THE MYTH OF THE 'WATERFALL' SDLC

Source: http://www.bawiki.com/wiki/Waterfall.html Downloaded 1st February 2024

This is admittedly my[1] interpretation, but I have tried to provide links to all of the source material I have used so that you can do your own research if you don't agree with my conclusions.

Also known as: Phased Development, Plan-Driven Development, Specification-Driven Development, Cascade Model

Introduction

This is the 5th major version of this wiki article, and I thought it might be useful to provide a bit of introduction as to why I have written (and occasionally update) such a long article about a ‘discredited’ methodology like Waterfall.

Several years ago, my employer started making efforts to formally adopt some Agile processes (Scrum at the time, DAD currently) for some project types. Wanting to educate myself more I started doing Google searches and reading what material I could find. But my reading got hijacked as I came across descriptions of ‘Waterfall’ over and over again that simply did not match the way we did it at my employer, or the way any other BA I had talked to had described the way they did projects.

So, I got curious. Did my employer adopt some ‘improved’ version of Waterfall that wasn’t so rigid? Were there actually people out there at some point advocating the highly-rigid form of Waterfall I kept reading about on ‘Agile’-oriented web sites? If so, what was their logic and what could I learn from it?

So I started researching Waterfall. I found out where it supposedly originated from and dug up the original paper (the Royce paper I discuss in depth below) on the internet. And surprisingly, it had almost NOTHING in common with the rigid process I kept reading about on Agile sites.

So, I started researching more, and I was surprised to start coming across research papers that cited the Royce paper, but which completely mis-characterized what it said. These were papers published in academic journals, not some random blog, so you would expect a higher-level of quality. So I researched more and more; and what fascinated me was the dichotomy of what people were consistently referring to as ‘Waterfall’, and what had actually been published in what were cited as the main ‘sources’ of the Waterfall methodology.

So thus was born this article on ‘Waterfall’. It is not an ‘apology’ for or a ‘defense’ of Waterfall. It is my best effort (so far) to show that as far as I can tell no one has ever seriously advocated for the highly rigid software process that is commonly called Waterfall [with the caveat that I may not have come across the right reference yet]. And not only had no one ever advocated it, that the people "blamed" for this process called Waterfall had advocated for things very different than what the current understanding is. Lastly, I would argue that if you consider the environment that early software development took place in that there are a number of very valid reasons for a planning-centric, documentation-heavy development process to be used as the best option available.

Most importantly though, I think it is important to challenge some of the common assumptions and 'common knowledge' that exists in the technology world and question whether that

[1] David Olson and I’ve been a Business Analyst with a global financial services firm since 2004.

information is true or is presented in certain ways to serve personal or commercial agendas. I happen to think that 'Waterfall' is a strong example of this sort of thing that by understanding the background you as a reader may be less inclined to accept statements at face value and do more critical thinking of your own. This is why all the sources I have used are fully documented and for the majority of them I provide direct links where possible. It gives you the option to validate my work and form your own opinion.

I would also say that it's important to understand the context under which different ideas and approaches arose. By understanding that context you can understand what may need to be re-evaluated as the context changes and identify similar contexts in your own situation. This way you may have a better idea of what may work in your current context and why, saving you the trouble of having to learn the hard way.

You may not care about any of this. Or the history involved. Or what the 'sources' actually said. You may find all the contextual information tedious and uninteresting. In that case, feel free to read the What is it? section immediately below and then skip down past the Origins section and ignore everything in between.

But if you do find it interesting, please read the whole article. If you know of references I haven't touched on that are appropriate, please let me know. I am completely open to changing this article, but I want to make sure that most of the article is grounded in actual references. People can then form their own opinions based on the evidence.

My conclusion after reading all that I have so far is the same as it was before I started doing all this research. There is no universally applicable development methodology. You need to tailor your development process to the situation at hand. Sometimes that will look very 'Agile', sometimes it will look very 'Waterfall', and sometimes have parts of both or neither. As has been repeatedly said, There is no silver bullet.

What is it?

