**Пишем JUnit тесты на Java**

После написания программы хороший программист покрывает свой код тестами, дабы убедиться, что методы работают так, как он планировал.

Тесты бывают нескольких видов:

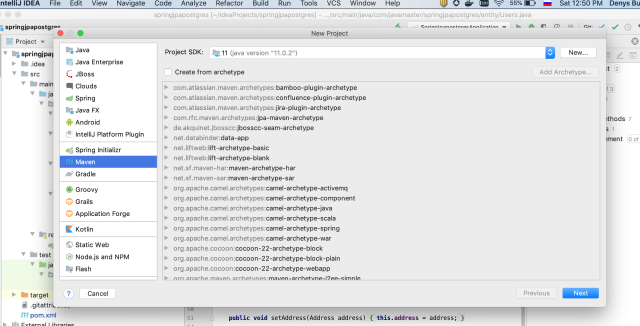
- юнит тесты (как правило тестируются отдельные методы или сервисы);

- интеграционные тесты (тестируются взаимодействия нескольких участков системы)

Сегодня нас интересуют юнит тесты. Де-факто фреймворком для написания юнит тестов на языке Java является JUnit.

Я сейчас не буду расписывать процент проектов, которые используют JUnit, историю создания, популярность в Интернете. Вместо этого я предлагаю сразу начать писать код.

Для того, чтобы начать писать тесты — нам нужен код, который мы будем этими тестами покрывать. Предлагаю создать простой Maven проект и написать в нем небольшой сервис. Я буду все делать в Intellij idea, но, как я говорю каждый раз: выбор среды программирования не имеет значения. Главное, чтобы Вам было удобно и понятно. Шаги по созданию проекта следующие: File->New->Project. Далее будет окно с выбором типа проекта:

создание нового Maven проекта в среде intellij idea

Выбираем Maven и нажимаем Next. Далее заполняем groupId и artifactId значениями com.javamaster и junit\_example соответственно.

JUnit — это сторонняя библиотека и ее нужно подключить. Просто добавим зависимость в pom.xml. В результате наш мавен файл стал:

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.javamaster</groupId>

<artifactId>junit\_example</artifactId>

<version>1.0-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

</dependencies>

</project>

Перед тестами я написал небольшие классы Users, CustomFieldException и UsersService.

Класс UsersService выступит в качестве кода, который мы попытаемся покрыть тестами.

import java.time.LocalDate;

import java.util.List;

import java.util.stream.Collectors;

import static java.util.Objects.isNull;

class **Users** {

private String name;

private LocalDate dateOfBirth;(это день рождения) //такой тип нужно брать из ***Java 8 Date/Time API***

public Users(String name, LocalDate dateOfBirth) {

/\* В этом примере *(в отличаи от файла “Примеры Junit1.doc”)* проверка на правильность вводимых значений, а также заполнение коллекции будет делаться в классе с логикой! При этом проверки на существование уже таких данных в коллекции нет.

Такая логика написания кода не правильная, т.к. любому из ребят команды кому необходимо будет воспользоваться нашими переменными придется самому создавать стандартные обязательные проверки, а потом их еще и тестировать!!!!!

Проверку на правильность заполнения необходимо делать как в сеттерах так и в конструкторе (только так мы однозначно закроемся о попыток неправильного введения данных как через сеттер так и при создании нового объекта класса)!!!

