# Report 2.

Magdalena Thomas
nr 155998

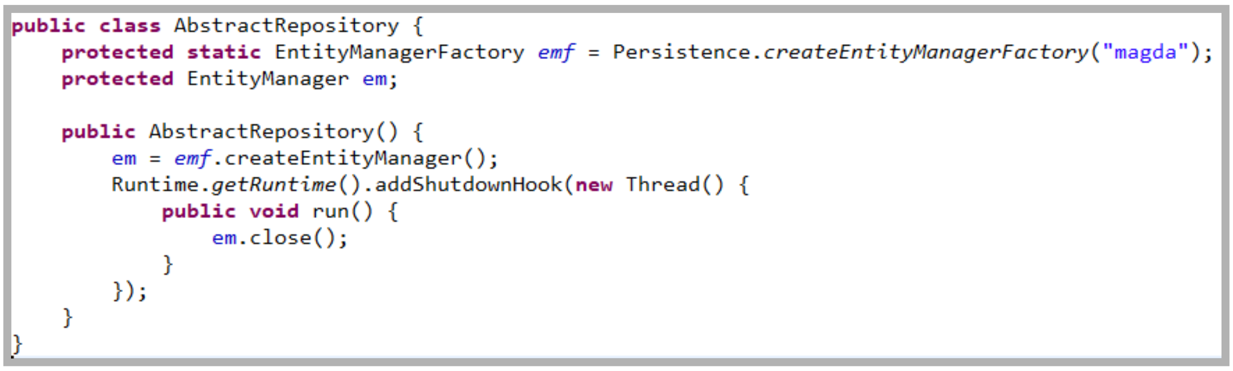
## Tests associated with multithreaded access

Program which simulates real traffic in the database is written in Java with a use of executors which allow us to perform tasks asynchronously. Four diffrent situations (TestCase) and three types of users are prepared.

- User – can only display the data from database (Picture 12.)
- Moderator – can update and display the data from database (Picture 13.)
- Admin – can delete, update and display the data from database (Picture 14.)

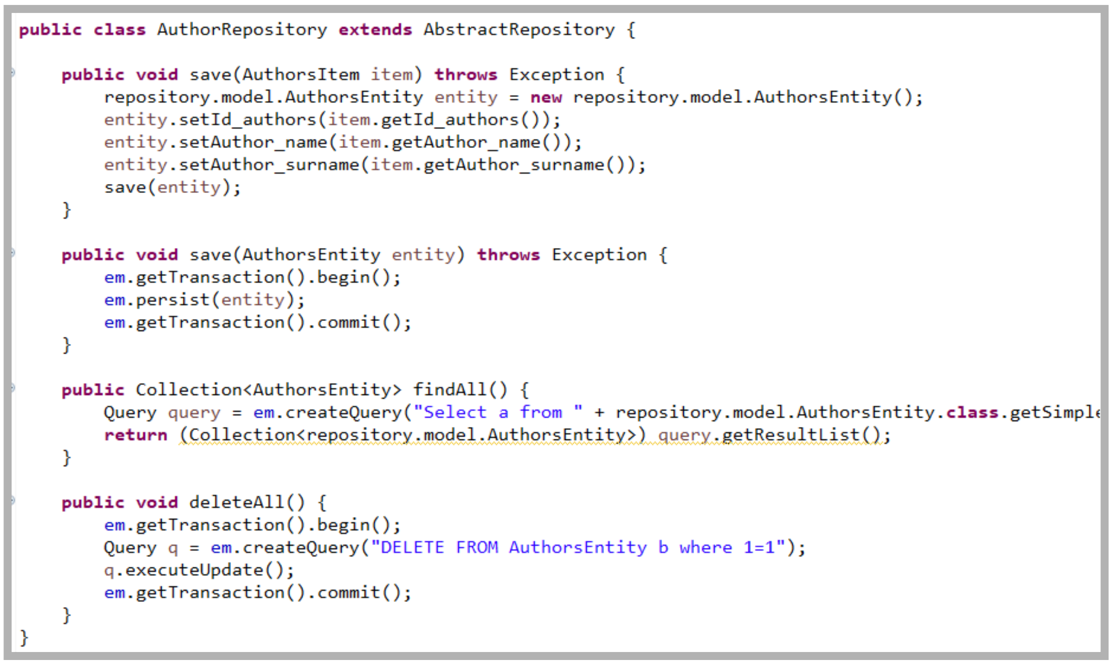
The threads make breaks between activites to implement the simulation, where the database's user can interrupt the activity and return to it after some period of time.

