Agile Software Design Learning Outcomes

International Consortium for Agile

LICENSING INFORMATION

SPECIAL THANKS

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

CONTENTS

2 LICENSING INFORMATION

3 SPECIAL THANKS

4 TABLE OF CONTENTS

5 HOW TO READ THIS DOCUMENT

6 LEARNING OUTCOMES

6 1. DESIGN

6 1.1. Architecture

6 1.2. Design-In-The-Large

7 1.3. Sacrificial and Evolutionary Architecture

7 1.4. Design for Automated Testing

7 1.5. Technical Debt

8 2. COLLABORATIVE DEVELOPMENT

8 2.1. Technical Leadership

8 2.2. Communication

9 3. DESIGN PRINCIPLES

9 3.1. Simplicity - Simple and Good Design

9 3.2. Patterns

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. DESIGN

1.1. ARCHITECTURE

1.1.1. Enterprise Architecture

Software development is only useful if it helps the organization achieve its goals.

Convey the value of enterprise architecture and aligning the IT strategy with the business strategy. Enterprise architecture provides three core services to the organization - strategy, standards and support.

1.1.2. Views and Viewpoints of Stakeholders

There is no one-size-fits-all perspective that will meet the needs of all stakeholders.

Discuss that different stakeholders have different needs and concerns, resulting in a variety of different viewpoints. Different models may be required for each of these viewpoints, even though they may refer to the same view (or aspect) of the system.

1.2. DESIGN-IN-THE-LARGE

1.2.1. Capability and Application Architecture

It is a common misconception that in Agile, any form of global design is bad, that all design must be immediate, small and refactored into a macro shape.

Introduce the roles of coarse-grained design, of dealing with costly-to-change decisions and of evolutionary architecture. This LO includes such things as choosing implementation languages, operating systems and technology frameworks.

1.2.2. Design Throughout the Lifecycle

Design is a process that continues throughout the life of the project.

Explain what design is required throughout the different stages of the project. The nature of the work and the level of any deliverable varies depending on whether we are evaluating the initial viability of the project, planning the work, building the product, or preparing for a release.

1.2.3. Sequencing the Work

Highest-business-value first is a good initial approach to sequencing work, but not always the best.

Discuss how to sequence work across functional, non-functional and risk aspects. This includes selection strategies such as easiest first, hardest first, highest business value next, highest ROI, technical risk, technical debt, minimized rework and possibly others.

1.3. SACRIFICIAL AND EVOLUTIONARY ARCHITECTURE

1.3.1. Sacrificial Architecture

Good architecture plans for change, but not all change can be accommodated.

Discuss how accumulated change can eventually overwhelm an architecture, requiring a new architecture and a possible rewrite. Continuing with an architecture that is no longer suitable makes further work difficult and reduces agility.

1.3.2. Evolutionary Architecture

Architectures can change over time.

Compare some strategies (e.g. a complete rewrite, the Strangler pattern) for changing an existing architecture to a new structure.

1.4. DESIGN FOR AUTOMATED TESTING

1.4.1. Testing the System Bypassing the User Interface

Too many people think that testing a system requires going through the user interface.

Practice and apply strategies and techniques for automating acceptance tests without using the user interface, including the architectural support required.

1.4.2. Testing the System Through the User Interface

Not being able to test bypassing the user interface does not mean that the ATDD strategies cannot be used.

Explain strategies and techniques for building maintainable and intention-revealing automated acceptance tests when it is not possible or practical to bypass the user interface. This includes using adapters or wrappers around the user interface so that good UI-agnostic acceptance tests can be used.

1.4.3. Non-Functional Testing

It is a common misconception and bad habit to think that testing non-functional attributes (also called cross-functional testing) gets done only at the end of the project.

Describe the role of test automation in verifying cross-functional attributes, including capacity and response time, security, etc., and how to fit these into an Agile process.

1.5. TECHNICAL DEBT

1.5.1. Recognizing Technical Debt

The cruft that accrues in the code either from neglect or intentionally when hitting deadlines reduces code malleability – it is a technical debt that needs addressing.

Describe the concept of technical debt and its mounting negative impact on productivity, and discuss how to recognize this technical debt. This includes understanding technical debt born from taking short cuts, learning better ways of doing things and gaining new technical capabilities that help simplify existing code. This also includes recognizing technical debt by symptoms such as high cost in adding new functionality, high frequency of new bugs when changing code, difficulty in automating testing and code that is difficult to understand.

1.5.2. Discussing Technical Debt Choices with Stakeholders

Programmers all too often complain that they have to create technical debt just because the boss says to, even despite the boss saying not to.

Model dialogue with various stakeholders about technical debt and what (if anything) should be done about it.

2. COLLABORATIVE DEVELOPMENT

2.1. TECHNICAL LEADERSHIP

2.1.1. The Need for Technical Leadership

Technical leadership is more than just delivering a design.

Define technical leadership and what it entails. This should address the Why (do we need it), What (is it), How (do we do it), When (is it needed) and Who (should exhibit it).

2.1.2. Characteristics of Technical Leadership

How we work as technical leadership is just as important as what we do.

Introduce the fact that **how** the technical leader works with others affects the value and acceptance of the work they produce. This also includes looking at how some behaviors such as "ivory-tower-architecture" can be a hindrance rather than a help.

2.2. COMMUNICATION

2.2.1. Stakeholder Engagement

How we engage our stakeholders is crucial, not just for identifying the requirements, but also for giving them confidence that we are able to meet their needs.

Explain who stakeholders are. Discuss the need for a communications plan for our stakeholders and strategies to assist in developing one.

2.2.2. Development Team Engagement

Development teams with no commitment to an architecture are not likely to apply that architecture.

Describe why an architect needs to be a salesperson and leader, inspiring a team with the architecture. Contrast how different strategies achieve this or work against it.

3. DESIGN PRINCIPLES

3.1. SIMPLICITY - SIMPLE AND GOOD DESIGN

3.1.1. Simple Design

It is natural to think that the more complicated design is the better one; however, simpler designs better retain agility in the codebase and therefore also in the company.

Illustrate the need for and value of "simple designs" in order to avoid over-engineering; when faced with alternatives, to choose the simpler of them. This should include several taste-tests for simple design, such as single responsibility, McCabe complexity, Beck's 4 rules of simple design, "whichever is easier-to-test", etc.

3.1.2. Evaluating Designs and Design Principles

If it is still not known how to invent good designs, quite a lot is known about how to evaluate designs.

Practice evaluating designs and applying concepts/techniques that make them easier to evolve. This should include practice in some of the known published principles and techniques: DRY principle, SOLID, etc.

3.2. PATTERNS

3.2.1. Design Patterns

Design patterns are idioms that solve common or difficult design problems.

Teach and practice a few of the basic and well-known design patterns. This includes balancing use of design patterns with "simple design", noting that design patterns can complicate the design unnecessarily; and learning to refactor to design patterns.

3.2.2. Architecture Patterns

Architecture patterns address the broad structure or big picture of the system.

Introduce a few of the basic and well-known architecture patterns. The patterns discussed should include layers, model-view-controller and microservices.

3.2.3. Patterns for Continuous Delivery

Continuous delivery is more than just automating the deployment.

Introduce common patterns for continuous delivery. This includes the latent code patterns of branch-by-abstraction, feature-toggles, dark-launching, canary-releases and blue-green deployment. Call out to the DevOps track for a much deeper view of DevOps.