Strangely enough, this is probably the most controversial aspect of Waterfall. There seems to be a common 'understanding' of what Waterfall is among a great many people in the technology world, but I would argue that that 'understanding' is not supported by ANY of the references that are named as the 'source' of Waterfall.

The best initial description that I can come up with based on my reading is that the Waterfall model is a Systems (or Software) Development Life Cycle (SDLC) process model that is project management and solution-design focused; and which utilizes a highly-planned, specification-driven development process. It was probably the first formalized instance of a SDLC model, and nearly every software development model since has incorporated some of its features.[32]

What is now called Waterfall is frequently thought to have been first proposed by Winston W. Royce in a 1970 paper,[2] but aspects of the conceptual process go back to at least a 1956 paper by Herbert W. Bennington. [1, 32]

However, the Waterfall model is also seen as highly controversial by some in the software development community. Indeed, some have gone so far as to call it a toxic concept,[4] and the most costly mistake in the world.[17] Partly, this is due to wide-ranging differences in how people define the waterfall model. And I think it is partly due to a desire to have Waterfall as a useful straw-man to use for promoting personal agendas. These are due to both common misunderstandings of the history of Waterfall and because of the changing nature of what has been called Waterfall over time.

In general, the way the Waterfall model is described in current times seems to be highly dependent on the knowledge, agenda, and personal biases of the person describing it.

What's in a Name?

So why is it called the Waterfall model? First, you need to know that Royce never used that term in describing his work and to understand that the Waterfall model name was assigned by others.

Second, it helps to look at a common visual representation of the model such as the one below (which is similar to the one used on Wikipedia):

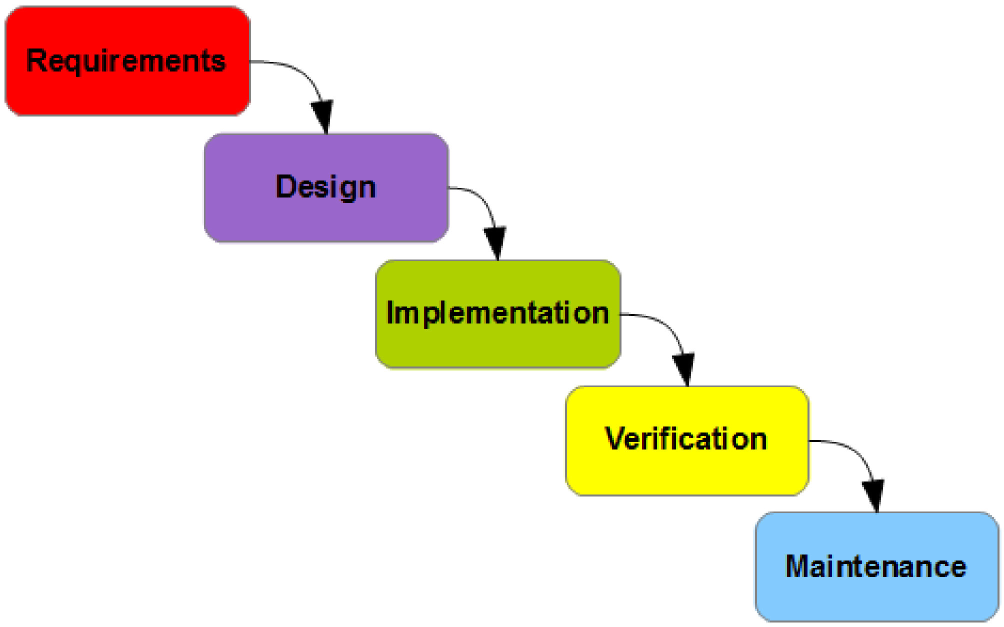


Figure A

If you look at that diagram, the common description of 'Waterfall' as … each phase cascades down into the next, you know, like a waterfall begin to make sense. [10]

However, the oldest reference to Waterfall does not use the 'Waterfall' name in that way and allows for far more subtlety in interpretation. That earliest reference to 'Waterfall' comes in a 1976 research paper by Bell & Thayer,[12] 6-years after Royce published his paper. The full quote is included below because I think it provides a good example of the way the Royce paper was perceived initially and that the 'Waterfall' nomenclature was not initially meant the way it later came to be interpreted.