To the previous code, the Repository classes have been added (Abstarct Repository, AuthorRepository, BookRepository and CompanyRepository). AbstractRepository is a basic class with an Entitymanager Factory (Picture 1.). Other repository classes inherit from AbstractRepository class. Picture 2. presents the AuthorRepository class, which consists all of the methods connected with „Authors" table.



```
public class AbstractRepository {
    protected static EntityManagerFactory emf = Persistence.createEntityManagerFactory("magda");
    protected EntityManager em;

    public AbstractRepository() {
        em = emf.createEntityManager();
        Runtime.getRuntime().addShutdownHook(new Thread() {
            public void run() {
                em.close();
            }
        });
    }
}
```

*Picture 1. Abstract Repository class*

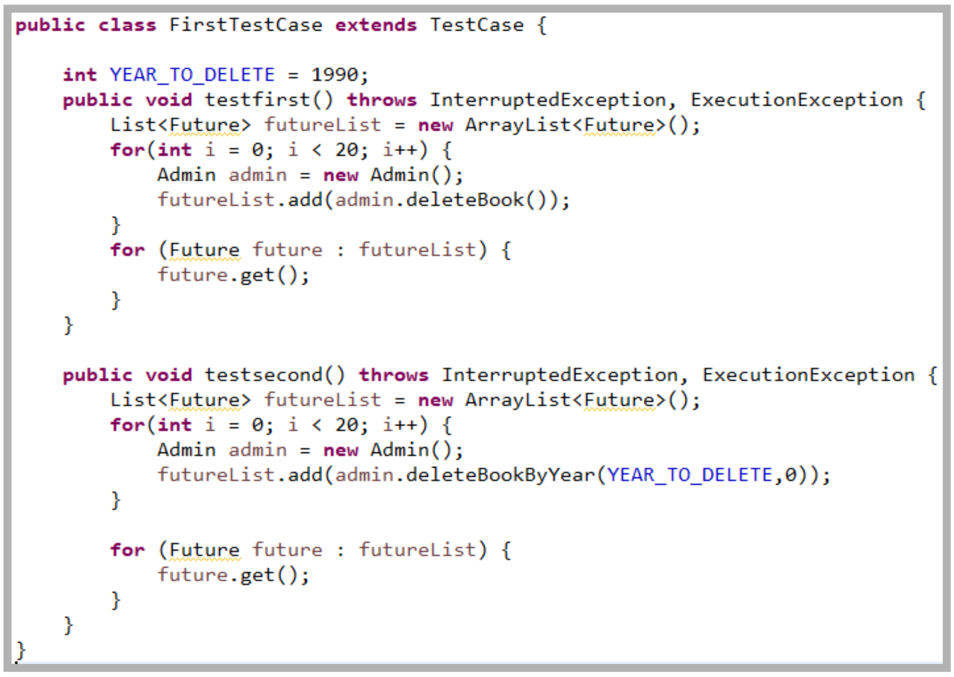


```
public class AuthorRepository extends AbstractRepository {

    public void save(AuthorsItem item) throws Exception {
        repository.model.AuthorsEntity entity = new repository.model.AuthorsEntity();
        entity.setId_authors(item.getId_authors());
        entity.setAuthor_name(item.getAuthor_name());
        entity.setAuthor_surname(item.getAuthor_surname());
        save(entity);
    }

    public void save(AuthorsEntity entity) throws Exception {
        em.getTransaction().begin();
        em.persist(entity);
        em.getTransaction().commit();
    }

    public Collection<AuthorsEntity> findAll() {
        Query query = em.createQuery("Select a from " + repository.model.AuthorsEntity.class.getSimple
        return (Collection<repository.model.AuthorsEntity>) query.getResultList();
    }

    public void deleteAll() {
        em.getTransaction().begin();
        Query q = em.createQuery("DELETE FROM AuthorsEntity b where 1=1");
        q.executeUpdate();
        em.getTransaction().commit();
    }
}
```

*Picture 2. AuthorRepository class as a example of inheritance AbstractRepository class*

## FirstTestCase

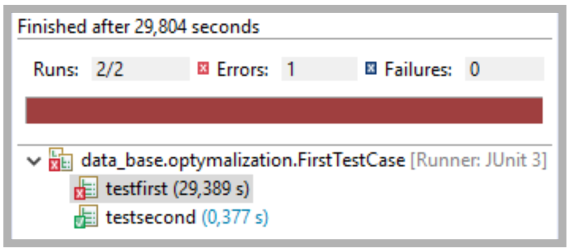
First Test (Picture 3. ) consists two methods – testfirst() during which admins delete first found book from database, and testsecond() – admins delete all books issued in a given year (in this example - 1990).



```
public class FirstTestCase extends TestCase {

    int YEAR_TO_DELETE = 1990;
    public void testfirst() throws InterruptedException, ExecutionException {
        List<Future> futureList = new ArrayList<Future>();
        for(int i = 0; i < 20; i++) {
            Admin admin = new Admin();
            futureList.add(admin.deleteBook());
        }
        for (Future future : futureList) {
            future.get();
        }
    }

    public void testsecond() throws InterruptedException, ExecutionException {
        List<Future> futureList = new ArrayList<Future>();
        for(int i = 0; i < 20; i++) {
            Admin admin = new Admin();
            futureList.add(admin.deleteBookByYear(YEAR_TO_DELETE,0));
        }

        for (Future future : futureList) {
            future.get();
        }
    }
}
```

*Picture 3. The code of FirstTestCase*

First test finished with error presented below:



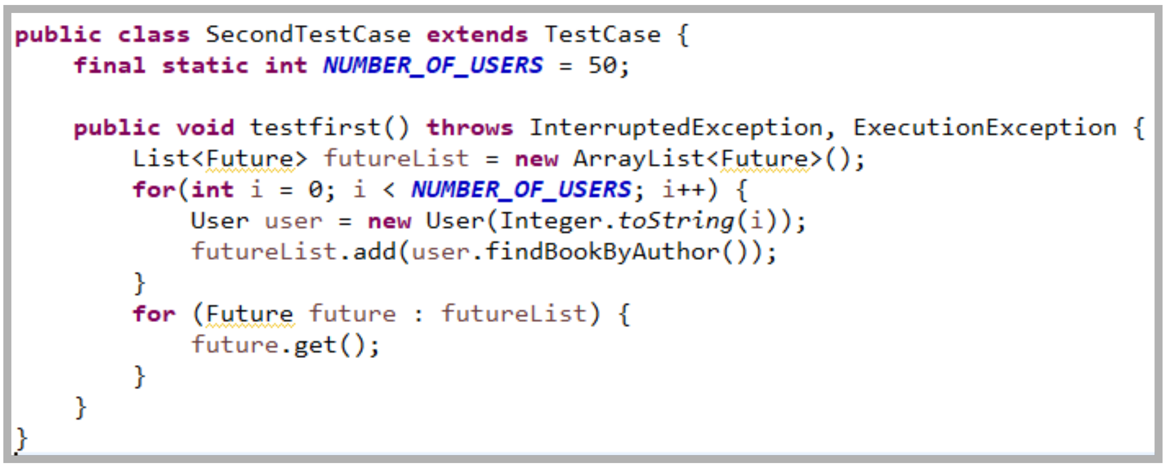
```
Finished after 29,804 seconds
Runs: 2/2    Errors: 1    Failures: 0
data_base.optymalization.FirstTestCase [Runner: JUnit 3]
    testfirst (29,389 s)
    testsecond (0,377 s)
```

java.util.concurrent.ExecutionException: javax.persistence.RollbackException: *Error while committing the transaction*

Caused by: java.sql.SQLException: *ORA-00060: podczas oczekiwania na zasób wykryto zakleszczenie*

**SecondTestCase**

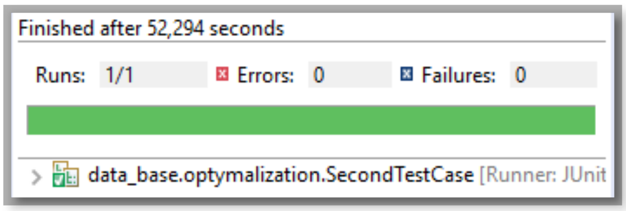
The second tested situation (Picture 4. ) was when users want to display the number of books divided into authors.



```java
public class SecondTestCase extends TestCase {
    final static int NUMBER_OF_USERS = 50;

    public void testfirst() throws InterruptedException, ExecutionException {
        List<Future> futureList = new ArrayList<Future>();
        for(int i = 0; i < NUMBER_OF_USERS; i++) {
            User user = new User(Integer.toString(i));
            futureList.add(user.findBookByAuthor());
        }
        for (Future future : futureList) {
            future.get();
        }
    }
}
```

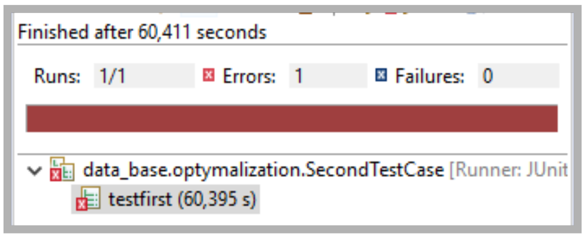
*Picture 4. The SecondTestCase code is presented.*

When number of users is lower or equals 50, the test finished good, without errors (Picture 5. ). Picture 6. presents errors observed when the number of users equals 100.



```
Finished after 52,294 seconds
Runs: 1/1    Errors: 0    Failures: 0
> data_base.optymalization.SecondTestCase [Runner: JUnit
```

*Picture 5. SecondTestCase with a 50 users*



```
Finished after 60,411 seconds

Runs:  1/1    Errors:  1    Failures:  0

data_base.optymalization.SecondTestCase [Runner: JUnit
  testfirst (60,395 s)
```

