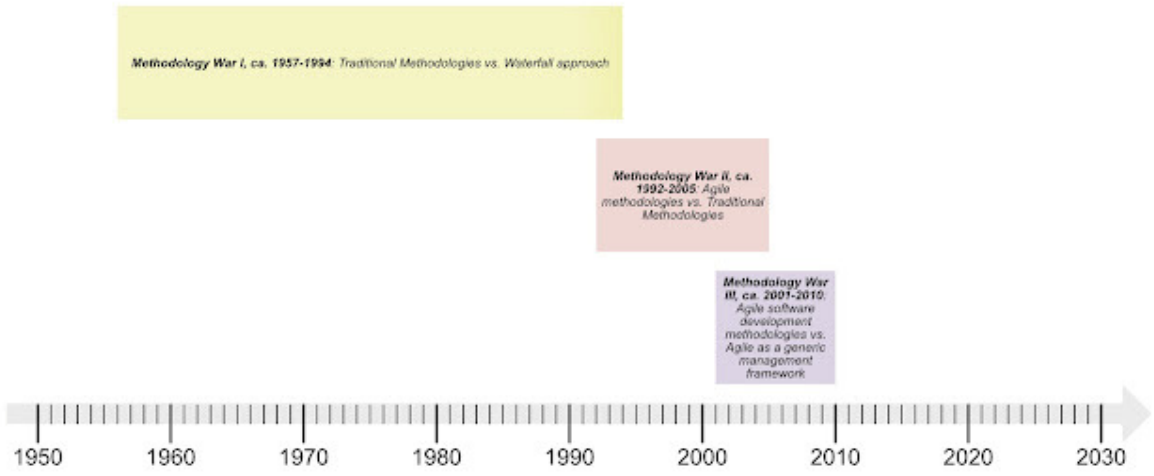
Waterfall vs. Agile: Battle of the Dunces or A Race to the Bottom?

https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

November 16, 2023



The major software methodology wars since the mid 1950's.

Someone was wrong on the Internet, so here we go…

In a recent HBR article, It's Time to End the Battle Between Waterfall and Agile, the author sets up a false premise, that there is a war between Waterfall methodology and Agile, that it must end, and that you can combine the approaches to get the best of both worlds.

This sounds good, but the article is based on a misunderstanding of both Waterfall and Agile. Also, there is no war between Waterfall methodology and Agile. There can't be, because Waterfall methodology does not exist! Waterfall is a name for large projects that failed in the 1960's. Waterfall was never a methodology, but a failure to apply the methodologies that existed back then. As I will show towards the end of this article, at least one of the "successful" Waterfall projects mentioned in the HBR article was neither successful, nor a Waterfall project.

I got invaluable help from Alistair Cockburn. He fact checked the two time-line illustrations in the introduction. If you find screwups in the article text, those are purely mine.

Since the 1950's there have been at least three major software development methodology wars. The first, starting in the 1950's, gathering steam in the 1970's and 1980's, and petering out in the 1990's, was a war between traditional heavyweight methodologies, with features like parallelization of activities, prototyping, and (sometimes) iterative development, and Waterfall, which has none of those things.

The second methodology war was in the 1990's and early 2000's, between traditional, heavyweight methodologies and lightweight agile methodologies. The agile methodologies won, but the victory would not last for long. The agile movement had a civil war brewing even before it had won the war against traditional methodologies.

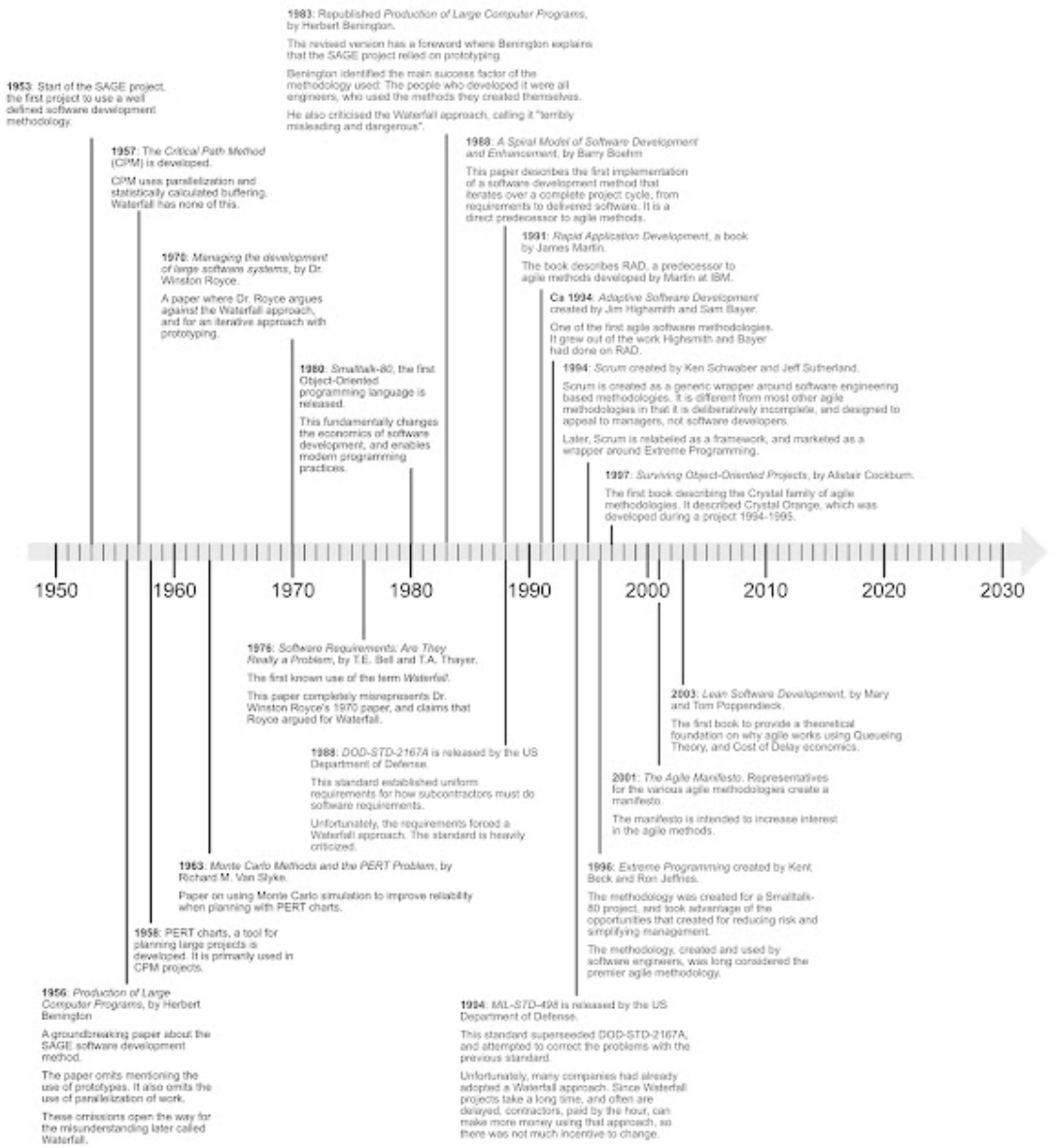
The third methodology war started soon after the Agile Manifesto was published, and was a war between the *agile lightweight methodologies*, based on a combination of software development practices and management practices, and *Agile*, with a capital "A", that focused on management almost exclusively. By 2010, the battle was more or less over, and *Agile* with a capital "A" had beaten *agile* with a lowercase "a" in terms of market share. (Yes, we did fight over whether to use a capital "A" or a lowercase "a" in those days, but there were also more serious things at stake, like the spread, and use, of good software development practices.)

The reason why all of this is important, is that the software industry, at least parts of it, is now trying to revive Waterfall, not as a warning of a failed approach, but as a viable methodology. The past few months I have seen several job descriptions where companies proclaim that they need people for Waterfall projects. There are tons of articles that spread misinformation about Waterfall. The misinformation also proliferates on social media. Waterfall projects don't just waste money, they grind people down, putting them under enormous pressure in no-win situations. It is the antithesis of good software development, and of good management.

I wrote another blog post recently where I used historical documents to correct some of the misinformation. I also provided an example where I compared Waterfall style planning with both traditional heavyweight methods, and with Agile planning. Waterfall lost, even under conditions that are supposedly optimal for Waterfall. Then this HBR article, advocating for marrying the worst aspects of traditional methodologies with the worst aspects of agile frameworks and methodologies, started popping up everywhere I looked, and I…got a bit upset. Hence, this blog post.

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

To understand why advocating for Waterfall software development is bonkers, it helps if we look back, to the origins of the approach, and how it was shaped by the realities of software engineering in the early 1950's.

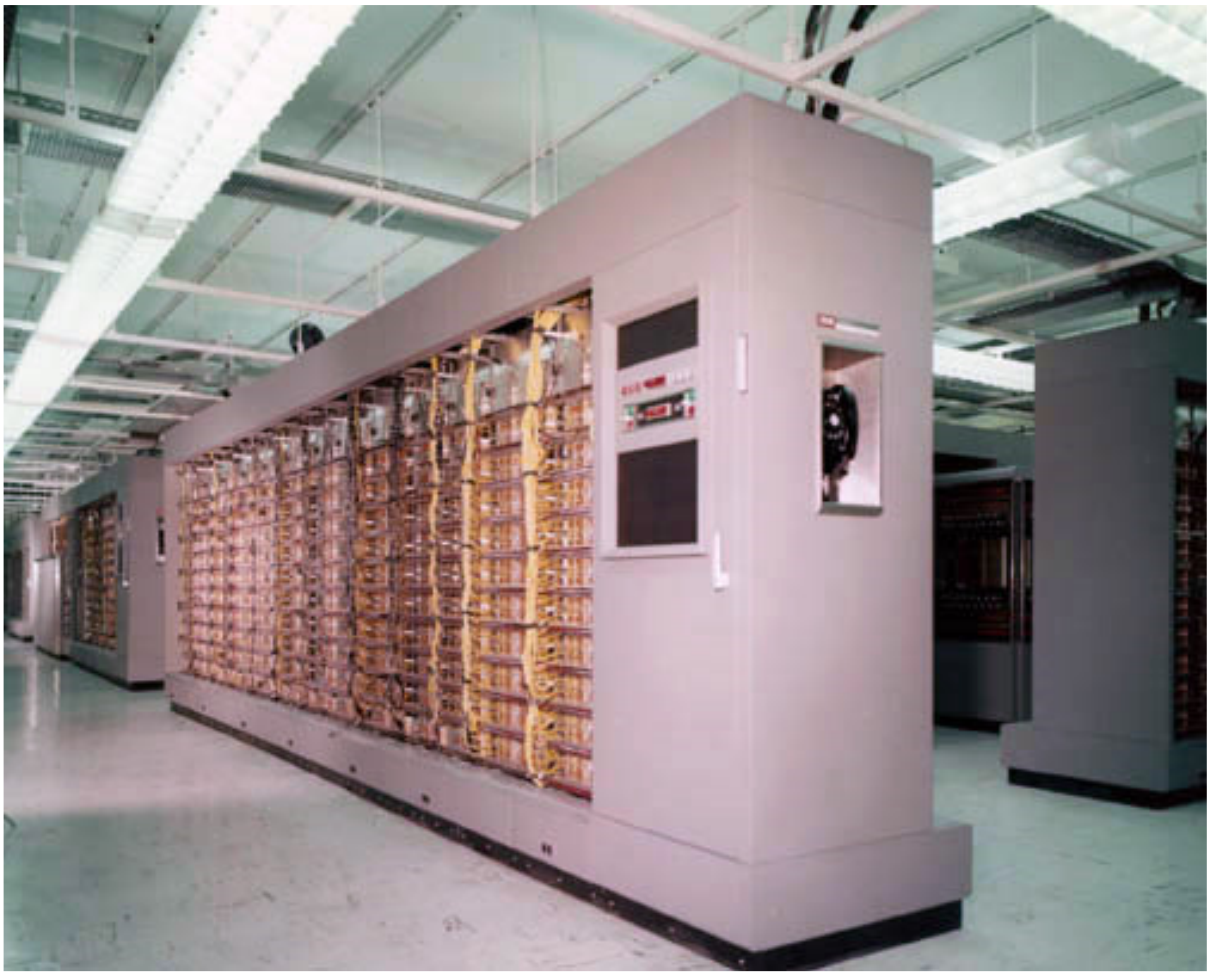


Timeline covering some of the major events during the time period of the three methodology wars in this blog post.