Bell and Thayer say the following:

> *"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 MILSTD 490/483 recognized this newer approach by 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 [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 I. [Figure 1 is a standard Waterfall diagram like the one immediately above]*
>
> *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. With so many levels of requirements documents, and with so few software projects mapping nicely into the scheme, we must be more specific about what we mean by the term "software requirements" as used in our studies.* ***We do not mean all the various levels of requirements, but only a single one, one that can usually be identified in large software development projects that have ended successfully. At this level the software requirements describe functions that the software must perform, but not how they must be implemented.*** *For example, they might contain a requirement that a BMD system identify erroneous RADAR returns that have any of five specific characteristics. However, the software to meet this requirement might be spread through twelve subroutines using any of a large number of identification techniques."* [12] *(emphasis mine)*

A Tale of Two Waterfalls

The problem with discussing the Waterfall Model is that there are at least two significantly different understandings of what it is. The most common interpretation is what I am calling the 'Frozen Waterfall' in this article. The other is what Royce originally described in his paper. Because Royce is most commonly cited as the 'source' of the Waterfall method, I am going to give what he described priority for the use of the term 'Waterfall'.

Thus, for the purposes of this article, I am going to treat Waterfall as being the process that Royce fully espoused in his 1970 paper. But first, I think we need to address the 'frozen' [10] Waterfall interpretation.

'Frozen' Waterfall

Most commonly modeled using a diagram such as Figure A above (which is a recreation of the one on Wikipedia), the 'frozen' Waterfall approach is usually described as having the following characteristics:

1. It is a sequential, rigidly-planned process consisting of several phases that must be completed in sequence. [25, 33, 34, 38]
2. Each phase is a silo, completely separate from the other stages and the results of each phase are frozen once the phase is complete. [10, 38]

3. Each phase must be completed with 100% certainty before the project team moves on to the next phase. [5, 10, 33, 34]
4. It ignores end-user development and end-user involvement outside of requirements specification. [4] Or that if requirements changes are made after coding starts, they are made without involving stakeholders. [24]
5. That requirements can never be changed once the requirements phase is complete [5, 10, 34, 36, 37], and as a result it "assumes human developers are capable of correctly getting the requirements, design and tests correct **on the first try**." [4, 36] (emphasis mine)
6. It "separates analysis from design", forcing developers to "generate a solution, without providing any guidance as to how the solution is generated". [4]

There are lots of references to this 'frozen' waterfall methodology, especially from 'Agile' proponents, but I cannot find a single paper that espouses this version of the Waterfall methodology. I can find quite a few who attribute these characteristics to 'Waterfall', while usually referencing the Royce paper, but none that actually espouse this (including Royce). From what I can tell, this 'frozen' Waterfall has never been advocated for by anyone, although it may have existed due to the ignorance of those trying to follow what they thought was the 'Waterfall' process.

A less opinionated, more academic description of the "problems" with 'frozen' Waterfall can be found in a white paper by QinetiQ, a British defense contractor that was formerly part of a UK government agency. [25] The QinetiQ paper states that the Waterfall model was based on a number of assumptions:

- that all its stages could be completed in sequence
- that the costs and benefits of an information system could be calculated in advance
- that users knew what they wanted
- that the work needed was known and could be measured
- that programs once written could be altered
- that the right answer could be produced first time

But even this description seems to be largely unsupportable when you read the various documents that provide the foundation for "Waterfall".

Royce's Waterfall

So, if the description above is for the 'Frozen' interpretation of Waterfall, what exactly is the "Waterfall model" as proposed by Royce and modified by others as an idea of 'what to do', rather than 'what not to do'?

Based on my reading of the literature (mostly covered below), I would say that in general the 'Waterfall' model has the following characteristics:

- It is a process that seeks to reduce risk and costs through planning, documentation, and process controls.
- It includes a significant up-front effort to elicit, analyze, evaluate, and confirm both user and "business" needs before development begins in order to reduce the amount of re-coding that needs to be done later, and to ensure (as much as possible) that the system architecture is sufficient for the current and future needs of the client.

