

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
3.1 Выбранный фреймворк тестирования	3
3.1.1 Общее описание и функциональность фреймворка.....	3
3.1.2 Причины выбора данного фреймворка.....	3
3.2 Установка и настройка инструмента	3
3.2.1 Установка.....	3
3.2.2 Настройка инструмента.....	4
3.3 Стратегия тестирования	5
3.3.1 Общие подходы.....	5
3.3.2 Класс AdminServiceImplTest.....	5
3.3.3 Класс AuthServiceImpl	6
3.3.4 Класс RoleServiceImpl	7
3.3.5 Класс MenuServiceImpl.....	8
3.4 Написание и запуск тест-кейсов.....	9
3.4.1 Написание модульных тестов.....	9
3.4.2 Результат первого запуска тестов	10
3.5 Анализ результатов выполнения теста	11
3.6 Исправление кода и модульных тестов	12
3.6.1 Исправление кода	12
3.6.2 окончательный результат.....	13
3.7 Статистика покрытия кода и таблица ошибок.....	15
3.7.1 Статистика покрытия кода.....	15
3.7.2 Таблицы ошибок	15
3.8 Вывод.....	16
Приложение 1	16
Модульные тесты для класса AdminServiceImplTest	16
Модульные тесты для класса AuthServiceImpl	18
Модульные тесты для класса RoleServiceImpl.....	20
Модульные тесты для класса MenuServiceImpl.....	22

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 Mockito через Maven.

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

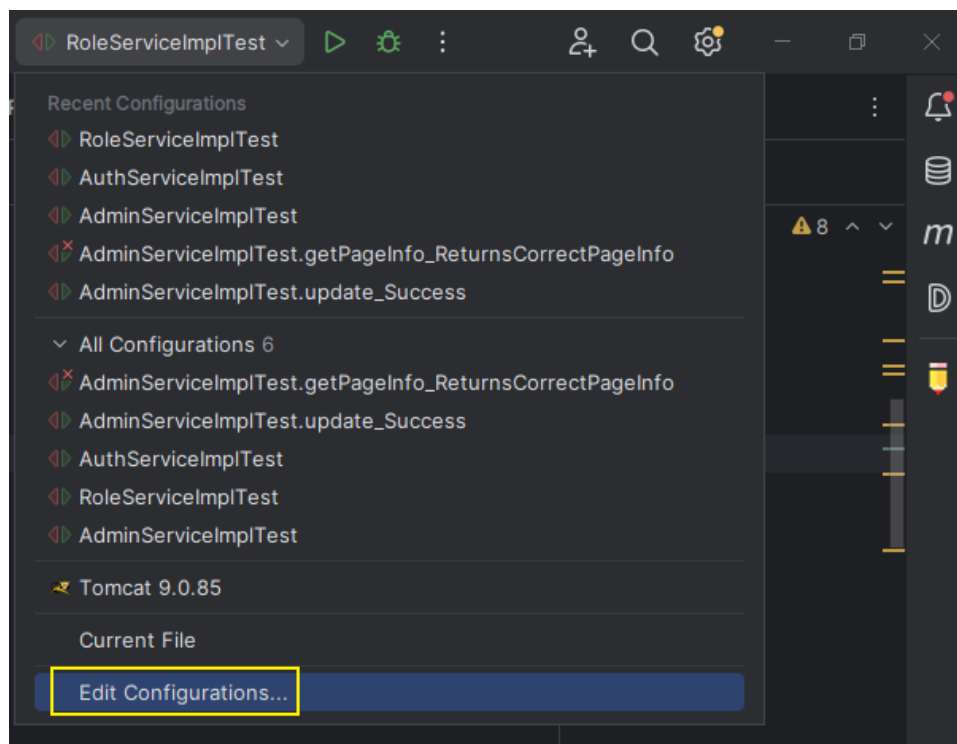
```

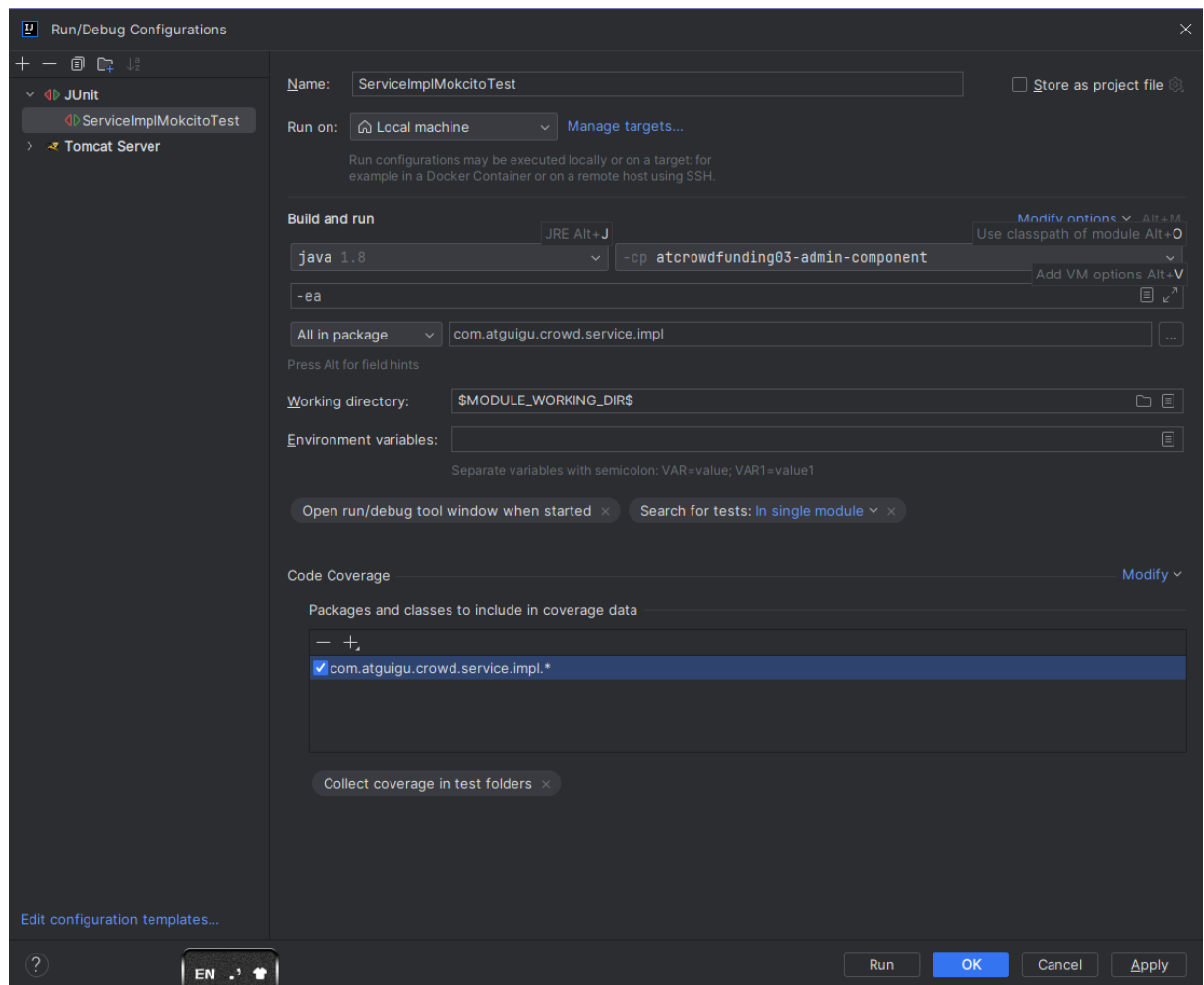
146      <!-- junit5 Test -->
147      <dependency>
148          <groupId>org.junit.jupiter</groupId>
149          <artifactId>junit-jupiter-api</artifactId>
150          <version>5.8.2</version>
151          <scope>test</scope>
152      </dependency>
153      <!-- Mockito Framework -->
154      <dependency>
155          <groupId>org.mockito</groupId>
156          <artifactId>mockito-core</artifactId>
157          <version>4.2.0</version>
158          <scope>test</scope>
159      </dependency>
160      <dependency>
161          <groupId>org.junit.jupiter</groupId>
162          <artifactId>junit-jupiter-engine</artifactId>
163          <version>5.8.2</version>
164          <scope>test</scope>
165      </dependency>

