

INSIDE UNIT TESTING

PRINCIPLES & PRACTICES



Doug Klugh



doug@CodeFlair.io



@DougKlugh



/Klugh

AGENDA

- ✓ THE GOALS OF SOFTWARE TESTING
- ✓ TEST AUTOMATION PYRAMID
- ✓ BASICS OF UNIT TESTING
- ✓ CORE TECHNIQUES
- ✓ AVOIDING TEST SMELLS
- ✓ MAKING CODE TESTABLE
- ✓ UNDERSTANDING TEST DoubLES
- ✓ TEST DRIVEN DEVELOPMENT
- ✓ CONCLUSION



GOALS OF SOFTWARE TESTING

WHY TEST?



BUG REPELLENT



CodeFlair.io

DEFECT LOCALIZATION



CodeFlair.io

TESTS AS SAFETY NET



TESTS AS SPECIFICATION

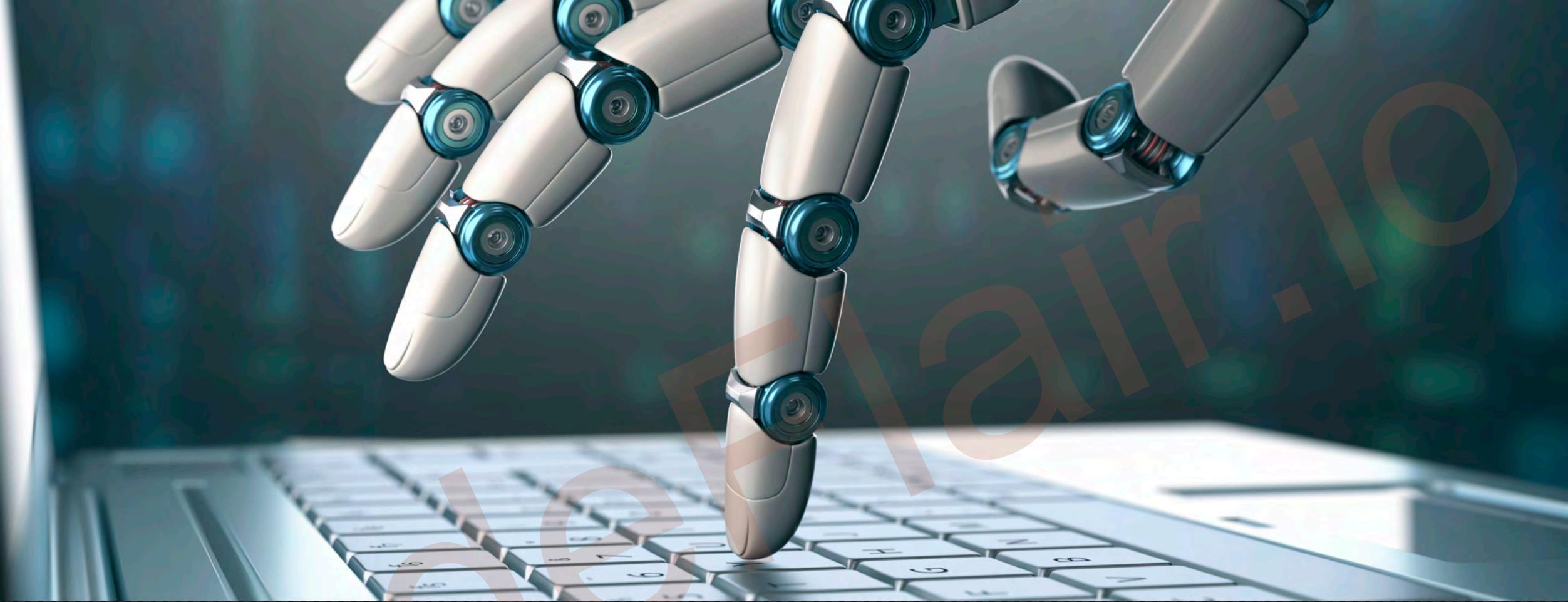


ENABLE
CONTINUOUS
DELIVERY



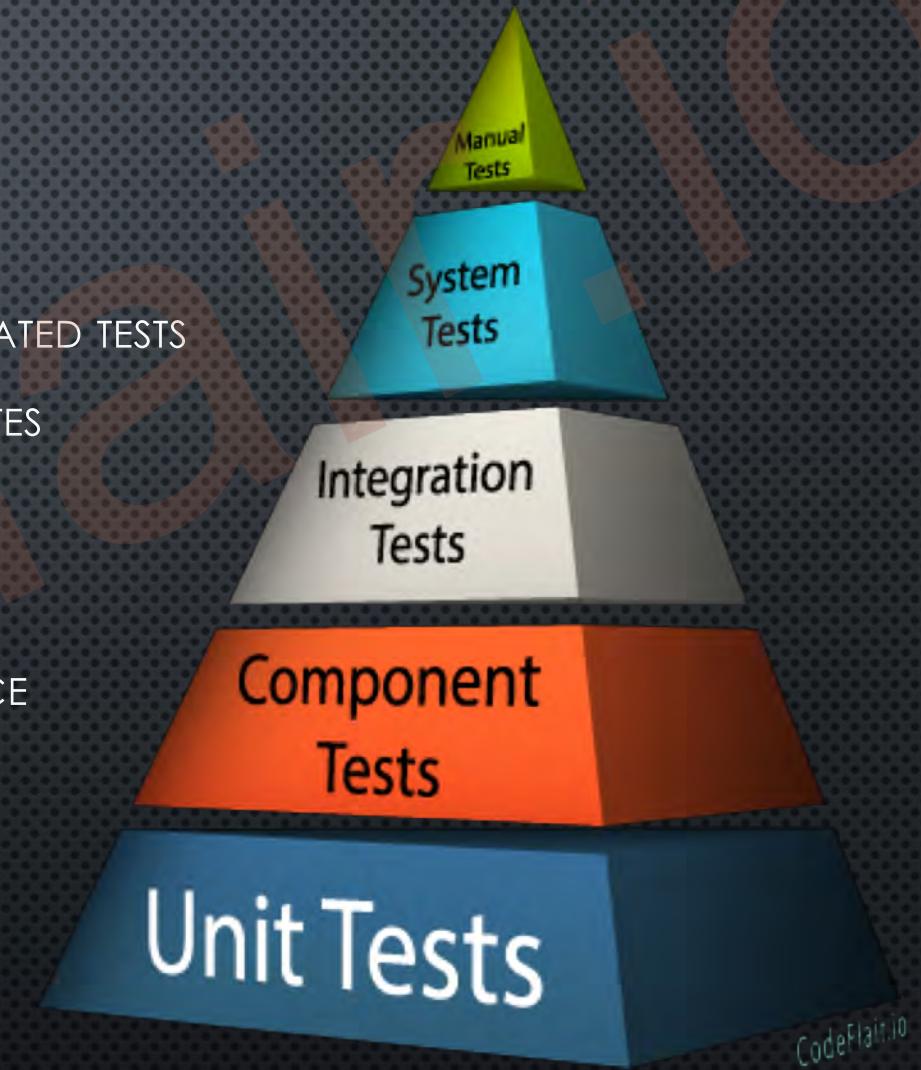
TEST AUTOMATION

PAVING THE WAY TOWARDS CONTINUOUS DELIVERY



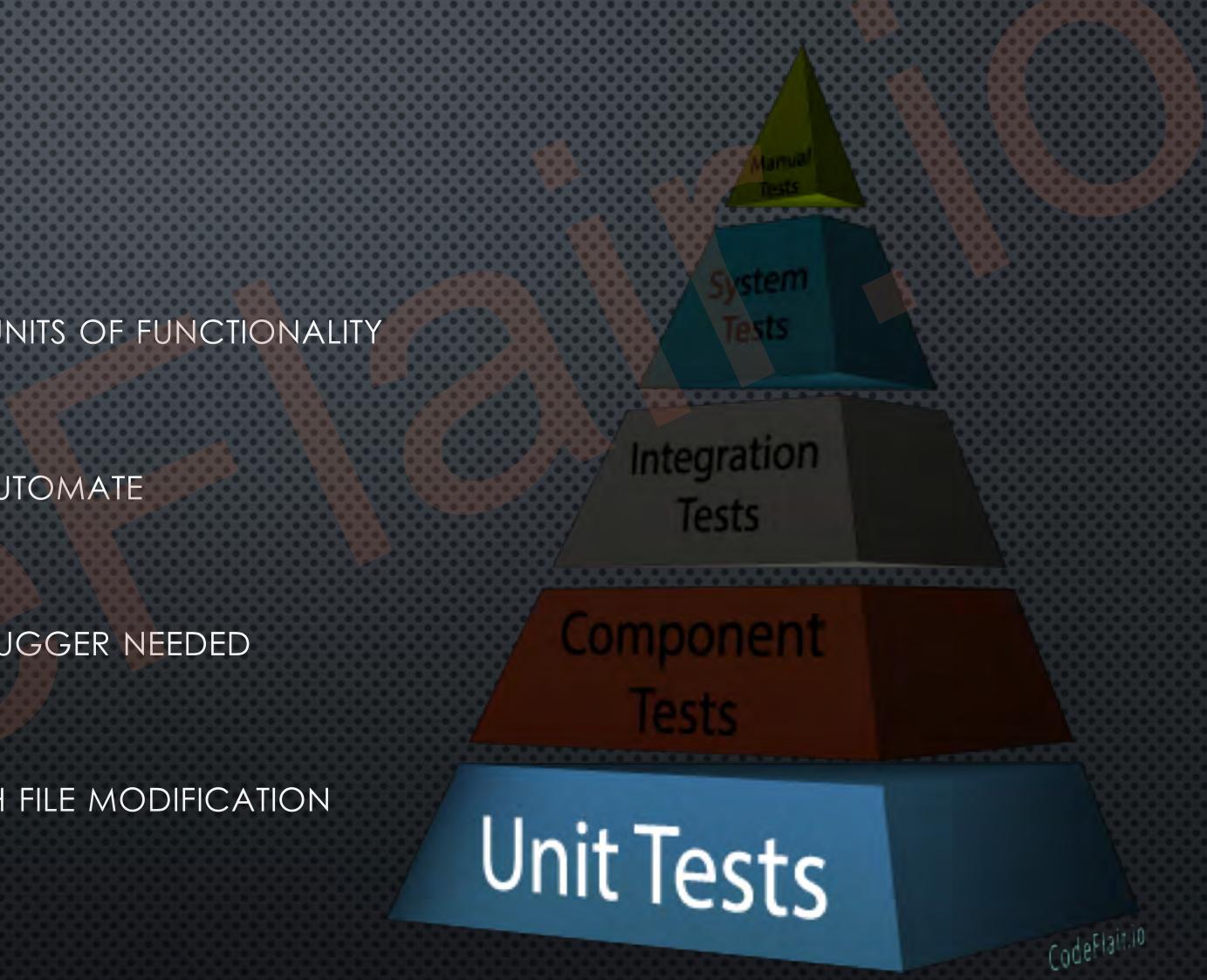
BUILDING CODE COVERAGE

