




Lesson 13

29.08.2024



Phase 1:
Requirement
Analysis

Phase 2:
Feasibility
Study

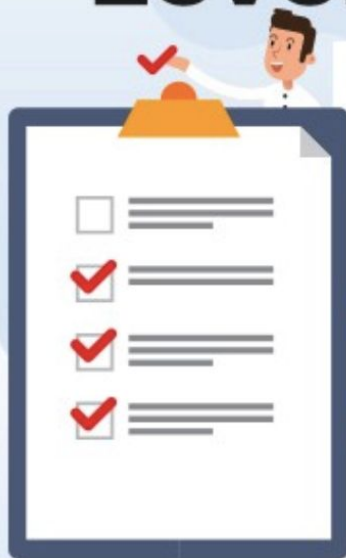
Phase 3:
Architectural
Design

Phase 4:
Software
Development

Phase 5:
Testing

Phase 6:
Deployment

Levels of Testing



UNIT TESTING

Test Individual Component

INTEGRATION TESTING

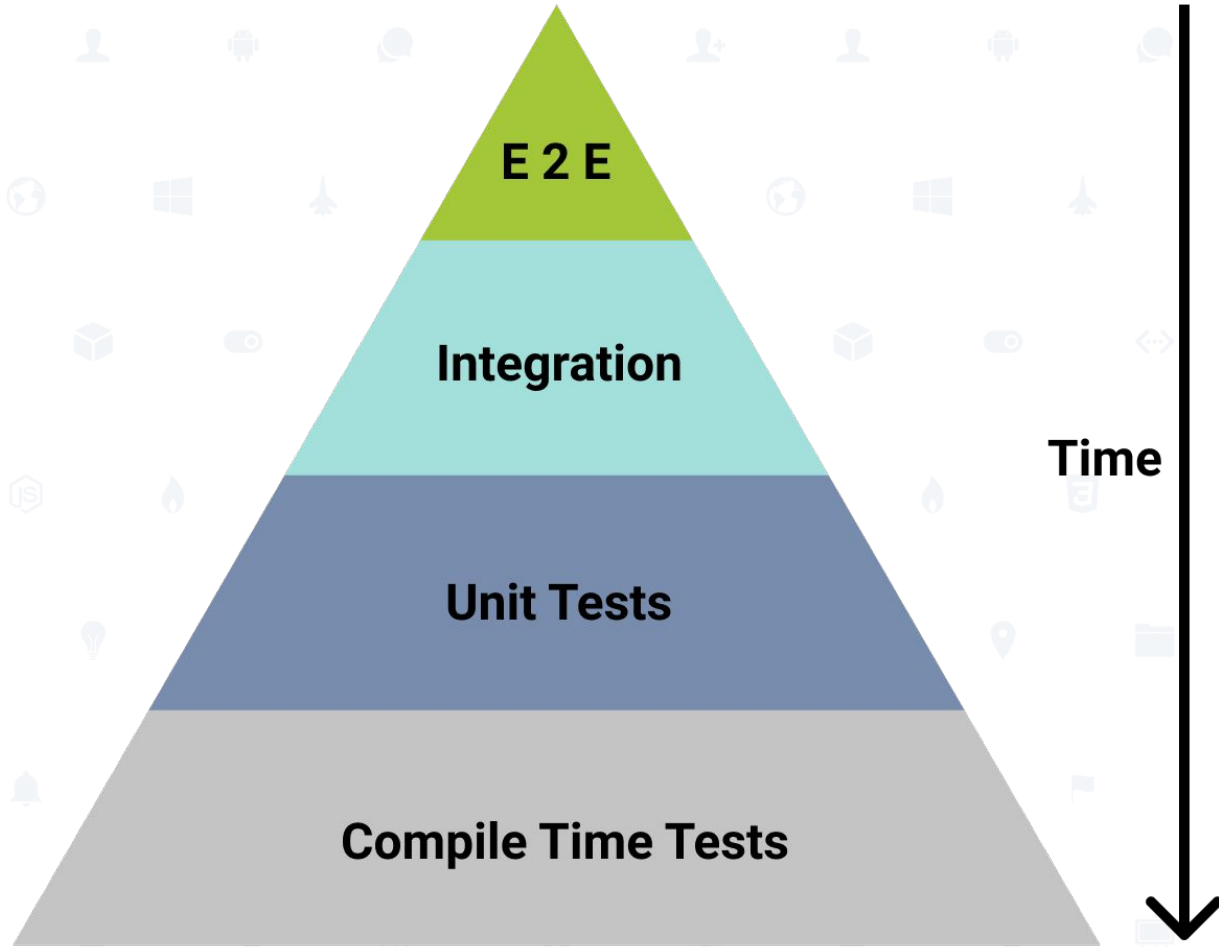
Test Integrated Component

+ SYSTEM TESTING

+ Test the entire System

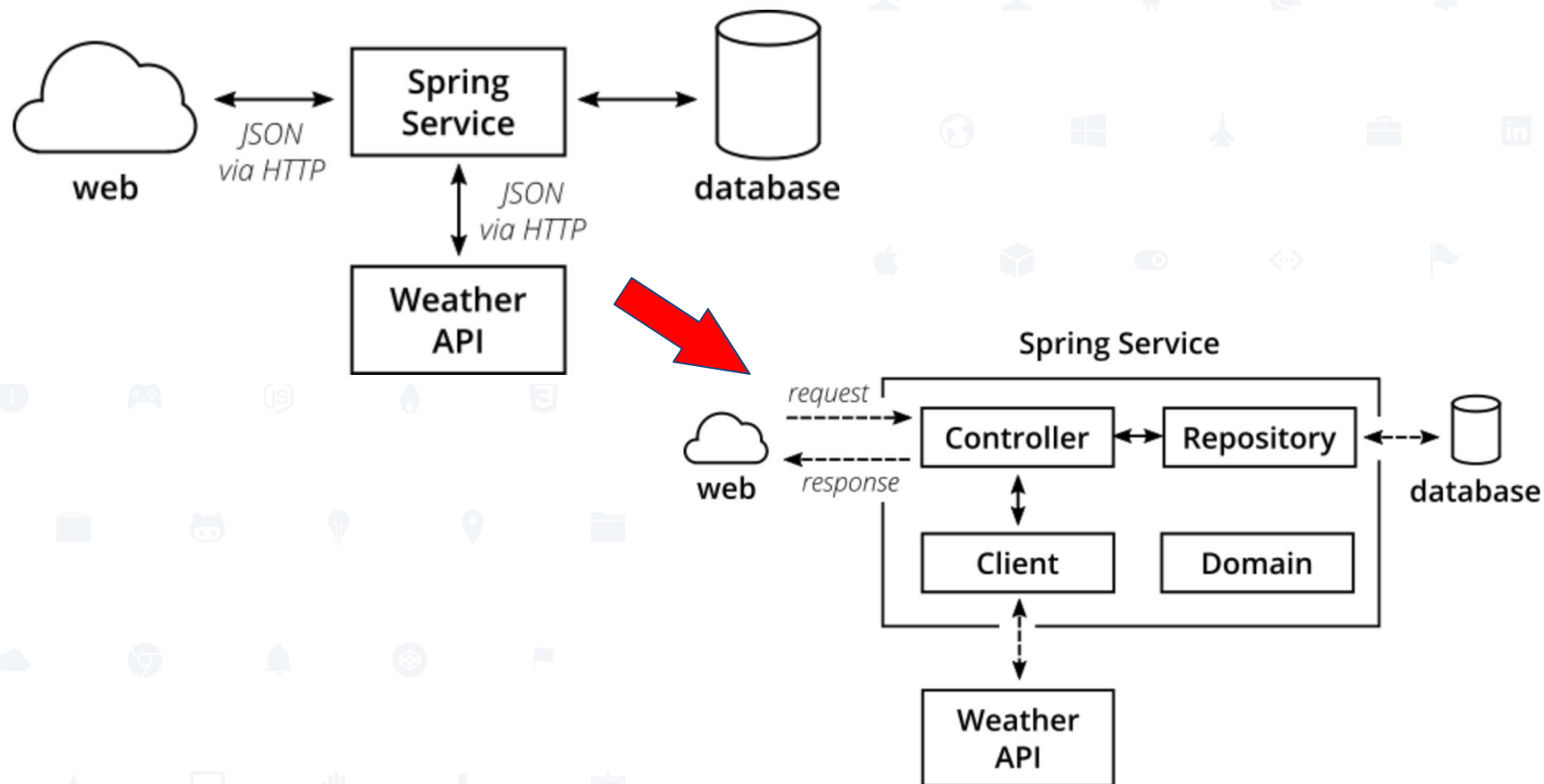
ACCEPTANCE TESTING

Test the final System





The Sample Application





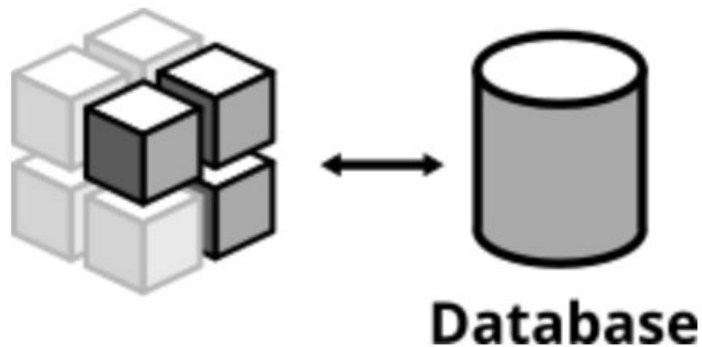
Unit tests



Основу вашого набору тестів складатимуть модульні тести. Ваші модульні тести гарантують, що певний блок (тестований предмет) вашої кодової бази працює належним чином. Модульні тести мають найвужчий обсяг з усіх тестів у вашому наборі тестів. Кількість одиничних тестів у вашому наборі тестів буде значно перевищувати кількість інших типів тестів.



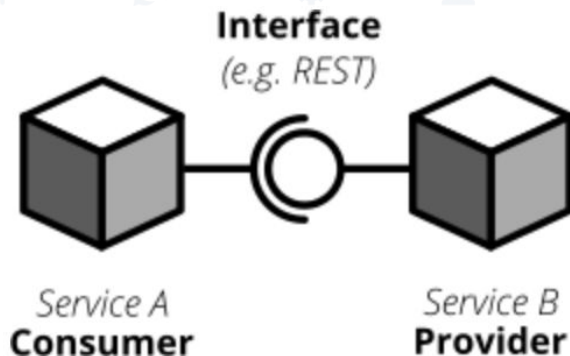
