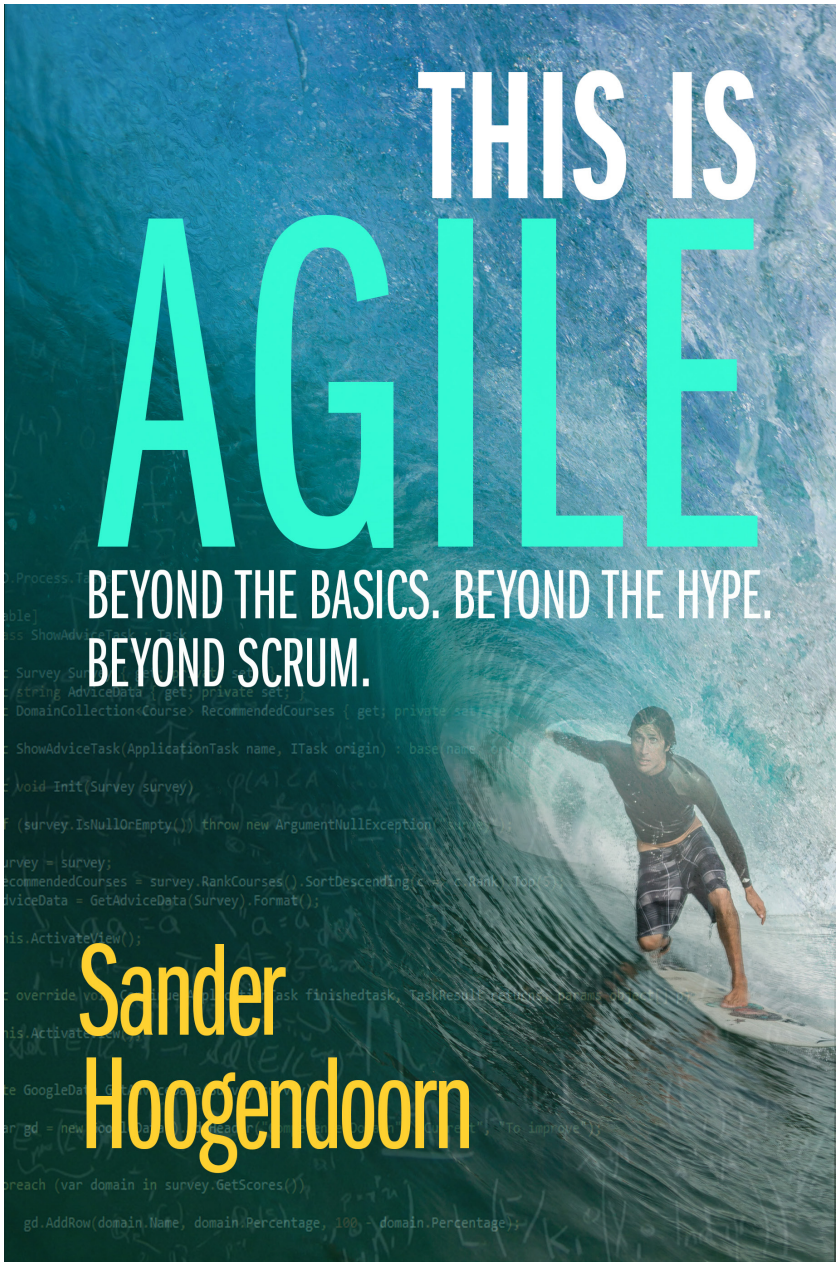

THIS IS
AGILE
BEYOND THE BASICS. BEYOND THE HYPE.
BEYOND SCRUM.
Sander
Hoogendoorn

This is Agile

Sander Hoogendoorn

Beyond the basics. Beyond the Hype.
Beyond Scrum.

DYMAXICON

Escerpted from *This is Agile*
ISBN 978-1-937965-21-1

Published by Dymaxicon, Sausalito, CA

This title originally appeared in Dutch as
Dit is Agile © 2012 by Sander Hoogendoorn
published in the Netherlands by Pearson Benelux B.V.
English translation courtesy of Capgemini S.A.

Contents

This is agile

@Quote_Soup: The main thing is to keep the main thing the main thing—Stephen Covey.

More and more organizations are realizing that their software development process is not delivering the desired result. Projects are seldom delivered on time. Too often they overrun their budget. The quality of the software often leaves a lot to be desired. Often only part of the requirements are delivered. The added value is lower than expected. Usability is nowhere near where it should be. So things need to change. And that is why agile is "hot."

Agile is an umbrella term for a new way of running projects, with entirely different dynamics, different phasing and different forms of collaboration. But many still see agile as the personal hobby of developers. A playground for techies.

Nothing could be further from the truth. Agile methods, techniques and best practices provide starting points for running projects more manageably, efficiently, cheaply and better than traditional approaches. Agility offers important benefits for everyone involved in a project. For managers and project managers. For analysts, designers, developers, testers and even administrators. And last but not least, for the customer.

Implementing agile is more difficult than it seems. Increasingly, I see partial or failed implementations. Agility requires change, but organizations and people are difficult to change. Sometimes agile implementation becomes the goal itself, instead of a way to reach a certain business goal. Moreover,

every project is unique. There is no one-size-fits-all agile.

