МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

Отчет о лабораторной работе по предмету «Тестирование програмного обеспечение»

Выполнил студент
Группы P3318
Курников Д. Н.

Санкт-Петербург,
2019

Цель работы:
Необходимо разработать консольное приложение в соответствие с методологий TDD.

Задачи:
Разработка консольного приложения “Словарь”.

Программа представляет собой словарь однокоренных слов.

Запустив программу, пользователь может вводить слова. Если введенное слово известно программе, она выводит список однокоренных слов в порядке увеличения количества словообразующих частей. При этом выводимые слова разбиваются тире в соответствии с составом.

В случае, когда введенное слово программе неизвестно, пользователю задается вопрос, не хочет ли он добавить новое слово в словарь. Если да, то пользователь последовательно вводит в программу предкоренные части слова, затем пустую строку, корень и посткоренные части. Объединив введенные части слова, программа проверяет по изначально введенному слову, что все части введены правильно, и возвращается к ожиданию следующего слова.

Функциональные требования:
1 - добавление слова в базу
2 - два и более префиксов
3 - два и более постфиксов
4 - удаление слова из базы
5 - вывод всех слов находящихся в базе
6 - поиск слов по корню
7- создание нового пользователя
8 - удаление пользователя
9 - Вывод всех слов которые создал пользователь
10 - добавление роли администратора
11 - бан пользователя
12 - разбан пользователя
13 - забаненый пользователь не может добавлять слова
14 - вывод слов в порядке возрастания словообразующих частей

15 - поиск пользователя по имени
16 - Вывод всех пользователей
17 - проверка идентичности имени пользователя
18 - logout
19 - авторизация пользователя
20 - Валидация пользовательских пароля и логина

TDD

TDD или test-driven development – разработка через тестирование. Методика разработки через тестирование заключается в организации автоматического тестирования разрабатываемых приложений путем написания модульных, интеграционных и функциональных тестов, определяющих требования к коду непосредственно перед написанием этого самого кода. Сначала пишется тест, который проверяет корректность работы еще не написанного программного кода. Этот тест, разумеется, не проходит. После этого разработчик пишет код, который выполняет действия, требуемые для прохождения теста. После того, как тест успешно пройден, по необходимости осуществляется рефакторинг написанного кода, причём рефакторинг осуществляется под контролем прохождения тестов.

Эта методология позволяет добиться создания пригодного для автоматического тестирования приложения и очень хорошего покрытия кода тестами, так как ТЗ переводится на язык автоматических тестов, то есть всё, что программа должна делать, проверяется. Также TDD часто упрощает программную реализацию: исключается избыточность реализации — если компонент проходит тест, то он считается готовым. Архитектура программных продуктов, разрабатываемых таким образом, обычно лучше (в приложениях, которые пригодны для автоматического тестирования, обычно очень хорошо распределяется ответственность между компонентами, а выполняемые сложные процедуры декомпозированы на множество простых). Стабильность работы приложения, разработанного через тестирование, выше за счёт того, что все основные функциональные возможности программы покрыты тестами и их работоспособность постоянно проверяется. Сопровождаемость

проектов, где тестируется всё или практически всё, очень высока — разработчики могут не бояться вносить изменения в код, если что-то пойдёт не так, то об этом сообщат результаты автоматического тестирования

Преимущества

- меньше используется отладчик при разработке
- Тесты заставляют делать свой код более приспособленным для тестирования.
- Тесты защищают от ошибок.
- Разработка через тестирование способствует более модульному, гибкому и расширяемому коду.

