories);
    break;
```

Ну вот и закончили мы очередной этап создания приложения. Теперь надо сохранить наши изменения в репозитории git.
Выполним следующие действия в консоле:
git add .
git commit -m "Шаг 5. Решение пятой задачи."

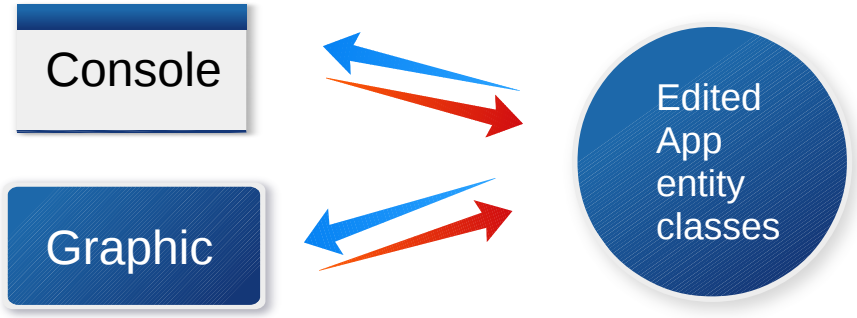
Чтобы сохранить наш код на github.com выполним следующую команду:
git push origin master

Теперь наше приложение может работать и выполнять возложенные на него задачи вводя данные через консоль и выводя их через консоль.
Но вот вопрос, когда мы научимся писать графические приложения придется полностью переписать наше приложение, чтобы сделать его графическим или можно поступить по другому.

Сейчас наше приложение выглядит вот так:

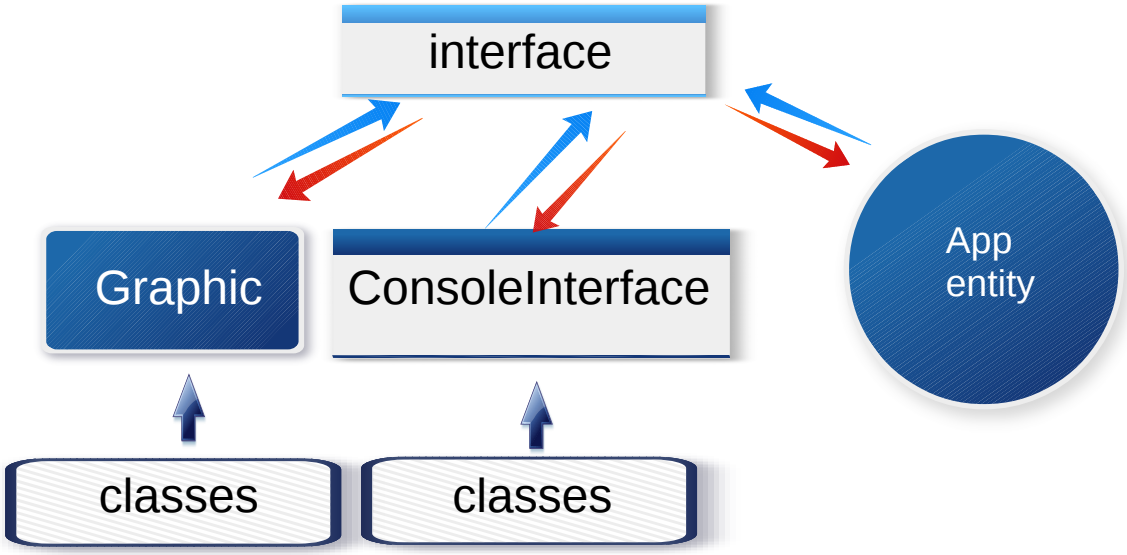


Есть Console, которая связана с App с сущностями и классами из пакета classes.
Если мы напишем ввод данных с графического интерфейса, то придется сделать изменения и в правой части приложения, чтобы они могли взаимодействовать и с Console и с Graphic.



Но компьютерная наука нашла способ делать по другому и предлагает на этапе написания приложения заложить возможность добавления нового функционала без изменения основной части, т. е. без переписывания старого кода.
Компьютерная наука говорит, что связи одной части программы с другой ее частью должны проектироваться через абстракцию.
Подумает, какую абстракцию можно применить в нашем случае.

Что если мы свяжем код нашей **основной части** программы с частями, которые выполняют **ввод и вывод информации разными способами** через **интерфейс**. Тогда мы можем в интерфейсе указать необходимые методы, которые будет слушаться основная программа, и вспомогательные программы смогут знать, что через эти методы можно управлять основной программой. Interface — это некая абстракция, через которую одна часть программы связывается с другими своими частями. Этот прием позволяет создавать гибкие, расширяемые программы без ненужной дополнительной правки связанных частей.



Что же, хорошая идея. Попробуем реализовать этот способ связи объектов в наше программе. Создадим интерфейс с названием Manageable (Управляемый, потому как есть рекомендация называть интерфейсы именами прилагательными)
Объявим в нем несколько методов.

```
17  public interface Manageable {
18      public Book createBook();
19      public Reader createReader();
20      public LibHistory issueBook(List<Book> books, List<Reader> readers);
21      public boolean returnBook(List<LibHistory> libHistories);
22      public void returnHistory(List<LibHistory> libHistories);
23  }
```

Теперь создадим класс ConsoleInterface и заставим его реализовать интерфейс Manageable. Это приведет к тому что у этого класса обязательно будут реализованы методы, объявленные в интерфейсе. Причем реализация этих методов нами уже написана и находится в классе App.

```
case 1:
    BookCreator bookCreator = new BookCreator();
    books.add(bookCreator.returnNewBook());
    break;
case 2:
    ReaderCreator readerCreator = new ReaderCreator();
    readers.add(readerCreator.returnNewReader());
    break;
case 3:
    LibHistoryCreator libHistoryCreator = new LibHistoryCreator();
    libHistories.add(libHistoryCreator.returnNewLibHistory(books, readers));
    break;
case 4:
    BookReturner bookReturner  = new BookReturner();
    if(bookReturner.returnLibHistory(libHistories)){
        System.out.println("Книга возвращена");
    }else{
        System.out.println("Книгу вернуть не удалось");
    }
    break;
case 5:
    HistoryReturner historyReturner = new HistoryReturner();
    historyReturner.printListWhoTookBooks(libHistories);
    break;
```

Осталось только перенести из класса App соответствующие решения в класс ConsoleInterface. Будьте внимательны, изменения разумные все же есть.