This is exactly why I wrote this book. I will explain what agile is and what it is not, how and why agile works, and what the added value is of popular agile approaches like Scrum, Smart and Kanban. And of course we will look into teams, roles, requirements, starting a project, estimating and measuring, planning, best practices and pitfalls. And, inevitably, waterfall.

This book is about the choices you make every day in your agile projects. It gives you options, tips and advice based on my fifteen years of experience in agile development and working on short iterative projects in existing organizations. With a lot of examples and anecdotes from my own practice. With trial and error and learning every day.

This is agile. Beyond the basics. Beyond the hype. Beyond Scrum.

Why waterfall does not work

@TheTweetOfGod: To understand the universe you need to see it in the broader context.

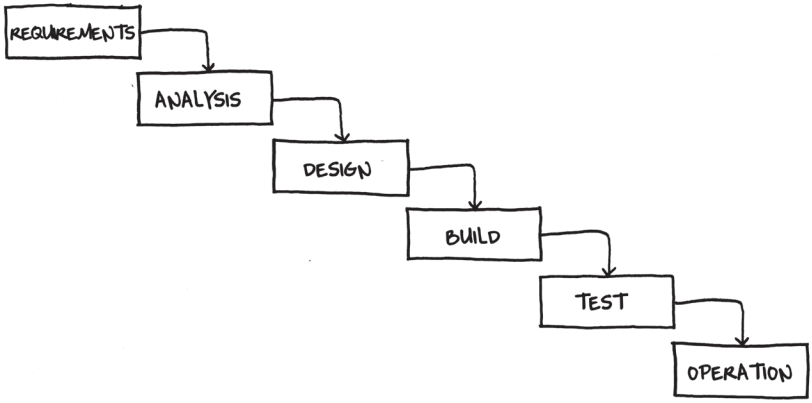
In the 1970's the American, Winston Royce, director of Lockheed's Software Technology Center, was the first to describe the model that everyone now knows as waterfall in his whitepaper, *Managing the Development of Large Software Systems*. Although this paper is seen as the mother of all waterfall methodologies, Royce is not a supporter of this model. To the contrary: Royce used the waterfall model as an example of a process that is not working properly.

The characteristics of waterfall

The waterfall model is a sequential process in which all activities of building software are carried out one after the other.

Royce sees these activities as the minimum necessary to successfully implement and maintain software. Even today, many organizations are hierarchically organized according to this model. There is a department for requirements or business analysis. A department for information analysis. For functional and technical design. There are developers. There are testers. There is a department with functional and technical maintenance staff. And of course there are the inevitable project managers whose task consists of keeping the other roles in check.

Even though Royce does not use the term, the "waterfall" model is derived directly from this picture from 1970. The waterfall model is characterized as follows:



- **Phases.** Each project consists of the phases as depicted in this illustration.
- **Once**. Each of these phases is performed once, and in a fixed sequence.
- **Roles**. Each of these phases has its own specific roles. Analysts do analysis, developers develop, and testers test.
- **Deliverables**. At the end of each phase, a final product is completed. It is signed, sealed and delivered. These are the so called "milestones." For example, the information analysis or the functional design.
- **Fully**. The next phase only starts when the product from the previous phase is fully completed. The development of the software does not start until the design is completed. Only once the software is fully developed will testing start.

- **Fixed**. Projects are usually performed with a fixed budget, date and scope.

Makes sense right?

The problems with waterfall

Ever since 1970, the majority of all projects have been carried out according to (variations on) the waterfall model. A significant percentage of these projects are not successful. Almost everyone in our field can remember working on projects that went far over budget, did not meet the agreed deadlines, implemented only part of the agreed requirements or were stopped prematurely, possibly after several restarts.

Looking at the characteristics of these traditional projects, one sees very clear reasons for this. Let me list the most common ones:

- **Knowledge disappears**. Each phase is carried out by a specific role. Once the final product of a phase is completed, the role is removed from the project. A lot of knowledge is lost after each phase of the project in this way. After all, no matter how much someone documents, it is impossible to document *everything* (and the kitchen sink).
- **Progressive insight**. During each phase of a project, the team learns more about the scope of the project, the needs of users and the (im)possibilities of the chosen technology. This is called progressive insight. However, since the end products of earlier phases are fixed, this insight can no longer be taken into account.
- **Changing requirements.** On average, 20 to 25 percent of the requirements for a project change while it is underway. New requirements are found, existing requirements change or become obsolete. But since all requirements were already detailed at an earlier phase, a

portion of this work becomes redundant or needs to be performed again. This is inefficient and expensive.

A project manager once told me: "Waterfall works fine. As long as you don't accept changes."

- **Complete and thorough.** Since the end product of an earlier phase cannot change, it is important to complete each phase 100 percent, especially during analysis and design. After all, if during the next phase new requirements are discovered, they will be treated as imperfections of previous phases.

  A designer: "If we had more time available to analyse, we would not have encountered these problems now."

