Project Management History

A Brief History of Agile

Introduction

The history of applying agile approaches to software development appear to be as old as the development of large software systems. This article will trace the evolution of the techniques used to develop software programs from SAGE, the first major system, through to the present with a focus on management methodologies.

To achieve this objective, this article will briefly consider the evolution of computer hardware, operating systems, and programming languages as they affect and enable the management processes. However, we have made no attempt to fully document these aspects (there are many other authoritative publications covering these topics).

There is also a need to limit the scope of 'agile' considered in this article and its counterpart loosely described as 'waterfall'. The Agile Manifesto covers a broad sweep including stakeholder (customer) engagement, the desirability of change during the project, and the way people are managed. Most of these concepts are simply good management, and conversely not taking these factors into consideration is bad management!

For example, it is just as easy to micro-manage a scrum team as it is to micro-manage a pre-planned (waterfall) project; micro-management is bad management regardless of the development approach used. Similarly, adapting or adopting beneficial change is desirable, but both the context of the development and the development approach used can limit the benefits of change. So, accepting there is a lot of bad management around, and agility is a generally desirable characteristics, these aspects are only considered in the body of this article when necessary.

The aspect of agile and system development this article focuses on is the use of iterative and/or incremental development processes versus a single pass, pre-planned development process, and the factors leading to the reintroduction of a single pass, pre-planned development process in the form of waterfall in the mid to late-1970s.

Foundations of the Computer Industry – 1940s and Before

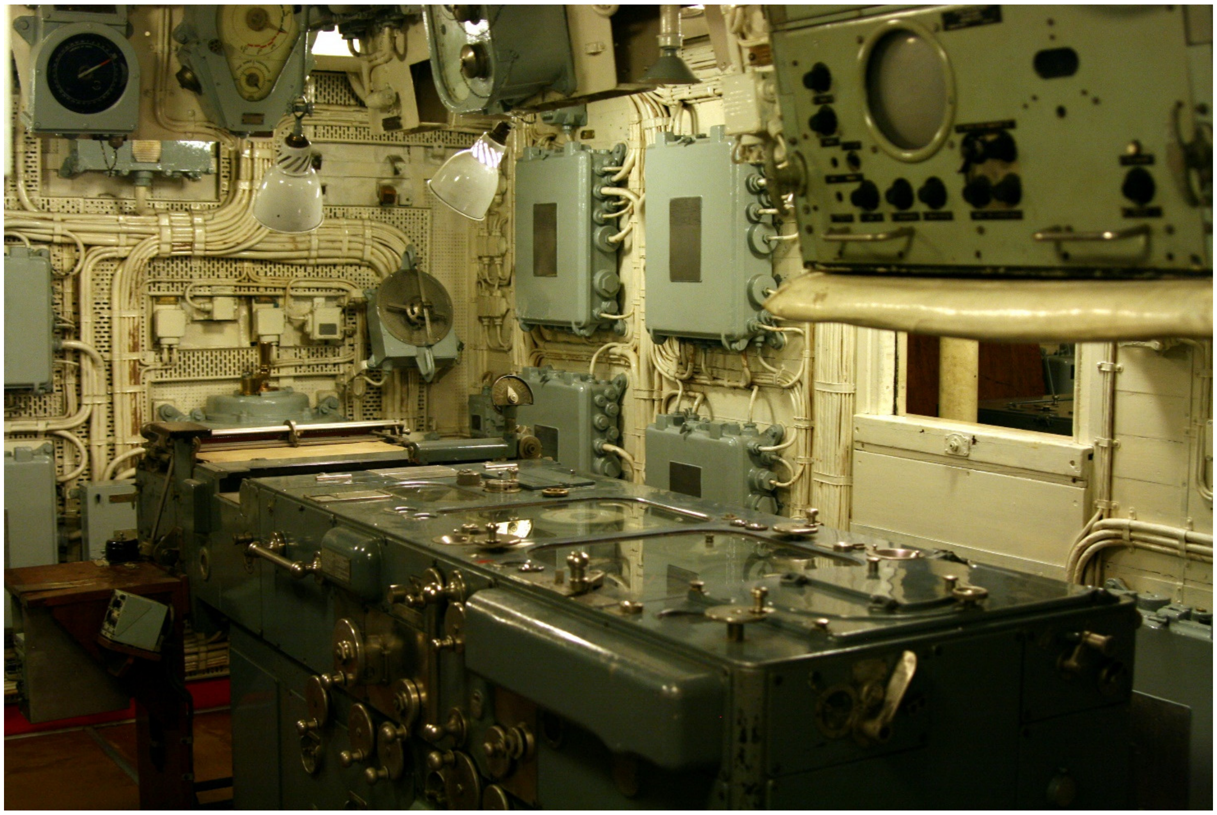
Using a relatively sophisticated mechanical device to produce or calculate a result has 1000s of years of history. One of the most notable from antiquity is the Antikythera Mechanism[1], the world's oldest existing analogue computer, created by the Ancient Greeks around 250 BCE to provide a 'ready reckoner' showing the Greek zodiac and an Egyptian calendar, information about lunar cycles and eclipses, and the movement of the five known planets.

Later, during the Industrial Revolution (1801), the Jacquard loom used punched cards to hold complex data. It was a mechanical loom that simplified the process of manufacturing textiles with complex patterns. The

[1] For more on the Antikythera Mechanism see: https://en.wikipedia.org/wiki/Antikythera_mechanism

loom was controlled by a replaceable chain of cards, a number of punched cards laced together into a continuous sequence. Multiple rows of holes were punched on each card, with one complete card corresponding to one row of the design. Charles Babbage knew of Jacquard looms and planned to use cards to store programs in his Analytical engine (1837).

By the 1940, the use of machines to calculate solutions to complex problems was evolving rapidly. One of the best examples was the fire control table found in most major warships. This was a true mechanical, analogue computer, processing multiple inputs to achieve a firing solution. Inputs included the speed and direction of the ship, the estimated speed and direction of the target, estimated range to the target (often greater than 20 miles / 30 Km), barometric pressure, air temperature, wind direction, and the curvature and relative speed of rotation of the earth in both locations. The output was the angle of elevation and direction of the guns to fire shells that would arrive a couple of minutes later within a few meters of the position the target was expected to be in (at speeds in excess of 30 miles per hour this would not be where the ship was at the time of firing).



HMS Belfast – Admiralty Fire Control Table

Designing and building this type of equipment and the manufacture of all of its component parts required a complete and highly detailed design to be 100% correct before the cutting, machining, and welding could start. There was only one chance to get the final assembly correct. Prototyping and carrying forward learned experience would be essential but the manufacture of a table for installation in a ship was a case of do it once and do it right. Over the course of the 1940s this type of equipment transitioned from pure mechanical, to electro/mechanical and then to full electrical systems in the 1950s.

The next area of development was the building of computers to fulfil a specific role. The example below is the famous Bombe built to decode the German Enigma cypher used during WW2. The Bombe was an electro/mechanical computer designed to mimic the workings of the Enigma machines and rapidly test

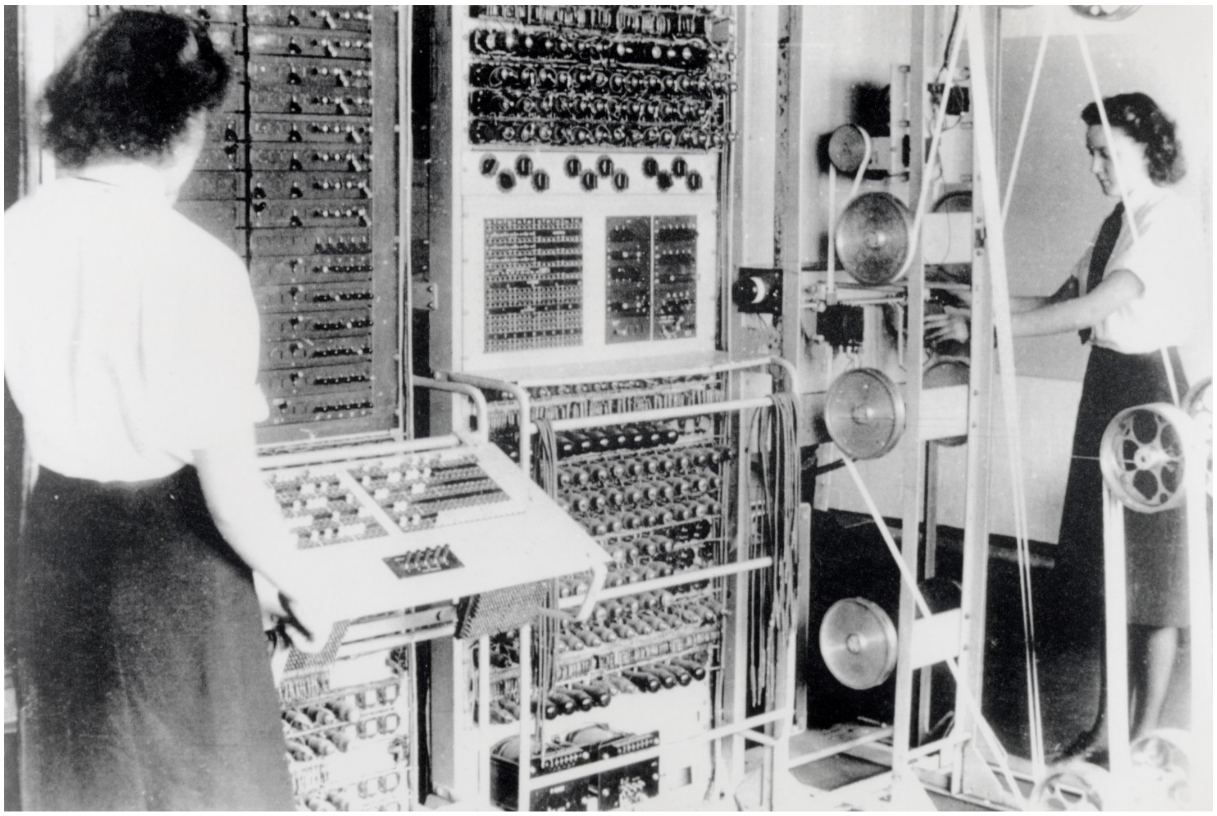
1000s of combinations to find which settings out of the millions of possibilities were being used on a particular day. When the Germans changed the physical structure of their Enigma machines, the Bombe had to be rebuilt. Similar machines were built for various military and civilian purposes. But again, the requirement for a complete design before building the machine was essential.