- ✓ TO OPTIMIZE PRODUCTIVITY, MAXIMIZE THE NUMBER OF AUTOMATED TESTS
- ✓ UNIT TESTS SHOULD COMPOSE THE MAJORITY OF YOUR TEST SUITES
- ✓ NOT ALL LEVELS HAVE THE SAME PROBABILITY OF A BUG
- ✓ TESTABILITY IS MORE IMPORTANT THE FARTHER DOWN YOU GO
- ✓ LOWER TESTS SHOULD BE DECOUPLED FROM THE USER INTERFACE



UNIT TESTS

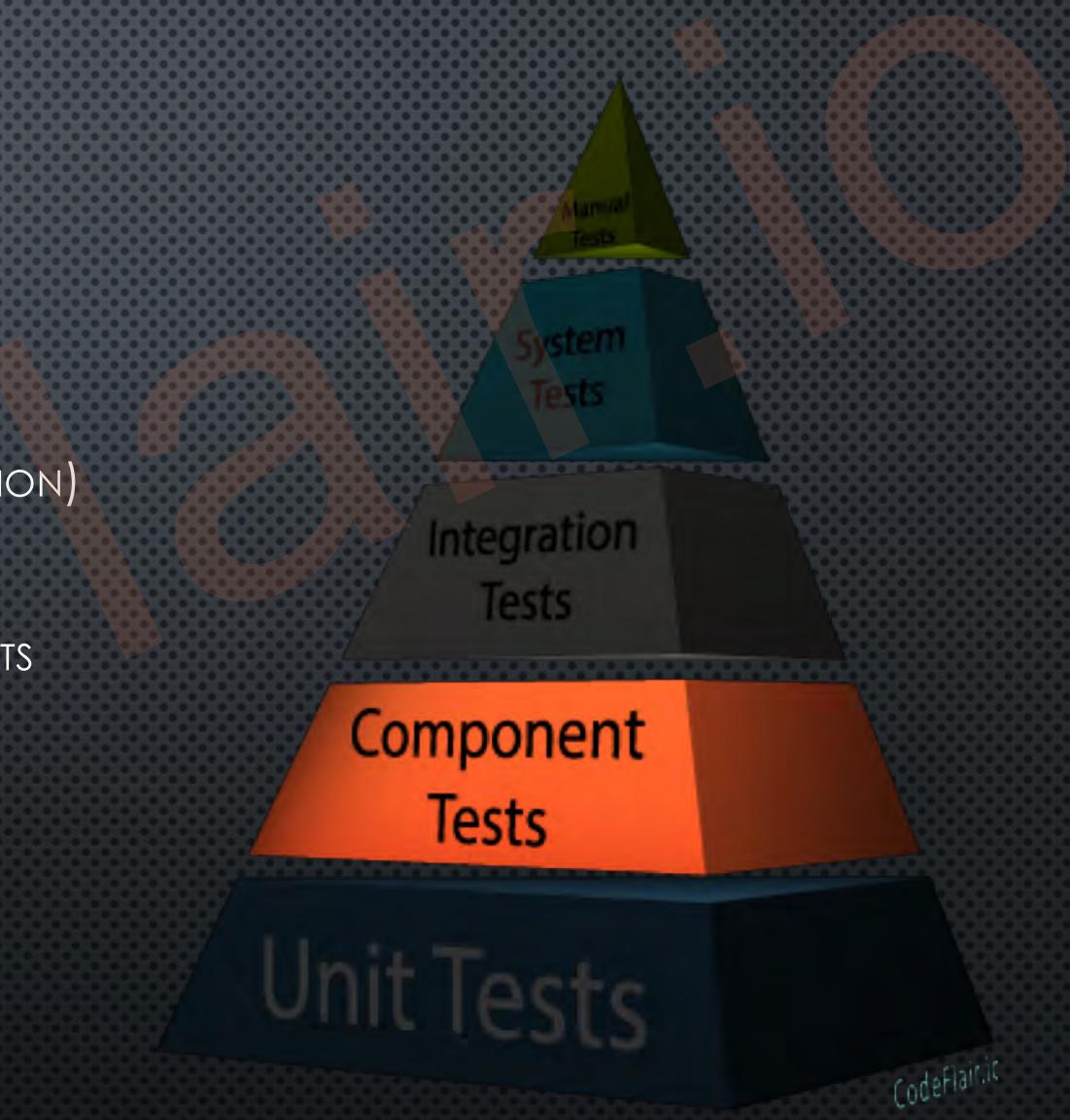
- ✓ VALIDATE SMALL, LOW-LEVEL, ISOLATED UNITS OF FUNCTIONALITY
- ✓ SIMPLE: CYCLOMATIC COMPLEXITY = 1
- ✓ EASY TO WRITE, EASY TO RUN, EASY TO AUTOMATE
- ✓ CAN SIMULATE ALL ERROR CONDITIONS
- ✓ EASY TO REPRODUCE FAILURES – NO DEBUGGER NEEDED
- ✓ EXECUTE IN MILLISECONDS
- ✓ DEVELOPERS CAN RUN THESE AFTER EACH FILE MODIFICATION
- ✓ DOCUMENT THE SYSTEM
- ✓ TARGET 100% CODE COVERAGE



