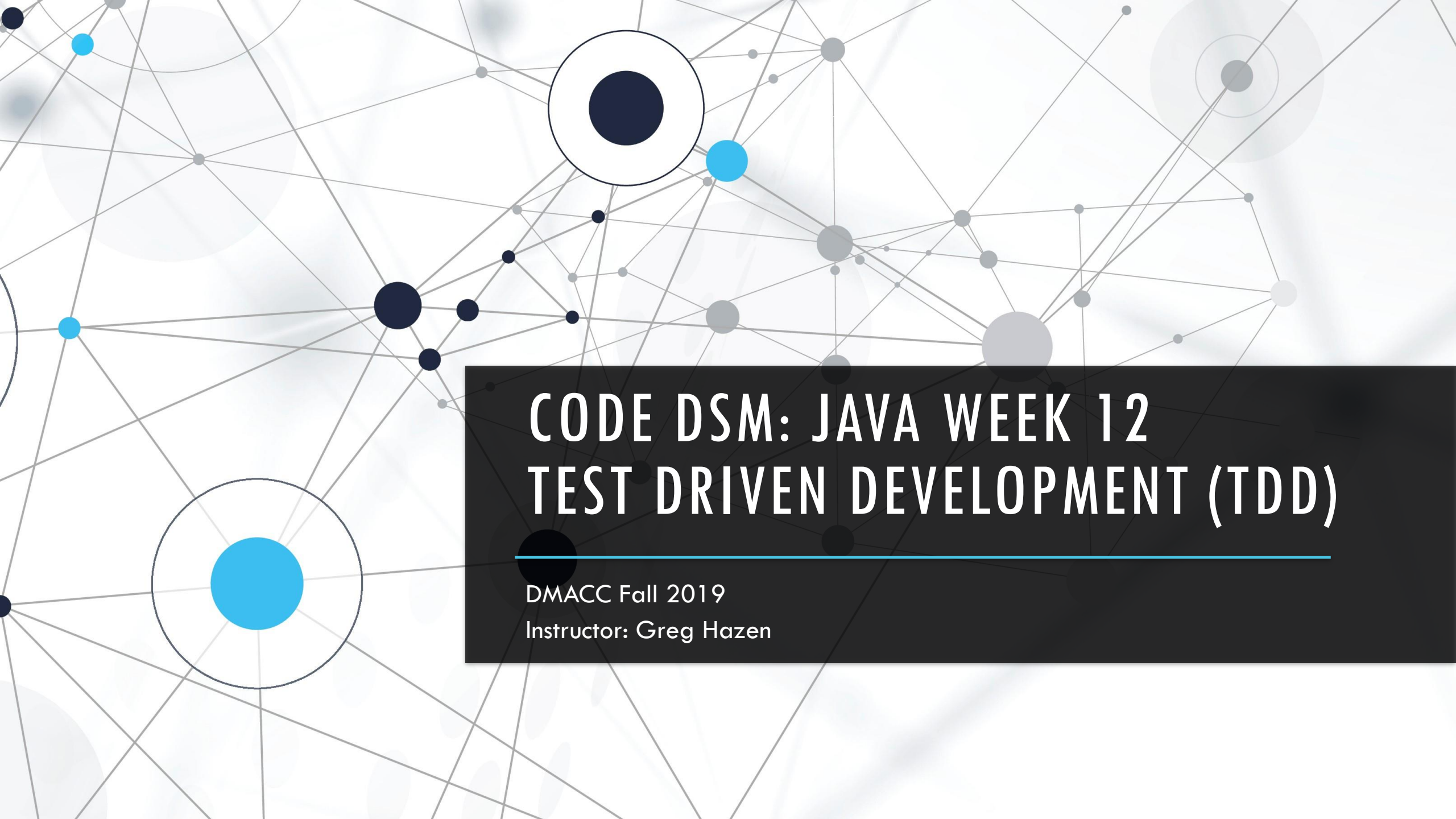


CODE DSM: JAVA WEEK 12 TEST DRIVEN DEVELOPMENT (TDD)

DMACC Fall 2019

Instructor: Greg Hazen

AGENDA

UNIT TESTING

ISOLATING UNITS

TEST DRIVEN DEVELOPMENT

UNIT TESTING

Unit testing is a way to insure the individual units or components of a piece of software function according to a set of expectations and assertions

The objective is to isolate a section of code and verify its correctness

A ‘unit’ in Java is generally a

- Public method or constructor
- Function
- Any other set of concise instructions that produce a behavior

