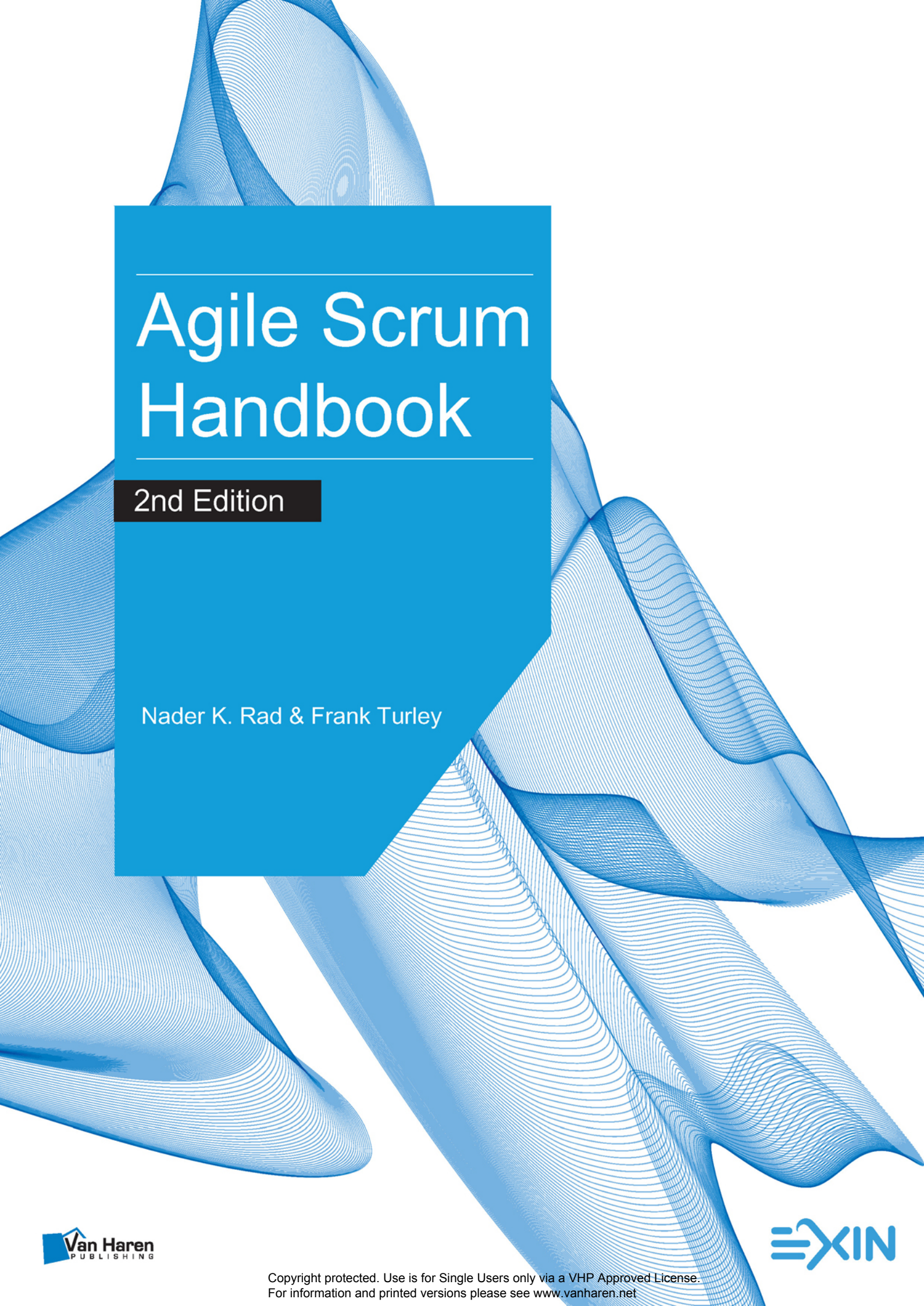


Agile Scrum Handbook

2nd Edition

Nader K. Rad & Frank Turley

Van Haren PUBLISHING

EXIN

Agile Scrum Foundation

Colophon

Title: Agile Scrum Foundation
Authors: Nader K. Rad & Frank Turley

Publisher: Van Haren Publishing, Zaltbommel ISBN
Hard copy: 978 94 018 0279 6
ISBN ebook: 978 94 018 0278 9