Листинг

DictionaryServiseTest

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
internal class DictionaryServiseTest {
    private lateinit var wordDAO: WordDAO
    private  var dictionaryServise: DictionaryServise = DictionaryServise()
    private  var authService: AuthServise = AuthServise()

    @BeforeAll
    fun  init(){
        wordDAO = WordDAO(sessionOf("jdbc:postgresql:tpo", "postgres",
"santa666"))
        authService.setSessionDB(wordDAO.getSession())
        dictionaryServise.setWodrDAO(wordDAO.getSession())
        dictionaryServise.setAuthService(authService)
        authService.login("santa", "santa".md5())
    }
    private fun String.md5(): String {
        val md = MessageDigest.getInstance("MD5")
```

```
        return BigInteger(1, md.digest(toByteArray())).toString(16).padStart(32,
'0')
    }
 @Test
    fun getWordsByRootName() {
        val rootName = "ход"
        val words = dictionaryServise.getWordsByRootName(rootName)
        val roots = wordDAO.getRootsByName("ход")
        roots.forEach {
            val prefs = wordDAO.getPrefByRootID(it.id)
            val posts = wordDAO.getPostByRootId(it.id)
            assert(it.name == rootName)
            assert(prefs.size == words.find { word -> it.id == word.root.id
}?.prefs?.size)
            assert(posts.size == words.find { word -> it.id == word.root.id
}?.posts?.size)
        }
    }

    @Test
    fun deleteWordById() {
        val rootId = wordDAO.insertRoot("чит", authService.getUser()?.id)
        wordDAO.insertPref("про", rootId)
        wordDAO.insertPost("ать", rootId)
        dictionaryServise.deleteWordById(rootId)
        assert(wordDAO.getRootsByName("чит").find { it.id == rootId} == null)
    }

    @Test
    fun addWord() {
        val root = "test"
        val pref = listOf<String>("pre", "p")
        val post = listOf<String>("post", "st")
        dictionaryServise.addWord(pref, root, post)
        assert(wordDAO.getRootsByName(root).isNotEmpty())
```

```
dictionaryServise.deleteWordById(wordDAO.getRootsByName(root).first().id)
    }

    @Test
    fun getUserWords() {
        authService.login("admin", "admin".md5())
        val words = dictionaryServise.getUserWords()
        val roots = wordDAO.getRootsByUserId(authService.getUser()!!.id)

        roots.forEach{
            val prefs = wordDAO.getPrefByRootID(it.id)
            val posts = wordDAO.getPostByRootId(it.id)
            assertAll(
                Executable {
            assert(words.find { word -> word.root.name == it.name } != null)},
                Executable {
                assert(prefs.size == words.find { word -> it.id == word.root.id
}?.prefs?.size)},
                Executable {
                assert(posts.size == words.find { word -> it.id == word.root.id
}?.posts?.size)}
            )
        }
    }

    @Test
    fun getAdminWords() {
        authService.login("santa", "santa".md5())
        val words = dictionaryServise.getUserWords()
        val roots = wordDAO.getRootsByUserId(authService.getUser()!!.id)

        roots.forEach{
            val prefs = wordDAO.getPrefByRootID(it.id)
            val posts = wordDAO.getPostByRootId(it.id)
```

```
            assertAll(
                Executable {
                    assert(words.find { word -> word.root.name == it.name } != null)},
                Executable {
                    assert(prefs.size == words.find { word -> it.id == word.root.id
}?.prefs?.size)},
                Executable {
                    assert(posts.size == words.find { word -> it.id == word.root.id
}?.posts?.size)}
            )
        }
    }
    @Test
    fun getAllWordsForAdmin() {
        authService.login("santa", "santa".md5())
        val words = dictionaryServise.getAllWords()
        val roots = wordDAO.getAllRoots()

        roots.forEach{
            val prefs = wordDAO.getPrefByRootID(it.id)
            val posts = wordDAO.getPostByRootId(it.id)
            assert(words.find { word -> word.root.name == it.name } != null)
            assert(prefs.size == words.find { word -> it.id == word.root.id
}?.prefs?.size)
            assert(posts.size == words.find { word -> it.id == word.root.id
}?.posts?.size)
        }
    }

    @Test
    fun getAllWordsForUser() {
        authService.login("admin", "admin".md5())
        val words = dictionaryServise.getAllWords()
        val roots = wordDAO.getAllRoots()
```