  Due to this, analysts and designers in particular find it difficult to deliver their final end products, so early phases of the project are often seriously delayed.
- **Documenting too much.** In an attempt to be complete and thorough, sometimes so much is documented that it is impossible to ever build the software, let alone test it.

  In a project for a bank in Belgium, two years were spent on the functional analysis. Over 2,500 pages of text were produced. The intended developers told me they did not dare to estimate the scope of work, let alone get started on development. When I asked why, they said: "Well, we don't know whether we've yet fully captured all the functionality."
- **Difficulty estimating.** Each phase contains a different type of work and thus has its own pace. As a result, it is difficult to give good estimates. If the analysis takes six months, what does this tell you about the total duration of a project?

- **Late risks.** In waterfall, the software is actually developed in a late phase in the project. Sometimes several years have elapsed before the first code is written. This brings great risks with it. The proposed technology could be outdated. Testing of the software often only starts after the code is fully completed. No matter how great the analysts, designers and developers are, testers will always find imperfections.

  With the passing of a project's phases, more and more work has been performed. The cost for resolving errors in a project grows exponentially. This is known as the Law of Boehm.

  A good example is a project for a large international bank, developing new software for handling policy requests. The architect had devised a beautiful architecture whereby validations were fired dynamically the moment the user moved to a next field. The project progressed steadily. After eighteen months the developers handed the software over to the testers.

  The developers had heavy development machines at their disposal, whereas the testers used the same machines as the end users. On the first day of testing, the performance of the software proved to be wretched. The changes to the architecture needed to improve performance ensured that the project delivery date was delayed by six months.

Why does waterfall still exist?

An interesting question is why the waterfall model is still used. First, it is a simple model that is easy to explain. Moreover, it gives the illusion of predictability and measurablility. Not infrequently, project managers envisage a Gantt chart in the model to plan their project. It is also man's nature to be conservative. We hold on to what we have done before—success-

ful or not. “I worked like this for twenty years, why should I change now?” In the end, projects often appear to be successful because the end result is no longer compared to the original objectives and budget. “We’re glad to even deliver.”

In fact, the waterfall model is a copy of similar models in other industries used for *repeated* production. Think of the automotive industry. Software development, however, can hardly be compared with repeated production. Unlike the production of the same one automobile over and over again, software development requires a high degree of creativity.

Decisions are continually made that influence outcome, even during the development and testing of the software. The production metaphor does not hold. A better metaphor from such industries would be that of the development of a new type of car.

These disadvantages of waterfall are sufficient to conclude that the model does not work, and never worked. Abundant research supports this conclusion. Project managers who indicate that their waterfall projects are going well are the proverbial exception to the rule. They usually don’t compare the outcome of the project with the original budget, or have adjusted the method to minimize these disadvantages.

From waterfall to iterative

Remarkably, Winston Royce already came to the conclusion that waterfall does not work, in 1970. In his paper, he uses this model to show how *not* to organize your project. Royce is in favor of a *do-it-twice* approach. The software is analyzed in small increments, designed, developed, tested and delivered. The developers continuously incorporate the feedback from each of the activities to improve the previous milestones. Thus, if the development process shows that the design should be improved, this is done. Immediately.

Royce actually preaches a rather iterative model. This is striking because his paper is seen as the mother of all waterfall methodologies. Even more interesting is that after years of evolution we find the ideas of Royce in the current generation of methodologies, each with a highly iterative character.

This is agile.

What is agile?

@Quote_Soup: Never be afraid to try something new.
Remember, amateurs built the ark.
Professionals built the Titanic.

Almost every book about agile begins with the *Agile Manifesto*. This one too. The *Manifesto* offers an excellent summary of the principles of agile. It was drafted during a weekend in Utah in 2001 by a group of people with a track record of discovering innovative approaches to software development. Many of these approaches pointed in the same direction: away from heavy, process-oriented waterfall development methodologies.

Although the group included significantly different views, the *Manifesto* emphasized the common values of their approaches. Everybody had experienced the creation of new ways of collaboration that worked demonstrably better than the ones in traditional methods.

The *Manifesto* consists of four powerful statements that convey the common values of these new-generation approaches:

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

Each of these statements contains the word *over*. This indicates that the right-hand side of the statements is important, but that the left-hand side is even more important. And not,

as is often wrongly interpreted, that the right-hand side is prohibited. “No. In agile we don’t document.” “No. We don’t write a plan.” Why? The right-hand side of the four statements is also important. Even in agile.

Approaches and methodologies

Of course, it cannot hurt to follow a process when working on a project. But applying any approach or methodology to the letter is often too rigid. A lightweight process is preferable.

PEOPLE AND INTERACTIONS
OVER PROCESSES AND TOOLS

The *Agile Manifesto* puts special emphasis on teams, and how the way people collaborate makes projects successful. Agile approaches offer numerous techniques for collaboration, especially between the different roles on projects.

What is the difference between an approach and a methodology? An approach briefly describes a process or procedure without supplying a concrete definition of all activities, roles and products. Approaches are therefore widely applicable. Think of Scrum, Kanban or Extreme Programming, but also PRINCE2. Scrum advertises itself primarily as a framework. The use of an approach or framework allows a project liberties. Sometimes this can also leave a project team in confusion.

