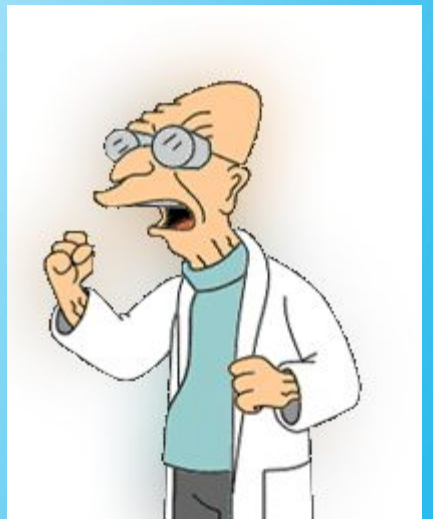
Unit-тесты

Что такое Unit-тест и с чем его едят

« *Фрагмент кода, написанный разработчиком, для проверки очень маленького, специфичного фрагмента функциональности кода.* »

Зачем нужно возиться с тестами

- они делают жизнь проще
- они делают дизайн приложения лучше
- они значительно уменьшают время, затрачиваемое на отладку.



Чего мы добиваемся написанием тестов

Ответов на вопросы:

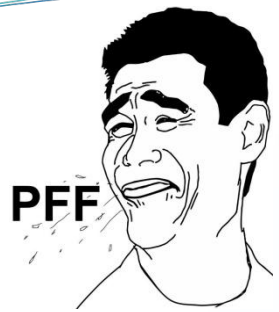
- Делает ли код то, что ожидает от него разработчик?
- Делает ли он это все время?
- Можно ли положиться на код?

Побочные эффекты:

- Документирование кода и способов работы с ним

Как писать юнит-тесты

1. Ответить на вопрос: как мы будем тестировать новый метод.
2. Написать тест и тестируемый класс.
3. Запустить тест.
4. Запустить ВСЕ тесты системы.



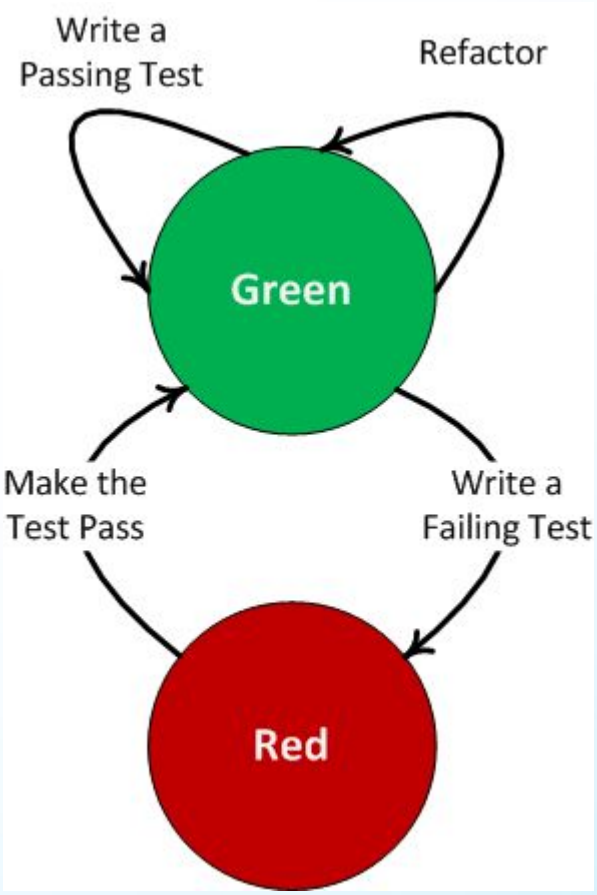
Типичные отговорки

- Написание тестов занимает слишком много времени
- Запуск тестов занимает слишком много времени
- Это не моя работа – тестировать мой код
- Я не знаю точно, как код должен работать, поэтому я не могу написать тест
- Но он же компилируется!
- Мне платят за написание кода, а не тестов
- Я чувствую вину, оставляя QA без работы
- Моя компания не позволит запустить юнит-тесты на live-системе

Как исправлять баги

- Идентифицировать баг
- Написать тест, который «поломается» от этого бага, чтобы подтвердить наличие бага.
- Исправить код так, чтобы тест выполнился.
- Запустить **ВСЕ** остальные тесты.