Unit Tests

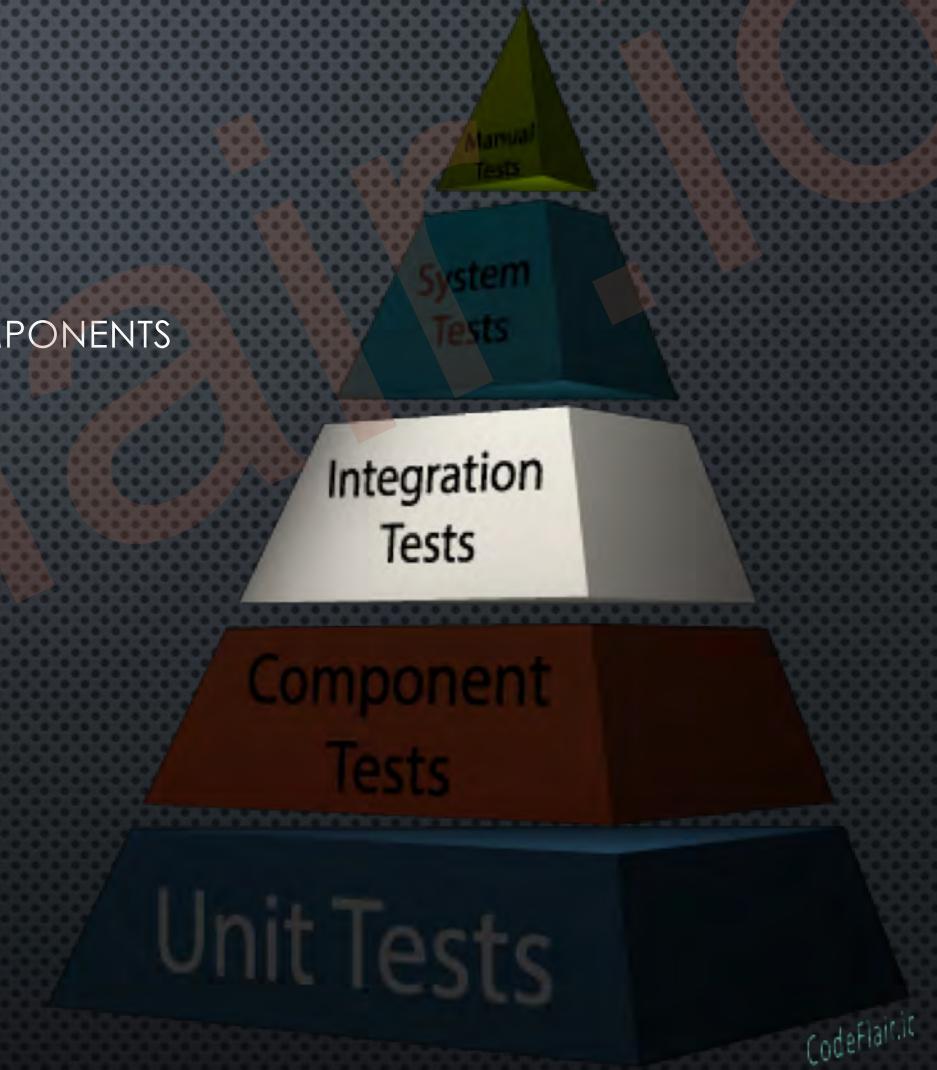
COMPONENT TESTS

- ✓ VALIDATE THE BEHAVIOR OF INDIVIDUAL COMPONENTS
- ✓ THIS IS WHERE BUSINESS RULES ARE VALIDATED (IN ISOLATION)
- ✓ ENSURE PROPER RELATIONSHIPS BETWEEN CLASSES
- ✓ CAN SIMULATE CONDITIONS NOT POSSIBLE IN SYSTEM TESTS
- ✓ EASILY ISOLATE DEPENDENCIES USING TEST DOUBLES
- ✓ EASY TO WRITE, EASY TO RUN, EASY TO AUTOMATE
- ✓ EXECUTE QUICKLY
- ✓ TARGET ~50% CODE COVERAGE



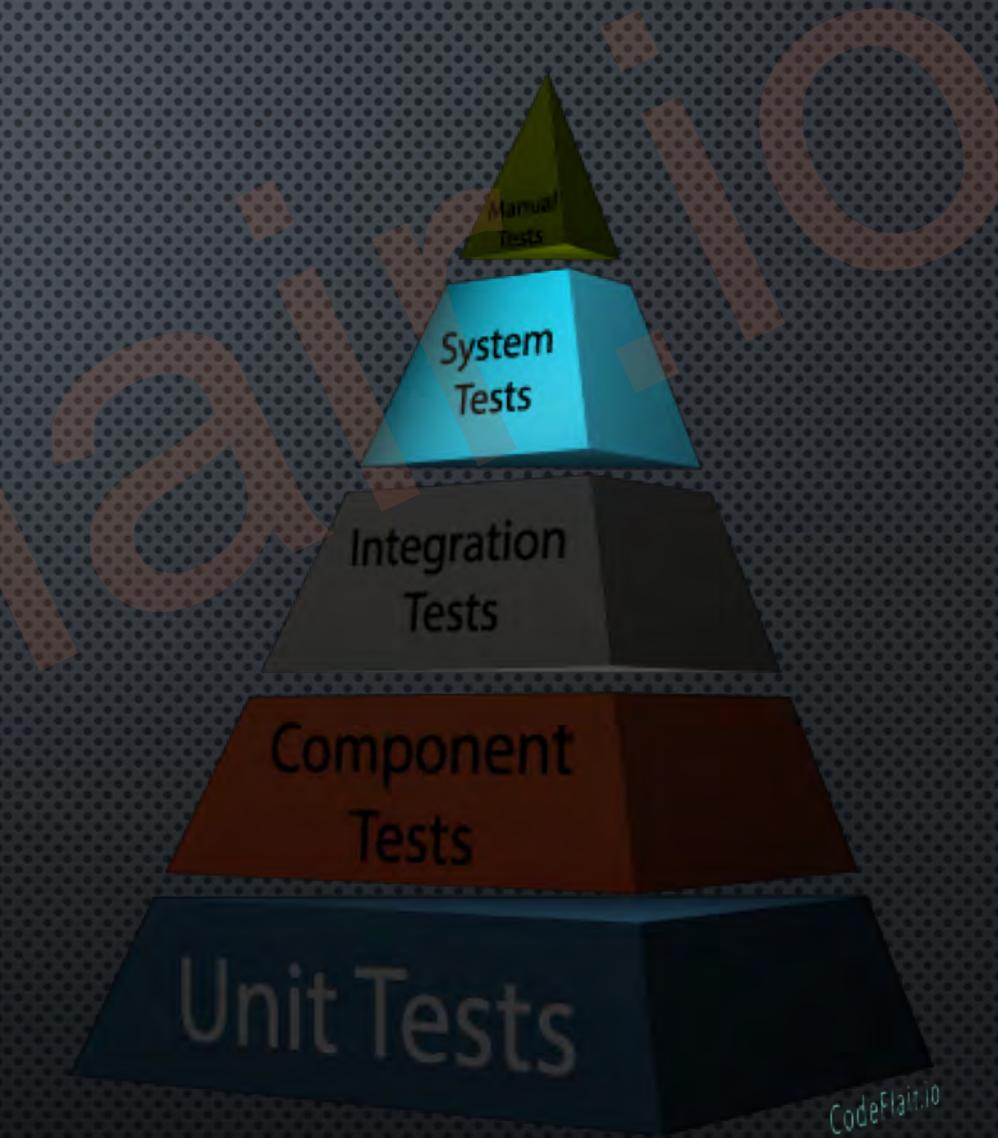
INTEGRATION TESTS

- ✓ VALIDATE THE LINKAGE AND COMMUNICATION BETWEEN COMPONENTS
- ✓ ENSURE A PROPER ARCHITECTURE IS IN PLACE
- ✓ TARGET ~20% CODE COVERAGE



