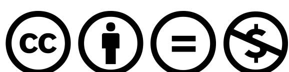
# Agile Programming Learning Outcomes



# LICENSING INFORMATION

The work in this document was facilitated by the International Consortium for Agile (ICAgile) and done by the contribution of various Agile Experts and Practitioners. These Learning Outcomes are intended to help the growing Agile community worldwide.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

### YOU ARE FREE TO:

**Share** — copy and redistribute the material in any medium or format

### UNDER THE FOLLOWING TERMS:

**Attribution** — You must give appropriate credit to The International Consortium for Agile (ICAgile), provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests ICAgile endorses you or your use.

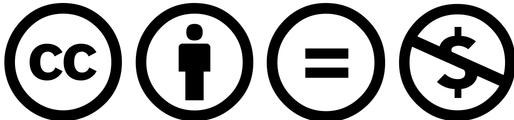
**NonCommercial** — You may not use the material for commercial purposes.

**NoDerivatives** — If you remix, transform, or build upon the material, you may not distribute the modified material.

### NOTICES:

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.



# SPECIAL THANKS

ICAgile would like to thank the contributors to the Agile Programming Learning Outcomes:
Alistair Cockburn • Colin Garlick • Gerard Meszaros

# CONTENTS


| Page | Section |
| --- | --- |
| 2 | LICENSING INFORMATION |
| 3 | SPECIAL THANKS |
| 4 | TABLE OF CONTENTS |
| 5 | HOW TO READ THIS DOCUMENT |
| 6 | LEARNING OUTCOMES |
| 6 | 1. AUTOMATED TESTS |
| 6 | 1.1. Unit Testing |
| 6 | 1.2. Readable Tests |
| 6 | 1.3. Test Coverage |
| 7 | 1.4. Testing Patterns |
| 7 | 1.5. Speed of Builds |
| 8 | 1.6. Test Doubles |
| 8 | 2. REFACTORING |
| 8 | 2.1. Code Smells |
| 9 | 2.2. Refactoring with Existing Tests |
| 9 | 2.3. Dealing with Legacy Code |
| 10 | 3. TEST-DRIVEN DEVELOPMENT |
| 10 | 3.1. TDD |
| 10 | 3.2. BDD |
| 10 | 4. ACCEPTANCE TESTING |
| 10 | 4.1. Living Documentation |
| 11 | 4.2. ATDD |
| 11 | 5. THE BUILD PROCESS |
| 11 | 5.1. Continuous Integration |

# HOW TO READ THIS DOCUMENT

This document outlines the Learning Outcomes that must be addressed by accredited training organizations intending to offer ICAgile's Agile Programming certification.

Each LO follows a particular pattern, described below.

0.0.0. Learning Outcome Name

*Additional Context, describing why this Learning Outcome is important or what it is intended to impart.*

The Learning Outcome purpose, further describing what is expected to be imparted on the learner (e.g. a key point, framework, model, approach, technique, or skill).

# LEARNING OUTCOMES

## 1. AUTOMATED TESTS

### 1.1. UNIT TESTING

#### 1.1.1. Types of Tests to Automate

*It is not enough to have just unit tests.*

Illustrate the difference between unit, component and acceptance tests, what benefits they each offer, when you should use each, how they fit into an automation strategy and how they affect the choice of tools.

#### 1.1.2. Fixture Setup

*It is easy to misunderstand test fixtures as trivial code.*

Write test fixtures that are clear to the reader, minimize duplication and execute fast. Apply both fresh and shared fixtures, creation and finder methods, persistent and transient fixtures, and test case classes. Structure test case classes per class, per feature or per fixture, and explaining how those influence the naming of test case classes and test methods. Note: by "fixture" is meant everything that needs to be present to verify the behavior.

### 1.2. READABLE TESTS

#### 1.2.1. Coding Tests by Intention

*A common misconception is that unlike program code, test code just needs to run and is not used for communicating to others.*

Write tests as readable as the best production code, that show to the next reader what is being verified. Make the four phases (setup, exercise, verify, cleanup) of each test obvious, expressing each phase at a suitable level of detail, and omitting anything that doesn't contribute to the specification of behavior.

#### 1.2.2. Verifying Results

*Part of creating tests is understanding how they communicate to the next programmer, help development and stay fast.*

Express expected outcomes so they are easy to understand and minimize code duplication. This includes the use of built-in and custom assertions, guard assertions, mock objects and test spies.

