COMP33711 - Agile Software Engineering

Christopher Williamson

January 2, 2017

Contents

1 Introduction

'We are uncovering better ways of developing software by doing it and helping others to do it. Through this work we have come to value:

- **Individuals and interactions** over processes and tools.
- **Working software** over comprehensive documentation.
- **Customer collaboration** over contract negotiation.
- **Responding to change** over following a plan.

That is, while there is value in the items on the right, we value items on the left more.'

1.1 Key Ideas for Agile Approaches

One of the key ideas is to avoid waste which commits people to delivering real value. Trust is also a key idea. This includes having a self-organising team, empowering developers to make key decisions and including the customer as a member of the team. Simplicity of both the process and product is important. YAGNI (You Aren't Gonna Need It) is a principle that is often followed which basically means that a programmer should not add functionality until deemed necessary.

Finally, feedback is another key idea. We should aim to 'fail fast' by making sure that failure is visible quickly. Short iteration, close customer involvement in development, team retrospectives, high-coverage automated test suites and pair programming are all useful idea that aid the feedback process.

1.2 12 Agile Principles

To help people gain a better understanding of what agile software development is all about, the members of the agile alliance refined the philosophies captured in their manifesto into a collection of twelve principles:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter time scale.
4. Business people and developers must work together daily throughout the project.

5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversations.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity - the art of maximising the amount of work not done - is essential.
11. The best architectures, requirements, and designs emerge from self-organising teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.

1.3 N Agile Practices

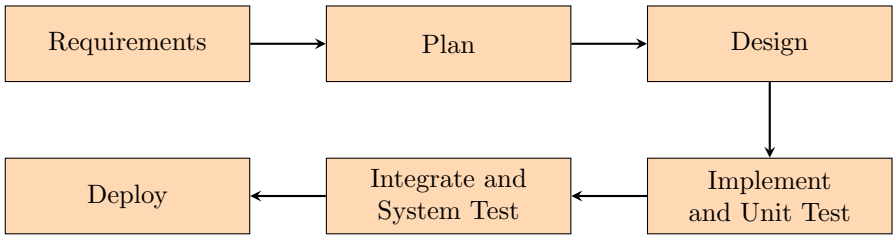
There is no limit on agile practices but some examples are short iterations, user stories, story points, planning games and TDD.

1.4 M Agile Methodologies

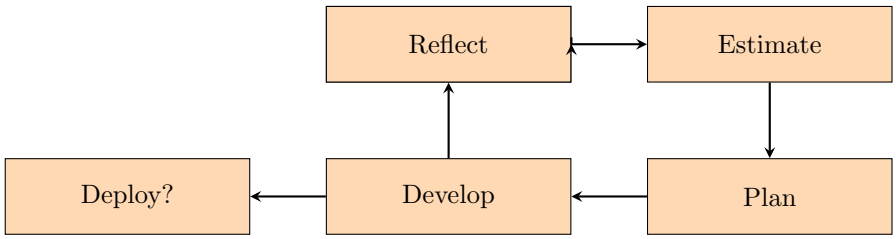
As above, there is no limit on agile methodologies but some examples are eXtreme Programming (XP), Scrum and Crystal.

1.5 Waterfall VS Agile

When using the waterfall approach, we usually have the following structure:



When using the agile approach, our diagram looks like this:



This diagram shows that the process is iterative and incremental as we loop back to the start (with a reflection stage) at the end of every development cycle (iteration).

1.6 Iteration Stages

1.6.1 Agile Practices: User Stories

Agile takes a different approach to requirements gathering. Instead of using a typical requirements specification, requirements are gathered as 'user stories'.

Story: As a student, I want to know which course units I'm not reaching my target average mark in.			
Business value:	200		
Effort:	Small	Medium	Large

1.6.2 Estimation

You are given a set of user stories. Before starting to build, you need to decide what to commit to.

1.6.3 Planning

Next, you select the stories from your backlog that you will implement in the iteration. The key here is balancing business value achieved, effort expended, dependencies between stories and the amount of learning gained from implementation effort. Documenting the plan can be done by using a task board.

1.6.4 Development and Showcase

Next you implement as many of your selected stories as you can and show the customer what you've achieved. The customer will sign off the stories that have been completed satisfactorily and the rest will be put into the backlog for the next iteration.

1.6.5 Feedback

Finally we hit the feedback stage where we receive feedback from both the customer and the development team in the hope that the feedback will improve out process for the next iteration.

2 The Trouble with Big Upfront Requirements Gathering

2.1 Advantages of a BUFR Specification

- Means that the initial requirements can be put into a contract so that the customer is tied to them.
- As the requirements won't change, you can work out how much it'll cost and how much time it'll take to build.

These advantages are based on some assumptions:

- There exists a reasonably well-defined set of requirements if we only take the time to understand them.
- Changes to requirements will be small enough that we can manage them without substantially rethinking or revising our plans.
- Software innovation and the research and development that is required to create a significant new software application can be done on a predictable scale.

If all of these assumptions hold it should be fine to proceed with big upfront requirements gathering.

