Agile Scrum Handbook

3rd Edition

Nader K. Rad

Van Haren PUBLISHING

Other publications by Van Haren Publishing

Van Haren Publishing (VHP) specializes in titles on Best Practices, methods and standards within four domains:

- IT and IT Management
- Architecture (Enterprise and IT)
- Business Management and
- Project Management

Van Haren Publishing is also publishing on behalf of leading organizations and companies: ASLBiSL Foundation, BRMI, CA, Centre Henri Tudor, CATS CM, Gaming Works, IACCM, IAOP, IFDC, Innovation Value Institute, IPMA-NL, ITSqc, NAF, KNVI, PMI-NL, PON, The Open Group, The SOX Institute.

Topics are (per domain):

IT and IT Management

ABC of ICT
ASL®
CMMI®
COBIT®
e-CF
ISO/IEC 20000
ISO/IEC 27001/27002
ISPL
IT4IT®
IT-CMF™
IT Service CMM
ITIL®
MOF
MSF
SABSA
SAF
SIAM™
TRIM
VeriSM™

Enterprise Architecture

ArchiMate®
GEA®
Novius Architectuur Methode
TOGAF®

Project Management

A4-Projectmanagement
DSDM/Atern
ICB / NCB
ISO 21500
MINCE®
M_o_R®
MSP®
P3O®
PMBOK® Guide
Praxis®
PRINCE2®

Business Management

BABOK® Guide
BiSL® and BiSL® Next
BRMBOK™
BTF
CATS CM®
DID®
EFQM
eSCM
IACCM
ISA-95
ISO 9000/9001
OPBOK
SixSigma
SOX
SqEME®

For the latest information on VHP publications, visit our website: www.vanharen.net.

Agile Scrum Handbook

3rd edition

Nader K. Rad

Van Haren PUBLISHING

Colophon

Title:	Agile Scrum Handbook – 3rd edition
Author:	Nader K. Rad
Text editor:	Stephen Brightman
Publisher:	Van Haren Publishing, 's-Hertogenbosch-NL, www.vanharen.net
DTP:	Coco Bookmedia, Amersfoort
ISBN Hard copy:	978 94 018 0759 3
ISBN eBook (pdf):	978 94 018 0760 9
ISBN ePUB:	978 94 018 0761 6
Edition:	Third edition, first impression, April 2021

For further information on Van Haren Publishing, e-mail to: info@vanharen.net.

Contents

1. The Agility Concept

There are many myths and misleading concepts about Agile, starting with the answer to the most basic question in this context: **What is Agile?**

What you may often hear is ambiguous statements such as “Agile is a mindset”. Agile, like almost everything else, **needs** a particular mindset, but it’s not correct to say that Agile **is** a mindset. Saying that “Agile is a mindset” has only one practical consequence: It lets certain people do whatever they want and just call it Agile because it’s fashionable these days.

Another common problem in our community is the illusion of the external enemy. Those of you familiar with the way authoritarian systems work know that they always need to have an enemy. It helps cover the gaps they have in their system by creating distractions, and creates a common goal to cover the lack of real, achievable internal goals. It’s sad to see that many Agile practitioners have the same approach, usually for the personal gain of a few *leaders*.

It’s best for your professional life to be open to different ideas and learn from all of them without you becoming a cult member. This approach is the first principle in the Nearly Universal Principles of Projects: https://nupp.guide