### 1.3. TEST COVERAGE

#### 1.3.1. Identifying Completeness Conditions

*Writing tests first provides a thinking structure for determining all the conditions that will need to be included in the target behavior.*

Practice and apply various heuristics to identify conditions that drive detailed design of required behavior. This includes positive, negative and exception conditions, as well as having the mnemonics to remember them by (CORRECT and Right BICEP as examples).

### 1.3.2. Avoiding Duplication in the Conditions

*It is easy to think that having more tests is better, whereas each additional test slows the feedback cycle, grows maintenance costs and eventually brings diminishing additional value.*

Identify and avoid test duplication to reduce the number of conditions used to drive the development of the code, and to select the highest value condition from a larger set of possible conditions (for example, concentrating on the different classes of output and the input values around where the behavior is expected to change, using boundary values or all-pairs technique).

## 1.4. TESTING PATTERNS

### 1.4.1. Listening to Your Tests

*Problems with the automated tests are often symptoms of poor design.*

Explain test smells that hint at the need for changes to the design to improve testability. Smells include fragility in tests, indirect tests, difficulty in writing tests and slow tests. Design changes to make include dependency injection, test doubles, headless apps, "humble" (plain-ordinary-testable) objects, design for zero-deployment testing and other ideas.

### 1.4.2. Testing the Tests

*Just because a test passes doesn't mean it is testing anything useful.*

Verify that the tests are, in fact, testing what you think they are. This includes learning the value of running the test before implementing the code, deliberately breaking the production code to verify the test, holding test inspections and writing unit tests for test utility code.

### 1.4.3. Refactoring Tests

*Sometimes the tests themselves grow too awkward to read.*

Explain when and how to refactor the tests to improve readability or performance. This includes indicators that it is time to refactor (code smells in the test code, behavior smells such as slow, interacting or unrepeatable tests and project smells such as tests not getting run), and practice refactoring to intention-revealing code.

## 1.5. SPEED OF BUILDS

### 1.5.1. Test Speed

*The speed at which the tests run has a surprising impact on the designer's ability to keep the design flowing.*

Explain the importance of getting feedback quickly, strategies for keeping tests that fast and strategies for dealing with the tests that can't be made that fast. This includes test-speedup strategies of stubbing out slow dependencies, keeping tests simple, minimizing fixture size, etc.

### 1.6. TEST DOUBLES

#### 1.6.1. Use Test Doubles

*It is all too common a mistake to test using the real database, thinking that test doubles are hard to use and make tests frAgile.*

Identify when and how to use test doubles, including stubs (that return canned answers), mocks (programmed with expected client behavior and fails on misbehavior), fakes (lightweight functional equivalents, used typically to go faster) and spies.

#### 1.6.2. Dependency Injection

*Creating unit tests should not result in constructors or setters that are only there for testing purposes.*

Explain how inversion of control (Icon) containers allow objects to be created using dependency injection (DI) constructors and setters but without needing to create complex chains of objects. Describe different dependency injection patterns, and how IoC containers are configured.

## 2. REFACTORING

### 2.1. CODE SMELLS

#### 2.1.1. Clean Programming

*Poorly written code is hard to understand and hard to evolve, making the codebase more expensive to maintain.*

Explain the value of writing easily understood code, and practice ways to do that. These include naming variables, methods, constants, functions, modules, classes, and so on; indentation; managing the sizes of methods and classes; guard clauses and conditional logic within methods; and encapsulation.

#### 2.1.2. Common Code Smells

*The industry has worked out generally agreed upon indicators that code is in bad shape.*

Assess code quality by recognizing code smells. The minimum set of specific code smells that must be learned are long methods, comments, duplicated code, large classes, speculative generality, and nested conditionals. Others may be included.

## 2.2. REFACTORING WITH EXISTING TESTS

### 2.2.1. Principles of Refactoring

*Refactoring is merely "rewriting code" – by now there are well established principles that keep the programmer moving forward and the code safe at all times.*

Explain the principles of refactoring, namely, starting with working code, identifying code smells, taking small steps that improve the code while preserving its overall behavior, so that there is always working code.