Replica of the Bletchley Park Bombe Computer

This type of early computer was complex electrical/mechanical engineering device that had to be built right to function correctly, meaning detailed design was needed before work on building the computer could start. Bug fixing to correct errors required physically changing the machine.

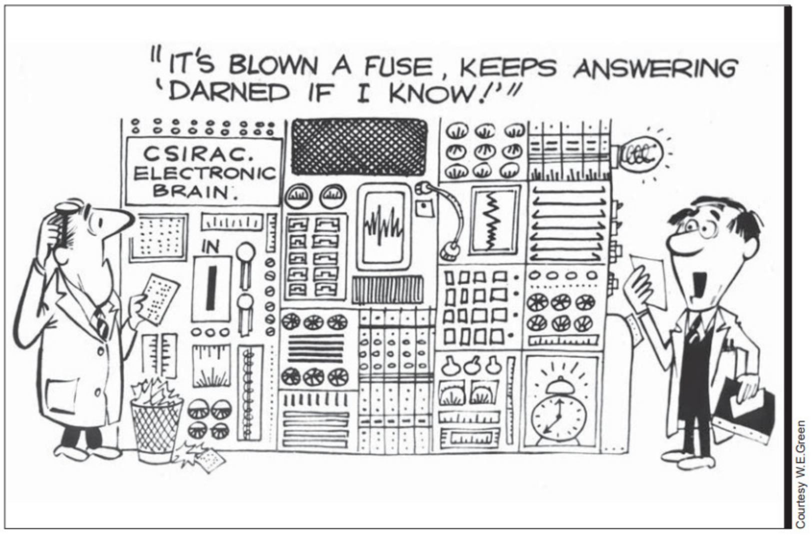
Early Programable Computers – 1950s

The development of programmable computers also began in the 1940s at Bletchley Park. The Colossus computer was built to help in the cryptanalysis of the German Lorenz cypher. It used thermionic valves (vacuum tubes) to perform Boolean and counting operations. Colossus is arguably the world's first programmable, electronic, computer, although it was programmed by physically changing switches and plugs, not by a stored program. Again, because of the physical nature of the machine, detailed design before building of the computer was essential. These developments were closely guarded secrets until the 1970s, but the technology was shared with the USA, and helped kick-start the modern computer industry.



Part of the Colossus Computer

John Mauchly and J. Presper Eckert of the University of Pennsylvania are credited with designing the Electronic Numerical Integrator And Computer, or ENIAC, the world's first general purpose electronic digital computer, completed in 1945. ENIAC was also programmed through a physical system of adjusting switches and cables manually[2].


"IT'S BLOWN A FUSE, KEEPS ANSWERING 'DARNED IF I KNOW!'"
CSIRAC. ELECTRONIC BRAIN.
IN
Courtesy W.E.Green

Australia's CSIRAC was the fifth electronic stored program computer in the world, and ran its first test program late in November 1949[3].

Computers that could be programmed by running software required improvements in memory. This started to be available in the form of magnetic-core memory in

[2] For more on ENIAC see ***ENIAC and the Origins of Computers***: https://mosaicprojects.wordpress.com/2024/05/19/eniac-and-the-origins-of-computers/ and ***Computers before ENIAC***: https://mosaicprojects.wordpress.com/2024/05/23/computers-before-eniac/

[3] For more on CSIRAC, and computers of this generation see ***The Last of the First, CSIRAC: Australia's First Computer***: https://mosaicprojects.com.au/PDF-Gen/CSIRAC_the-last-of-the-first.pdf

1950 when the United States government received the UNIVAC (Universal Automatic Computer) 1101 or ERA 1101. This computer is considered to be the first computer that was capable of storing and running a program from memory. As with everything associated with computers in the 1950s and 60s, both processing power and memory improved rapidly, making the concept of computer programming viable as a separate discipline to designing and building the computer hardware.

The Development of Programming Languages

The earliest computer programs were written in either machine code (specific to the computer), or Assembly language which was a type of computer programming language that was easier to use and could then be compiled into machine code language to run.

One example was Autocode, published in September 1952 by Alick Glennie, a student at the University of Manchester, England. This was the first of several programs called Autocode written for the Manchester Mark I computer. Autocode was in two parts, the programming language, and the compiler used to convert the program into machine language[4].

Through to the mid-1950s, developments were fragmented and scattered. The first general programming language was developed in 1957 by John Backus, when he created *FORTRAN*, which is a computer programming language for working with scientific, mathematical, and statistical projects. This was quickly followed by Algol in 1958. *Algol* was a precursor to programming languages such as Java and C. Then *COBOL* was created by Dr. Grace Murray Hopper in 1959 as a language that could operate on all types of computers.

Prior to the development of *FORTRAN* and *COBAL*, most software developments were undertaken by small teams led by a mathematician or scientist. For example, the team that developed the CPM software for DuPont between 1956 and 1958[5] consisted of:

1 Mathematician (James Kelley) who developed the model,

1 Technical Supervisor (John W. Mauchly), and

5 programmers (not full time: three worked on the initial UNIVAC1 version, two rewrote the software for the Univac 1103A/1105 computer in *Unicode* an Assembly/compiler language).

It is also interesting to note, the development of the CPM software went through three main phases and was incremental.

After 1960, the rapidly increasing power of the computers, and the availability of general computer languages changed computer programming from a bespoke cottage industry led by scientist and engineers into a core capability. Ever larger, more complex, programs were envisaged and developed, creating the management challenges this article is focused on.

Two developments in particular enabled the use of iterative and incremental development. The first was the concept of Object Oriented Programming (OOP) which evolved through the 1960s. The second and

[4] From 1943 to 1945 Konrad Zuse designed Plankalkül, the first high-level programming language, however, this seems to have had very little influence.

[5] For a firsthand account of this project, see ***The Origins of CPM, A Personal History by James E. Kelley and Morgan R. Walker***, pmNetwork, Feb 1989: https://mosaicprojects.com.au/PDF-Gen/Kelley+Walker-PMN-1989.pdf

more important was the public release of Smalltalk 80 in 1981 (the Smalltalk program had been evolving for a decade within PARC[6]). Smalltalk is a purely object oriented programming language.

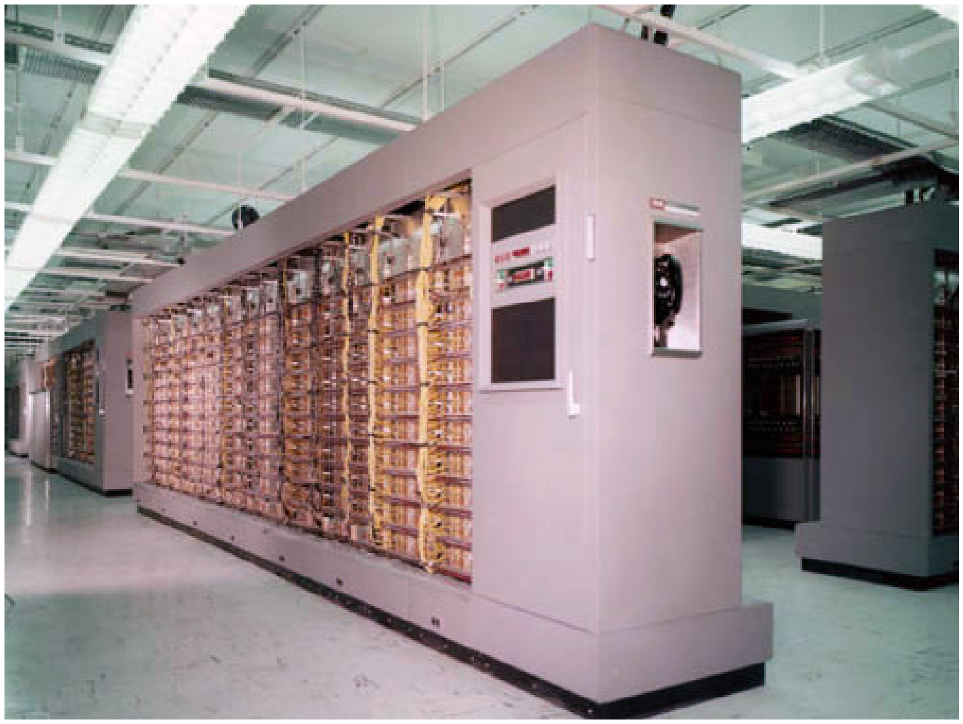
These developments were initially on mainframe computers, Mini-computers stared to appear in the mid-1960s and what most people consider computers these days, the Micro-computer (IBM PC, Apple, etc.) were not developed until the mid-1980s.

