**ОТЧЁТ**

**«Модульное тестирование выбранной части кода (этап 3 курсовой работы)»**

по дисциплине «Процессы управления качеством программного обеспечения»

|  |  |
| --- | --- |
| Выполнила  студентка гр. 3530904/90102 | Ли Ицзя |
| Руководитель | Котлярова Л. П. |

**Оглавление**

[3 Модульные тесты (Unit test) 3](#_Toc162433626)

[3.1 Выбранный фреймворк тестирования 3](#_Toc162433627)

[3.1.1 Общее описание и функциональность фреймворка 3](#_Toc162433628)

[3.1.2 Причины выбора данного фреймворка 3](#_Toc162433629)

[3.2 Установка и настройка инструмента 3](#_Toc162433630)

[3.2.1 Установка 3](#_Toc162433631)

[3.2.2 Настройка инструмента 4](#_Toc162433632)

[3.3 Стратегия тестирования 5](#_Toc162433633)

[3.3.1 Общие подходы 5](#_Toc162433634)

[3.3.2 Класс AdminServiceImplTest 5](#_Toc162433635)

[3.3.3 Класс AuthServiceImpl 8](#_Toc162433636)

[3.3.4 Класс RoleServiceImpl 8](#_Toc162433637)

[3.3.5 Класс MenuServiceImpl 10](#_Toc162433638)

[3.4 Написание и запуск тест-кейсов 10](#_Toc162433639)

[3.4.1 Написание модульных тестов 10](#_Toc162433640)

[3.4.2 Результат первого запуска тестов 11](#_Toc162433641)

[3.5 Анализ результатов выполнения теста 12](#_Toc162433642)

[3.6 Исправление кода и модульных тестов 13](#_Toc162433643)

[3.6.1 Исправление кода 13](#_Toc162433644)

[3.6.2 окончательный результат 14](#_Toc162433645)

[3.7 Статистика покрытия кода и таблица ошибок 16](#_Toc162433646)

[3.7.1 Статистика покрытия кода 16](#_Toc162433647)

[3.7.2 Таблицы ошибок 16](#_Toc162433648)

[3.8 Вывод 17](#_Toc162433649)

[Приложение 1 17](#_Toc162433650)

[Модульные тесты для класса AdminServiceImplTest 17](#_Toc162433651)

[Модульные тесты для класса AuthServiceImpl 19](#_Toc162433652)

[Модульные тесты для класса RoleServiceImpl 21](#_Toc162433653)

[Модульные тесты для класса MenuServiceImpl 23](#_Toc162433654)

# 3 Модульные тесты (Unit test)

Для написания модульного тестирования были отобраны следующие классы:

* AdminServiceImpl
* AuthServiceImpl
* RoleServiceImpl
* MenuServiceImpl

В сумме эти классы составляют 323 строк исходного кода.

Для написания и выполнения модульных тестов использован фреймворк Mockito.

## 3.1 Выбранный фреймворк тестирования

### 3.1.1 Общее описание и функциональность фреймворка

Mockito в настоящее время является самой популярной платформой Java Mock. Используя Mock framework, мы можем виртуализировать внешнюю зависимость, уменьшить связь между тестовыми компонентами, сосредоточиться только на процессе и результатах кода и по-настоящему достичь цели тестирования.

### 3.1.2 Причины выбора данного фреймворка

Классы, которые мы тестируем, часто зависят от многих объектов. Чтобы избежать создания вручную всей цепочки зависимостей bean-компонентов, мы решили использовать Mock framework.

## 3.2 Установка и настройка инструмента

### 3.2.1 Установка

Мы представляем фреймворк Mokito через Maven.

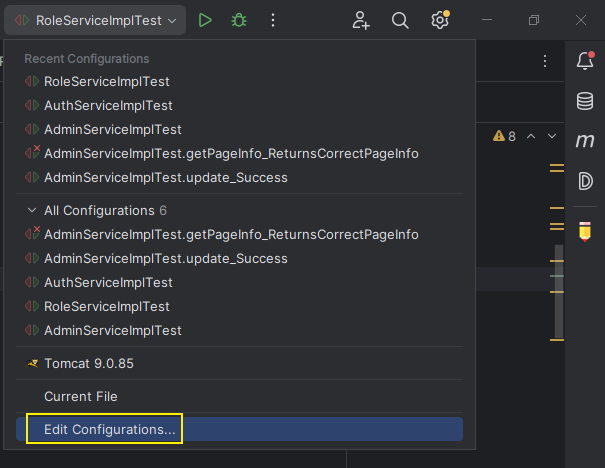
Импорт зависимости JUnit и Mokito в файл pom.xml проекта:

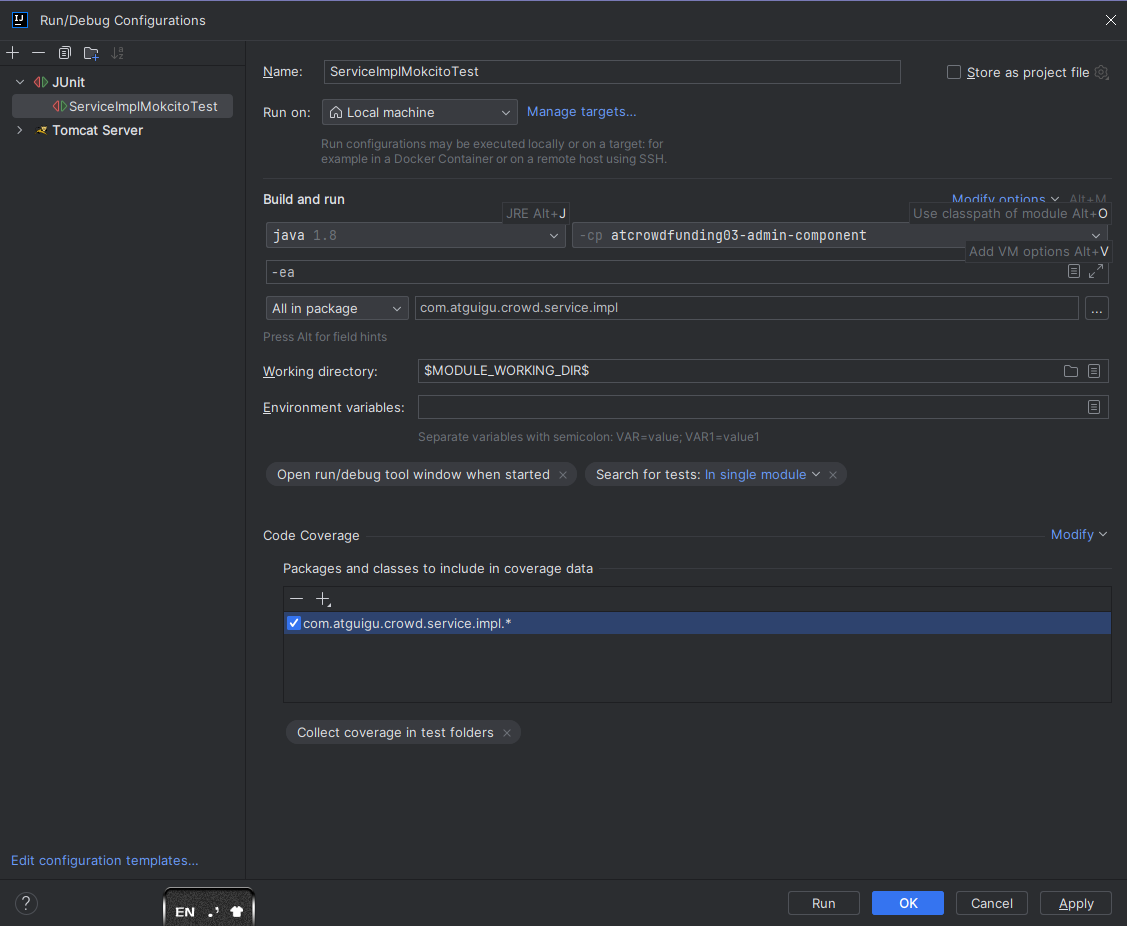
文本

描述已自动生成

### 3.2.2 Настройка инструмента

Чтобы реализовать пакетное тестирование и генерировать отчеты о тестовом покрытии, нам необходимо настроить JUnit.





## 3.3 Стратегия тестирования

### 3.3.1 Общие подходы

1. Изолированное тестирование: Каждый метод будет тестироваться отдельно, чтобы обеспечить точность результатов.