```

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.

3.3.2 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)`
 - Положительный тест: Проверка успешного сохранения администратора с корректными данными.
 - Отрицательный тест: Проверка обработки ситуации с дублированием ключа (`DuplicateKeyException`). Ожидается выброс исключения `LoginAcctAlreadyInUseException`.

2. getAll()

- Положительный тест: Проверка получения списка всех администраторов. Проверка сценария, когда список не пуст.

3. getAdminByLoginAcct(String loginAcct, String userPswd)

- Положительный тест: Проверка успешного получения администратора по корректным логину и паролю.
- 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: Проверка получения полного списка прав доступа. Особое внимание на проверку корректности возвращаемых данных.

- Отрицательный тест: Проверка поведения метода при отсутствии прав в базе данных. Ожидается пустой список.
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. `saveRoleAuthRelathinship(Map<String, List<Integer>> map)`
- Положительный тест:
 - Проверка успешного сохранения связей между ролью и набором прав. Валидация корректности обновления данных в базе.
 - Проверка сценария с пустым списком `authIdArray`, ожидается только удаление старых связей без добавления новых.
 - Отрицательные тесты:
 - Проверка с неправильными ключами в передаваемом `map`. Ожидается обработка ошибок или игнорирование неправильных ключей без влияния на процесс обновления.
 - Проверка поведения метода при передаче `null` в качестве значения `map`. Необходимо проверить устойчивость метода к таким ситуациям.

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)