The sequence of agile programming methods is included in the ***Flavors of Agile – 1970s to 2020s*** section of this article (page 16).

Managing Complex Software Developments

SAGE

The *Semi-Automated Ground Environment* (SAGE) involved a system of networked computers used to coordinate data from many radar stations, and collate it into one unified picture. The project started in 1953, and went operational in the late 1950s as part of the NORAD (North American Air Defense Command) early warning infrastructure. SAGE used a centralized computer built by IBM known as the AN/FSQ-7 Combat Direction Central, it weighed about 250 tons, had 60,000 vacuum tubes, required 3 MW of energy to run, and could execute 75,000 instructions per second.



Part of one of the SAGE computer rooms, USAF

Developing the SAGE software was a huge undertaking that ran concurrent to, but independent of the manufacturing processes. The scale of the challenge made it the first software development project large enough to require a software development methodology; so the engineers working on SAGE created one. The

6 PARC: Xerox Palo Alto Research Center

methodology used for SAGE, and opportunities for improvement, are described in a 1956 presentation given by Herbert D. Benington: ***Production of Large Computer Programs***[7].

The SAGE program was written by making holes in punch cards that could be read by the computer as machine language. This made checking and testing the code difficult. The approaches used to minimize issues included prototyping, as well as detailed design and extensive testing.

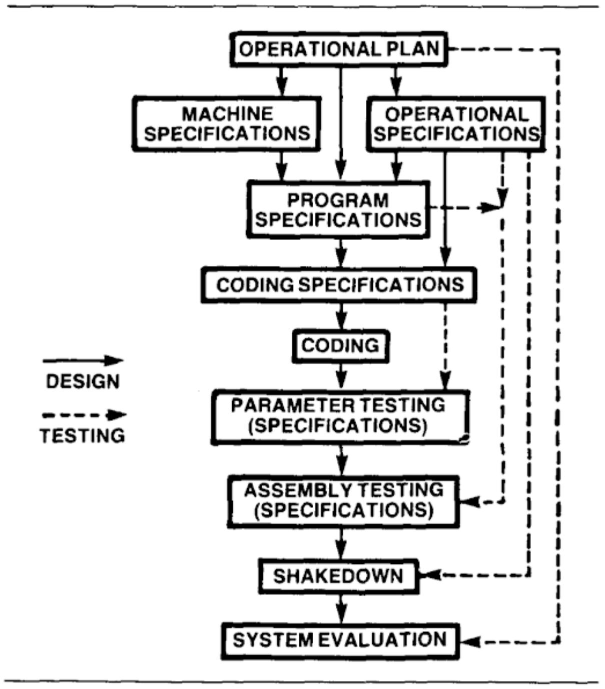


Figure 4. Program production. Production of a large-program system proceeds from a general operational plan through system evaluation; for example, assembly testing verifies operational and program specifications.

But to quote Benington *"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* [the prototype] *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. 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."*

While Figure 4 from Benington's paper may look like a waterfall approach; this flow was applied to each subprogram and was in large part a consequence of the programming language and use of punch cards.

In the foreword to his 1983 paper, Benington wished the SAGE team could have used an evolving approach, based on prototyping and testing, in other words, an *emergent design*! Which of course is a similar concept to those applied in Extreme Programming and some other agile methodologies. Benington's own estimate is that iterative development would have reduced the cost of the SAGE project by 50%. But, as he also noted, software development tools and processes had vastly improved since the late 1950s[8].

Project Mercury –1958 to 1963

Project Mercury started in 1958[9]. The software team had their own computer and the new Share Operating System, whose symbolic modification and assembly allowed them to build the system incrementally, which they did, with great success. They ran with very short (half-day) iterations that were time boxed. The development team conducted a technical review of all changes, and applied the Extreme Programming practice of test-first development, planning and writing tests before each micro-increment. They also practiced top-down

[7] Download the 1983 paper adapted from the 1956 presentation ***Production of Large Computer Programs*** from: https://mosaicprojects.com.au/PDF-Gen/Benington_-_Production_of_Large_Computer_Programs.pdf

[8] A detailed discussion on SAGE and Benington's paper is included in the Blog ***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

[9] ***Project Mercury*** was the first human spaceflight program of the United States: https://en.wikipedia.org/wiki/Project_Mercury

development with stubs. Project Mercury was the seed bed out of which the IBM Federal Systems Division grew based on a history and tradition of incremental development[10].

Winston Royce

The next significant paper, discussing his experience developing major software programs for satellites, is a 1970 whitepaper by Dr. Winston Royce, ***Managing the Development of Large Software Systems***[11]. While Royce did introduce the concept of waterfall in this paper. What many commentators leave out is what he wrote about the waterfall approach:

…the [waterfall] *implementation described above is risky and invites failure.*

… The testing phase which occurs at the end of the development cycle is the first event for which timing, storage, input/output transfers, etc., are experienced.

… one can expect up to a 100-percent overrun in schedule and/or costs.

While Dr. Royce did believe in breaking a project down into a linear sequence of phases, he understood very well that feedback and making corrections iteratively are necessary[12]. His summary of his preferred model is:

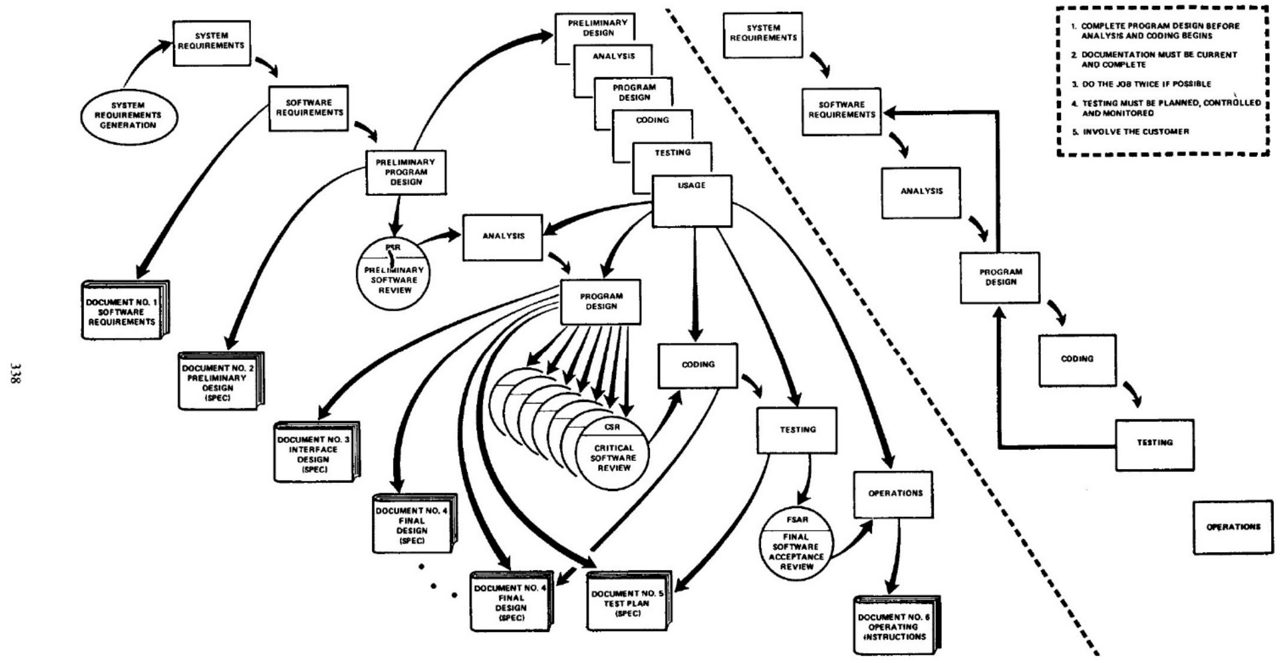

338
SYSTEM REQUIREMENTS GENERATION
SYSTEM REQUIREMENTS
SOFTWARE REQUIREMENTS
PRELIMINARY PROGRAM DESIGN
PSR PRELIMINARY SOFTWARE REVIEW
ANALYSIS
PROGRAM DESIGN
DOCUMENT NO. 1 SOFTWARE REQUIREMENTS
DOCUMENT NO. 2 PRELIMINARY DESIGN (SPEC)
DOCUMENT NO. 3 INTERFACE DESIGN (SPEC)
DOCUMENT NO. 4 FINAL DESIGN (SPEC)
DOCUMENT NO. 4 FINAL DESIGN (SPEC)
DOCUMENT NO. 5 TEST PLAN (SPEC)
DOCUMENT NO. 6 OPERATING INSTRUCTIONS
PRELIMINARY DESIGN
ANALYSIS
PROGRAM DESIGN
CODING
TESTING
USAGE
CSR CRITICAL SOFTWARE REVIEW
CODING
TESTING
FSAR FINAL SOFTWARE ACCEPTANCE REVIEW
OPERATIONS
SYSTEM REQUIREMENTS
SOFTWARE REQUIREMENTS
ANALYSIS
PROGRAM DESIGN
CODING
TESTING
OPERATIONS
1. COMPLETE PROGRAM DESIGN BEFORE ANALYSIS AND CODING BEGINS
2. DOCUMENTATION MUST BE CURRENT AND COMPLETE
3. DO THE JOB TWICE IF POSSIBLE
4. TESTING MUST BE PLANNED, CONTROLLED AND MONITORED
5. INVOLVE THE CUSTOMER

