INSIDE UNIT TESTING

PRINCIPLES & PRACTICES

 Doug Klugh

 doug@CodeFlair.io

 @DougKlugh

 /Klugh

 /Klugh

 /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

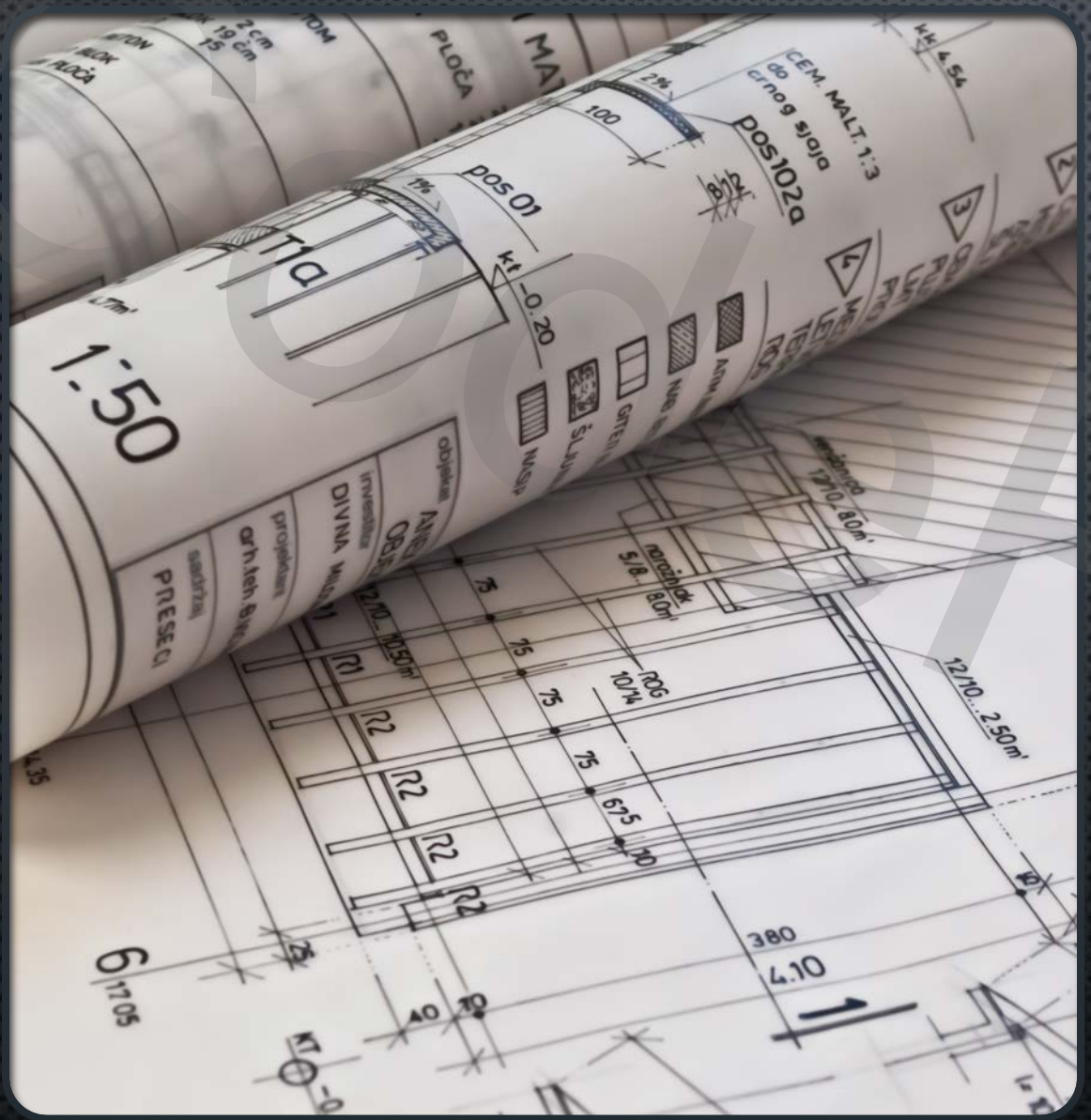
GOALS OF SOFTWARE TESTING

WHY TEST?