```
        roots.forEach{
            val prefs = wordDAO.getPrefByRootID(it.id)
            val posts = wordDAO.getPostByRootId(it.id)
            assert(words.find { word -> word.root.name == it.name } != null)
            assert(prefs.size == words.find { word -> it.id == word.root.id
}?.prefs?.size)
            assert(posts.size == words.find { word -> it.id == word.root.id
}?.posts?.size)
        }
    }

    @Test
    fun checkSomePrefs(){
        authService.login("santa", "santa".md5())
        val prefs = listOf<String>("first", "second", "third")
        val root = "prefs"
        dictionaryServise.addWord(prefs, root, List<String>(0){""})
        val id = wordDAO.getRootsByName("prefs").first().id
        assert(wordDAO.getPrefByRootID(id).size == prefs.size)
        wordDAO.deleteWordById(id)
    }

    @Test
    fun checkSomePosts(){
        val posts = listOf<String>("first", "second", "third")
        val root = "posts"
        dictionaryServise.addWord(posts, root, List<String>(0){""})
        val id = wordDAO.getRootsByName("posts").first().id
        assert(wordDAO.getPrefByRootID(id).size == posts.size)
        wordDAO.deleteWordById(id)
    }

    @Test
    fun checkAddWordByBanUser(){
        authService.login("banUser", "password".md5())
```

```
        try{
            dictionaryServise.addWord(listOf("tr"), "root", listOf("rtr"))
        }catch  (e: WordExeption){
            assert(e.message == "Ban user could not add word!")
        }
    }