2. Данные для тестирования: Для тестов будут созданы специфические тестовые данные, включая положительные и отрицательные сценарии.

**3. Не только проверять правильность вывода метода, но и проверять ошибки, которые могут быть вызваны каждым оператором, в зависимости от структуры метода.**

### 3.3.2 Класс AdminServiceImplTest

Класс AdminServiceImplTest это конкретная реализация интерфейса AdminService, реализующая логику работы объектов Admin. Все внешние зависимости, такие как AdminMapper, будут замокированы с использованием библиотеки мокирования, чтобы изолировать тестируемую логику. Буду протестировать методы:

**1. saveAdmin(Admin admin)**

Для метода saveAdmin класса AdminService, реализующего логику сохранения сущности администратора с предварительным шифрованием пароля и установкой времени создания, можно предложить следующие тестовые случаи:

* Положительный тест (Успешное Сохранение):
  + Сценарий: Попытка сохранить валидную сущность администратора.
  + Предпосылки: Передача в метод объекта Admin с валидными данными.
  + Ожидаемый результат: Администратор успешно сохранён в базу данных, пароль зашифрован, установлено текущее время создания. Никаких исключений не возникает.
* Отрицательный тест: Проверка обработки ситуации с дублированием ключа (DuplicateKeyException). Ожидается выброс исключения LoginAcctAlreadyInUseException.

1) Обработка Дублирования Ключа

* + Сценарий: Попытка сохранить администратора с логином, который уже существует в базе данных.
  + Предпосылки: Передача в метод объекта Admin с loginAcct, уже существующим в базе.
  + Ожидаемый результат: Возникновение исключения DuplicateKeyException и его обработка с пробросом LoginAcctAlreadyInUseException.

2) Сохранение Администратора с Null Параметрами

* Сценарий: Попытка сохранить администратора с null значениями в обязательных полях.
* Предпосылки: Передача в метод объекта Admin с null в одном из обязательных полей (например, userPswd или loginAcct).
* Ожидаемый результат: Возникновение и обработка исключения (если предусмотрено логикой) или отклонение операции сохранения с соответствующей ошибкой.

3) Попытка Сохранения Сущности С Null Значением

* Сценарий: Попытка вызова метода saveAdmin с null в качестве аргумента.
* Предпосылки: admin == null.
* Ожидаемый результат: Корректная обработка ситуации без возникновения NullPointerException.
* Тесты Шифрования

4) Тесты Шифрования

* Сценарий: Проверка, что пароль администратора был зашифрован перед сохранением.
* Предпосылки: Передача в метод объекта Admin с известным паролем.
* Ожидаемый результат: Пароль, сохранённый в объекте Admin, переданный в adminMapper.insert, зашифрован с использованием MD5.

5) Тесты Установки Времени

* Сценарий: Установка текущего времени создания при сохранении администратора.
* Предпосылки: Вызов метода saveAdmin с валидным объектом Admin.
* Ожидаемый результат: В объекте Admin, переданном в adminMapper.insert, установлено текущее время в соответствующем формате.

**2. getAll()**

* Положительный тест: Получение Непустого Списка Администраторов
  + Сценарий: В базе данных присутствуют записи администраторов.
  + Предпосылки: База данных содержит несколько записей администраторов.
  + Ожидаемый результат: Метод возвращает список, содержащий все записи администраторов, присутствующих в базе данных. Размер списка соответствует количеству записей в базе.

**3. getAdminByLoginAcct(String loginAcct, String userPswd)**

* Положительный тест:
  + Сценарий: Попытка аутентификации с валидными учетными данными.
  + Предпосылки: Администратор с данным логином и паролем существует в системе.
  + Ожидаемый результат: Метод возвращает объект Admin, соответствующий введенным учетным данным.
* Отрицательные тесты:
* Проверка обработки ситуации, когда администратор не найден (ожидается LoginFailedException).
* Проверка ситуации с неуникальным результатом поиска (ожидается RuntimeException).
* Проверка несовпадения паролей (ожидается LoginFailedException).

**4. remove(Integer adminId)**

* Положительный тест: Проверка успешного удаления администратора по идентификатору.
* Отрицательный тест: Проверка обработки ситуации при попытке удаления несуществующего администратора.