Integration Tests



Усі програми будуть інтегруватися з деякими іншими частинами (базами даних, файловими системами, мережевими викликами інших програм). Під час написання модульних тестів це зазвичай ті частини, які ви пропускаєте, щоб отримати кращу ізоляцію та швидші тести. Проте ваша програма взаємодіятиме з іншими частинами, і це потрібно перевірити. Тести інтеграції тут, щоб допомогти. Вони перевіряють інтеграцію вашої програми з усіма частинами, які знаходяться поза вашою програмою.



Contract Tests

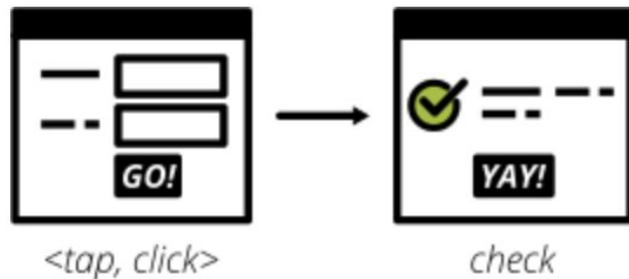


Більш сучасні організації, що займаються розробкою програмного забезпечення, знайшли способи масштабування своїх зусиль у розробці, розподіливши розробку системи між різними командами. Окремі команди створюють окремі, слабко пов'язані служби, не наступаючи одна одній на ноги, і інтегрують ці служби у велику згуртовану систему. Останні галаси навколо мікросервісів зосереджені саме на цьому.