Red-Green-Refactor



JUnit

«*Библиотека (фреймворк) для модульного тестирования программного обеспечения на языке Java.*»

JUnit API: Assertion (Утверждения)

- assertEquals
- assertFalse
- assertTrue
- assertNotNull
- assertNull
- assertNotSame
- assertSame

Junit API: Annotations

- @Test
- @Before
- @After
- @Ignore

JUnit Example: Class under test

```
public class User {
    private String name;

    public User(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public boolean isNamesake(User user) {
        return getName().equals(user.getName());
    }
}
```

Junit Example: Test class

```
public class UserTest {
    @Test
    public final void testIsNameSake() {
        User user1 = new User("Софья");
        User user2 = new User("Алёшка");

        Assert.assertNotNull(user1);
        Assert.assertNotNull(user2);

        Assert.assertNotSame(user1, user2);

        Assert.assertFalse(user1.isNamesake(user2));
    }
}
```

Легко? Всё меняется, когда приходят **ОНИ**

Внешние зависимости

Внешняя зависимость — это объект, с которым взаимодействует код и над которым нет прямого контроля. Для ликвидации внешних зависимостей в модульных тестах используются **тестовые объекты**

Внешняя зависимость от БД

```
public class User {
    private static final UserDAO userDAO = new UserDAO();
    private Integer id;
    private String name;

    public User(Integer id) {this.id = id;}

    public String getName() {
        if (name == null) {name = getUserDAO().getName(id);}
        return name;
    }

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

    UserDAO getUserDAO() {return userDAO;}

    public boolean isNamesake(User user) {return getName().equals(user.getName());}

    public void save() {getUserDAO().save(this);}
}
```

Виды тестовых объектов

- dummy
- stub
- spy
- mock
- fake

Dummy (Пустышка)

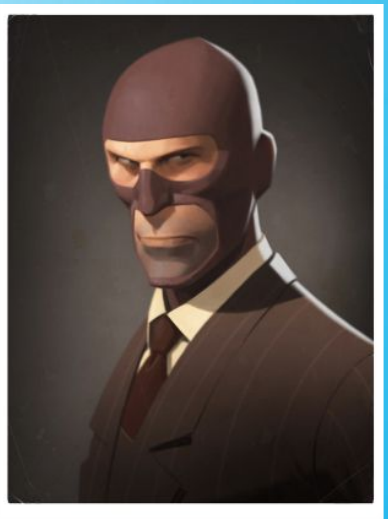
Объект, который обычно передается в тестируемый класс в качестве параметра, но не имеет поведения, с ним ничего не происходит, никакие методы не вызываются. Примером таких dummy-объектов являются new object(), null, «Ignored String» и т.д.

Stub (Заглушка)

Используется для получения данных из внешней зависимости, подменяя её. При этом игнорирует все данные, могущие поступать из тестируемого объекта в stub. Один из самых популярных видов тестовых объектов. Тестируемый объект использует чтение из конфигурационного файла? Передаем ему ConfigFileStub, возвращающий тестовые строки конфигурации для избавления зависимости на файловую систему.

Spy (Шпион)

Используется для тестов взаимодействия, основной функцией является запись данных и вызовов, поступающих из тестируемого объекта для последующей проверки корректности вызова зависимого объекта. Позволяет проверить логику именно нашего тестируемого объекта, без проверок зависимых объектов.