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

1953-1956 A.D.: In the beginning was SAGE!

The *Semi-Automated Ground Environment* (SAGE) was a system of networked computers used to coordinate data from many radar stations, and collate it into one unified picture. The SAGE development project started in 1953, and went operational in the late 1950's. Developing the SAGE software was a huge undertaking at the time. It was the first software development project large enough to require a software development methodology, so the engineers at SAGE created one.



Part of the SAGE computer room.

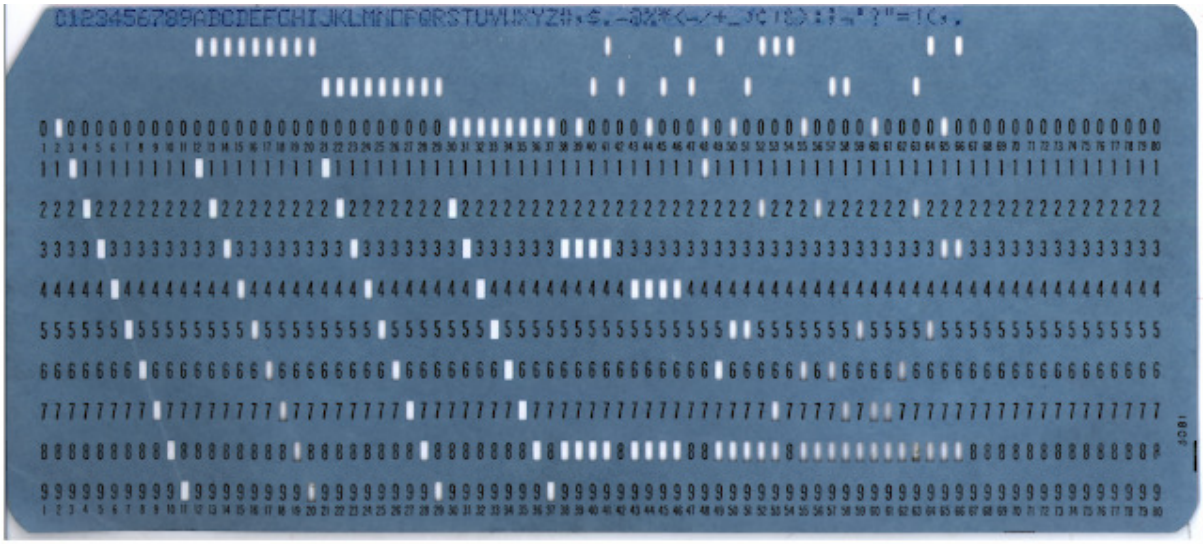
SAGE used a centralized computer known as the AN/FSQ-7 Combat Direction Central, or Q7, for short. The Q7 weighed about 250 tons, had 60,000 vacuum tubes, required 3 MW of energy to run, and could execute 75,000 instructions per second. For comparison, the

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

neural engine in an iPhone 14 Pro can execute 17,000,000,000,000 (17 trillion) instructions per second. The iPhone also weighs considerably less than 250 tons.

Thanks to a 1956 paper written by Herbert D. Benington, a software engineer at SAGE, we know a lot about how the people at SAGE worked. Benington is often credited with creating Waterfall in the SAGE project. What is funny, is that Dr. Winston Royce, of NASA, is also often credited with creating Waterfall, but later, in 1970. What is even funnier, is that neither of them did it, and both were very much against Waterfall.

I have written about Dr. Winston Royce and his 1970 paper in a previous blog post. This time, I'll focus on Benington's 1956 paper, Production of Large Computer Programs.



A punch card, similar to the ones used in the SAGE project.

Programs are prepared in machine language because automatic coding techniques developed to date do not guarantee the efficient programming required for a real-time system.

— Production of Large Computer Programs, Herbert Benington

In the SAGE project, programmers wrote machine code instructions directly, using punch cards. A computer program was written on stacks of cards.

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

A large computer program, written on punch cards. This is what the programs in the SAGE project looked like, before they were entered into the Q7 computer.

Because of the way programs were written, and because computer time was very expensive, the engineers at SAGE paid close attention to economics:

Let us assume an overhead factor of 100 percent (for supporting programs, management, etc.), a cost of $15,000 per engineering man-year (including overhead), and a cost of $500 per hour of computer time (this is probably low since a control computer contains considerable terminal equipment). Assuming these factors, the cost

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

of producing a 100,000- instruction system program comes to about $5,500,000 or $55 per machine instruction.

— Production of Large Computer Programs, Herbert Benington

$55 per machine instruction in 1956, recalculated with an inflation calculator, works out to about $608 per machine code instruction today (SKR 6636, if you live in Sweden, like me). Today, we don't even bother figuring out what it costs to produce one machine code instruction. A software developer today can produce many machine code instructions by writing a single line of code in a high level language.

Writing a 100,000 machine code instruction system was a major undertaking in 1956. Today, hobbyist programmers write applications much, much larger than that, just for the fun of it.

Because writing programs was so expensive, the SAGE engineers wanted to be very sure they did not make errors. Get one machine code instruction wrong, and fixing that instruction might not be enough. You could easily get cascading errors, making it necessary to rewrite large chunks of code, at prohibitive cost.

The risk of errors was high. Writing machine code is difficult and risky at best. Doing it by punching holes in punch cards increases the risk a lot. Testing machine code was difficult and expensive. Today, a developer working with modern object-oriented languages can quickly write tests for small pieces of code, even before writing the code to be tested, using Test-Driven Development, or Behavior-Driven Development. In 1956, testing and fixing errors was not an easy task.

So, if the risk of errors is very high, and finding and fixing the errors is incredibly expensive, what do you do? You try to prevent the errors. How do you prevent errors?

The Waterfall advocates would have you believe the only solution is to follow a strictly linear process, with planning everything up front, before starting to code, then coding everything before starting to test. However, that was not quite what the SAGE team did.

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

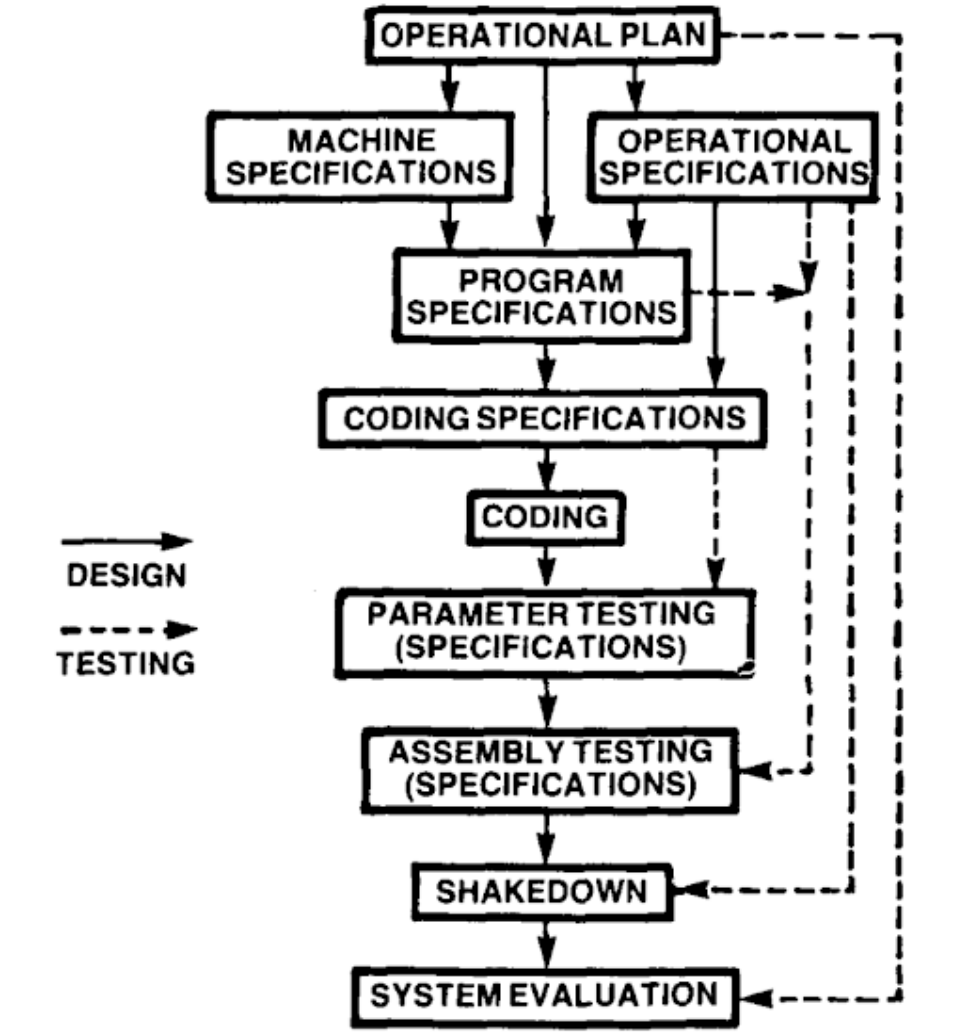


Figure 4. Program production. Production of a large-program system proceeds from a general operational plan through system evaluation; for example, assembly testing verifies operational and program specifications.

An outline of the software development process from Benington's 1956 paper. In 1983, Benington reissued the paper, and pointed out that very important parts of the process were missing in the original paper.

If you had read Benington's paper in 1956, it would be very easy to believe that Benington described a rigid, strictly top down process, proceeding from an overarching operational plan, through step by step refinement, until, finally, the program could be evaluated and released. However, that was not how they actually worked at SAGE, and it wasn't Benington's intent that it should be perceived that way.

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

In 1983, Benington reissued his 1956 paper with a new foreword. What he wrote in the foreword is illuminating:

When I got back into the computer programming business several years ago, I read a number of descriptions of top-down programming. The great majority seemed to espouse the following approach: we must write the initial top-down specification (for example, the A Spec), then the next one (typically, the B Spec), so we will know precisely what our objectives are before we produce one line of code. This attitude can be terribly misleading and dangerous.

— Foreword to the 1983 reissue of Production of Large Computer Programs, Herbert Benington

"Terribly misleading and dangerous", is what Benington himself thought about the Waterfall approach. Even in 1956, the SAGE engineers understood that a strict top down approach was not the answer. Even though the process was basically linear, they were not as rigid about it as Waterfall is, and they did one very important thing. They built prototypes!

I do not mention it in the attached paper, but we undertook the programming only after we had assembled an experimental prototype of 35,000 instructions of code that performed all of the bare-bone functions of air defense.

