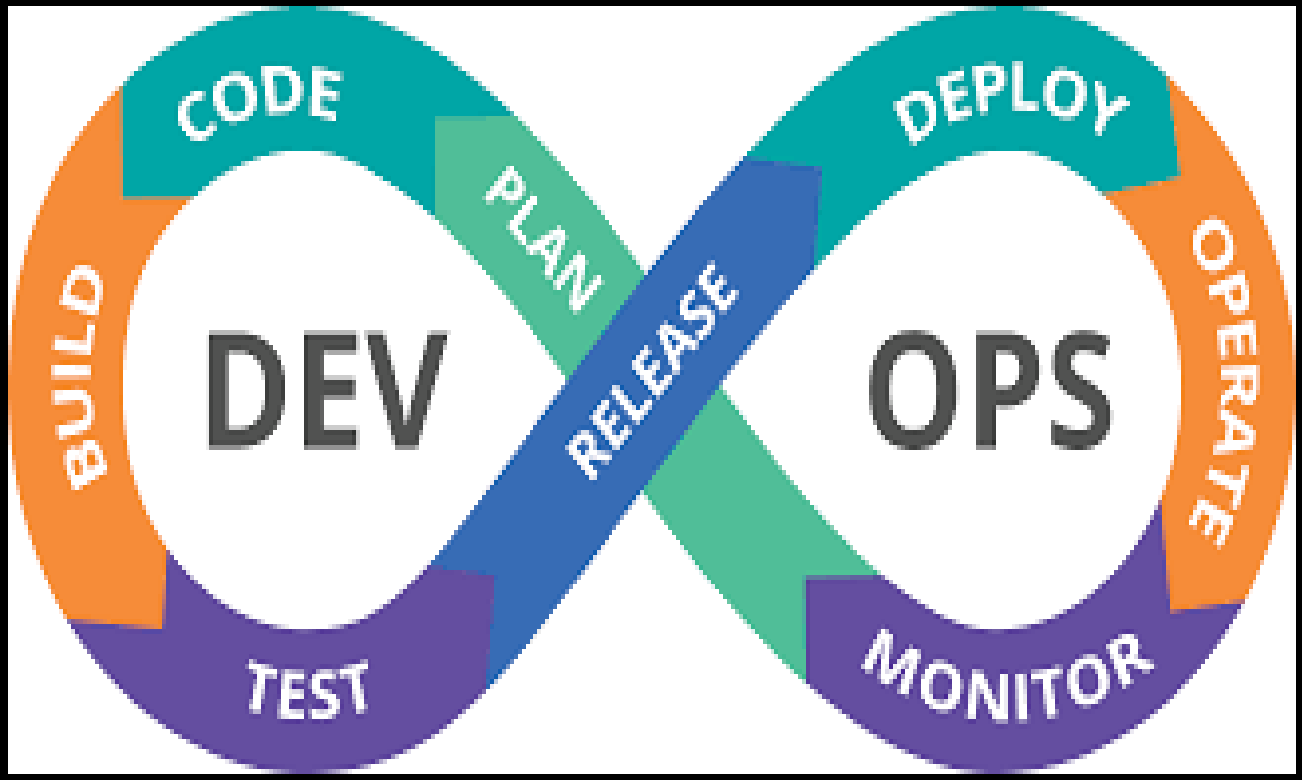
Automated Testing (Overview).

Axiom:

‘Code that isn’t tested doesn’t work’

‘Code that isn’t regression tested suffers from code rot (breaks eventually)’