```
23  public class ConsoleInterface implements Manageable{
24
25      @Override
26      public Book createBook() {
27          BookCreator bookCreator = new BookCreator();
28          return bookCreator.returnNewBook();
29      }
30
31      @Override
32      public Reader createReader() {
33          ReaderCreator readerCreator = new ReaderCreator();
34          return readerCreator.returnNewReader();
35      }
36
37      @Override
38      public LibHistory issueBook(List<Book> books, List<Reader> readers) {
39          LibHistoryCreator libHistoryCreator = new LibHistoryCreator();
40          return libHistoryCreator.returnNewLibHistory(books, readers);
41      }
42
43      @Override
44      public boolean returnBook(List<LibHistory> libHistories) {
45          BookReturner bookReturner  = new BookReturner();
46          return bookReturner.returnLibHistory(libHistories);
47      }
48
49      @Override
50      public void returnHistory(List<LibHistory> libHistories) {
51          HistoryReturner historyReturner = new HistoryReturner();
52          historyReturner.printListWhoTookBooks(libHistories);
53      }
54
55  }
```

Ну вот, теперь пора подправить и класс App.

```
25  public class App {
        private List<Book> books = new ArrayList<>();
        private List<Reader> readers = new ArrayList<>();
        private List<LibHistory> libHistories = new ArrayList<>();
        private Manageable manager = new ConsoleInterface();
30      public void run(){
31          String repeat = "r";
32          Scanner scanner = new Scanner(System.in);
33          int task;
34          do{
35              System.out.println("Выберите действие:");
36              System.out.println("0 - выход из программы");
37              System.out.println("1 - добавить книгу в библиотеку");
```

Синим я выделил новое объявление объекта manager имеющего тип Manageable, а создан он на основе реализации этого интерфейса классом ConsoleInterface.
Следующее, что надо изменить, так это содержимое оператора switch. Используем объект manager, который умеет управлять нашим приложением, в данном случае через консоль.

```
48                  case 1:
49                      books.add(manager.createBook());
50                      break;
51                  case 2:
52                      readers.add(manager.createReader());
53                      break;
54                  case 3:
55                      libHistories.add(manager.issueBook(books, readers));
56                      break;
57                  case 4:
58                      if(manager.returnBook(libHistories)){
59                          System.out.println("Книга возвращена");
60                      }else{
61                          System.out.println("Книгу вернуть не удалось");
62                      }
63                      break;
64                  case 5:
65                      manager.returnHistory(libHistories);
66                      break;
```

Если мы удосужимся создать графический интерфейс нашего приложения вместо консольного, то изменится всего лишь одна строчка в сласе App, а именно эта:

```
private Manageable manager = new ConsoleInterface();
```

Вместо класса ConsoleInterface(), будет использоваться другой класс, например, с названием GraphicInterface. Но, так как у этого класса тоже будут все методы, которые объявлены в **Manageable**, то больше менять нечего.

Ну вот и все, мы решили поставленную задачу и изменили архитектуру нашего приложения

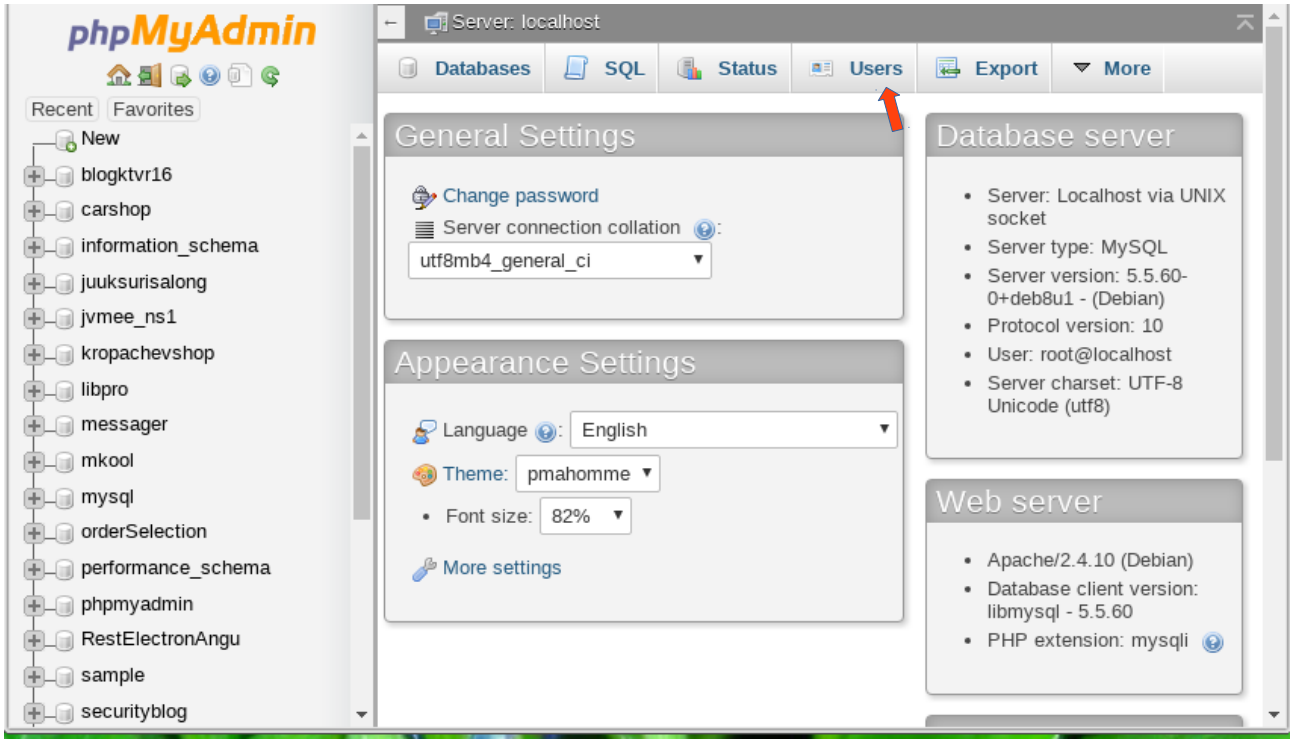
на более совершенную. Связи частей программы между собой ослабли и нам открылся простор для добавления нового функционала.
А пока зафиксируем изменения и зальем в облако наш код.
git add .
git commit -m"Шаг 6. Изменение структуры приложения. Использование интерфейса для добавления гибкости приложению"

Сохранение состояния модели в базе данных.

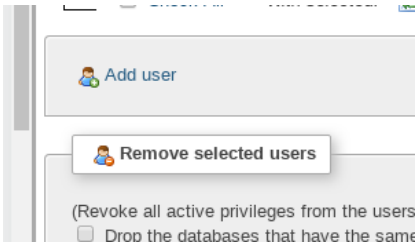
Для хранения состояний сущностей обычно используется база данных. В процессе работы созданной программы, в энергозависимой памяти компьютера хранятся экземпляры сущностей Book, Reader и LibHistory. Чтобы они не исчезли после выключения, эти состояния экземпляров могут быть сохранены на накопителе, который не зависит от электрического тока, т. е. На жестком диске или твердотельном накопителе.
Есть два способа хранения информации: фалы или база данных.
Далее мы попробуем хранить информацию в базе данных. Для этого необходимо произвести соответствующую подготовку и добавить необходимые инструменты (библиотеки, фреймворки) в наш проект.
Для того чтобы пользоваться базой данных необходимо иметь доступ к ней. Таким образом нам нужен токен (логин и пароль) доступа к базе данных. На наших компьютерах уже установлена база данных MySql, которая поставляется вместе с пакетом XAMPP. Доступ к этой базе мы можем получить, если проделаем следующие шаги:
1. Запустим панель управления XAMPP.
2. Запустим вебсервер Apache2.
3. Запустим базу данных MySql.
4. Запустим браузер и в адресной строке наберем: http://localhost/phpmyadmin/

phpmyadmin — это программа, которая написана на языке PHP и представляет собой удобную утилиту управления базой данных.

