Test-Driven Development

Types of Testing

- **Unit tests:** Validate individual components in isolation, ensuring each part functions correctly.
- **Integration tests:** Verify interactions between components, confirming proper system functioning.
- **Functional tests:** Assess application functionality from a user's perspective, including UI, APIs, and databases.
- **Regression tests:** Re-run existing tests on modified code to catch new or re-introduced issues.
- **Performance tests:** Evaluate application efficiency, speed, and scalability under various conditions.

Types of Testing

- **Security tests:** Identify vulnerabilities and threats through penetration testing, static/dynamic code analysis.
- **Usability tests:** Examine user experience, measuring ease of interaction with the application.
- **Compatibility tests:** Ensure application works across platforms, devices, browsers, and operating systems.
- **Localization tests:** Verify application adaptation for different languages, regions, and cultures.

Before the production

INTEGRATION TEST

SYSTEM TEST

ACCEPTANCE TEST

Complicated Code (Example)

Before making any changes, give me a bit of time to figure out what it is doing first !

What is wrong with this big class?

How to make a complicated program safe when :

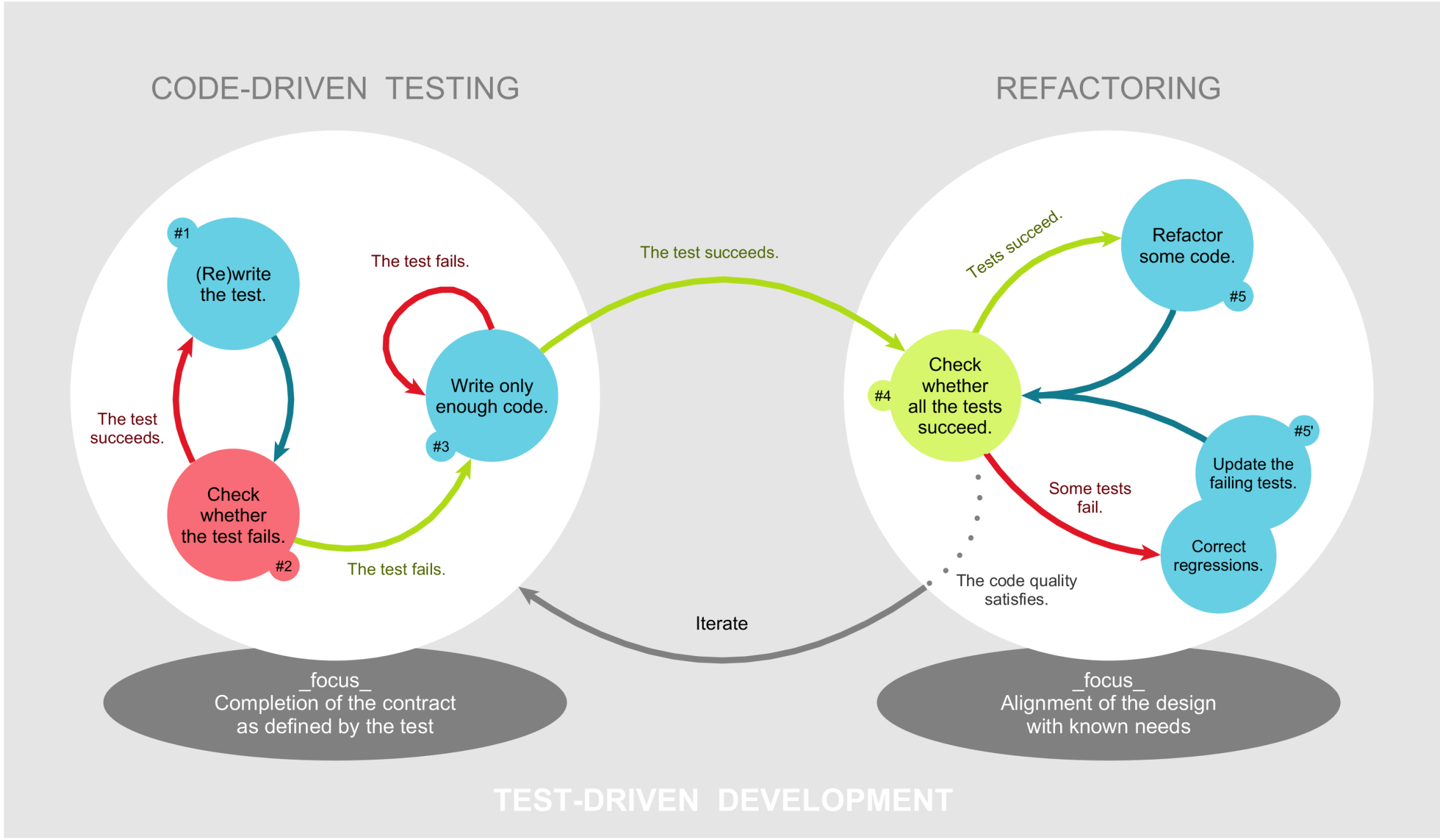
- Refactoring existing code
- Fixing bugs