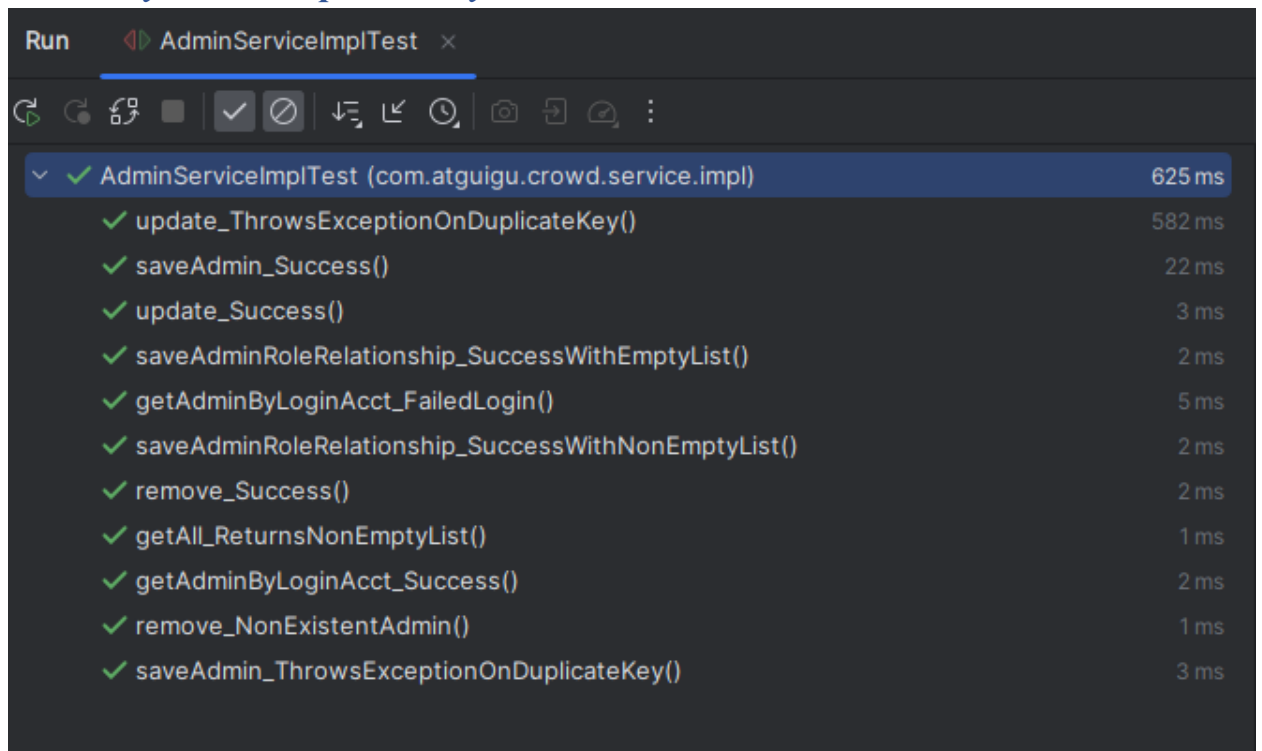
- Положительный тест: Проверить, что метод удаляет объект Menu по заданному ID.
- Отрицательный тест: Проверить, как метод обрабатывает ситуацию с попыткой удаления объекта по несуществующему ID. Также стоит проверить обработку ситуации, когда в метод передается null.

3.4 Написание и запуск тест-кейсов

3.4.1 Написание модульных тестов

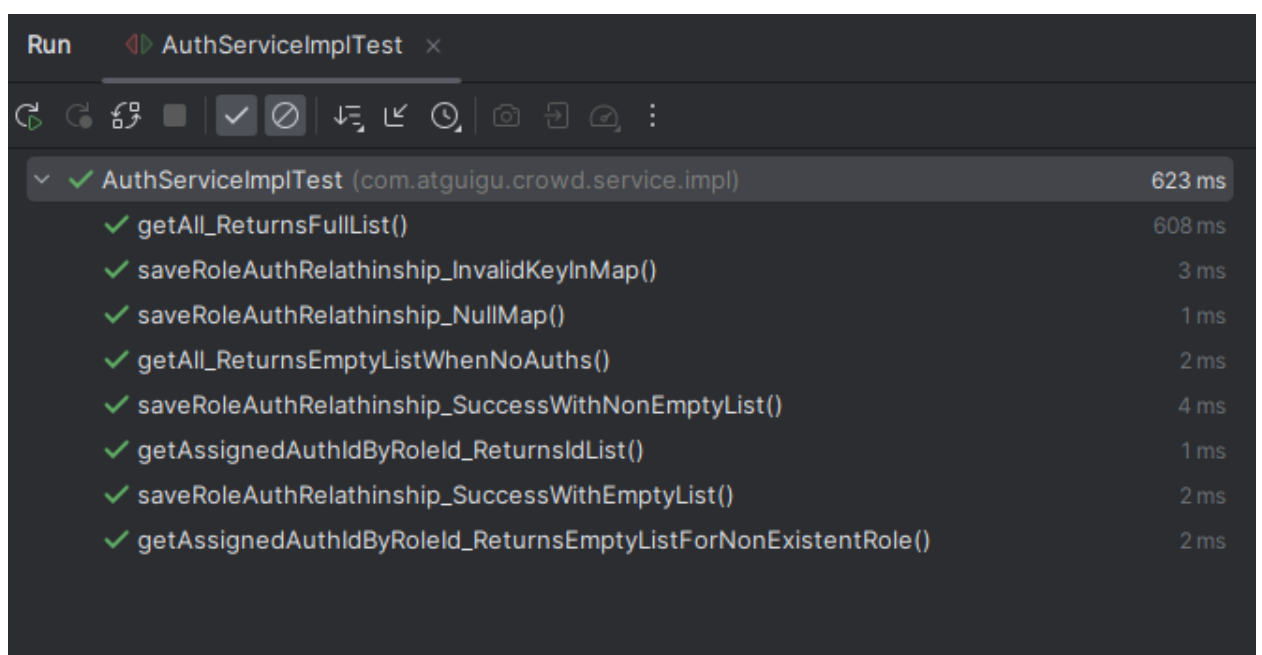
см. Приложение 1.