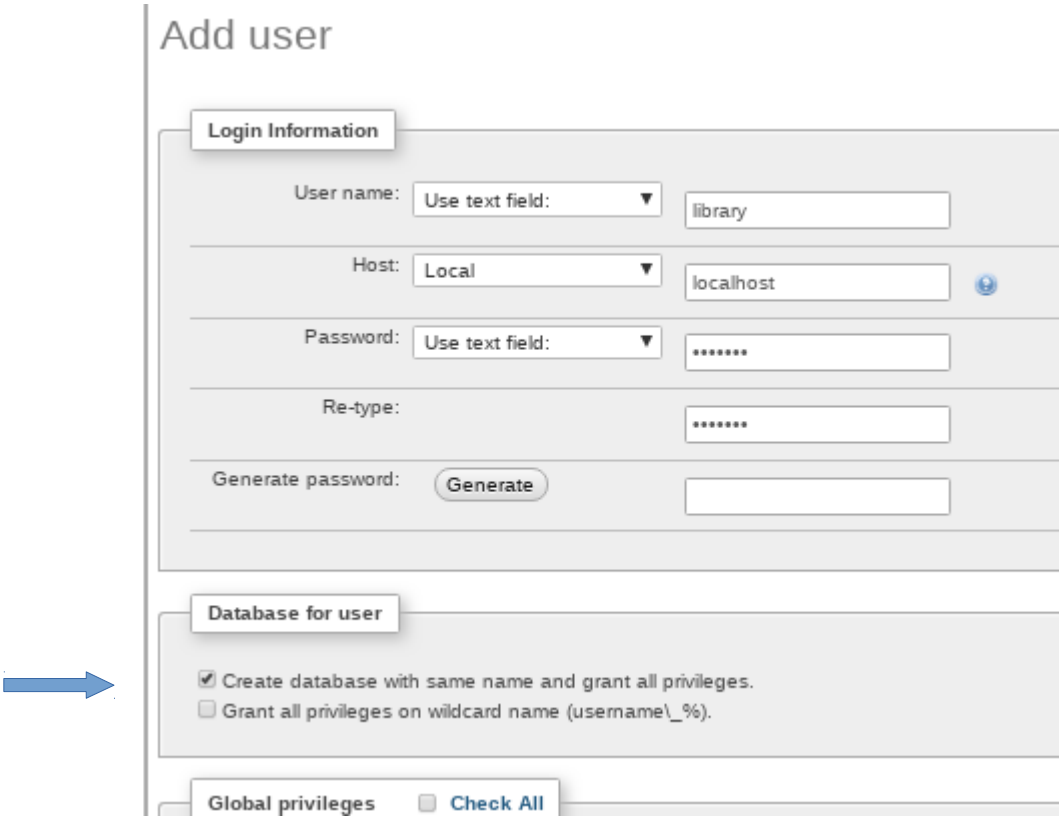
На наших учебных компьютерах Username: root, а Password: пустой.
После входа в phpmyadmin вы увидите следующее.



Теперь нам надо создать базу данных, которую свяжем с нашими сущностями. Назовем базу также как и приложение (обычно так и делают), т. е. **library.** Обратите внимание, название базы данных с маленькой буквы! Для этой базы необходимо создать пользователя со всеми привилегиями только для этой базы. Имя пользователя, который будет управлять базой будет выглядеть так: **library**. Да именно так как и название базы. Это удобно при поддержке, **имя приложения, базу данных и пользователя базы данных называют одинаково**. Для учебных проектов мы будем пароль тоже создавать такой же, для удобства. В реальных условиях пароль должен быть сложным и держаться в строжайшей тайне.
Итак начнем создавать пользователя, базу и пароль.
1. Перейдите на вкладку *Users* (на рисунке указано красной стрелкой)
2. Выберите ссылку **Add user**



3. Заполните поля как указанно на рисунке



Обратите внимание на выбранный чекбокс.
4. Сделайте скрол в самый низ страницы и нажмите кнопку **Go**
Будет создана база данных с именем library, с пользователем library и паролем library.
Отлично, это нам и надо. Осталось только настроить базу данных, чтобы она отображала русский или эстонский языки также хорошо, как и английский.
Для этого необходимо выбрать базу данных с именем library и нажать на вкладку Operations

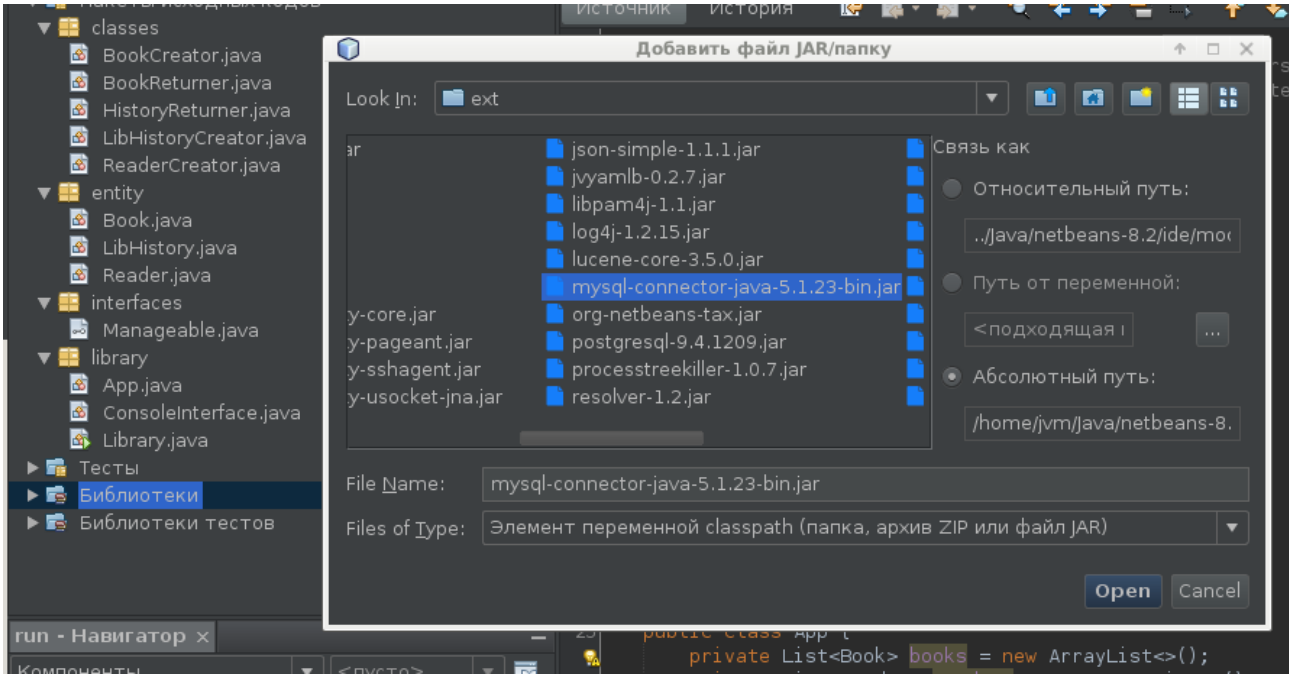
Найдите редактор как на рисунке и выберите **utf8_general_ci** и нажмите **Go.**

Вот и все. База создана и настроена для использования.

Следующий шаг — настройка проекта в IDE NetBeans.

Настройка проекта в IDE NetBeans 8.2 для использования базы данных MySql

Прежде всего надо добавить **драйвер управления базой данных** из Java. Эти драйверы создаются разработчиками баз данных по спецификациям, которые предоставляет им Java. Драйвер можно найти в Интернет, самой последней версии или использовать поставляемый средой NetBeans. Он лежит по следующему пути:
C:\Programm Files\NetBeans\ide\modules\ext\mysql-connector-java-5.1.23-bin.jar
Его мы и добавим в наш проект.
Во вкладке Projects среды NetBeans выберите проект Library и сделайте клик правой кнопкой мышки на строчке Libraries → add file/jar/folder.
На появившемся диалоговом окне вы поле **Look in** найдите драйвер (используйте путь выше описанный) и нажмите **Open** диалогового окна.



В библиотеке проекта появится ссылка на выбранный файл с расширением .jar

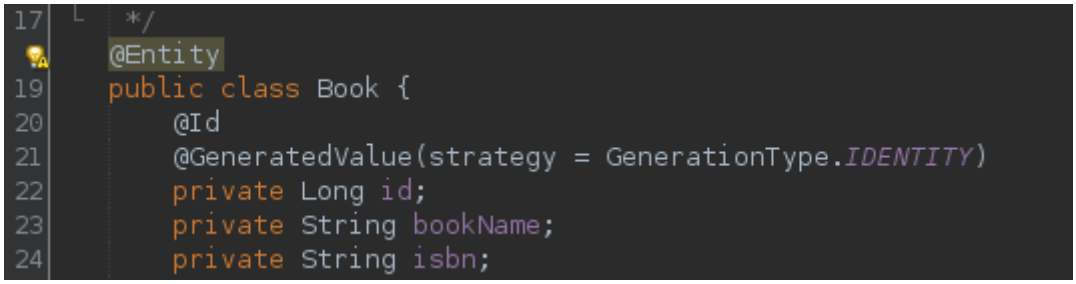
Следующий шаг — добавление удобных в использовании фреймворков.

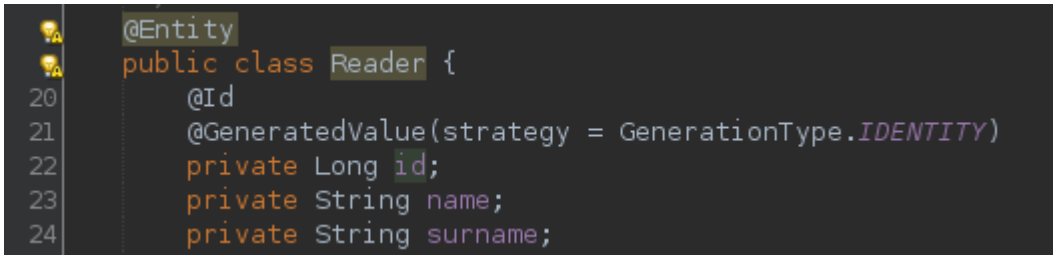
Сделайте клик правой кнопкой мышки на строчке Libraries своего проекта и выберите **add library**
В появившемся диалоговом окне необходимо выбрать две библиотеки:
- EclipseLink (JPA 2.1)
- Persistence (JPA 2.1)
и нажать на кнопку диалогового окна **add library**
Библиотеки можно выбирать по очереди, главное чтобы они были добавлены.

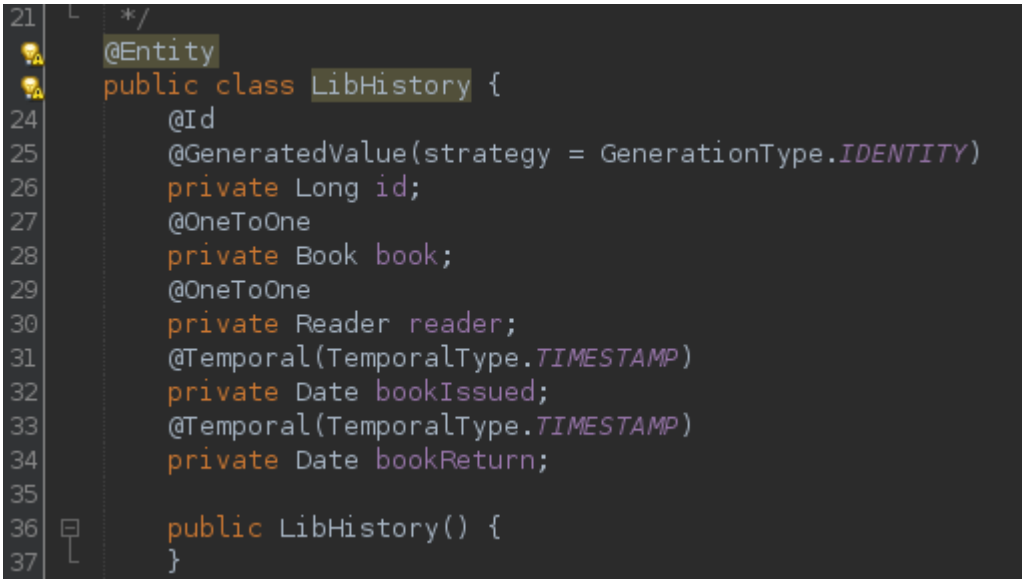
Вот и все. Теперь наш проект готов использовать базу данных и удобные инструменты позволяющие работать с базой данных легко и просто.

Следующий шаг - это связь наших сущностей с базой данных.
Дело в том, что сущности — это классы Java и их состояния необходимо записать в таблицы. Чтобы отобразить класс сущности в таблицу придумали ORM фреймворк и назвали его Persistence. А чтобы было легко управлять сохранением состояний классов в таблицы или наоборот считывать из таблиц записи и инициировать ими классы сущностей, создали фреймворк EclipseLink.
Эти фреймворки работают в паре и используют для связи с базой данных драйвер mysql-connector-java-5.1.23-bin.jar.

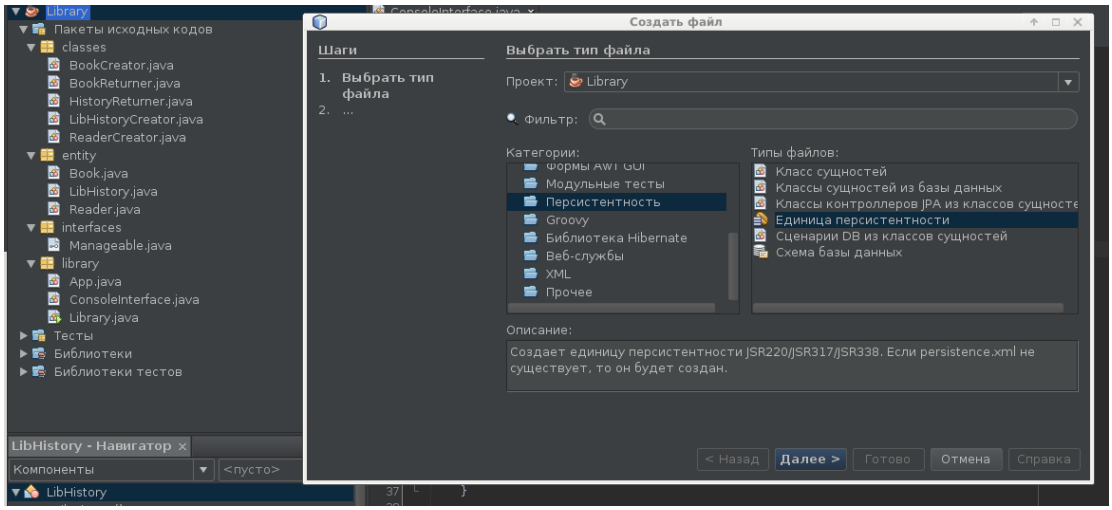
Что же, начнем использовать эти инструменты в нашем проекте.
Первое, что нужно сделать, это написать необходимые аннотации к сущностям нашего проекта. Аннотацией называется строчка специальных слов, которая начинается с символа @ Например, все классы, которые мы называем сущностями, должны перед строкой объявления класса иметь аннотацию @Entity. Тогда фреймворк Persistence будет знать как отображать этот класс в таблицу базы данных.