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

1.1 The Development Approaches

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

- Analyze
- Design
- Construct
- Integrate
- Test

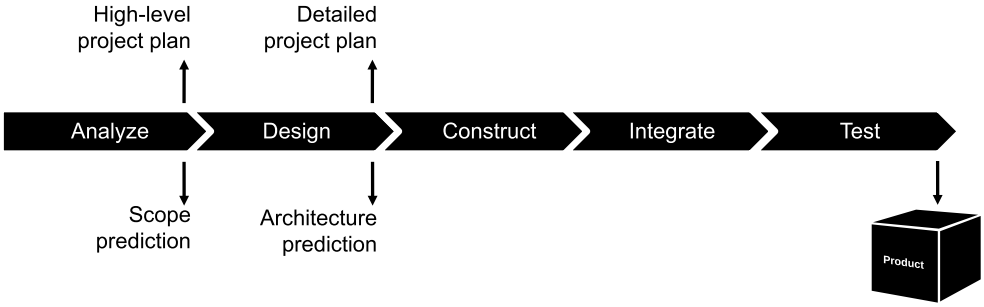
You can, of course, use other names for these steps, merge them into fewer steps, or split them into more – that's all fine. These steps can be called **delivery processes**, which are different from management processes such as planning and monitoring.

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

1.1.1 The predictive approach

You probably have a few options in mind, and they all belong to one of the two generic forms, which we will discuss next. Each of these options can be called a **development lifecycle** or a **development approach**.

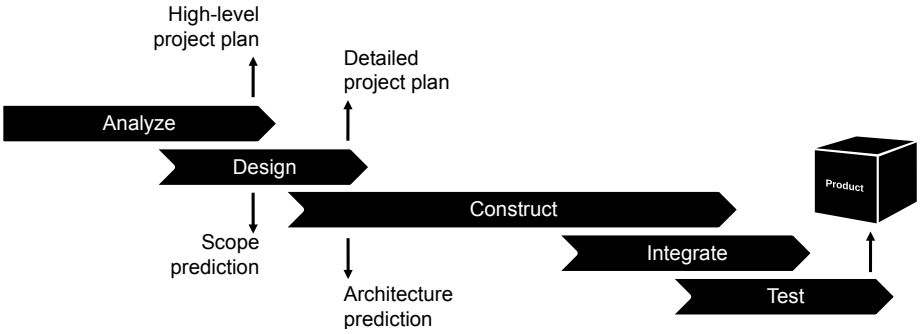
The next figure shows one generic development lifecycle.



In this lifecycle, each process is completed before we proceed to the next one:

1. First, we completely analyze the requirements and decide what we want to have in the solution.
2. We then design the architecture of the whole solution and find out the best way to form the features.
3. Programmers then start building the units.
4. The units are then integrated into one solution.
5. Finally, the solution is fully tested and errors are fixed.

Obviously, the steps can overlap; e.g., you don't need to wait until all units are complete before integrating and testing them. As a result, the same lifecycle would look like the following figure with overlaps:



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

This type of lifecycle is based on an initial investigation to understand what we need to produce. We have an upfront specification, an upfront design, and consequently, an upfront plan. That's why some people call it **plan-driven development**. Furthermore, we try to predict what we need and how it can be produced, and that's why a common name for it is **predictive development**.

Predictive lifecycles are the normal and appropriate way to develop many types of projects, such as construction. You plan and design first, and then follow those optimized, well-formed plans and designs. However, this is not a comfortable way of working in some projects, such as typical IT development projects. You can spend a lot of time specifying and analyzing the requirements, and then base everything else on that. What happens next, though? 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 commonly remarked in the IT industry, the customer doesn't know what they want until they see the product. But when do they see the product in a predictive lifecycle? Towards the end of the project – at which point, the cost of change is at its maximum.

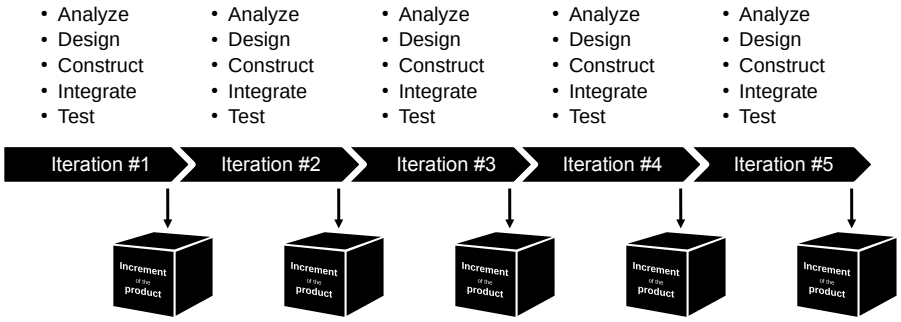
The Agile community usually refers to predictive systems as **waterfall** systems. However, it's not a good idea to use this term because it has developed a negative connotation, and its use would bias an otherwise rational conversation about the development approaches.