A methodology also describes a process or procedure, but states concrete guidelines and best practices for activities, techniques, roles and products. Methodologies are therefore less widely applicable than approaches, but offer more support. Sometimes a little too much support. Think of SDM or IBM’s Rational Unified Process (RUP).

A characteristic of agile projects is that they do not get locked into a particular approach or methodology, but have a focus on continuous improvement of the way they work. And because each project is different, they also work in different ways. This despite the fundamentalist wind blowing from various agile communities, indicating what exactly is and is not allowed. “Modelling is not allowed in Scrum.” Or: “In Kanban you always use work-in-progress limits.”

This fundamentalism is especially visible in popular agile approaches. There the inflow of newly certified people is so huge it makes for slim pickings. Unfortunately, too often newbies act to the letter of the law, not to the spirit, creating a very dogmatic use of these approaches. Keep in mind that people and their interaction are always more important than processes and tools. Especially in agile!

Working software

Many people in traditional projects mainly produce paper. Indeed, analysts and designers have already left the project before code and tests are written. As an analyst or designer you don’t even see the result of your hard work. In agile this is different.

WORKING SOFTWARE OVER COMPREHENSIVE DOCUMENTATION

In agile, all roles work together. Even the analysts and designers. In agile, the people in these roles are usually considered domain experts, but the customer and users also work together with each other. Not one after the other, but at the same time. That way everyone gets to see the result of their work immediately. It is therefore important that teams find the best way to document their work. Because remember:

Even agile teams document!

Agile teams document just enough. Enough for the architecture to be determined. Enough to test. And also enough for the software to be handed over to the maintenance team after the project. Agile projects try to find the minimal set of documentation to reach their objectives. It is wise to involve the maintenance team in deciding on the objectives, as they are also stakeholders on the project.

Collaboration and contracts

In traditional projects it is customary to fix the scope, deadline and budget of a project up front. This model is called *fixed price*. In reality it is often *fixed price, fixed scope and fixed date*. Fixed price seems necessary to get clarity on the agreement between customer and contractor. Frequently public tenders are based on this concept. This seems plausible. But in projects in which each phase is only executed exactly once, each phase needs to be spot on. Complete and thorough. That's hard.

CUSTOMER COLLABORATION OVER CONTRACT NEGOTIATION

In agile, the scope is not completely fixed in advance. This offers the customer the opportunity to change the requirements during the project, thus creating space for progressive insight. Although this is not easy, agile projects try to create trust between customer and contractor instead of setting it in stone.

Dealing with change

When something unforeseen happens on a traditional fixed-price project, this will always lead to problems. Maybe the analysts are unable to deliver the requirements on time. Or new but crucial requirements are found during development. Or gross errors in requirements or software are uncovered during the acceptance tests. Progressive insight, after all, is inevitable.

The later progressive insight occurs, the greater the efforts needed to fit it into the project. Traditionally, changes are therefore not implemented immediately, but listed in *change requests*. Once the project is completed, it will be decided whether these changes will be realized. This way progressive insight is eliminated. The disadvantage is that projects realize the software that has been agreed upon, but not the software actually needed.

RESPONDING TO CHANGE OVER FOLLOWING A PLAN

In agile, progressive insight is normal. *It's all in a day's work*. Simply because the team gradually learns. About the domain and the requirements. About the techniques and technology used. About collaboration. Agile provides mechanisms that ensure that progressive insights can be incorporated once they occur because progressive insight improves the quality of the product and the process.

A good observer notices that the scope of an agile project is in constant motion. That's right. Moreover, this also holds true for waterfall projects. The scope of projects grows about 20 to 25 percent on average. The difference is that traditional projects try to avoid changes, while agile projects welcome them.

How? Read on!

What makes a project agile?

Apart from the *Agile Manifesto*, and independent of the selected agile approach, the important question remains: what makes a project agile? Which characteristics are displayed in agile projects that distinguish them from traditional projects? In my view, agile projects share the following characteristics:

- **Short iterations.** An agile project is divided into short iterations. During each iteration, a portion of the software is analyzed, designed, developed, tested, accepted and delivered.
- **Collaborating in teams.** People in different roles, like analysts, developers and testers do not work sequentially, one after the other, but with each other as a multidisciplinary teams fully delivering software, each and every iteration.
- **Small unit of work.** During each iteration, a portion of the software is delivered. This therefore requires a small unit of work. Often the units are user stories, sometimes screens. Sometimes smart use cases or features. I prefer to use the term work items.

  During a COBOL-to-web migration, we used transactions as our unit of work.

- **Responding to change.** Agile projects embrace changes. At the start of each iteration, work items are selected from the list of remaining work items based on their priority at that moment in time. During the project, new work items are simply added to this list and in principle could be realized in the next iteration.
- **Ongoing planning and measurement.** Realizing work items in short iterations makes it possible to continuously measure and plan. That way, there is still time

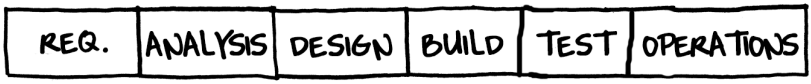
to respond to changes in the scope, method or team. Eisenhower once said: "Plans are nothing, planning is everything."