\*/

this.name = name;

this.dateOfBirth = dateOfBirth;

}

public User() {}

//геттеры и сеттеры

public String getName() { return name; }

public void setName(String name) { this.name = name; }

public LocalDate getDateOfBirth () { return dateOfBirth; }

public void setDateOfBirth (LocalDate dateOfBirth) { this. dateOfBirth = dateOfBirth; }

//дальше идут переопределения стандартных методов

@Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 Users users = (Users) o;  
 return Objects.equals(name, users.name) && dateOfBirth.equals(users. dateOfBirth);  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.hash(name, dateOfBirth);  
 }  
  
 @Override  
 public java.lang.String toString() {  
 return "Users{" +  
 "name='" + name + '\'' +  
 ", dateOfBirth =" + dateOfBirth + '}';  
 }  
  
}

-----------------------------------------------------------------------------------------------------------------------------------------------

class **CustomFieldException** extends Exception {

private String message;

public CustomFieldException(String message) {

this.message = message;

}

public String getMessage() {

return message;

}

}

------------------------------------------------------------------------------------------------------------------------------------------------

public class **UsersService** {

private List<Users> users;

public UsersService(List<Users> users) {

this.users = users;

}

//геттер

public List<Users> **getUsers**() {

return users;

}

// после проверки на правильность введения данных идет заполнение коллекции юзерами

public List<Users> **createNewUser**(String name, LocalDate dateOfBirth) throws Exception {

validateUser(name, dateOfBirth);

Users user = new Users(name, dateOfBirth);

users.add(user);

return users;

}

//Удаление пользователя из коллекции. Мы меняем значение переменной класса “users”. Создаем из нашей коллекции новую List коллекцию из которой будет исключена запись *(если такая есть)* с нашим именем и присваиваем нашей переменной “users” значение этой новой коллекции.

public void **removeUser**(String name) {

users = users.stream().filter(x -> !x.name.equals(name)).collect(Collectors.toList());

}

// проверка *“date”* ,“*user” + в* “*user”-е “dateOfBirth”* на null с выводом своего *CustomFieldException*, после чего проверяет на соответствие дня и месяца в “*user” дню и месяцу в “date” и получаем на вывод метода true/false.*

public boolean **isBirthDay**(Users user, LocalDate date) throws CustomFieldException {

if (isNull(user) || isNull(user.dateOfBirth)) {

throw new CustomFieldException("User or date of birth is null");

}

if (isNull(date)) {

throw new CustomFieldException("Compare date must not be null");

}

return (date.getDayOfMonth() == user.dateOfBirth.getDayOfMonth()) &&

(date.getMonth().equals(user.dateOfBirth.getMonth()));

}

//проверка “*name”* на null/empty и “*dateOfBirth”* на null.

private void **validateUser**(String name, LocalDate dateOfBirth) throws Exception {

if (isNull(name) || name.isBlank()) {

throw new CustomFieldException("Name could not be empty or null");

}

if (isNull(dateOfBirth)) {

throw new CustomFieldException("Date of birth could not be null");

}

}

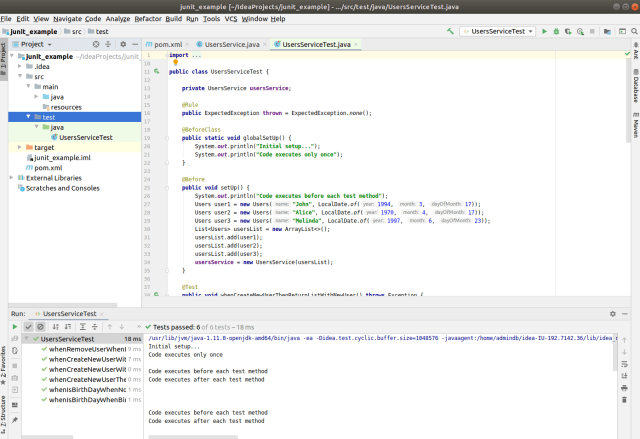
}

Код выше — очень простой пример, который не нуждается в дополнительных объяснениях. Если же он Вам по какой-то причине не понятен или очень сложный — я настоятельно советую ознакомиться со статьями Java для новичка.

Теперь приступим к написанию самих тестов.