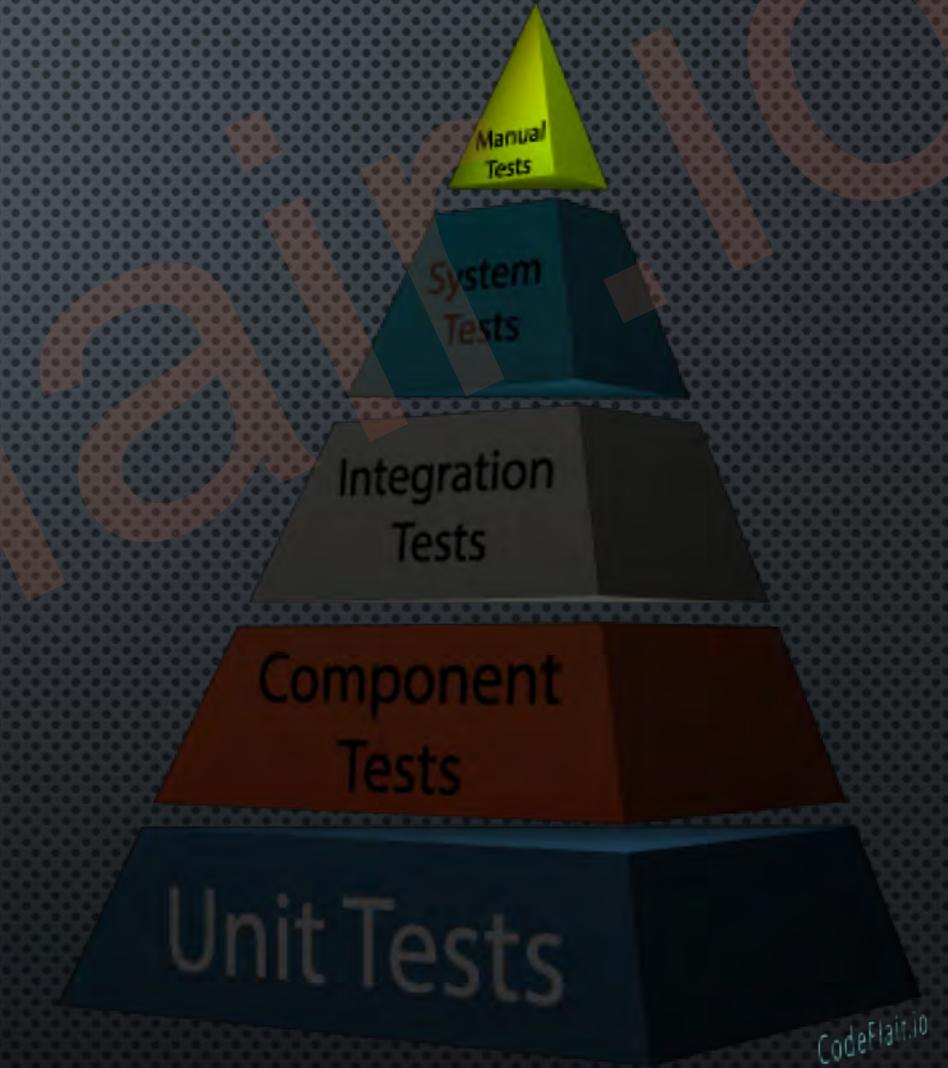
SYSTEM TESTS

- ✓ VALIDATE ALL LAYERS OF THE APPLICATION
- ✓ INCLUDES LOAD TESTING AND PENETRATION TESTING
- ✓ FIRST TIME TESTING VIA THE USER INTERFACE
- ✓ EXPENSIVE AND TIME-CONSUMING TO WRITE AND EXECUTE
- ✓ TEND TO BE FRAGILE
- ✓ DIFFICULT TO TEST CORNER CASES
- ✓ DIFFICULT TO REPRODUCE FAILURES
- ✓ NEED DEBUGGER TO DETERMINE WHAT WENT WRONG
- ✓ TARGET ~10% CODE COVERAGE



MANUAL TESTS

- ✓ ADHOC EXPLORATION FOR UNEXPECTED BEHAVIOR
- ✓ USER ACCEPTANCE TESTING (UAT)
- ✓ DETERMINING ROOT CAUSE OF FAILURES CAN BE DIFFICULT
- ✓ SLOWEST
- ✓ MOST EXPENSIVE
- ✓ BEST KEPT TO A MINIMUM





BASICS OF UNIT TESTING

DEFINING UNIT TESTS

DEFINING TESTS

Setup



Test Fixture establishes a fixed state in which the SUT can be tested.

Exercise



Exercise the SUT by interacting with methods in its public or private interface.

Verify



Verify that the expected outcome has occurred using calls to Assertion Methods.

Teardown



The test fixture that was built by the test should be destroyed to avoid side effects.

COMMON TESTING CONVENTIONS

Setup



Test Fixture establishes a fixed state in which the SUT can be tested.

Exercise



Exercise the SUT by interacting with methods in its public or private interface.

Verify



Verify that the expected outcome has occurred using calls to Assertion Methods.

Teardown



The test fixture that was built by the test should be destroyed to avoid side effects.

COMMON NAMING CONVENTIONS

Setup

Arrange

Build

Given

Exercise

Act

Operate

When

Verify

Assert

Check

Then

Teardown

Annihilate

EXAMPLE

```
[Test]
public void Adding1And2_ShouldReturn3()
{
    //Setup
    Calculator calc = new Calculator();

    //Exercise
    int result = calc.Sum(1, 2);

    //Verify
    Assert.Equal(3, result);
}
```

EXAMPLE

```
[Test]
public void Adding1And2_ShouldReturn3()
{
    //Arrange
    Calculator calc = new Calculator();

    //Act
    int result = calc.Sum(1, 2);

    //Assert
    Assert.Equal(3, result);
}
```

EXAMPLE

```
[Test]
public void Adding1And2_ShouldReturn3()
{
    //Build
    Calculator calc = new Calculator();

    //Operate
    int result = calc.Sum(1, 2);

    //Check
    Assert.Equal(3, result);
}
```

EXAMPLE

```
[Test]
public void Adding1And2_ShouldReturn3()
{
    //Given
    Calculator calc = new Calculator();

    //When
    int result = calc.Sum(1, 2);

    //Then
    Assert.Equal(3, result);
}
```

JUNIT EXAMPLE

```
@Test  
public void Adding1And2_ShouldReturn3() throws Exception  
{  
    //Arrange  
    Calculator calc = new Calculator();  
  
    //Act  
    int result = calc.Sum(1, 2);  
  
    //Assert  
    assertEquals(3, result);  
}
```

NUNIT EXAMPLE

```
[Test]
public void Adding1And2_ShouldReturn3()
{
    //Arrange
    Calculator calc = new Calculator();

    //Act
    int result = calc.Sum(1, 2);

    //Assert
    Assert.AreEqual(3, result);
}
```

XUNIT EXAMPLE

```
[Fact]
public void Adding1And2_ShouldReturn3()
{
    //Arrange
    Calculator calc = new Calculator();

    //Act
    int result = calc.Sum(1, 2);

    //Assert
    Assert.Equal(3, result);
}
```



CORE TECHNIQUES

BUILDING A FOUNDATION FOR TESTABILITY

TECHNIQUES OF UNIT TESTING

- ✓ WRITE THE TESTS FIRST
- ✓ DESIGN FOR TESTABILITY
- ✓ USE THE FRONT DOOR FIRST
- ✓ COMMUNICATE INTENT
- ✓ DON'T MODIFY THE SUT
- ✓ KEEP TESTS INDEPENDENT
- ✓ ISOLATE THE SUT
- ✓ MINIMIZE TEST OVERLAP
- ✓ MINIMIZE UNTESTABLE CODE
- ✓ KEEP TEST LOGIC OUT OF PRODUCTION CODE
- ✓ VERIFY ONE CONDITION PER TEST
- ✓ TEST CONCERNs SEPARATELY



CodeFlair

TEST SMELLS

SYMPTOMS OF PROBLEMS

A CATALOG OF SMELLS

Project Smells



Code Smells



Behavior Smells



PROJECT SMELLS

1. PRODUCTION DEFECTS
2. INTEGRATION BUILD FAILURES
3. DEVELOPERS NOT WRITING TESTS
4. BUGGY TESTS
5. HIGH TEST MAINTENANCE COST



Symptoms of problems on a project

Project smells are usually the first sign
that there is a problem with testing.