— Foreword to the 1983 reissue of Production of Large Computer Programs, Herbert Benington

Prototyping is something that shows up over and over again in software development, because it is incredibly useful. Dr. Winston Royce proposed a method that used it in 1970. Some other heavyweight methods, including Boehm's Spiral Model, and the Rational Unified Process (RUP) used it. Agile methodologies like Crystal Clear, Lean Software Development, Dynamic Systems Development Method (DSDM), and Adaptive Software Development (ASD), all use prototyping. Extreme Programming uses *spikes*, a kind of small, partial prototypes.

Both Benington in 1956 and Royce in 1970 advocated for prototyping, as did most agile software development methods. In stark contrast, Waterfall, Scrum, and Agile (with a capital "A") do not.

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

In the foreword to the 1983 reissue of his paper, Benington writes about what the SAGE project could have done better:

To underscore this point, the biggest mistake we made in producing the SAGE computer program was that we attempted to make too large a jump from the 35,000 instructions we had operating on the much simpler Whirlwind I computer to the more than 100,000 instructions on the much more powerful IBM SAGE computer. If I had it to do over again, I would have built a framework that would have enabled us to handle 250,000 instructions, but I would have transliterated almost directly only the 35,000 instructions we had in hand on this framework.

Thus, Benington wanted *more* prototyping, not zero prototyping, as in Waterfall.

Benington continues:

Then I would have worked to test and evolve a system. I estimate that this evolving approach would have reduced our overall software development costs by 50 percent.

This is interesting! In 1983, Benington wished the SAGE team would have used an evolving approach, based on prototyping and testing, in other words, an *emergent design*!

Wow! This is the same idea Extreme Programming, Crystal, and other agile methodologies would advocate in the 90's and early 2000's. Benington's own estimate is that it would have reduced project cost by 50%!

Benington also points out that software development had changed a lot since 1956. He listed the technical advances that he believed had the largest impact on project methodology. In Benington's own words:

- We now use higher-order languages in virtually all situations.
- Almost all software development and unit testing are done interactively at consoles in a timesharing mode.
- We have developed a large family of tools that allow us to do much precise design and flow analysis before coding. (I still say that we should use these techniques before we start finalizing our top-down requirements.)

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

- We have developed organizational approaches that improve or at least guarantee the quality of the systems much earlier in the game. These include some of the structured languages, code reviews, walk-throughs, etc.

This is a *brutal* takedown of Waterfall!

Waterfall proponents today do of course not advocate that developers should write machine code using punch cards. However, they are advocating we should manage developers *as if* they were writing machine code on punch cards…and that we should do it badly, without prototyping and with a top down approach Benington called "misleading and dangerous".

In 1956, Herbert Benington and his peers understood very well that the way you plan and manage projects must be adapted to the way people work.

In 2023, managers, including Agile managers, have divorced the management methodologies and frameworks from what they manage, and the results are disastrous. Instead of trying to understand what the problem is, they are desperately seeking for a quick fix, something to latch on to, and Waterfall just happened to sound cool to them. What it is, and whether it works, does not matter, as long as it can be packaged and sold.

Methodology War I, ca. 1957-1994: Traditional Methodologies vs. Waterfall approach

The initial misunderstanding of Benington's 1956 paper, that gave rise to the Waterfall approach, is very understandable. The paper omitted key practices, and over-emphasized working top down. Waterfall tended to produce large cost overruns and bad software, but it was easy to follow, step by step.

The poor results from Waterfall was a problem, and that lead to the development of techniques, methods, and methodologies designed to improve software project results.

1957-1963: Critical Path, PERT, and Monte Carlo

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

The Critical Path Method (CPM), a project scheduling method, was invented in 1957. With CPM, a project was broken down into relatively small work packages. The next step was to figure out how to sequence work packages, and which sequences could be done in parallel. The parallelization of work made it possible to massively reduce project lead times compared to Waterfall. This also reduced project cost. Because software could be released earlier than with Waterfall, it could be put to use earlier, and begin to generate revenue earlier.

CPM projects sometimes, but not always, made multiple deliveries with partial functionality in each delivery. Waterfall projects cannot do that, because of the strictly linear sequencing of project activities. With Waterfall, you either have a big bang release at the end of the project, or the whole thing fizzles, you have nothing, and you must start over more or less from square one again. (This was one of the reasons Dr. Winston Royce criticized Waterfall in his 1970 paper.)

Program Evaluation and Review Technique (PERT) charts were invented in 1958. The main use was as a planning tool for CPM projects. With CPM, you plan by identifying the longest path of activities from the beginning to the end of the project. This makes it possible to identify which activities are critical on the Critical Path, and which activities are not, usually because they are on other paths. PERT charts made it easier to visualize the pathways through the project.

A problem with CPM was that projects were often delayed. Not as much as with Waterfall, but serious enough. In 1963, Richard Van Slyke wrote a paper named Monte Carlo and the PERT Problem, where he suggested using Monte Carlo simulation to improve project planning.

The Monte Carlo method is a statistical simulation method originally developed by Stanislav Ulam at the Manhattan Project in 1946. Ulam got the idea while playing solitaire. He tried to figure out a way to calculate the probability of successfully laying out a 52 card Canfield solitaire. He soon realized that the method could also be used to solve nuclear physics problems.

Over time, Monte Carlo simulation spread into project planning, finance, engineering, climate change research, computational biology, computer graphics, applied statistics, artificial intelligence, US Coast Guard search and rescue operations, and other areas. Today, it is even used by a few people in the Agile community.

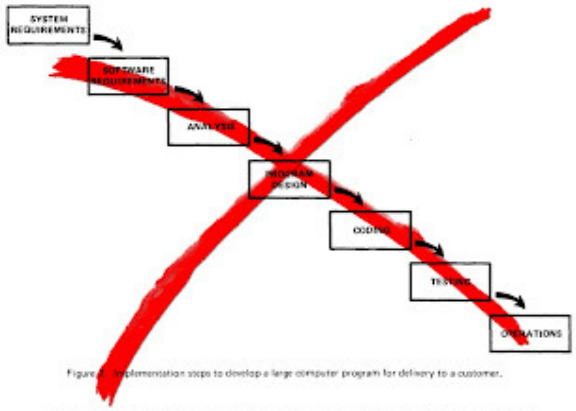
Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

1962-1980: New languages begets new methodologies

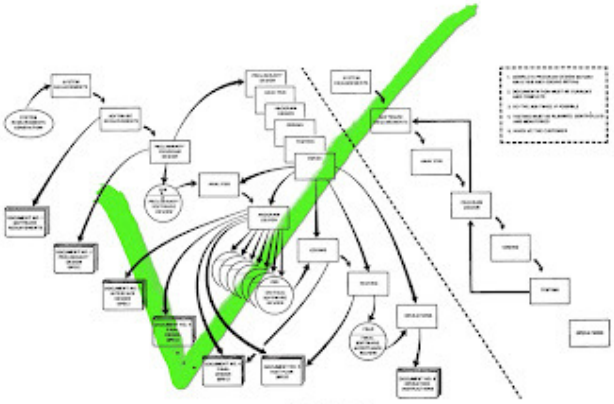
Computer programming changed a lot during the sixties and seventies, with the rise of high level languages. There had been high level languages around earlier, but in the sixties and seventies, we got languages that were more powerful and easier to use: APL (1962), PL/I (1964), BASIC (1964), Forth (1968), Pascal (1970), C (1972), Prolog (1972), SQL (1972), ML (1973), and MATLAB (1978-ish), to name just a few.

These languages, and of course the corresponding advances in hardware, profoundly changed the nature of programming. With those changes came just as profound changes in software development methodology and economics.

1970-1976: The most misunderstood methodology paper of all time



This is Waterfall. It is what Dr. Winston Royce argued *against* in his 1970 paper *Managing the Development of Large Computer Systems.*



This is what Dr. Royce argued *for* in his 1970 paper: An iterative method, with prototyping, and close continuous contact with customers.

In his 1970 paper, Managing the Development of Large Computer Systems, Dr. Winston Royce argued against Waterfall, and for a method that used prototyping, iterations, and continuous contact with customers.

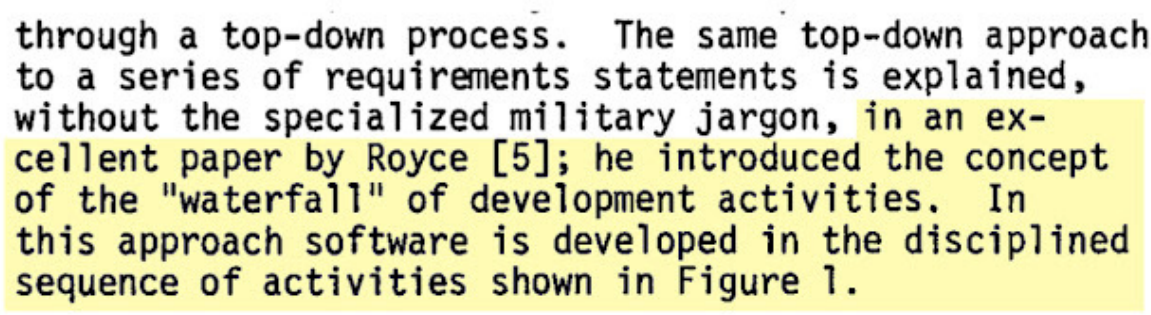
There was a lot of dissatisfaction with the Waterfall approach in the 1960's. This prompted Dr. Winston Royce of NASA to write a paper, Managing the Development of Large Software Systems, where he criticized Waterfall.

…the implementation described above is risky and invites failure.

— Managing the Development of Large Software Systems, Dr. Winston Royce, 1970

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

Unfortunately, the Waterfall advocates were undeterred. They simply claimed that Dr. Winston Royce was with them.


through a top-down process. The same top-down approach to a series of requirements statements is explained, without the specialized military jargon, in an excellent paper by Royce [5]; he introduced the concept of the "waterfall" of development activities. In this approach software is developed in the disciplined sequence of activities shown in Figure 1.

From a 1976 paper Software Requirements: Are They Really a Problem? by T. E. Bell and T. A. Thayer. The problem with their praising of Royce, is that they praised him for the thing he was against.

It is difficult to find out exactly who first misunderstood Winston Royce. T.E. Bell and T.A. Thayer are two of the main contenders though. In the 1976 paper Software Requirements: Are They Really a Problem?, they praised Royce's approach to software development. It becomes very apparent that something is wrong when you look at the illustration of what they praised.