3.4.2 Результат первого запуска тестов



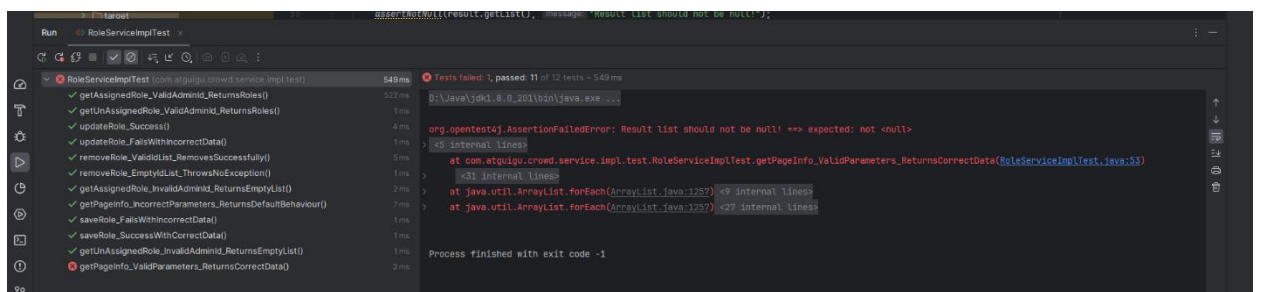
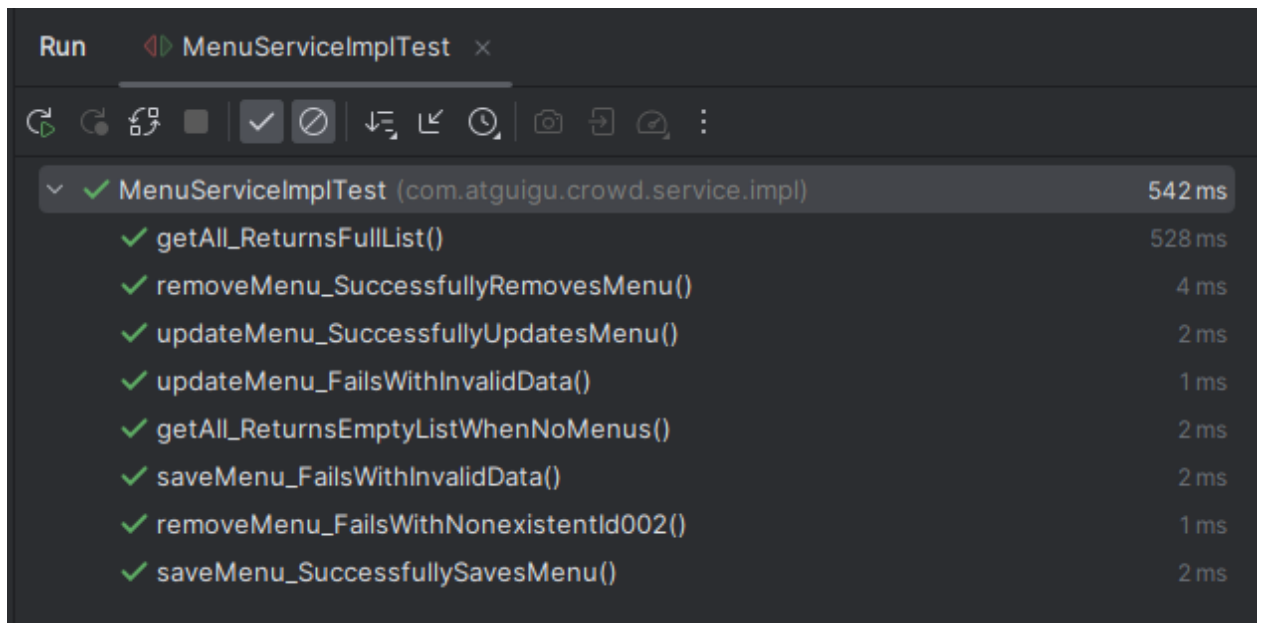
The screenshot shows the Run console in IntelliJ IDEA for the test class `AdminServiceImplTest` from the package `com.atguigu.crowd.service.impl`. The test suite passed with a total duration of 625 ms. The individual test methods and their durations are listed below:

Test Method	Duration (ms)
update_ThrowsExceptionOnDuplicateKey()	582
saveAdmin_Success()	22
update_Success()	3
saveAdminRoleRelationship_SuccessWithEmptyList()	2
getAdminByLoginAcct_FailedLogin()	5
saveAdminRoleRelationship_SuccessWithNonEmptyList()	2
remove_Success()	2
getAll_ReturnsNonEmptyList()	1
getAdminByLoginAcct_Success()	2
remove_NonExistentAdmin()	1
saveAdmin_ThrowsExceptionOnDuplicateKey()	3



The screenshot shows the Run console in IntelliJ IDEA for the test class `AuthServiceImplTest` from the package `com.atguigu.crowd.service.impl`. The test suite passed with a total duration of 623 ms. The individual test methods and their durations are listed below:

Test Method	Duration (ms)
getAll_ReturnsFullList()	608
saveRoleAuthRelathinship_InvalidKeyInMap()	3
saveRoleAuthRelathinship_NullMap()	1
getAll_ReturnsEmptyListWhenNoAuths()	2
saveRoleAuthRelathinship_SuccessWithNonEmptyList()	4
getAssignedAuthIdByRoleId_ReturnsIdList()	1
saveRoleAuthRelathinship_SuccessWithEmptyList()	2
getAssignedAuthIdByRoleId_ReturnsEmptyListForNonExistentRole()	2