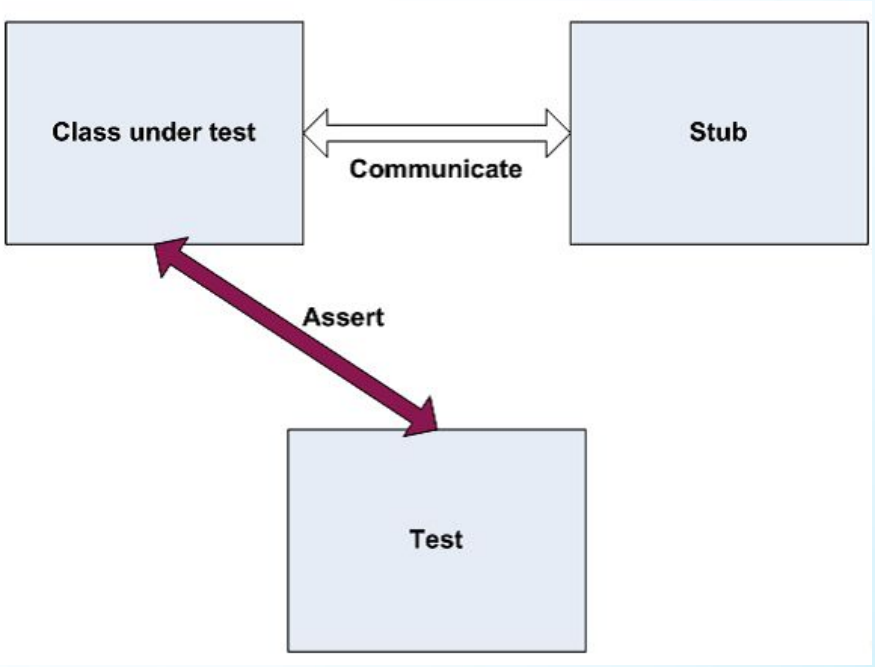
Mock

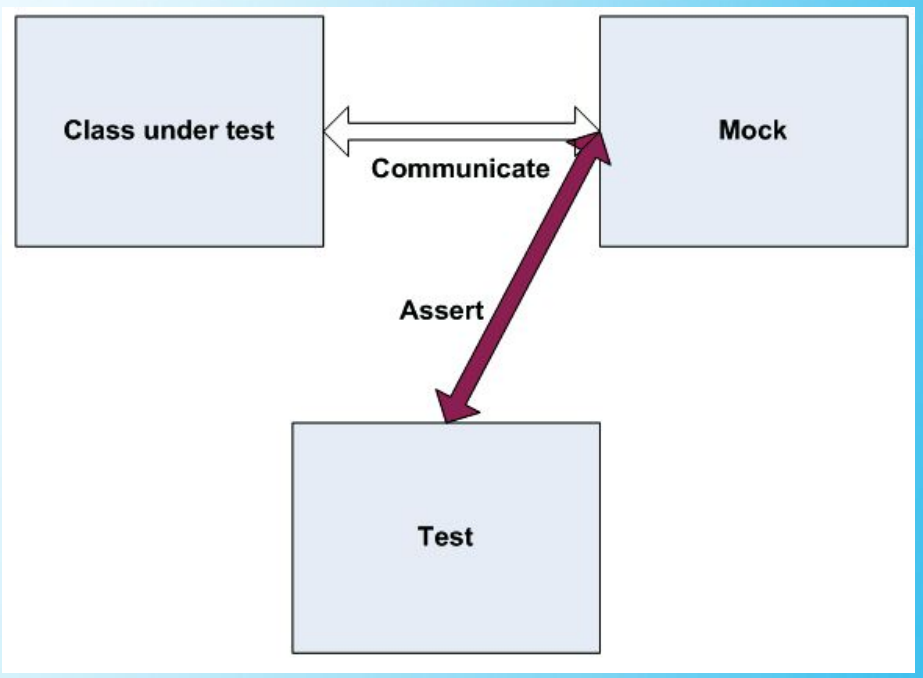
Очень похож на spy, однако не записывает последовательность вызовов с переданными параметрами для последующей проверки, а может сам выкидывать исключения при некорректно переданных данных. Т.е. именно мок-объект проверяет корректность поведения тестируемого объекта.

Fake

Используется в основном чтобы запускать (незапускаемые) тесты (быстрее) и ускорения их работы. Эдакая замена тяжеловесного внешнего зависимого объекта его легковесной реализацией. Основные примеры — эмулятор для конкретного приложения БД в памяти (fake database) или фальшивый вебсервис.