Edition: First edition, first edition, October 2014
Second edition, first edition, May 2018

Design: Van Haren Publishing, Zaltbommel

For further information about Van Haren Publishing please e-mail us at: info@vanharen.net

Table of Contents

About the Authors

Nader K. Rad is a project management author, speaker, and adviser at Management Plaza. His career started in 1997, and he has been involved in many projects in different industries. He has designed a number of project management courses, prepared a number of e-learning courses, and written more than 40 books.

Nader has been an official reviewer or consultant for PRINCE2® 2017, PRINCE2 Agile®, P3.express, and EXIN Agile Scrum Master™.

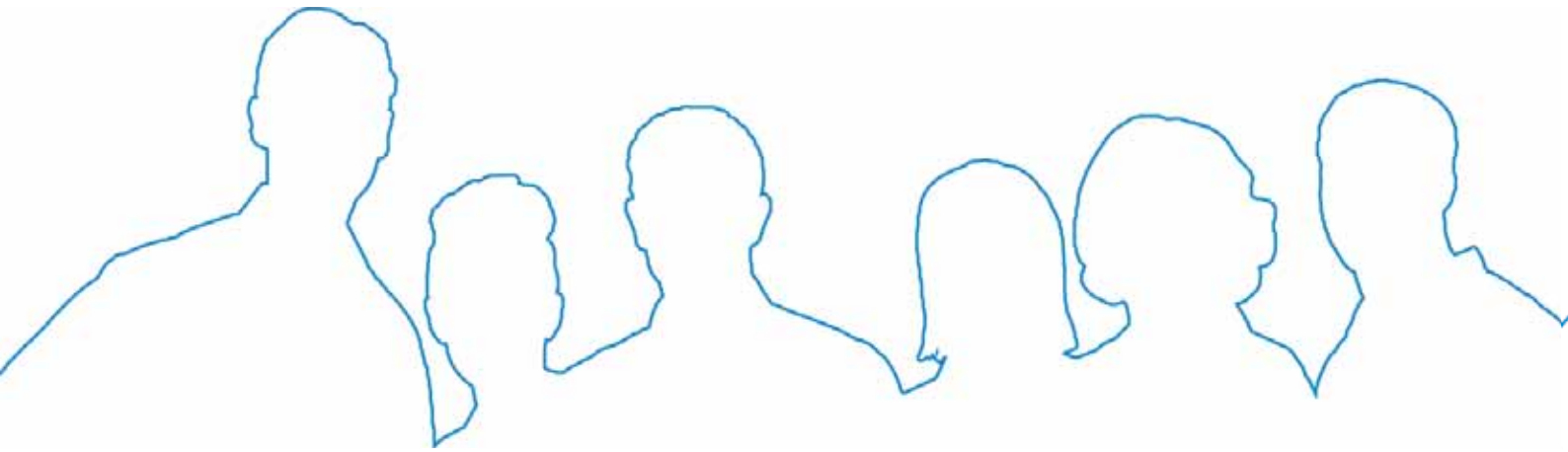
More about the author:	http://nader.pm
Author's website:	https://mplaza.pm
Author's LinkedIn Profile:	be.linkedin.com/in/naderkrad

Frank Turley has been a project manager for more than 15 years. He is a PRINCE2® Practitioner, a Scrum Master, and a PRINCE2 and Project Management trainer and coach. He has written a number of PRINCE2® and Project Management related books and is best known in the PRINCE2 world for his work in creating the most popular PRINCE2 self-study training materials.