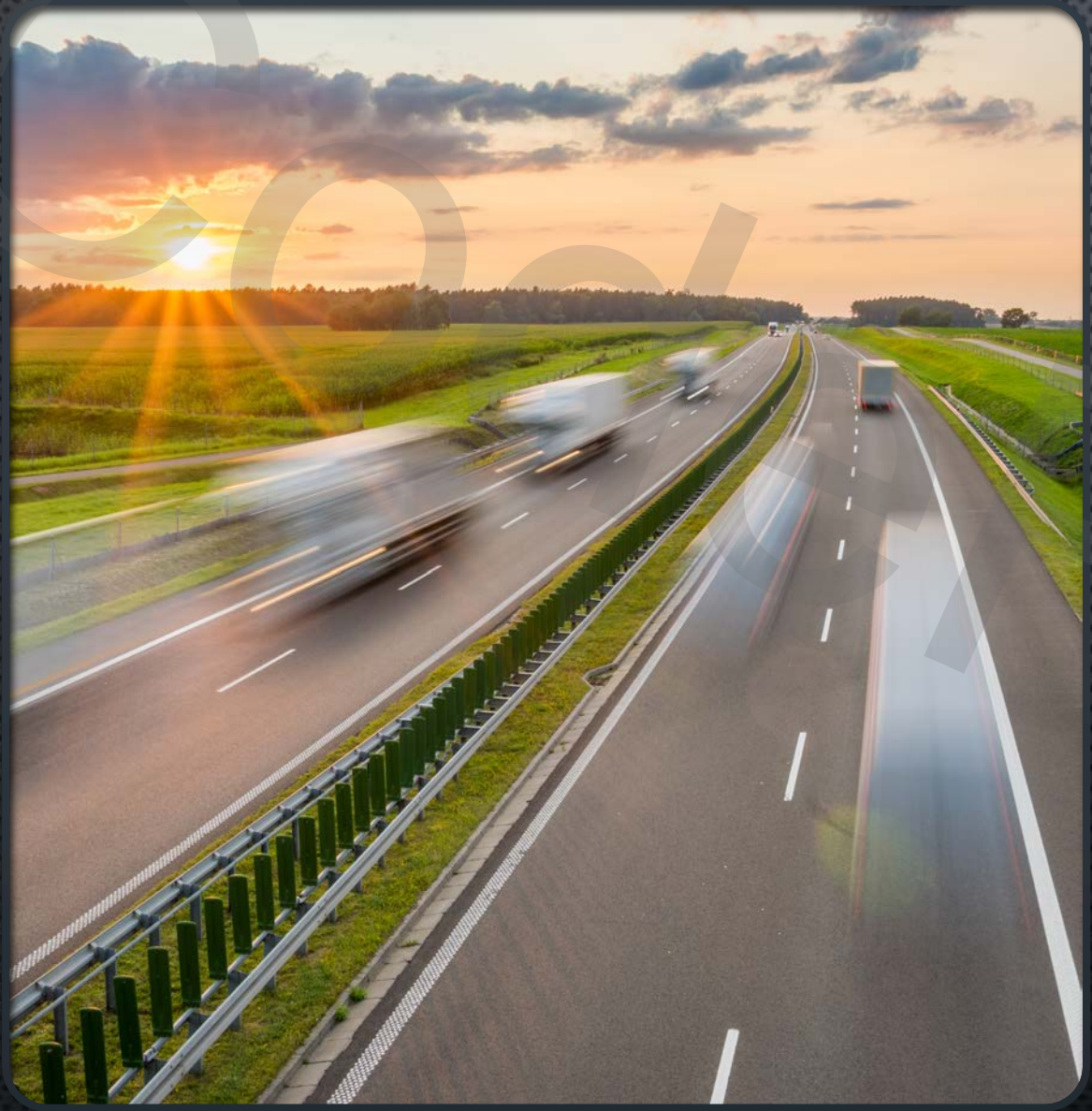
BUG REPELLENT

DEFECT LOCALIZATION

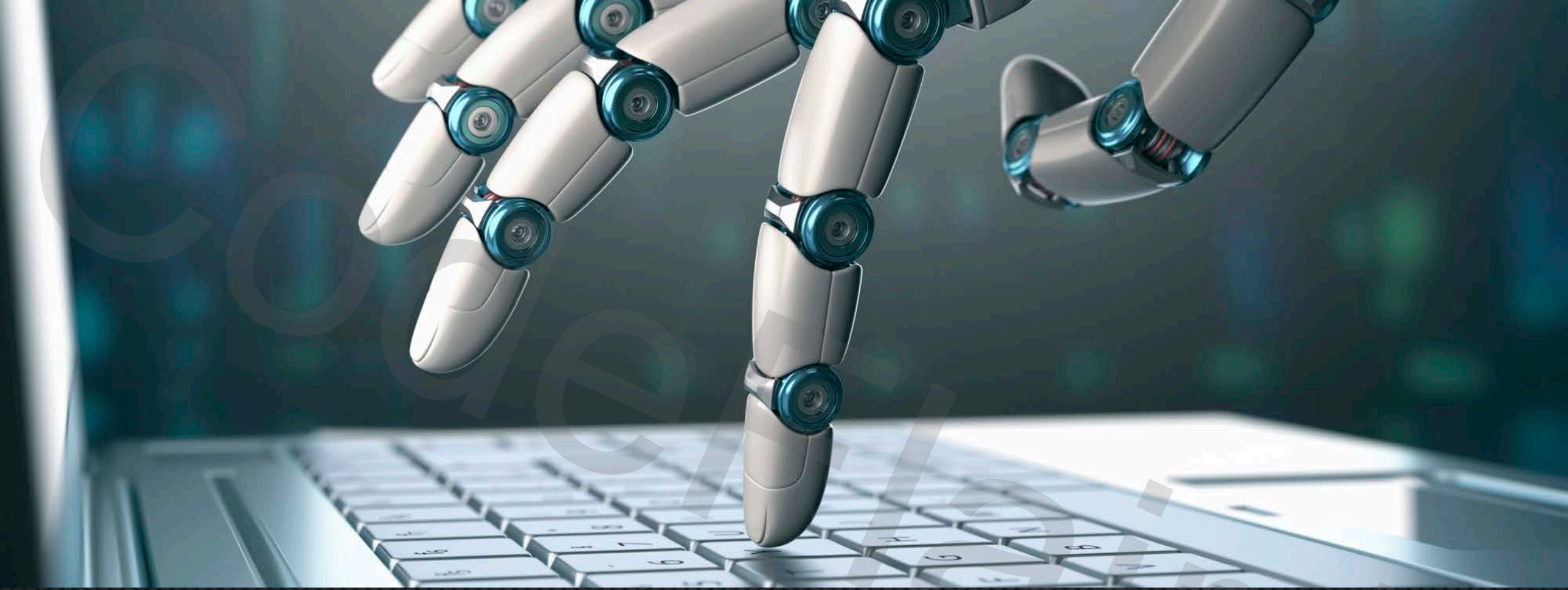
TESTS AS SAFETY NET



TESTS AS SPECIFICATION



ENABLE CONTINUOUS DELIVERY

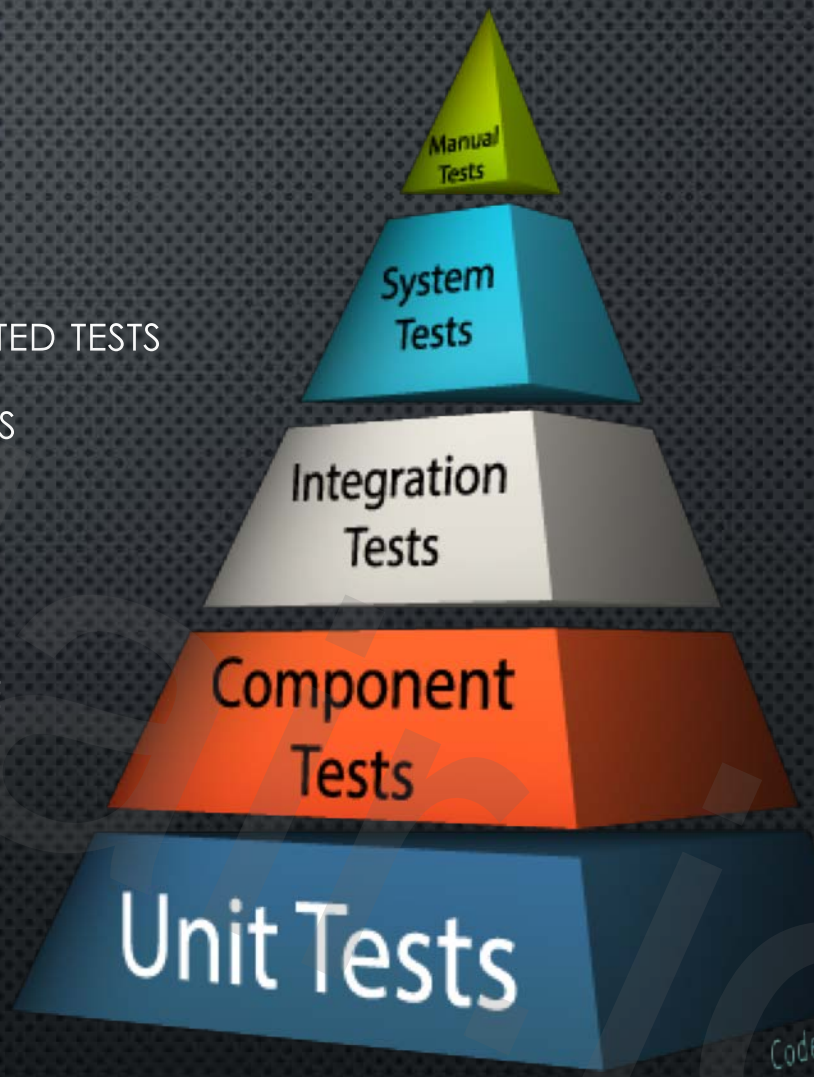


TEST AUTOMATION

PAVING THE WAY TOWARDS CONTINUOUS DELIVERY