Agile Software Development

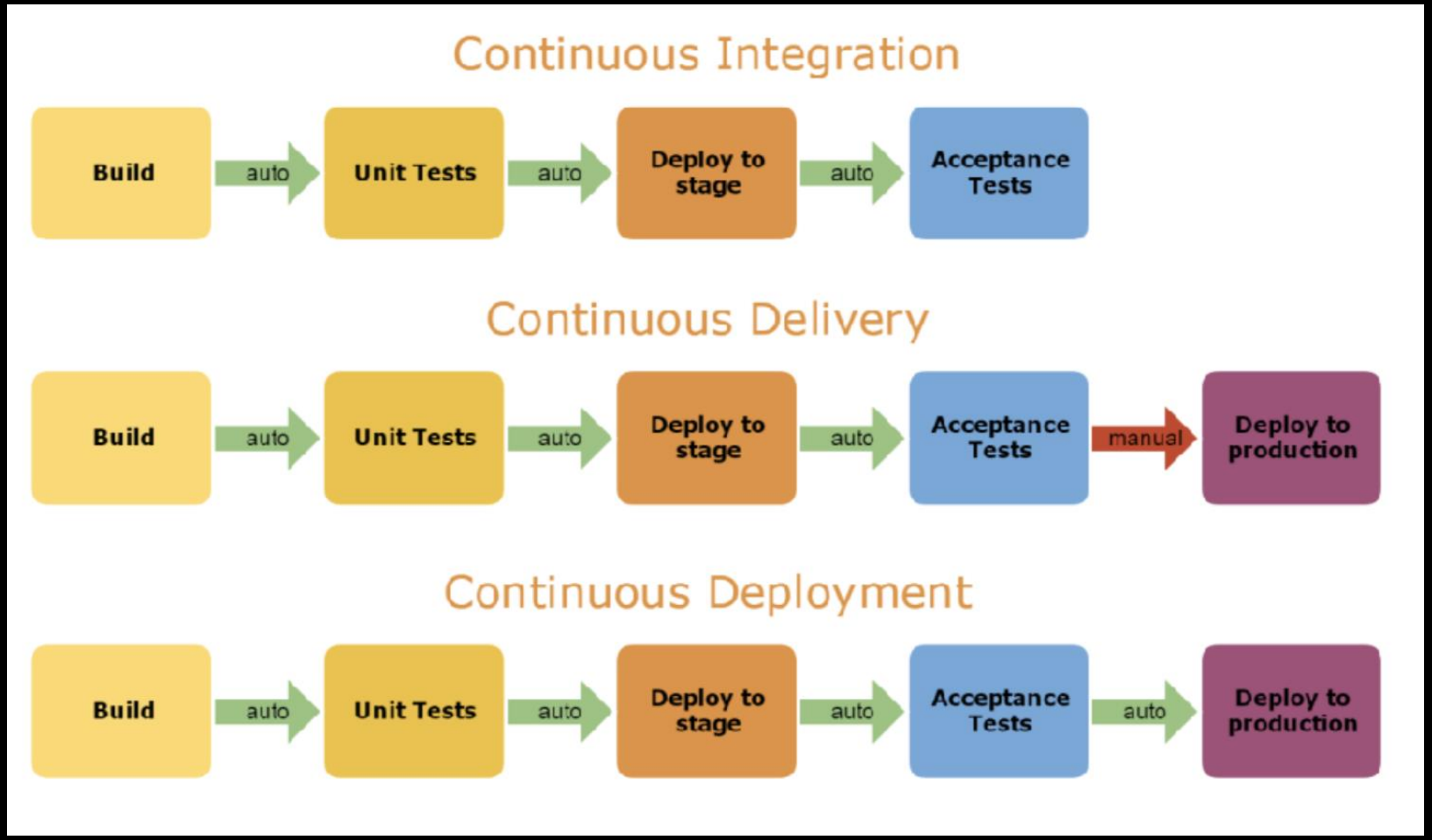


Testing levels

1. Unit testing - testing each discrete ‘unit’ of an application (e.g. class, module, function) in isolation.
2. Integration testing – testing the application works with external resources, e.g. databases, remote servers, cloud services.
3. System testing – testing the entire system/application from the UI perspective.
   - Functionality; Performance/Stress/Volume testing.
4. Acceptance testing – testing to customer’s requirements.

- Web app context: Unit testing; API testing; End-to-end (E2E) testing.

Automated Testing is core to CI/CD



Testing principles for Agile Dev

1. Use a test framework to facilitate automation.
2. All application code must have associated tests
3. When bugs occur, fix them AND write tests to prevent reoccurrence.
4. Follow the Test-Driven Development (TDD) process (TDD).

- Consequences:
  - Fewer bugs in released code.
  - During development, the code always works.
  - Schedule slippage =\= less testing.
  - Code is more maintainable.

Regression testing.

- Adding new functionality can affect (maybe break) existing functionality => Must re-run all existing tests, termed regression testing.
- It ensures a stable, maintainable code base.
- This means test code retains its “”value” over time.

Designing test cases.

- Pre-requisite: A clear specification of the required behaviour (functionality) of the target.
- We test the specification (What), not the implementation (How).
- Write tests in terms of the expectations of the target's behaviour for a given circumstance and specific inputs.

  e.g. I expect to get result X when the input is Y and the database has data Z.
- Test Target:
  - Unit testing – Function / Class.
  - System/Acceptance testing – End-to-end application.

Designing test cases.

- User stories are a product of User Requirements Analysis.

- User stories structure:

  Given some context

  When some action is carried out

  Then a particular set of observable consequences should obtain.

User Stories.

- Examples:

  Given a bank account is 500 euro in credit and the debit card limit is 200 euro

  When the user withdraws 300 euro

  Then the error message “Debit Card limit exceeded” displayed.