3.5 Анализ результатов выполнения теста

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

Сценарий этого теста: введем 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.

```

42  @Test
43  void getPageInfo_ValidParameters_ReturnsCorrectData() {
44      List<Role> expectedRoles = Arrays.asList(new Role(), new Role());
45      when(roleMapper.selectRoleByKeyword(anyString())).thenReturn(expectedRoles);
46      PageInfo<Role> result = roleService.getPageInfo( pageNum: 1, pageSize: 10, keyword: "admin");
47      assertNotNull(result);
48      assertEquals( expected: 2, result.getList().size());
49  }

```

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

Начнем отладку:

```

40  @Test
41  void getPageInfo_ValidParameters_ReturnsCorrectData() {
42      List<Role> expectedRoles = Arrays.asList(new Role(), new Role()); expectedRoles: size = 2
43      when(roleMapper.selectRoleByKeyword(anyString())).thenReturn(expectedRoles); roleMapper: "roleMapper" expectedRoles: size = 2
44      PageInfo<Role> result = roleService.getPageInfo( pageNum: 1, pageSize: 10, keyword: "admin"); result: "PageInfo{pageNum=0, pageSize=0, size=
45      assertNotNull(result);
46      assertEquals( expected: 2, result.getList().size()); result: "PageInfo{pageNum=0, pageSize=0, size=0, startRow=0, endRow=0, total=0, pages=
47  }

```

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. Идеальной ситуацией было бы сделать это в рамках интеграционного теста, где мы можем работать с реальной базой данных.

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.

```

28 class RoleServiceImplTest {
    5 usages
29     @Mock
30     private RoleMapper roleMapper;
31
    10 usages
32     @InjectMocks
33     private RoleServiceImpl roleService;
34
    new *
35     @BeforeEach
36     > public void setup() { MockitoAnnotations.openMocks( testClass: this); }
37
38
39
40     // -----
41     // @Test
42     // void getPageInfo_ValidParameters_ReturnsCorrectData() {
43     //     List<Role> expectedRoles = Arrays.asList(new Role(), new Role());
44     //     when(roleMapper.selectRoleByKeyword(anyString())).thenReturn(expectedRoles);
45     //     PageInfo<Role> result = roleService.getPageInfo(1, 10, "admin");
46     //     assertNotNull(result.getList(), "Result list should not be null!");
47     //     assertEquals(2, result.getList().size());
48     // }
49
50     // @Test
51     // void getPageInfo_IncorrectParameters_ReturnsDefaultBehaviour() {
52     //     assertDoesNotThrow(() -> roleService.getPageInfo(-1, -10, "admin"));
53     // }
54
55     // -----

```

Выполнив модульный тест еще раз, получаем окончательный результат

✓ RoleServiceImplTest	48 ms
✓ getAssignedRole_ValidAdminId_ReturnsRoles()	32 ms
✓ getUnAssignedRole_ValidAdminId_ReturnsRoles()	1 ms
✓ updateRole_Success()	2 ms
✓ updateRole_FailsWithIncorrectData()	1 ms
✓ removeRole_ValidIdList_RemovesSuccessfully()	5 ms
✓ removeRole_EmptyIdList_ThrowsNoException()	1 ms
✓ getAssignedRole_InvalidAdminId_ReturnsEmptyList()	2 ms
✓ saveRole_FailsWithIncorrectData()	1 ms
✓ saveRole_SuccessWithCorrectData()	2 ms
✓ getUnAssignedRole_InvalidAdminId_ReturnsEmptyList()	1 ms

3.6.2 окончательный результат

✓ test (com.atguigu.crowd.service.impl)	1 sec 280 ms
✓ AuthServiceImplTest	987 ms
✓ getAll_ReturnsFullList()	966 ms
✓ saveRoleAuthRelathinship_InvalidKeyInMap()	5 ms
✓ saveRoleAuthRelathinship_NullMap()	2 ms
✓ getAll_ReturnsEmptyListWhenNoAuths()	3 ms
✓ saveRoleAuthRelathinship_SuccessWithNonEmptyList()	6 ms
✓ getAssignedAuthIdByRoleId_ReturnsIdList()	2 ms
✓ saveRoleAuthRelathinship_SuccessWithEmptyList()	1 ms
✓ getAssignedAuthIdByRoleId_ReturnsEmptyListForNonExistentRole()	2 ms
✓ AdminServiceImplTest	172 ms
✓ update_ThrowsExceptionOnDuplicateKey()	127 ms
✓ saveAdmin_Success()	22 ms
✓ update_Success()	2 ms
✓ saveAdminRoleRelationship_SuccessWithEmptyList()	2 ms
✓ getAdminByLoginAcct_FailedLogin()	7 ms
✓ saveAdminRoleRelationship_SuccessWithNonEmptyList()	2 ms
✓ remove_Success()	1 ms
✓ getAll_ReturnsNonEmptyList()	1 ms
✓ getAdminByLoginAcct_Success()	2 ms
✓ remove_NonExistentAdmin()	2 ms
✓ saveAdmin_ThrowsExceptionOnDuplicateKey()	4 ms
✓ MenuServiceImplTest	73 ms
✓ getAll_ReturnsFullList()	28 ms
✓ removeMenu_SuccessfullyRemovesMenu()	37 ms
✓ updateMenu_SuccessfullyUpdatesMenu()	2 ms
✓ updateMenu_FailsWithInvalidData()	2 ms
✓ getAll_ReturnsEmptyListWhenNoMenus()	1 ms
✓ saveMenu_FailsWithInvalidData()	1 ms
✓ removeMenu_FailsWithNonexistentId002()	1 ms
✓ saveMenu_SuccessfullySavesMenu()	1 ms
✓ RoleServiceImplTest	48 ms
✓ getAssignedRole_ValidAdminId_ReturnsRoles()	32 ms
✓ getUnAssignedRole_ValidAdminId_ReturnsRoles()	1 ms
✓ updateRole_Success()	2 ms
✓ updateRole_FailsWithIncorrectData()	1 ms