BEHAVIOR SMELLS

1. ASSERTION ROULETTE
2. FRAGILE TESTS
3. FREQUENT DEBUGGING
4. MANUAL INTERVENTION
5. SLOW TESTS

Symptoms that are easily noticed

Behavior smells will take the form of compile errors or test failures.



CODE SMELLS

1. OBSCURE TESTS
2. CONDITIONAL TEST LOGIC
3. TEST CODE DUPLICATION
4. TEST LOGIC IN PRODUCTION
5. HARD-TO-TEST CODE



Symptoms of poorly written tests
Code smells affect maintenance cost of tests and may be warning signs of behavior smells.



MAKING CODE TESTABLE

TESTING IS EASY IN THE PRESENCE OF GOOD DESIGN



Doug Klugh

@DougKlugh

The best way to write great tests is to write great (production) code. Keep it loose.

6:38 AM - 18 Oct 2018 from Scottsdale, AZ



QUALITIES OF GOOD CODE

- ✓ SIMPLE
- ✓ FLEXIBLE
- ✓ REUSABLE
- ✓ LOOSELY COUPLED
- ✓ HIGHLY COHESIVE

SEPARATION OF CONCERNS

- ✓ LAYERED ARCHITECTURE
- ✓ SINGLE RESPONSIBILITY PRINCIPLE
- ✓ INTERFACE SEGREGATION PRINCIPLE
- ✓ COMMON CLOSURE PRINCIPLE
- ✓ ONE ACTOR / USE CASE



SoC - Design Principle

Encapsulates cohesive information and/or logic in a function, class, module, or component.

SOLID PRINCIPLES

Single Responsibility Principle

A class should have only one reason to change.

Open Closed Principle

A class should be open to extension, but closed for modification.

Liskov Substitution Principle

An object should be replaceable with its subtypes without altering any system behavior.

Interface Segregation Principle

Many specific interfaces are preferable to a single generic interface.

Delegation Inversion Principle

A class should only depend upon abstractions, not implementations.

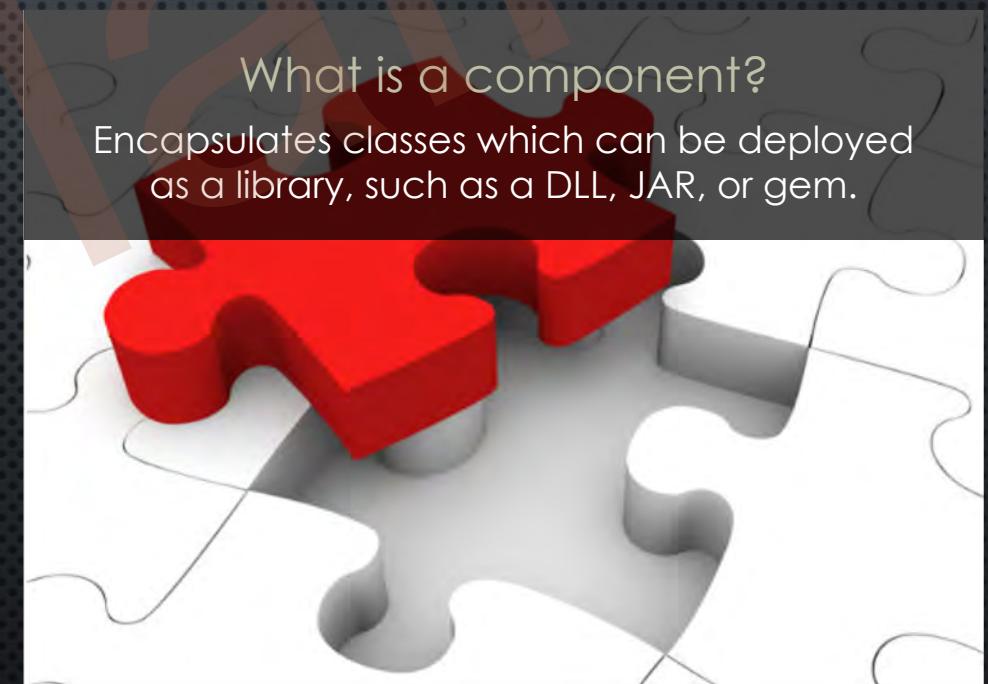
STRIVE TO BE SOLID

SOLID PRINCIPLES GREATLY ENHANCE THE TESTABILITY OF OUR SOFTWARE.

SO IT'S IMPORTANT TO BE SOLID.

SOFTWARE COMPONENTS

- ✓ FACILITATE REUSE
- ✓ FACILITATE INDEPENDENT DEVELOPMENT
- ✓ FACILITATE INDEPENDENT DEPLOYMENT
- ✓ SUPPORT DEVELOPMENT OF CLOUD-NATIVE APPS
- ✓ ENABLE CONTINUOUS DELIVERY WITH ZERO DOWNTIME
- ✓ SUPPORT DevOps



What is a component?

Encapsulates classes which can be deployed as a library, such as a DLL, JAR, or gem.

COMPONENT COHESION PRINCIPLES

✓ RELEASE REUSE EQUIVALENCY PRINCIPLE

COMPONENTS NEED TO BE LARGE ENOUGH TO JUSTIFY THE COST OF MANAGING THE RELEASE CYCLE.

✓ COMMON CLOSURE PRINCIPLE

CLASSES THAT CHANGE FOR THE SAME REASON SHOULD BE GROUPED TOGETHER IN THE SAME COMPONENT.

✓ COMMON REUSE PRINCIPLE

CREATE COMPONENTS WHERE IF ONE CLASS IN THE COMPONENT IS USED, THEY'RE ALL USED.

What goes into a component?
Which classes do we group together and
which classes do we keep apart?



COMPONENT COUPLING PRINCIPLES

✓ ACYCLIC DEPENDENCIES PRINCIPLE

THE DEPENDENCIES BETWEEN COMPONENTS MUST NOT FORM A CYCLE.

✓ STABLE DEPENDENCIES PRINCIPLE

COMPONENTS SHOULD DEPEND UPON EACH OTHER IN THE DIRECTION OF INCREASING STABILITY.