Поділ вашої системи на багато невеликих служб часто означає, що ці служби повинні спілкуватися одна з одною через певні (сподіваємось, чітко визначені, іноді випадково розширені) інтерфейси.



UI Tests

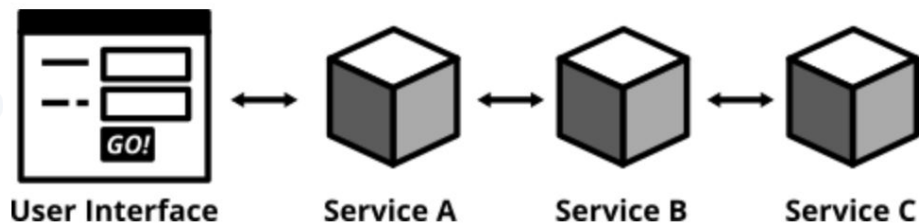


Більшість програм мають певний інтерфейс користувача. Зазвичай ми говоримо про веб-інтерфейс у контексті веб-додатків. Люди часто забувають, що REST API або інтерфейс командного рядка є таким же інтерфейсом користувача, як і модним веб-інтерфейсом користувача.

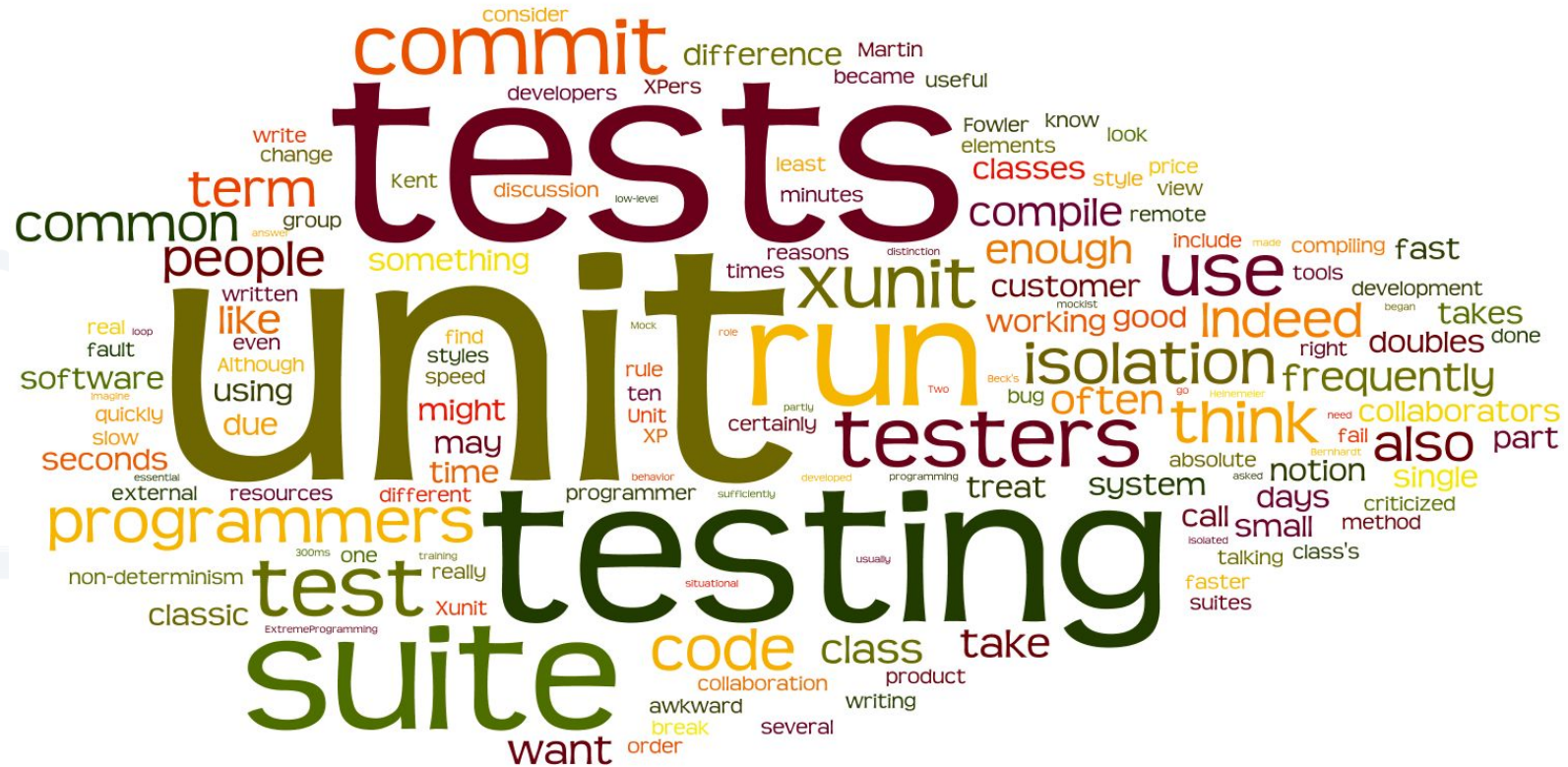
Тести інтерфейсу користувача перевіряють, чи правильно працює інтерфейс користувача вашої програми. Введення користувача має викликати правильні дії, дані повинні бути представлені користувачеві, стан інтерфейсу користувача має змінюватися, як очікувалося.