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

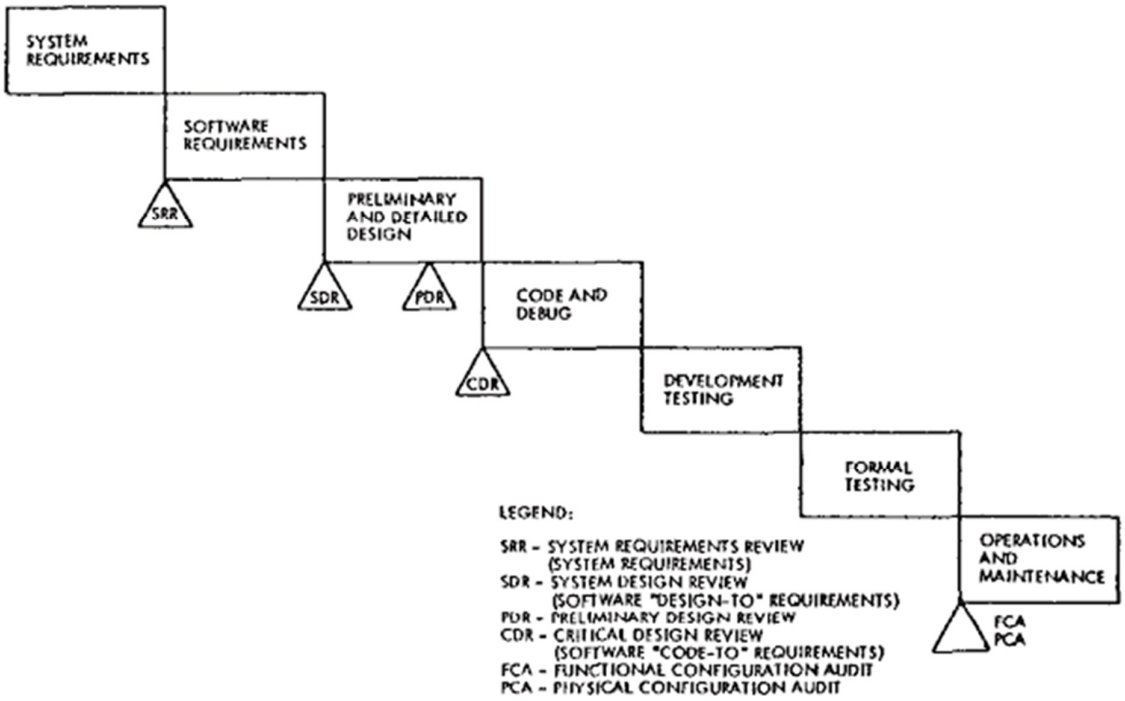


Figure 1. Development Phases of the System Development Cycle

This is Figure 1 from Bell's and Thayer's paper. The model they are praising, the Waterfall model, is the model Royce argued against.

Looking at Bell's and Thayer's figure 1 in their paper, it becomes clear they confused Royce's description of the problems with Waterfall, with a description of a solution.

This may well be one of the three biggest blunders in the history of software development methodology! (The other two would be moving from agile methodologies to Agile with a capital "A", and trying to combine Agile and Waterfall.)

It is worth noting that the Bell and Thayer paper *may* be the first time the term "Waterfall" is used. If you are interested in a more detailed account of how Royce's paper was misunderstood, you may wish to read a previous blog post of mine.

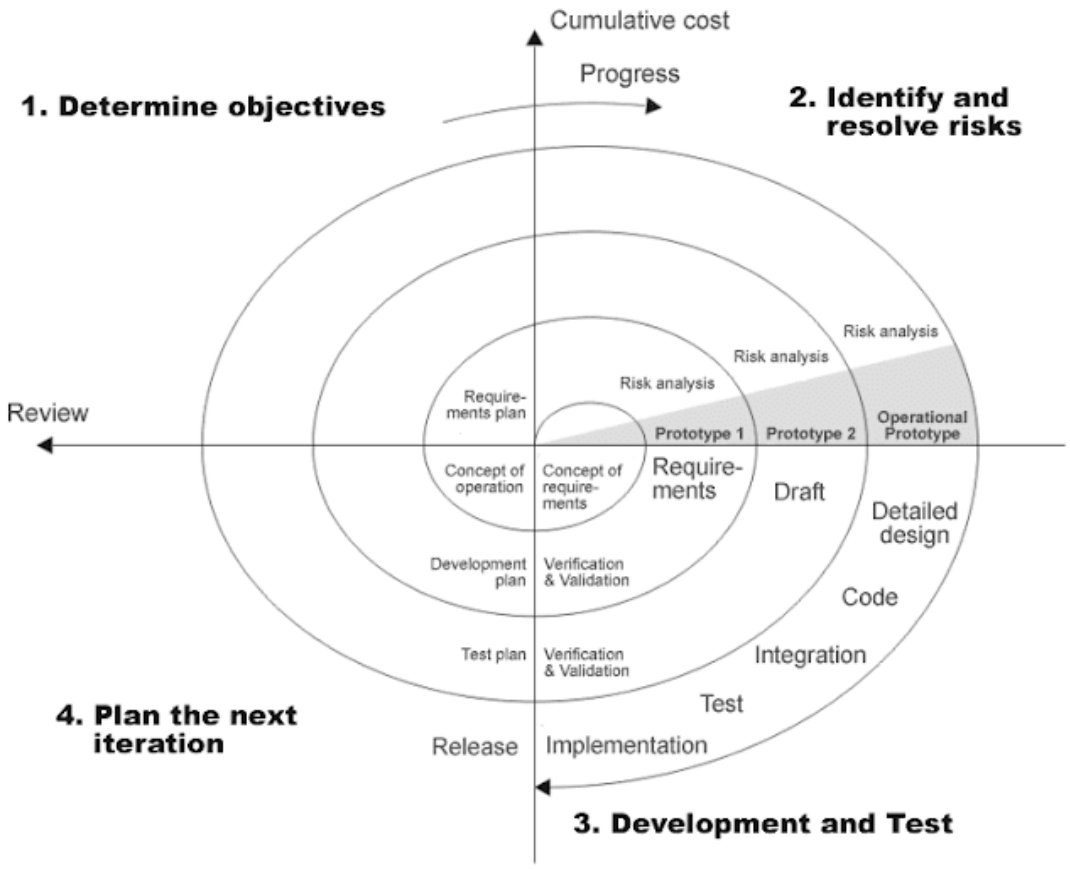
Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

1980: Smalltalk-80 - The first object-oriented programming language

In 1980, the first object-oriented language, Smalltalk-80, was released. This sparked yet another revolution in programming techniques, and software architecture, and would eventually enable the emergence of agile software development methodologies in the 90's. Today, object-oriented languages like Java, C#, Python, and Ruby are all descended from Smalltalk-80, and use the same programming paradigm (though there are significant differences in how they apply the paradigm).

New software development methodologies that, to varying degrees, took advantage of the improvements in technology and programming languages emerged throughout the 60's, 70's, and 80's. These methodologies were in competition with Waterfall, but also with each other.

1988-1994: The rise of the spiral, and the death of Waterfall



Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

Barry Boehm's Spiral Model of Software Development was designed to solve problems with the Waterfall approach.

In 1988, Barry Boehm's paper A Spiral Model of Software Development and Enhancement revolutionized software development methodology. It was a fully iterative method, it used prototyping, and it was explicitly intended to solve the problems with Waterfall:

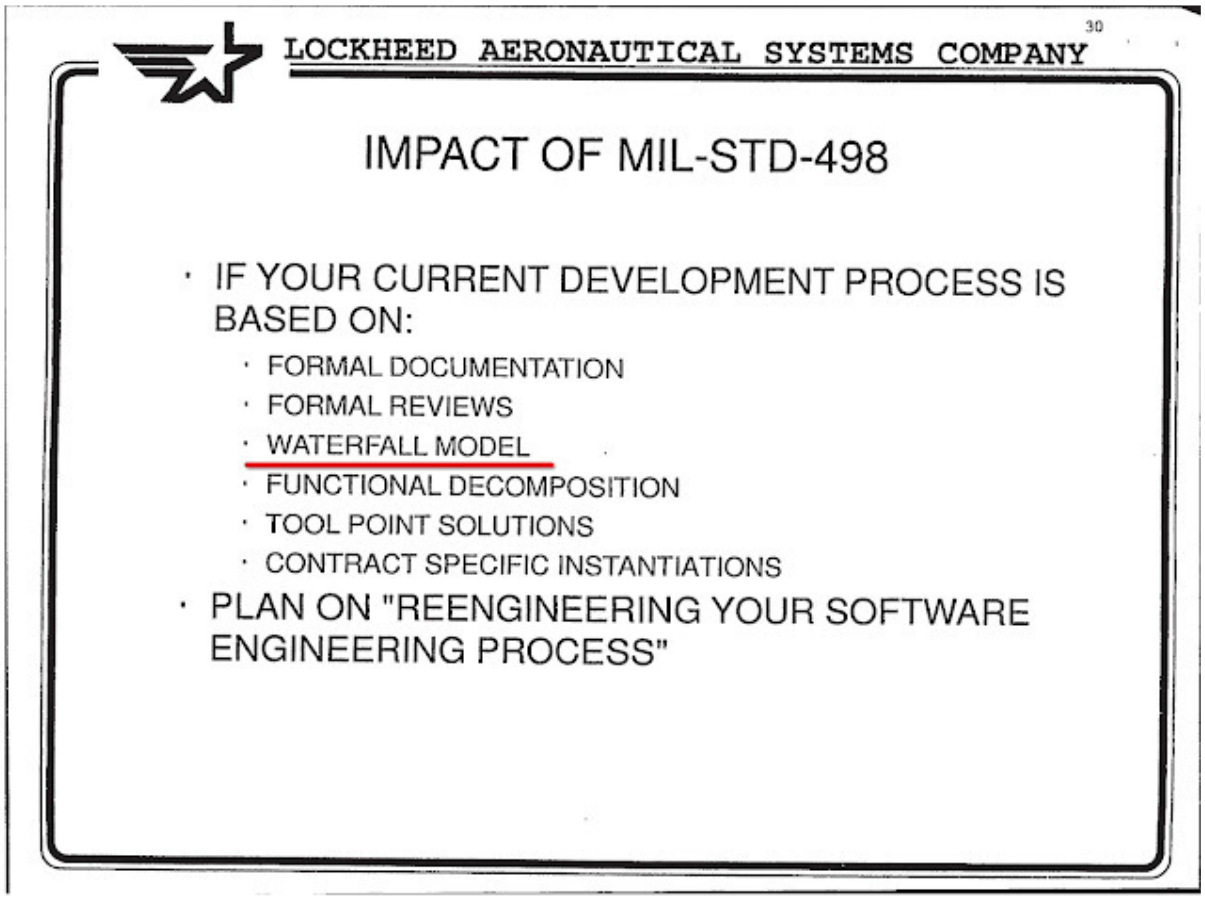
A primary source of difficulty with the waterfall model has been its emphasis on fully elaborated documents as completion criteria for early requirements and design phases. For some classes of software, such as compilers or secure operating systems, this is the most effective way to proceed. However, it does not work well for many classes of software, particularly interactive end-user applications. Document-driven standards have pushed many projects to write elaborate specifications of poorly understood user interfaces and decision support functions, followed by the design and development of large quantities of unusable code.

— A Spiral Model of Software Development and Enhancement, by Barry Boehm, 1988.

By this time, the enthusiasm for Waterfall was diminishing on all fronts. Waterfall simply did not deliver. It was an anachronism from the days of punch cards, and the problems were glaringly obvious for all to see…except maybe for the US Department of Defense.

In 1988, the US Department of Defense released DOD-STD-2167A, a military standard for how to run software projects. Unfortunately, though the standard did not prohibit more modern methodologies, it was written in a way that encouraged Waterfall projects.

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html



From a presentation about MIL-STD-498 produced by the Software Productivity Consortium in 1994.

Waterfall projects kept failing, and new, improved methodologies kept cropping up, for example James Martin's *Rapid Application Development* (RAD), in 1991. In the beginning, RAD was basically a way of creating prototypes very rapidly, but it would inspire the development of several true agile software development methodologies.

DOD-STD-2167A did meet with heavy criticism because it locked military contractors into using bad methodology, and also locked them out of using technical improvements. In 1994, it was replaced by MIL-STD-498, a standard designed to rectify some of the earlier mistakes. It explicitly removed the Waterfall model implied by the previous standard.

By this time, Waterfall had little or no credibility. No one would deliberately choose to run a Waterfall project, even though badly run projects still could devolve into Waterfall.

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

Methodology War II, ca. 1992-2005: Agile methodologies vs. Traditional Methodologies

The traditional methodologies continued to develop during the 90's. They still had a heavy emphasis on documentation, and projects could sometimes end up producing a lot of documentation, but no working software. They were still a marked improvement over Waterfall. In the late 90's, the *Unified Process*, and the *Rational Unified Process*, two very closely related iterative heavyweight methods emerged.