  Given the database has three movies that Cillian Murphy acted in

  When the user searches for all movies with Cillian Murphy

  Then the three matching movies should display.

User Stories.

- Given-When-Then also applies to unit testing,
- e.g. studentArray = [. . Student records . . .]

  getByRegion(region) {....} // Test target

  Given the array has students from Wexford (10), Waterford (20), Spain (2) and China (12)

  When the region is set to ‘county’

  Then the result should be:

  { “Wexford” : [. . . 10 students ..],

  “Waterford”’: [.. 20 students ...] }

  . . . . . .

  When the region is set to ‘continent’

  Then the result should be

  { “Europe” : [32 students.],

  “Asia”’: [3 students] }

Automated ***Unit*** Testing .

Simple Case Study.

- The Catalogue class.
  - The addProduct(product) method.
  - Spec: The method adds a product to the catalogue, provided it's id is unique, and the new product has a name and price. Boolean true is returned when a product is successfully added; otherwise false is returned.

Simple Case Study.

- addProduct() Test cases:
  1. Given the catalogue is initialized; When we add a valid product; Then true is returned.
  2. Given the catalogue has some products; When we add a product with an id already in use; Then false is returned.
  3. Given the catalogue has some products; When we add a product with no price; Then false is returned.
  4. Given the catalogue has some products; When we add a product with no name; Then false is returned.

Simple Case Study.

- We could test the class (target) with a custom script.
- PROBLEM: As the class expands, the number of test cases grows and the custom script becomes unmaintainable and difficult to understand.
- SOLUTION: Use a Testing framework.

Unit Testing framework.

- Unit testing frameworks improve the structure and maintainability of test code.
- Examples:
  - Java: JUnit.
  - C#: Nunit.
  - JavaScript: Mocha, Jest.

- Framework Features:
  - Impose a structure on the test set.
  - Provide a syntax for expressing test outcomes (expectations).
  - Allow flexible running options, i.e. run all / one / some tests.

The Mocha framework.

- Provides an execution environment for automated JS tests.
- Test code constructs:
  - describe blocks.
  - it blocks.
  - before / after hooks.
- Test code structure:
  - A test case is wrapped in an ‘it’ block.
  - Related test cases are grouped inside a ‘describe’ block.
  - describe blocks can be nested.
  - describe blocks can have ‘before’ (and ‘after’) hooks – execute code BEFORE each it block within a describe block.

The Chai library..

- The Chai library – used for expressing test outcome/expectation.

Mocha – Test code structure

```
describe('Unit name (e.g. Class name)', function() {
  beforeEach(function() {
     ... GIVEN (Initialize the context) ...
  });
  describe('Test group name (method name)', function () {
     it('state expectation', function() {
        . . . . WHEN (call target method with inputs) ..
        . . . . THEN (express expectation) . . .
      });
     . . . . Other it blocks …….. . . . .
  }) // end inner describe / test group
  . . . . Other test groups / describe blocks
 }) // end Unit test / outer describe
```

Terminology.

- Target – The application construct being tested, e.g. class, function.
- A target's test code is comprised of test cases.
- Each test case concerns one user story (GWT).
- The entire set of tests for an application is called a test suite.

Test code principles.

1. Test case isolation.
   - Each test case should execute independently of others.
   - Mocha's before (and after) hooks facilitate this objective.
   - Mocha does not guarantee test case execution order.
2. A test's expectations should focus on:
    1. Does target returns the correct result? AND
    2. Did the target correctly update its state? (if relevant)
3. The silent principle - Avoid unnecessary console output (except when debugging!!)
4. Have informative test case documentations,

Test case documentation

```
describe('Unit name (e.g. Class name)', function() {
  beforeEach(function() {
     ... GIVEN (Initialize the context) ...
  });
  describe('Test group name (method name)', function () {
    it('state expectation', function() {
       . . . . WHEN (call target method with inputs) ..
       . . . . THEN (express expectation) . . .
     });
    . . . . Other it blocks …….. . . . .
  }) // end inner describe / test group
  . . . . Other test groups / describe blocks
 }) // end Unit test / outer describe
```

How much testing?

- Test all scenarios of the target's behaviour.
- Three types of scenario:
  1. Normal – target's pre-test state is 'normal' and the inputs are 'normal'.

  2. Error – Invalid target inputs or ones that could cause an invalid state change, i.e. testing target's error-handling code.