- **Early and continuous testing.** The sooner a project starts testing, the less effort is needed to correct errors. Because iterations deliver work item by work item, testing can start really early in the project.
- **Early and frequent delivery.** During each iteration, all activities that are necessary for the work items to be delivered are performed. The realized work items are immediately delivered, sometimes even deployed into production. That way all infrastructural hurdles are also cleared at an early stage.
- **Simplified communication.** Although it is not mandatory, agile projects make use of the simplest possible communication tools. Post-its with work items on the wall. A simple spreadsheet with a list of all work items.
- **Co-location.** The concept of co-location expresses that it is best for teams to work in one location. Preferably along with the customer and the users of the software. This makes communication quick and easy and is an asset on any project. Unfortunately, co-location is not always possible. For example, when teams are geographically distributed.

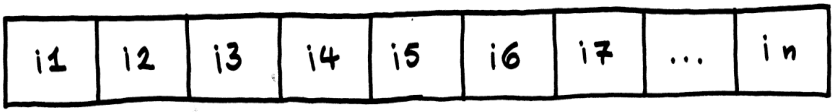
There, that's that. In my view, these are the characteristics that make a project agile. That is not to say that I consider projects that do not meet all these characteristics to be wrong. Or even non-agile. It is immaterial whether a project is truly agile, or only partially so. What is important is that each project is performed in the most optimal way. Sometimes that is very agile, sometimes a little less so.

Agile in a nutshell

A traditional project is divided into phases, during which each phase consists of only one type of work, such as analysis, design, build or testing.

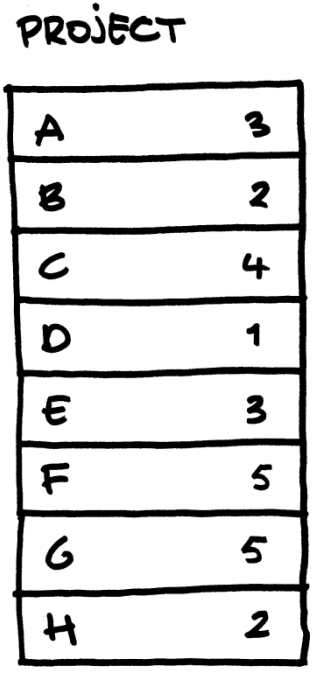


An agile project is divided into small timeboxes. *Iterations*.



Iterations are *timeboxed*. They are never extended. Most agile projects opt for iterations with a fixed length, for example, two or four weeks. Each iteration has the same construction, starting with a kick-off in which the work items for the iteration are chosen. Subsequently, the selected work items are realized. At the end of the iteration, both the realized work items and the method are evaluated. Iterations are the heartbeat of the project.

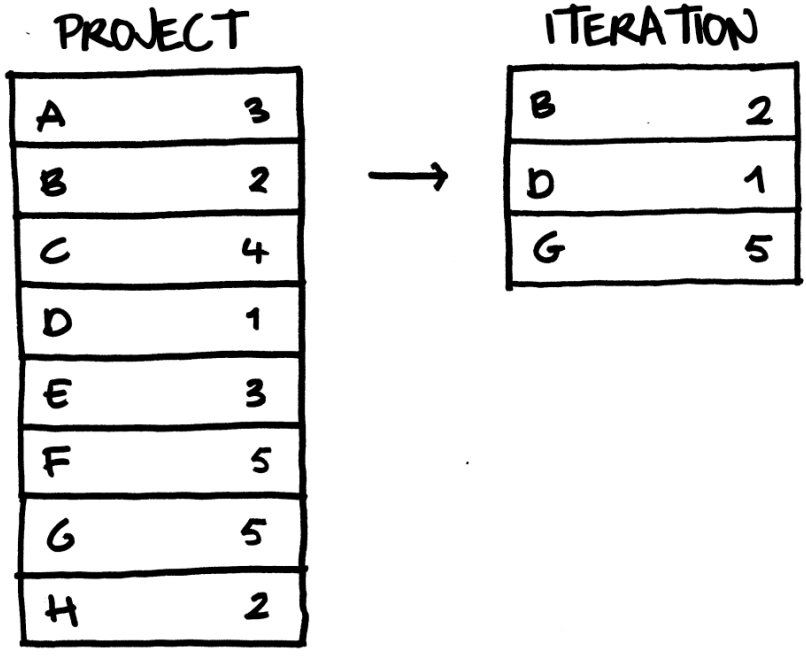
The work items for the starting iteration are selected from the list of all work items yet to be realized. This list is often called the *project* or *product backlog*.