The most effective approach to ensuring the safety of a complex program is to initiate thorough testing from the very outset. As the complexity builds up, there is almost no way you to guarantee your program is safe or you will have to pay a high price.


Introducing TDD

TDD Cycle

- Add a test
- Run all tests. The new test *should fail* for expected reasons
- Write the simplest code that passes the new test
- All tests should now pass
- Refactor as needed, using tests after each refactor to ensure that functionality is preserved
- Repeat the process


CODE-DRIVEN TESTING
REFACTORING
#1
(Re)write the test.
The test fails.
The test succeeds.
Write only enough code.
#3
The test succeeds.
Check whether the test fails.
#2
The test fails.
Tests succeed.
Refactor some code.
#5
#4
Check whether all the tests succeed.
#5'
Update the failing tests.
Some tests fail.
Correct regressions.
The code quality satisfies.
Iterate
focus
Completion of the contract as defined by the test
focus
Alignment of the design with known needs
TEST-DRIVEN DEVELOPMENT

TDD Cycle – Example

```
1 public class ValidateISBN {
2     public boolean checkISBN(int i) {
3         return false;
4     }
5 }
```

TDD Cycle - Add a Test and make it fail first

```java
import org.junit.jupiter.api.Test;

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

public class ValidateISBNTest {

    @Test
    public void checkAValidISBN() {
        ValidateISBN validator = new ValidateISBN();
        boolean result = validator.checkISBN(140449116);
        assertTrue(result);
    }
}
```

TDD Cycle – ISBN

ISBN	0	1	4	0	4	4	9	1	1	6
x	10	9	8	7	6	5	4	3	2	1
=	0	9	32	0	24	20	36	3	2	6

Total = 0 + 9 + 32 + 0 + 24 + 20 + 36 + 3 + 2 + 6 = 132 ÷ 11 = **12**

ISBN	0	1	4	0	4	4	9	1	1	7
x	10	9	8	7	6	5	4	3	2	1
=	0	9	32	0	24	20	36	3	2	7

Total = 0 + 9 + 32 + 0 + 24 + 20 + 36 + 3 + 2 + 7 = 133 ÷ 11 = **12.0909**

TDD Cycle - Implement it fast and simple

```
1  public class ValidateISBN {
2      public boolean checkISBN(String s) {
3          int t = 0;
4
5          for (int i = 0; i < 10; i++) {
6              t += s.charAt(i) * (10 - i);
7          }
8          if (t % 11 == 0) {
9              return true;
10         } else {
11             return false;
12         }
13     }
14 }
```

TDD Cycle - Refactor the code

```
3  public class ValidateISBN {
4
5      private static final int ISBN_LENGTH = 10;
6
7      public boolean checkISBN(String isbn) {
8          if (isbn == null || isbn.length() != ISBN_LENGTH) {
9              return false;
10         }
11
12         int sum = 0;
13         for (int i = 0; i < ISBN_LENGTH; i++) {
14             int digit = Character.getNumericValue(isbn.charAt(i));
15             if (digit < 0 || digit > 9) {
16                 return false;
17             }
18             sum += digit * (ISBN_LENGTH - i);
19         }
20
21         return sum % 11 == 0;
22     }
23 }
```

TDD Cycle - Repeat

- Adding more logics like checking 13 digits.