### 2.2.2. Common Refactorings

*Some refactorings are so common and basic that every programmer should be capable of them.*

Apply common refactorings. Mandatory for all languages are: Rename (variable, constant, function/class/module), extract method/function, introduce parameter, and inline method/function. Additionally mandatory for object-oriented languages are: split class, pull up, pushdown, extract interface.

### 2.2.3. Refactoring Tools

*Some people think that specialized refactoring tools are unnecessary, that they are nothing more than just typing faster.*

Explain the role that refactoring tools play in safely refactoring code, and the limitation of refactoring tools. This LO includes how the sequencing of refactoring can have an effect, the use of code-sniffing tools (such as PMD and others as they appear on the market), and the limitations introduced by reflection and dynamic languages.

## 2.3. DEALING WITH LEGACY CODE

### 2.3.1. Approaching Legacy Code

*Many programmers incorrectly think that code must be either refactored easily or thrown away and rewritten from scratch.*

Explain how to approach adding functionality to a legacy codebase. It includes being able to discuss refactoring versus rewriting; the danger of adding functionality without tests; the possibility of refactoring without tests; the refactoring-first approach and the adding-tests-first approach; and running two codebases in parallel (treating one as an oracle).

### 2.3.2. Refactoring Without Tests

*When refactoring a legacy codebase, sometimes refactoring first, before adding tests, is called for.*

Apply methods to refactoring code when adding tests isn't yet practical. This includes strategies for making changes without tests, legacy-specific refactorings without tests, and hands-on experience in a codebase that isn't designed for testability.

### 2.3.3. Retrofitting Tests Onto Legacy Code

*When refactoring a legacy codebase, sometimes adding tests before refactoring is called for.*

Apply methods for adding tests without refactoring. This includes exposing test APIs and dependency-breaking techniques such as object seams, linker seams and preprocessor seams.

# 3. TEST-DRIVEN DEVELOPMENT

## 3.1. TDD

### 3.1.1. The Value of TDD

*Test-driven development is often mistakenly assumed to be about writing tests for code, whereas it has much broader purpose.*

Demonstrate the value TDD provides to lower the cost of evolutionary development. Explain TDD's role in providing quick feedback cycles, improving the design of program interfaces, thinking about the design itself, as documentation, and for regression testing. It also includes configuring the development environment for quick turnaround (encouraging, for example, the absence of the debugger).

### 3.1.2. Red-Green-Refactor

*Despite common usage, TDD is not just a matter of writing tests, but is built on a rhythm of failing test, passing test, clean code – known as "red, green, refactor".*

Experience and explain the rhythm and benefits of driving development using the red-green-refactor steps, including a feel for the benefits of shorter cycle times.

## 3.2. BDD

### 3.2.1. Identifying Usage Patterns to Define the Object or Function Interface

*Writing tests before writing code gives the designer a taste for how the interface will feel in use, and provides concreteness to the proposal of its calling parameters.*

Practice and apply creating examples of intended usage to design and grow the interface before the implementation (this is also referred to as outside-in design, design from the client side, and example-driven-design).

# 4. ACCEPTANCE TESTING

## 4.1. LIVING DOCUMENTATION

### 4.1.1. Tests as Specification and Documentation

*It is easy to think that the only purpose of acceptance tests is to test existing code, and overlook their role in understanding what is to be built.*

Explain how using acceptance tests as a specification changes the way the product is built and eliminates documentation that would otherwise become outdated.

#### 4.1.2. ATDD as an Aid to Design Thinking

*Acceptance tests additionally reveal information about the modularity of the system's design.*

Illustrate how ATDD tests influence the modularity of the system by enabling/ encouraging independent testing of components.

### 4.2. ATDD

#### 4.2.1. ATDD Process

*Writing acceptance tests before starting to code supports the entire test-driven design sequence.*

Apply using acceptance test to drive development and how to step into the TDD cycle.

#### 4.2.2. ATDD Styles and Tools

*There are multiple forms, tools and formats for expressing automated acceptance tests.*

Classify the different ways ATDD tests can be captured and understand when each is appropriate.

## 5. THE BUILD PROCESS

### 5.1. CONTINUOUS INTEGRATION

#### 5.1.1.Continuous Integration

*Continuous integration is more than just having a build server and running the build once a day.*

Demonstrate checking in frequently to an automated build server that builds the system and runs the suite of tests against it. This includes explaining the benefit of reducing integration debt through frequent check-ins and fewer code branches.