More about the author:	https://mplaza.pm/frank-turley/
Author's website:	https://mplaza.pm
Author's LinkedIn Profile:	http://linkedin.com/in/frankturley

1

Agility Concept



The Agility Concept

If your goal is to learn something that can benefit you in your projects, you should consider two crucial things that are misunderstood most of the time:

1. What you may often hear is “Agile is a mindset”. The fact is that Agile needs a mindset, like everything else, but it’s not correct to say that it is a mindset. Saying “Agile is a mindset” has only one practical consequence: being able to work as you wish, just calling it Agile, without accepting criticism and looking for real improvements.
2. If you have the slightest familiarity with the way authoritarian systems work, you know that they always need to have an *enemy*. This concept fills in the gaps they have in their system and helps them control the crowd. Many Agile practitioners use the word “waterfall” to refer to the enemy; and while this “waterfall” is never clearly defined, they imply that it’s about the established project management systems. If your goal is success in projects, you don’t need the illusion of an external enemy; and you should remember that any successful system builds on top of the existing systems instead of starting from scratch; and while criticism is absolutely necessary, it has to be with respect and knowledge.

So, let’s start talking about the real nature of Agile.

Project Delivery Method and Lifecycle

When you’re developing a piece of software, the following steps are done in one way or another, for separate features, or for the solution as a whole:

- Analyze
- Design
- Construct
- Integrate
- Test

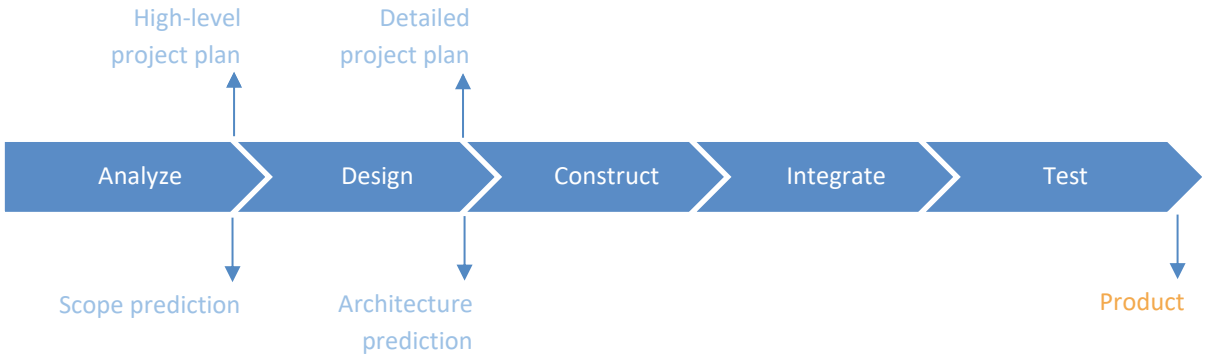
You can, of course, use other names for those steps, merge them into fewer steps, or split them into more; that’s fine. These steps can be called *delivery processes*.

Now, the question is, how are you going to arrange and run these processes? Think about a few options before reading the rest of this chapter.

So, how many options did you think of?

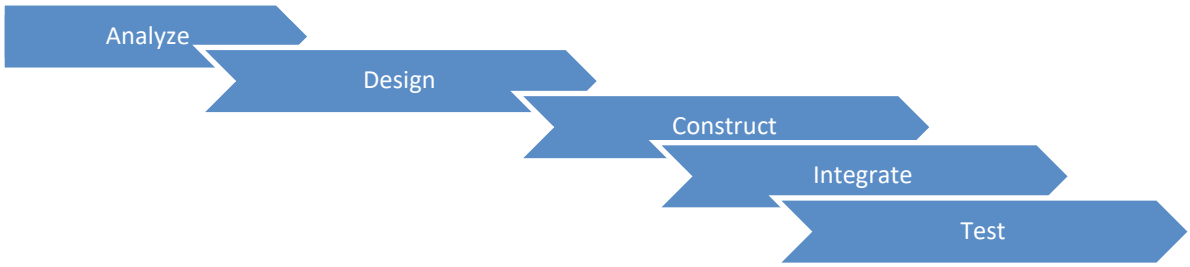
You may have many options in mind, but they should all belong to one of the two generic forms. By the way, these options can be called the *development lifecycle*.