End-to-End Tests



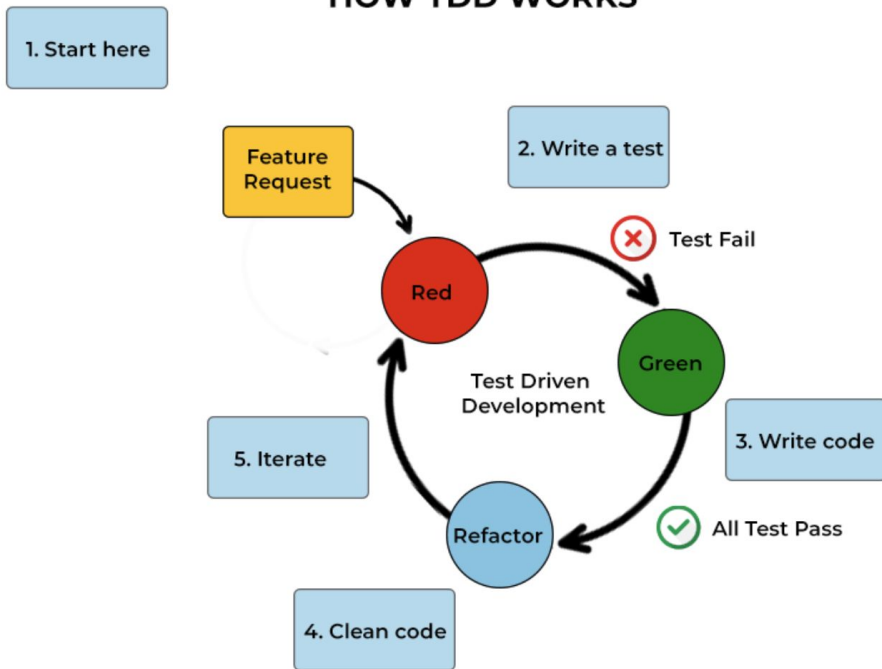
Наскрізні тести (також звані тестами широкого стеку) дають вам найбільшу впевненість, коли вам потрібно вирішити, чи працює ваше програмне забезпечення чи ні. Selenium і протокол WebDriver дозволяють вам автоматизувати ваші тести, автоматично керуючи (без головного) браузером вашими розгорнутими службами, виконуючи кліки, вводячи дані та перевіряючи стан вашого інтерфейсу користувача. Ви можете використовувати Selenium безпосередньо або використовувати інструменти, побудовані на його основі, Nightwatch — один із них.





Test-driven development (TDD) (Unit test lifecycle)

HOW TDD WORKS



1. Add a test, which will certainly FAIL. (Red) ...
2. Run all the tests. See if any test fails. ...
3. Write only enough code to pass all the tests. (Green) ...
4. Run all the tests. If any test fails, go back to step 3. ...
5. Refactor the code. (Refactor) ...
6. If a new test is added, repeat from step 1.



FIRST

FAST - INDEPENDENT - REPEATABLE - SELF-VALIDATING - TIMELY

UNIT TESTING **FIRST** PRINCIPLES

- **Fast** — Each test should run fast, really fast.
- **Isolated** — It should have no dependency involved.
- **Repeatable** — It should be idempotent.
- **Self-verifying** — Result should be just pass/fail, no extra investigation.
- **Timely** — Every code change should result a new test.



F.I.R.S.T.

Fast: коли тести виконуються повільно, ви не захочете запускати їх часто. Згодом код почне гнити.

Independent: Тести не повинні залежати один від одного. Один тест не повинен створювати умови для наступного тесту. Ви повинні мати можливість запускати кожен тест окремо та запускати тести в будь-якому порядку.

Repeatable: тест має бути повторюваним у будь-якому середовищі. Ви повинні мати можливість запускати тести у виробничому середовищі, середовищі контролю якості та на своєму ноутбуці, коли ви їдете додому в поїзді без мережі.

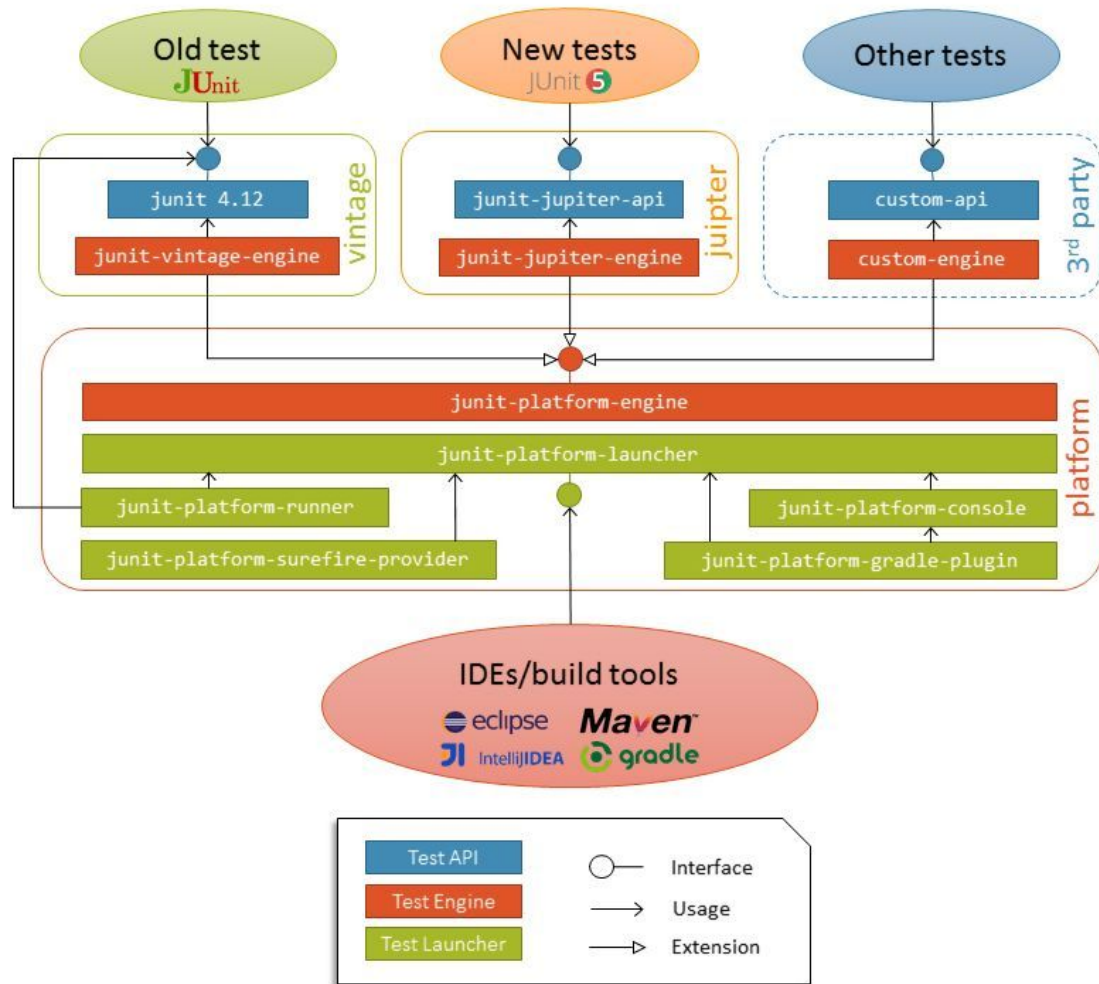
Self-Validating: тести повинні мати несправний результат. Або вони проходять, або не. Вам не потрібно читати файл журналу, щоб визначити, чи тести пройшли.