To give you an idea of how administration heavy these projects could be:

In 2005 I worked as a consultant for a large company using a heavyweight methodology. In my first meeting with my team, we introduced ourselves. It turned out we were eight managers and administrators, and a single developer.

What was my job? I was the process navigator. It was my job to figure out what the next step in our process was, what documents we needed to produce to pass each of the fourteen gates, what permissions we needed to proceed at each stage, and a lot of other things.

The project eventually ground to a halt due to process errors. For example, we used a new version of the methodology, released just the week before our project started. At one point, we needed to produce a document in order to pass a gate. To produce the document, we needed access to certain information, but we were not allowed to access the information until after we had passed the gate.

At another point in the project, we needed permission from a department in another country in order to proceed. I managed to find the name of a contact person, but when I called him up, he told me he could not give us permission to proceed, because he did not work at that department anymore. It had been closed down for three years.

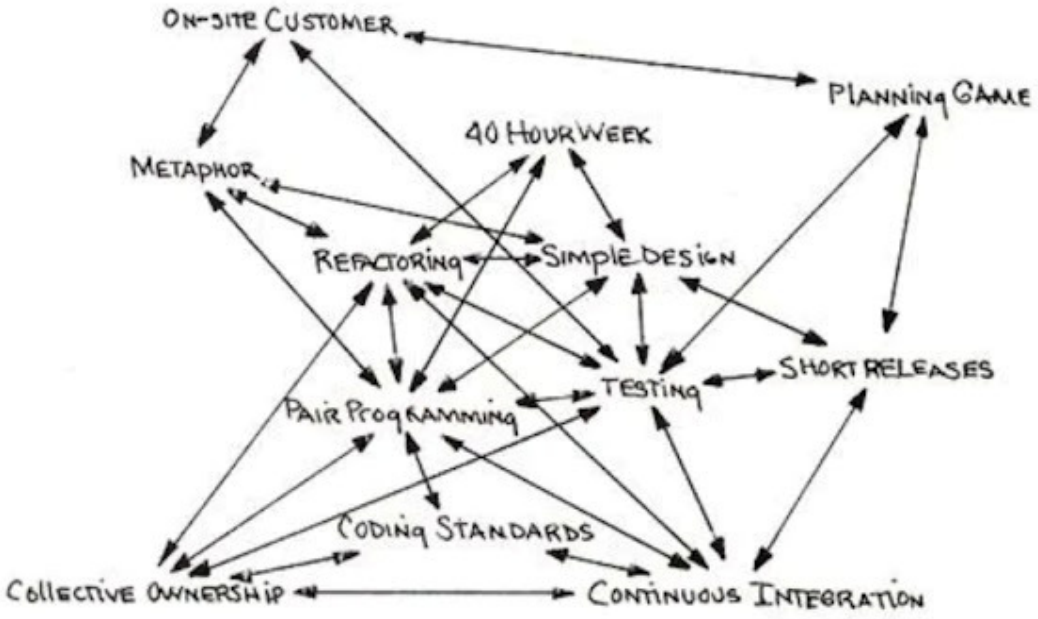
Then I found out, after a bit of digging, that we were the fourth team trying to build the same thing. Each time a project was started, it ran into problems like the ones I've described. The project ground to a halt. Then, instead of fixing the process problems, the company restarted the same project, but with new people, over, and over again.

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

It is no wonder that some people wanted to create new, more flexible and responsive software development methodologies.

1991-1998: The Rise of Agile Software Development Methodologies

Rapid Application Development (RAD) (1991), an important advancement in methodology, inspired the development of at least two agile methodologies *Adaptive Software Development* (1994) by Jim Highsmith and Sam Bayer, and Dynamic Systems Development Method (DSDM) (1994).

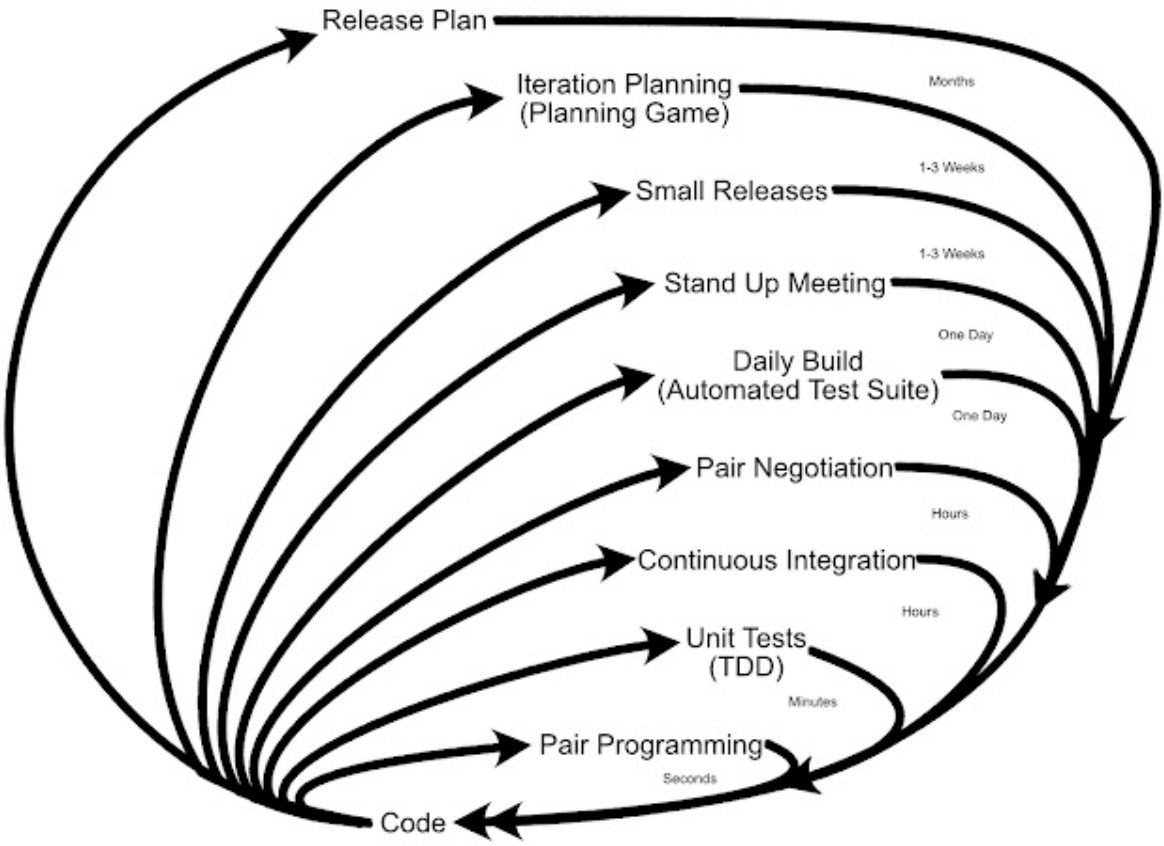


Extreme Programming (XP) is a system of supporting practices. For a long time, XP was considered the heart of agile.

Likewise, the agile methodology Extreme Programming (XP) (1996), created by Kent Beck and Ron Jeffries, was partially inspired by Boehm's Spiral Method. Like the Spiral Method, the overarching idea with Extreme Programming was to reduce risk. Extreme Programming did this, to a large extent, by taking advantage of programming techniques made possible by Smalltalk-80, the first object-oriented language.

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

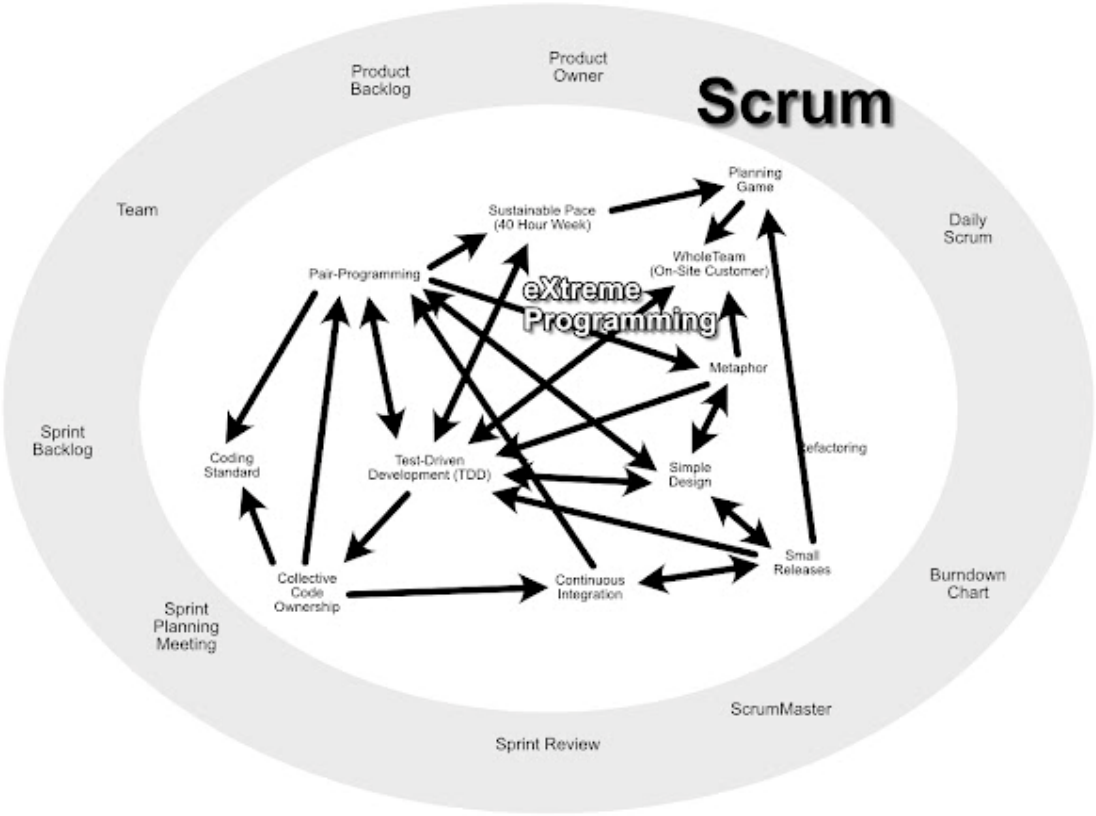
Extreme Programming was, very deliberately, designed as a system of supporting practices. Used together, these practices kept the development project stable, and developers sane.



Extreme Programming feedback loops lasted from mere seconds with Pair Programming, to months with Release Plans.

Extreme Programming also borrowed ideas about feedback loops from Systems Thinking, specifically the idea of using multiple feedback loops to keep a system, in this case a project, on track.

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html



Scrum is "deliberately incomplete". The intent was that it should serve as a wrapper around other agile methodologies. Often, that other methodology was Extreme Programming.

Scrum, created by Ken Schwaber and Jeff Sutherland, arrived publicly with a paper published in 1994. This was followed up by the first book about Scrum in 1995. At the time, Scrum was a bit of an odd duck in the agile community, because it had no software development practices. It focused entirely on management practices, and was marketed as a wrapper around other agile methods.

Developers were not very interested in Scrum, because it did not have any software development practices. However, Scrum had a hidden feature that, a few years later, would give it a decisive advantage in the third methodology war, the *agile civil war*.

Crystal (1994-1995), by Alistair Cockburn, was a whole family of agile methodologies. Cockburn held that agile methodologies should be adapted according to the type of project. He rated projects on two properties, size and criticality. Most agile

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

methodologies at the time were designed for small systems development, with a single team, and relatively low criticality, usually no higher than loss of discretionary money. Crystal could be adapted from 6-200 people, and a criticality from loss of comfort to loss of life.