Кроме того, у сущности должны быть поставлены аннотации и перед некоторыми полями.



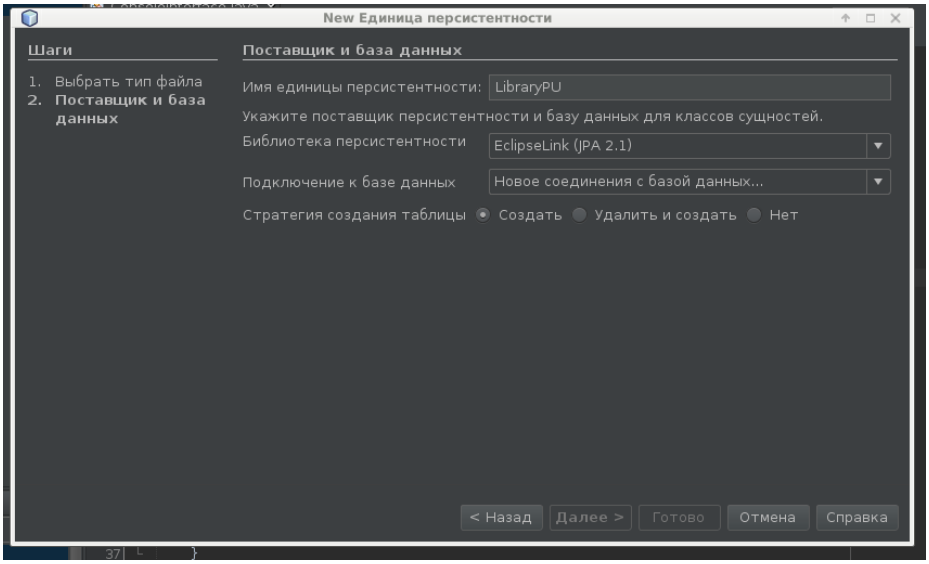




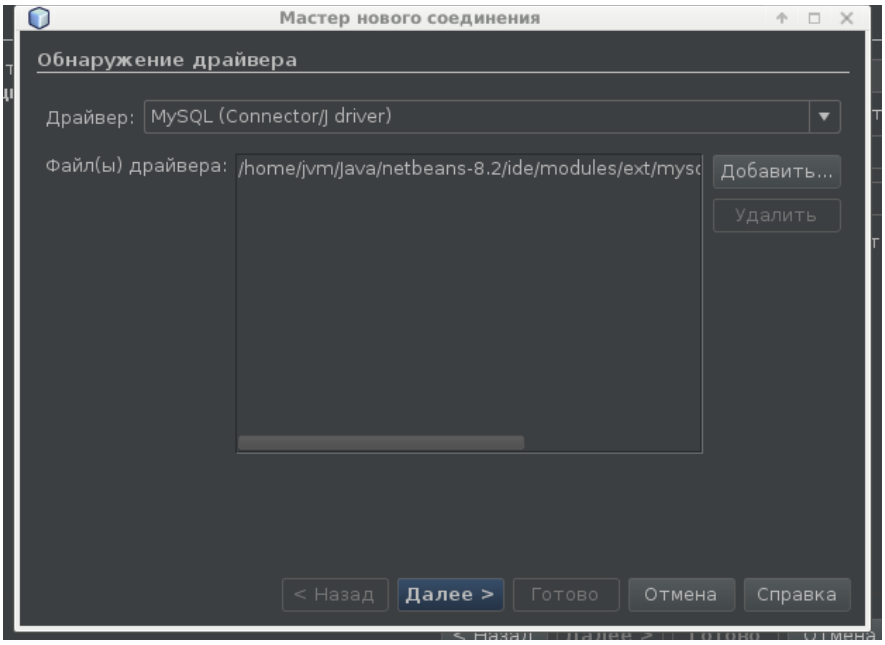
Последний шаг настройки — создание файла persistence.xml.
В этом файле в специальных тегах записываются правила подлючения к базе данных нашего приложения. Именно здесь указывается имя базы данных, пользователя базы данных и пароль. После создания этого файла наши фреймворки будут знать как соединиться с базой данных.
Приступим.
Нажмите File → New File → Persistence → PersistenceUnit



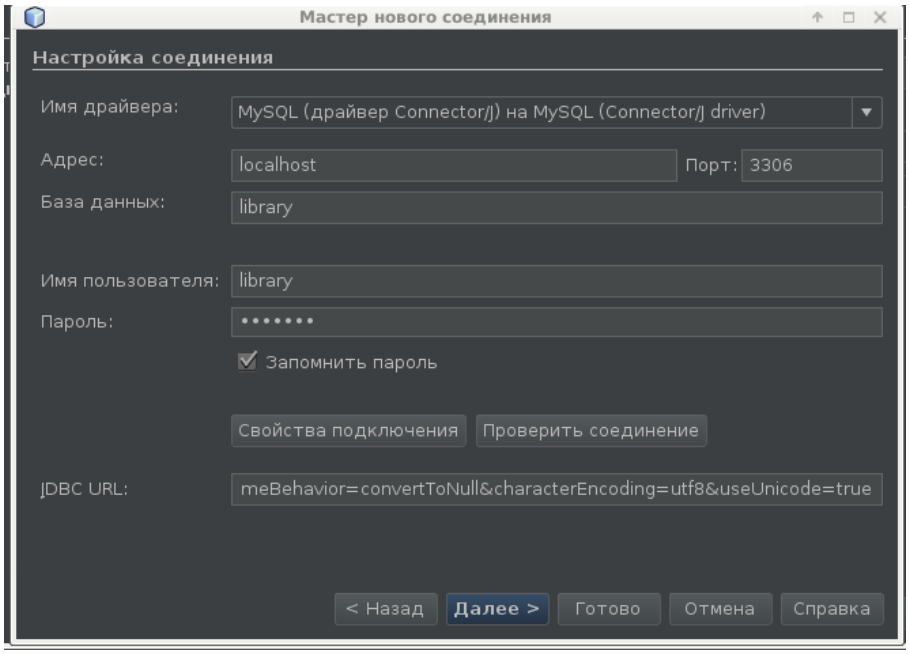
Нажмите далее



Выберите new connection with database, появится всплывающее диалоговое окно



Выберите **MySQL (Connector/J driver)** и нажмите **Next** (Далее)

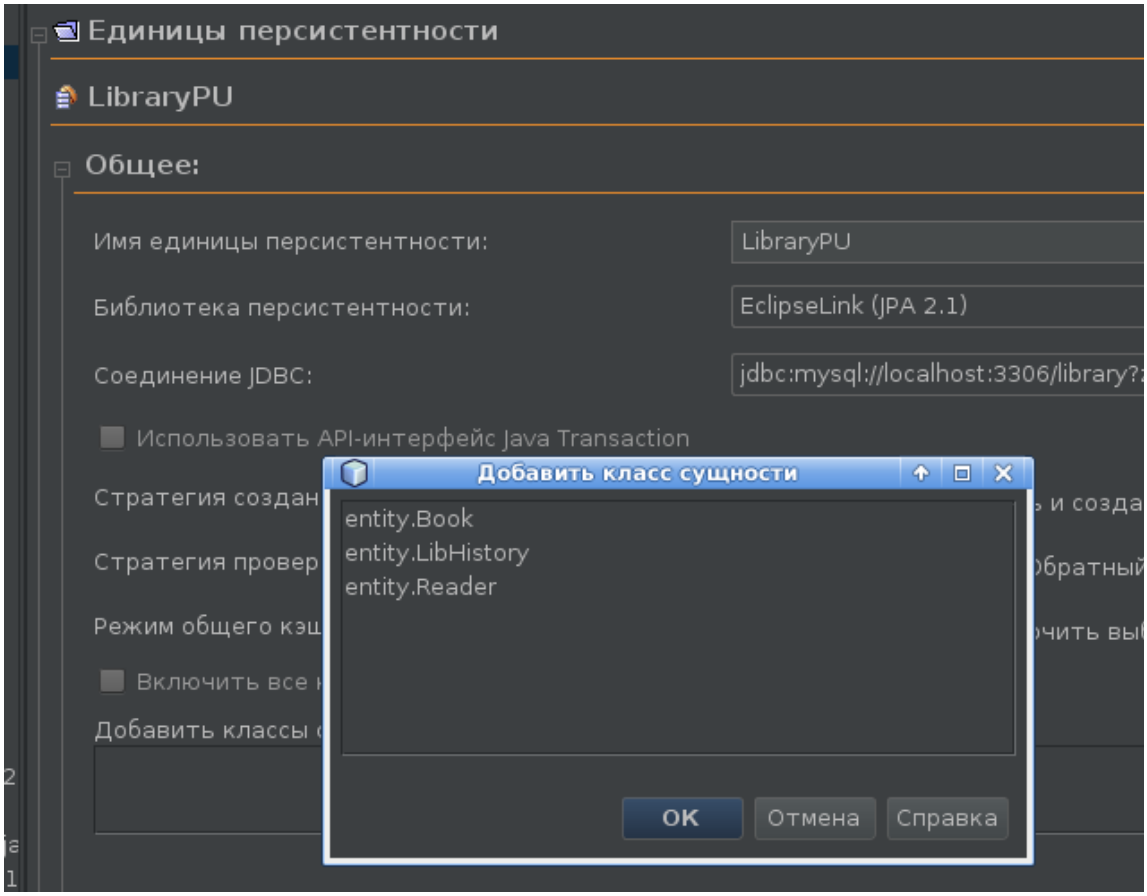


Заполните поля как указано на рисунке.
Обратите внимание на поле
JDBC URL: jdbc:mysql://localhost:3306/library?
zeroDateTimeBehavior=convertToNull&characterEncoding=utf8&useUnicode=true
красным шрифтом отмечено то, что необходимо дописать руками.
Все, теперь нажимайте Next, или далее, или Ок, или Готово, до закрытия всех диалоговых окон.

В результате этих действий будет создан файл с названием persistence.xml в папке META-INF. Этот файл содержит информацию как соединиться с базой данных.
Выглядит он примерно так:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://w
  <persistence-unit name="LibraryPU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <properties>
      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/library?z
      <property name="javax.persistence.jdbc.user" value="library"/>
      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
      <property name="javax.persistence.jdbc.password" value="library"/>
      <property name="javax.persistence.schema-generation.database.action" value="create"/>
    </properties>
  </persistence-unit>
</persistence>
```