Figure 10. Summary

[10] A detailed history of IID is contained in ***Iterative and Incremental Development: A Brief History***: https://mosaicprojects.com.au/PDF-Gen/History_of_Iterative_and_Incremental_Development.pdf

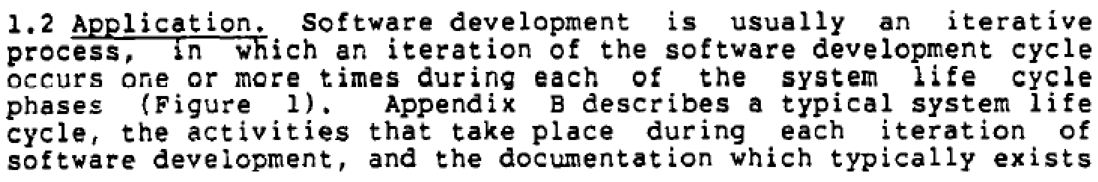
[11] Download ***Managing the development of large software systems***, Proceedings, IEEE WESCON, August 1970: Dr. Winston W. Royce https://mosaicprojects.com.au/PDF-Gen/Royce_-_Managing_the_development_of_large_software_systems.pdf

[12] For more in-depth discussion on Royce's paper see: https://kallokain.blogspot.com/2023/09/waterfall-dark-age-of-software.html https://mosaicprojects.wordpress.com/2024/01/14/the-problem-with-waterfall/

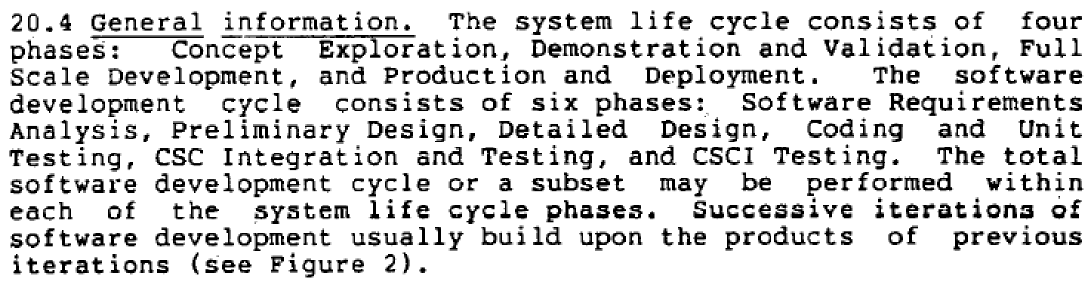
US DoD Specifications

The development of major software programs remained high-risk through the 1970s. This was particularly true in the development of major defense projects, where everything tended to be bleeding edge. To help mitigate the risk, in 1985 the US Department of Defense issued *DOD-STD-2167 Defense System Software Development – Requirements for the development of mission critical software*[13].

This specification unequivocally advocates an iterative approach to software development:


1.2 Application. Software development is usually an iterative process, in which an iteration of the software development cycle occurs one or more times during each of the system life cycle phases (Figure 1). Appendix B describes a typical system life cycle, the activities that take place during each iteration of software development, and the documentation which typically exists

Appendix B:


20.4 General information. The system life cycle consists of four phases: Concept Exploration, Demonstration and Validation, Full Scale Development, and Production and Deployment. The software development cycle consists of six phases: Software Requirements Analysis, Preliminary Design, Detailed Design, Coding and Unit Testing, CSC Integration and Testing, and CSCI Testing. The total software development cycle or a subset may be performed within each of the system life cycle phases. Successive iterations of software development usually build upon the products of previous iterations (see Figure 2).

Through to this point in time, the consensus of major software developers seems to have been:

1. Iterative development reduces risk and cost
2. Prototyping reduces risk and cost
3. Testing and feedback are essential at every stage of development.

So where did waterfall come from?

The Waterfall Dead-End

The start of the waterfall concept, or at least the first publication to use the term Waterfall, was the 1976 paper ***Software Requirements: Are They Really a Problem***[14], by T.E. Bell and T.A. Thayer.

The key paragraph states:

"The evolution of approaches for the development of software systems has generally paralleled the evolution of ideas for the implementation of code. Over the last ten years more structure and discipline have been adopted, and practitioners have concluded that a top-down approach is superior to the bottom-up approach of the past. The Military Standard set MIL STD 490/483[15] *recognized this newer approach by*

[13] Download a copy of ***DOD-STD-2167 Defense System Software Development – Requirements for the development of mission critical software*** (1985) from: https://mosaicprojects.com.au/PDF-Gen/DOD-STD-2167.pdf

[14] Download ***Software Requirements: Are They Really a Problem***: https://mosaicprojects.com.au/PDF-Gen/software_requirements_are_they_really_a_problem.pdf

[15] **Note**: These are configuration management standards, not software development documentation.

specifying a system requirements document, a "design-to" requirements document that is created in response to the system requirements, and then a "code-to" requirements document for each software module in the design. Each of these is at a lower level of detail than the former, so the system developers are led 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; 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 I.

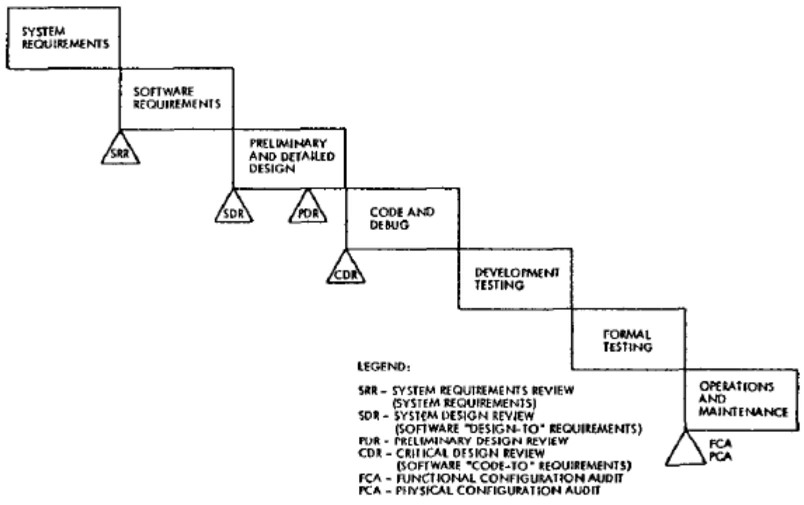

SYSTEM REQUIREMENTS
SOFTWARE REQUIREMENTS
SRR
PRELIMINARY AND DETAILED DESIGN
SDR
PDR
CODE AND DEBUG
CDR
DEVELOPMENT TESTING
FORMAL TESTING
OPERATIONS AND MAINTENANCE
FCA
PCA
LEGEND:
SRR - SYSTEM REQUIREMENTS REVIEW (SYSTEM REQUIREMENTS)
SDR - SYSTEM DESIGN REVIEW (SOFTWARE "DESIGN-TO" REQUIREMENTS)
PDR - PRELIMINARY DESIGN REVIEW
CDR - CRITICAL DESIGN REVIEW (SOFTWARE "CODE-TO" REQUIREMENTS)
FCA - FUNCTIONAL CONFIGURATION AUDIT
PCA - PHYSICAL CONFIGURATION AUDIT

Figure 1. Development Phases of the System Development Cycle

Each of the documents in the early phases of the ***waterfall*** *can be considered as stating a set of requirements. At each level a set of requirements serve as the input and a design is produced as output. This design then becomes the requirements set for the designer at the next level down."*

Note: Neither of the authors of this paper (Bell or Thayer) appear to have any expertise in the management of large software development projects. Their focus is on requirements management and configuration control, and their conclusion is the false assumption you can fully specify a software system before doing any development – if the documentation is not correct, do more documentation…. Once the documentation is perfect, development of the software will just happen.