✓ STABLE ABSTRACTIONS PRINCIPLE

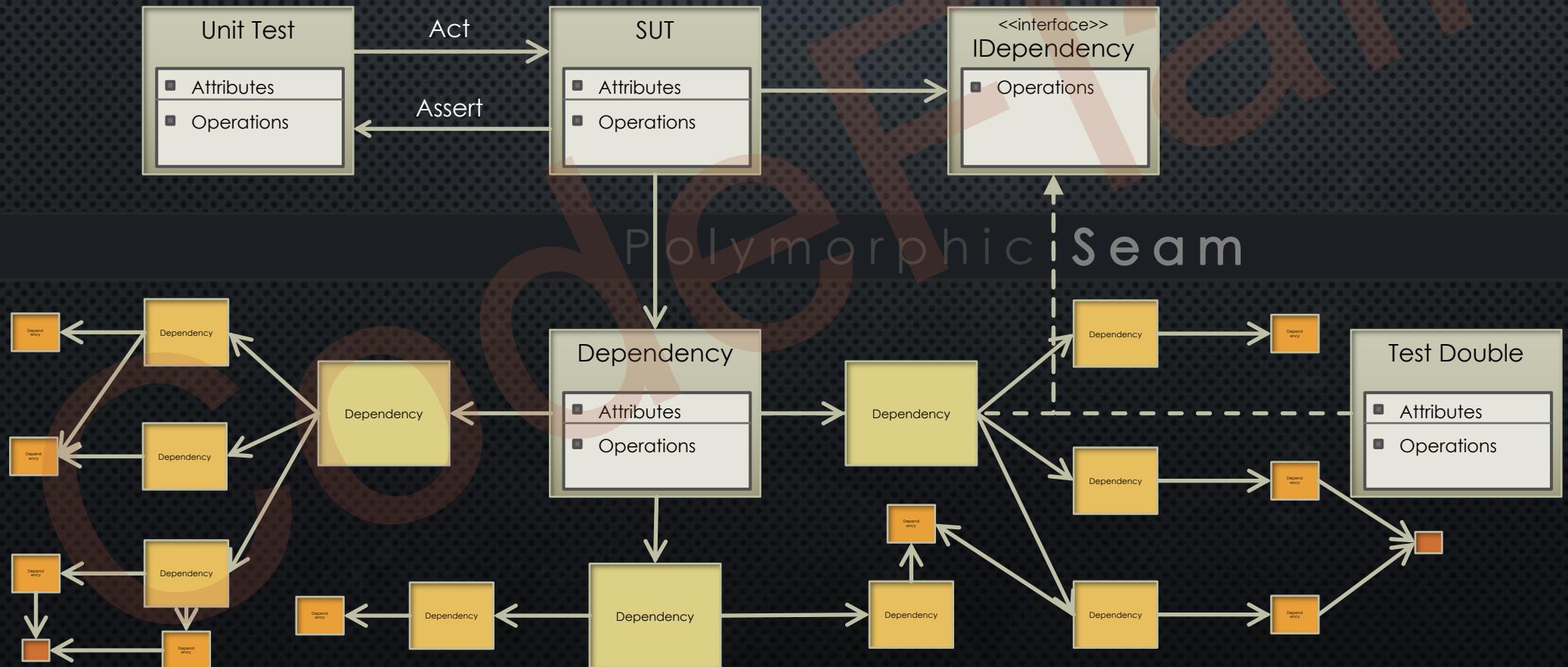
THE MORE STABLE A COMPONENT IS, THE MORE ABSTRACT IT SHOULD BE – SO THAT IT CAN CONFORM TO THE OPEN/CLOSED PRINCIPLE.



How are components related?

Effectively managing dependencies between components will facilitate testability.

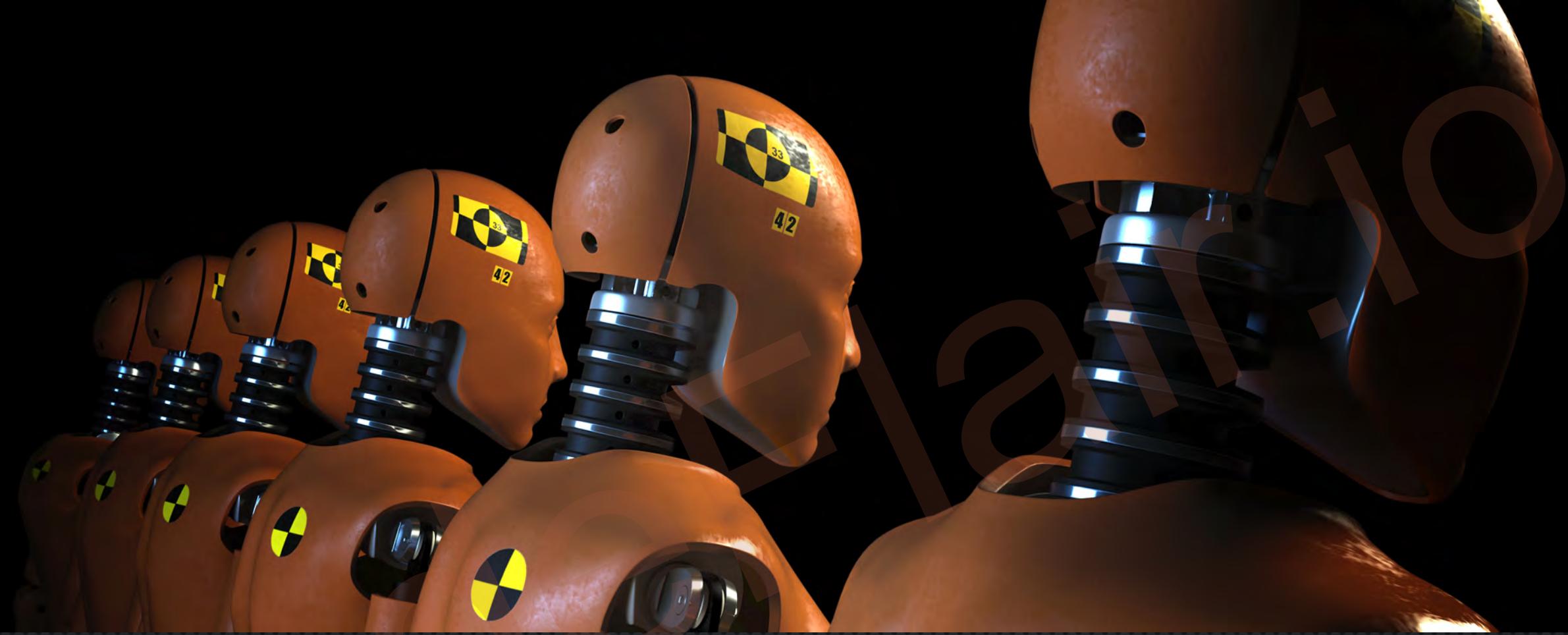
KEEPING IT LOOSE



CO

TEST DOUBLES

THE STUNT DOUBLES OF SOFTWARE TESTING

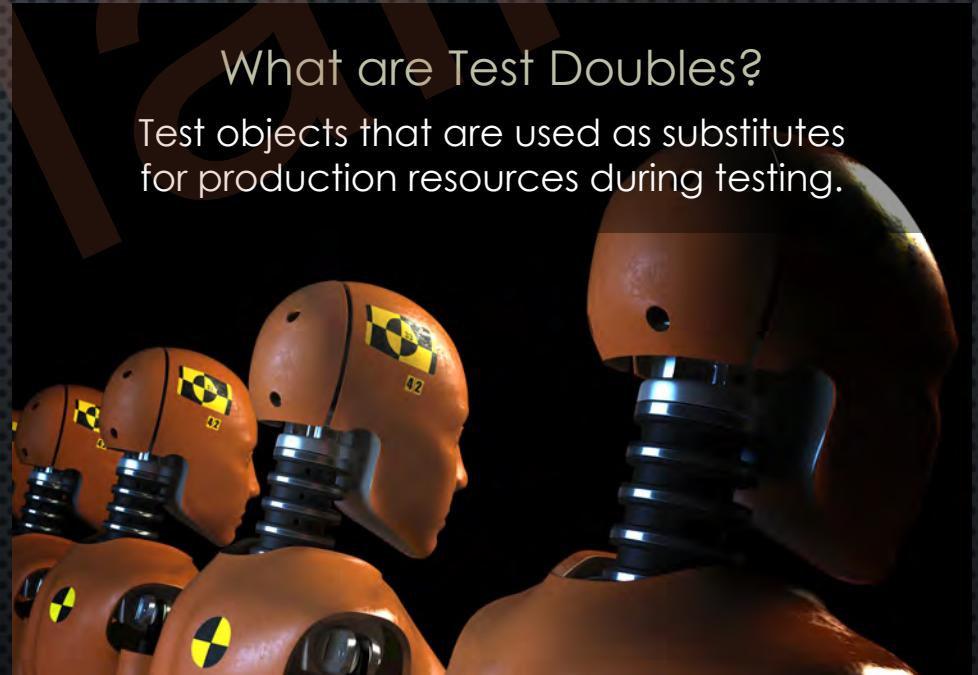


UNDERSTANDING TEST DOUBLES

- ✓ DUMMIES
- ✓ STUBS
- ✓ SPIES
- ✓ MOCKS
- ✓ FAKES

What are Test Doubles?