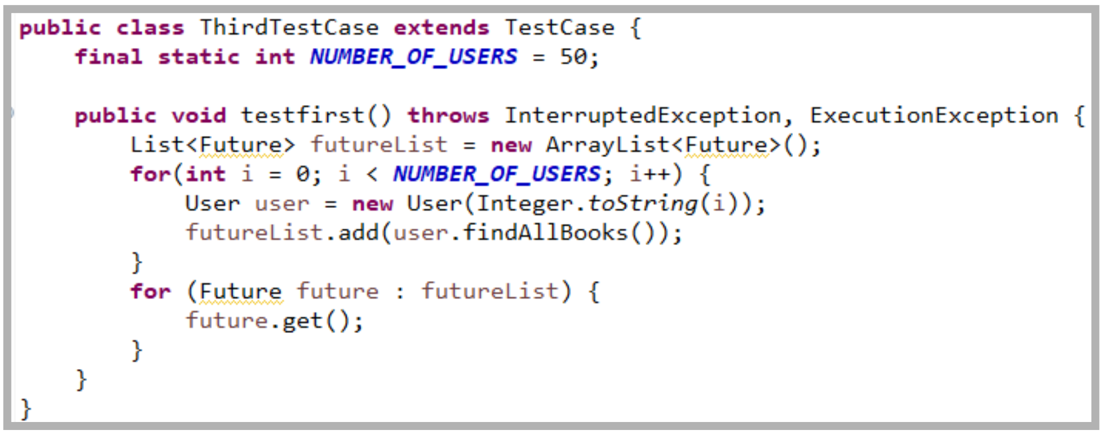
java.util.concurrent.ExecutionException: javax.persistence.PersistenceException: org.hibernate.HibernateException: *The internal connection pool has reached its maximum size and no connection is currently available!*

Caused by: javax.persistence.PersistenceException: org.hibernate.HibernateException: *The internal connection pool has reached its maximum size and no connection is currently available!*

*Picture 6. SecondTestCase with a 100 users*

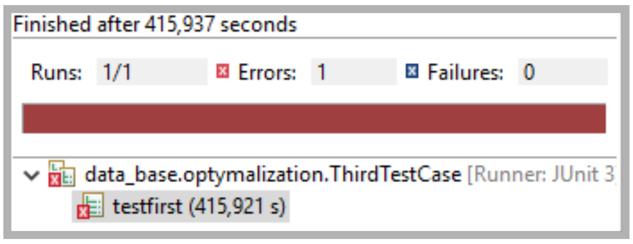
## ThirdTestCase

During the third activity (Picture 7. ) 50 users were trying to display all the books from database. Unfortunately, the „GC overhead limit exceeded” has been observed (Picture 8.).



```java
public class ThirdTestCase extends TestCase {
    final static int NUMBER_OF_USERS = 50;

    public void testfirst() throws InterruptedException, ExecutionException {
        List<Future> futureList = new ArrayList<Future>();
        for(int i = 0; i < NUMBER_OF_USERS; i++) {
            User user = new User(Integer.toString(i));
            futureList.add(user.findAllBooks());
        }
        for (Future future : futureList) {
            future.get();
        }
    }
}
```

*Picture 7. The code of ThirdTestCase*



```
Finished after 415,937 seconds

Runs:  1/1    Errors:  1    Failures:  0

data_base.optymalization.ThirdTestCase [Runner: JUnit 3
  testfirst (415,921 s)
```

java.util.concurrent.ExecutionException: java.lang.OutOfMemoryError: *GC overhead limit exceeded*

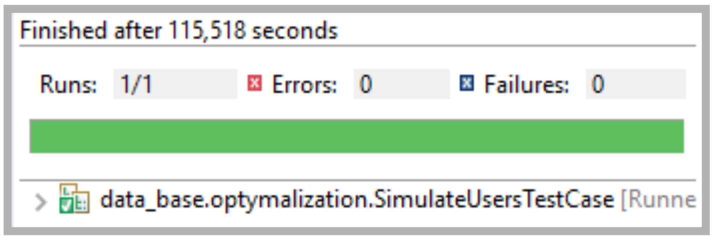
Caused by: java.lang.OutOfMemoryError: *GC overhead limit exceeded*

*Picture 8. The results of ThirdTestCase*

**FourthTest**

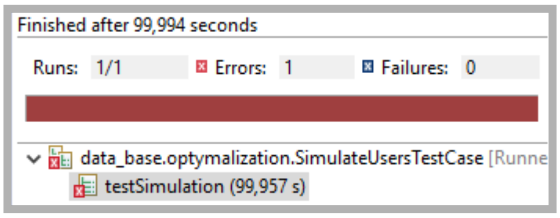
The last tested situation presents users, admins and moderators who in random manner display all the books from database, update the city in which book was issued, or remove books by random year. The choice of which activity will take place depends on random number. The fragments of test code is shown in Picture 11. The program has run three times and in every case diffrent results have been observed. When the number of threads equals 50 the test finished good for the first time, but during the second test an error showed on Picture 9. has been occured. If the number of threads have been changed to 52 an error has been noticed (Picture 10.). Diffrent errors arise as a result of random nature of this test.

a)



```
Finished after 115,518 seconds
Runs: 1/1    Errors: 0    Failures: 0
> data_base.optymalization.SimulateUsersTestCase [Runne
```

b)

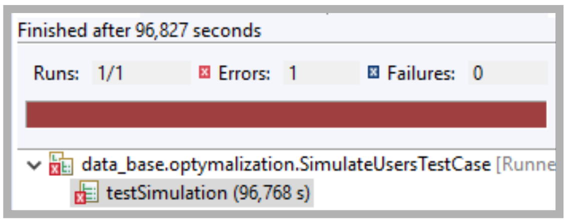


```
Finished after 99,994 seconds
Runs: 1/1    Errors: 1    Failures: 0
v data_base.optymalization.SimulateUsersTestCase [Runne
    testSimulation (99,957 s)
```

java.util.concurrent.ExecutionException: javax.persistence.PersistenceException: org.hibernate.exception.JDBCConnectionException: *Error calling Driver#connect*