2.2 Change in Software Requirements

What can cause requirements to change?

- The customer can change the budget.
- They don't know what they want through the whole process.

- The scale of the project may change.
- There may be a big technology change.

Given these, there is one major source of change that is often unacknowledged. Changes can occur even on projects where the requirements are not very volatile when the change is in our understanding (customer & developers) of what the requirements are as we build the system.

2.3 Written Requirements Specifications

BUFR means that you have to write the specification down. Written specifications are a lot of work and have to constantly be kept up-to-date. Natural language is inherently imprecise and ambiguous which leads to people jumping to their preferred interpretation which can cause problems further down the development process where the developers and the customer had a different understanding of the meaning of the specification. People are also psychologically less likely to question the correctness of something if it is written down.

On an agile project we gather requirements throughout the whole project rather than just at the beginning. We also put off requirements gathering for as long as possible. We try to stray away from writing down all of the requirements in a formal requirements document and we use face-to-face conversations between customers and business analysts/developers to communicate requirements to the people writing the code.

The key points here are that BUFR would be the best way to gather requirements if the underlying assumptions held in practice. Unfortunately they do not. Requirements are rarely clear from the outset, they change frequently, we learn the most about them after delivery and written documents are usually more trouble than they are worth.

Instead, agile approaches aim for just in time/just enough iterative, incremental requirements gathering based on conversations between customers and developers.

3 How Do We Know What to Build?

To avoid BUFR, we need an approach to requirements gathering that is:

- Just-in-time, just-enough
- Iterative AND incremental
- Value driven

An iterative process model is, at its simplest, one in which an artefact is created by repeating a process more than once. This implies that the artefact created

by a single pass through the process is incomplete. In an incremental process, we divide the system to be built into smaller units that are each in themselves complete. In each pass through the process, we work to create one of these units (called increments), so that at the end of each pass we have a version of the system that delivers only a part of the full requirements but which is complete in that it can be executed and maybe even deployed and used.

Our highest priority is to satisfy the customer through early and continuous delivery of valuable software. There are several value oriented agile practices we can use to aid in this such as impact mapping, user stories and road-maps but it all depends on having a good understanding of what value you are aiming for. An aim is like a behaviour change:

- Get more visitors to our website
- Get more customers to buy our product
- Get claim assessors to make fewer mistakes when deciding to pay-out for insurance claims

Our first step is to clarify the value. We should ask what impacts are we aiming to achieve and link it to something concrete such as saving money/time. Denne and Cleland-Huang suggest the following types of value for software:

- Revenue generation
- Cost saving
- Competitive differentiation
- Brand projection
- Enhanced loyalty

3.1 User Stories

A user story is any brief description of the functionality the software system we are building might usefully have. All of the following are acceptable examples:

- I want to see my current average overall mark for the semester and the year
- I can find out which course units I'm signed up for
- Students are reminded when the deadline for making course choices is approaching
- Show list of the course units the student can pick from

We want to specify features that are about to be implemented in detail and specify a high level picture for features that will not be implemented for a

while. User stories can be written at different levels of abstraction which can cover these ideas. Compare the following user stories:

- I want to watch films on demand over the internet.
- I want to manage the collection of films I've purchased for on demand viewing.
- I want to continue watching a film I've already started to watch, from the point I left off.

As you can see, the first user story is at a high level of abstraction whereas the final one is quite a low level story.

Each story should deliver a useful, usable version of the system (an increment).Each story should describe a (thin) **end-to-end slice** of functionality. Many beginning story writers find it helpful to use a template when writing stories. We will use the Connextra template:

As a *type of person*, I want *some functionality* so that *some value is created*

On a real agile project, the use of a story template is optional. We are using it because it is well known, prompts us to include important elements of the story and avoids 'blank page syndrome'. Some examples of this template in use are below:

- As an existing customer, I want to browse the list of newly acquired films, so that I can quickly discover films I might want to watch.
- As a company purchaser, I want to know what the most popular new films are, so that I can focus our licensing discussions on those that will maximise return to the company.
- As a media purchases manager, I want to compare projected and actual revenue from each film, so that I can make better decisions about how to adjust our models for the next quarter.

3.1.1 Gathering User Stories

Some ways of gathering user stories are story writing workshops, impact mapping and story mapping. Story mapping is where we tell the story of how the system will be used and then add details of how the software will support the story.

3.1.2 The INVEST Properties for Useful User Stories

As writing good stories can sometimes be difficult, Bill Wake coined this mnemonic for the properties that we want out user stories to have:

- **Independent:** the stories should be able to be implemented in any order (i.e. they shouldn't be technically dependent on each other)
- **Negotiable:** no aspect of the story is deemed unchangeable throughout the lifetime of the project.
- **Valuable:** the story should have real value to some customer/stakeholder.
- **Estimable:** it should be possible to estimate the size of the story.
- **Small/Appropriately Sized:** stories should be sized relative to the iteration size the team is using.
- **Testable:** It should be possible to devise a concrete and unambiguous test that can tell us whether the story has been implemented or not.

4 How Do We Know What to Build Next?