TDD Test vs Traditional Test

TDD integrates testing into the development process.
While Traditional Testing tests software after development.
TDD emphasizes unit testing and continuous feedback.
While Traditional Testing covers a broader range of testing levels.
TDD promotes better code design and maintainability.
While Traditional Testing focuses on overall correctness and quality attributes.

Benefits of TDD

- **Catch early and fewer bugs** TDD catches errors **early,** leading to accurate and reliable code.
- **Accelerated development:** TDD reduces **debugging time** by detecting and fixing issues early in the process.
- **Simplified maintenance:** Tests serve as documentation, making code updates and refactoring easier.
- **Increased test coverage:** TDD typically results in higher test coverage compared to traditional development methodologies.

Benefits of TDD(Cont.)

- **Effective collaboration:** TDD fosters clear communication, efficient teamwork, and shared project goals.
- **Modularity and design:** Writing tests first promotes modular, decoupled code that's easier to maintain. Using interface, mock...
- **Early integration issue detection:** TDD identifies integration problems early, making them simpler to address.
- **Agile compatibility:** TDD supports Agile principles, emphasizing iterative, adaptive development with fast feedback.

Drawbacks of TDD

- **Upfront time investment**: Writing tests first requires additional time upfront, which can be a challenge for teams with tight deadlines or limited resources.
- **Complexity**: TDD can be more complex than other development approaches, as it requires developers to write tests, implement code, and refactor continuously.
- **Test maintenance**: Over time, the number of tests in a project can grow, leading to increased maintenance costs. Developers must ensure that tests remain relevant and useful as the codebase evolves.

Drawbacks of TDD(Cont.)

- **Hard to write**: Writing tests can be more complex than writing code, which can make it difficult to write and understand tests(Complecated Object setup Ex.Hibernate).
- **Tests are dependent on external dependencies**: This can make tests more brittle and difficult to maintain.(Ex.Database availability)

TDD Best Practices

- **Write tests that are independent**: Tests should be independent of each other so that they can be run in any order
- **Avoid functional complexity**: Focus on **one functionality or feature** at a time
- **Repeatable and consistent**: Tests should produce the same test result every time in order to run the test suite pass without confusion.

TDD Best Practices (Not Repeatable Example)

```
@Test
public void notRepeatableTest(){
    Date today = new Date();
    FoodCourt fc = new FoodCourt();
    assertTrue (fc.isOpen(today));
}
```

TDD Best Practices (Not Repeatable Example Cont.)

```
@Test
public void betterTest(){
    Date today = new GregorianCalendar
    (2023,4,6,13,9,7).getTime();
    FoodCourt fc = new FoodCourt();
    assertTrue (fc.isOpen(today));
}
```

TDD Best Practices (Cont.)

- **Write tests that are fast**: Tests should be fast so that they can be run frequently, some time-consuming tests like database connection, use mock.
- **Test business logic not method:** Test focus on business ,not method, sometimes one method contains one business logic while some time are not.

Practices to avoid in TDD

- Avoid testing implementation details.
- Avoid testing private methods.
- Avoid Mega test.
- Avoid system performance / timing test
- Avoid testing inside the black box
- Avoid slow running tests (use mocks to speed them up)
- Avoid dependencies between test cases Test

TDD Questions

- How to hand a test which has a lot of combinations like a multiple dropdown list (example)
- How to hand a test to a big data set? The way to collect the data sample (example)

Techniques Applied in TDD

- Using Stub
- Using Mock
- Programming to interface
- Using Dependency Injections
- Using Tools like Junit5 Mockito

Techniques in TDD (Interface)

```
1  public interface DataService {
2      int[] retrieveAllData();
3  }
4
```

Techniques in TDD (DI)

```
1  public class SomeBusinessImpl {
2      private DataService dataService;
3
4      public SomeBusinessImpl(DataService dataService) {
5          this.dataService = dataService;
6      }
7
8      int findTheGreatestFromAllData() {
9          int[] data = dataService.retrieveAllData();
10         int greatest = Integer.MIN_VALUE;
11
12         for (int value : data) {
13             if (value > greatest) {
14                 greatest = value;
15             }
16         }
17
18         return greatest;
19     }
20 }
```