Его содержание можно увидеть, если открыть его в редакторе NetBeanse и выбрать вкладку Source или Источник.
Осталось только указать какие классы мы называем сущностями. Для этого необходимо добавить в файл persistence.xml названия наших классов-сущностей.



Для этого на вкладке Конструктор (Constructor) необходимо нажать на кнопку «Добавить классы/Add classes», **выбрать классы-entity** и кликнуть на Ok

Сделаем commit в гит репозитории под именем "Шаг 7. Добавление анотаций к классам-сущностям и создание файла persistense.xml".

Создание инструментов сохранения данных в базу.
Теперь очередь подумать, как мы будем сохранять наши данные в базе. Для решения этой задачи создадим интерфейс (помните рекомендации компьютерной науки) и назовем его Retentive (Сохраняющий).

```
public interface Retentive {
    public void saveBook(Book book);
    public void saveReader(Reader reader);
    public void saveLibHistory(LibHistory libHistory,boolean update);
    public List<Book> loadBooks();
    public List<Reader> loadReaders();
    public List<LibHistory> loadLibHistoryes();
    public void freeResources();
}
```

Таким образом, мы предписываем всем классам, которые будут сохранять данные нашего приложения иметь реализации методов со всеми перечисленными названиями и указанными параметрами. Из названий методов понятно что они делают. Эти классы должны уметь сохранять и считывать в приложение сохраненные раннее данные.
Теперь настала очередь создать класс, который будет реализовывать наш интерфейс.
Создадим класс, который назовем PersistToDatabase и реализуем Retentive.

```
    public class PersistToDatabase implements Retentive{
```

Для управления сущностями нам потребуется инструмент, который реализован фреймворком **EclipseLink**. Создадим поля, которые будут хранить объект класса **EntityManager** и назовем его **em.**

```
public class PersistToDatabase implements Retentive{

    private final EntityManager em;
```

Объект **em** умеет сохранять сущности в базе, считывать их сохраненные состояния и возвращать инициированные экземпляры сущностей. Смотри описание этого инструмента в книге «Изучаем Java EE 7», глава 4. Java Persistence API (стр. 137).

Хорошим тоном считается оборачивать действия с базой в транзациции. Для реализации этого создадим поле типа **EntityTransaction** и назовем его **tx**

```
public class PersistToDatabase implements Retentive{

    private final EntityManager em;
    private final EntityTransaction tx;
```

Теперь надо подумать, как мы будем инициировать эти переменные. Для инициализации полей любого класса обычно используют конструктор. Последуем этому примеру.

```
28    public PersistToDatabase() {
29        EntityManagerFactory emf = Persistence.createEntityManagerFactory("LibraryPU");
30        em = emf.createEntityManager();
31        tx = em.getTransaction();
32    }
```

Опишу как мы это реализовали. Объект EntityManager создается специальным статическим методом класса Persistense, который имеет имя createEntityManagerFactory. У этого метода есть параметр — строка, в которой указывается название persistenceUnit. Открываем файл persistence.xml и находим там тег *<persistence-unit name="LibraryPU" transaction-type="RESOURCE_LOCAL">*. Параметр «name» представляет именно ту строку, которую надо вставить в фабрику. По этому имени фабрика найдет файл persistence.xml и считает путь подключения к базе данных, ее имя, пользователя и пароль.
Итак создаем объект фабрики **EntityManagerFactory emf**. У этого класса есть метод createEntityManager(), который возвращает экземпляр класса EntityManager. С его помощью содаем екземпляр класса и кладем его в поле **em.**
Следующий шаг — создание объекта типа EntityTransaction с помощью метода класса EntityManager, который называется getTransaction(). Кладем полученный объект EntityTransaction в поле **tx**.