One generic lifecycle is something like this:



In this lifecycle, each process is completed before we proceed to the next; i.e. we completely analyze the requirement and decide what we want to have in the solution; then we design the architecture of the solution and find out what's the best way to form the features. Then, programmers start working on different units, and then units are integrated into one solution, and then the solution is tested.

Obviously, the steps can overlap; e.g. you don't need to wait until all units are complete before integrating and testing them. Your lifecycle may look like this:



This is not fundamentally different from the previous lifecycle; we still have a sequence of development processes as the main driver for the lifecycle.

As you see, this type of lifecycle is based on an initial effort to understand what we're going to produce. We have an upfront specification, upfront design, and consequently, an upfront plan. That's why some people call it a *plan-driven* development. Also, we try to predict what we want and how it can be produced, and that's why a common name for it is *predictive*.

Predictive lifecycles are the normal and appropriate way to develop many projects, such as a construction project. You plan and design first, and then follow them. However, this is not a comfortable way to work for some projects.

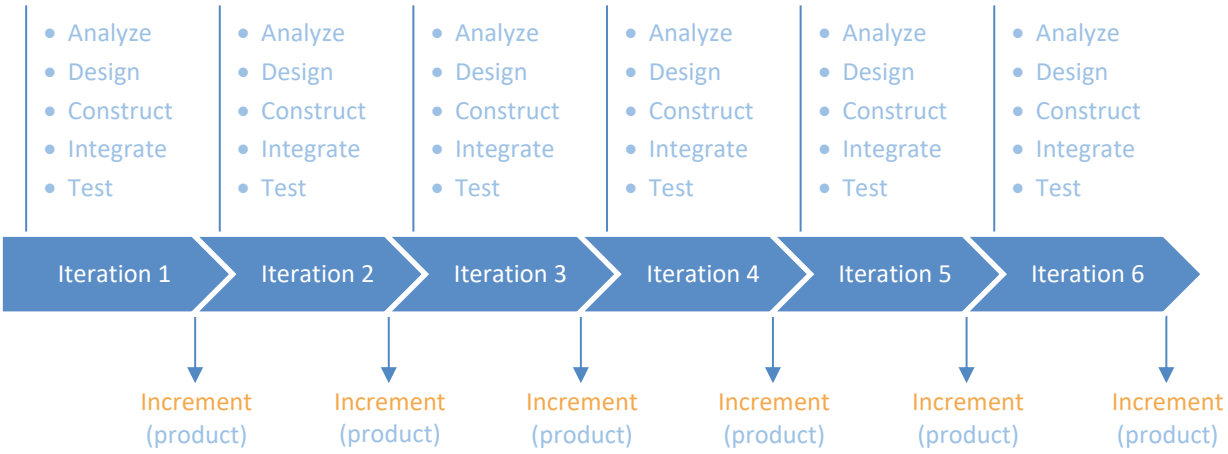
Think of a typical IT development project. You can spend a lot of time specifying and analyzing the requirements, and then base everything else on that. What happens next? The customer won't be happy when they see the result! They will ask for changes, and changes are expensive in this lifecycle because you may have to revise all the previous work.

As it's common to say in this industry, the customer doesn't know what they want until they see the product. When do they see the product? Towards the end of the project. At that point, cost of change is maximum.

To overcome this problem, we can sacrifice the comfort and structure of a predictive lifecycle and use one that creates the product *incrementally*, i.e. in multiple versions, each time with more features. This is a luxury we have in IT development projects that not everyone else can have: multiple versions of working software, each time with more features. Think about a construction project; there are no meaningful increments for that, and the product is not usable until the end.

To be fair, this disadvantage of a construction project is balanced with the fact that if you start a project to build a hospital, it doesn't matter how many changes you have, the final result will be a hospital, not, for example, a theme park! However, in IT development, you may start a project to create something like a hospital and end up with something like a theme park.