Caused by: java.net.ConnectException: *Connection timed out: connect*

*Picture 9. The SimulationTestCase for 50 threads*
*a) when the test finished good, b) when an error has been observed*

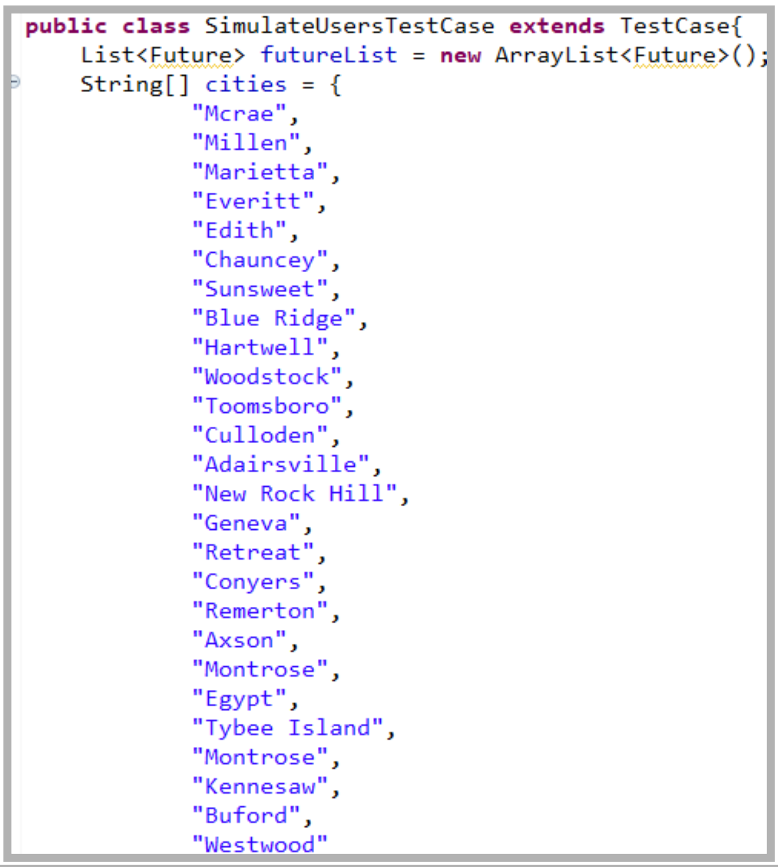


```
Finished after 96,827 seconds
Runs: 1/1    Errors: 1    Failures: 0
v data_base.optymalization.SimulateUsersTestCase [Runne
    testSimulation (96,768 s)
```

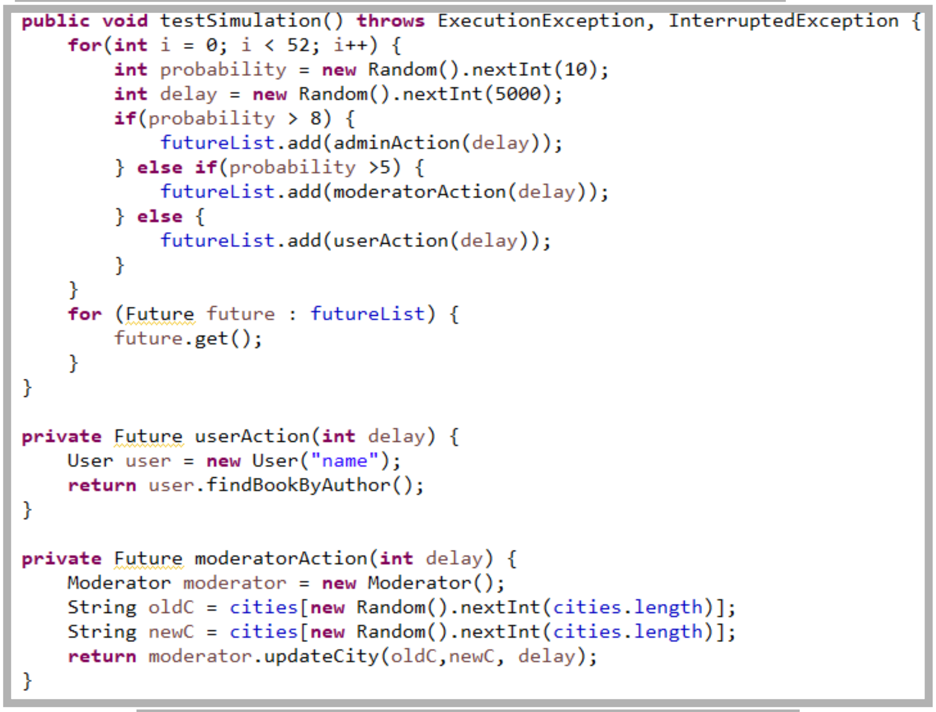
java.util.concurrent.ExecutionException: javax.persistence.PersistenceException: org.hibernate.HibernateException: *The internal connection pool has reached its maximum size and no connection is currently available!*