Techniques in TDD (Mock)

```
1  import org.junit.Test;
2  import static org.junit.Assert.assertEquals;
3  import static org.mockito.Mockito.mock;
4  import static org.mockito.Mockito.when;
5
6  public class SomeBusinessMockTest {
7      @Test
8      public void testFindTheGreatestFromAllData() {
9          DataService dataServiceMock = mock(DataService.class);
10         when(dataServiceMock.retrieveAllData()).thenReturn(new int[] {24,
           15,3});
11
12         SomeBusinessImpl businessImpl = new SomeBusinessImpl(
           dataServiceMock);
13         int result = businessImpl.findTheGreatestFromAllData();
14
15         assertEquals(24, result);
16     }
17 }
```

Techniques in TDD (Stub)

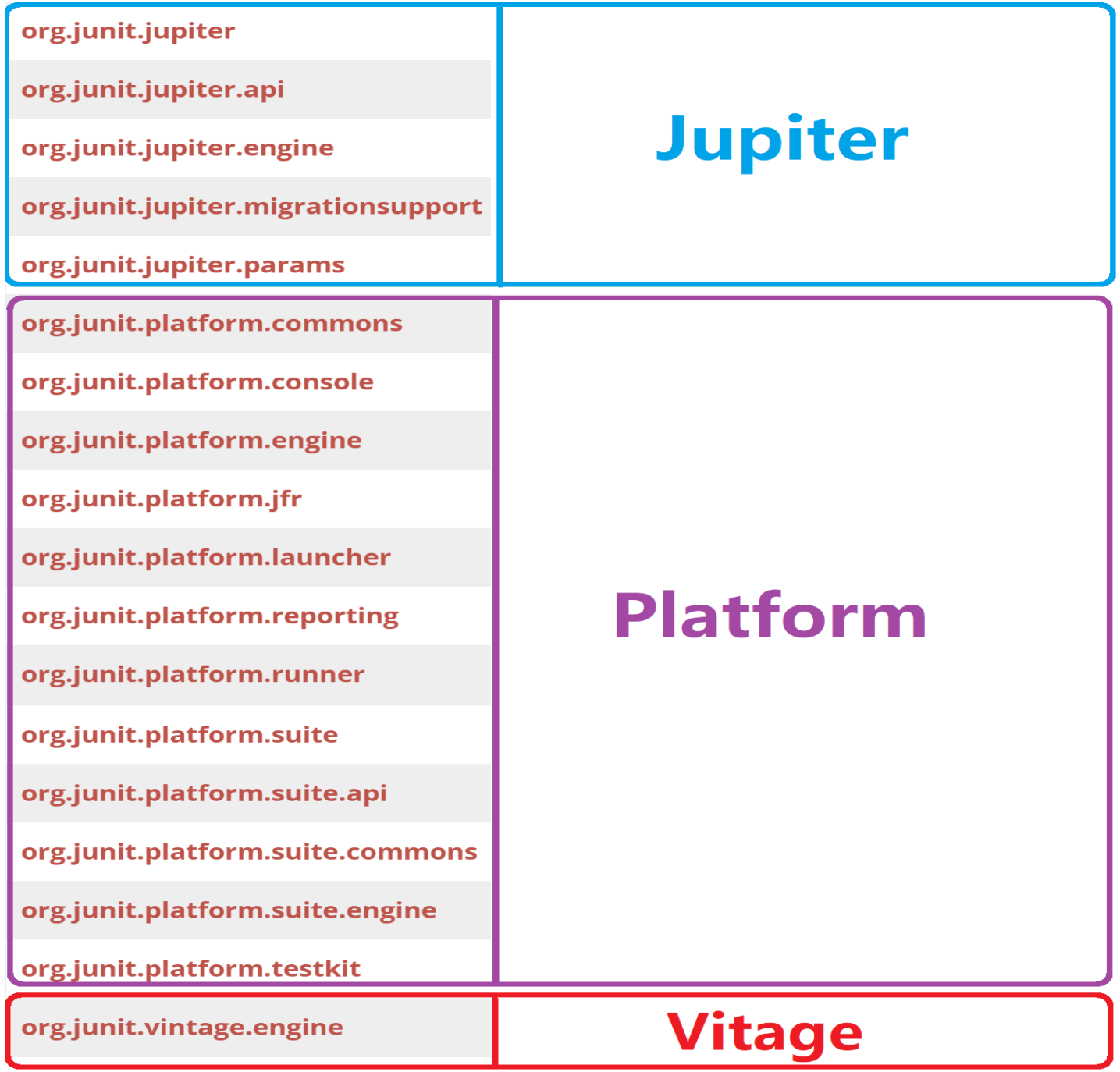
```java
public class DataServiceStub implements DataService {
    @Override
    public int[] retrieveAllData() {
        // Return a fixed set of data for testing purposes
        return new int[] {5, 9, 3, 12};
    }
}
```