✓ RoleServiceImplTest	48 ms
✓ getAssignedRole_ValidAdminId_ReturnsRoles()	32 ms
✓ getUnAssignedRole_ValidAdminId_ReturnsRoles()	1 ms
✓ updateRole_Success()	2 ms
✓ updateRole_FailsWithIncorrectData()	1 ms
✓ removeRole_ValidIdList_RemovesSuccessfully()	5 ms
✓ removeRole_EmptyIdList_ThrowsNoException()	1 ms
✓ getAssignedRole_InvalidAdminId_ReturnsEmptyList()	2 ms
✓ saveRole_FailsWithIncorrectData()	1 ms
✓ saveRole_SuccessWithCorrectData()	2 ms
✓ getUnAssignedRole_InvalidAdminId_ReturnsEmptyList()	1 ms

3.7 Статистика покрытия кода и таблица ошибок

3.7.1 Статистика покрытия кода

Coverage				
MenuServiceImplTest x ServiceImplMokcitoTest x				
Element	Class, %	Method, %	Line, %	Branch, %
com.atguigu.crowd.service.im	100% (4/4)	85% (18/21)	86% (64/74)	63% (14/22)
RoleServiceImpl	100% (1/1)	83% (5/6)	75% (9/12)	100% (0/0)
MenuServiceImpl	100% (1/1)	100% (4/4)	100% (5/5)	100% (0/0)
AuthServiceImpl	100% (1/1)	100% (3/3)	100% (9/9)	75% (3/4)
AdminServiceImpl	100% (1/1)	75% (6/8)	85% (41/48)	61% (11/18)

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	12
MenuServiceImpl	5	5
AuthServiceImpl	9	9
AdminServiceImpl	9	9

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`

```
public class AdminServiceImplTest {
    @Mock
    private AdminMapper adminMapper;

    @InjectMocks
    private AdminServiceImpl adminService;
```



```

@BeforeEach
public void setup() {
    MockitoAnnotations.openMocks(this);
}

@Test
public void saveAdmin_Success() {
    Admin admin = new Admin(null, "testUser", "password", "Test User",
"test@example.com", null);
    when(adminMapper.insert(admin)).thenReturn(1);
    adminService.saveAdmin(admin);
    verify(adminMapper).insert(admin);
}

@Test
public void saveAdmin_ThrowsExceptionOnDuplicateKey() {
    Admin admin = new Admin(null, "testUser", "password", "Test User",
"test@example.com", null);
    doThrow(new DuplicateKeyException("Duplicate
Key")).when(adminMapper).insert(any(Admin.class));
    assertThrows(LoginAcctAlreadyInUseException.class, () ->
adminService.saveAdmin(admin));
}

// -----
@Test
public void getAll_ReturnsNonEmptyList() {
    List<Admin> expectedList = Arrays.asList(new Admin());
    when(adminMapper.selectByExample(any())).thenReturn(expectedList);
    List<Admin> resultList = adminService.getAll();
    assertFalse(resultList.isEmpty());
}

// -----
@Test
public void getAdminByLoginAcct_Success() {
    String loginAcct = "testUser";
    String originPwd = "password";
    String afterPwd = CrowdUtil.md5(originPwd);
    when(adminMapper.selectByExample(any())).thenReturn(
Collections.singletonList(new Admin(1, loginAcct, afterPwd, "Test
User", "test@example.com", null)));
    assertDoesNotThrow(() -> adminService.getAdminByLoginAcct(loginAcct,
originPwd));
}

@Test
public void getAdminByLoginAcct_FailedLogin() {
    when(adminMapper.selectByExample(any())).thenReturn(Collections.emptyList());
    assertThrows(LoginFailedException.class, () ->
adminService.getAdminByLoginAcct("wrongUser", "password"));
}

// -----
@Test
public void remove_Success() {
    when(adminMapper.deleteByPrimaryKey(anyInt())).thenReturn(1);
    assertDoesNotThrow(() -> adminService.remove(1));
}

@Test
public void remove_NonExistentAdmin() {