**JUnit тест** — это простой класс java, методы которого помечены аннотацией @Test. Как правило, один метод отвечает за тестирование одного кейса. В библиотеке также предусмотрены и другие аннотации которые применяют для предварительной подготовки к тестированию: @BeforeClass, @Before, @AfterClass, @After. Их примеры я тоже покажу в коде. Также, в библиотеке есть готовый класс *Assert*, с помощью которого можно проводить сравнения фактических значений с ожидаемыми.

Теперь, создадим класс *UsersServiceTest* в директории *src/main/test/java.* Если Вы создавали проект строго по инструкции выше — эта директория должна быть. Но, часто такое бывает, что она отсутствует. В этом нет ничего страшного. Просто создайте ее руками. Структура проекта должна быть примерно как на изображении ниже.

структура проекта

Теперь сам код теста:

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertNotNull;

import static org.junit.jupiter.api.Assertions.assertThrows;

import static org.junit.jupiter.api.Assertions.assertTrue;

import org.junit.\*;

import org.junit.rules.ExpectedException;

import java.time.LocalDate;

import java.util.ArrayList;

import java.util.List;

import static org.hamcrest.CoreMatchers.is;

import static org.junit.Assert.\*;

public class UsersServiceTest {

private UsersService usersService;

@Rule

// ExpectedException это класс для тестирования ожидаемых исключений в Junit 4

public ExpectedException thrown = ExpectedException.none();

@BeforeClass

public static void globalSetUp() {

System.out.println("Initial setup...");

System.out.println("Code executes only once");

}

@Before

public void setUp() {

System.out.println("Code executes before each test method");

Users user1 = new Users("John", LocalDate.of(1994, 3, 17));

Users user2 = new Users("Alice", LocalDate.of(1970, 4, 17));

Users user3 = new Users("Melinda", LocalDate.of(1997, 6, 23));

List<Users> usersList = new ArrayList<>();

usersList.add(user1);

usersList.add(user2);

usersList.add(user3);

// дополнительная строка кода, для создания коллекции из введенных данных

usersService = new UsersService(usersList);

}

// **assertThat()** дальше это тоже самое, что и Assert.equals() только немного отличается написание

**//проверяет работу метода *createNewUser() (*до добавления элементов в коллекцию и после)**

@Test

public void whenCreateNewUserThenReturnListWithNewUser() throws Exception {

assertThat(usersService.getUsers().size(), is(3));

usersService.createNewUser("New User", LocalDate.of(1990, 2, 1));

assertThat(usersService.getUsers().size(), is(4));

}

**// проверяет работу метода *removeUser()* удаляя один элемет и проверяя кол-во записей**

@Test

public void whenRemoveUserWhenRemoveUserByName(){

usersService.removeUser("Melinda");

List<Users> usersList = usersService.getUsers();

assertThat(usersList.size(), is(2));

}

**//Дальше идут два теста проверяющих работу собственного исключения из класса CustomFieldException при попытке ввода null в “name” / ” dateOfBirth ”**

@Test

//для Junit4

public void whenCreateNewUserWithoutNameThenThrowCustomFieldException() throws Exception {

thrown.expect(CustomFieldException.class);

thrown.expectMessage("Name could not be empty or null");

usersService.createNewUser(null, LocalDate.of(1990, 2, 1));

//для Junit5

void whenCreateNewUserWithoutNameThenThrowCustomFieldException() throws Exception {

// ниже написано, что данное исключение должно быть вызвано при выполнении такого метода с такими

// параметрами и тогда тест будет пройден (если не вызвано, то не пройден)

assertThrows(CustomFieldException.class, () -> {usersService.createNewUser(null, LocalDate.of(1990, 2, 1)); });

//ниже может быть добавлена еще дополнительная проверка на *assertNotNull*()/*assertEquals*()

}

@Test

//для Junit4

public void whenCreateNewUserWithoutDateOfBirthThenThrowCustomFieldException() throws Exception {

thrown.expect(CustomFieldException.class);

thrown.expectMessage("Date of birth could not be null");

usersService.createNewUser("Dave", null);