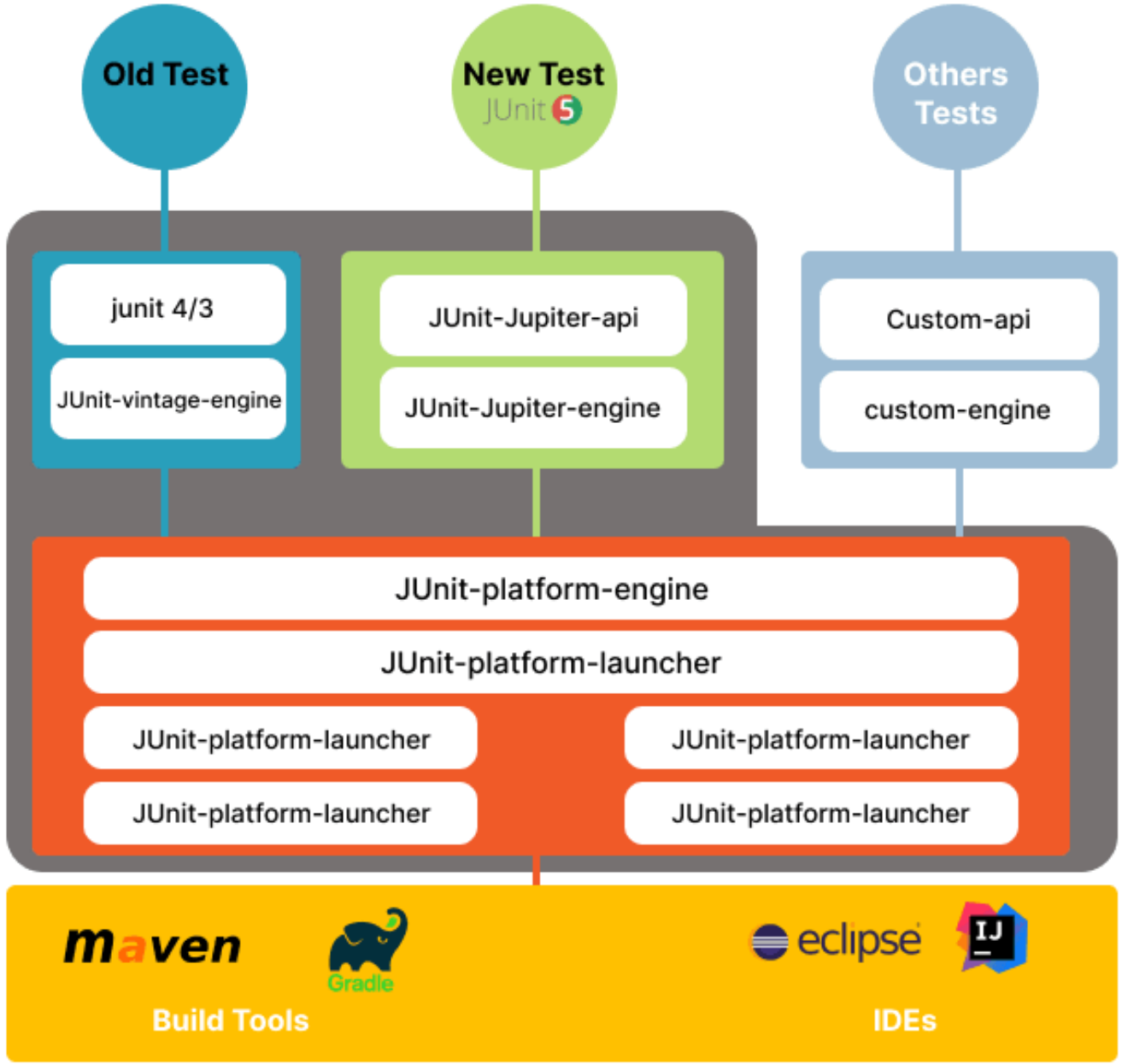
TDD Tools and Framworks

- Junit 5
- Mockito
- Selenium

Introducing Junit 5

- JUnit is a popular open-source framework used for unit testing in Java applications.
- JUnit 5 is the latest version of the framework, released in 2017, which provides many new features and improvements over JUnit 4.
- JUnit 5 is composed of several modules, including JUnit Platform, JUnit Jupiter, and JUnit Vintage.


org.junit.jupiter
org.junit.jupiter.api
org.junit.jupiter.engine
org.junit.jupiter.migrationsupport
org.junit.jupiter.params
Jupiter
org.junit.platform.commons
org.junit.platform.console
org.junit.platform.engine
org.junit.platform.jfr
org.junit.platform.launcher
org.junit.platform.reporting
org.junit.platform.runner
org.junit.platform.suite
org.junit.platform.suite.api
org.junit.platform.suite.commons
org.junit.platform.suite.engine
org.junit.platform.testkit
Platform
org.junit.vintage.engine
Vitage


Old Test
New Test
JUnit 5
Others Tests
junit 4/3
JUnit-vintage-engine
JUnit-Jupiter-api
JUnit-Jupiter-engine
Custom-api
custom-engine
JUnit-platform-engine
JUnit-platform-launcher
JUnit-platform-launcher
JUnit-platform-launcher
JUnit-platform-launcher
JUnit-platform-launcher
maven
Gradle
Build Tools
eclipse
IJ
IDEs

Junit 5 Advancement

- **Modular architecture**: JUnit 5 has a modular architecture that allows you to use only the parts of the framework that you need, and makes it easier to extend and customize the framework.
- **Improved assertions**: JUnit 5 includes a new set of assertion methods, such as assertAll() and assertTimeout(), that make it easier to write and maintain tests.
- **More powerful test annotations**: JUnit 5 introduces new test annotations, such as @BeforeEach, @AfterEach, and @BeforeAll, that allow you to control the lifecycle of your tests and set up and tear down resources more easily.

Junit 5 Key Annotations

- **@Test** denotes a test method, supporting parameterized tests in JUnit 5 for multiple inputs. Run the same method repeatedly with varying values.
- **@BeforeEach and @AfterEach** will run before and after individual tests, so it can rest the conditions for the next one.