**5. update(Admin admin)**

* Положительный тест: Проверка успешного обновления данных администратора.
* Отрицательный тест: Проверка обработки дублирования ключевых полей, ожидается LoginAcctAlreadyInUseForUpdateException.

**6. saveAdminRoleRelationship(Integer adminId, List<Integer> roleIdList)**

* Положительный тест: Проверка сохранения связей ролей для администратора с корректным списком ролей.
* Отрицательный тест: Проверка поведения метода при передаче пустого списка или null. Ожидается корректное удаление старых связей без добавления новых.

### 3.3.3 Класс AuthServiceImpl

Класс AuthServiceImpl реализует логику назначения разрешений ролям. Все внешние зависимости, такие как AuthMapper, будут замокированы с использованием библиотеки мокирования, чтобы изолировать тестируемую логику.Буду протестировать методы:

**1. getAll()**

* Положительный тест: Получение Непустого Списка назначения разрешений ролям
  + Сценарий: В базе данных присутствуют записи назначения разрешений ролям.
  + Предпосылки: База данных содержит несколько записей назначения разрешений ролям.
  + Ожидаемый результат: Метод возвращает список назначения разрешений ролям, присутствующих в базе данных. Размер списка соответствует количеству записей в базе.

**2. getAssignedAuthIdByRoleId(Integer roleId)**

* Положительный тест: Проверка получения списка идентификаторов прав доступа, назначенных определенной роли. Валидация возвращаемого списка на соответствие ожидаемым правам.
* Отрицательный тест: Проверка метода с несуществующим идентификатором роли. Ожидается возвращение пустого списка.

### 3.3.4 Класс RoleServiceImpl

Класс RoleServiceImpl реализует логические операции над ролями. Все внешние зависимости, такие как RoleMapper, будут замокированы с использованием библиотеки мокирования, чтобы изолировать тестируемую логику.

Буду протестировать методы:

**1. getPageInfo(Integer pageNum, Integer pageSize, String keyword)**

* Положительный тест: Проверка возвращения корректной PageInfo при валидных параметрах pageNum, pageSize и keyword. Тест должен убедиться, что возвращаемая информация соответствует ожиданиям и содержит правильный набор ролей.
* Отрицательный тест: Проверка поведения метода при некорректных параметрах пагинации (отрицательные значения pageNum и pageSize). Ожидается, что метод корректно обрабатывает такие ситуации, возможно, применяя значения по умолчанию.

**2. saveRole(Role role)**

* Положительный тест: Проверка успешного сохранения роли с корректными данными. Тест должен подтвердить, что метод insert вызывается с правильным объектом Role.
* Отрицательный тест: Попытка сохранения роли с некорректными данными (например, null или отсутствие обязательных полей). Ожидается обработка такой ситуации без возникновения исключений.

**3. updateRole(Role role)**

* Положительный тест: Проверка успешного обновления роли с корректными изменениями. Необходимо проверить, что метод updateByPrimaryKey вызывается с правильным объектом Role.
* Отрицательный тест: Попытка обновления роли с некорректными данными или несуществующим ID. Тест должен убедиться, что метод адекватно обрабатывает такие случаи.

**4. removeRole(List<Integer> roleIdList)**

* Положительный тест: Проверка удаления списка ролей по идентификаторам. Тест должен подтвердить вызов метода deleteByExample с правильными параметрами.
* Отрицательный тест: Попытка удаления ролей с несуществующими идентификаторами или передача пустого списка. Тест должен показать, что сервис корректно обрабатывает такие ситуации.

**5. getAssignedRole(Integer adminId) и getUnAssignedRole(Integer adminId)**

* Положительный тест: Проверка получения списков назначенных и неназначенных ролей для администратора с существующим ID. Тесты должны удостовериться, что возвращаемые списки соответствуют ожиданиям.
* Отрицательный тест: Попытка получения списков ролей для несуществующего администратора. Ожидается, что методы вернут пустые списки без возникновения ошибок.

### 3.3.5 Класс MenuServiceImpl

Класс MenuServiceImpl реализует функцию администраторов по обслуживанию меню страниц. Администраторы могут добавлять элементы, изменять элементы, удалять элементы и т. д. в меню.

Буду протестировать методы:

**1. getAll()**

* Положительный тест: Проверить, что метод возвращает полный список объектов Menu, и эти объекты соответствуют данным, хранящимся в базе данных.
* Отрицательный тест: Проверить поведение метода при отсутствии записей в базе данных. Ожидается получение пустого списка.

