федеральное государственное автономное образовательное учреждение высшего образования

Санкт-Петербургский национальный исследовательский УНИВЕРСИТЕТ информационных технологий, механики и оптики

**Факультет программной инженерии и компьютерной техники**

**ЛАБОРАТОРНАЯ РАБОТА №2**

**ПО ДИСЦИПЛИНЕ «ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ»**

**Вариант 19**

Выполнил: Чудаков М.И.

Группа: P3319

Преподаватель: Исаев И.В.

Санкт-Петербург

2019

**Цель**

Необходимо разработать консольное приложение в соответствие с методологий TDD.

**Задачи**

1. Необходимо из краткого описания сформировать функциональные требования к программному продукту (не менее 20, т.е. дополнить);
2. Необходимо полностью покрыть тестами данные требования используя разработку через тестирование;
3. Для хранения данных использовать PostgreSQL или MongoDB;
4. Реализовать продукт 100% покрытый тестами.

**Вариант 19 (4)**

Разработка консольного приложения “Музыкальный каталог”.

При запуске программы пользователю предлагается справочная информация по использованию. Далее предлагается ввести команду, позволяющую выполнить одно из действий:

* осуществить поиск музыкальной композиции в каталоге по определенному критерию;
* вывести информацию о всех существующих в каталоге композициях;
* добавить информацию о композиции в каталог;
* удалить существующую в каталоге запись;
* выйти из программы.

Критериями поиска могут служить: имя (название) автора/исполнителя или название композиции. В качестве результата поиска в консоль должен выводиться список композиций в виде «исполнитель – название». Удаление или добавление записи осуществляется после ввода всей информации о композиции.

**TDD**

Разработка через тестирование (test-driven development, TDD) — техника разработки программного обеспечения, которая основывается на повторении очень коротких циклов разработки: сначала пишется тест, покрывающий желаемое изменение, затем пишется код, который позволит пройти тест, и под конец проводится рефакторинг нового кода к соответствующим стандартам.

**BDD**

BDD (англ. Behavior-driven development, дословно «разработка через поведение») — это методология разработки программного обеспечения, являющаяся ответвлением от методологии разработки через тестирование (TDD). Основной идеей данной методологии является совмещение в процессе разработки чисто технических интересов и интересов бизнеса, позволяя тем самым управляющему персоналу и программистам говорить на одном языке. Для общения между этими группами персонала используется предметно-ориентированный язык, основу которого представляют конструкции из естественного языка, понятные неспециалисту, обычно выражающие поведение программного продукта и ожидаемые результаты.

BDD методология является расширением TDD в том смысле, что перед тем, как написать какой-либо тест необходимо сначала описать желаемый результат от добавляемой функциональности на предметно-ориентированном языке. После того как это будет проделано, конструкции этого языка переводятся специалистами или специальным программным обеспечением в описание теста.

**Сформулированные функциональные требования**

1. Музыкальная композиция содержит следующие критерии: название, автор, жанр. Каждый критерий является отдельной сущностью БД: Композиция (название, автор, жанр); Автор (имя); Жанр (название).
2. Все данные хранятся в БД под управлением СУБД postgresql
3. Сочетание названия и автора композиции должно быть уникальным. Имя автора должно быть уникальным. Название жанра должно быть уникальным.
4. Добавление новой композиции в каталог. При добавлении новой композиции пользователь передает название композиции, автора и жанр. В случае если автор не существует данные заносятся в соответствующую таблицу, в случае если существует, данные в таблице “автор”, не дублируются, при создании композиции, в поле “автор” заносится ссылка на существующая запись. Запись о жанре должна содержаться в соответствующей таблице, в случае если запись не существует, новая композиция не создается.
5. Добавление нового жанра. Пользователь передает название жанра, если данный жанр уже существует, новая запись не создается.
6. Удалить существующий жанр со всеми привязанными композициями. Пользователь передает имя жанра.
7. Вывести информацию о всех существующих композициях.
8. Вывести все композиции одного автора, имя автора передается пользователем.
9. Найти композицию по названию, пользователь передает название композиции и имя автора.
10. Удалить существующую в каталоге запись о композиции, пользователь передает название композиции и имя автора.
11. Поиск композиций по жанру, пользователь передает название жанра.
12. Вывести информацию о всех существующих авторах.
13. Вывести информацию о всех существующих жанров.
14. Создание плейлиста, плейлист является сущностью БД: Плейлист (название). Связан с сущностью “Композиция” связью многие ко многим. Пользователь передает название плейлиста.
15. Добавление существующей композиции в плейлист. Пользователь передает название плейлиста, название композиции и имя автора.
16. Удаление композиции из плейлиста, пользователь передает название плейлиста, название композиции и имя автора.
17. Возможность удалить плейлист. Пользователь передает название плейлиста.
18. Просмотр названий всех плейлистов.
19. Просмотр всех композиций в плейлисте, пользователь передает название плейлиста.
20. При запуске приложения пользователь получает справочную информацию по использованию.