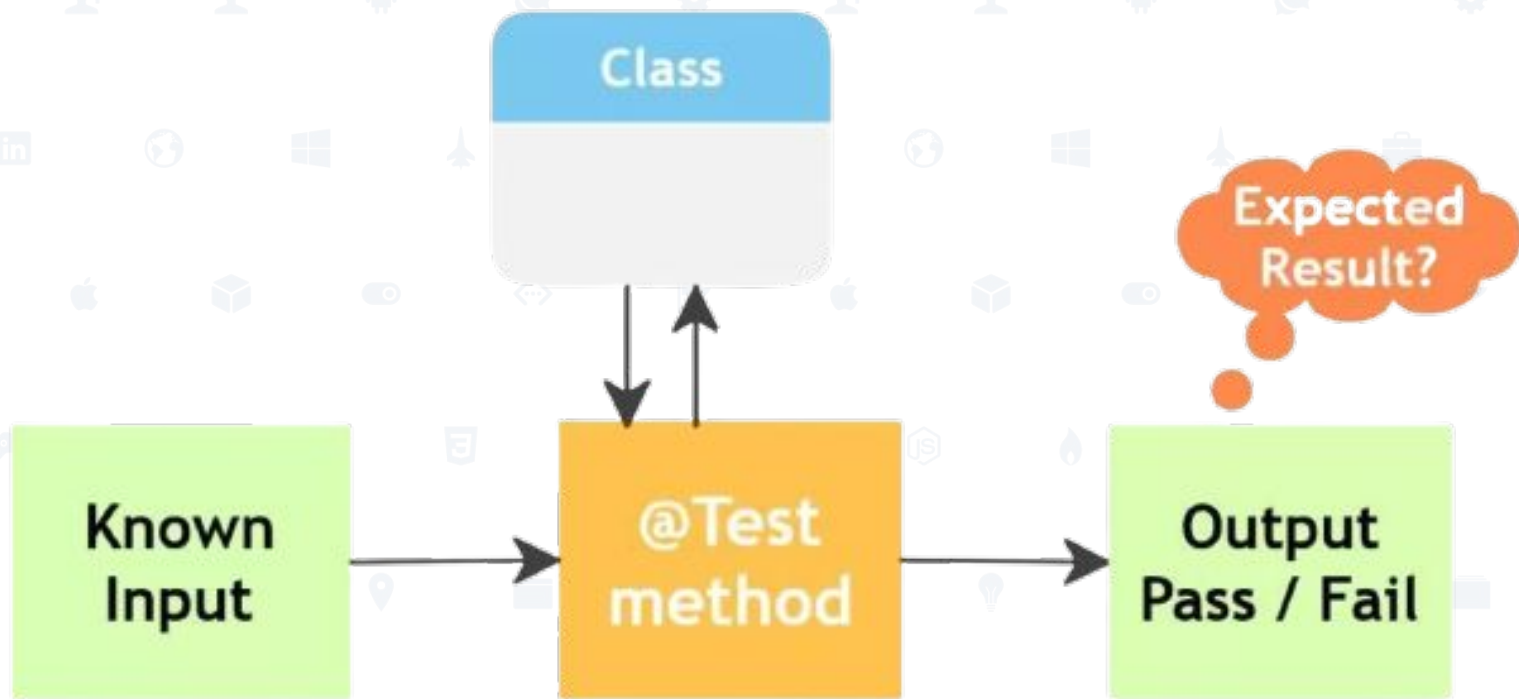
Timely: модульні тести слід писати безпосередньо перед робочим кодом, який забезпечує їх проходження. Якщо ви пишете тести після робочого коду, ви можете виявити, що робочий код буде важко протестувати.



mockito







Unit Test Method

GivenWhenThen pattern

GIVEN

- In this step we are putting the system into known state by defining start data. For example we can put some data into database, set up environment data and so on.