Осталось предусмотреть возможность освободить память от объекта em, когда он нам будет не нужен. Это мы можем и не делать, так как у JVM есть сборщик мусора, который уничтожит неиспользуемый объект сам. Но мы можем это сделать и попробуем помочь сборщику. Для этого мы объявляли в интерфейсе публичный метод, вызов которого должен уничтожать объект **em**.

```
34        @Override
35        public void freeResources() {
36            if(em != null) em.close();
37        }
```

Следующий шаг — реализация остальных методов, объявленных в интерфейсе Retentive.

```
38
39        @Override
40        public void saveBook(Book book) {
41            tx.begin();
42            em.persist(book);
43            tx.commit();
44        }
```

Опишу что делает этот метод:
- запускается транзакция записи в базу данных;
- передается экземпляр класса Book в базу данных;
- сохраняются в базе данных переданные объекты и закрывается транзакция.

Теперь вы понимаете сами, что делает этот метод.

```
47      @Override
        public void saveReader(Reader reader) {
49          tx.begin();
50          em.persist(reader);
51          tx.commit();
52      }
```

А вот следующий метод можно прокомментировать.

```
54      @Override
        public void saveLibHistory(LibHistory libHistory, boolean update) {
56          tx.begin();
57          if(update){
58              em.merge(libHistory);
59          }else{
60              em.persist(libHistory);
61          }
62          tx.commit();
63      }
```

В зависимости от значения параметра update, используется метод класса EntitytManager, который называется merge(), или persist().
Метод merge() **обновляет раннее созданную запись** сохраненного состояния объекта новым состоянием объекта переданного методу в параметре.
Метод persist() **создает новую запись** в таблице, в которую отображается класс LibHistory.

Следующий метод — метод чтения из базы, который возвращает инициированный список объектов Book.

```
65      @Override
        public List<Book> loadBooks() {
67          try {
68              return em.createQuery("SELECT b FROM Book b").getResultList();
69          } catch (Exception e) {
                return new ArrayList<Book>();
71          }
72      }
```

Здесь используется специальный язык запросов, чтобы создать запрос в базу данных.
Этот язык называется JPQL (см. подробнее Изучаем Java EE 7, стр. 240), он был создан для того, чтобы можно было абстрагироваться от таблиц базы данных и для составления запросов пользоваться классами-сущностями. Это довольно удобно и при небольшой тренировке, позволяет составлять сложные запросы.
Запрос, который написан в методе createQuery, звучит примерно так:
("SELECT **b** FROM Book **b**") найти все состояния объекта **b** типа Book в таблице базы данных в которую отображается класс Book.
Метода getResultList() вернет экземпляр класса ArrayList с инициированными объектами Book, которые найдет в таблице.
Заметьте, часть «FROM Book **b**» в запросе означает создание объекта типа Book с именем **b**.
А часть «SELECT **b**» в запросе означает выбрать строку из таблицы, которую отображается

класс **Book** и инициировать этим состоянием экземпляр **b**.
Вот и оставшиеся методы.

```
74      @Override
        public List<Reader> loadReaders() {
76          try {
77              return em.createQuery("SELECT r FROM Reader r").getResultList();
78          } catch (Exception e) {
                return new ArrayList<Reader>();
80          }
81      }
82
83      @Override
        public List<LibHistory> loadLibHistoryes() {
85          try {
                return em.createQuery("SELECT h FROM libHistory h").getResultList();
87          } catch (Exception e) {
                return new ArrayList<LibHistory>();
89          }
90      }
```

Осталось подправить класс App, чтобы он начал использовать инструменты, которые мы создали.

Во первых, добавим поле типа нашего нового интерфейса и назовем его saver.
Инициируем его экземпляром класса, который реализует наш интерфейс.

```
    private Retentive saver = new PersistToDatabase();
```

Теперь надо считать из базы данных список книг, список читателей, список историй и инициировать соответствующие поля. Для этих целей предназначен конструктор.

```
28      public App() {
29          this.books = saver.loadBooks();
30          this.readers = saver.loadReaders();
31          this.libHistories=saver.loadLibHistoryes();
32      }
```

Еще необходимо изменить содержимое case из switch

```
case 1:
    Book book = manager.createBook();
    books.add(book);
    saver.saveBook(book);
    break;
```

следующие кейсы необходимо поменять соответственно.
Создадим commit -m"Шаг 8. Создание интерфейса Retentive и класса его реализующего PersistToDatabase. Исправление класса App"

Еще один шаг нам надо сделать. Добавить возможность выхода из функций добавления читателя и книги без добавления. Для этого необходимо сделать некоторые изменения некоторых классах.
А именно в классе App изменятся некоторые case в swich так:

```
case 1:
    Book book = manager.createBook();
    if(book != null){
        books.add(book);
        saver.saveBook(book);
    }
    break;
```

Таким образом, надо изменить и остальные case. Заметьте, что бы это работало, надо в некоторых классах добавить несколько строчек. Например в классе BookCreator изменим следующее:

```
26          System.out.println("Год издания книги:");
27          book.setYearPublishing(scanner.nextInt());
28          System.out.println(book.toString());
29          System.out.println("Для добавления введите любой символ\nДля отмены наберите -1");
            String yes = "";
31          yes = scanner.next();
32          if(!"".equals(yes)){
33              System.out.println("----Книа не добавлена----");
34              return null;
35          }else{
36              System.out.println("----Книга добавлена----");
37              return book;
38          }
```

Сравните самостоятельно с предыдущей версией. Логика такова. Предлагается ввести любой символ и тогда будет добавлена новая книга, если пользователь введет -1, то книга добавлена не будет и вместо экземпляра book будет возвращен null, в результате программа начнет новый цикл работы.

После рефакторинга в git репозитории добавим изменения и создадим новый commit с комментарием «Шаг 9. Добавление возможности выхода из функций без изменений»

Не забудьте «протолкнуть» изменения на github.com командой **git push origin master**.