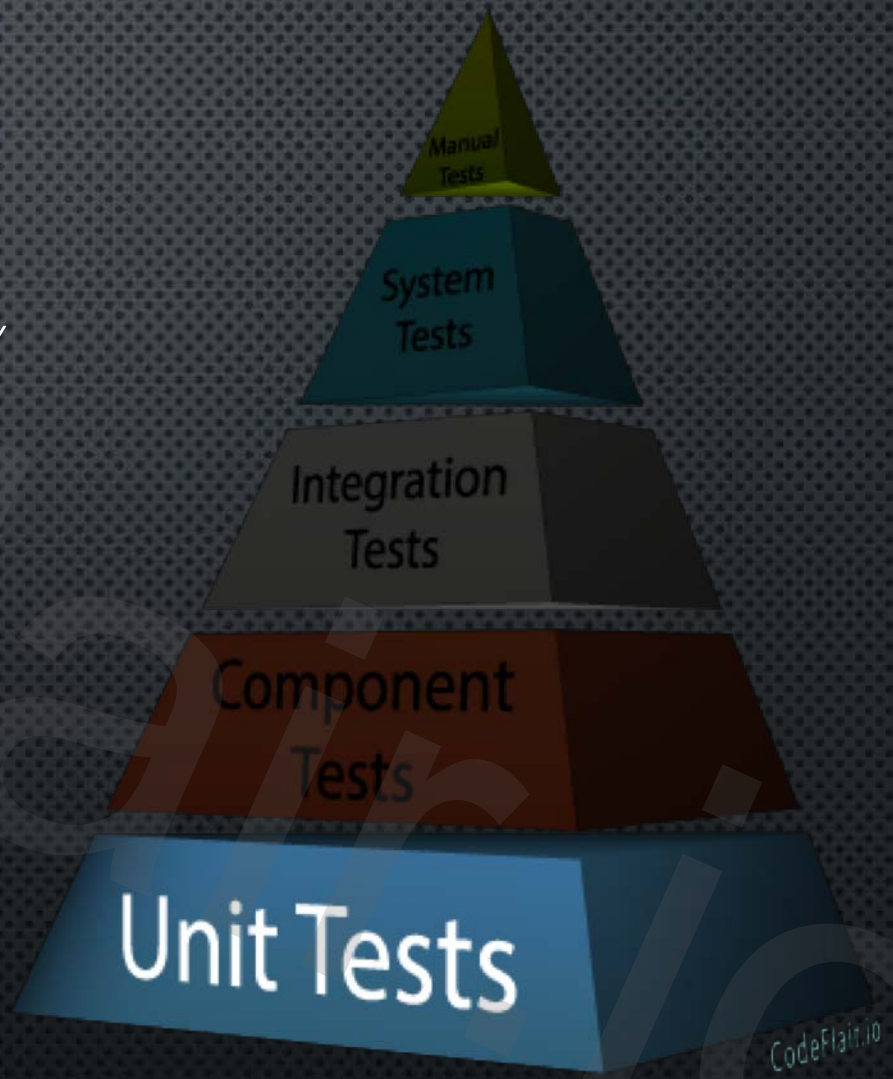
BUILDING CODE COVERAGE

- ✓ 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



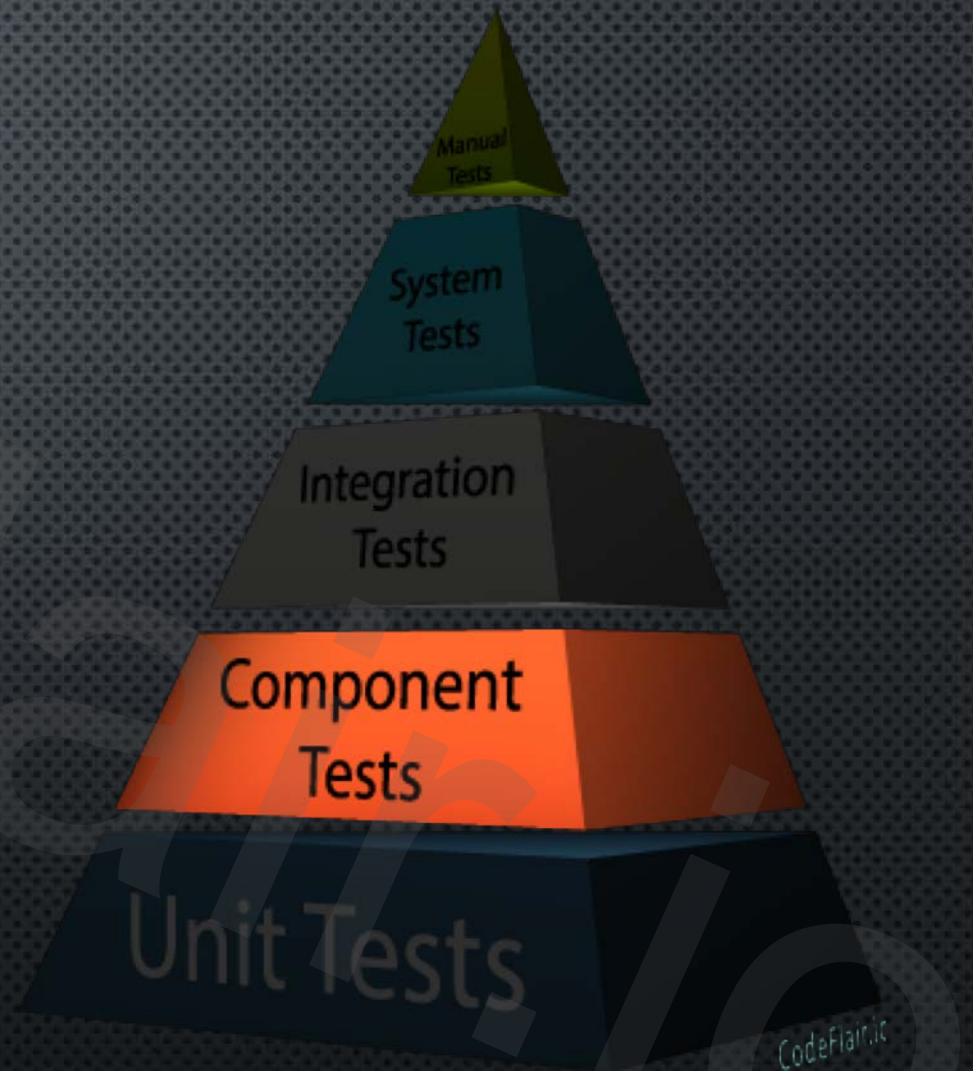
UNIT TESTS

- ✓ 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



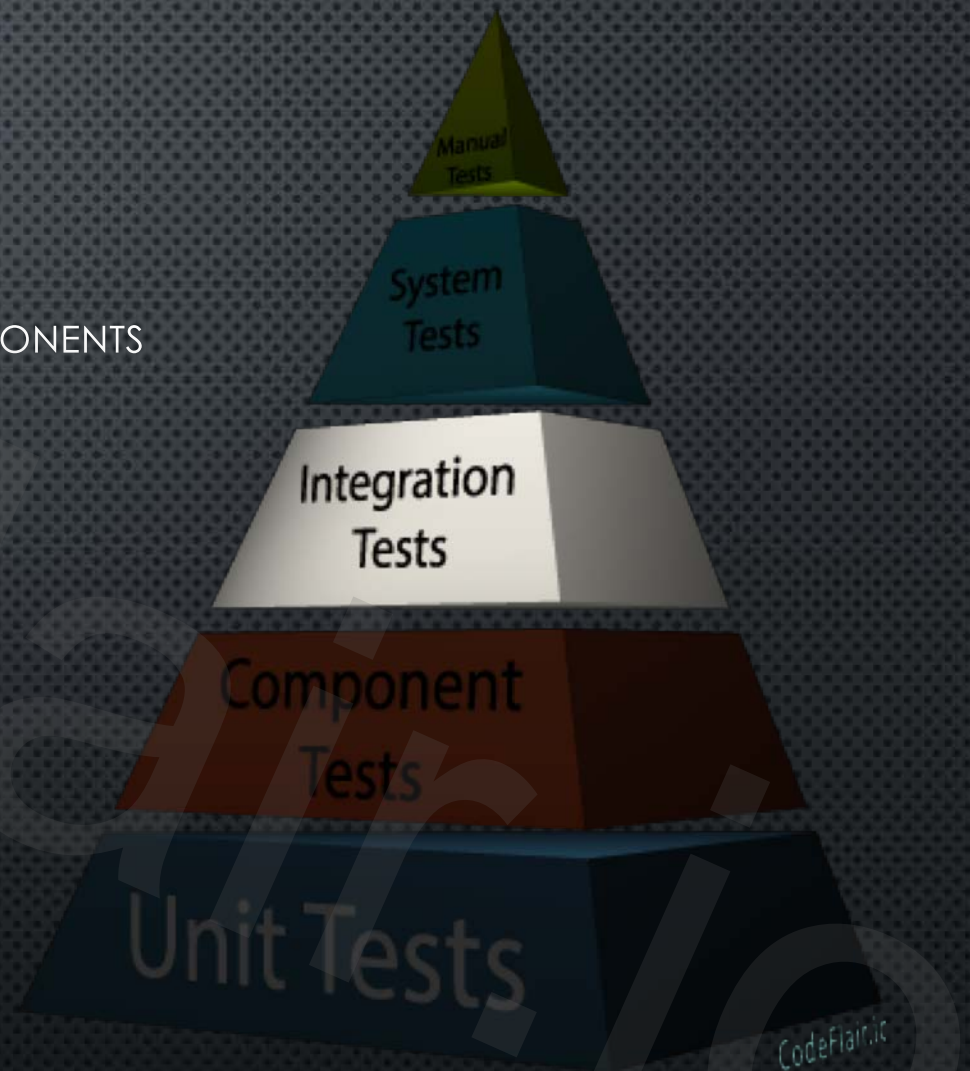