1.1.2 The adaptive approach

To overcome the problems that IT development projects have with predictive lifecycles, we can sacrifice the comfort and structure of a predictive system and use a different lifecycle that creates the product incrementally, to check it with the customers and end-users along the way. This is a luxury we have in IT development projects that not everyone else has. Think about a construction project: There are no meaningful increments for it, and the product is not usable until the end.

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

So, based on the fact that we can have incremental delivery in IT development projects, let's exploit this opportunity with a lifecycle like the on in the next figure.



There's no real prediction in this lifecycle, as instead of predicting the product and relying on that prediction, we have short iterations in which we create increments of the product. Each iteration is focused on a few features that seem promising. We build each one, show the increment to the customer and end-users, receive their feedback, and decide what to do in the next iteration. So, instead of predicting, we carry on with the project and **adapt** to the feedback. This approach uses an **adaptive lifecycle**. "Agile" is the popular name for adaptive systems.

To create each increment, we need to iterate through all the development processes during each time window, and that's why we call those windows **iterations**, and this way of development **iterative development**. In iterative development, each process (such as design) is repeated multiple times for different elements in the product, instead of being run once for the whole product.

Normally, iterative development and incremental delivery occur together.

1.1.2.1 Fixed-scope vs. fixed-duration iterations

In your opinion, is it better to have fixed-scope iterations or fixed-duration ones?

Theoretically, both of them can work, but in practice, fixed-duration iterations are superior, because keeping the scope of the iteration fixed, can have the following results:

- 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.
- The time you need to complete the scope is usually longer than you expect, which makes the iterations longer and reduces the number of feedback loops. When there's less feedback, there will be less adaptation.

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

1.1.2.2 Duration of iterations

Now that the iterations are supposed to be timeboxed, how long should that be for?

We can receive feedback at any time, but the structured feedback we receive at the end of each iteration is key. Therefore, shorter iterations give us more structured feedback, and therefore, more opportunities for adaptation. On the other hand, each iteration needs to have enough time to produce a number of features worthy of a serious review with the customer, which means that they can't be too short.

In the early days of the Agile systems, 4 to 8 weeks seems like a good idea. Nowadays, shorter durations are more acceptable. The maximum acceptable duration is 4 weeks in most systems, and durations as short as 1 week seem practical for the current technologies.

1.1.2.3 Same duration or different durations

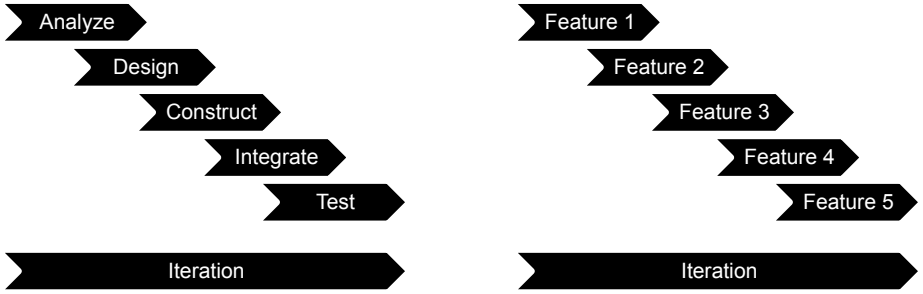
In your opinion, is it better to have the same duration for all iterations, or to keep them flexible?