UNIT TESTING

Unit tests

- Detect bugs early in the development lifecycle
- Act as documentation for application code
- Can be used to prove extensibility and reuse of objects

Testing is an Investment

JUNIT

JUnit is a library for writing unit tests in Java

Can be executed from an IDE

Supports

- Test methods with the ability to assert expectations
- Set-up methods before a test is run
- Teardown methods after a test is run
- Custom test runners to support more complex situations (such as Spring)

JUNIT

All test methods in Junit 4 begin with the '@Test' annotation

Junit 4 Assertions

- Utility methods provided by JUnit through the Assert class
- Generally are written in an assert -> expected -> actual format

Common assertions

- assertTrue / assertFalse
- assertNull / assertNotNull
- assertEquals
- assertArrayEquals
- fail

JUNIT

Example test method:

```
@Test
public void whenAssertingEquality() {
    String expected = "Nate";
    String actual = "John";

    assertEquals(expected, actual);
}
```

JUNIT

Example setup method:

```
Object classUnderTest;

@Before
public void setup() {
    classUnderTest = new Object();
}
```

JUNIT 4 VS JUNIT 5

Last time we only saw Junit 4

Junit 5

- Allows multiple tests running at once
- Supports Java 8 features (e.g. functions)

JUNIT 4 VS JUNIT 5

@Before renamed to @BeforeEach

@After renamed to @AfterEach

@BeforeClass renamed to @BeforeAll

@AfterClass renamed to @AfterAll

JUNIT 4 VS JUNIT 5

With Java 8 support, we can now lazy create messages only when they're needed

```
@Test
public void shouldFailBecauseTheNumbersAreNotEqual_lazyEvaluation() {
    Assertions.assertTrue(
      2 == 3,
      () -> "Numbers " + 2 + " and " + 3 + " are not equal!");
}
```

JUNIT 4 VS JUNIT 5

Junit 4

```
@Test(expected = Exception.class)
public void shouldRaiseAnException() throws Exception {
    // ...
}
```

JUNIT 4 VS JUNIT 5

Junit 5

```
@Test
public void shouldRaiseAnException() throws Exception {
    Assertions.assertThrows(Exception.class, () -> {
        //...
    });
}
```

JUNIT 4 VS JUNIT 5

Lots of other differences and improvements

There'll be a link in course contents this week

TEST DRIVEN DEVELOPMENT

1. Write a test
2. Write the implementation to make the test pass
3. Clean up the implementation

TEST DRIVEN DEVELOPMENT

1. Write a test (**Red**)
2. Write the implementation to make the test pass (**Green**)
3. Clean up the implementation (**Refactor**)

If you refactor before it's green, how do you know that you haven't broken something

TEST DRIVEN DEVELOPMENT

If your start all code with a test, all code will be covered by a test

Obvious, but that's the point!

TEST DRIVEN DEVELOPMENT

1. Your first test is to call your method that doesn't exist yet (Red)
2. You create the method but don't implement it (Green)

TEST DRIVEN DEVELOPMENT

1. Update your test to assert the return value of the method (Red)
2. Add the implementation to return that value (Green)
3. Clean up your implementation (Refactor)

TEST DRIVEN DEVELOPMENT

Try to

- Write the least amount of test for
- The least amount of implementation

By writing the least amount, you're less likely to miss test scenarios

ISOLATING UNITS IN JUNIT

We want to test small units of code

What if the unit depends on other objects?