    @Test
    fun checkSorting(){
        val words = dictionaryServise.getAllWords()
        assert(words.first().count < words[1].count)
    }
}
```

AuthServiseTest

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
internal class AuthServiseTest {
    private lateinit var userDAO: UserDAO
    private  var authService: AuthServise = AuthServise()

    private fun String.md5(): String {
        val md = MessageDigest.getInstance("MD5")
        return BigInteger(1, md.digest(toByteArray())).toString(16).padStart(32,
'0')
    }

    @BeforeAll
     fun  init(){
        userDAO = UserDAO(sessionOf("jdbc:postgresql:tpo", "postgres",
"santa666"))
        authService.setSessionDB(userDAO.getSession())
    }


    @Test
```

```
fun login() {
    authService.login("admin", "admin".md5())
    assert(authService.getUser()?.id == 6)
}

@Test
fun grand() {
    authService.login("santa", "santa".md5())
    authService.grand("admin")
    assert(authService.getUserByName("admin")?.permit == 1)
    authService.grand("admin")
}

@Test
fun testGrantToYourself(){
    authService.login("santa", "santa".md5())
    try {
        authService.grand("santa")
    }catch (e: UserException){
        assert(e.message == "You could not ban yourself")
    }
}

@Test
fun testUnexeptebleUserName(){
    authService.login("santa", "santa".md5())
    try {
        authService.grand("unxept")
    }catch (e: UserException){
        assert(e.message == "User is not exist")
    }

}

@Test
```

```
fun resrAnAdminUserGrant(){
    authService.login("admin", "admin".md5())
    try {
        authService.grand("commonUser")
    }catch (e: UserException){
        assert(e.message == "User is not administrator!")
    }
}

@Test
fun unban() {
    authService.login("santa", "santa".md5())
    userDAO.insert("userForBan", "banme".md5())
    userDAO.setPermitToUserByUserName("userForBan", 2)
    authService.unban("userForBan")
    assert(authService.getUserByName("userForBan")?.permit == 0)
    userDAO.deleteAccount(userDAO.getIdByUserName("userForBan"))
}

@Test
fun checkPermitforUnbun(){
    authService.login("admin", "admin".md5())
    userDAO.insert("userForBan", "banme".md5())
    userDAO.setPermitToUserByUserName("userForBan", 2)
    try {
        authService.unban("userForBan")
    }catch (e: UserException){
        assert(e.message == "User is not administrator!")
        userDAO.deleteAccount(userDAO.getIdByUserName("userForBan"))
    }
}

@Test
fun chackBanYourself(){
    authService.login("santa", "santa".md5())
```

```
    try {
        authService.ban("santa")
    }catch (e: UserException){
        assert(e.message == "You could not ban yourself")
    }
}

@Test
fun checkBanAdministartor(){
    authService.login("santa1", "santa1".md5())
    try {
        authService.ban("santa")
    }catch (e: UserException){
        assert(e.message == "You could ban administrator")
    }
}

@Test
fun checkPermitforbun(){
    authService.login("admin", "admin".md5())
    userDAO.insert("userForBan", "banme".md5())
    userDAO.setPermitToUserByUserName("userForBan", 2)
    try {
        authService.ban("userForBan")
    }catch (e: UserException){
        assert(e.message == "User is not administrator!")
        userDAO.deleteAccount(userDAO.getIdByUserName("userForBan"))
    }
}

@Test
fun ban() {
    authService.login("santa", "santa".md5())
    userDAO.insert("userForBan", "banme".md5())
    authService.ban("userForBan")
```

```
    assert(authService.getUserByName("userForBan")?.permit == 2)
    userDAO.deleteAccount(userDAO.getIdByUserName("userForBan"))
}

@Test
fun viewAllUsers() {
    val usersD: List<User> = userDAO.getAllUsers()
    val users: List<User> = authService.viewAllUsers()
    assert(users.size == usersD.size)
}

@Test()
fun checkGetAllUserForCommonUser(){
    authService.login("admin", "admin".md5())
    try{
        val users: List<User> = authService.viewAllUsers()
    }catch (e: UserException){
        equals(e.message == "User is not administrator!")
    }
}

@Test
fun viewUserByName() {
    val username = "admin"
    val userD: User? = userDAO.getUserByName(username)
    val user = authService.getUserByName(username)
    if (userD != null) {
        if (user != null) {
            assert(userD.id == user.id)
        }
    }
    assert(user != null)
}

@Test
```

```
fun deleteAccount() {
    userDAO.insert("userForDelete", "deleteme".md5())
    authService.login("userForDelete", "deleteme".md5())
    authService.deleteAccount()
    assert(!authService.isLogind())
}

@Test
fun logout() {
    authService.login("admin", "admin".md5())
    authService.logout()
    assert(!authService.isLogind())
}

@Test
fun registration() {
    authService.registration("newUser", "pass".md5())
    assert(userDAO.find("newUser", "pass".md5()) != null)
    userDAO.deleteAccount(userDAO.getIdByUserName("newUser"))
}

@Test
fun chechRegistretedUser(){
    try {
        authService.registration("admin", "admin".md5())
    }catch (e: UserException){
        assert(e.message == "User already exist!")
    }
}

@Test
fun getUsernameById() {
    val user = authService.getUsernameById(6)
    val userD = userDAO.getUserById(6)
    assert(user == userD)
```

```
    }

    @Test
    fun checkUserLogin() {
        authService.login("commonUser", "pass".md5())
        assert(authService.isLogind())

    }
    @Test
    fun checkAdmin() {
        authService.login("santa", "santa".md5())
        assert(authService.isAdmin())
    }

    @Test
    fun incorrectLogin(){
        try {
            authService.login("blabla", "bla".md5())
        }catch (e: UserException) {
            assert(e.message == "Incorrect userName or Password!")
        }

    }
}
```

main

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
internal class AuthServiseTest {
    private lateinit var userDAO: UserDAO
    private  var authService: AuthServise = AuthServise()

    private fun String.md5(): String {
        val md = MessageDigest.getInstance("MD5")
        return BigInteger(1, md.digest(toByteArray())).toString(16).padStart(32,
'0')
```

```
    }

    @BeforeAll
     fun  init(){
        userDAO = UserDAO(sessionOf("jdbc:postgresql:tpo", "postgres",
"santa666"))
        authService.setSessionDB(userDAO.getSession())
    }


    @Test
    fun login() {
        authService.login("admin", "admin".md5())
        assert(authService.getUser()?.id == 6)
    }