COMPONENT TESTS

- ✓ 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



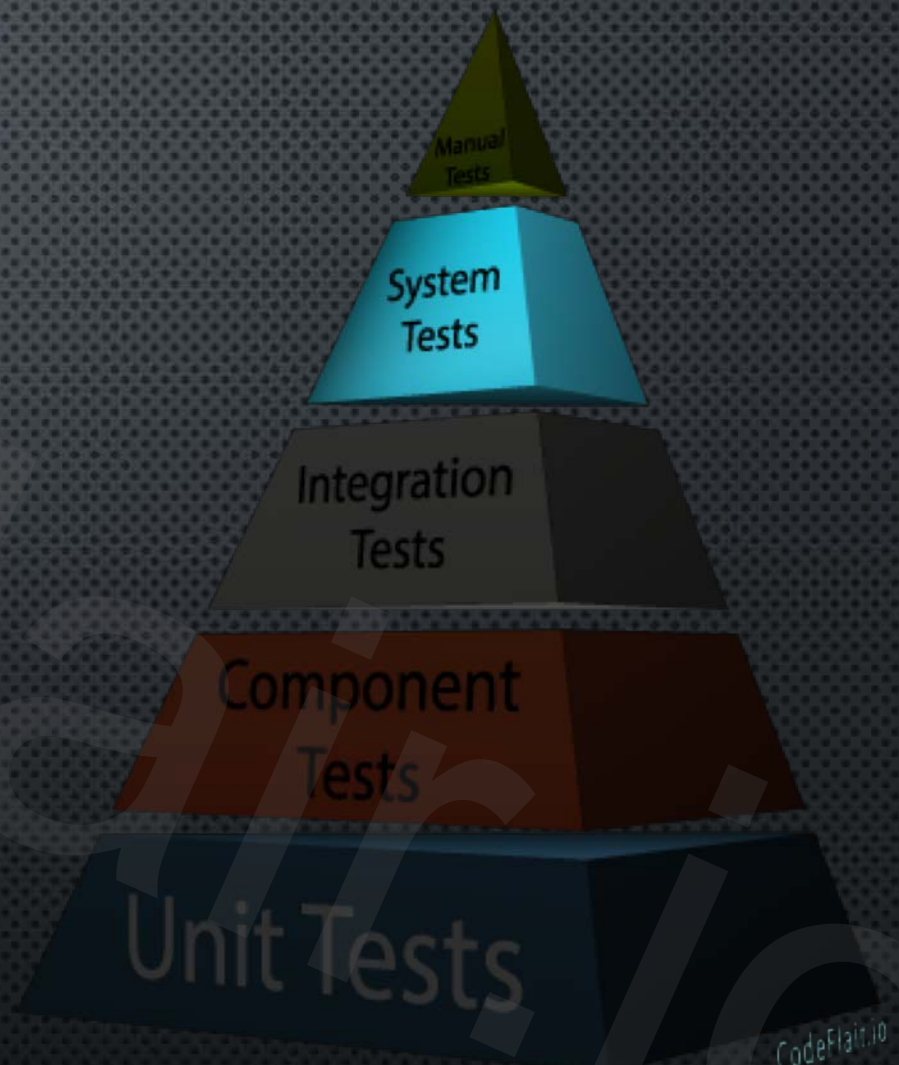
INTEGRATION TESTS

- ✓ Validate the linkage and communication between components
- ✓ Ensure a proper architecture is in place
- ✓ Target ~20% code coverage