**2. saveMenu(Menu menu)**

* Положительный тест: Проверить, что метод корректно сохраняет объект Menu в базу данных. Валидация успешного сохранения может включать проверку вызова соответствующего метода MenuMapper с правильными параметрами.
* Отрицательный тест: Проверить поведение метода при попытке сохранить null или объект Menu с некорректными данными (например, с отрицательным ID или null в обязательных полях).

**3. updateMenu(Menu menu)**

* Положительный тест: Проверить, что метод обновляет существующую запись в базе данных без изменения поля "pid", если оно не предоставлено.
* Отрицательный тест: Проверить обработку методом попытки обновить объект Menu с некорректными данными, например, null или с несуществующим в базе данных ID.

**4. removeMenu(Integer id)**

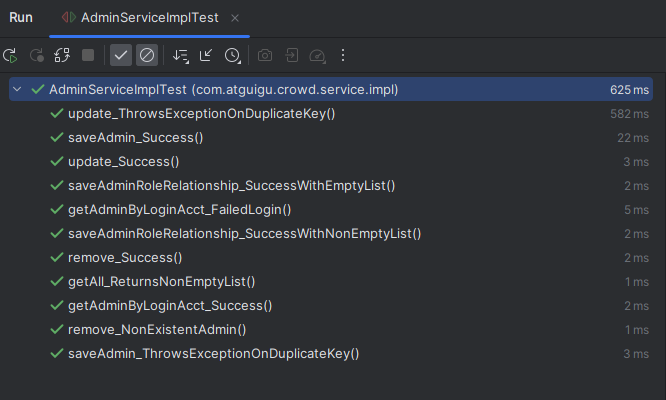
* Положительный тест: Проверить, что метод удаляет объект Menu по заданному ID.
* Отрицательный тест: Проверить, как метод обрабатывает ситуацию с попыткой удаления объекта по несуществующему ID. Также стоит проверить обработку ситуации, когда в метод передается null.