- **@BeforeAll and @AfterAll** run only once before and after the tests, and can be used to facilitate one-time test environment preparation and resource release etc.

Configure Junit 5 to Run

To start using JUnit 5, we can add the following dependency to our *pom.xml*:

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.1.0</version>
    <scope>test</scope>
</dependency>
```

or to our *build.gradle* file:

```
testCompile('org.junit.jupiter:junit-jupiter-api:5.2.0')
testRuntime('org.junit.jupiter:junit-jupiter-engine:5.2.0')
```

Demo:

- @Test,@BeforeAll,@AfterAll,@BeforeEach,@AfterEach

Junit 5 Key Annotations (Cont.)

- **@DisplayName** allows custom naming for test methods/classes, enhancing debugging and readability with descriptive names.
- **@Disabled** temporarily disables failing or unimplemented tests for a method or class, allowing focused execution.
- **@Tag** assigns tags to methods/classes, enabling test grouping and selective running based on tag categories.

Mocks

In object-oriented programming, mock objects are simulated objects that mimic the **behaviour** of real objects in **controlled ways**, most often as part of a software testing initiative. - Wikipedia

Mock vs Stub

Stub:

- A stub object provides predefined responses to method calls.
- Stubs are mainly used to simulate the behavior of a collaborator (a dependent object) by returning specific values or exceptions when its methods are called.
- Stubs don't track how their methods are called or how many times they are called.

Mock vs Stub (Cont.)

Mock:

- A mock object can be dynamically configured to behave in a certain way, similar to a stub, but it also records the interactions it has with other objects.
- Mocks are used to both simulate the behavior of a collaborator and to verify that the component under test interacts with the collaborator correctly (i.e., the correct methods are called with the correct parameters and the expected number of times).
- Mocks can be created and configured using mocking frameworks such as Mockito, EasyMock, or JMock.

Mock vs Stub(Cont.)