Caused by: javax.persistence.PersistenceException: org.hibernate.HibernateException: *The internal connection pool has reached its maximum size and no connection is currently available!*

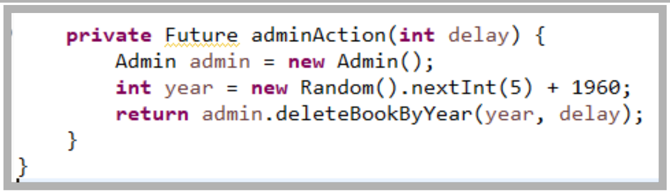
Picture 10. The SimulationTestCase when the numer of threads equals 52.



```java
public class SimulateUsersTestCase extends TestCase{
    List<Future> futureList = new ArrayList<Future>();
    String[] cities = {
            "Mcrae",
            "Millen",
            "Marietta",
            "Everitt",
            "Edith",
            "Chauncey",
            "Sunsweet",
            "Blue Ridge",
            "Hartwell",
            "Woodstock",
            "Toomsboro",
            "Culloden",
            "Adairsville",
            "New Rock Hill",
            "Geneva",
            "Retreat",
            "Conyers",
            "Remerton",
            "Axson",
            "Montrose",
            "Egypt",
            "Tybee Island",
            "Montrose",
            "Kennesaw",
            "Buford",
            "Westwood"
```

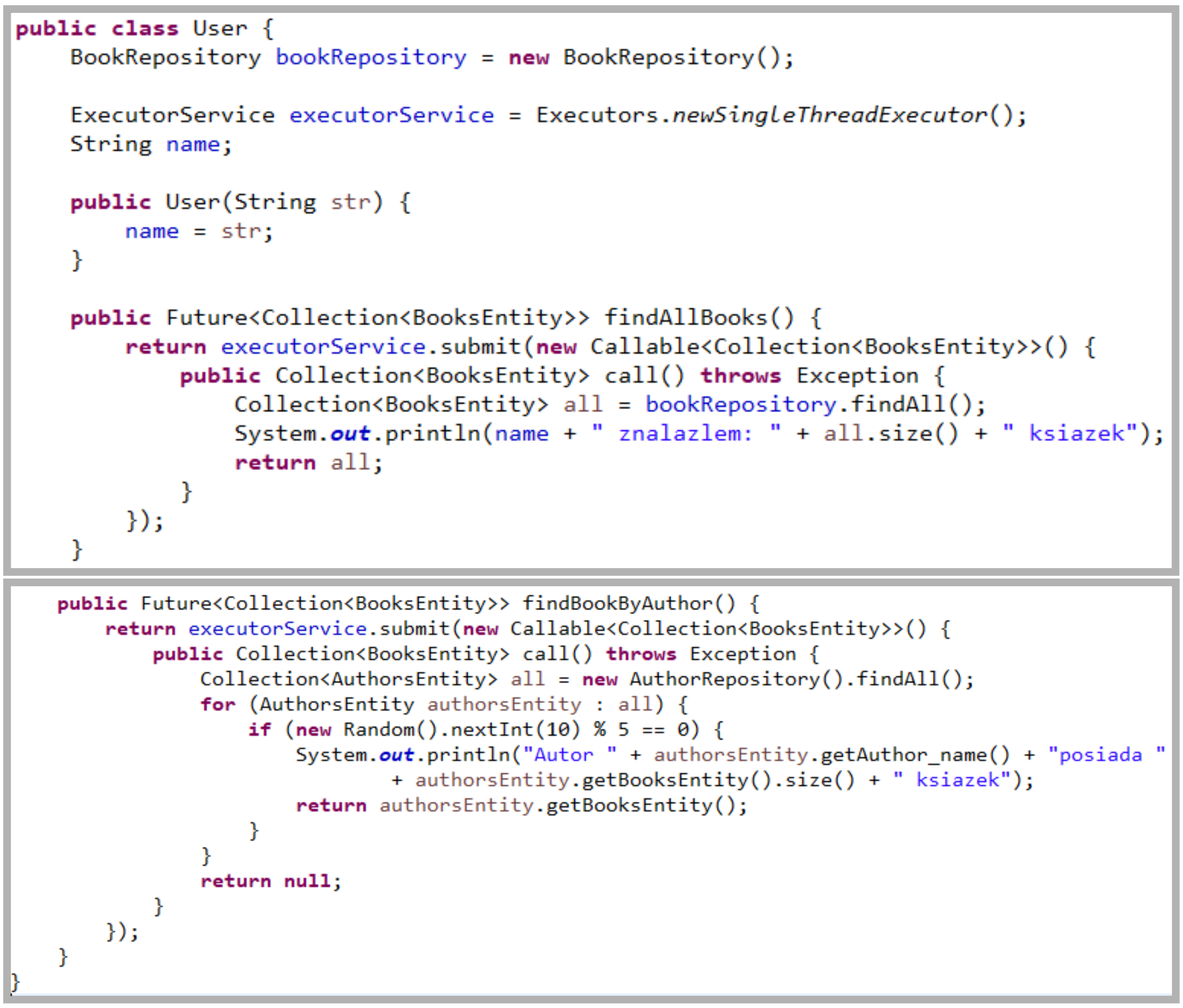


```java
public void testSimulation() throws ExecutionException, InterruptedException {
    for(int i = 0; i < 52; i++) {
        int probability = new Random().nextInt(10);
        int delay = new Random().nextInt(5000);
        if(probability > 8) {
            futureList.add(adminAction(delay));
        } else if(probability >5) {
            futureList.add(moderatorAction(delay));
        } else {
            futureList.add(userAction(delay));
        }
    }
    for (Future future : futureList) {
        future.get();
    }
}

private Future userAction(int delay) {
    User user = new User("name");
    return user.findBookByAuthor();
}

private Future moderatorAction(int delay) {
    Moderator moderator = new Moderator();
    String oldC = cities[new Random().nextInt(cities.length)];
    String newC = cities[new Random().nextInt(cities.length)];
    return moderator.updateCity(oldC,newC, delay);
}
```



```java
    private Future adminAction(int delay) {
        Admin admin = new Admin();
        int year = new Random().nextInt(5) + 1960;
        return admin.deleteBookByYear(year, delay);
    }
}
```