  3. Boundary (Edge/Corner-cases) – inputs are just on their limit value or the target state is on its limit.

The Chai library

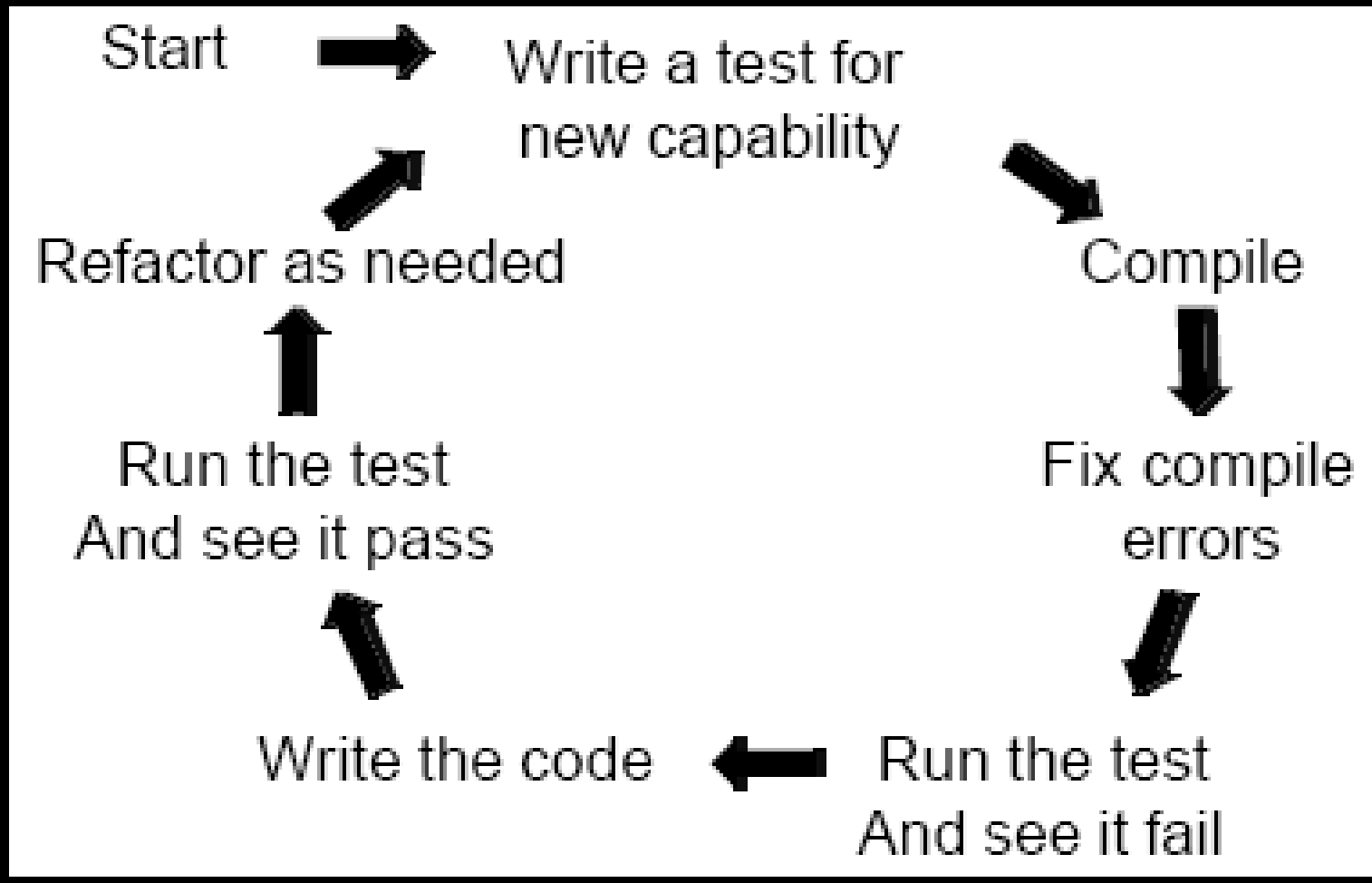
- Deals with a wide range of possible response/state types – primitives, arrays, objects, function

```
expect(result)
  .equal(value)
  .be.a('string')
  .include(value)
  .be.true
  .be.false
  .be.null
  .be.undefined
```

```
expect(res)
   .be.empty
   .gt(num)
   .gte(…)
   .lt(…)
    etc
```

```
expect(res)
.have.members([… ])
  .have.keys([…])
  .have.key(value)
 .be.function
  .be.instanceOf(….)
```

The Test-Driven Development (TDD) cycle



The Test-Driven Development (TDD) cycle

- Key points:
  - Write tests BEFORE the application code.
  - It's an iterative process.
  - Refactoring – Changing the source code design without affecting its behaviour.