```
1 public class UserStub implements UserService {
2     @Override
3     public User getUserById(int id) {
4         return new User(id, "John Doe");
5     }
6 }
```

```
1 import static org.mockito.Mockito.*;
2
3 UserService userServiceMock = mock(UserService.class);
4 when(userServiceMock.getUserById(1)).thenReturn(new User(1, "John Doe"));
5
```

What to Mock ?

Mocks Slow Process

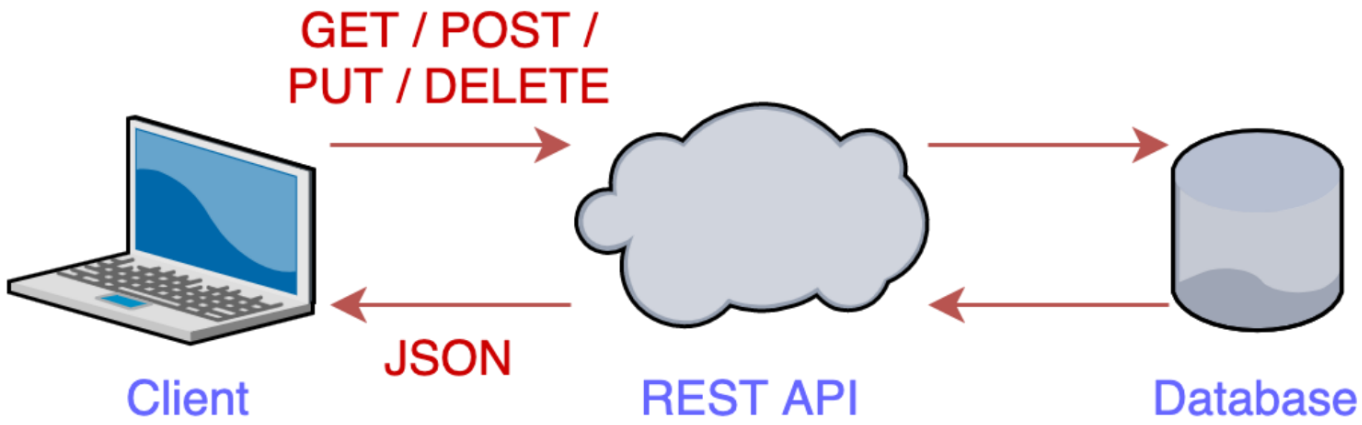
Ex. Database access: When testing a class that interacts with a database, you can create a mock object for the data access layer. This allows you to simulate database responses, avoid the overhead of actual database operations, and focus on the logic in the class being tested.

Mock Database Access (Ex.)

```
1  public interface UserRepository {
2      User getUserById(int userId);
3  }
4
5  public class UserService {
6      private UserRepository userRepository;
7
8      public UserService(UserRepository userRepository) {
9          this.userRepository = userRepository;
10     }
11
12     public String getUserName(int userId) {
13         User user = userRepository.getUserById(userId);
14         return user.getName();
15     }
16 }
```

Mocks External Resources

Ex. REST API calls: When testing a class that makes HTTP requests to an external REST API, you can mock the HTTP client or the service class that handles API calls. This helps you isolate the class under test, simulate various API responses, and avoid reliance on the actual API.



Mock REST API calls (Ex.)