So, we can have incremental delivery in IT development projects; let's exploit this opportunity by a lifecycle like this:



There's no real prediction in this lifecycle. Instead of predicting the product and relying on that, we have short periods of time in which we create increments of the product. We will show the increment (latest version of the product) to the customer and end-users, receive their feedback, and decide what to do in the next period of time. So, instead of prediction, we go on with the project and *adapt* to the feedback. What would you like to call this lifecycle? "Adaptive" is a great name: adaptive lifecycle.

To create each increment, we need to run all development processes during that period of time. In the next period, we will *repeat* the processes: we *iterate*. That's why this method of development is sometimes called *iterative development*. Respectively, the periods of time within which we iterate can be called iterations. This is not the only name used for that; you may already know at least one more name for iterations. We'll get back to this topic soon.

Predictive vs. Adaptive Lifecycles

Each of the predictive and adaptive lifecycles have advantages and disadvantages. The right choice depends on many factors, but the most important one is the nature of the product.

You can ask two essential questions before deciding the type of lifecycle you need for your project:

1. Do I need to be adaptive? Because if you don't, a predictive lifecycle is…well, it's more predictive! It's easier and more structured. An adaptive system is needed when there is a risk of starting with the idea of creating something like a hospital and ending up with something like a theme park.
2. Can I be adaptive? This question is even more important. To be adaptive, you must have the possibility of developing *iteratively* and delivering *incrementally*. Let's think of a construction project once again: can you design the building iteratively? Can you design the foundation without designing the rest of the building that will determine the amount of load on the foundation? The answer is simply NO! It's not possible to have iterative development for a construction project. Also, incremental delivery is not possible, as we discussed before. So, we can't use an adaptive lifecycle to build a building (don't confuse this with interior design and decoration, or even renovation, for which we may be able to use an adaptive system).

My main message is that Predictive vs. Adaptive is not a matter of good and evil.

As a little practice, think about an IT project for upgrading the operating systems of 300 computers in an organization or an IT project to create a networking infrastructure for a very large organization with offices in six locations. In your opinion, which development lifecycle is more suitable for these two projects?

Agile vs. Waterfall

"Agile" is the popular name for systems that use Adaptive lifecycles. That's how one can really define Agile, instead of saying "Agile is a mindset"!

Agile "fans" use the word Waterfall to refer to Predictive lifecycles. The word Waterfall is commonly used to refer to Predictive lifecycles used in IT projects; you don't hear people saying "This building was built using a Waterfall method".

To make sure you know everything when it comes to terminology, you should be aware that the word Waterfall is practically a curse word these days, and you have the right to get angry and offended if someone tells you that you are using Waterfall! That's why I suggest we use the more formal name in this book: Predictive lifecycle.

Is Agile New?

Agile is usually advertised as the new thing. The use of the term Agile to refer to Adaptive lifecycles is certainly new, but what about the lifecycle itself?

I don't know about you, but I have a hard time imagining a long history of human beings with many projects and programs that have been done without any form of adaptive lifecycles. Can you think of an example?

I can give you one. Think of a very popular initiative (project or program) in the old days: going to war. Can you manage a war with a Predictive approach? They plan and design everything in the beginning? Certainly not. You may have a high-level plan that is more like a strategy than a plan, and manage the war one battle (i.e. iteration) at a time (or a few in parallel), and based on the outcome of each battle, adapt for the rest of the initiative.

Not a pleasant example, but a clear one that shows Adaptive lifecycles can't be new.

So, what is it that is new?

In a certain time, the so-called scientific management approach and Taylorism became the norm, so much so that every other approach was perceived inferior and even wrong. Taylorism was entirely and strongly based on Predictive systems; therefore, Predictive systems dominated the whole world, so to speak.

Then we reached the time that more and more IT development projects were initiated, and Predictive lifecycles were not really the best way to manage those projects. People tried to tolerate it, while the pressure was increasing, until demonstrations and eventually revolution happened! Like any other revolution, it devours its children, but that's a topic for another time.