Having the same duration is more disciplined and instills regularity. In most cases, there's no real need to decide about the duration of each timebox separately, which is why most systems set the same duration for all iterations. You can revise this timeboxed duration, but you won't decide about the duration of each iteration separately.

1.1.2.4 What happens inside iterations?

An iteration is a period of time in which we repeat the development processes. How do you do that, though?

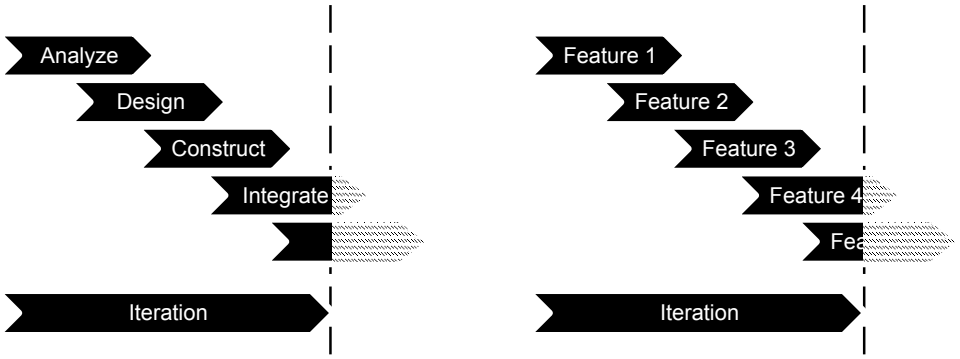
Here are the two possibilities:



The one on the left goes through the development processes and runs each of them for all the features that belong to the iteration. Maybe we can call it mini-predictive.

The one on the right goes through the features, one or a few at a time, and runs all the development processes for each of them. We can consider it a mini-mini-predictive system (i.e., almost not predictive).

We prefer the second, feature-based option, mainly because it's the one that's compatible with timeboxed iterations.



When there's a maximum duration, we may not be done with everything at the end of the iteration, which means that with the feature-based approach there are a few features we won't be done with, while with the other approach, we won't be done with one or more of the development processes of each of the features, which means that we won't have any usable output at the end of the iteration and we won't be able to demonstrate it and receive feedback.

1.1.2.5 Increment vs. deliverable

Each increment is a deliverable, but not every deliverable is an increment.

We use the term “increment” to refer to the increments of the product, which are, in the case of IT development, different versions of working software. Each new increment is a usable version of the same product but with more features, and it has to be usable to enable reliable feedback.

In contrast, a deliverable can be almost anything you produce in your project. For example, in a predictive project, the upfront design and upfront plan are deliverables that can’t be considered increments of the product.

Since being Agile is fashionable, some people just call their deliverables increments and claim to be Agile based on that.

1.1.2.6 Iterations vs. cycles

Every iteration is a cycle, but not every cycle is an iteration.

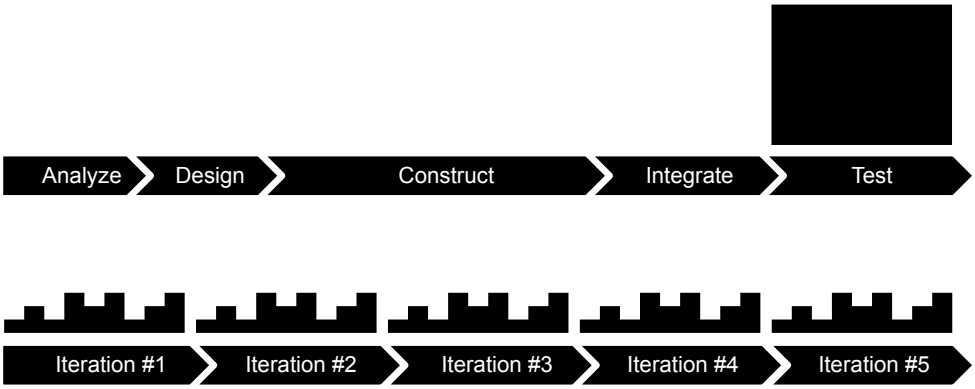
An iteration is a special type of cycle wherein we repeat our **development processes** as well as our **management processes**. Many systems have cycles, but those cycles only repeat the management processes and not necessarily the development processes. The big, monthly cycle and the small, weekly cycle in P3.express, the stages in PRINCE2®, and the phases in the PMBOK® Guide are all examples of that.