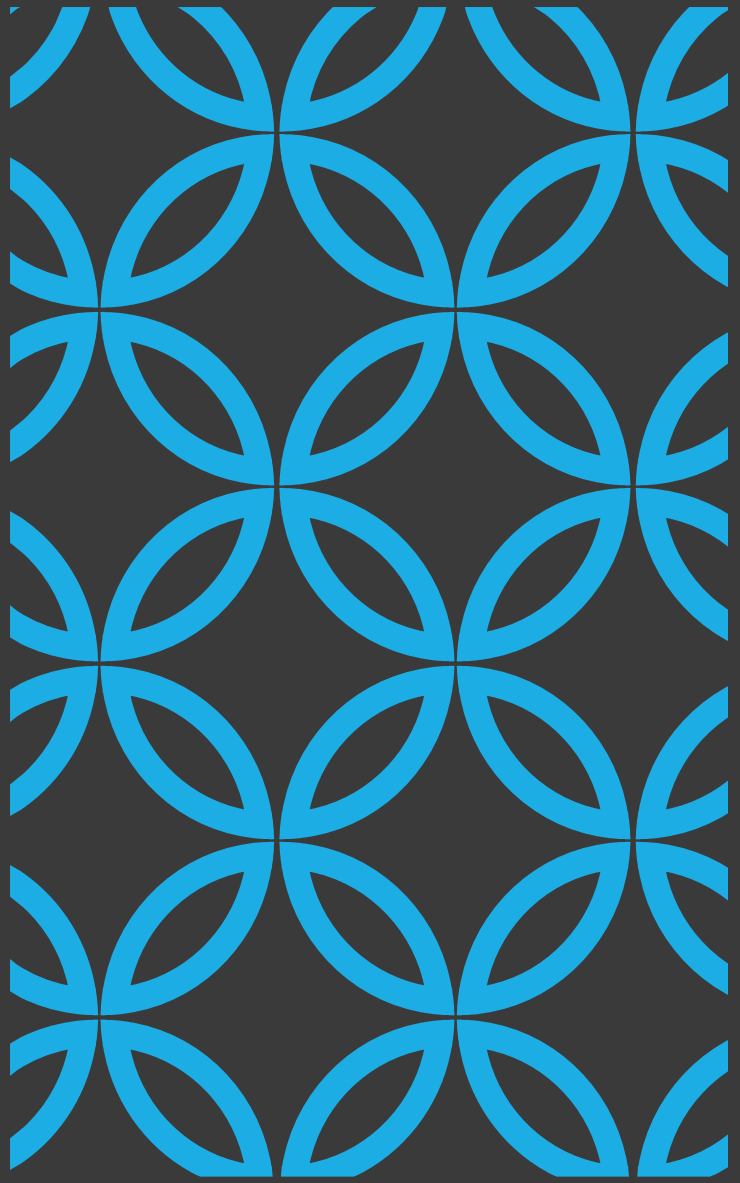
We need a way to make the other objects always behave the same way to isolate their complexity from our unit's complexity

ISOLATING UNITS IN JUNIT

Patterns of isolation

- Stub – create a custom implementation
- Fake – a fake version that always responds the same way
- Mock – use a library to create expected behaviors

CLONE THE LECTURE NOTES



LET'S SEE IT IN INTELLIJ

Given an interface for a database and a naming service, let's

- Stub
- Fake

MOCKING

Every method in a **mock** returns default values

The implementation of a mock's methods aren't used

```
Restaurant restaurant = mock(Restaurant.class);
restaurant.getMenu(); // returns null
```

MOCKING

You can tell a **mock** to return different values

```
Restaurant restaurant = mock(Restaurant.class);
Menu expected = new Menu();
doReturn(expected).when(restaurant).getMenu();
restaurant.getMenu(); // returns expected
```

MOCKING

Why use a mock? To isolate units of code

If I want to test unit A, but it uses unit B

- Mock unit B to return expected values
- Now you only need to test unit A's code

MOCKING

Why use a mock? To isolate units of code

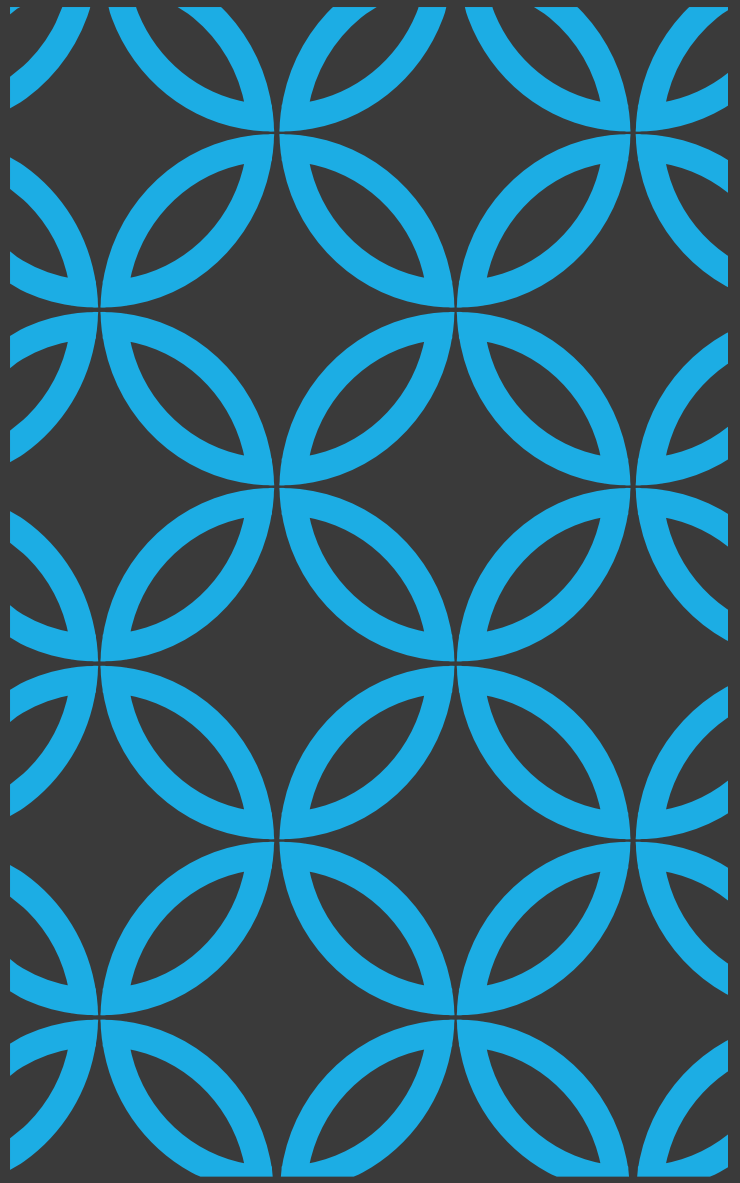
For example, if I'm testing that a restaurant returns a certain menu