The Agile Manifesto

Some people started using Adaptive systems for IT development and gradually structured them into repeatable management processes. A group of these pioneers got together in 2001 to make it official by giving it a name and creating a manifesto.

Let's start with the name. As legend has it, the two final options were Agile and Adaptive. Unfortunately, their decision was Agile. Adaptive would be much better because it describes the nature of the approach and prevents many misunderstandings.

So, the Agile Manifesto, which is available in the highly advanced and modern website at AgileManifesto.org, is this:

We are uncovering better ways of developing software by doing it and helping others 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 the items on the left more.

Kent Beck	Ward Cunningham	Andrew Hunt	Robert C. Martin	Dave Thomas
Mike Beedle	Martin Fowler	Ron Jeffries	Steve Mellor	
Arie van Bennekum	James Grenning	Jon Kern	Ken Schwaber	
Alistair Cockburn	Jim Highsmith	Brian Marick	Jeff Sutherland	

Unfortunately, this manifesto itself has never been subject to adaptation during its life.

A usually overlooked part of the manifesto is the last sentence. I'd like to invite you to read the manifesto again with the last sentence in mind.

So, let's review these four statements.

Value 1: Individuals and interactions over processes and tools

Overlooking the importance of individuals and interactions is a very fast way to fail. After all, it's the people who do the project. Some managers think they can overcome problems in this area by using a more sophisticated "system", but that rarely, if ever, works.

Many of us have been disappointed by the naive optimism that implementing a sophisticated tool will solve problems caused by overlooking human aspects, or even methods, for that matter. Still, managers spend huge sums of money implementing and maintaining tools, hoping for them to do magic. The fact is that tools can only facilitate a system; they don't replace the need for a system. On the upside, these tools are sophisticated pieces of software that need years of development and maintenance and create many projects and jobs, which make it possible for us to invest in thinking about better ways to do IT development projects!

The part about processes in this statement is a little tricky. It's actually about a certain type of process, not processes in general. It's about processes that are designed to replace the need for human interactions and complexities. I personally know managers who believe that if they have a better process, they

won't have to hire highly expert professionals. In the meantime, one great aspect about Agile systems is that they have ALL embedded human aspects in their processes, instead of just bolting them in or even just talking about the importance of human aspects, which is, unfortunately, the case with established project management systems.

So, in summary, processes that try to ignore or replace human aspects are bad, and processes that address those aspects and make them part of the system are good.

Value 2: Working software over comprehensive documentation

In contrast to the previous statement, which is correct for all types of projects, this one is specific to Adaptive systems. It refers to the fact that, instead of using upfront documentation to predict what needs to happen in a project, we just go on, create pieces of working software (increments), and use them to adapt.

Value 3: Customer collaboration over contract negotiation

Any project would be more successful with higher levels of customer collaboration. In Adaptive systems, it's more than important; it's necessary. The customer has to collaborate with you all the time when you're constantly specifying new requirements and asking them to check the increments and give you feedback. If they don't do it, you won't be able to adapt.

And contract negotiation is something we all love ;) An ideal Agile project with a time and material contract and a customer that doesn't think suppliers are criminals doesn't need much contract negotiation. The two parties work together towards creating a valuable product. However, the ideal is just the ideal, and customers are still looking for fixed-scope, fixed-price contracts, which have a fundamental contradiction with Adaptive methods, which is a source of never-ending contract negotiations similar to those in Predictive projects.

Value 4: Responding to change over following a plan

This statement, similar to the second statement, is specific to Adaptive systems. Instead of having a Predictive, upfront plan that can show us the way, we are dependent on adaptation. The latter is usually referred to as "change" in Agile, probably because it makes customers happy to know they are free to change everything, but in fact, it's not a change unless it doesn't match the initial baselined plan, which we don't have in Adaptive systems. Technically, what we have is a continuous stream of new ideas. However, let's keep calling them changes, just for the sake of all customers out there.

Chances are high that you will have questions about the Agile Manifesto in your exam. It's not a bad idea to review it multiple times and to even be able to remember the four statements.