Stubs vs Mocks





“Ручные” “моки” и “стабы”

Stub

```
public class UserStub extends User {
    private String name;

    public UserStub(String name) {
        super(null);
        this.name = name;
    }

    @Override
    public String getName() {return name;}

    @Override
    public void setName(String name) {}

    @Override
    public boolean isNamesake(User user) {return false;}

    @Override
    public void save() {}
}
```

Stub: Test case

```
public class UserTest {
    @Test
    public final void testManualStub() {
        User user1 = new UserStub("Софья");
        User user2 = new UserStub("Алёшка");

      Assert.assertFalse(user1.getName().equals(user2.getName()));
    }
}
```

Mock

```
public class UserMock extends User {
    private UserDAO userDAO;

    public UserMock(Integer id, UserDAO userDAO) {
        super(id);
        this.userDAO = userDAO;
    }

    @Override
    public UserDAO getUserDAO() {
        return userDAO;
    }
}
```

Stub for Mock

```
public class UserDAOStub extends UserDAO {
    @Override
    public String getName(Integer id) {
        switch(id) {
        case 1:
            return "Софья";
        case 2:
            return "Алёшка";
        default:
            return "Noname";
        }
    }

    @Override
    public void save(User user) {}
}
```

Mock with Stub: Test case

```
public class UserTest {
    @Test
    public final void testManualMock() {
        UserDAO userDAOStub = new UserDAOStub();
        User user1 = new UserMock(1, userDAOStub);
        User user2 = new UserMock(2, userDAOStub);
        User user3 = new UserMock(1, userDAOStub);
        Assert.assertFalse(user1.isNamesake(user2));
        Assert.assertTrue(user1.isNamesake(user3));
    }
}
```

Слишком сложно? Отведайте mockito!

```
import static org.mockito.Mockito.*;
```

Mockito: stubbing

```
public class UserTest {
    @Test
    public final void testMockitoStub() {
        User user1 = mock(User.class);
        when(user1.getName()).thenReturn("Софья");

        User user2 = mock(User.class);
        when(user2.getName()).thenReturn("Алёшка");

        Assert.assertFalse(user1.getName().equals(user2.getName()));
    }
}
```

Mockito: mocking

```
public class UserTest {
    @Test
    public final void testMockitoMock1() {
        User user1 = mock(User.class);
        when(user1.getName()).thenReturn("Софья");
        when(user1.isNamesake(any(User.class))).thenCallRealMethod();

        User user2 = mock(User.class);
        when(user2.getName()).thenReturn("Алёшка");

        Assert.assertFalse(user1.isNamesake(user2));
    }
}
```

Mockito: spying & stubbing

```
public class UserTest {
    @Test
    public final void testIsMockitoMock2() {
        UserDAO userDAOStub = mock(UserDAO.class);
when(userDAOStub.getName(anyInt())).thenReturn("Софья").
                                                    thenReturn("Алёшка");
        doNothing().when(userDAOStub).save(any(User.class));

        User user1 = spy(new User(1));
        when(user1.getUserDAO()).thenReturn(userDAOStub);

        User user2 = spy(new User(2));
        when(user2.getUserDAO()).thenReturn(userDAOStub);

        Assert.assertFalse(user1.isNamesake(user2));
    }
}
```

Mockito: spying

```
public class UserTest {
    @Test
    public final void testMockitoSpy() {
        UserDAO userDAOStub = mock(UserDAO.class);
        when(userDAOStub.getName(anyInt())).thenAnswer(new Answer<String>() {
            @Override
            public String answer(InvocationOnMock invocation) throws Throwable {
                Integer id = (Integer) invocation.getArguments()[0];
                switch (id) {
                case 1:
                    return "Софья";
                case 2:
                    return "Алёшка";
                default:
                    return "Noname";
                }
            }
        });
        doThrow(new RuntimeException()).when(userDAOStub).save(any(User.class));
        User user1 = spy(new User(1));
        when(user1.getUserDAO()).thenReturn(userDAOStub);
        User user2 = spy(new User(2));
        when(user2.getUserDAO()).thenReturn(userDAOStub);
        Assert.assertFalse(user1.isNamesake(user2));
    }
}
```

Подведём итоги

- Тестируйте все, что может сломаться
- Тестируйте все, что уже сломалось
- Новый код признается виновным, пока не доказана его невиновность.
- Тестового кода должно быть как минимум сколько же, сколько и production-кода.
- Запускайте юнит-тесты локально при каждой компиляции.
- Запускайте все юнит-тесты перед check-in-ом в репозиторий.

Спасибо за внимание!