There were several other agile methodologies. Compared to heavyweight methodologies, they minimized administrative overhead, and were designed to enable quick change. Most had software engineering practices that enabled *emergent design*. That is, the software architecture could evolve during the course of a project, just as Herbert Benington had recommended in 1983.

1994 was the year things started to come together for the agile software development methodologies. Alistair Cockburn, Jim Highsmith, Ken Schwaber and Jeff Sutherland, all published their first methodology papers that year. All of them also wrote and published books about their methodologies in the following year.

Though the pieces were in position, agile software development wasn't quite a movement yet. While working on this blog post, I asked Alistair Cockburn to review the timeline pictures in the introduction. He told me that by 1997-1998, the creators of some of the methodologies had begun talking to each other, and that was when a sense of community began to build.

It is important to understand that at this time, the agile movement was to a large extent a software developer and software engineer movement. The methodologies combined good software development practices with good management practices, and that is what made them so successful.

Just like Herbert Benington and the other SAGE engineers in 1956, and Dr. Winston Royce in 1970, the agile software engineers designed methods that took advantage both of advancements in technology and programming languages, and the latest advancements in management.

Though Scrum existed, it was not a major part of the movement, and would not come into focus until after the Agile Manifesto was published in 2001.

2001: The Agile Manifesto

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

On February 11-13, 2001, at The Lodge at Snowbird ski resort in the Wasatch mountains of Utah, seventeen people met to talk, ski, relax, and try to find common ground—and of course, to eat. What emerged was the Agile 'Software Development' Manifesto. Representatives from Extreme Programming, SCRUM, DSDM, Adaptive Software Development, Crystal, Feature-Driven Development, Pragmatic Programming, and others sympathetic to the need for an alternative to documentation driven, heavyweight software development processes convened.

Now, a bigger gathering of organizational anarchists would be hard to find, so what emerged from this meeting was symbolic—a Manifesto for Agile Software Development—signed by all participants.

— History: The Agile Manifesto page at the Agile Software Development Manifesto website

Contrary to popular opinion, the Agile Software Development Manifesto did not mark the start of the agile movement. As we have seen, it had been brewing for a long time. The various agile methodologies started to coalesce into a movement around 1997-1998.

Nor is the Agile Software Development Manifesto a recipe for how to do agile development. It is a *symbolic* statement, outlining the parts the participants at the meeting in Snowbird could agree on.

To understand what the Manifesto is a symbol of, it helps to look at who came up with the idea, who the people who signed it were, and what they represented. Knowing that will help us understand what happened during the following Agile Civil War. Understanding the Agile Civil War helps us understand the current state of Agile, why the entire Agile movement is in trouble now, and why we get hare-brained, but influential, ideas about combining dysfunctional Agile with dysfunctional Waterfall. We might, just might, also gain some insight into what to do actually improve software development, for users, for developers, and for everyone else involved

Fortunately, looking up who took the initiative to the Snowbird meeting is easy to do, because the information is available on the Agile Manifesto website.

The meeting at Snowbird was incubated at an earlier get together of Extreme Programming proponents, and a few "outsiders," organized by Kent Beck at the Rogue River Lodge in Oregon in the spring of 2000.

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

— History: The Agile Manifesto page at the Agile Software Development Manifesto website

In September 2000, Bob Martin from Object Mentor in Chicago, started the next meeting ball rolling with an email; "I'd like to convene a small (two day) conference in the January to February 2001 timeframe here in Chicago. The purpose of this conference is to get all the lightweight method leaders in one room. All of you are invited; and I'd be interested to know who else I should approach."

— History: The Agile Manifesto page at the Agile Software Development Manifesto website

The Agile Manifesto meeting was initiated by Extreme Programming proponents, these were extremely dedicated software engineers. Kent Beck did not just create eXtreme Programming, he also created (according to Beck himself "re-discovered") Test-Driven Development, invented xUnit, and with Ward Cunningham, the less known, but very useful, Class-Responsibility-Collaboration Cards (CRC Cards).

Bob Martin ("Uncle Bob"), who took the initiative to the conference in Snowbird, is also a very influential software engineer. His company, Object Mentor, provided training on Extreme Programming. He is also credited with coining the acronym SOLID, encompassing five important object-oriented software design principles. Martin has also done important, and very influential, work on software development metrics.

It was Bob Martin that came up with the idea of writing an agile manifesto.

Ok, now we know the people who got the ball rolling were some of the most hardcore software engineers in the world. What about the people who attended? Who were they? Let's do a walk-through:

- *Mike Beedle*: Software Engineer and methodologist. A proponent of XBreed, a method that combined Scrum and Extreme Programming.
- *Arie van Bennekum*: Methodologist and DSDM proponent.
- *Alistair Cockburn*: Software engineer, methodologist, and creator of the Crystal family of agile methodologies.
- *Ward Cunningham*: Software engineer. Inventor of the Wiki, and co-inventor of Class-Responsibility-Collaboration Cards. Extreme Programming Proponent.

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

- *Martin Fowler*: Chief scientist at Thoughtworks. Software engineer. Has written several influential books. Extreme Programming proponent.
- *Jim Highsmith*: Methodologist. Proponent of Adaptive Software Development.
- *Andrew Hunt*: Software engineer. Proponent of Pragmatic Programming.
- *Ron Jeffries*: Software engineer. Co-creator and proponent of Extreme Programming.
- *Jon Kern*: Software engineer and methodologist. Proponent of Feature-Driven Development (FDD).
- *Brian Marick*: Software testing consultant and programmer.
- *Robert C. Martin* ("Uncle Bob"): Software engineer. Extreme Programming proponent.
- *Ken Scwaber*: Methodologist and software developer. Co-creator of Scrum.
- *Jeff Sutherland*: Methodologist and computer scientist. Co-creator of Scrum
- *Dave Thomas*: Software engineer. Co-authored The Pragmatic Programmer with Andrew Hunt.

Present at the convention, signatories, but not listed at authors:

- *Kent Beck*: Software engineer. Co-creator of Extreme Programming.
- *James Grenning*: Extreme Programming proponent.
- *Steve Mellor*: Computer scientist.

Of the seventeen, how many were people that actually programmed, i.e. software engineers, software developers, and software scientists?

Based on the information on the Agile Manifesto website, *fifteen out of seventeen wrote, or had written, code*!

I may be off by a couple of people, but it is pretty clear that the Agile Manifesto was, for the most part, written by people who wrote code. Like Herbert Benington, Dr. Winston Royce, and Barry Boehm before them, they understood how programmers work, because they did that kind of work themselves.

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

simplify and accelerate software development. For the last year, we have also wrapped XP with Scrum with superb results: XP enhances the quality of the software developed and Scrum enhances the day-to-day management of the projects. I call this union of Scrum and XP XBreed, which stands for "crossbreed". Find more information on this at: www.xbreed.net.

Excerpt from Mike Beedle's part of the introduction to Software Development with Scrum, written by Mike Beedle and Ken Schwaber. Published in 2002.

Most of the agile methodologies are complete, containing both management and software development practices. The exceptions are *Pragmatic Programming*, which is a collection of useful programming techniques, not a method, and *Scrum*. Scrum is not a methodology per sé, but a framework that can be used as a methodology wrapper. In the early days, Scrum was billed as a wrapper around Extreme Programming.

2001-2005: Agile Victory?

The Agile Software Development Manifesto was a landmark in the development of the Agile movement. The playing field was fairly even. If there was a leading agile methodology at the time the Agile Manifesto was written, it was Extreme Programming.

My development team adopted Extreme Programming in 2002. (I remember it as 2001, but a friend who was also on the team, insists it was 2002. His memory may be more reliable than mine.) We did not stop there though. We learned as much as we could from as many methods we could: Crystal, Pragmatic Programming, Feature-Driven Development, and Adaptive Software Development. We picked up ideas from most of them.

There were a couple of exceptions. We looked at DSDM, but it was not a good fit for the things we were doing, and the way we did them. We also looked at Scrum, but frankly, we were a bit baffled, because everything Scrum brought to the table was well covered by the other methodologies. We did not see any use for it. I missed completely that Scrum had one key advantage that would eventually make it victorious in the Agile Civil War.

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

In the years 2001-2005, the popularity of Agile methodologies exploded. With the Agile Manifesto, agile methodologies had reached, and passed, a tipping point. For us developers, the future looked very bright.

2003 was a particularly important year. That was when Mary and Tom Poppendieck published their book Lean Software Development: an Agile Toolkit. LSD (not an official acronym, but I have used it since 2003) was revolutionary. It connected agile methodologies with Lean principles, queuing theory, and Cost of Delay economics.

Earlier work on agile economics had focused on the value of practices. Lean Software Development gave us an overarching economic framework. Suddenly, it was possible to explain to a manager why agile methodologies provided an economic advantage over traditional methods. The book also opened up possibilities to tweak and improve existing methodologies. The book was all about working smarter, not harder. It was also a darn good read.

Methodology War III, ca. 2001-2010: Agile Civil War

The growing popularity of agile methodologies in the years following the publishing of the Agile Manifesto led to a demographic shift in the movement. Up until 2001, most people interested in agile methodologies were software engineers. Agile methodologies were not even on the radar of most managers. Following the Agile Manifesto, that quickly changed.

Suddenly, company after company wanted to go agile. That created a problem. What to do with all project managers, requirements engineers, and other administrative personnel that had worked in heavyweight methodology projects? There was little room for them in an Extreme Programming project, because in those, everyone, regardless of what else they did, also wrote code. In Scrum, however, there were no software development practices, but there were two administrative roles, Scrum Master, and Product Owner.

The non-programmer roles in Scrum made it possible for companies to move people from the old way of doing things to the new. Scrum also had what would become a really decisive advantage: *Certifications*!

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

“In Okinawa, belt mean no need rope to hold up pants.”

— Mr. Miyagi, in the movie Karate Kid, 1984

Ken Schwaber set up Scrum Alliance in 2002, an organization where companies could buy courses that in a mere two days transformed anyone into a Scrum *Master* (later also Product Owner). Previous experience with software development was not necessary.

Companies loved it: Send a project manager in, and two days later, you got a *master* of Scrum back, with a certification to prove it. No need to train people yourself. No need to organize for continuous training, because anyone can be a master in just two days. It’s right there in the name, so it has to be true, right?

Of course, a Scrum Master has to renew her or his license every two years, or all that mastery goes POOF! And disappears. There is also a program where you can earn Scrum Education Units (SEU) by taking courses, and doing volonteer work for the Scrum Alliance. There are also courses for Product Owners, courses in Agile Leadership, and software development courses. Details are a bit sketchy, but it looks like the developer courses focus almost exclusively on TDD and CI/CD. This is a very small subset of the basic things a developer needs to know. The content of the leadership courses are even sketchier. There are descriptions of the general areas they focus on, but nothing on the specific practices taught.

It’s a beautifully closed system, a walled garden, where people are taught Scrum and a bare minimum of everything else. It is worth noting that the agile software engineering practices taught by the Scrum Alliance are not nearly enough to fill in the gaps in the Scrum framework. If the developers know TDD (including refactoring) and CI/CD, the methodology will still be incomplete, albeit a little less so.

This setup was very appealing to the people who control the money and make the decisions in most large companies, so the Scrum business model became very successful. It became so successful that other agile methodologies were pushed out.

This was of course very annoying to proponents of other agile methodologies. Part of it was because it stung that Scrum suddenly was so much more successful than everyone else, but there was also a very real worry that with loss of large bodies of knowledge in software engineering, queuing theory, economics, and management, the agile movement would eventually wither and die.