}

**//проверка на правильность выведения False/True методом isBirthDay()**

@Test

public void whenIsBirthDayWhen**Birthday**Then**ReturnTrue**() throws CustomFieldException {

assert**False** (usersService.isBirthDay**(**usersService.getUsers().get(0), LocalDate.of(1990, 2, 1)**)**);

}

@Test

public void whenIsBirthDayWhen**NotBirthday**Then**ReturnFalse**() throws CustomFieldException {

assert**True** (isBirthDay(usersService.getUsers().get(0), LocalDate.of(1994, 3, 17)));

}

@AfterClass

public static void tearDown() {

System.out.println("Tests finished");

}

@After

public void afterMethod() {

System.out.println("Code executes after each test method");

}

}

Как видно по коду выше, первое, что я сделал — добавил необходимые мне классы и произвел предварительную настройку перед тестом. Я добавил переменную UsersService, ведь именно этот класс я и планирую протестировать. Далее я добавил

@Rule

public ExpectedException thrown = ExpectedException.none();

Аннотация Rule в JUnit 4 — это компонент, который перехватывает вызовы тестового метода и позволяет нам что-то делать до запуска тестового метода и после запуска тестового метода.

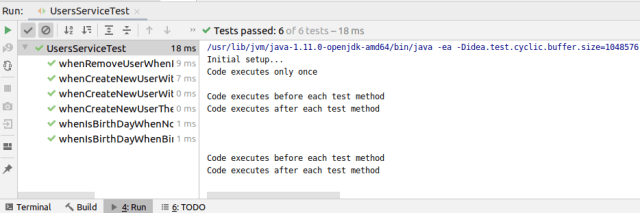
Класс ExpectedException — это тоже из библиотеки JUnit. Правило ExpectedException позволяет вам проверить, что ваш код вызывает конкретное исключение. А если хотите проще: используйте конструкцию выше, для «отлова» нужных Вам исключительних ситуаций.

Далее идут сами методы, которые тестируют функциональность класса UsersService. Из названий методов видно какой функционал они тестируют.

Иногда, бывает полезно не только настроить предварительные ресурсы перед тестированием, но и очистить некоторые из них. С этой задачей отлично позволяют справиться аннотации @AfterClass и @After. В данном примере им приминения не нашлось, но я создал их для примера.

Для запуска тестов нужно нажать правой кнопкой на классе и выбрать Run. Или запустить тест, нажав на зеленую кнопку в intellij idea. В eclipse механизм запуска должен быть похожим. Причем запускать можно как весь класс тестов, так и каждый тест по отдельности.

В результате успешного запуска и отработки тестов Вы должны увидеть в консоли примерно такой результат:



результат успешно отработанных тестов

Цель тестов — выявить ошибки в программе на ранних стадиях, когда код еще не успел уйти в продакшин. Помните, не нужно подстраивать тесты под функционал или, что еще хуже — функционал под тесты. Если определенные тесты не проходят — нужно обратить внимание на бизнес логику, ошибки в коде. Перед этим нужно внимательно ознакомиться с результатом теста и понять, что же все-таки не так.

Возьмем для примера метод isBirthDay. Его функционал и назначение понятны: метод должен возвращать есть ли у пользователя день рождение в зависимости от даты рождения и входной даты. Почему я использовал входную дату вместо использования например текущей даты? Дело в том, что я не знаю когда будут запускаться данные тесты. И может быть такой вариант, что **запуск тестов** на текущую дату совпадет с датой написанной нами “от фонаря” для проверки неправильности даты, и это даст нам неверный результат их работы.

Функционал isBirthDay покрывают два теста:

@Test

public void whenIsBirthDayWhenBirthdayThenReturnTrue() throws CustomFieldException {

boolean isBirthday = usersService.isBirthDay(usersService.getUsers().get(0), LocalDate.of(1990, 2, 1));

assertFalse(isBirthday);

}

@Test

public void whenIsBirthDayWhenNotBirthdayThenReturnFalse() throws CustomFieldException {

boolean isBirthday = usersService.isBirthDay(usersService.getUsers().get(0), LocalDate.of(1990, 3, 17));

assertTrue(isBirthday);

}

Если бы вместо даты сравнения я использовал бы текущую дату, запуск теста 2 месяца 1 дня — вернул бы true в то время, когда я ожидал совсем другое.