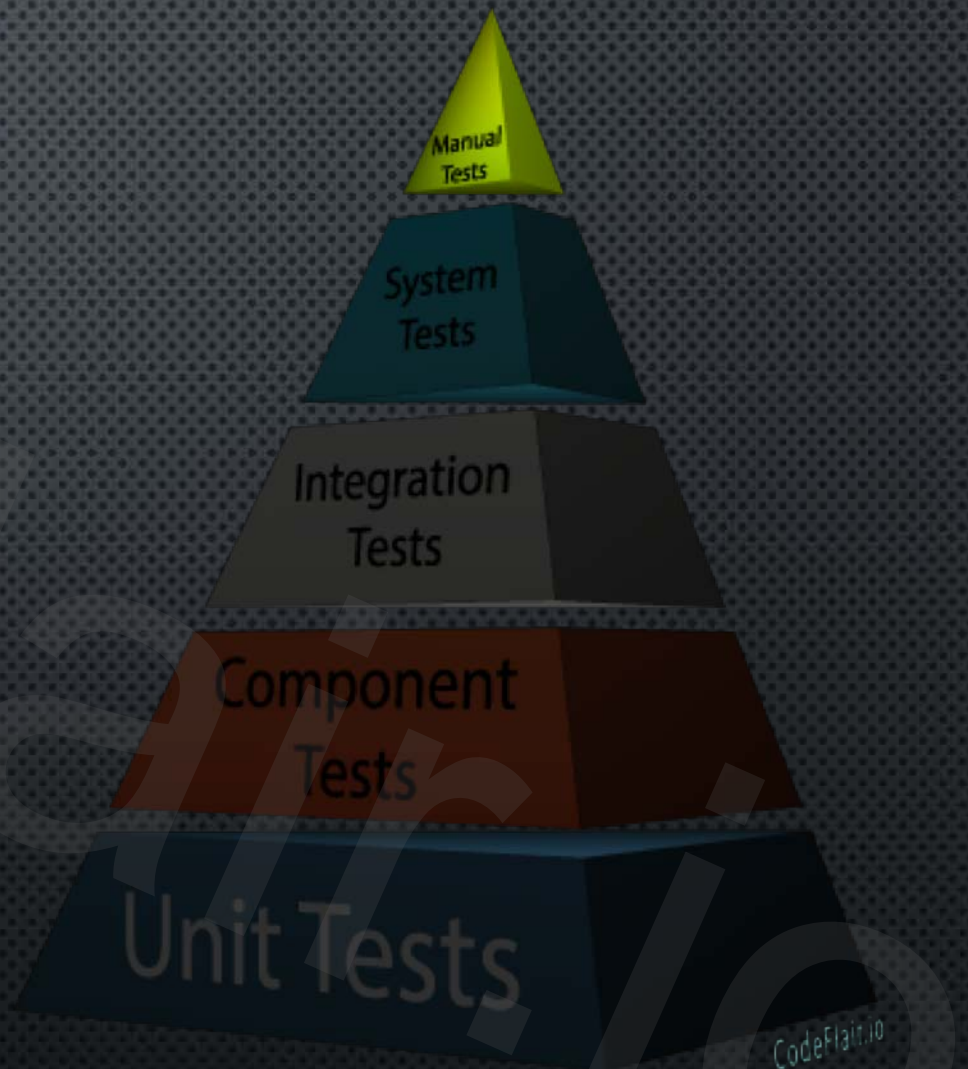
System Tests

- ✓ 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 TEST NAMING 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

TEST SMELLS

SYMPTOMS OF PROBLEMS

A CATALOG OF SMELLS

Code Smells

Project 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

BEHAVIOR SMELLS

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

Symptoms that are easily noticed

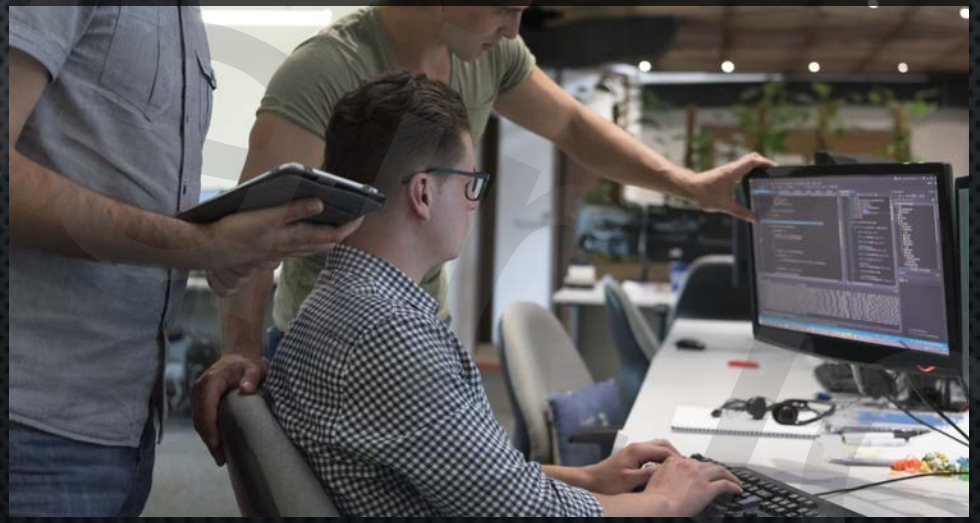
Behavior smells will take the form of compile errors or test failures.

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.

Dependency 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?

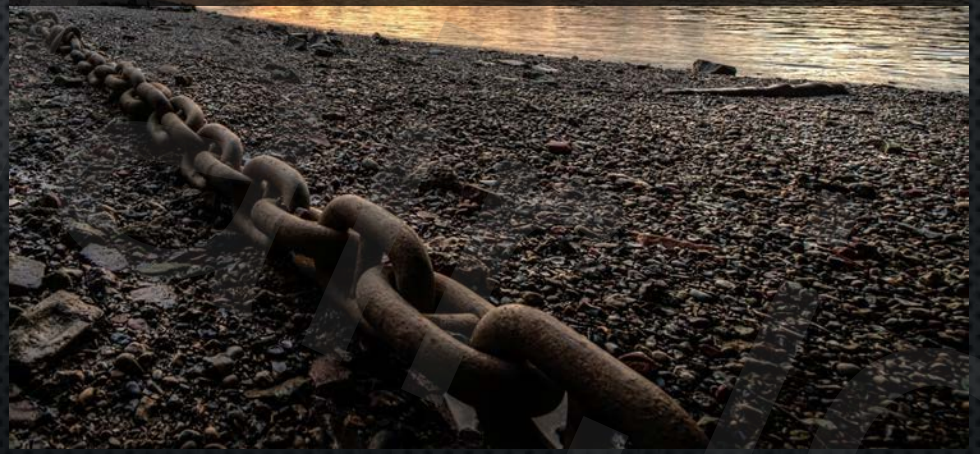
Which classes do we group together and which classes do we keep apart?

COMPONENT COUPLING PRINCIPLES

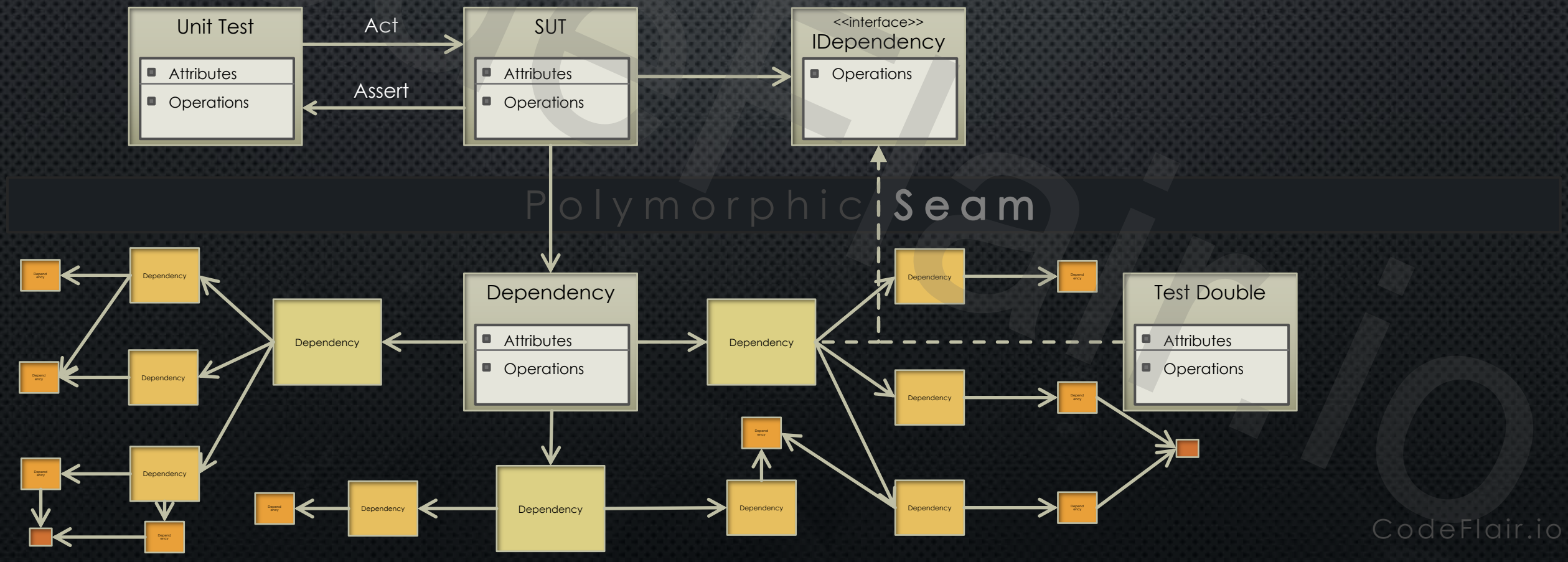
- ✓ **ACYCLIC DEPENDENCIES PRINCIPLE**
  THE DEPENDENCIES BETWEEN COMPONENTS MUST NOT FORM A CYCLE.
- ✓ **STABLE DEPENDENCIES PRINCIPLE**
  COMPONENTS SHOULD DEPEND UPON EACH OTHER IN THE DIRECTION OF INCREASING STABILITY.
- ✓ **STABLE ABSTRACTIONS PRINCIPLE**
  THE MORE STABLE A COMPONENT IS, THE MORE ABSTRACT IT SHOULD BE – SO THAT IT CAN CONFORM TO THE OPEN/CLOSED PRINCIPLE.