Agile Principles

The Agile Manifesto is pleasantly short. However, the authors thought it might be a good idea to elaborate on the newly named Agile idea, so they created the following twelve principles:

Principle 1: *Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.*

After all, we're doing business, and we need to have happy customers. That's obvious. Well, nowadays, we prefer to say that the end-users' satisfaction is the ultimate measure because that generates profit for the customer and, sooner or later, will satisfy the customer in a sustainable way. Too idealistic?

So, how do we satisfy them? That's by the software we create, which has the potential to generate value (e.g. money). When we deliver early and continuously, we will generate the value sooner, and we also have the opportunity to adapt and create something that the market really wants and will pay for, rather than something that we expect them to want.

Principle 2: *Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.*

Let's do a little more marketing around the word "change" that customers love ;)

Principle 3: *Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.*

Do you remember the iterations that we talked about – those periods of time in which we iterate (repeat the development processes) to create an increment of the product? This principle says they shouldn't be longer than a couple of months. In Scrum, the maximum is one month. We have to talk about it a lot until the end of this book.

Do you also see the suggestion about a couple of weeks? Many people used to laugh at it in those days – the idea of having a new increment in only a couple of weeks. However, now we have projects with even shorter iterations.

Principle 4: *Business people and developers must work together daily throughout the project.*

This goes against the idea of separating the business people (customer or otherwise) from the technical people, which is still a problem in projects. They sometimes see each other as enemies, which is not the best thing that can happen in a project.

In addition to that, we can't adapt if the business people are not available all the time. Think about continuous analysis of new features and testing of the completed units. Besides, it's always more fun to have more people when celebrating the completion of another iteration!

Principle 5: *Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.*

Soon we'll discuss more aspects of Adaptive systems. One of them is that we need to have empowered people in the project level; not just because it's a good thing, but mainly because Adaptive lifecycles need it. Maybe you can think about the reasons for this, until we discuss it in the next sections.

Principle 6: *The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.*

Instead of emails! Note that this is the most exam-friendly principle of all time.

I'll get back to this in a separate chapter when we talk about *osmotic communication*.

Principle 7: *Working software is the primary measure of progress.*

Most projects measure the wrong things. It's a fundamental problem because what you measure is what you get. If you measure how many lines of code are produced, you will just get more lines of code. If you measure how busy the developers are, you will get busier developers. If you measure velocity (a common Agile measure about the speed of development that we will discuss later), you will get higher velocity (which is not the goal).

So, what should be measured?

The main measure is the amount of value generated. Because that's difficult to measure, the next best thing is the working software that creates the capacity for generating that value.

Principle 8: *Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.*

No over-over-time work before releases. It's about maximizing value in the long term. It's not about short-term gains that may lead to a lower production rate and lower quality.

Principle 9: Continuous attention to technical excellence and good design enhances agility.

There's a risk of having poor design in Adaptive systems because design is done gradually instead of upfront. There are certain practices for overcoming this problem.

Principle 10: Simplicity – the art of maximizing the amount of work not done – is essential.

It's a very complicated way of saying something simple: that having more features is not always a good thing.

It's a good idea to keep the solution simple and have only the really useful features in it because it saves time and money (which can be used for other projects) and reduces the maintenance cost.

Principle 11: The best architectures, requirements, and designs emerge from self-organizing teams.

Self-organization means having empowered people in the project who are involved in the decisions, and it's usually a good idea.