Вернемся к нашему експерименту. Сейчас мы увидим, почему так важно иметь тесты на существующий функционал. Особенно перед тем как потом этот функционал будет изменен. Часто так бывает, что над проектом работал один человек, потом другой, далее проект попадает к другим людям и возникает необходимость изменить уже существующий функционал или его часть.

Допустим, Вас попросили изменить реализацию метода isBirthDay. Возможно, пример примитивный и простой, но в целях учебы вполне себе сойдет.

Вы что-то там делали и написали вот такое решение:

public boolean isBirthDay(Users user, LocalDate date) throws CustomFieldException {

if (isNull(user) || isNull(user.dateOfBirth)) {

throw new CustomFieldException("User or date of birth is null");

}

if (isNull(date)) {

throw new CustomFieldException("Compare date must not be null");

}

return user.dateOfBirth.isEqual(date);

}

Можете думать, что данный пример я выдумал, но в свое время, у нас был метод в приложении, который определял есть ли день рождения у пользователя. Причем, отрабатывал шедулер, который выбирал из базы данных пользователей, брал дату их рождения и потом отсылал поздравление на почту. Проблема была в том, что пользователей было примерно пол миллиона, а день рождения у них хранился не в очень удобном для java формате. Код поздравлений с днем рождения отрабатывал очень медленно. Программист, который его написал явно был не очень опытным: он брал из базы дату рождения пользователя в виде строки, потом вынимал из строки день и месяц, и сравнивал их с текущим днем и месяцем.

В результате, мы пришли к решению, что код нужно оптимизировать. Это дело поручили одному программисту, который и предложил сравнивать дату рождения пользователя с текущей :). Удивительно, но тестов на тот функционал не было ни до, ни после «оптимизации». Вот он пришел ко мне примерно с таким кодом:

public boolean isBirthDay(Users user){

return user.dateOfBirth.isEqual(LocalDate.now());

}

и говорит: «Не понимаю, в базе данных так много пользователей но пока никому не отправилось поздравление». Вот такой вот казус случился однажды с реальным приложением. А были бы тесты — они бы все показали.

Давайте попробуем запустить тесты и посмотреть, что же будет отрабатывать когда мы используем такое решение для наших именинников:

public boolean isBirthDay(Users user, LocalDate date) throws CustomFieldException {

if (isNull(user) || isNull(user.dateOfBirth)) {

throw new CustomFieldException("User or date of birth is null");

}

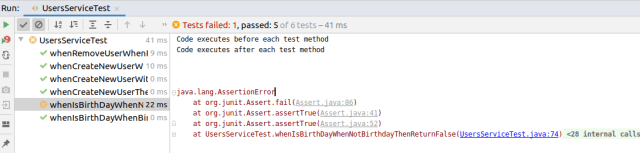
if (isNull(date)) {

throw new CustomFieldException("Compare date must not be null");

}

return user.dateOfBirth.isEqual(date);

}



результат запуска с падением тестов

Как видно на изображении выше — тесты очень хорошо справились с поставленной задачей.

У Вас часто будет возникать вопрос: сколько нужно тестов, чтобы убедиться в правильности написанного кода? На это нет конкретного ответа. Старайтесь писать столько тестов, сколько сами считаете нужным. На реальных проектах Вы будете писать столько, на сколько будет хватать времени. Я советую покрывать тестами некоторые подозрительные и опасные места в коде. Особено тщательно нужно покрывать места, где идет работа кода с финансами, округлениями. Как показывает опыт, именно они наиболее часто дают ошибки.