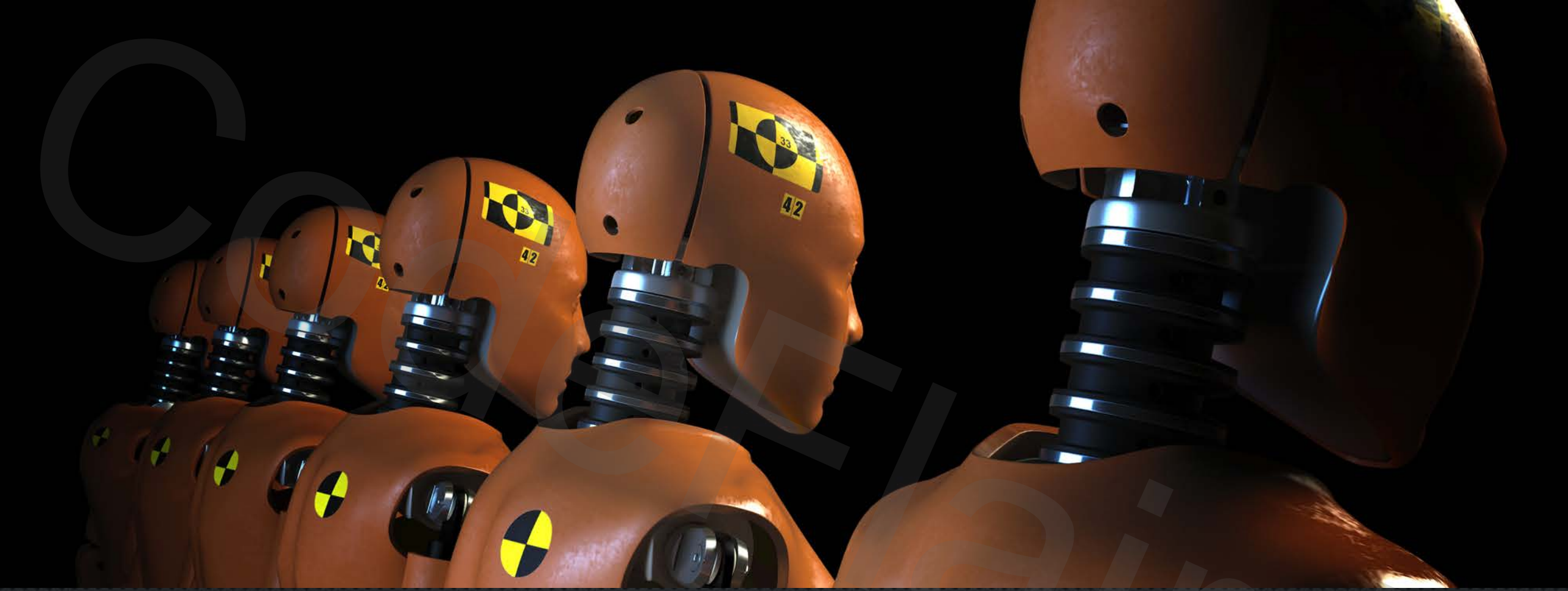
How are components related?

Effectively managing dependencies between components will facilitate testability.



KEEPING IT LOOSE



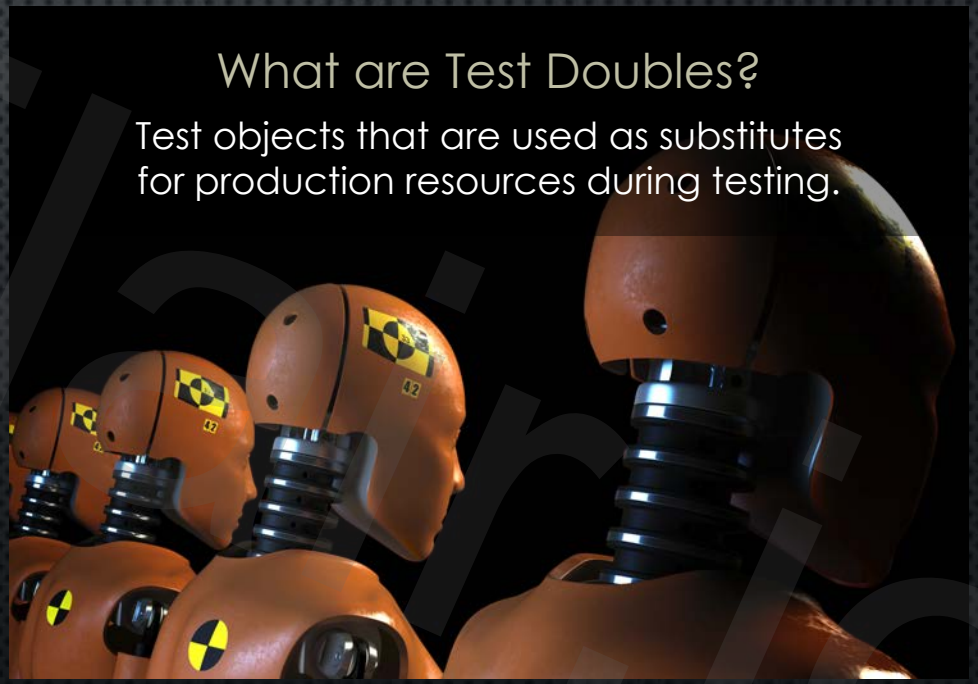


TEST DOUBLES

THE STUNT DOUBLES OF SOFTWARE TESTING

UNDERSTANDING TEST DOUBLES

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



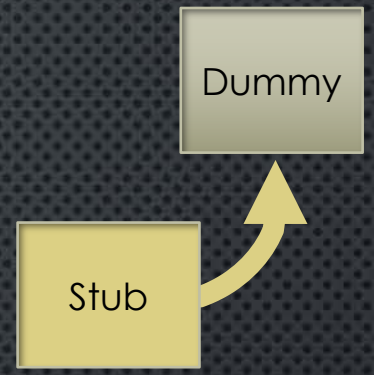
Dummy

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

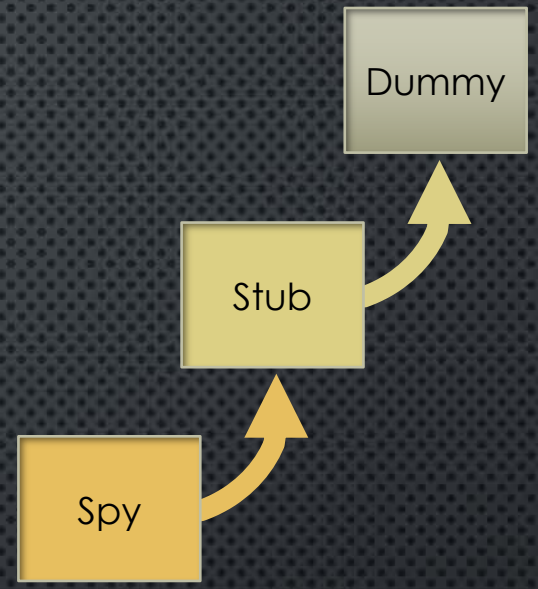


- ✓ A STUB IS A DUMMY
- ✓ ITS FUNCTIONS DO NOTHING
- ✓ RETURNS FIXED VALUES THAT ARE CONSISTENT WITH THE NEEDS OF THE TESTS