**Исходный код**

Пример сценария

**Feature:** Add composition  
  
 @addComposition  
 **Scenario Outline:** add composition  
 **Given** we have new composition "<compositionName>", "<authorName>" and "<genreName>"  
 **When** we try add this composition  
 **Then** new composition equals null "<result>"  
  
 **Examples:  
 | compositionName | authorName | genreName | result |  
 | name | author | pop | true |  
 | name | author1 | pop | false |  
 | name1 | author | pop | false |  
 | name | author | rap | true |**

Реализация

**class** CompositionAddSteps : En {  
 **private val dbManager**: DBManager = PgDBManager()  
  
 **private lateinit var compositionName**: String  
 **private lateinit var authorName**: String  
 **private lateinit var genreName**: String  
  
 **private var composition**: Composition? = **null  
  
 init** {  
 Given(**"we have new composition {string}, {string} and {string}"**) **{** compositionName: String, authorName: String, genreName: String **->  
 this**.**compositionName** = compositionName  
 **this**.**authorName** = authorName  
 **this**.**genreName** = genreName  
 **}** When(**"we try add this composition"**) **{  
 composition** = **dbManager**.addComposition(**compositionName**, **authorName**, **genreName**)  
 **}** Then(**"new composition equals null {string}"**) **{** result: String **->** *assertEquals*(result.*toBoolean*(), **composition** == **null**)  
 **}** }  
  
 @After(**"@addComposition"**)  
 **fun** after() {  
 *transaction* **{  
 composition**?.*let* **{** CompositionDao.find **{** Compositions.**name** *eq* **it**.**name }**.*firstOrNull* **{** c **->** c.**author**.**name** == **it**.**author**.**name }**?.delete()  
 **}** AuthorDao.find **{** Authors.**name** *eq* **"author1" }**.*firstOrNull*()?.delete()  
 **}** }  
}

Бизнес логика

**interface** DBManager {  
  
 **fun** addComposition(compositionName: String, authorName: String, genreName: String): Composition?  
  
 **fun** deleteCompositions(compositionName: String, authorName: String)  
  
 **fun** getAllCompositions(): List<Composition>  
  
 **fun** addGenre(name: String): Genre?  
  
 **fun** deleteGenres(name: String)  
  
 **fun** getAllGenres(): List<Genre>  
  
 **fun** findCompositionByName(compositionName: String, authorName: String): Composition?  
  
 **fun** findCompositionsByAuthor(authorName: String): List<Composition>  
  
 **fun** findCompositionsByGenre(genreName: String): List<Composition>  
  
 **fun** getAllAuthors(): List<Author>  
  
 **fun** createPlaylist(name: String): Playlist?  
  
 **fun** deletePlaylist(name: String)  
  
 **fun** getAllPlaylistNames(): List<String>  
  
 **fun** getPlaylist(name: String): Playlist?  
  
 **fun** addCompositionToPlaylist(playlistName: String, compositionName: String, authorName: String): Boolean  
  
 **fun** deleteCompositionFromPlaylist(playlistName: String, compositionName: String, authorName: String): Boolean  
  
 **fun** getMessage(): String  
}

**object** Compositions : IntIdTable() {  
 **val name** = varchar(**"name"**, 50)  
  
 **val author** = reference(**"author\_id"**, Authors, ReferenceOption.**CASCADE**, ReferenceOption.**CASCADE**)  
  
 **val genre** = reference(**"genre\_id"**, Genres, ReferenceOption.**CASCADE**, ReferenceOption.**CASCADE**)  
  
 **init** {  
 index(**true**, **name**, **author**)  
 }  
}

**class** CompositionDao(id: EntityID<Int>) : IntEntity(id) {  
 **companion object** : IntEntityClass<CompositionDao>(Compositions)  
  
 **var name by** Compositions.**name  
  
 var author by** AuthorDao referencedOn Compositions.**author  
  
 var genre by** GenreDao referencedOn Compositions.**genre**}

**data class** Composition(  
 **val name**: String,  
 **val author**: Author,  
 **val genre**: Genre  
)

**override fun** addComposition(compositionName: String, authorName: String, genreName: String): Composition? {  
 **var** composition: Composition? = **null** *transaction* **{  
 val** genre = GenreDao.find **{** Genres.**name** *eq* genreName **}**.*firstOrNull*()  
  
 **if** (genre == **null**) {  
 **message** = **"This genre doesn't exist"  
 return**@transaction  
 }  
  
 **val** author = AuthorDao.find **{** Authors.**name** *eq* authorName **}**.*firstOrNull*()  
 ?: AuthorDao.new **{ name** = authorName **}  
  
 val** isExist = CompositionDao.find **{** Compositions.**name** *eq* compositionName  
 **}**.*any* **{ it**.**author**.**id** == author.**id }  
  
 if** (isExist) {  
 **message** = **"This composition already exist"  
 return**@transaction  
 }  
  
 **val** dao = CompositionDao.new **{  
 name** = compositionName  
 **this**.**author** = author  
 **this**.**genre** = genre  
 **}** composition = Composition(dao.**name**, Author(author.**name**), Genre(genre.**name**))  
 **}  
  
 return** composition  
}  
  
**override fun** deleteCompositions(compositionName: String, authorName: String) {  
 *transaction* **{** CompositionDao.find **{** Compositions.**name** *eq* compositionName **}** .*firstOrNull* **{ it**.**author**.**name** == authorName **}**?.delete()  
 **}**}

**Вывод**

В ходе выполнения лабораторной работы мной было разработано консольное приложение “Музыкальный каталог”, по методологии BDD – behavior driven development. В результате я изучил технологию разработки через поведение, такую как Cucumber – программный инструмент для тестирования кода, написанного через поведение с помощью сценариев на предметно-ориентированном языке и SQL библиотеку – Exposed, написанную на языке Kotlin. Данный подход разработки позволил проще представить необходимую функциональность приложение, за счет изначальной фокусировки на тестах. Также стоит отметить, что при таком подходе я реже использовал отладчик. Из минусов стоит отметить, что общий объем кода больше чем при стандартном подходе разработки, при этом время разработки отличается не существенно, так как данный подход уменьшил количество ошибок.