```java
public interface WeatherApiClient {
    WeatherData getWeatherData(String location);
}

public class WeatherService {
    private WeatherApiClient weatherApiClient;

    public WeatherService(WeatherApiClient weatherApiClient) {
        this.weatherApiClient = weatherApiClient;
    }

    public String getCurrentWeather(String location) {
        WeatherData data = weatherApiClient.getWeatherData(location);
        return data.getCurrentWeather();
    }
}
```

Introducing Mockito

- Mockito is a popular open-source Java testing framework that allows developers to create mock objects in unit tests.
- Mockito creates dummy objects that mimic the behavior of real objects in a controlled way, allowing developers to isolate the code being tested and focus on a specific piece of functionality.
- Mockito is easy to use and integrates well with other testing frameworks such as JUnit and TestNG.

Why Mockito

- **Test Readability:** Mockito enhances test readability and conciseness, aiding developers in understanding test purposes and behaviors, and reducing debugging time.
- **Test Isolation:** Mockito enables testing specific code by simulating external dependencies with mock objects, ensuring the focus remains on the functionality being tested.
- **Test Maintainability:** Mockito facilitates modifying mock object behavior, enhancing test adaptability for complex or evolving systems, and ensuring long-term test maintenance.

Why Mockito (Continues)

- **Test Coverage:** Mockito enables testing complex object interactions, which may be challenging with traditional methods, improving test coverage and code thoroughness.
- **Increased Productivity:** Mockito expedites efficient test creation, boosting developer productivity and fostering higher quality code delivery.

Key Features of Mockito

- **Mocking:** Mockito enables creating mock objects to simulate external dependencies, facilitating isolated functionality testing.
- **Stubbing:** Mockito allows defining mock object behavior, enabling predictable test scenarios and code functionality verification.
- **Verification:** Mockito's API supports interaction verification between objects, ensuring intended code operation and validating necessary interactions.

Key Features of Mockito(Continues)

- **Annotations:** Mockito offers annotations like @Mock and @InjectMocks, simplifying mock object creation and management in tests.
- **BDD Support:** Mockito supports Behavior-Driven Development testing, emphasizing system behavior definition through scenarios and outcomes.
- **Exception Handling:** Mockito enables testing code's error handling by supporting exception management in tests.

Key Annotations in Mockito

- **@Mock**: Creates a mock instance of a class or an interface. It initializes the mock object and is typically used as a field annotation in test classes.
- **@Spy**: Creates a spy **real instance** of a class. It is used to **partially** mock an object, allowing you to mock specific methods while calling real methods for others.
- **@Captor**: Defines an ArgumentCaptor object to capture the arguments of method calls on mock objects. This allows you to verify or assert the captured arguments.

@Captor example

```
1  @RunWith(MockitoJUnitRunner.class)
2  public class MyTestClass {
3
4      @Mock
5      private List<String> mockedList;
6
7      @Captor
8      private ArgumentCaptor<String> argCaptor;
9
10     @Test
11     public void shouldDoSomething() {
12         mockedList.add("one");
13
14         Mockito.verify(mockedList).add(argCaptor.capture());
15
16         assertEquals("one", argCaptor.getValue());
17     }
18 }
```

Key Annotations in Mockito(continues)

- **@InjectMocks**: Automatically injects mock objects into the class being tested. It creates an instance of the class and injects the mocked dependencies.
- **@ExtendWith(MockitoExtension.class)**: Registers MockitoExtension with JUnit 5. It provides integration between Mockito and JUnit 5, allowing you to use Mockito annotations in your tests.

Mockito Demos

What about UI test - Selenium

- **Cross-browser testing:** Selenium enables testing web applications across major browsers like Chrome, Firefox, Safari, and Internet Explorer for consistent performance.
- **Supports multiple languages:** Selenium accommodates various programming languages, including Java, Python, Ruby, and C#, providing flexibility for developers and testers.
- **Integration with testing frameworks:** Seamlessly integrating with frameworks like JUnit and TestNG, Selenium facilitates structured and organized test execution.

Selenium (continues)

- **Real browser interactions**: Selenium enables realistic browser simulations, including clicks, form fills, and navigation, crucial for testing web applications as users would.
- **Open-source:** Selenium is free and backed by a vast developer community, offering numerous online resources for learning and troubleshooting.

Selenium Demo

Project TDD Requirements

- **All business logic must be 100% covered by unit tests**
- **Unit tests must be thoroughly** test not just standard inputs but also boundary conditions, exceptional inputs, bad inputs, etc. If something CAN go wrong, you should have a test to catch it and ensure your code handles the bad eventuality.
- **Unit test classes must SHADOW business logic classes**. I.e. if you have a class User, you need a test class UserTest where the unit tests for User are implemented. If it's hard to find your unit tests they don't count.