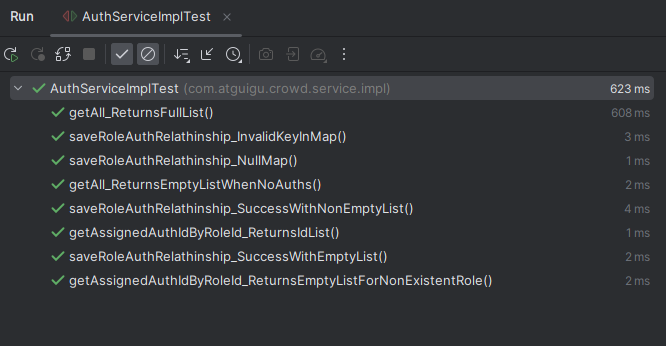
## 3.4 Написание и запуск тест-кейсов

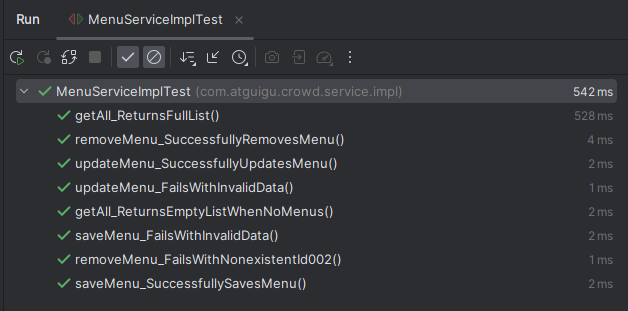
### 3.4.1 Написание модульных тестов

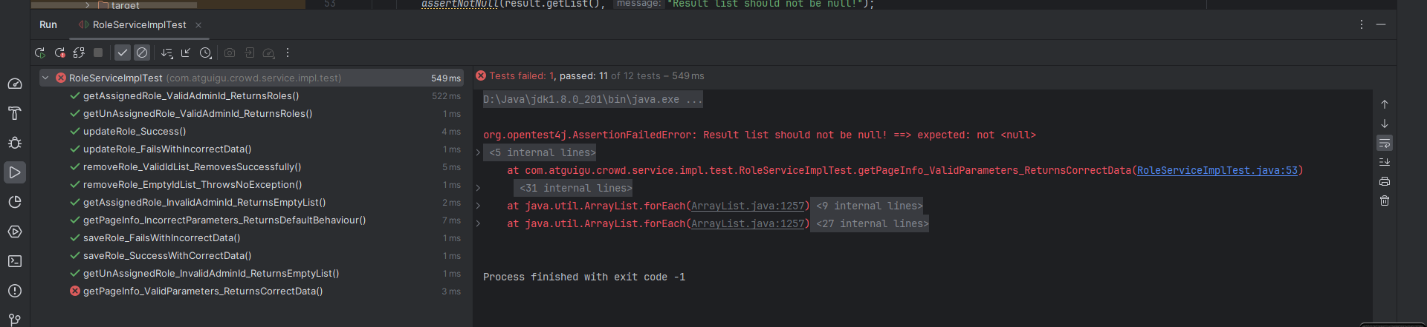
см. Приложение 1.

### 3.4.2 Результат первого запуска тестов









## 3.5 Анализ результатов выполнения теста

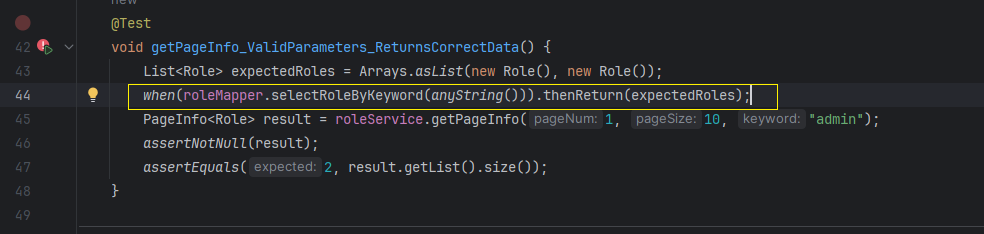
На скриншоте мы видим, что проверка getPageInfo класса RoleServiceImpl не удалась.

Сценарий этого теста: введим параметры исключения в метод getPageInfo и ожидаем, что getPageInfo вернет правильные данные.

Однако тест не прошел. Согласно предоставленной трассировке стека, тест не пройден, поскольку при выполнении метода RoleServiceImplTest.getPageInfo\_ValidParameters\_ReturnsCorrectData возникло исключение NullPointerException. Обычно это указывает на попытку доступа или манипулирования объектом, который не был должным образом инициализирован.

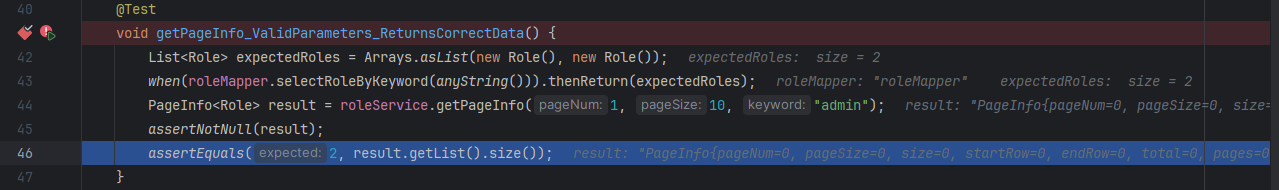
Мы предполагаем, что причиной могут быть:

1. Код в строке 44 возвращает не ожидаемый массив, а нулевое значение.



2. Сам метод getPageInfo во время выполнения обращается к нулевому объекту.

Начинем отладку:



После отладки мы нашли проблему! Результат (тип PageInfo<Role>), возвращаемый методом getPageInfo, не был инициализирован, поэтому получение массива результатов приведет к исключению для доступа к нулевому указателю, поэтому тест не пройден.