- Mock the menu so that we don't have to worry about reading files and other complex menu code
- Write the test to simply ask the restaurant for the menu

MOCKING

Mockito is a popular mocking library

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>3.2.4</version>
    <scope>test</scope>
</dependency>
```



LET'S SEE IT IN INTELLIJ

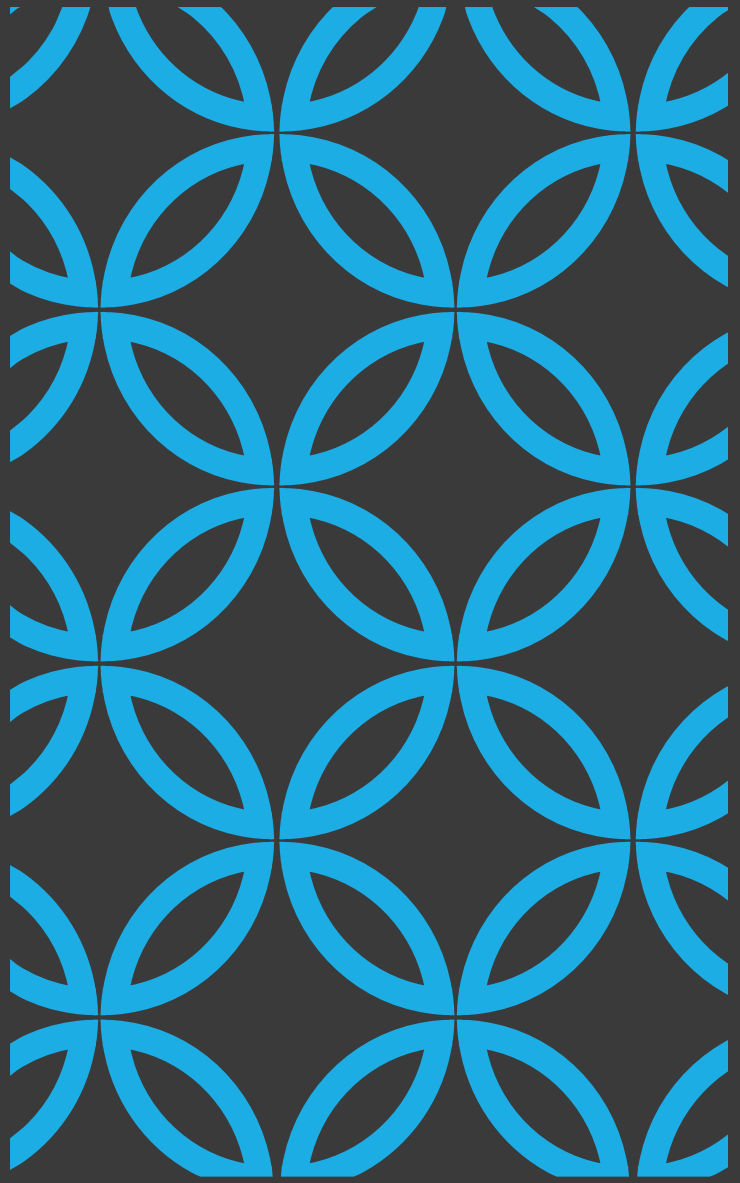
Given an interface for a database, let's

- Mock

MOCKING

We may also want to verify certain methods were called