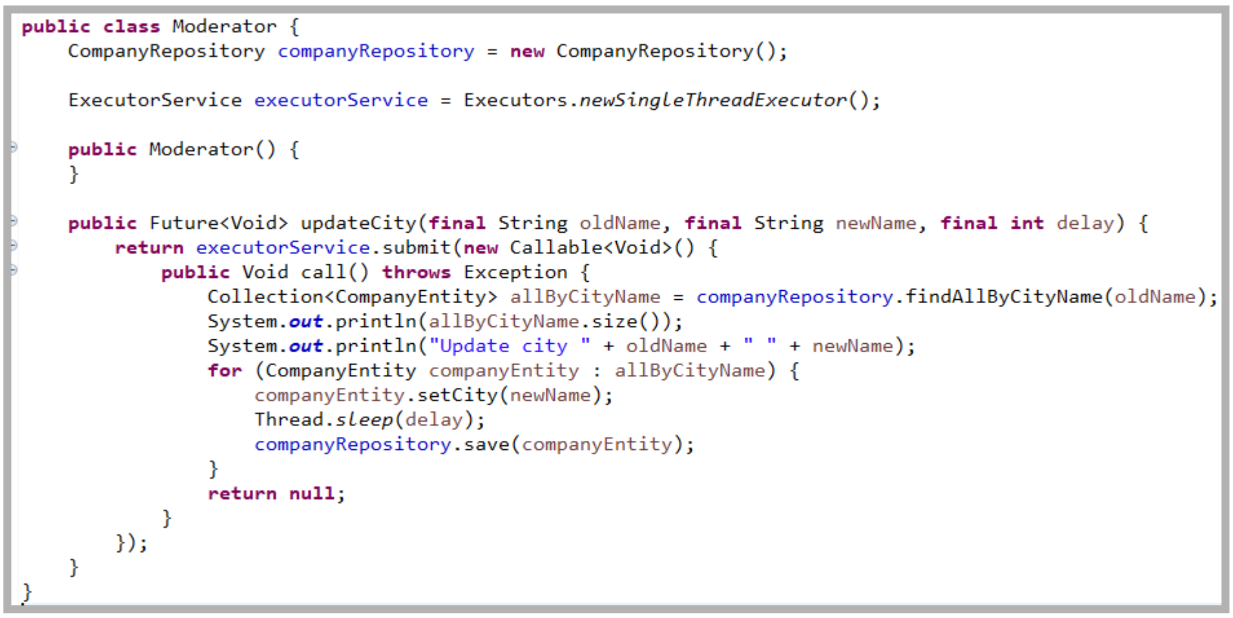
*Picture 11. Fragments of fourth test's code with multithread access*



```java
public class User {
    BookRepository bookRepository = new BookRepository();

    ExecutorService executorService = Executors.newSingleThreadExecutor();
    String name;

    public User(String str) {
        name = str;
    }

    public Future<Collection<BooksEntity>> findAllBooks() {
        return executorService.submit(new Callable<Collection<BooksEntity>>() {
            public Collection<BooksEntity> call() throws Exception {
                Collection<BooksEntity> all = bookRepository.findAll();
                System.out.println(name + " znalazlem: " + all.size() + " ksiazek");
                return all;
            }
        });
    }
```