```

```

        doThrow(new RuntimeException("Admin not
found")).when(adminMapper).deleteByPrimaryKey(anyInt());
        // Так как метод не предполагает обработку исключений в случае
отсутствия администратора, тестовый кейс
        // адаптирован под общий подход к исключениям
        assertThrows(RuntimeException.class, () -> adminService.remove(-1));
    }

    // -----
    @Test
    public void update_Success() {
when(adminMapper.updateByPrimaryKeySelective(any(Admin.class))).thenReturn(1)
;
        assertDoesNotThrow(() -> adminService.update(new Admin()));
    }

    @Test
    public void update_ThrowsExceptionOnDuplicateKey() {
        doThrow(new DuplicateKeyException("Duplicate Key")).when(adminMapper)
            .updateByPrimaryKeySelective(any(Admin.class));
        assertThrows(LoginAcctAlreadyInUseForUpdateException.class, () ->
adminService.update(new Admin()));
    }

    // -----
    @Test
    public void saveAdminRoleRelationship_SuccessWithNonEmptyList() {
        doNothing().when(adminMapper).deleteOldRelationship(anyInt());
        doNothing().when(adminMapper).insertNewRelationship(anyInt(),
anyList());
        assertDoesNotThrow(() -> adminService.saveAdminRoleRelationship(1,
Arrays.asList(1, 2, 3)));
    }

    @Test
    public void saveAdminRoleRelationship_SuccessWithEmptyList() {
        doNothing().when(adminMapper).deleteOldRelationship(anyInt());
        // Поскольку в методе нет явной проверки на пустой список для вставки
новых связей, предполагается, что операция
        // просто не выполняется, не вызывая ошибок.
        assertDoesNotThrow(() -> adminService.saveAdminRoleRelationship(1,
new ArrayList<>()));
    }

    // -----
}

```

Модульные тесты для класса AuthServiceImpl

```

class AuthServiceImplTest {
    @Mock
    private AuthMapper authMapper;

    @InjectMocks
    private AuthServiceImpl authService;

    @BeforeEach
    public void setup() {
        MockitoAnnotations.openMocks(this);
    }
}

```

```

// -----
@Test
public void getAll_ReturnsFullList() {

when(authMapper.selectByExample(any())).thenReturn(Collections.singletonList(
new Auth()));
    List<Auth> result = authService.getAll();
    assertNotNull(result);
    assertFalse(result.isEmpty());
}

@Test
public void getAll_ReturnsEmptyListWhenNoAuths() {

when(authMapper.selectByExample(any())).thenReturn(Collections.emptyList());
    List<Auth> result = authService.getAll();
    assertNotNull(result);
    assertTrue(result.isEmpty());
}

// -----
@Test
public void getAssignedAuthIdByRoleId_ReturnsIdList() {

when(authMapper.selectAssignedAuthIdByRoleId(anyInt())).thenReturn(Collection
s.singletonList(1));
    List<Integer> result = authService.getAssignedAuthIdByRoleId(1);
    assertNotNull(result);
    assertFalse(result.isEmpty());
}

@Test
public void
getAssignedAuthIdByRoleId_ReturnsEmptyListForNonExistentRole() {

when(authMapper.selectAssignedAuthIdByRoleId(anyInt())).thenReturn(Collection
s.emptyList());
    List<Integer> result = authService.getAssignedAuthIdByRoleId(-1);
    assertNotNull(result);
    assertTrue(result.isEmpty());
}

// -----
@Test
public void saveRoleAuthRelathinship_SuccessWithNonEmptyList() {
    Map<String, List<Integer>> map = new HashMap<>();
    map.put("roleId", Collections.singletonList(1));
    map.put("authIdArray", Collections.singletonList(1));

    doNothing().when(authMapper).deleteOldRelationship(anyInt());
    doNothing().when(authMapper).insertNewRelationship(anyInt(),
anyList());

    assertDoesNotThrow(() -> authService.saveRoleAuthRelathinship(map));
}

@Test
public void saveRoleAuthRelathinship_SuccessWithEmptyList() {
    Map<String, List<Integer>> map = new HashMap<>();
    map.put("roleId", Collections.singletonList(1));
    map.put("authIdArray", Collections.emptyList());

    doNothing().when(authMapper).deleteOldRelationship(anyInt());

```

```

        assertDoesNotThrow(() -> authService.saveRoleAuthRelathinship(map));
    }

    @Test
    public void saveRoleAuthRelathinship_InvalidKeyInMap() {
        Map<String, List<Integer>> map = new HashMap<>();
        map.put("wrongKey", Collections.singletonList(1));

        // Настройка и ожидание не требуется, так как метод не будет вызван
        // из-за неправильного ключа

        assertThrows(Exception.class, () ->
            authService.saveRoleAuthRelathinship(map));
    }

    @Test
    public void saveRoleAuthRelathinship_NullMap() {
        assertThrows(NullPointerException.class, () ->
            authService.saveRoleAuthRelathinship(null));
    }
}

```

Модульные тесты для класса RoleServiceImpl

```

class RoleServiceImplTest {
    @Mock
    private RoleMapper roleMapper;

    @InjectMocks
    private RoleServiceImpl roleService;

    @BeforeEach
    public void setup() {
        MockitoAnnotations.openMocks(this);
    }

    // -----
    @Test
    void getPageInfo_ValidParameters_ReturnsCorrectData() {
        List<Role> expectedRoles = Arrays.asList(new Role(), new Role());
        when(roleMapper.selectRoleByKeyword(anyString())).thenReturn(expectedRoles);
        PageInfo<Role> result = roleService.getPageInfo(1, 10, "admin");
        assertNotNull(result.getList(), "Result list should not be null!");
        assertEquals(2, result.getList().size());
    }

    @Test
    void getPageInfo_IncorrectParameters_ReturnsDefaultBehaviour() {
        assertDoesNotThrow(() -> roleService.getPageInfo(-1, -10, "admin"));
    }

    // -----
    @Test
    public void saveRole_SuccessWithCorrectData() {
        Role role = new Role(null, "newRole");
    }
}