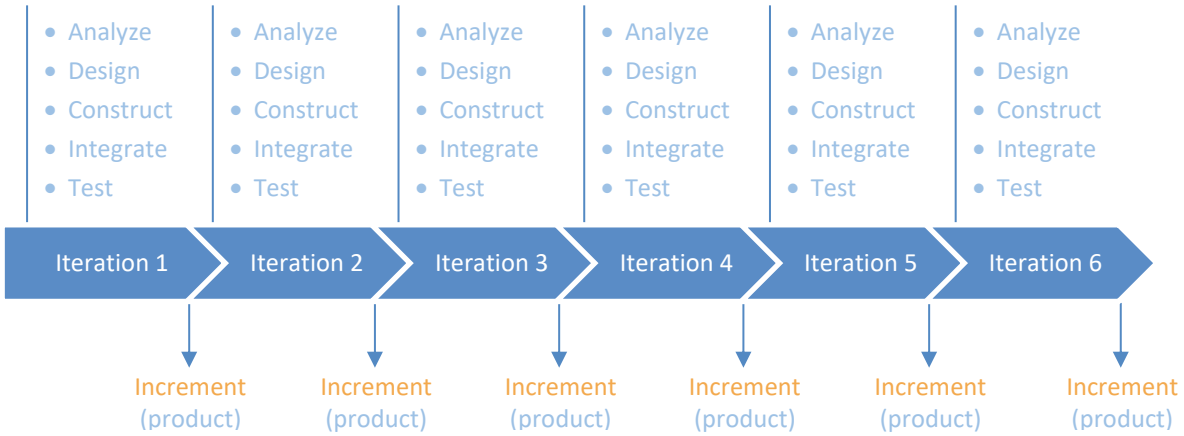
Principle 12: At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

You need to accept that the way you work is not perfect, and you can always improve it in small steps. Also, don't look at the way the Agile Manifesto and Agile Principles are improved; do as they say, not as they do ;)

OK, we're done with reviewing the Agile Principles. Don't forget that these are very common topics for your exam.

Practical Considerations about Adaptive Lifecycles

Do you remember how Adaptive lifecycles work? Here's the picture again, just to make sure the book will be thick enough:



There are a few things we need to talk about. First, for each iteration, we choose a number of features, and our goal is to create a piece of working software (increment) by the end of the iteration that, hopefully, contains all the features. Now, in your opinion, should it be a fixed-scope iteration or a fixed-duration one?

By the way, I'm using the word "feature" very loosely here.

Fixed-Scope vs. Fixed-Duration Iterations

Theoretically, both of them can work, but in practice, fixed-duration iterations are significantly superior because if you keep the scope of the iteration fixed, then:

- You will usually need more time to finish the scope, which reduces the amount of feedback and therefore opportunities for adaptation.
- You may spend too much time on each feature and add too many bells and whistles. Having a fixed duration continuously pushes you to focus on the most valuable things first.

So, that's why almost all Agile methods have fixed-duration iterations, and they usually insist on respecting these *timeboxes*. A timebox is a period of time with a maximum (or fixed) amount of time, and we don't extend it under any circumstances (if you do it once, you will do it all the time).

Duration of Iterations

So, now that we've established that iterations are timeboxed, how long should they be? It was mentioned in the Agile Principles; do you remember?

The maximum is two months based on the Agile Principles. In Scrum, the maximum is one month.

This time is enough to develop a few features and show them to the customer and end-users to generate feedback. We don't want it to be too long because we won't have enough feedback in that case.

Same Duration or Different Durations for Iterations?

In your opinion, is it better to have the same duration for all iterations or keep them flexible?

Having the same duration is more disciplined, and it's usually unnecessary to decide about a new duration all the time.

Scrum requires you to have the same length. This is, however, subject to adaptation. What it means is that you can start your project with two-week Sprints (yes, Sprint is the Scrum term for iteration) and, after a while, when you realize that's not long enough to create enough features, you can decide to make them three weeks long. This is fine. What's not fine in Scrum is to get together before each Sprint and say "OK, how long shall we Sprint this time?"

Not all methods are the same in this. In DSDM, which is another Agile method, you plan the duration of the timeboxes (yes, DSDM calls iterations timeboxes) when you are planning their scope.

What If Some Features Are Not Done?

So, we pick a number of features for the iteration that is timeboxed. What happens if we can't finish everything by the end of the iteration?

It's absolutely fine because our main goal is to create an increment of software that can generate feedback for our adaptation and, later, generate the most value when it's put into production. Our goal is NOT to develop as many features as possible.

Three of the Agile Principles support this claim; can you say which ones?

Answer: 1, 7, 10