Test objects that are used as substitutes for production resources during testing.



DUMMY

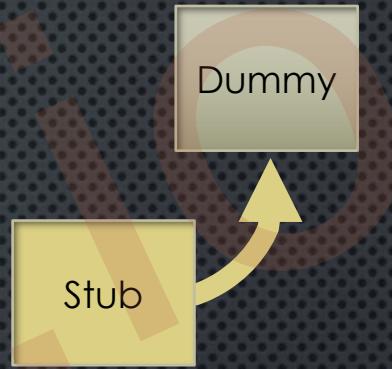
Dummy

- ✓ SIMPLEST FORM OF A TEST DOUBLE
- ✓ IMPLEMENTS AN INTERFACE WHERE ALL THE FUNCTIONS DO NOTHING
- ✓ IF THE FUNCTIONS RETURN A VALUE, THEY RETURN AS CLOSE TO NULL OR ZERO AS POSSIBLE
- ✓ MOST OFTEN USED AS AN ARGUMENT TO A FUNCTION, WHERE NEITHER THE TEST NOR THE FUNCTION CARES WHAT HAPPENS TO THAT ARGUMENT

SEVERS AS A PLACEHOLDER WHERE AN OBJECT IS REQUIRED.

STUB

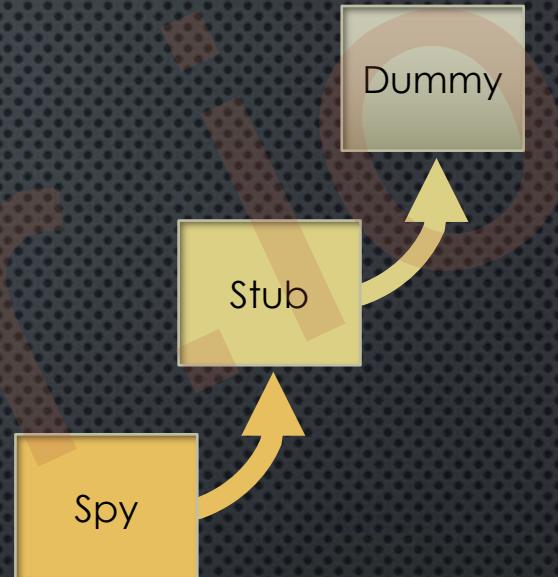
- ✓ A STUB IS A DUMMY
- ✓ ITS FUNCTIONS DO NOTHING
- ✓ RETURNS FIXED VALUES THAT ARE CONSISTENT WITH THE NEEDS OF THE TESTS



DIRECTS EXECUTION OF THE CODE THROUGH CERTAIN PATHWAYS TO BE TESTED.

SPY

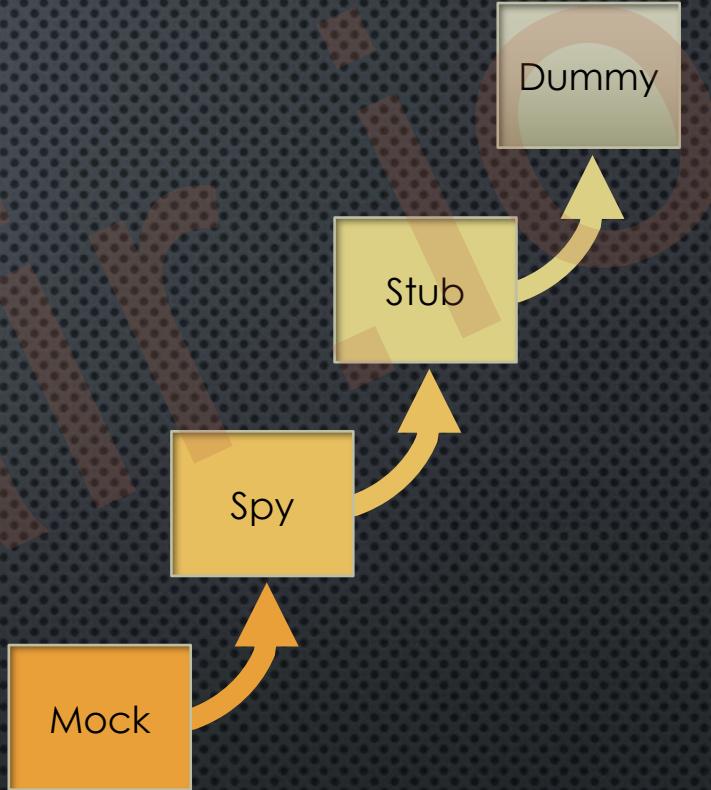
- ✓ A SPY IS A STUB
- ✓ ITS FUNCTIONS PERFORM NO EXTERNAL ACTIONS
- ✓ RETURNS VALUES THAT ARE USEFUL TO THE TESTS
- ✓ REMEMBERS FACTS SO THE TESTS CAN VERIFY THE FUNCTIONS WERE CALLED PROPERLY
 - ✓ WAS A FUNCTION EVER CALLED?
 - ✓ HOW MANY TIMES WAS IT CALLED?
 - ✓ HOW MANY ARGUMENTS WERE PASSED IN?
 - ✓ WHAT WERE THOSE ARGUMENTS?



THE ONLY THING A SPY DOES IS WATCH AND REMEMBER.

MOCK

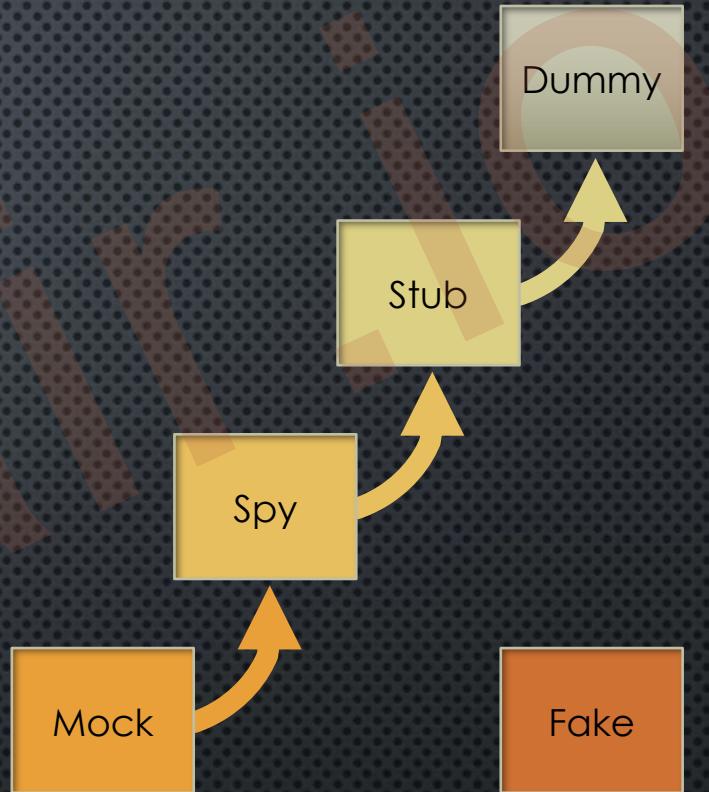
- ✓ A MOCK IS A SPY
- ✓ ITS FUNCTIONS DO NOTHING
- ✓ RETURNS VALUES THAT ARE USEFUL TO THE TESTS
- ✓ REMEMBERS FACTS ABOUT THE WAY IT WAS CALLED
- ✓ SETS UP CONDITIONS THAT ARE TO BE TESTED AND EVALUATES WHETHER THOSE CONDITIONS HAVE BEEN MET
- ✓ THE TEST DOES NOT CHECK WHAT THE MOCK SPIED ON



A MOCK KNOWS WHAT SHOULD HAPPEN AND REPORTS BACK TO THE TEST.