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

SPY

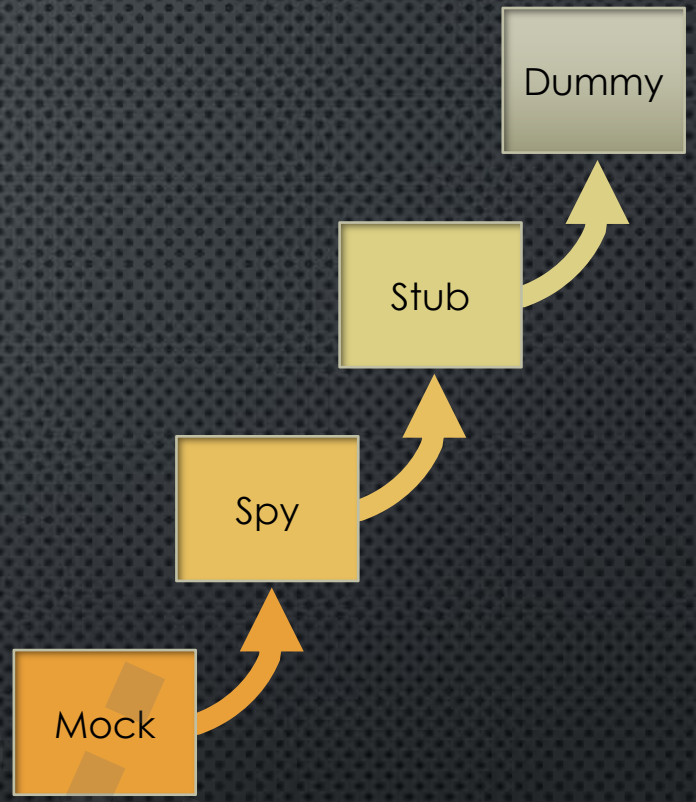
- ✓ A SPY IS A STUB
- ✓ ITS FUNCTIONS PERFORM NO EXTERNAL ACTIONS
- ✓ RETURNS VALUES THAT ARE USEFUL TO THE TESTS
- ✓ REMEMBERS FACTS SO THE TESTS CAN VERIFY THE FUNCTIONS WERE CALLED PROPERLY
  - ✓ WAS A FUNCTION EVER CALLED?
  - ✓ HOW MANY TIMES WAS IT CALLED?
  - ✓ HOW MANY ARGUMENTS WERE PASSED IN?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



THE ONLY THING A SPY DOES IS WATCH AND REMEMBER.

MOCK

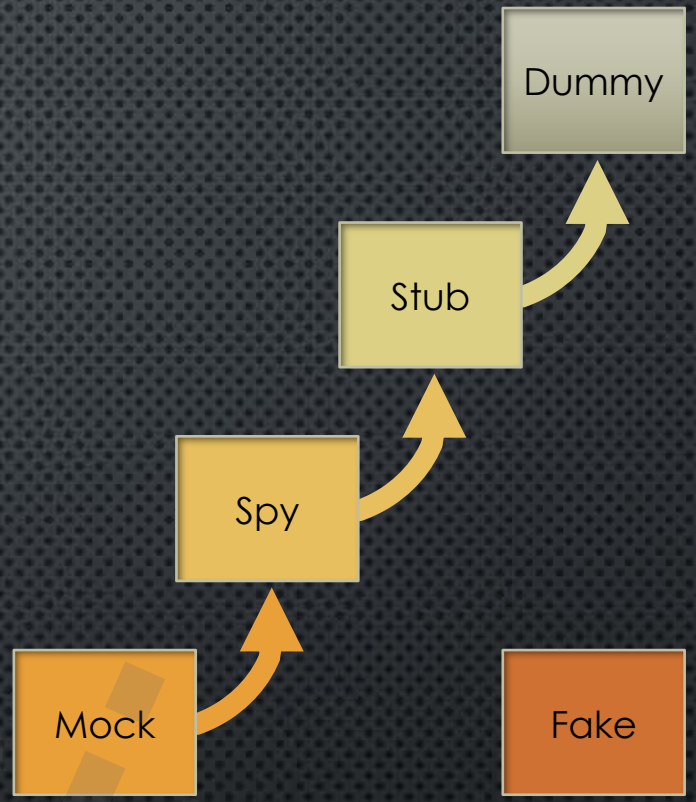
- ✓ A MOCK IS A SPY
- ✓ ITS FUNCTIONS DO NOTHING
- ✓ RETURNS VALUES THAT ARE USEFUL TO THE TESTS
- ✓ REMEMBERS FACTS ABOUT THE WAY IT WAS CALLED
- ✓ SETS UP CONDITIONS THAT ARE TO BE TESTED AND EVALUATES WHETHER THOSE CONDITIONS HAVE BEEN MET
- ✓ THE TEST DOES NOT CHECK WHAT THE MOCK SPIED ON



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.

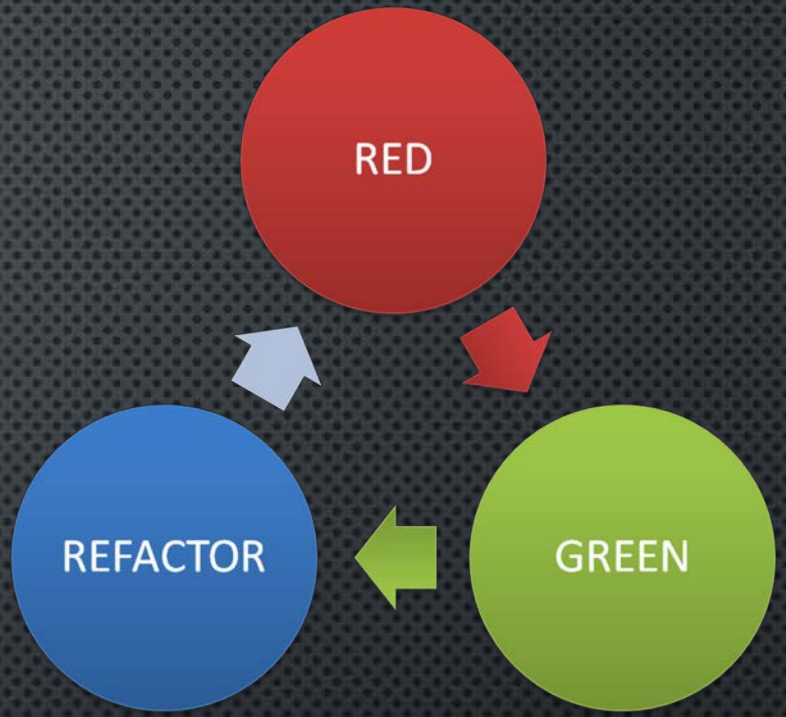
TEST DRIVEN DEVELOPMENT

TESTING CODE BEFORE IT IS WRITTEN

TEST-DRIVEN DEVELOPMENT

- ✓ eXtreme Programming
- ✓ It's not just about testing… it's more about design
- ✓ Architecture and implementation evolve from tests that are written prior to writing the code
- ✓ Short development cycle
- ✓ Validate the tests thru failure