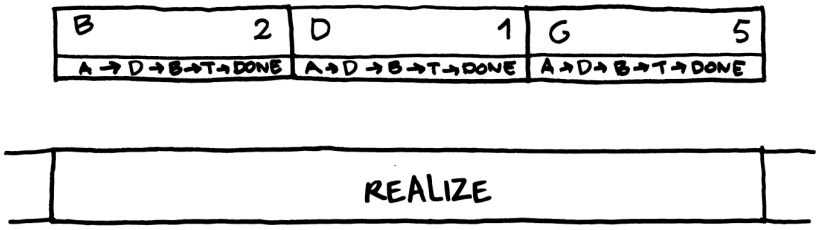
The team advises on selecting work items, but it is always the customer who chooses. For example, on the basis of the needs of users or risk reduction. The number of work items the customer can choose for an iteration depends on the speed of the team. How many work items can the team realize during an iteration? On agile projects, the complexity or size of the individual work items is estimated on a scale of points. Points express the relative complexity of work items. Different approaches use different point scales. The speed of a team is expressed as the number of points that each iteration typically realizes. This is called the *iteration velocity*. The customer selects approximately this number of points for the next iteration. These selected work items are called the *iteration backlog*.

During an iteration, all the work required to get the selected work items realized and accepted is performed. For this pur-

pose, a *definition of done* for the work items is prepared. It contains the conditions a realized work item should satisfy before it is accepted.



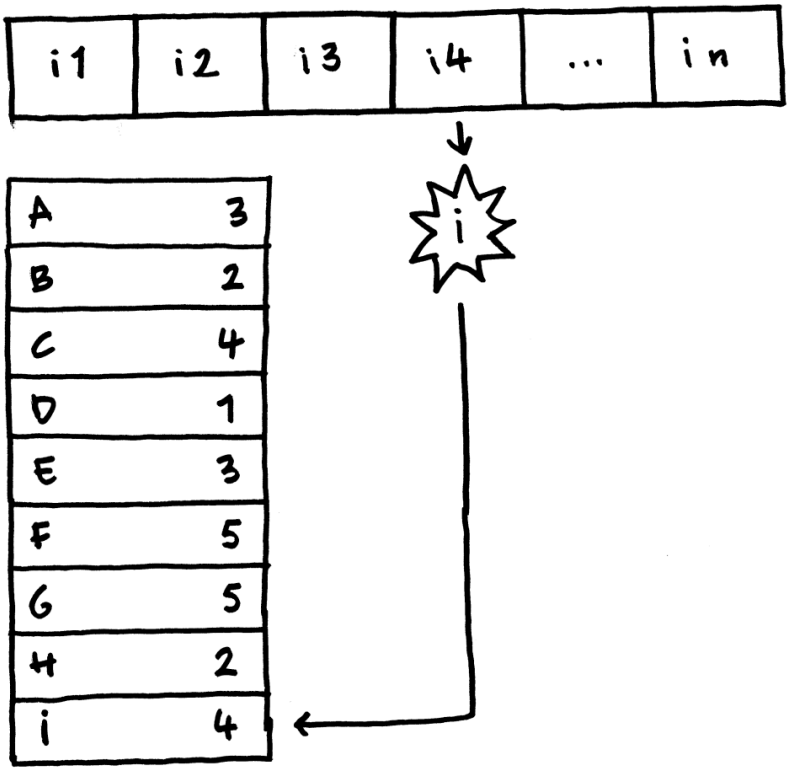
Then all the work that is usually divided into long phases in traditional projects, such as analysis, design, build and test, is performed for each work item. Analysts, designers, developers and testers work closely together in a *multidisciplinary* team. Once the team has realized a work item, and the customer has accepted it on the basis of the definition of done, the team earns the points associated with the work item.



At the end of each iteration, the customer and team evaluate the iteration. At this point, the realized work items are usually looked at again, often in a demo. In addition, the method is scrutinized. What went well? What can be improved upon in the next iterations? Agile projects and teams improve continuously.

Then comes the progressive insight. When requirements change or new requirements are identified, the corresponding work items are modified or new work items are added to the project backlog. Re-prioritizing at the start of each new iteration makes it possible for these work items to be realized during the next iteration.

Naturally, the potential scope of a project grows, which heightens the risk of the project not being delivered on time. The key question is how agile projects guard their scope. Will the project delivery date be delayed as the scope increases? Or will the team still meet the deadline?



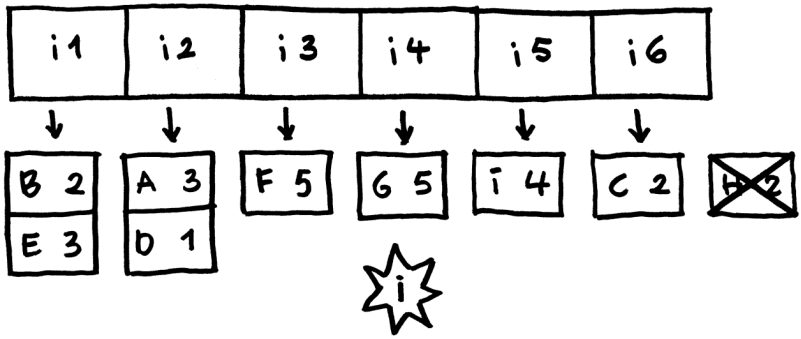
First, due to the fact that a number of work items will be fully realized during each iteration, scope creep is discovered early, making it possible to implement corrective measures at an earlier stage. For instance, deciding to modify or extend the team. If this is done early on in a project, it is effective.