FAKE

- ✓ A FAKE IS DIFFERENT FROM OTHER TYPES OF TEST DOUBLES
- ✓ OFTEN USED TO SIMULATE EXTERNAL DEVICES OR SERVICES
- ✓ USUALLY CONTAINS A LOT OF LOGIC AND CAN BECOME COMPLEX
- ✓ THE MORE THE SYSTEM GROWS, THE MORE THE FAKE GROWS
- ✓ CAN PRESENT MAINTENANCE CHALLENGES



BECAUSE OF THEIR COMPLEXITY, FAKES SHOULD BE KEPT TO A MINIMUM.



C

TEST DRIVEN DEVELOPMENT

TESTING CODE BEFORE IT IS WRITTEN

CodeFlair.io

TEST-DRIVEN DEVELOPMENT

- ✓ EXTREME PROGRAMMING
- ✓ TDD ISN'T JUST ABOUT TESTING... IT'S ABOUT DESIGN
- ✓ ARCHITECTURE AND IMPLEMENTATION EVOLVE FROM TESTS THAT ARE WRITTEN PRIOR TO WRITING THE CODE

Going well beyond quality control
TDD results in a high quality design that is easily maintainable, extensible, and yes, **testable**.



RED → GREEN → REFACTOR

1. MAKE IT FAIL – WRITE A FAILING TEST
2. MAKE IT WORK – WRITE THE SIMPLEST CODE TO PASS THE TEST
3. MAKE IT BETTER – CLEAN, IMPROVE, AND OPTIMIZE THE CODE JUST WRITTEN

THE THREE LAWS OF TDD

1. WRITE NO PRODUCTION CODE EXCEPT TO PASS A FAILING TEST
2. WRITE ONLY ENOUGH OF A TEST TO DEMONSTRATE A FAILURE
3. WRITE ONLY ENOUGH PRODUCTION CODE TO PASS THE TEST

Disciplines of Test-Driven Development
The three laws of Test-Driven Development define the discipline of TDD.





JERRY BRONWENHEIM PRODUCTION

GONE IN 60 SECONDS

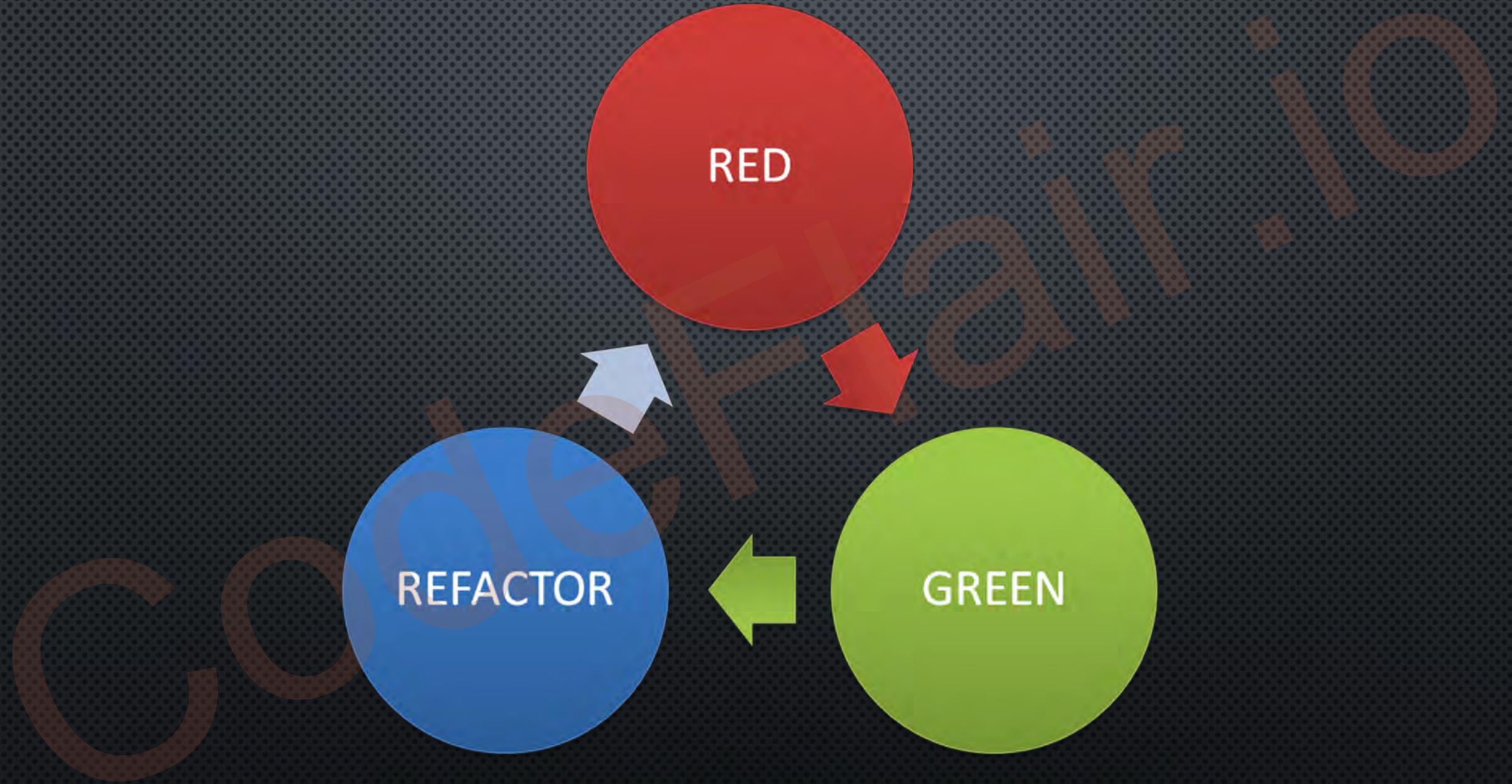
NICOLAS CAGE



ANGELINA JOLIE

GIOVANNI RIBISI

ROBERT DUVALL



CONCLUSION

MANY BENEFITS TO UNIT TESTING



WHY UNIT TEST?

- ✓ IMPROVES QUALITY
 - ✓ PROMOTES GOOD PRODUCTION CODE
 - ✓ PREVENTS BUGS FROM BEING INTRODUCED
- ✓ REDUCES RISK
 - ✓ TEST SUITES PROVIDE SAFETY NET
 - ✓ TEST SMELLS WARN OF UNDER-LYING PROBLEMS
- ✓ REDUCES TIME-TO-MARKET
 - ✓ REDUCES DEBUG TIME
 - ✓ RELIABLE TEST COVERAGE SUPPORTS CONTINUOUS DELIVERY

RELATED COURSES



Enhancing Software Testability

Software testability is impacted by practices throughout the development lifecycle. This course teaches engineering principles which greatly increase software testability, thereby raising code quality, test coverage, test effectiveness, and productivity, while reducing testing time, defects, and time to market. Methods for enhancing testability of requirements, design, and software components are taught, along with practices to best utilize test doubles.

Principles & Practices of Test-Driven Development

This course teaches the principles and practices of Test-Driven Development (TDD) and demonstrates how proper software design evolves through application of the eXtreme Programming principle of Test First. Unit testing principles are also covered, along with a thorough discussion on the benefits of TDD. An application is developed (from start to finish) during this course to explain step-by-step and demonstrate first-hand how a high quality, testable design evolves by applying the three laws of Test-Driven Development.

Understanding Test Doubles

Test doubles are used as substitutes for production resources during testing. And effective use of test doubles is essential to maximizing software testability. This course explains what test doubles are, five different types of test doubles, how they are used, and the benefits of each. A comparison is also made between mocking frameworks and hand-rolled mocks.

THANK YOU

WWW.GITHUB.COM/KLUGH/COURSE-INSIDEUNITTESTING



Doug Klugh



doug@CodeFlair.io



@DougKlugh



/Klugh