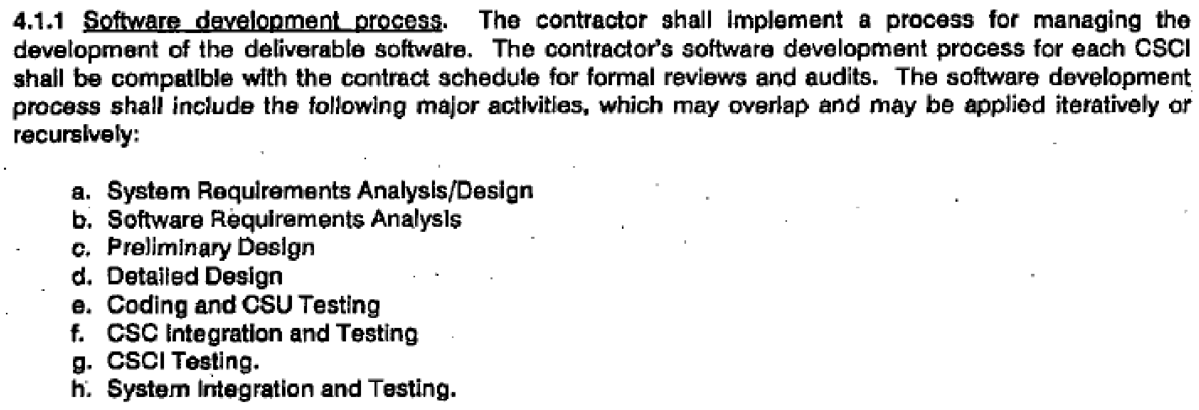
The Bell & Thayer paper clearly misrepresents the 1970 paper ***Managing the development of large software systems***, by Dr Winston Royce (which shows a limited understanding of software development). However, what it does show is a desire to instill discipline and control into the software development process similar to the engineering discipline needed to design and manufacture mechanical computers in the 1940s and still needed to manufacture complex mechanical equipment in the 1970s and 80s.

This attempt to force mechanical engineering discipline onto the software development process fed into the 1988 update to *DOD-STD-2167A Defense System Software Development*[16]. This updated standard shifted the requirements for the development of mission critical software to a sequential approach [waterfall] by default].

[16] Download a copy of ***DOD-STD-2167A Defense System Software Development*** (1988) from: https://mosaicprojects.com.au/PDF-Gen/DOD-STD-2167A.pdf

DOD-STD-2167A does reference iterative and recursive approaches to development, but tends to make implementing this type of approach difficult. If not impossible, in the detailed requirements.

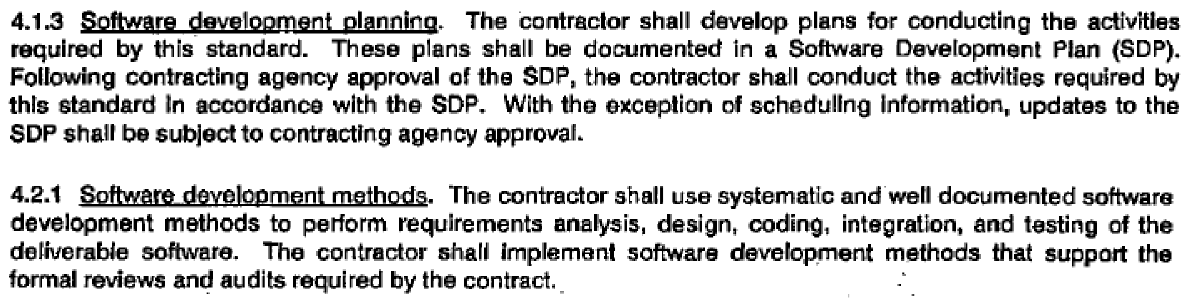
The same general flow in development as previously used is detailed, with an option that the sequence '*may be applied iteratively or recursively*'.



4.1.1 Software development process. The contractor shall implement a process for managing the development of the deliverable software. The contractor's software development process for each CSCI shall be compatible with the contract schedule for formal reviews and audits. The software development process shall include the following major activities, which may overlap and may be applied iteratively or recursively:

a. System Requirements Analysis/Design
b. Software Requirements Analysis
c. Preliminary Design
d. Detailed Design
e. Coding and CSU Testing
f. CSC Integration and Testing
g. CSCI Testing.
h. System Integration and Testing.

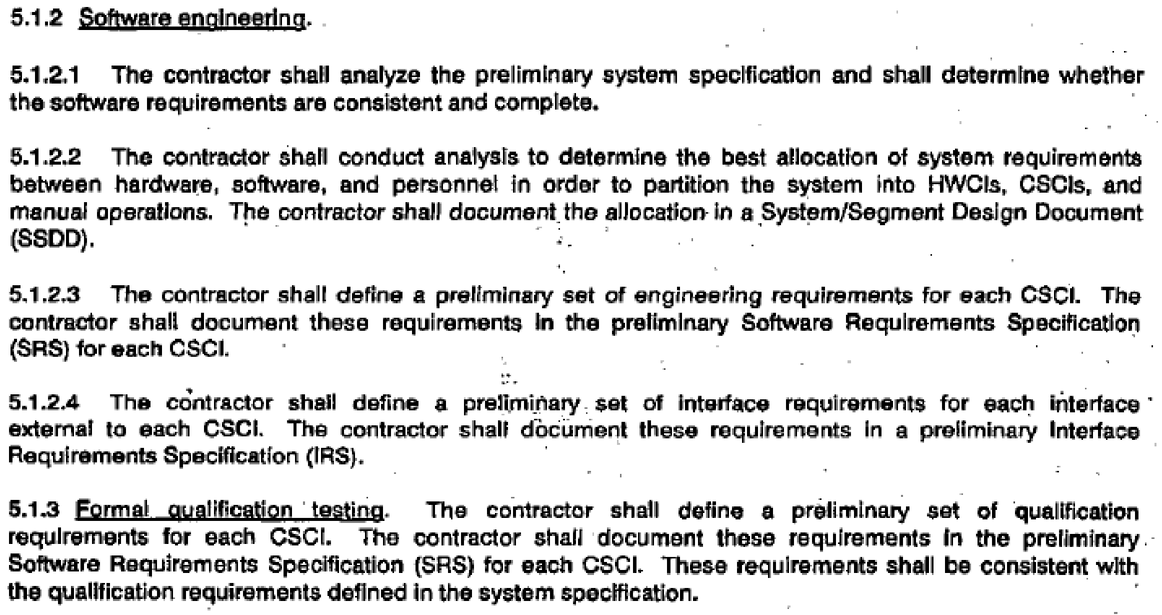
However, the Standard then goes on to require:



4.1.3 Software development planning. The contractor shall develop plans for conducting the activities required by this standard. These plans shall be documented in a Software Development Plan (SDP). Following contracting agency approval of the SDP, the contractor shall conduct the activities required by this standard in accordance with the SDP. With the exception of scheduling information, updates to the SDP shall be subject to contracting agency approval.

4.2.1 Software development methods. The contractor shall use systematic and well documented software development methods to perform requirements analysis, design, coding, integration, and testing of the deliverable software. The contractor shall implement software development methods that support the formal reviews and audits required by the contract.

These general requirements are followed by pages of mandatory documentation and analysis (part only shown):



5.1.2 Software engineering.

5.1.2.1 The contractor shall analyze the preliminary system specification and shall determine whether the software requirements are consistent and complete.

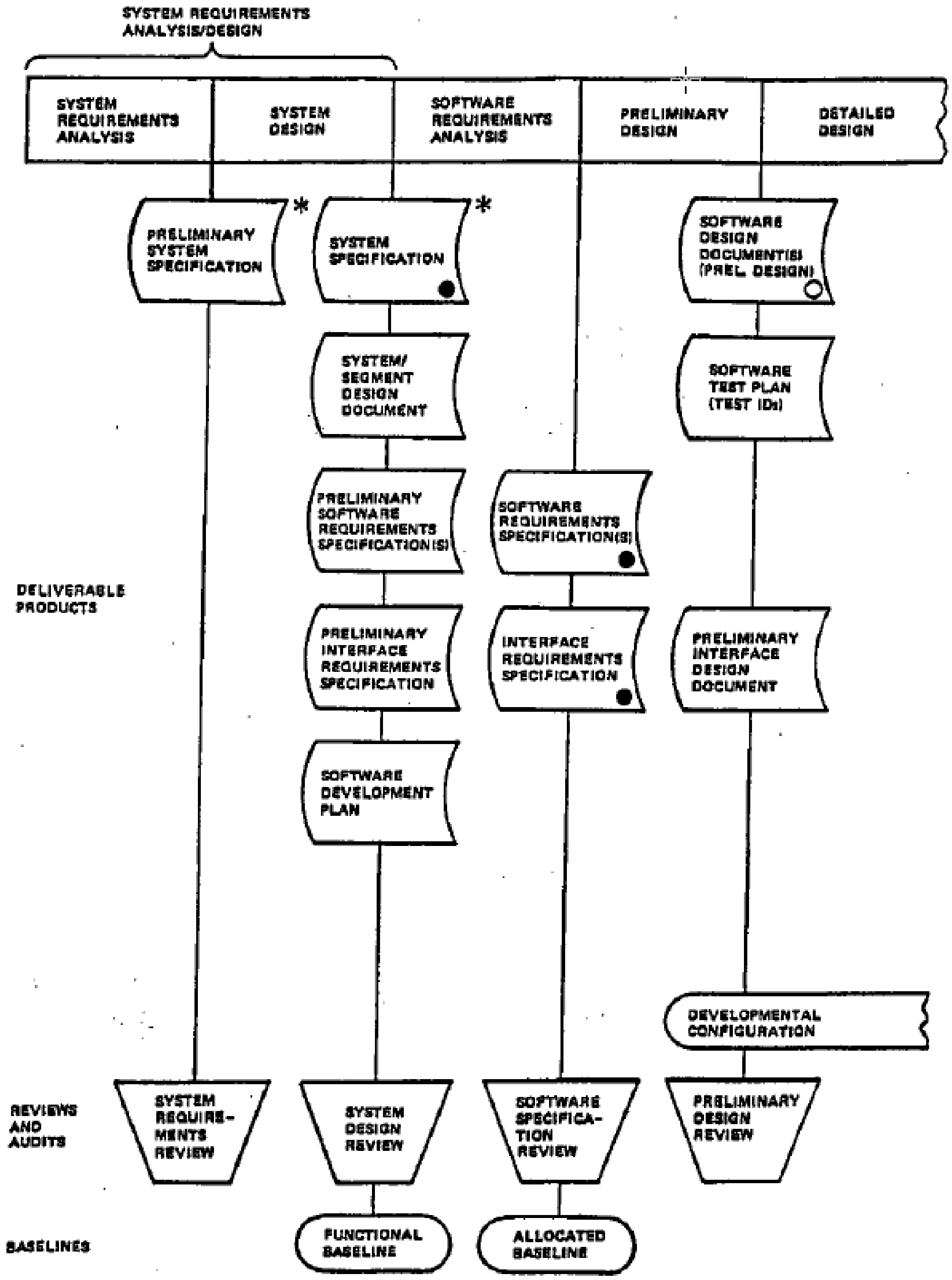
5.1.2.2 The contractor shall conduct analysis to determine the best allocation of system requirements between hardware, software, and personnel in order to partition the system into HWCIs, CSCIs, and manual operations. The contractor shall document the allocation in a System/Segment Design Document (SSDD).