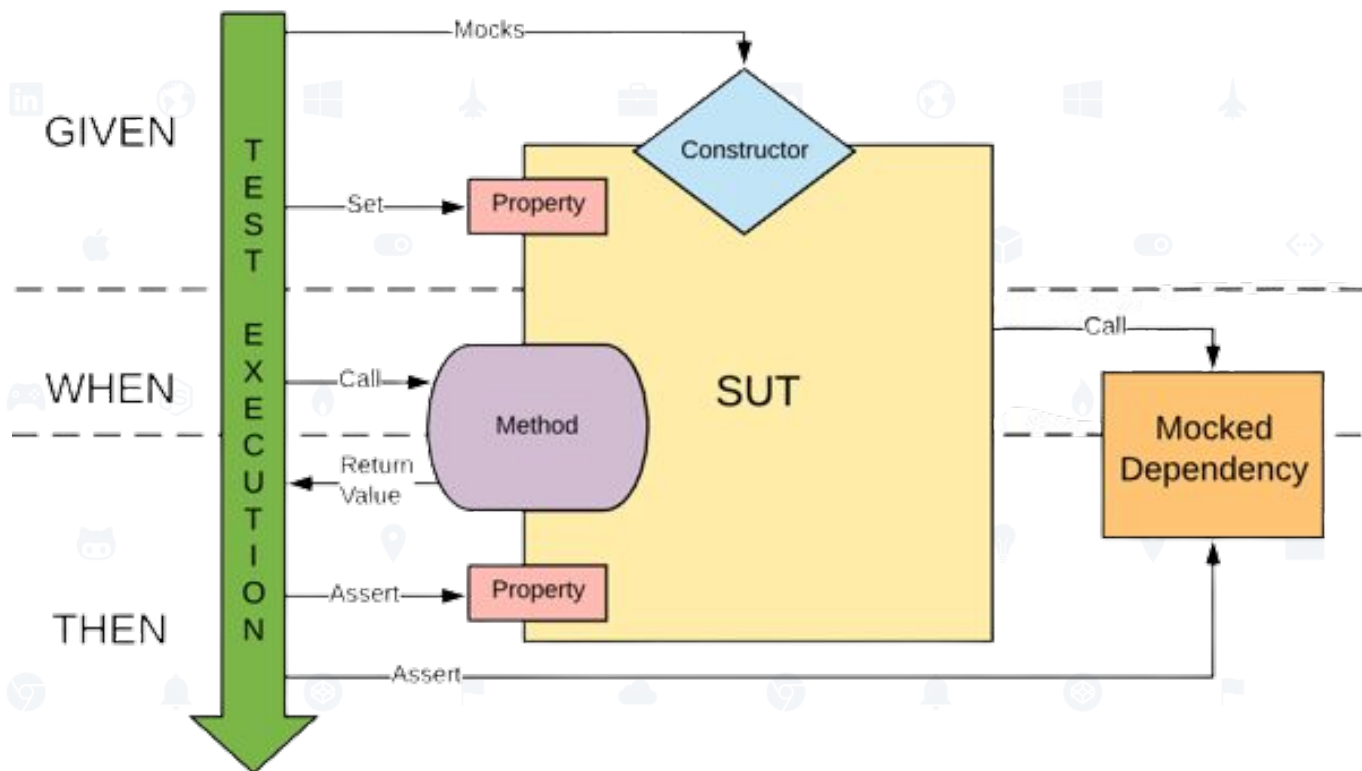
WHEN

- This step defines a key action. It means that in this part of the test system does think what we expect from him.

THEN

- This is a verification step. In this part we observe system output, we check if system behave in the expected way or not and we draw conclusions.

UNIT TEST



System Under Test (SUT)

1 **GIVEN** bean with
#id available



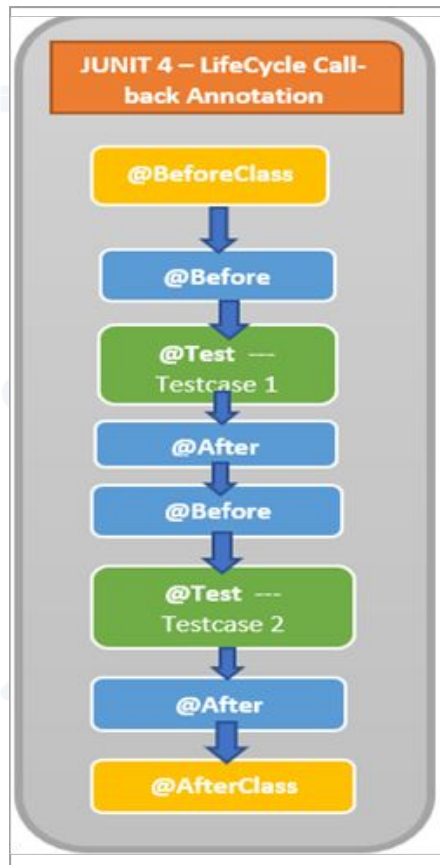
3 **THEN** expect bean
to returned

2 **WHEN** bean found in
database

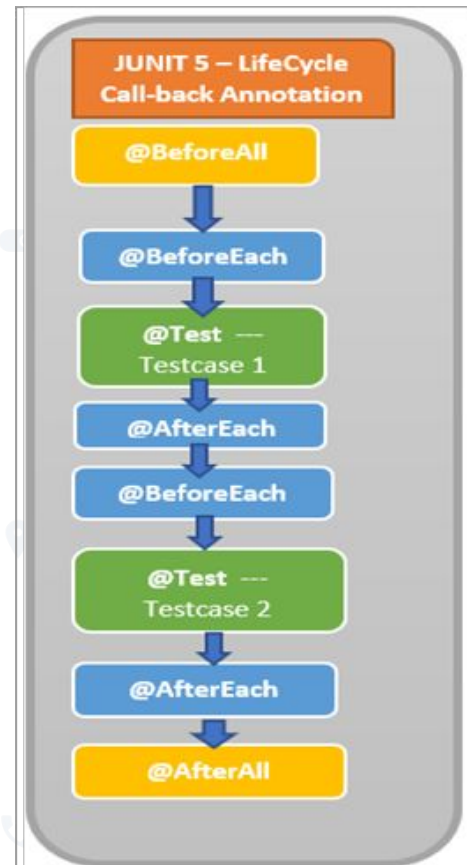
```
Bean findBean(int id){  
    if(foundBeanInDatabase){  
        return bean;  
    } else{  
        return null;  
    }  
}
```