```
Database database = mock(Database.class);
…
verify(database).connect();
```



LET'S SEE IT IN INTELLIJ

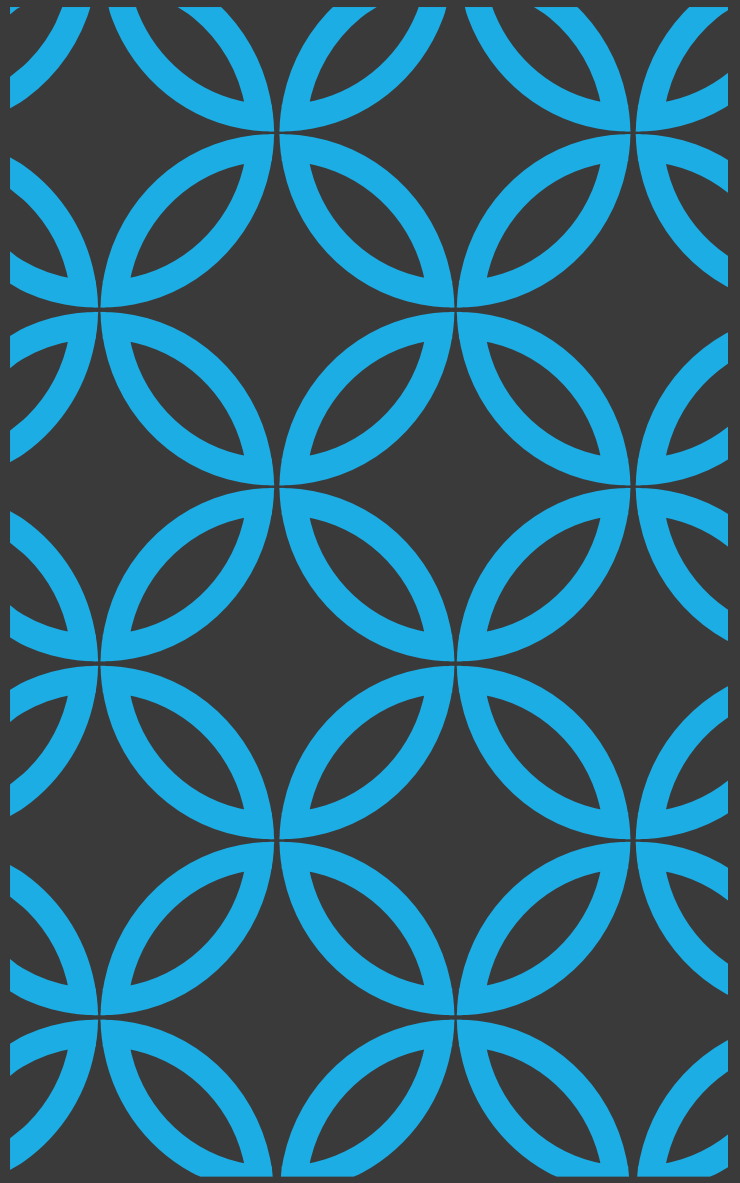
Verify

- Database connection was created
- Query was called
- Database connection was closed

MOCKING

We may also want to test what happens when exceptions occur

```
Database database = mock(Database.class);
doThrow(new RuntimeException())
        .when(database).getNextResults();
```



LET'S SEE IT IN INTELLIJ

Verify

- Database connection was closed even if there's an exception

ISOLATING UNITS IN JUNIT

When to use each pattern

- Stub – if you have limited scenarios or can't use a mocking library
- Fake – if you always return a single value
- Mock – for most testing scenarios where objects depend on each other

FINAL PROJECT

Hangman!

- Now available in Blackboard Course Contents under the Projects folder

Due Sunday January 19th at Noon

- *Some* in-class time to work on the project each week

FINAL PROJECT

In-class time now to design your classes

- **Before you leave**, send me your design in Slack!

Include your **best guess** of all

- Classes
- Methods in each class

ASSIGNMENT

No Homework or quiz!

SEE YOU JANUARY 10TH!

Don’t forget to ask question early in the week

REFERENCES

- Baeldung.com. (2019). Migrating from JUnit 4 to JUnit 5. [online] Available at: https://www.baeldung.com/junit-5-migration [Accessed 23 Dec. 2019].