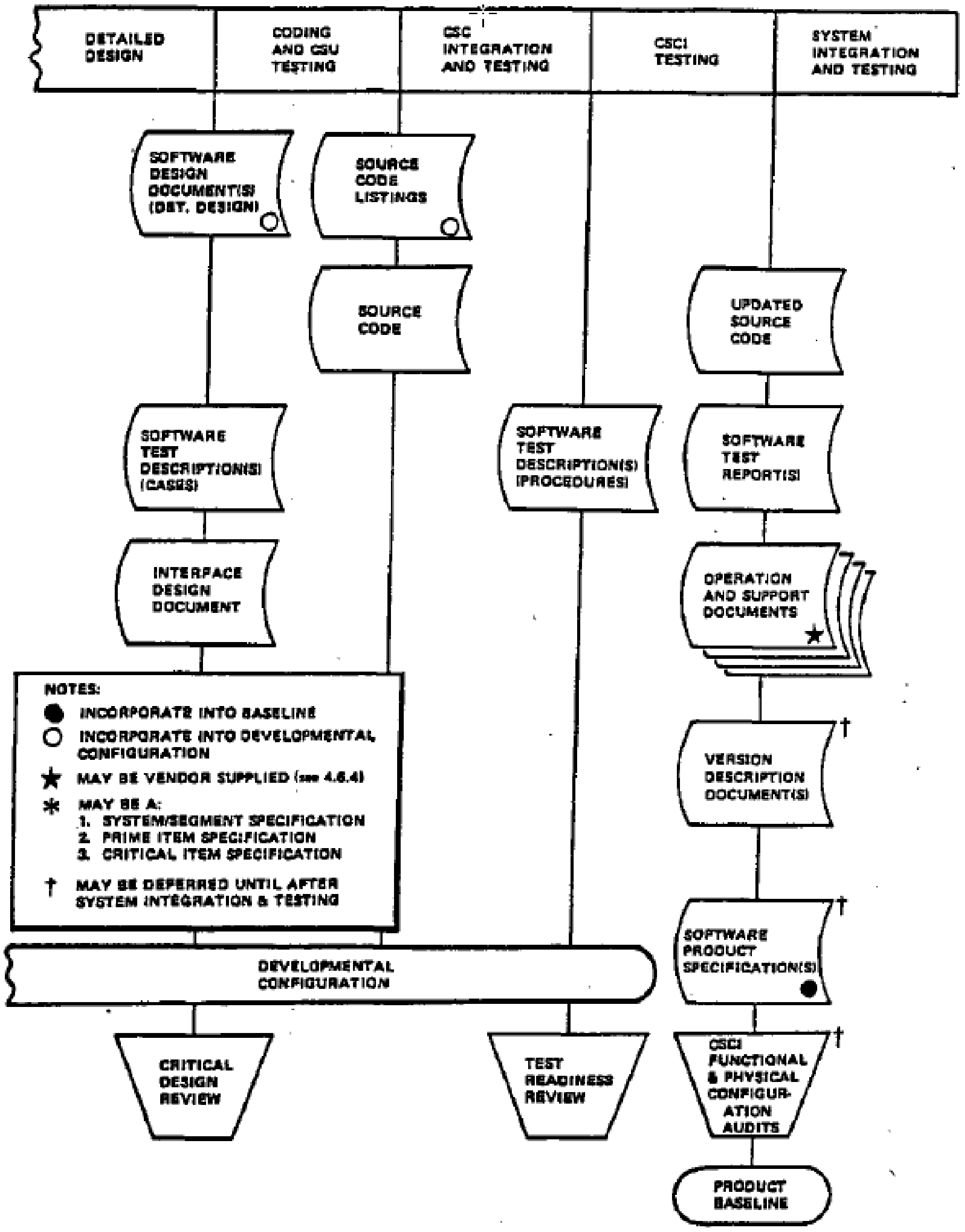
5.1.2.3 The contractor shall define a preliminary set of engineering requirements for each CSCI. The contractor shall document these requirements in the preliminary Software Requirements Specification (SRS) for each CSCI.

5.1.2.4 The contractor shall define a preliminary set of interface requirements for each interface external to each CSCI. The contractor shall document these requirements in a preliminary Interface Requirements Specification (IRS).

5.1.3 Formal qualification testing. The contractor shall define a preliminary set of qualification requirements for each CSCI. The contractor shall document these requirements in the preliminary Software Requirements Specification (SRS) for each CSCI. These requirements shall be consistent with the qualification requirements defined in the system specification.

And, the required documentation shown in Figure 2 is waterfall in everything but name:


SYSTEM REQUIREMENTS ANALYSIS/DESIGN
SYSTEM REQUIREMENTS ANALYSIS
SYSTEM DESIGN
SOFTWARE REQUIREMENTS ANALYSIS
PRELIMINARY DESIGN
DETAILED DESIGN
PRELIMINARY SYSTEM SPECIFICATION
SYSTEM SPECIFICATION
SOFTWARE DESIGN DOCUMENT(S) (PREL. DESIGN)
SYSTEM/ SEGMENT DESIGN DOCUMENT
SOFTWARE TEST PLAN (TEST IDs)
PRELIMINARY SOFTWARE REQUIREMENTS SPECIFICATION(S)
SOFTWARE REQUIREMENTS SPECIFICATION(S)
DELIVERABLE PRODUCTS
PRELIMINARY INTERFACE REQUIREMENTS SPECIFICATION
INTERFACE REQUIREMENTS SPECIFICATION
PRELIMINARY INTERFACE DESIGN DOCUMENT
SOFTWARE DEVELOPMENT PLAN
DEVELOPMENTAL CONFIGURATION
REVIEWS AND AUDITS
SYSTEM REQUIRE-MENTS REVIEW
SYSTEM DESIGN REVIEW
SOFTWARE SPECIFICA-TION REVIEW
PRELIMINARY DESIGN REVIEW
BASELINES
FUNCTIONAL BASELINE
ALLOCATED BASELINE

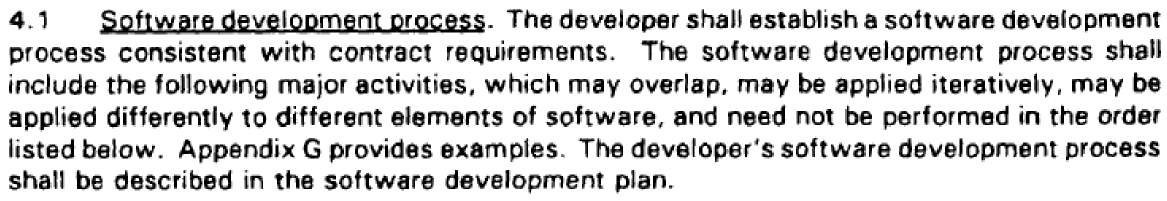

DETAILED DESIGN
CODING AND CSU TESTING
CSC INTEGRATION AND TESTING
CSCI TESTING
SYSTEM INTEGRATION AND TESTING
SOFTWARE DESIGN DOCUMENT(S) (DET. DESIGN)
SOURCE CODE LISTINGS
SOURCE CODE
UPDATED SOURCE CODE
SOFTWARE TEST DESCRIPTION(S) (CASES)
SOFTWARE TEST DESCRIPTION(S) (PROCEDURES)
SOFTWARE TEST REPORT(S)
INTERFACE DESIGN DOCUMENT
OPERATION AND SUPPORT DOCUMENTS
NOTES:
INCORPORATE INTO BASELINE
INCORPORATE INTO DEVELOPMENTAL CONFIGURATION
MAY BE VENDOR SUPPLIED (see 4.6.4)
MAY BE A:
1. SYSTEM/SEGMENT SPECIFICATION
2. PRIME ITEM SPECIFICATION
3. CRITICAL ITEM SPECIFICATION
MAY BE DEFERRED UNTIL AFTER SYSTEM INTEGRATION & TESTING
VERSION DESCRIPTION DOCUMENT(S)
SOFTWARE PRODUCT SPECIFICATION(S)
DEVELOPMENTAL CONFIGURATION
CRITICAL DESIGN REVIEW
TEST READINESS REVIEW
CSCI FUNCTIONAL & PHYSICAL CONFIGURATION AUDITS
PRODUCT BASELINE

Add the normal government inspection and validation requirements at each stage and you have the waterfall methodology.