    @Test
    fun grand() {
        authService.login("santa", "santa".md5())
        authService.grand("admin")
        assert(authService.getUserByName("admin")?.permit == 1)
        authService.grand("admin")
    }

    @Test
    fun testGrantToYourself(){
        authService.login("santa", "santa".md5())
        try {
            authService.grand("santa")
        }catch (e: UserException){
            assert(e.message == "You could not ban yourself")
        }
    }

    @Test
```

```
fun testUnexeptebleUserName(){
    authService.login("santa", "santa".md5())
    try {
        authService.grand("unxept")
    }catch (e: UserException){
        assert(e.message == "User is not exist")
    }

}

@Test
fun resrAnAdminUserGrant(){
    authService.login("admin", "admin".md5())
    try {
        authService.grand("commonUser")
    }catch (e: UserException){
        assert(e.message == "User is not administrator!")
    }
}

@Test
fun unban() {
    authService.login("santa", "santa".md5())
    userDAO.insert("userForBan", "banme".md5())
    userDAO.setPermitToUserByUserName("userForBan", 2)
    authService.unban("userForBan")
    assert(authService.getUserByName("userForBan")?.permit == 0)
    userDAO.deleteAccount(userDAO.getIdByUserName("userForBan"))
}

@Test
fun checkPermitforUnbun(){
    authService.login("admin", "admin".md5())
    userDAO.insert("userForBan", "banme".md5())
    userDAO.setPermitToUserByUserName("userForBan", 2)
```

```
    try {
        authService.unban("userForBan")
    }catch (e: UserException){
        assert(e.message == "User is not administrator!")
        userDAO.deleteAccount(userDAO.getIdByUserName("userForBan"))
    }
}

@Test
fun chackBanYourself(){
    authService.login("santa", "santa".md5())
    try {
        authService.ban("santa")
    }catch (e: UserException){
        assert(e.message == "You could not ban yourself")
    }
}

@Test
fun checkBanAdministartor(){
    authService.login("santa1", "santa1".md5())
    try {
        authService.ban("santa")
    }catch (e: UserException){
        assert(e.message == "You could ban administrator")
    }
}

@Test
fun checkPermitforbun(){
    authService.login("admin", "admin".md5())
    userDAO.insert("userForBan", "banme".md5())
    userDAO.setPermitToUserByUserName("userForBan", 2)
    try {
        authService.ban("userForBan")
```

```
    }catch (e: UserException){
        assert(e.message == "User is not administrator!")
        userDAO.deleteAccount(userDAO.getIdByUserName("userForBan"))
    }
}

@Test
fun ban() {
    authService.login("santa", "santa".md5())
    userDAO.insert("userForBan", "banme".md5())
    authService.ban("userForBan")
    assert(authService.getUserByName("userForBan")?.permit == 2)
    userDAO.deleteAccount(userDAO.getIdByUserName("userForBan"))
}

@Test
fun viewAllUsers() {
    val usersD: List<User> = userDAO.getAllUsers()
    val users: List<User> = authService.viewAllUsers()
    assert(users.size == usersD.size)
}

@Test()
fun checkGetAllUserForCommonUser(){
    authService.login("admin", "admin".md5())
    try{
        val users: List<User> = authService.viewAllUsers()
    }catch (e: UserException){
        equals(e.message == "User is not administrator!")
    }
}

@Test
fun viewUserByName() {
    val username = "admin"
```

```
    val userD: User? = userDAO.getUserByName(username)
    val user = authService.getUserByName(username)
    if (userD != null) {
        if (user != null) {
            assert(userD.id == user.id)
        }
    }
    assert(user != null)
}

@Test
fun deleteAccount() {
    userDAO.insert("userForDelete", "deleteme".md5())
    authService.login("userForDelete", "deleteme".md5())
    authService.deleteAccount()
    assert(!authService.isLogind())
}

@Test
fun logout() {
    authService.login("admin", "admin".md5())
    authService.logout()
    assert(!authService.isLogind())
}