JUnit 5 Test LifeCycle



JUNIT 4 ANNOTATION	JUNIT 5 ANNOTATION
@Before	@BeforeEach
@After	@AfterEach
@BeforeClass	@BeforeAll
@AfterClass	@AfterAll
@Test	@Test



JUnit 5 Annotations	Descriptions
<code>@Test</code>	Declares a test method
<code>@TestFactory</code>	Denotes a method is a test factory for dynamic tests in JUnit 5
<code>@DisplayName</code>	Define custom display name for a test class or test method
<code>@BeforeEach</code>	Denotes that the annotated method will be executed before each test method(annotated with <code>@Test</code>) in the current class.
<code>@AfterEach</code>	Denotes that the annotated method will be executed after each test method (annotated with <code>@Test</code>) in the current class.
<code>@BeforeAll</code>	Denotes that the annotated method will be executed before all test methods in the current class.
<code>@AfterAll</code>	Denotes that the annotated method will be executed after all test methods in the current class.
<code>@Nested</code>	Denotes that the annotated class is a nested, non-static test class
<code>@Tag</code>	Declare tags for filtering tests.
<code>@Disable</code>	Is used to disable a test class or method.
<code>@ExtendWith</code>	Is used to register custom extensions in JUnit 5
<code>@RepeatedTest</code>	Is used to repeat tests



```
class JUnit5Test {

    @Test
    void helloJUnit5() {
        assertEquals(10, 5+5);
    }
}
```

```
class JUnit5Test {

    @ParameterizedTest
    @ValueSource(strings = { "cali", "bali", "dani" })
    void endsWithI(String str) {
        assertTrue(str.endsWith("i"));
    }
}
```

```
class JUnit5Test {

    @RepeatedTest(value = 5, name = "{displayName} {currentRepetition}/{totalRepetitions}")
    @DisplayName("RepeatingTest")
    void customDisplayName(RepetitionInfo repInfo, TestInfo testInfo) {
        int i = 3;
        System.out.println(testInfo.getDisplayName() +
            "-->" + repInfo.getCurrentRepetition()
        );

        assertEquals(repInfo.getCurrentRepetition(), i);
    }
}
```

```
@Tag("smoke")
class JUnit5Test {

    @Test
    @Tag("login")
    void validLoginTest() {

    }

    @Test
    @Tag("search")
    void searchTest() {

    }
}
```

```
@Disabled
class DisabledClassDemo {

    @Test
    void testWillBeSkipped() {

    }
}
```

```
@Disabled
@Test
void testWillBeSkipped() {

}
```



```
@Test
public void testAssertTrue() {
    assertTrue(true); // will pass
    assertTrue(false); // will fail
}
```

```
@Test
public void testAssertFalse() {
    assertFalse(false); // will pass
    assertFalse(true); // will fail
}
```

```
@Test
public void testAssertNull() {
    assertNull(null);
    assertNull("test"); // will fail
}
```

```
@Test
public void testAssertNonNull() {
    assertNotNull("test");
    assertNotNull(null); // will fail
}
```

```
@Test
public void testAssertEquals() {
    String actual = "1";
    String expected = "1";

    assertEquals(expected, actual);
}
```

```
@Test
public void testAssertNotEquals() {
    String actual = "1";
    String expected = "2";

    assertNotEquals(expected, actual);
}
```

```
assertAll(
    () -> assertEquals("1", "1"),
    () -> assertEquals("1", "2"),
    () -> assertTrue(1 == 2)
);
```

```
@Test
public void testAssertEquals() {
    String actual = "1";
    String expected = "1";

    assertEquals(expected, actual);
}

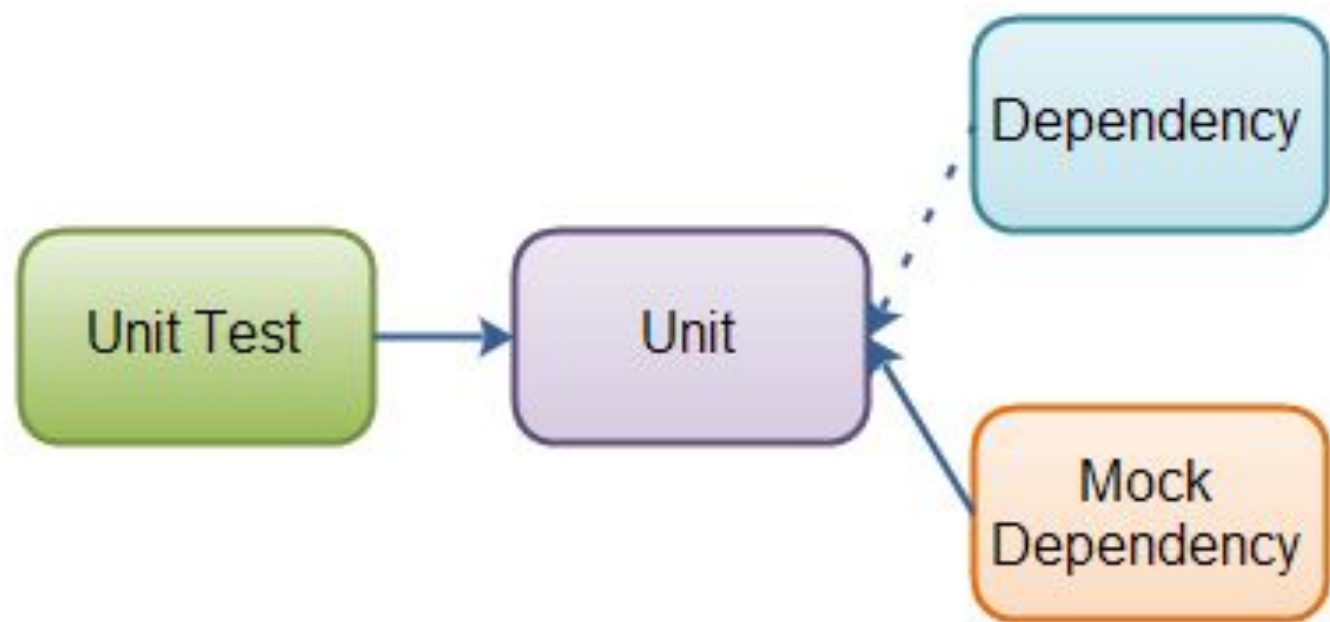
@Test
public void testAssertNotEquals() {
    String actual = "1";
    String expected = "2";

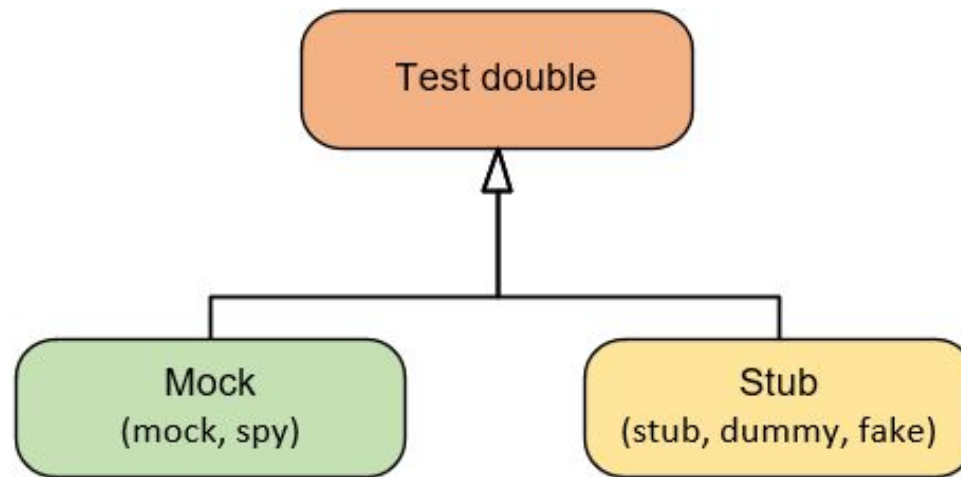
    assertNotEquals(expected, actual);
}
```

```
@Test
void testAssertDoesNotThrowException() {
    assertDoesNotThrow(() -> {
        int[] arr = new int[1];
        arr[0] = 0;
        // does not throw any exception
    });
}
```

```
@Test
void testAssertThrows() {
    assertThrows(IndexOutOfBoundsException.class, () -> {
        int[] arr = new int[1];
        arr[0] = 0;

        // will throw IndexOutOfBoundsException
        int value = arr[1];
    });
}
```