DOD-STD-2167A Defense System Software Development was superseded in 1994 by the publication of *MIL-STD-498: Software Development and Documentation*[17].

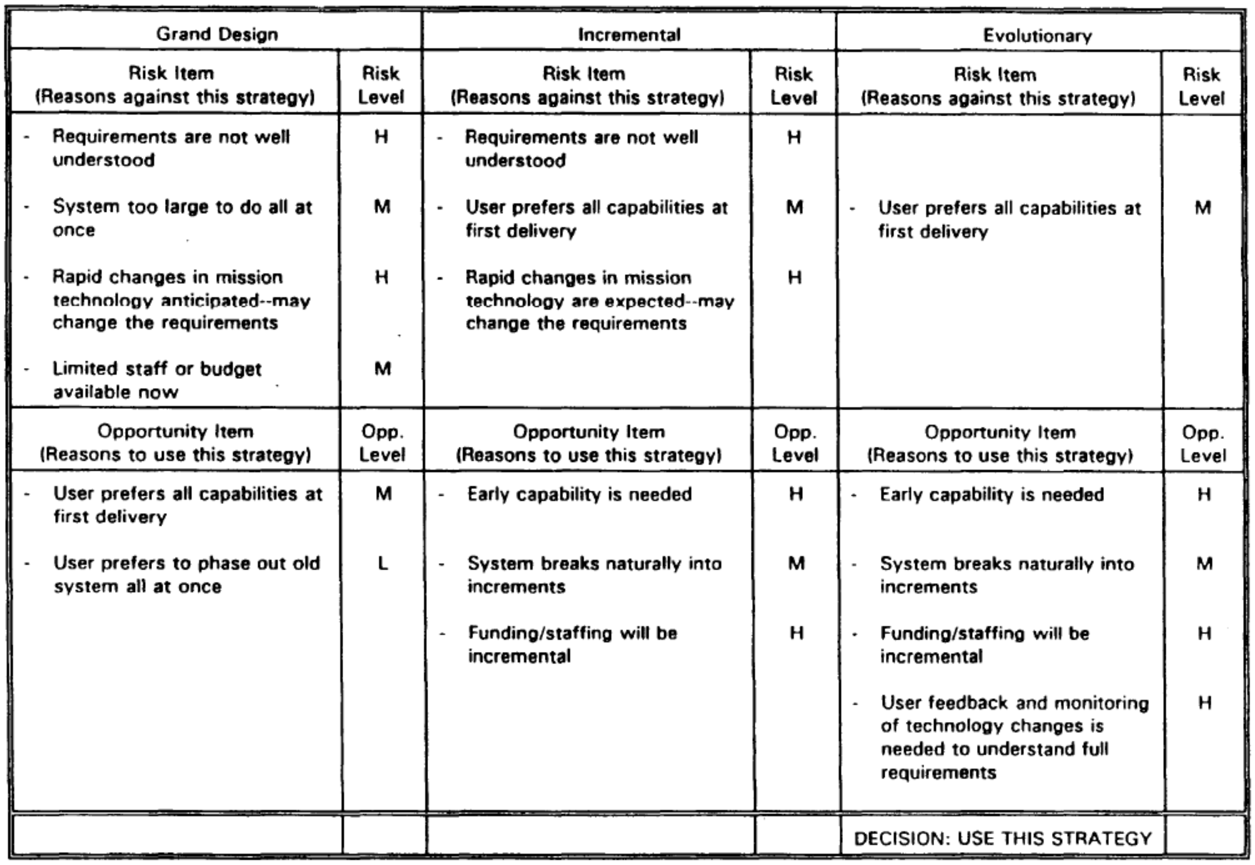
[17] Download a copy of ***MIL-STD-498: Software Development and Documentation*** (1994): https://mosaicprojects.com.au/PDF-Gen/MIL-STD-498.pdf

This is a much thicker standard, and it allows any appropriate approach to software development to be used:



4.1 Software development process. The developer shall establish a software development process consistent with contract requirements. The software development process shall include the following major activities, which may overlap, may be applied iteratively, may be applied differently to different elements of software, and need not be performed in the order listed below. Appendix G provides examples. The developer's software development process shall be described in the software development plan.

And provides guidance on the appropriate approach to use based on a risk assessment:



Grand Design		Incremental		Evolutionary	
Risk Item (Reasons against this strategy)	Risk Level	Risk Item (Reasons against this strategy)	Risk Level	Risk Item (Reasons against this strategy)	Risk Level
- Requirements are not well understood	H	- Requirements are not well understood	H		
- System too large to do all at once	M	- User prefers all capabilities at first delivery	M	- User prefers all capabilities at first delivery	M
- Rapid changes in mission technology anticipated--may change the requirements	H	- Rapid changes in mission technology are expected--may change the requirements	H		
- Limited staff or budget available now	M				
Opportunity Item (Reasons to use this strategy)	Opp. Level	Opportunity Item (Reasons to use this strategy)	Opp. Level	Opportunity Item (Reasons to use this strategy)	Opp. Level
- User prefers all capabilities at first delivery	M	- Early capability is needed	H	- Early capability is needed	H
- User prefers to phase out old system all at once	L	- System breaks naturally into increments	M	- System breaks naturally into increments	M
		- Funding/staffing will be incremental	H	- Funding/staffing will be incremental	H
				- User feedback and monitoring of technology changes is needed to understand full requirements	H
				DECISION: USE THIS STRATEGY	

FIGURE 8. Sample risk analysis for determining the appropriate program strategy.

Over the intervening years, the USA DoD, NASA, and government in general have adopted agile with a focus on iterative and incremental development across most aspects of their software development and maintenance[18]. Another important development (which may be the subject of a future paper) is a focus on increasing the probability of success for complex system of systems by integrating systems engineering with agile project management[19].

[18] A detailed listing of current US publications focused on using Iterative and Incremental Development (IID) and agile in major programs, including linking agile with EVM, has been published by Glen Alleman, see ***Agile in NASA, DoD, and DOE***: https://www.linkedin.com/pulse/agile-nasa-dod-doe-glen-alleman/

[19] For an overview on systems engineering and agile see ***Agile is Systems Engineering***: https://www.linkedin.com/pulse/agile-systems-engineering-glen-alleman/

However, by the time *MIL-STD-498* was published, the habit of waterfall was entrenched in many organizations (not just the DoD and its contractors) and continues to be used through to the present time.

Waterfall's attraction

The attraction of waterfall to DoD management is to an extent understandable:

1. The management paradigm was hard engineering - you need the design of an aircraft or missile to be close to 100% before cutting and riveting metal - computers were very new big pieces of 'metal' why treat them differently?
2. For the cost of 1 hour's computer time, you could buy a couple of months of software engineering time - spending time to get the design right before running the computer nominally made cost-sense.
3. The ability to work on-line was only just emerging and memory was very limited. Most input was batch loaded using punch cards or tape (paper or magnetic). Expecting skilled engineers to do-it right, load-it-once and see working code may not have seemed too much to expect from skilled engineers.

The problem was in the 1980s no one really understood the effects of complexity so when problems emerged it was easier to blame human error rather than the challenges of developing systems of systems.

Waterfall's resilience

There seems to be two reasons for the long-term survival of waterfall.

The first is the illusion of control – the waterfall process generates documentation that can be checked and a detailed plan that rarely achieved. But many managers like to feel in control even if it is an illusion. An iterative, emergent process may deliver better outcomes but requires a collaborative approach, skill, agility, and trust. Unfortunately, far too many managers seem to opt for the illusion of 'command and control' – they can always blame the contractor (or project manager) when the inevitable delays and cost overruns occur.

The other driver is financial. From the perspective of a contractor, and given many of the contracts using this standard were cost reimbursable, and/or cost increases were relatively easy to obtain if changes were required to previously approved documents, which option would you prefer:

- An approach that requires less training, and compensates you for delays and cost overruns, or
- An approach that will require you to spend money on extra training for your people, and will simultaneously reduce project lead time and cost?

There was a powerful financial incentive for contractors to stick to a waterfall approach.

However, the situation is more complex. In his discussions on agile and waterfall, Glen Alleman[20] often suggests ***'Don't so stupid things on purpose!'***, this is sound advice. But, *'stupid'* depends on your perspective:

- There are some projects, similar in nature to the early mechanical computers discussed at the start of this article that need to be designed and developed sequentially, these are typically 'hard'

[20] Glen Alleman, Herding Cats Blog: https://herdingcats.typepad.com/

engineering and construction type projects. The processes used are not waterfall but are design driven and sequential (often with overlaps between phases)[21].