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

Newcomers to agile soon got the idea that the agile movement began with the Agile Manifesto. Scrum proponents began rewriting history, so that other methodologies got a much less prominent role in the development of agile methodologies. They also re-branded the new version of Agile, based almost entirely on the principles in the manifesto, as Agile, with a capital A.

Here is an example of the rewriting that took place. These quotes are from a book published in 2019, but the same kind of rewrite happened much earlier:

In the late nineties, some of the most successful software developers and programmers began putting their heads together in order to come up with a brand-new approach which could encompass the needs and characteristics of the software industry. This led to the emergence of the Agile Manifesto in 2001.

— Scrum Fundamentals: A Beginner's Guide to Mastery of The Scrum Project Management Methodology, by Jeff Cohn

Note that there is no mention that Extreme Programmers took the initiative to the meeting. Nor is there any mention of which methodologies that were represented.

Cohn does mention Extreme Programming in previous passages:

And so, the nineties brought about innovations in the field of programming and software development. One of the new methods that emerged is known as "extreme programming" or XP. This led to one of the most popular versions of the Windows operating system known as "Windows XP."

— Scrum Fundamentals: A Beginner's Guide to Mastery of The Scrum Project Management Methodology, by Jeff Cohn

…and…

Extreme programming became one of the precursors to what would eventually become known as Agile.

— Scrum Fundamentals: A Beginner's Guide to Mastery of The Scrum Project Management Methodology, by Jeff Cohn

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

No mention of the other agile methodologies. No mention of the role Extreme Programming played in the creation of the manifesto. This is a radical rewrite of the historical record on the Agile Manifesto website.

The purpose of the Agile Manifesto was also retconned.

The group of 17 minds which came together to develop the Agile Manifesto was intent on building a list of ideas and concepts in a framework which could serve their own industry. Nevertheless, the Agile Manifesto wasn't just about the software industry; it was intended to have a cross-cutting appeal which could lead it to be applied to virtually any known industry.

— Scrum Fundamentals: A Beginner's Guide to Mastery of The Scrum Project Management Methodology, by Jeff Cohn

This is utter nonsense! You may recall what is written about the purpose of the manifesto on the manifesto website:

On February 11-13, 2001, at The Lodge at Snowbird ski resort in the Wasatch mountains of Utah, seventeen people met to talk, ski, relax, and try to find common ground—and of course, to eat. What emerged was the Agile 'Software Development' Manifesto. Representatives from Extreme Programming, SCRUM, DSDM, Adaptive Software Development, Crystal, Feature-Driven Development, Pragmatic Programming, and others sympathetic to the need for an alternative to documentation driven, heavyweight software development processes convened.

Now, a bigger gathering of organizational anarchists would be hard to find, so what emerged from this meeting was symbolic—a Manifesto for Agile Software Development—signed by all participants.

— History: The Agile Manifesto page at the Agile Manifesto website

To boil it down a bit:

- The manifesto was written by representatives for several agile methodologies: Extreme Programming, SCRUM, DSDM, Adaptive Software Development, Crystal, Feature-Driven Development, and Pragmatic Programming. The methodologies

came first! The manifesto was extracted from the ideas in the methodologies, and represented the common ground of the methodologies, nothing more.

- The manifesto is *symbolic*! It is *not* a "list of ideas and concepts in a framework".
- The manifesto is a manifesto for agile *software* development! It was *not* " intended to have a cross-cutting appeal which could lead it to be applied to virtually any known industry".

Why rewrite history in this way?

- It reduces interest in other methodologies than Scrum.
- It makes it possible to sell Agile everywhere, and so, expands the business.today, we have Agile HR, Agile Finance, Agile, Agile Healthcare, etc.

Principles are, to a certain extent, transferable between different areas. Methodologies, on the other hand, are not transferable in the same manner. Methodologies are also harder to create, because you must understand *how* to implement the principles in a specific area of endeavor. That cannot be done without domain specific knowledge.

It is also relatively easy to create the *illusion* that a principle is understood. For example:

Responding to change over following a plan

A coach like me can talk about the importance of responding to change over following a plan to just about any audience. The catch is that implementations are very different in different domains. If you do not know how to translate the principle into methods, the principle will remain a fond wish. Some examples of how the principle can be translated:

- *Emergent design* in software development requires knowledge about how to create loosely coupled designs. A methodology has methods for this, like:
    - Iterative development (Not incremental development! Big difference. Scrum and SAFe have shifted the focus from iterative to incremental. Probably because iterative development requires higher levels of software engineering skills.)
    - Test-Driven Development(TDD)
    - Design Patterns
    - Continuous Integration/Continuous Deployment
    - Domain-Driven Design

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

- In the military you might find:
    - Commander's Intent
    - Reversed Command Chains
    - Generic rules about what to do when the plan fails, i.e. "take the high ground…"
- In the automotive industry
    - Single Minute Exchange of Die (SMED) made it possible to respond to different kinds of orders by producing small quantities of different cars on the same production line. This gave Toyota an economic advantage over all other car manufacturers in the world after WWII, and helped pave the way for Toyota's success.

What Agile, with a capital "A", did was that it raised a set of symbolic statements to the status of principles, and then focused on packaging and selling them. It moved away from methodologies and practical implementation of the ideas.

The software engineering based methodologies could not compete. Learning them required both time and effort. Scrum and Agile promised relatively quick and easy fixes.

2008: Agile in Decline

In 2008(-ish) I held a presentation at an Agile convention in Malmö, Sweden. I began by talking about the astonishing results you could get with agile methodologies: 5-6 times faster development. Every time I presented an example the audience cheered and clapped.

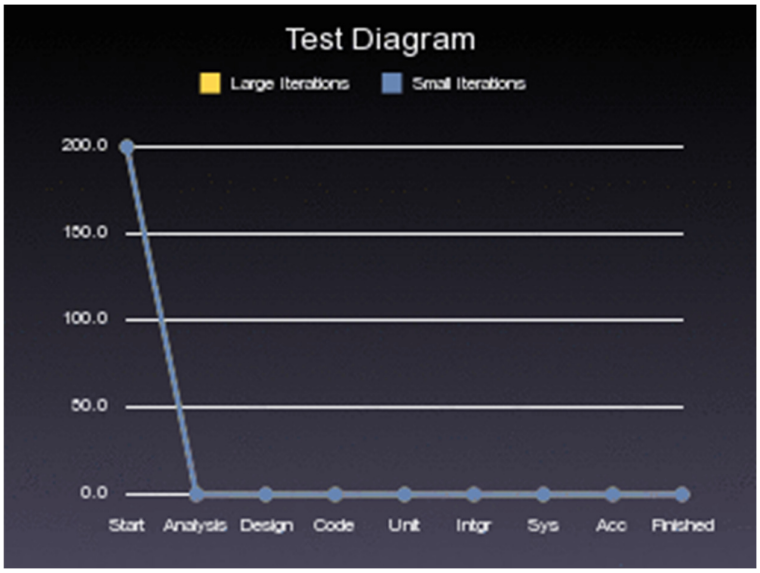
Then I asked: "How do you think that can be? The developers can't type 5-6 times faster. They can't think 5-6 times faster. How can they deliver 5-6 times faster?"

The audience went dead quiet. Nobody knew. In a room full of Agile practitioners and experts, it turned out nobody knew how Agile worked. Nor did they know if it really worked for themselves, just that studies said it had worked for someone else.

I am pretty sure lots of folk there could quote the Agile principles from the Manifesto though.

In this particular case, if you know a little bit about queuing theory, Lean, or Theory of Constraints (TOC), the question is easy to answer.

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html



In this animation the yellow and the blue projects do exactly the same things, and work equally fast, but the blue project finishes in half the time. Why?

The little animation above shows you why agile projects finish first: It shows work flowing from start to finish, in two projects, Blue and Yellow. The projects start at the same time, and do exactly the same things. At every stage in the process, both teams work at exactly the same velocity.

Despite all these things being equal, team Blue finishes in half the time of team Yellow. The one difference, is the length of the iterations. Team Blue has iterations half the length of team Yellow's iterations. Because of this, team Blue has twice as many iterations as team Yellow. Pieces of work is transferred much more often, wait times are cut in half, and the work is finished faster.

There are a number of ways to limit work in the process to achieve this effect: Iteration duration, kanban (as in the original Lean method), TOCs Drum-Buffer-Rope, CONWIP. They are all a little different, and which one you choose depends on what works best in your situation.

…except that since we do not teach those things in Agile anymore, it is a bit optimistic to believe you will get that kind of effect. For example, I fairly often see large "Agile" projects that follow the rules of iteration lengths and WIP limits in Scrum or Kanban, but get zero effect, because nobody knows that limiting WIP for each team in a project is not the same as limiting WIP for the project as a whole.

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

By 2010 the Agile Civil War was over. Agile won. Agile software development methodologies lost. Much of the original body of knowledge was discarded.

Waterfall - The Slowest There Is!

We are not quite done with the animation above yet. It shows one of the many reasons why Waterfall projects are such a bad idea:

What is the absolutely slowest way for a team to finish the project?

It is to have only one iteration. (Team Yellow, the slow team, has four iterations in the animation. Imagine them having only one iteration. They will be four times slower than they currently are, and eight times slower than team Blue.)

How many iterations does a Waterfall project have?

Only one!

That means, it is theoretically impossible for Waterfall projects to be as fast as Agile projects.

Waterfall is also slower than any methodology that supports parallelization of work, like the Critical Path Method (CPM), the Spiral Method, and RAD. Waterfall is slower than anything allowed in the MIL-STD-498 US Department of Defense standard from 1994. Waterfall is slower than the method the SAGE project *actually* used in 1953-1956.

This also means Waterfall is much more expensive than any other software development method in the world.

In the 2009 book Flow: The Principles of Product Development Flow, Donald Reinertsen lists other problems with having to much work in process:

- Increased variability
- Increased risk
- Increased cycle time
- Decreased efficiency
- Decreased quality
- Decreased motivation

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

Advantages? None! It's all bad.

The Worst of Two Worlds: Agile + Waterfall

Waterfall is the worst thing that came out of traditional methodologies. Agile, with a capital "A", is the worst thing that came out of the agile movement.

Combining two bad things is unlikely to produce good results. There are also number of ways that Waterfall and Agile come into direct conflict. Let's break it down:

- Waterfall is optimized for machine code programming using punch cards, with extremely high cost of change. Agile is optimized for developers working with object-oriented languages at a interactive terminals, with very fast feedback, and low cost of change. (The original agile methodologies were designed to actively reduce the cost of change, while Agile more or less makes the assumption that cost of change is low. This is one of the reasons why Agile is in trouble.)
- Waterfall is a push process. Agile relies on pull processes. Combining push and pull processes does not work very well. You are likely to see bloody wars between people in the push parts of the process and people in the pull parts.
- Waterfall uses a single iteration, and maximizes Work-In-Process. Agile uses short iterations and strives to minimize Work-In-Process.
- Waterfall strives to eliminate feedback loops, so you can move people out of a project as soon as the phase they are working on is finished. Agile has many complete iterations, with lots of feedback loops, so you need to keep teams as stable as possible over the duration of a project.
- Waterfall locks requirements as early as possible. Agile uses emergent design to respond to requirements changes. (Well, Agile does that when done right. Not if it has degraded too much.)
- Waterfall uses functional teams. Agile uses cross-functional teams.
- Waterfall breaks work down into functional units of work. Agile breaks work down into vertically sliced units of work. If you use Waterfall style breakdown, you can't do small frequent deliveries, and Agile project economics goes down the drain. If you do Agile style breakdown, you can't maximize resource utilization, and Waterfall style project economics goes down the drain.
- Both Waterfall and Agile have dysfunctional ideas about how to plan large projects. They fail to take statistical variation and probability distribution of work package duration into

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

account. (Neither Waterfallers, nor Agilists like to do math, so for the most part, they ignore it.)