@Test
fun registration() {
    authService.registration("newUser", "pass".md5())
    assert(userDAO.find("newUser", "pass".md5()) != null)
    userDAO.deleteAccount(userDAO.getIdByUserName("newUser"))
}

@Test
fun chechRegistretedUser(){
    try {
```

```
            authService.registration("admin", "admin".md5())
        }catch (e: UserException){
            assert(e.message == "User already exist!")
        }
    }

    @Test
    fun getUsernameById() {
        val user = authService.getUsernameById(6)
        val userD = userDAO.getUserById(6)
        assert(user == userD)
    }

    @Test
    fun checkUserLogin() {
        authService.login("commonUser", "pass".md5())
        assert(authService.isLogind())

    }
    @Test
    fun checkAdmin() {
        authService.login("santa", "santa".md5())
        assert(authService.isAdmin())
    }

    @Test
    fun incorrectLogin(){
        try {
            authService.login("blabla", "bla".md5())
        }catch (e: UserException) {
            assert(e.message == "Incorrect userName or Password!")
        }

    }
}
```

DictionaryService

```
class DictionaryServise {
    private lateinit var wordDAO: WordDAO
    private lateinit var authService: AuthServise

    fun getWordsByRootName(root: String): List<Word> {
        val roots: List<Root> = wordDAO.getRootsByName(root)
        val words = mutableListOf<Word>()
        if (roots.isEmpty())
            throw WordExeption("Word with this root is not exist!")
        roots.forEach {
            val pref: List<Pref> = wordDAO.getPrefByRootID(it.id)
            val post: List<Post> = wordDAO.getPostByRootId(it.id)
            words.add(Word(pref, it, post))
        }
        words.sortBy { word -> word.count }
        return words
    }

    fun deleteWordById(id: Int?) {
        wordDAO.deleteWordById(id)
    }

    fun addWord(pref: List<String>, root: String, post: List<String>) {
        if (authService.getUser()?.permit == 2)
            throw WordExeption("Ban user could not add word!")
        var rootId: Int? = null
        if (root != "")
            rootId = wordDAO.insertRoot(root, authService.getUser()!!.id)
        if (rootId == null)
            throw WordExeption("Could not insert word!")

        if (pref.isNotEmpty())
            pref.forEach {
```

```
            wordDAO.insertPref(it, rootId)
        }

    if (post.isNotEmpty())
        post.forEach{
            wordDAO.insertPost(it, rootId)
        }
}

fun getUserWords(): List<Word> {
    val roots = wordDAO.getRootsByUserId(authService.getUser()!!.id)
    val words = mutableListOf<Word>()
    if (roots.isEmpty())
        throw WordExeption("User have not words!")
    roots.forEach {
        val pref: List<Pref> = wordDAO.getPrefByRootID(it.id)
        val post: List<Post> = wordDAO.getPostByRootId(it.id)
        words.add(Word(pref, it, post))
    }
    words.sortBy { word -> word.count }
    return words
}

fun getAllWords(): List<Word> {
    val roots: List<Root> = wordDAO.getAllRoots()
    val words = mutableListOf<Word>()
    if (roots.isEmpty())
        throw WordExeption("Dictionary is empty!")
    roots.forEach {
        val pref: List<Pref> = wordDAO.getPrefByRootID(it.id)
        val post: List<Post> = wordDAO.getPostByRootId(it.id)
        words.add(Word(pref, it, post))
    }
    words.sortBy { word -> word.count }
    return words
```

```
    }

    fun setAuthService(authService: AuthServise) {
        this.authService = authService
    }

    fun setWodrDAO(sessionDB: Session) {
        wordDAO = WordDAO(sessionDB)
    }
}
```

AuthService

```
class AuthServise {
    private lateinit var userDAO: UserDAO
    private var user: User? = null

    fun login(username: String, password: String){
        try {
            user = userDAO.find(username, password)!!
        }catch(e: Exception){
            throw UserException("Incorrect userName or Password!")
        }