RED → GREEN → REFACTOR



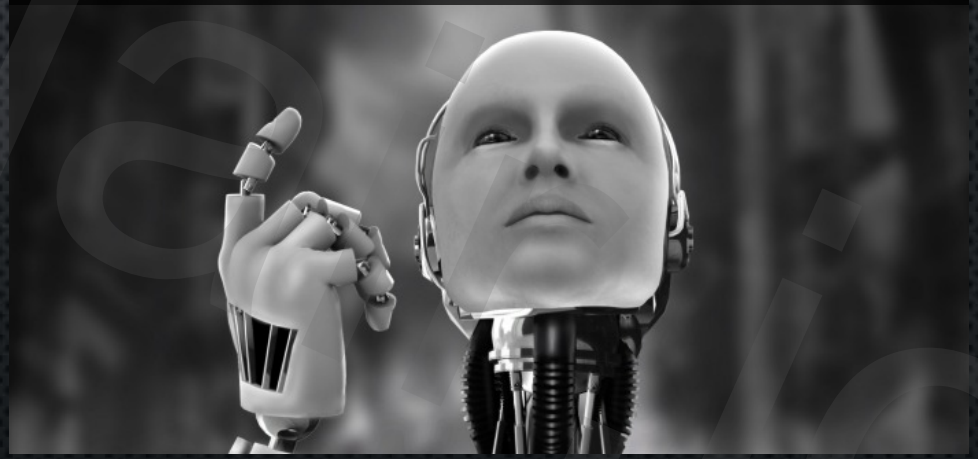
1. Make it Fail – Write a failing test
2. Make it Work – Write the simplest code to pass the test
3. Make it Better – Clean, improve, and optimize the code just written

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.

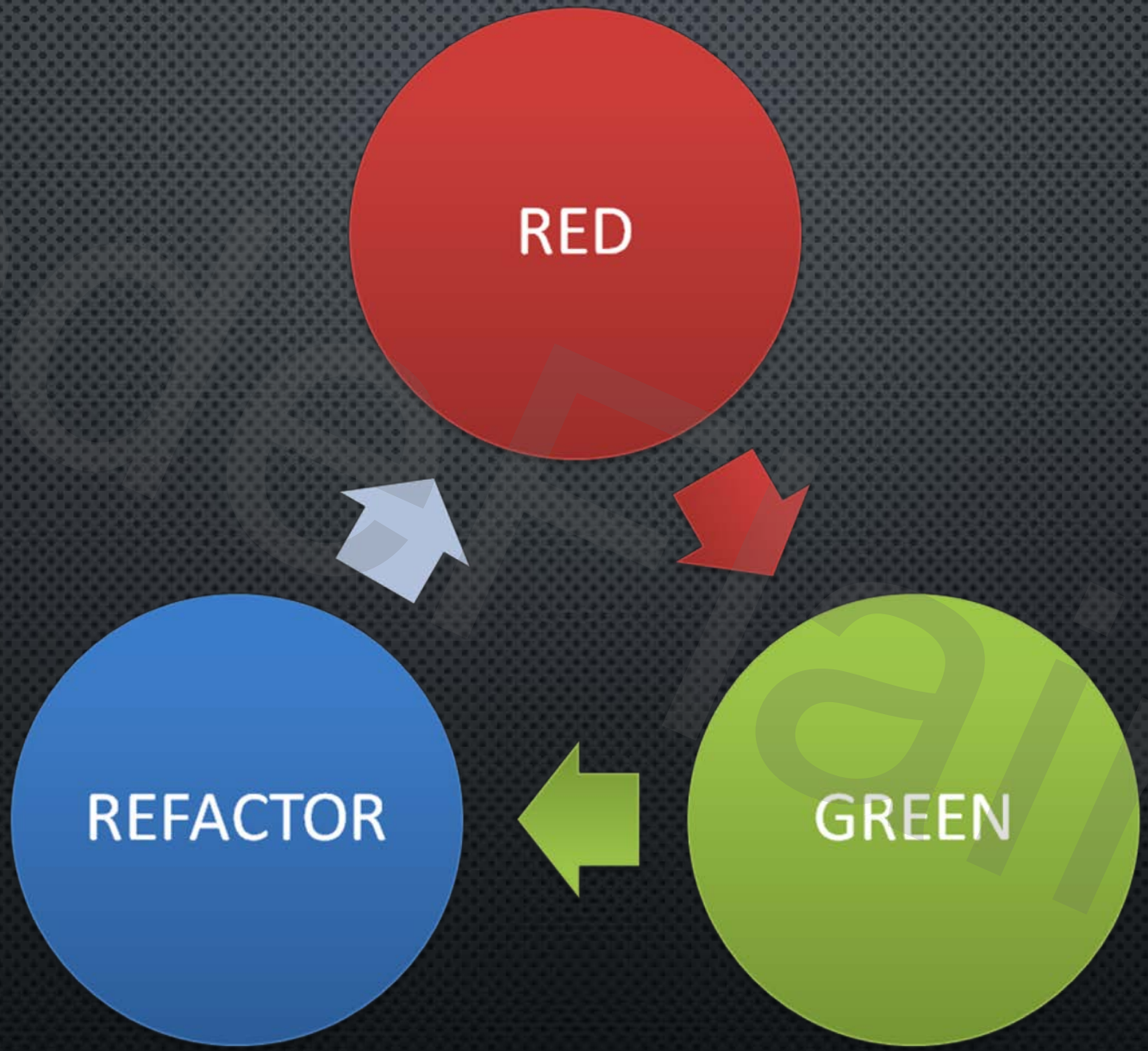


A JERRY BRUCKHEIMER PRODUCTION

GONE IN 60 SECONDS

NICOLAS CAGE

ANGELINA JOLIE GIOVANNI RIBISI ROBERT DUVALL



CONCLUSION

IT ALL STARTS WITH 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

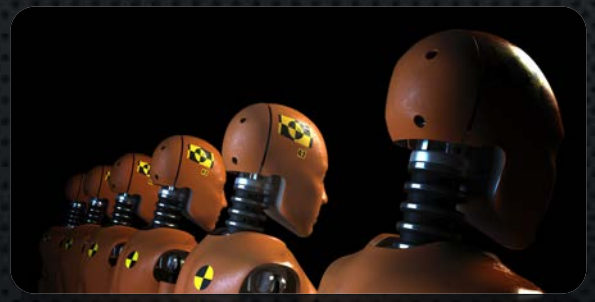
RELATED COURSES

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

 /Klugh

 /Klugh

ISO 9126 QUALITY MODEL

Functionality

- ✓ Suitability
- ✓ Accuracy
- ✓ Interoperability
- ✓ Security

Reliability

- ✓ Maturity
- ✓ Fault tolerance
- ✓ Recoverability

Usability

- ✓ Understandability
- ✓ Learnability
- ✓ Operability
- ✓ Attractiveness

Efficiency

- ✓ Time behavior
- ✓ Resource utilization

Maintainability

- ✓ Analyzability
- ✓ Changeability
- ✓ Stability
- ✓ Testability

Portability

- ✓ Adaptability
- ✓ Installability
- ✓ Co-existence
- ✓ Replaceability

REFERENCES