

Agile Programming Techniques

Duration

3 days

Intended for

- Developers who wish to design and develop systems using Agile techniques
- Technical Testers wanting a deeper understanding of Test-Driven Development, Behaviour-Driven Development, and Acceptance Test-Driven Development.

Prerequisites

You need to have a working knowledge of Java, C#, or JavaScript.

We also recommend that you have completed some formal Agile training, such as our Agile Fundamentals course.



This course is about quality – in particular, what can developers do to make sure that we are building the right system, and that we are building it the right way.

If we want software development to move from being an art to becoming more of an engineering discipline, we need to become more structured and disciplined in what we do. We look at various technical practices of agile software development and how to apply them towards the goal of quality.

Each of the three days of this course is geared around a primary topic, with each topic building on those before it:

- Day 1 revolves around unit testing – how to write effective unit tests to ensure that the system is working. Additional modules include: using test doubles and dependency injection to write true unit tests without complicating the production code; improving the technical quality of the code base without breaking existing behaviour; and how to apply all of this to legacy code bases that were developed without any consideration given to automated testing.
- Day 2 revolves around test-driven development (TDD) – how to ensure that the system is well designed and developed. TDD is all about design, not testing. Good design and code results in a system that is easily maintainable. TDD applies unit testing to ensure that the system is working as designed.
- Day 3 revolves around specification by example – how to ensure that the system meets the users' and stakeholders' needs. Writing automated acceptance tests, and using these to drive TDD, ensures that not only is the system well built, but that it also is the right system.

Learning outcomes

By the end of the course you will be able to:

- Write and maintain effective unit tests
- Refactor code without breaking the observable functionality
- Identify the seams to add unit tests to legacy code bases
- Explain various principles of good software design
- Apply test-driven development (TDD) to write tested code
- Write good specifications / acceptance tests
- Automate the acceptance tests
- Perform software development as an engineering discipline.

Content

• Unit Testing

This module looks at the structure of a unit test and how to write them. Common patterns and good practices of unit testing are investigated, as well as the need for ongoing maintenance and curation of the test suite.

Agile Programming Techniques

- **Dependency Injection**

Proper unit tests should only verify a single module, and often require the use of test doubles to achieve this isolation. The different types of test doubles (stubs, fakes, mocks) are considered, as well as how to inject the test doubles into the unit under test and the use of inversion of control frameworks.

- **Refactoring**

Making changes to existing code runs the risk of breaking something that was working previously. This module looks at how changes can be made in a structured fashion to minimise the risk of change.

- **Legacy Code**

Safe refactoring requires unit tests, but what if the code doesn't have any unit tests? This module introduces how to use seams and enabling points to add unit tests to code that was not designed with testing in mind.

- **Design**

Technical quality is a high priority in Agile. Good design principles are introduced, including SOLID, GRASP, and simplicity. Technical debt is also discussed.

- **Test-Driven Development and Behaviour-Driven Development**

Most software development practices focus on building the right thing for the users, but ignore technical quality. TDD came about as a way to help build technical quality in. This module looks at how to do TDD, as well as how BDD helps to make TDD easier.

- **Architecture**

This module looks at how the architecture can drive the structure and organisation of the source code projects.

- **Team Collaboration**

This module looks at some of the collaborative practices within the team: shared accountability, teamwork and paired programming; version control; build tools; and continuous integration.

- **Acceptance Test-Driven Development**

While technical quality is important, an elegant system that doesn't meet the users' needs is useless. ATDD uses automated acceptance testing to drive TDD and ensure that the resulting system is acceptable to the users. Good practices of ATDD are considered under the name Specification by Example.

- **Cucumber**

The Cucumber tool allows us to write automated acceptance tests using a language common to (and understandable by) both the technical team and the business. This module looks at the structure of this language (Gherkin) and how to write the fixture code that automates the language.

- **FitNesse**

FitNesse provides easy accessibility for writing and running automated acceptance tests through a wiki.

Method used

This is a hands-on development course where the learning is achieved through applying the practices and techniques in programming exercises.

While the concepts and principles apply to most languages, the exercises are carried out using Java, C#, or JavaScript.