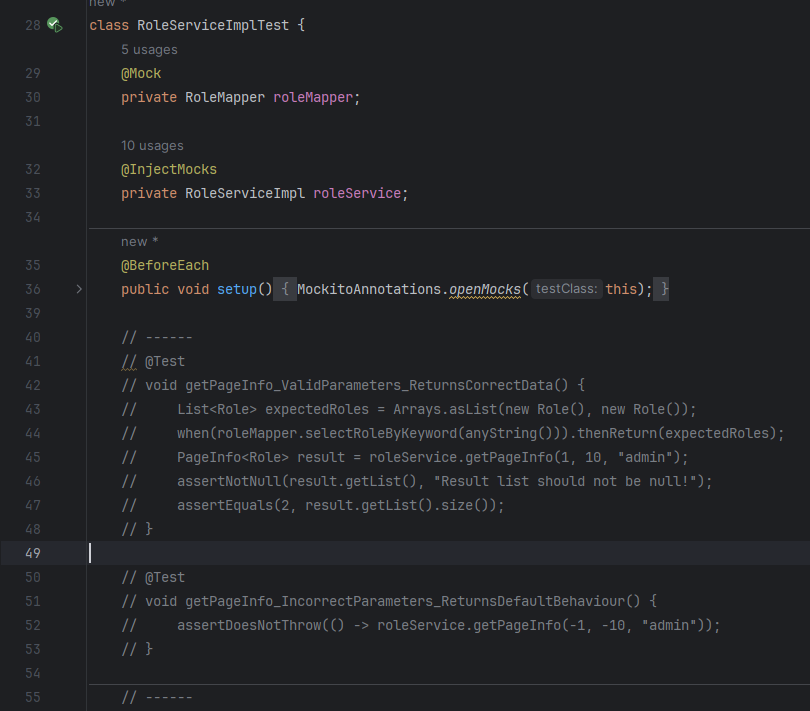
Почему возвращаемый результат result не инициализируется? После анализа мы выяснили, что это происходит потому, что метод getPageInfo внутренне использует подключаемый модуль подкачки MyBatis PageHelper. Роль PageHelper заключается в реализации подкачки **на уровне запроса к базе данных**. Однако при модульном тестировании PageInfo является всего лишь оболочкой для результатов запроса и не может фактически подключаться к базе данных и выполнять операторы SQL.

Итак, мы получаем следующий вывод: неразумно проводить модульное тестирование при подкачке, потому что эффект подкачки не может быть проверен при модульном тестировании.

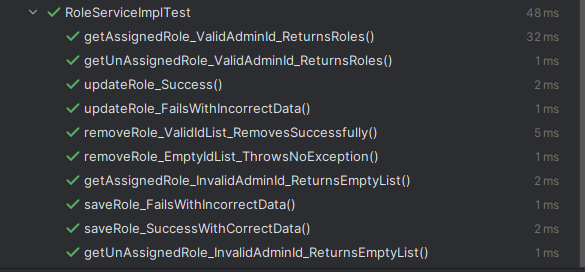
## 3.6 Исправление кода и модульных тестов

### 3.6.1 Исправление кода

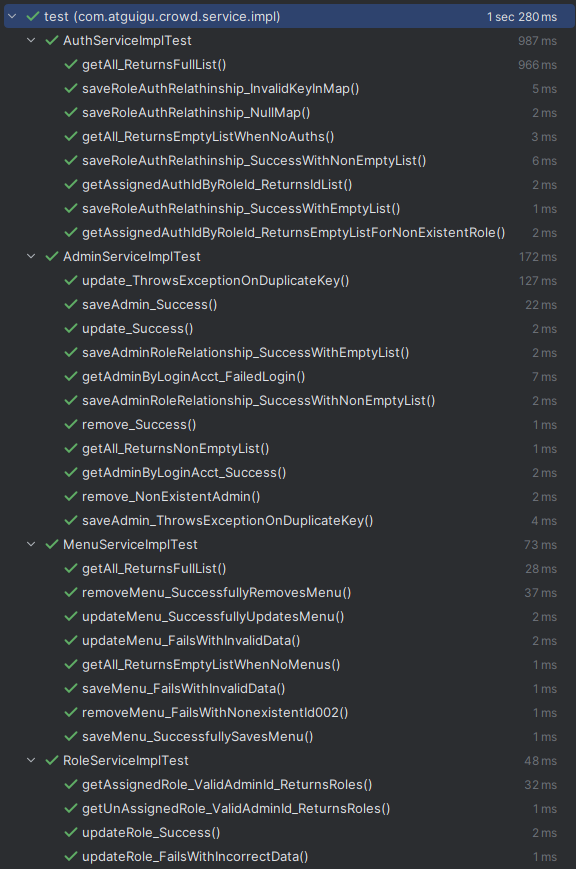
На основании приведенного выше анализа мы решили убрать тест метода getPageInfo из модульного теста. (**Буду сделать это в рамках интеграционного теста, где мы можем работать с реальной базой данных.**)