To make this difference clearer, imagine a cycle that has its own planning, monitoring and controlling, and closing. The fact that these managerial processes are repeated is the reason their containers are called cycles. Now, imagine it’s a predictive project, and one cycle is about specifying the requirements, the next cycle is about designing the product, and so on. This is a cyclic system without any iterations.

Unfortunately, some people think that as long as they can identify cycles in their projects, they can call them iterative and hence Agile, which is not correct. Even worse than mistaking managerial processes for development processes, some people just call arbitrary time periods in their projects iterations, and consider, for example, weekly “iterations” when there’s no real iteration of any processes in them.

1.1.2.7 Testing and quality in agile

The following diagram shows an over-simplified schema of the way testing is done in each approach:



Most of the testing activities are at the end of a predictive project, which is when we're probably late and under a lot of pressure to finish the project as soon as possible. This pressure may result in dropping some of the tests and compromising on quality.

How about adaptive systems, then?

Well, this problem doesn't exist in adaptive lifecycles because testing is done continuously, and so it doesn't matter when we stop the project, as we will always have the right ratio of testing.

There are other differences also. For example, the nature of adaptive systems makes it almost essential to have automated tests. Automated tests may not cover every single line of code, and there's an optimum **test code coverage** that we need to have in our project. Test code coverage is the ratio of the lines of code tested by automated tests to the total number of lines.

1.2 Selecting a Development Approach

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

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

1. **Does it need to be adaptive?** If you don't need to be adaptive, a predictive lifecycle is more straightforward, more structured, and more *predictive*. 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 it be adaptive?** This question is even more important than the previous one. To be adaptive, you must have the possibility of developing iteratively and delivering incrementally in order to receive feedback and adapt. Let's think of a construction project once again: Can you design the building iteratively? For example, can you design the foundation of the building without designing the rest of it, which is needed to determine the amount of load on the foundation? The answer is simply no. It's not possible to have iterative development (with the meaning we have for it in this context) for a construction project. Furthermore, incremental delivery is not possible in most situations because, on the one hand, the subsets of a building are not usable, and on the other hand, the feedback generated by one subset may not be applicable to the rest. So, we can't use an adaptive lifecycle to build a building (although, don't confuse this with interior design and decoration, or even renovation, for which we may be able to use an adaptive system).

The main message is that the decision between a predictive and an adaptive approach is not simply a matter of good and evil, but rather it depends on several factors. They are both valid approaches, and each of them is more suited to some types of product.

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

1.3 Is Agile Only Suitable for IT Development?

Most of the examples in this book, as well as other resources about Agile, are about IT development projects. Does that mean that Agile is limited to IT development projects?

1.3.1 Projects

There are some people who claim that Agile can be used for every type of project, and the same people usually claim that it's the only correct way of doing projects. They are usually people who have not experienced any serious project other than non-critical IT development ones. In reality, there are many types of project where an adaptive method is either not needed or not possible because we can't develop them iteratively and deliver them incrementally.

Aside from the simple fact that Agile is not the one absolute truth and cannot be used in every project, we can still consider the range of projects that can benefit from an adaptive system. Is it limited to IT development, or are there other suitable types?

It may be possible to use Agile in some other types of project, but it requires a professional, structured effort, which doesn't seem to have been done yet. There are some non-IT projects that claim to be Agile, but they usually mistake the meaning of Agility and are victims of the **Cargo Cult** effect. Notwithstanding this, IT development will probably remain the best type of project for adaptive methods.