- Then there are all of the other 'soft' projects[22] that produce largely intangible outputs, software, business change, marketing, etc. These projects will generally benefit from an adaptive, agile approach but:
    - Agile concepts only work effectively with management support, they require a supportive culture,
    - Hybrid systems need to be crafted to be effective,
    - If your focus is on making profits as a contractor, waterfall is not a bad idea, but
    - If your focus is on client satisfaction, the right agile approaches are likely to deliver better outcomes.

For a detailed discussion on this see ***The Myth of the waterfall SDLC***[23].

Flavors of Agile – 1970s to 2020s

Agile has many flavors and variations that continue to evolve, and as outlined above the ideas of iterative, incremental, and evolving development processes predate the publication of the Agile Manifesto in 2001.

Iterative and incremental development predates major software developments, IID grew from the 1930s work of Walter Shewhart, a quality expert at Bell Labs who proposed a series of short "plan-do-study-act" (PDSA) cycles for quality improvement. This concept was picked up and promoted in the 1940s by quality guru W. Edwards Deming.

The X-15 hypersonic jet was a milestone 1950s project applying IID, and the practice was considered a major contribution to the X-15's success. Although the X-15 was not a software project, it is noteworthy because some personnel moved across to NASA's early 1960s Project Mercury, which did apply IID in software and the IBM Federal Systems Division (FSD), another early IID proponent[24].

Some of the key concepts and methodologies used in software development are:

1953 **Prototyping**. Used on the SAGE project.

1968 **Iterative and Incremental Development (IID)**. Promoted by Brian Randell and F.W. Zurcher at the IBM T.J. Watson Research Center. Also advocated by Royce in his 1970 White Paper. IID focuses on an evolutionary development process with rigorous testing at each stage.

1981 **Object-Oriented Programming (OO)**. OOP was evolving for 20 years prior to the public release of Smalltalk 80 which allowed the widespread adoption of modern software development approaches.

[21] For more in-depth discussion on project types suited to either a planned approach, or agile, see ***The Problem with Waterfall***: https://mosaicprojects.wordpress.com/2024/01/14/the-problem-with-waterfall/

[22] For a definition of hard and soft projects see ***Hard -v- Soft Projects***: https://mosaicprojects.wordpress.com/2023/01/21/hard-v-soft-projects/

[23] For further discussion on this topic see ***The Myth of the waterfall SDLC***: http://www.bawiki.com/wiki/Waterfall.html

[24] A detailed history of IID is contained in ***Iterative and Incremental Development: A Brief History***: https://mosaicprojects.com.au/PDF-Gen/History_of_Iterative_and_Incremental_Development.pdf

1988 **Spiral**. Barry Boehm publishes *A Spiral Method of Software Development and Enhancement* advocating an iterative approach to development.

1991 **Rapid Application Development (RAD)**. Book published by James Martin based on his work at IBM. RAD puts less emphasis on planning and more emphasis on an adaptive process. Prototypes are often used.

1994 **Adaptive Software Development**. Created by Jim Highsmith and Bayer, developed from RAD and embodies the principle that continuous adaptation of the process to the work at hand is the normal state of affairs.

1994 **Scrum.** Created by Ken Schwaber and Jeff Sutherland, as a management framework to support the implementation of other software development (coding) methods. Scrum is designed for teams of ten or fewer members, who break their work into goals that can be completed within timeboxed iterations, called sprints.

1994 **Dynamic Systems Development Method (DSDM)**. DSDM covers a wide range of activities across the whole project lifecycle and includes strong foundations and governance. It is an iterative and incremental approach that embraces principles of Agile development, including continuous user/customer involvement.

1996 **Extreme Programming (XP)**. Created by Kent Beck and Ron Jeffries to reduce risk and simplify management. XP advocates frequent releases in short development cycles.

1997 **Crystal Orange**. One of the Crystal family of agile methodologies developed by Alistair Cook.

1997 **Feature-driven development (FDD)**. FDD is an iterative and incremental software development process, developed by Jeff De Luca, that blends a number of industry recognized best practices into a cohesive whole.

1998 **Rational Unified Process (RUP)**. Developed from 1996 onwards, as an iterative software development framework based on OOP

2000 **Adaptive software development (ASD)**. ASD grew out of RAD

2001 **Agile Manifesto published**[25]. Designed to increase awareness of the various agile methodologies.

2003 **Lean Software Development**. By Tom and Mary Poppendieck provides a theoretical foundation explaining why agile works.

2004 **Kanban**. Is a system that is designed to reduce WIP (Work-In-Progress) and contribute to self-managed teams continually delivering high value features. It was originally developed as part of the Toyota Production System (manufacturing) in 1953. The use of Kanban software development was pioneered by former Microsoft engineer David J. Anderson.

2004 **Agile modeling (AM)**[26]. AM is a method of modelling based on OO developed by Scott Ambler. AM iss used for modeling and documenting software systems based on a collection of values and principles that can be applied to a software development project. (Initially called eXtreme Modeling or XM)

[25] For more on the ***Agile Manifesto*** see: https://mosaicprojects.com.au/PMKI-ITC-040.php#Process1

[26] ***The Object Primer*** 2nd Edition published by Scott Ambler in 2004 described Agile Modelling (AM) in Chapter 2, this built on previous books and concepts developed by Ambler.

2008 **Agile unified process (AUP)**. AUP is a simplified version of RUP developed by Scott Ambler. It uses a simple, easy to understand approach to developing business application software using agile techniques.

2009 **Scrumban**. A hybrid of Scrum and Kanban.

2011 **Scaled Agile Framework (SAFe)**. Used to implement Agile practices at an enterprise scale. Created by Dean Leffingwell, SAFe 1.0 was a collection of best practices and principles to promote collaboration, alignment, and improved decision-making across teams, programs, and portfolios. It applied the principles of Agile development, Lean manufacturing, and systems thinking to large, complex, and distributed software and systems development environments[27].

2015 **Disciplined Agile Framework (DA)**. Was developed to provide a more cohesive approach to agile software development. Now owned by PMI. Disciplined agile delivery (DAD) is the software development portion of the disciplined agile toolkit. It enables teams to make simplified process decisions around incremental and iterative solution delivery.

2020s **Standard project control processes integrated with agile**. As agile becomes mainstream, integration with standard project controls becomes more important[28]:
- Linking EVM and agile
- The development of Work Performance Management[29].

Conclusions

Agile in the form of iterative development started with the SAGE, the world's first major software development project! And agile remains a real and evolving concept in project management.

Unfortunately, the development of agile is not being helped by the agile evangelists and anarchists. There may be a place for no planning, no estimating, and just letting people get on with developing code.
However, this is only likely to occur in a limited number of situations:

- Small start-ups and other situations where the consequences of failure is not significant, and the people doing the development own the outputs,
- Situations where agile approaches are being used for day-to-day maintenance[30],
- Situations where the work is urgent and simply must be done.

Fundamentally, these types of work-space are not projects, even if agile methodologies are being used. The Mosaic website is developed in this way.

For the rest of the world, various forms of agile with appropriate levels of discipline offer real advantages in the development of a quality (ie, fit for purpose) product in most software and other soft project environments. The appropriate methods to use and the levels of technical excellence and control required, depend on the situation – there are plenty of guidelines available to provide advice.

---

[27] For more on ***the evolution of SAFe*** see: https://www.linkedin.com/pulse/evolution-safe-look-frameworks-adaptations-over-time-daniel-michael/

[28] See ***Controlling Agile***: https://mosaicprojects.com.au/PDF_Papers/P205-Controlling_Agile.pdf

[29] For more on ***Work Performance Management*** see: https://mosaicprojects.com.au/PMKI-SCH-041.php#Overview

[30] For more on non-project uses of Agile see ***De-Projectizing IT Maintenance***: https://mosaicprojects.com.au/Mag_Articles/N010_De-Projectising_IT_Maintenance.pdf

There are also many situations where agile is totally inappropriate, typically engineering, construction, and other hard projects where significant design has to be completed before manufacturing, assembly, or construction can start.

Problems start when management attempts to force traditional hard project approaches onto a soft project. This is often done on the false assumption that following a heavily documented, pre-defined sequence generates control, and/or allows detailed management (micro-management). 60+ years of experience show this approach does not work when the product being developed requires innovation and creativity (ie, a typical software project).

The biggest furphy[31] is the illusion of 'Waterfall Project Management'[32]. Waterfall did exist for a few years as a software development methodology, but its origins were based on a false premise and the organization that standardized the approach abandoned it to return to iterative development approaches. The only people still talking about waterfall project management are either agile evangelists who use the term to describe all forms of bad management, or people who have been confused by the evangelists.

The way forward is for management to first decide on the appropriate management approach based on the type of project, then if agile is selected, work out how disciplined the development need to be and select the best methodology to achieve a successful outcome.

____

First Published 1st February 2024 – Augmented and Updated

[31] A **furphy** is Australian slang for an erroneous or improbable story that is claimed to be factual.

[32] See ***There Was No Such Thing as Waterfall Project Management***: https://www.linkedin.com/pulse/thing-waterfall-project-management-glen-alleman-mas4c/

Take the guesswork out of predicting project completion
PERFORMANCE
Work Performance Management
Bring your projects home on time:
WPM Calculates the completion date based on the work performed – no need for a schedule!