    }

    fun grand(username: String) {
        if (user!!.permit != 1)
            throw UserException("User is not administrator!")
        if (user!!.userName == username)
            throw UserException("You could not ban yourself")
        if (userDAO.getUserByName(username) == null)
            throw UserException("User is not exist")
        if (userDAO.getUserByName(username)!!.permit == 1)
            userDAO.setPermitToUserByUserName(username, 0)
        else
```

```
        userDAO.setPermitToUserByUserName(username, 1)
}

fun unban(username: String) {
    if (user!!.permit != 1)
        throw UserException("User is not administrator!")
    if (userDAO.getUserByName(username) == null)
        throw UserException("User is not exist")
    userDAO.setPermitToUserByUserName(username, 0)
}

fun ban(username: String) {
    if (user!!.permit != 1)
        throw UserException("User is not administrator!")
    if (user!!.userName == username)
        throw UserException("You could not ban yourself")

    if (userDAO.getUserByName(username) == null)
        throw UserException("User is not exist")
    if (userDAO.getUserByName(username)!!.permit == 1)
        throw UserException("You could ban administrator")
    userDAO.setPermitToUserByUserName(username, 2)
}

fun viewAllUsers(): List<User> {
    if (user!!.permit != 1)
        throw UserException("User is not administrator!")
    return userDAO.getAllUsers()
}

fun viewUserByName(username: String): User? {
    if (user!!.permit != 1)
        throw UserException("User is not administrator!")
    return userDAO.getUserByName(username)
}
```

```
fun deleteAccount() {
    if (user == null) {
        throw UserException("Unable to delete account: user is not logged in.");
    }
    userDAO.deleteAccount(user!!.id)
    logout()
}

fun logout() {
    user = null
}

fun registration(username: String, password: String) {
    if (userDAO.getUserByName(username) != null)
        throw UserException("User already exist!")
    userDAO.insert(username, password)
}

fun getUsernameById(userId: Int): String {
    return userDAO.getUserById(userId)
}

fun setSessionDB(sessionDB: Session) {
    userDAO = UserDAO(sessionDB)
}

fun getUser(): User? {
    if (user != null)
        return user!!
    throw UserException("You are not login")
}

fun isLogind():Boolean{
    return user != null
```

```
    }

    fun isAdmin(): Boolean{
      return user?.permit == 1
    }

    fun getUserByName(s: String): User? {
        return userDAO.getUserByName(s)
    }
}
```

Word

```
data class Word(
    val prefs: List<Pref>,
    val root: Root,
    val posts: List<Post>,
    val count: Int = prefs.size + posts.size
)

data class Root(
    val id: Int,
    val name: String,
    val userId: Int
)
val toRoot: (Row) -> Root = { row ->
    Root(
        row.int("id"),
        row.string("name"),
        row.int("user_id")
    )
}

data class Pref(
    val name: String,
    val root_id: Int
```

```
)
val toPref: (Row) -> Pref = { row ->
    Pref(
        row.string("name"),
        row.int("root_id")
    )

}

data class Post(
    val name: String,
    val root_id: Int
)
val toPost: (Row) -> Post = { row ->
    Post(
        row.string("name"),
        row.int("root_id")
    )
}
```

User

```
data class User(
    var id: Int,
    val userName: String,
    val password: String,
    val permit: Int
)

val toUser : (Row) -> User = { row ->
    User(
        row.int("id"),
        row.string("user_name"),
        row.string("pass"),
        row.int("permit")
    )
```

```
}
```

WordExeption

```
class WordExeption(message: String) : Exception(message)
```

UserExeption

```
class UserException(message: String) : Exception(message)
```

UserDAO

```
class UserDAO(private val session: Session) {

    fun find(username: String, password: String): User? {
        val queryString = "select * from lab2.user where user_name = ? and pass =
?"
        return session.run(queryOf(queryString, username,
password).map(toUser).asSingle)
    }

    fun getAllUsers(): List<User> {
        val queryString = "select * from lab2.user"
        return session.run(queryOf(queryString).map(toUser).asList)
    }

    fun getUserByName(username: String): User? {
        val queryString = "select * from lab2.user where user_name = ?"
        return session.run(queryOf(queryString, username).map(toUser).asSingle)