Once we added a middle-ware developer and tester after the second of nine iterations. In another project we replaced a developer with a tester after iteration sixteen of sixty. In both projects, productivity increased.

When a project has no fixed deadline, new iterations may be added. It may be agreed that the project will stop as soon as

the realization of the next work items delivers less value than the cost of realization, when the customer stops the project and implements the realized work items in production. The customer and contractor can agree on a *bonus-malus* system, for example, by sharing the remaining budget.

If the project has a fixed deadline and the scope increases, or the velocity of the team is unexpectedly lower than hoped, the constant re-prioritizing of the work items offers a solution. Indeed, the work items that are still on the backlog after the end date are, by definition, the least important. For the customer it is usually more important to meet the deadline than to get all the work items delivered. This way projects are delivered on time and within budget, but without the last—and least important—work items.

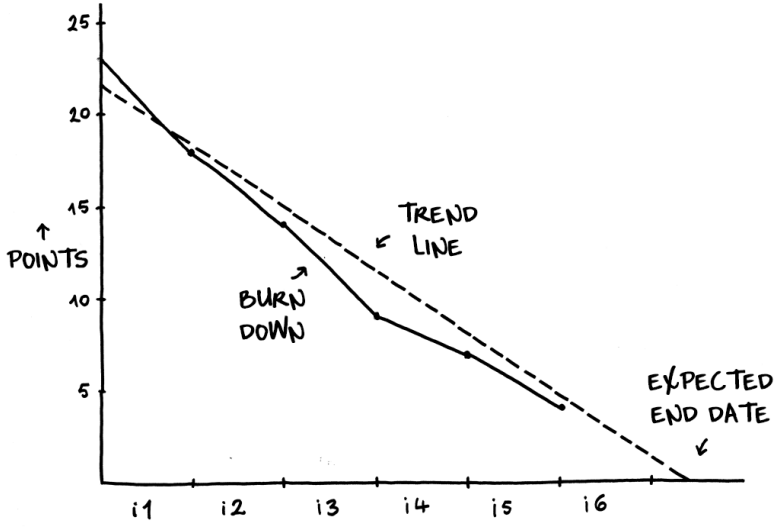


In a project for a pension fund, three smart use cases were left over when reaching the end date. For two of them, the customer mentioned that he would prefer not to get them delivered because each meant additional manual data conversion. The third described a pension statement. Once the customer concluded that the web page on which the pension statement was shown could be printed,this was also dropped. Problem solved.

One possible issue is: how does the contractor ensure that the team performs optimally and does not slack off? Since work

items can be estimated in points, it is possible to agree on a price per point. So the contractor receives less money if fewer points are realized. That way it pays to deliver as many work items per iteration as possible.

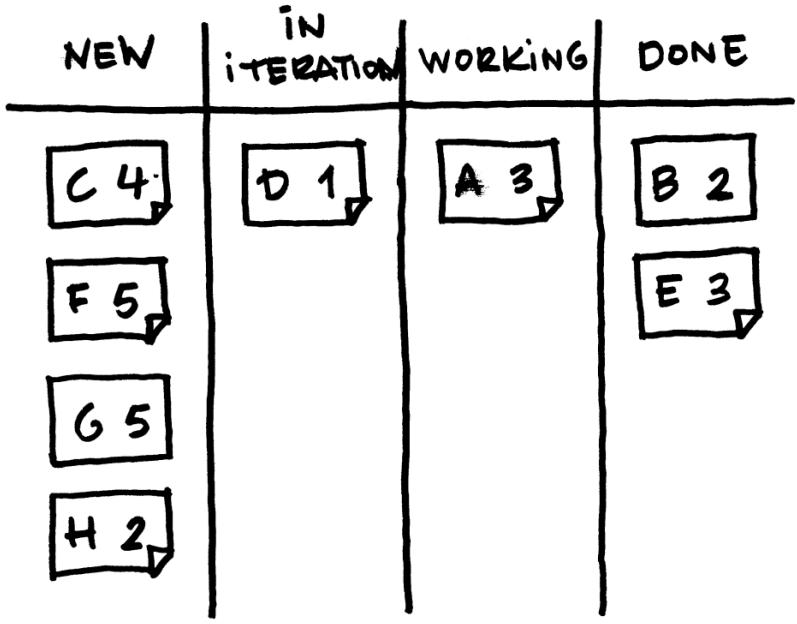
Progress on an agile project is simply monitored on the basis of the number of points earned. Every day the status is determined by comparing this number with the total number of points in the project backlog. Using extrapolation, the *probable* completion date of the project can be determined. This is done using a *burn-down chart*.