1.3.2 Programs

Everything said so far has been about **projects**, but things are different when it comes to **programs**. According to MSP®, which is a program management method from the same family as PRINCE2® and ITIL®, projects may be either adaptive or predictive, but programs always have to be adaptive. This is so because projects are about products, while programs are about results. We can predict how to build products, but we can't predict how to achieve results.

1.3.3 Operations

Project management methods always start by defining what a project is, because they are only applicable to projects and not to programs, portfolios, or business as usual (operations). This has never become a tradition in Agile systems – they don't insist on being used in projects, and some people have been using them in operations. This has its roots in IT development, where there's no clear line between projects (major changes) and operations, where minor changes are applied to the product. The extreme version of this notion is visible in DevOps, where the project side (development) and business as usual side (operations) are merged into one.

1.4 Is Agile Faster?

The word "agile" implies that these methods are faster. While it is very difficult to confirm or reject this hypothesis, there's one concept that really helps in Agile projects, and it's not about the speed with which we develop, but about the set of features we need to develop (the scope).

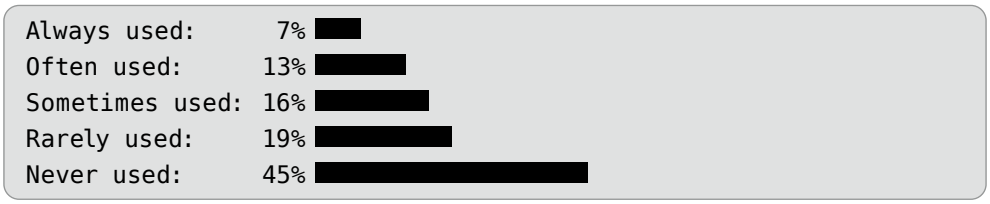
Think of an IT project that is supposed to be developed using a predictive method. One or a few customer representatives would be responsible for identifying and communicating the requirements. They know that if they miss a requirement, it will be expensive and troubling to add them in the future, and therefore, they do their best to identify all requirements. As it turns out, they become too creative in this area and add requirements that add insufficient value. These extra features require more time and resources, and also make the product more complicated, which is a serious problem for future maintenance and expansions.

In an adaptive system, on the other hand, the customer representatives are not forced to come up with all the requirements upfront, and the chances are therefore lower

that strange requirements will be added to the list. Even when there are such requirements, a proper adaptive development system at least helps the representatives understand their value so they can leave them for last, or even remove them.

In practice, an Agile project that is run properly has the chance of having a smaller scope, which makes the project faster and less complicated.

As an example, in 2002, Standish Group reported the following rate of use for the features of four of their internal applications:



Imagine how much faster their projects could have been, and how much simpler their products could have been, if those never-used and rarely-used features had not been included. This is, of course, only one example of a few applications in one organization, but the overall trend may not be so different.

1.5 Is Agile New?

Agile is usually advertised as the new approach. The use of the term “Agile” to refer to adaptive lifecycles is certainly new, but what about the lifecycle itself?

It’s difficult to imagine a long history of human beings with many projects and programs that have been done without any form of adaptive lifecycles. Think of a very popular type of initiative (project or program) in the olden days: going to war. Could you manage to wage a war using a predictive approach? Did they plan and design everything at the beginning? Certainly not. You may have a high-level plan (which is more like a strategy than a plan) and manage the war one battle (iteration) at a time, and based on the outcome of each battle, adapt for the rest of the initiative. It’s not a pleasant example, but a clear one that shows that adaptive lifecycles aren’t all that new.

So, what is it that is new? It’s mainly the use of adaptive systems in IT development and the name “Agile” that are new. In the old days, IT development projects were very different and required a precise, predictive method. Later on, as computers evolved, the nature of those projects and their audiences changed. In most cases, predictive systems weren’t a great choice anymore, but practitioners continued using them. That was the case until a group of people involved in those projects started reinventing the adaptive method.