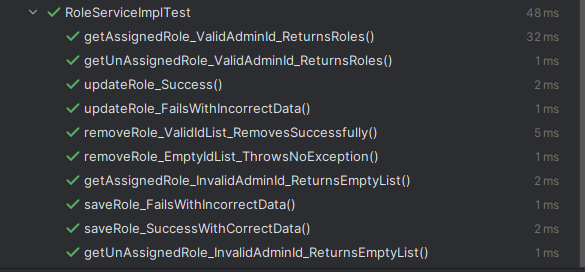


Выполнив модульный тест еще раз, получаем окончательный результат



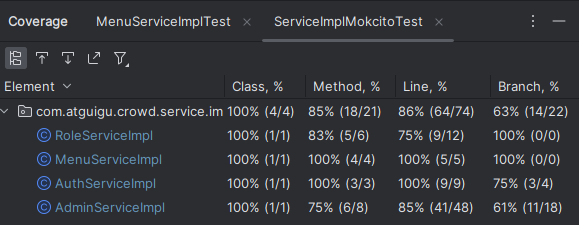
3.6.2 окончательный результат





## 3.7 Статистика покрытия кода и таблица ошибок

### 3.7.1 Статистика покрытия кода



### 3.7.2 Таблицы ошибок

Результат первого запуска тестов:

|  |  |  |
| --- | --- | --- |
| Класс | Succeed unit test | Total unit test |
| RoleServiceImpl | 10 | 12 |
| MenuServiceImpl | 5 | 5 |
| AuthServiceImpl | 9 | 9 |
| AdminServiceImpl | 9 | 9 |

Результат второго запуска тестов:

|  |  |  |
| --- | --- | --- |
| Класс | Succeed unit test | Total unit test |
| RoleServiceImpl | 10 | 10 |
| MenuServiceImpl | 5 | 5 |
| AuthServiceImpl | 9 | 9 |
| AdminServiceImpl | 9 | 9 |

На основании приведенного выше анализа мы решили убрать 2 теста метода getPageInfo из класса RoleServiceImpl. **Буду сделать это в рамках интеграционного теста, где мы можем работать с реальной базой данных.**

Остаточное количество ошибок: 0.

## 3.8 Вывод

В ходе выполнения данной лабораторной работы было проведено модульное тестирование ключевых компонентов системы, включая AdminServiceImpl, AuthServiceImpl, RoleServiceImpl, и MenuServiceImpl, с использованием фреймворка Mockito. Модульные тесты охватывали 432 строки исходного кода, что позволило выявить ряд потенциальных проблем и обеспечить более высокое качество разрабатываемого программного обеспечения.

Модульное тестирование продемонстрировало свою важность как критический этап в жизненном цикле разработки программного обеспечения, позволяя разработчикам проверить корректность работы отдельных компонентов системы в изоляции от внешних зависимостей. Это обеспечивает более высокую надежность и устойчивость кода, способствует раннему обнаружению и устранению ошибок, а также снижает затраты на последующие этапы разработки и поддержку продукта.

В процессе тестирования были выявлены различные типы ошибок, включая ошибки доступа к неинициализированным объектам (NullPointerException), ошибки в логике обработки данных и ошибки интеграции с внешними зависимостями. Для предотвращения подобных ошибок в будущем рекомендуется:

1. Тщательное планирование и проектирование интерфейсов и зависимостей между компонентами системы.
2. Использование mock-объектов для изоляции тестируемых компонентов и обеспечения контролируемого тестового окружения.
3. Проведение кодового ревью и рефакторинга для выявления и исправления потенциальных проблем в логике работы методов и обработке исключений.

Результаты проведенной работы подтверждают высокую эффективность модульного тестирования как инструмента повышения качества программного обеспечения и важность раннего внедрения тестирования в процесс разработки. Определенные в ходе тестирования проблемы были успешно устранены, что позволило улучшить стабильность и надежность разрабатываемой системы.

# Приложение 1

## Модульные тесты для класса AdminServiceImplTest

## Модульные тесты для класса AuthServiceImpl

## Модульные тесты для класса RoleServiceImpl

## Модульные тесты для класса MenuServiceImpl