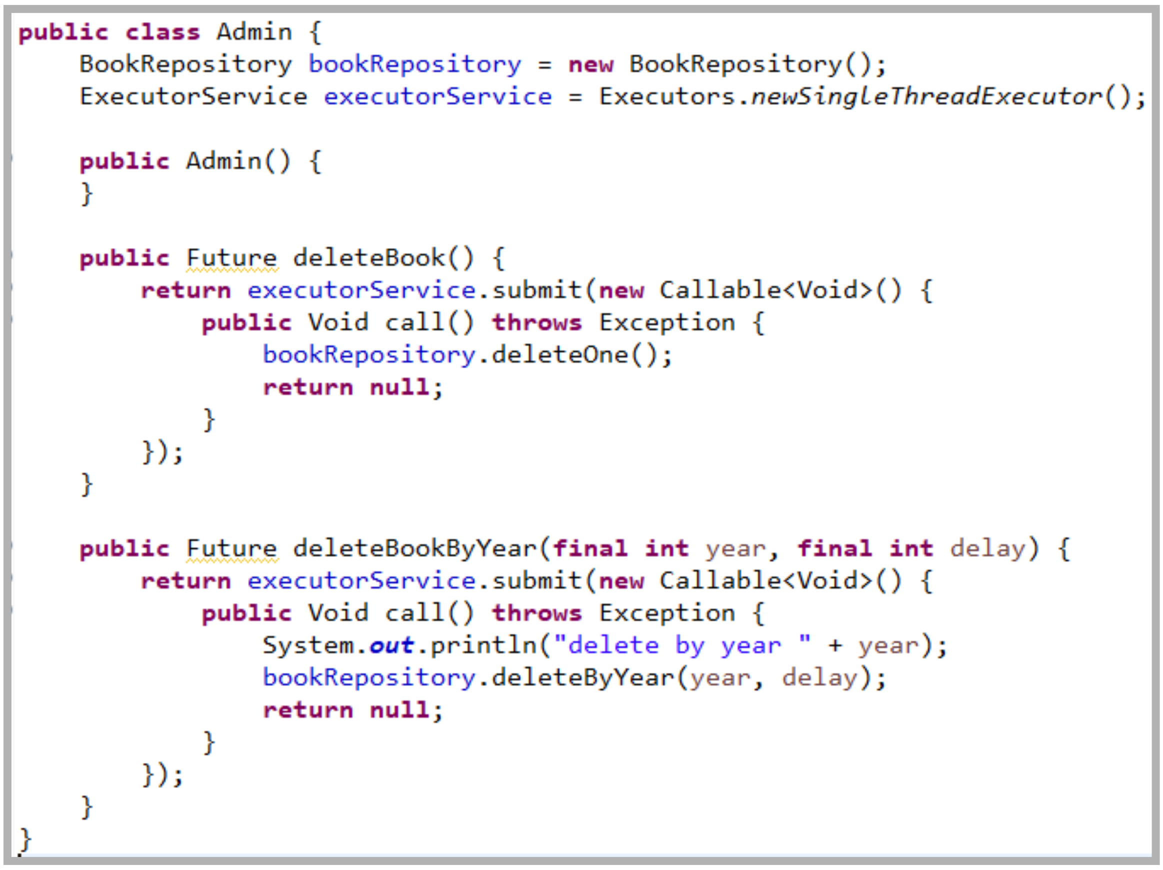
```java
    public Future<Collection<BooksEntity>> findBookByAuthor() {
        return executorService.submit(new Callable<Collection<BooksEntity>>() {
            public Collection<BooksEntity> call() throws Exception {
                Collection<AuthorsEntity> all = new AuthorRepository().findAll();
                for (AuthorsEntity authorsEntity : all) {
                    if (new Random().nextInt(10) % 5 == 0) {
                        System.out.println("Autor " + authorsEntity.getAuthor_name() + "posiada "
                                + authorsEntity.getBooksEntity().size() + " ksiazek");
                        return authorsEntity.getBooksEntity();
                    }
                }
                return null;
            }
        });
    }
}
```

*Picture 12. The User class*



```java
public class Moderator {
    CompanyRepository companyRepository = new CompanyRepository();

    ExecutorService executorService = Executors.newSingleThreadExecutor();

    public Moderator() {
    }

    public Future<Void> updateCity(final String oldName, final String newName, final int delay) {
        return executorService.submit(new Callable<Void>() {
            public Void call() throws Exception {
                Collection<CompanyEntity> allByCityName = companyRepository.findAllByCityName(oldName);
                System.out.println(allByCityName.size());
                System.out.println("Update city " + oldName + " " + newName);
                for (CompanyEntity companyEntity : allByCityName) {
                    companyEntity.setCity(newName);
                    Thread.sleep(delay);
                    companyRepository.save(companyEntity);
                }
                return null;
            }
        });
    }
}
```

*Picture 13. The Moderator class*



```java
public class Admin {
    BookRepository bookRepository = new BookRepository();
    ExecutorService executorService = Executors.newSingleThreadExecutor();

    public Admin() {
    }

    public Future deleteBook() {
        return executorService.submit(new Callable<Void>() {
            public Void call() throws Exception {
                bookRepository.deleteOne();
                return null;
            }
        });
    }

    public Future deleteBookByYear(final int year, final int delay) {
        return executorService.submit(new Callable<Void>() {
            public Void call() throws Exception {
                System.out.println("delete by year " + year);
                bookRepository.deleteByYear(year, delay);
                return null;
            }
        });
    }
}
```

*Picture 14. The admin class*