```

```

        assertDoesNotThrow(() -> roleService.saveRole(role));
    }

    @Test
    public void saveRole_FailsWithIncorrectData() {
        assertDoesNotThrow(() -> roleService.saveRole(null)); // Допуская,
        что обработка исключений происходит на уровне
        // mapper
    }

    // -----
    @Test
    public void updateRole_Success() {
        Role role = new Role(1, "updatedRole");
        assertDoesNotThrow(() -> roleService.updateRole(role));
    }

    @Test
    public void updateRole_FailsWithIncorrectData() {
        Role role = new Role(null, null); // Предполагая некорректность
        данных
        assertDoesNotThrow(() -> roleService.updateRole(role)); // Обработка
        ошибок не описана, предполагаем стандартное
        // поведение
    }

    // -----
    @Test
    void removeRole_ValidIdList_RemovesSuccessfully() {
        // doNothing().when(roleMapper).deleteByExample(any());
        when(roleMapper.deleteByExample(any())).thenReturn(1);
        assertDoesNotThrow(() -> roleService.removeRole(Arrays.asList(1, 2, 3)));
    }

    @Test
    void removeRole_EmptyIdList_ThrowsNoException() {
        assertDoesNotThrow(() -> roleService.removeRole(new ArrayList<>()));
    }

    // -----
    @Test
    void getAssignedRole_ValidAdminId_ReturnsRoles() {
        when(roleMapper.selectAssignedRole(anyInt())).thenReturn(Arrays.asList(new
        Role(), new Role()));
        List<Role> result = roleService.getAssignedRole(1);
        assertNotNull(result);
        assertEquals(2, result.size());
    }

    @Test
    void getUnAssignedRole_ValidAdminId_ReturnsRoles() {
        when(roleMapper.selectUnAssignedRole(anyInt())).thenReturn(Arrays.asList(new
        Role(), new Role()));
        List<Role> result = roleService.getUnAssignedRole(1);
        assertNotNull(result);
        assertEquals(2, result.size());
    }

    @Test
    void getAssignedRole_InvalidAdminId_ReturnsEmptyList() {
        when(roleMapper.selectAssignedRole(anyInt())).thenReturn(new
        ArrayList<>());
        List<Role> result = roleService.getAssignedRole(-1);
    }

```

```

        assertTrue(result.isEmpty());
    }

    @Test
    void getUnAssignedRole_InvalidAdminId_ReturnsEmptyList() {
        when(roleMapper.selectUnAssignedRole(anyInt())).thenReturn(new
        ArrayList<>());
        List<Role> result = roleService.getUnAssignedRole(-1);
        assertTrue(result.isEmpty());
    }
}

```

Модульные тесты для класса MenuServiceImpl

```

class MenuServiceImplTest {

    @Mock
    private MenuMapper menuMapper;

    @InjectMocks
    private MenuServiceImpl menuService;

    @BeforeEach
    void setUp() {
        MockitoAnnotations.openMocks(this);
    }

    // -----
    @Test
    void getAll_ReturnsFullList() {
        when(menuMapper.selectByExample(any())).thenReturn(Collections.singletonList(
        new Menu()));
        List<Menu> result = menuService.getAll();
        assertFalse(result.isEmpty(), "The result should not be empty when
        menus exist");
    }

    @Test
    void getAll_ReturnsEmptyListWhenNoMenus() {
        when(menuMapper.selectByExample(any())).thenReturn(Collections.emptyList());
        List<Menu> result = menuService.getAll();
        assertTrue(result.isEmpty(), "The result should be empty when no
        menus exist");
    }

    // -----
    @Test
    void saveMenu_SuccessfullySavesMenu() {
        Menu menu = new Menu();
        menu.setName("New Menu");
        // doNothing().when(menuMapper).insert(any(Menu.class));
        when(menuMapper.insert(any(Menu.class))).thenReturn(1);
        assertDoesNotThrow(() -> menuService.saveMenu(menu));
    }

    @Test
    void saveMenu_FailsWithInvalidData() {
        assertDoesNotThrow(() -> menuService.saveMenu(null), "Saving null
        should not throw an exception");
    }
}

```

```

    }

    // -----
    @Test
    void updateMenu_SuccessfullyUpdatesMenu() {
        Menu menu = new Menu();
        menu.setId(1);
        menu.setName("Updated Menu");

        when(menuMapper.updateByPrimaryKeySelective(any(Menu.class))).thenReturn(1);
        assertDoesNotThrow(() -> menuService.updateMenu(menu));
    }

    @Test
    void updateMenu_FailsWithInvalidData() {
        Menu menu = new Menu(); // Например, без установленного ID
        assertDoesNotThrow(() -> menuService.updateMenu(menu),
            "Updating menu without ID should not throw an exception");
    }

    // -----
    @Test
    void removeMenu_SuccessfullyRemovesMenu() {

        when(menuMapper.deleteByPrimaryKey(anyInt())).thenReturn(1);
        assertDoesNotThrow(() -> menuService.removeMenu(1));
    }

    @Test
    void removeMenu_FailsWithNonexistentId002() {
        assertDoesNotThrow(() -> menuService.removeMenu(-1),
            "Removing menu with nonexistent ID should not throw an
exception");
    }
}

```