- Waterfall is based on Theory X management. Agile is based on Theory Y management.
- Waterfall uses Cost Accounting. Agile projects (when run correctly) use Lean or Throughput Accounting, rolling budgets, and Profit & Loss models.

…and so on.

You can't just switch out one of the things above, without switching the others. They are connected. Another problem is that, while there are plenty of things wrong with Agile, no matter what part of Waterfall you swap in, the situation gets even worse.

That does not mean we are helpless though. There are lots of things that can be done. There are also several large bodies of knowledge that can be helpful.

What to do instead: The Best of Agile + the Best of Traditional Methods + New Solutions

"Thanks! Though, now I have to think again."
— Henrik Mårtensson (Yep! That's me!) after statistician Fredrik Johansson sent Henrik a paper on Monte Carlo simulation.

If we seriously want to fix the current problems in software development, rather than desperately flailing about grasping for straws, maybe, just maybe, we should try to understand what the problems are, and then start experimenting with solutions, learning as we go.

I am not going to give you a recipe for solving your problems. While the problems with Agile usually are similar, the causes may be different. The degrees of freedom in finding solutions are also different. Thus, while you can get a lot of inspiration from what others have done, copying it right off is unlikely to work.

I will however give you a few examples that can work in some situations, for some organizations. It is *not* a To Do list! Use it for inspiration.

Learn from the past, but don't copy it!

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

If we do not learn from the past, we are doomed to repeat old mistakes. If we do learn from the past, we get a better understanding of our current situation, and the trajectory we are on.

The entire Agile+Waterfall hybrid discussion exists because we have trouble learning from the past. It's proponents can't distinguish between Waterfall and traditional methods, for one thing. That makes it difficult to pick up the useful things that can be found in traditional methods. It also makes it easy to bring in the really bad parts of Waterfall. (Which would be just about any part of it, in case you are still wondering.)

The history of Waterfall has been completely rewritten. What was a series of misunderstandings and failures, have been retconned into a story about valuable treasure buried easily accessible just below the surface. The history of traditional, heavyweight methodologies is entirely ignored at worst, and confused with Waterfall at best. Agilists have similar problems with memory loss and the creation of an alternate history.

My advice is, do not rely on authorities, not even me! Go to the original sources and check for yourself.

I have provided plenty of links to source material in this article, but I recommend that in addition to checking them out, you should also do your own searches. You may, probably will, turn up important information that I missed. Just be very cautious about accounts of events that are written long after the event itself, by people who were not there.

Personally, in most cases, I do not even bother with articles that do not link to their source material. I do read the source material when links are available, and as you have noted, I often find that the source material says something completely different from what the article author claims.

Training

Early agile methodologies were simpler than traditional methods, but you still had to learn a lot in order to use them. For highly motivated managers and developers, this is not a problem. Learning stuff is fun! Seeking out new knowledge is fun! Practicing is fun!

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

If you are not a total nerd, or if you are a nerd who also wants a life when you are not working, you may need a bit of help. I am not a big fan of pre-packaged courses. They can be a good way to get started, but they do have a tendency to be a bit superficial, and their source material is often a bit questionable.

If you can, build your own learning program, and your own learning organization. Do not be too narrowly focused in what you learn. Great insights often come from combining different areas of knowledge. Here are some areas you might want to look into, in random order:

- Software engineering
- Systems Thinking
- Complexity Science
- Queuing Theory
- Statistics
- Economics
  - Profit & Loss statements
  - Cost of Delay
  - Lean and Throughput Accounting
- The Deming Knowledge System
- Lean
- Theory of Constraints
- Agile methodologies (Start with Lean Software Development)
- Traditional methodologies (Read up on queuing theory first)
- Photography (You wont get better at software development, but it is a nice hobby. There are other nice hobbies you can use to decompress, or so I've heard.)

Some companies encourage employees to use four hours (may vary) each week to learn and practice. If so, make the best of it. Build an informal network of learners. If necessary, build a formal network of learners. Personally, I have found that informal networks work best. They are usually broader in scope, and more enthusiast-driven.

Forget the movements, manifestos, and fads. Just figure out how to do the work!

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

Theory is highly useful, but movements, manifestos, and fads tend to be superficial and have less value than it initially looks. Focus on things that interest you, and things that actually help you get the work done.

If you join a movement, do not get so caught up in it that you forget to keep track of other points of view. Don't get caught up in cults.

When you find something that looks interesting, don't just go by information from the enthusiasts, check out what the critics say too. It may save you from wasting years on stuff that does not work.

By the way, Scrum Theory, is not theory. It is a set of hypotheses. Be aware of the difference.

Actively reduce the cost of change

To handle changing requirements, you need software that is easy, and cheap, to change. Your main tools to accomplish that, will probably be a set of software engineering tools.

Have a look at Extreme Programming, the Crystal family of methodologies, S.O.L.I.D. by Robert Martin, Pragmatic Programming, Domain-Driven Design, and Design Patterns to get started, but don't stop there.

Parallelization

To plan a large project well, you need to be able to plan parallel activities. Start by having a look at Critical Path Method (CPM), Critical Chain from TOC, and Blitz Planning from the Crystal methodologies.

It will pay off to also know a bit of queuing theory (WIP control, transfer batches, etc.). You need to know enough statistics to be able to figure out whether your time estimates work or not. If they don't, you will need Monte Carlo simulation and aging analysis, and enough statistics to figure out if *they* work.

Monte Carlo Simulation

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

Scatterplots showing why neither Agile nor Waterfall estimation methods work.

Monte Carlo simulation is a way to make a *prognosis* based on previous data. Use when estimates are uncertain. (My experience is that is pretty much all the time).

Measure the *correlation coefficient* between estimates and actual duration in order to figure out whether your estimates work. You can also use *scatter plots*, as in the figure above.

Did I mention you need to know a little bit about of statistics to plan a large project effectively? If you are a manager, understanding it on a conceptual level may be good enough, if you also hire a project statistician.

Aging analysis

Aging analysis is a good alternative when you have no clue what the business value and/or duration of work packages is.

Use Cases

User stories are intended for small projects with on site users. For large projects, or when users will not be readily available. Look at the stuff Alistair Cockburn and Ivar Jacobson are working on.

Dependency Jar

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

A very simple tool for identifying what kind of dependencies a team has. This one is my little invention.

Every time a developer is delayed by something, they write a note about it and drop it in the jar. If the delay is more than a day, they write a note each day.

Empty the jar at the end of each iteration, organize the notes in classes, and try to fix the most serious causes of delay.

Works for short periods of time, until the developers get tired of it, and you. Still, it can be very helpful. My experience is that you can identify about four times more dependencies than you can at planning sessions, like SAFe's PI planning. Mileage varies though.

Parallel experiments

When there is something you do not know, device multiple experiments, and run them in parallel. Stop failing experiments as early as possible. If you have more than one experiment that succeeds, see if you can combine them to get an even better solution.

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

Prototyping

Build prototypes to test ideas, and see whether they work. A prototype may be very simple, a paper prototype, perhaps built using CRC cards. It may also be a functioning miniature version of the thing you are building. Focus on the things you are most uncertain about.

Beware disinformation: The HBR article and the London Crossrail Project

Be very careful about which sources you believe. Whenever possible, go to the original sources, and look for yourself!

For example, the Harvard Business Journal article that got me started writing this article, is utterly confused about nearly everything. The author refers to several building projects, claims that they used Waterfall, and were very successful. Then he claims that this is proof Waterfall works great for software projects. There are several things wrong with this:

- Software development projects and building projects have very different cost of change. The correlation between estimates and actual implementation times is likely to be different. Task duration distribution probabilities are different. All of this means you need to adapt the methodology you use for the context you use it in. You can't just pick a methodology used in a building project, start using it for a software development project, and expect it to work.
- Building projects do not necessarily use Waterfall! Building projects use very effective parallelization of tasks, prototyping, WIP control, databases with task duration times, decoupling of workflows… Some building projects adapt Lean processes.
- I checked one of the projects the article refers to, The London Crossrail project, and it is not even close to using Waterfall. Nor was it a great success.
  - Project planning began in 1974. The building project started 2008, and finished in 2019. In other words, 34 years of planning, 11 years of execution, for a total of 45 years. This is *not* a speed record! For comparison, the Empire State Building took about eight months to build in 1930-1931, including planning time. Planning of the Empire State Building was done in parallel with building it.

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html

- The goal was to facilitate London's "continued development as a world city and sustain its position as the financial centre of Europe". (See The Execution Strategy for Delivering London's Elisabeth Line, page 2.) Brexit was in 2020, so that goal fell apart soon after the project finished in 2019.

If you want to know how good building projects really work, and how the *Empire State Building* was built in about eight months, I suggest you watch Mary Poppendieck's famous presentation The Tyranny of the Plan. You can also read a transcript of the presentation, if you prefer that.

Finally, let's have a closer look at the project I mentioned, the *London Crossrail Project*.

Before anyone started building anything, a pre-study, the London West-End study was made. The study identified three major delivery packages:

- Paddington-Liverpool Street
- Wimbledon-Liverpool Street
- Wimbledon-Hackney

The people doing the prestudy created an economic model of the three delivery packages, and calculated Net Present Value for each alternative. The model included cost of delay, and effects of capital cost increases, and six other criteria. They also assessed regional metros and regional express. This allowed the project to prioritize according to business value. Then they made their recommendations.

> *In the light of the assessment it is our recommendation that the Paddington to Liverpool Street Regional Metro should progress to the project definition stage and should form the backbone of the 20 year programme.*
> *— The London West-End Study, page 17*

In other words, they used an *iterative* approach with *multiple partial deliveries*, and prioritized them according to *business value*. That is a lot more like Agile than it is like Waterfall.

To top it off, the project was designed to allow for changing requirements:

> *A scheme exists for the extension of the existing Heathrow Express line from the central area of the Airport, under the site of the proposed Terminal 5 to join up with the*

South West network in the Staines area. It is envisaged that this would be constructed as an adjunct to the Terminal 5 works and would only proceed if consent for Terminal 5 is forthcoming.

— The London West-End Study, page 18

That is *emergent design*! The architecture is extended *if* the requirements change.

Summing up, the London Crossrail project was a Waterfall+Agile hybrid, kind of. The 34 years of up front planning was what you can expect from a very large Waterfall project. Once construction started, you had a far more agile project:

- iterations
- multiple deliveries
- prioritisation according to business value
- emergent design in response to changing requirements

I have no idea why the article author used the London Crossrail project as an example of a successful Waterfall project, when it clearly was not.

I can understand how someone that is not a methodologist gets Waterfall and traditional methodologies mixed up, but this is the author of the *Harvard Business Review Project Management Handbook*, a visiting professor in seven business schools. He ought to be able to distinguish between Waterfall, traditional methodologies, Agile, and agile software development methodologies. He also ought to know that starting the project with 34 years of planning does not make it a good template for how to run a successful software project. It's not really a good way to start a successful building project either.

There is always the possibility that I got it wrong, and he knows something I don't, but in this case I doubt it. There are gaps in the historical record from 1956 to the present, but the key papers, and writings of people who were present at the time, are available, and what they say is pretty clear:

Waterfall sucks!

Source: https://kallokain.blogspot.com/2023/11/waterfall-vs-agile-battle-of-dunces-or.html