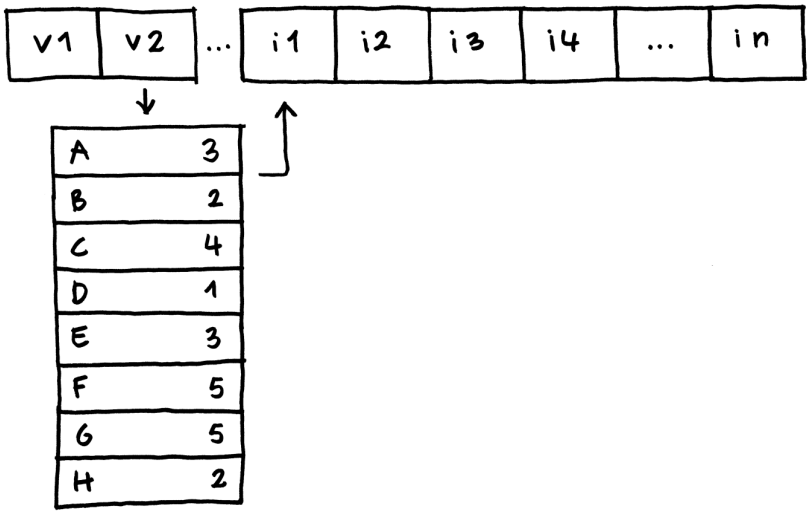
Agile projects facilitate the communication between customer and team as much as possible. Often a single spreadsheet is sufficient to serve as a backlog. Agile projects also use simple dashboards, which are made up of post-it notes on a wall. This provides insight into the work in progress to anyone who is interested. Agile projects are transparent.

Finally, agile projects use various techniques for multidisciplinary collaboration, such as the daily stand-up meeting, which is a to-the-point discussion of the project's status. Agile

projects have a preference for face-to-face communication, and are therefore ideally conducted by the entire team, on site, with the customer. This is called *co-location*.



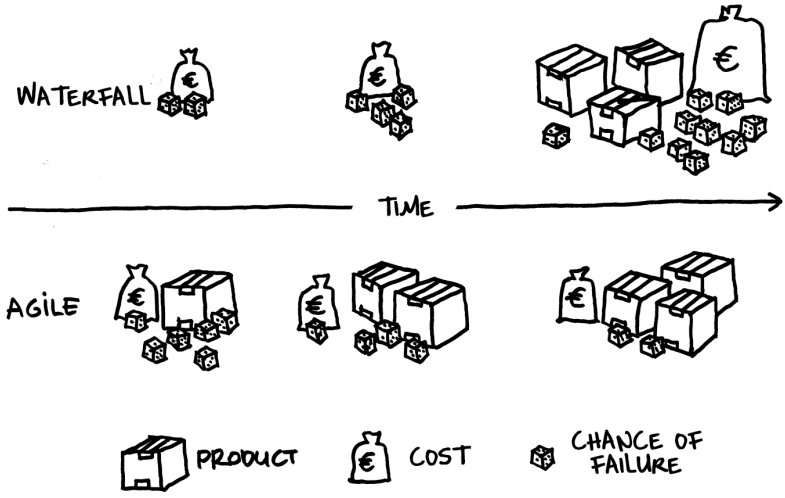
An important question that remains is: when should the backlog be created? There are quite a few differences in the ways the various agile approaches handle the backlog. In lightweight agile approaches such as Scrum and Extreme Programming, the project starts once the backlog is identified. Middleweight agile approaches such as DSDM, Feature Driven Development and Smart use one or several preliminary iterations leading to objective, scope, requirements, investigate architecture and technology and establishing the backlog. These preliminary iterations result in an estimate and project plan.



In all cases, all the work that is not yet needed is not performed. By postponing irrelevant work, agile projects save time and make optimal use of progressive insight. Agile provides tools to people to identify and solve problems at an earlier stage in the project. Agile however, does not provide tools to solve *all* problems. There will always be challenges. People leave the company or have limited availability. People get sick or give birth. Technology fails. Even agile is no silver bullet.

Products, costs and risks

Finally, a brief comparison of agile and traditional projects. By performing each phase once in a fixed sequence, the risks increase in traditional methods as the project progresses and the software is only delivered to the customer at the end of a project. It has to be like this, since earlier on in the project there is no software available, only documentation. This is the so-called *big-bang scenario*.



In agile projects, the software is delivered to the customer in iterations, or releases. When work items are constantly prioritized and are delivered complete and incrementally, the risk decreases as the project progresses, while the produced software is growing. Most agile projects choose a calculation model in which the cost is settled per iteration or release. The choice is yours.

About the Author

Sander Hoogendoorn has been involved in the innovation of software development for 20 years, coaching organizations, teams, projects and individuals at government agencies and organizations across the transportation, healthcare, oil, finance, retail, education and insurance industries. He currently serves as Principal Technology Officer and Global Agile Thought Leader at French international consulting firm Capgemini.

Sander has published over 250 articles in international magazines, has delivered over 100 keynote addresses, and has written several books in his native Dutch on modeling and agile, including this one, which was originally published as *Dit is Agile* by Pearson Benelux S.V., and as *Das kleine Agile-Buch i*n German by Addison Wesley Verlag. He regularly presents seminars and training courses on a variety of topics such as agile, Scrum, Kanban, software estimation, software architecture, design patterns, modeling in UML, .NET, coding, and testing. He is a member of several editorial and advisory boards. This is his first book to appear into English.

Sander lives in the Netherlands with his girlfriend and three children. He is an amateur photographer, guitarist and indoor footballer, and he still loves to write code.

Learn more at www.sanderhoogendoorn.com.