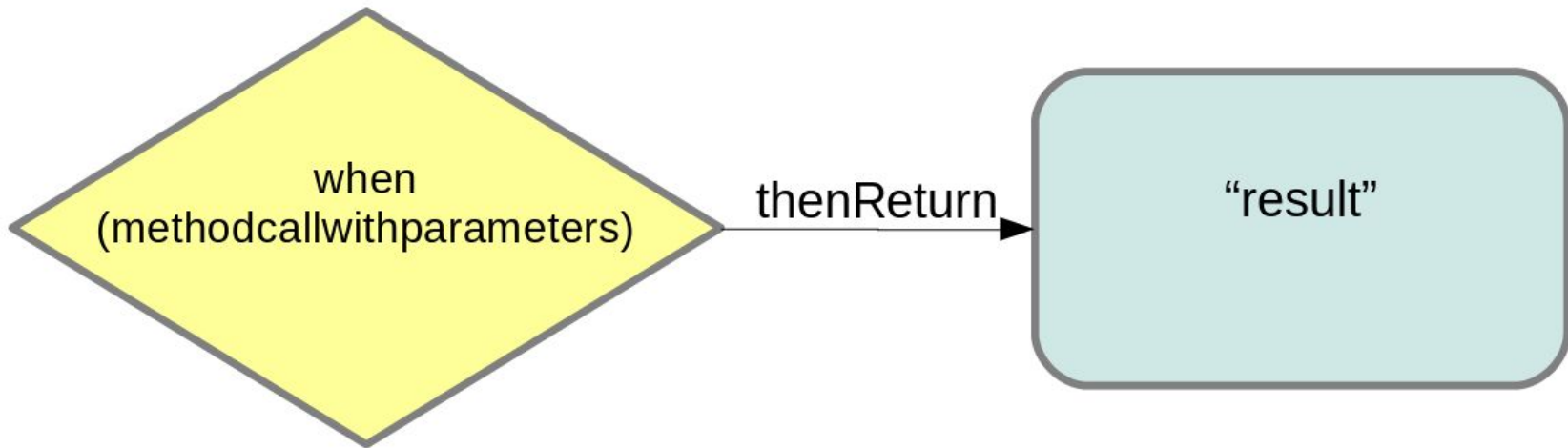




Різниця між цими двома видами зводиться до наступного:

Mocks — це динамічні обгортки для залежностей, що використовуються в тестах. Вони запрограмовані з очікуванням того, який саме метод викликається, у якій послідовності залежність повинна отримувати від тестованої системи (SUT), щоб SUT вважався працюючим правильно.

Stub — це рукописні класи, які імітують поведінку залежності, але роблять це за допомогою значних скорочень. Наприклад, заглушка для сховища замовлень може зберігати замовлення в пам'яті та повертати ці замовлення в результаті пошукових операцій замість запиту до реальної бази даних. Іншим прикладом може бути заглушка шлюзу електронної пошти, яка записує всі електронні листи, надіслані через нього. Заглушка також може бути дурною і мати лише мінімальну реалізацію, необхідну для задоволення інтерфейсу, який вона обертає.



```
@Mock
Database databaseMock;

@Test
void ensureMockitoReturnsTheConfiguredValue() {

    // define return value for method getId()
    when(databaseMock.getId()).thenReturn(42);

    Service service = new Service(databaseMock);
    // use mock in test...
    assertEquals(service.toString(), "Using database with id: 42");
}
```