    }

    fun setPermitToUserByUserName(username: String, permit: Int) {
        val queryString = "update lab2.user set permit = ? where user_name = ?"
        session.run(queryOf(queryString, permit, username).asUpdate)
    }

    fun deleteAccount(id: Int) {
```

```
        val queryString = "delete from lab2.user where id = ?"
        session.run(queryOf(queryString, id).asExecute)
    }

    fun insert(username: String, password: String) {
        val insertQuery = "insert into lab2.user (user_name, pass, permit) values (?,
?, ?)"
        session.run(queryOf(insertQuery, username, password, 0).asUpdate)
    }

    fun getUserById(userId: Int): String {
        val queryString = "select * from lab2.user where id = ?"
        return session.run(queryOf(queryString,
userId).map(toUser).asSingle)!!.userName
    }

    fun getSession(): Session {
        return session
    }

    fun getIdByUserName(s: String): Int {
        val queryString = "select * from lab2.user where user_name = ?"
        return session.run(queryOf(queryString, s).map(toUser).asSingle)!!.id
    }

}
```

WordDAO

```
class WordDAO (private val session: Session){
    fun getRootsByName(root: String): List<Root> {
        val queryString = "select * from lab2.root where name = ?"
        return session.run(queryOf(queryString, root).map(toRoot).asList)
    }

    fun getPrefByRootID(id: Int): List<Pref> {
```

```
        val queryString = "select * from lab2.pref where root_id = ?"
        return session.run(queryOf(queryString, id).map(toPref).asList)
    }

    fun getPostByRootId(id: Int): List<Post> {
        val queryString = "select * from lab2.post where root_id = ?"
        return session.run(queryOf(queryString, id).map(toPost).asList)
    }

    fun deleteWordById(id: Int?) {
        val queryString = "delete from lab2.root where id = ?"
        session.run(queryOf(queryString, id).asExecute)
    }

    fun insertRoot(root: String, userId: Int?): Int? {
        val insertQueryRoot: String = "insert into lab2.root (name, user_id) values
(?, ?) RETURNING id;"
        return session.single(queryOf(insertQueryRoot, root, userId)){ row ->
row.int("id")}
    }

    fun insertPref(name: String, rootId: Int?) {
        val insertQueryPref: String = "insert into lab2.pref (name, root_id) values
(?, ?)"
        session.run(queryOf(insertQueryPref, name, rootId).asUpdate)
    }

    fun insertPost(name: String, rootId: Int?) {
        val insertQueryPref: String = "insert into lab2.post (name, root_id) values
(?, ?)"
        session.run(queryOf(insertQueryPref, name, rootId).asUpdate)
    }

    fun getRootsByUserId(userId: Int): List<Root> {
        val queryString = "select * from lab2.root where user_id = ?"
```

```
        val i =session.run(queryOf(queryString, userId).map(toRoot).asList)
        return i
    }

    fun getAllRoots(): List<Root> {
        val queryString = "select * from lab2.root"
        return session.run(queryOf(queryString).map(toRoot).asList)
    }

    fun getSession(): Session {
        return session
    }

}
```

Вывод

В ходе выполнения работы было разработано консольное приложение. Перед разработкой основного функционала были реализованы тесты, это позволило избежать много ошибок во время реализации и заранее продумать защиту некоторых модулей.

Во время написания кода стало понятно преимущества DDT: меньшее использование отладчика, так как тесты отлично показывают ошибки. Так же проще реализация самой логики, так как она была продумана на этапе написания тестов и требований.

В целом, разработка через тестирование дает ряд неоспоримых преимуществ, таких упрощение разработки, удобство рефакторинга и сопровождаемости кода, более высокое качество продукта за счет полного покрытия тестами. Однако TDD требует куда более серьезного подхода к разработке требований и выбору архитектуры будущего продукта.