- It is a process that is focused not just on the software being developed, but strongly factors in the context the software will be used in and supporting the full lifecycle of a software product (development, testing, deployment, maintenance, enhancements, and retirement).
- The process is comprised of a set of phases that are roughly sequential in nature, in that later phases depend on input from prior phases. However, these phases can overlap, interact, and be revisited.
- Because of its planning-focused nature, the overall cycle is generally completed a small number of times (most commonly 1 to 2 times). If executed more than once, the first cycle is usually for the creation of a prototype.
- The process is generally documentation heavy, especially compared to “Agile” processes.
- The process is generally intended for use in large, complex efforts that require careful coordination among multiple teams in order to be successful.
- Once a certain stage in the overall process has been reached (usually requirements sign-off), a formal change-control process is used to manage change efforts.
- Quality Assurance is integrated by splitting each phase into two parts: the first part executes the work of the phase, and the second part validates it. [32]

Please note that the above description does NOT mean that each phase must be 100% complete before the next phase can be engaged. What it means is that in general, if you want to do analysis, you need to have something to analyze. So the results of a Software Requirements activity (there can be multiple activities) would be the input to an Analysis activity, which would in turn be the input into a Design activity.

Some Historical Context

Before discussing this history and details of what The Waterfall model are, I think it is critical for readers to first understand the context in which it developed.

The Hardware

The 1950’s and the 1960’s were the first 2 decades of the computer era. This was largely the era of mainframe computers, with mini-computers starting to appear on the scene in the mid-1960’s. Personal computers such as the IBM PC and Macintosh weren’t even a consideration during this time period, as they did not appear until the early to mid-1980’s.

Additionally, the capacity of even the best computers was far more limited than modern computers. Much of the software was being developed as parts of integrated systems, where issues of computing speed, memory amounts, and system design all impacted how software could operate and which posed challenges of resource scarcity that modern programmers rarely have to deal with.[1]

The Development Environment

Most development was probably being done in assembly, ALGOL (mid-1950's), Fortran (first compiler in 1957), COMTRAN (released in 1957), FLOW-MATIC (general compiler available in 1959), COBOL (replaced COMTRAN at IBM in 1962), and PL/I (first compiler 1966).

The C programming language was not developed until the early 1970's, and not formalized until the late 1970's. Most object-oriented languages did not receive wide-spread use until the mid-1980's.

The languages used were less concise, and thus more subject to coding errors. For example, a simple hello world program in a modern language like C can be done in 4 lines of code or less. In an older language like COBOL it took 17 lines of code. [20]

Lastly, software development tools were much simpler. While better development tools became more common in the mid-to-late 1960's, modern tools like code-highlighting, robust Integrated Development Environments (IDE's), version control systems like Git, and many other tools that make modern development easier and of higher quality did not exist at the time.

The Developers

While the early software developers of the 1950's were mostly engineers or mathematicians, by the late-1950's and early 1960's the need for programmers vastly exceeded the number of people available with engineering and mathematics backgrounds. The programming field was flooded with people whose training was in the humanities, social sciences, foreign languages, and fine arts. [21] Many of these were reportedly influenced by the "question authority" attitudes of the 1960's [21] and were resistant to attempts to organize and manage the development process.

Many of these issues continued well into the 1970's where a 1975 survey found that the "average coder in 14 large organizations had two years of college education and two years of software experience; was familiar with two programming languages and software products; and was generally sloppy, inflexible, 'in over his head', and undermanaged". [21] Of course, how much that statement represents reality and how much it represents bias is impossible to tell 40 years later. But it gives a perspective on the issues of the time.

The Development Process

Because of the type of Development Environment and Developers that were common at the time, it should be no surprise that the code and fix style of programming was the most common process during the 1960's. The developers were often very creative, but the process of coding, then fixing bugs, then coding more, then fixing more bugs, tended to result in heavily patched spaghetti code. [21]

The concepts of structured programming such as the use of sub-routines, block structures and loops [22] only started to become common in the late-1960's and even then there was resistance among many programmers to efforts to make code more readable, easy to maintain and fix.

The Cost Centers

A critical difference between modern development and what the situation was in the 1950’s, 1960’s, and 1970’s was in the cost centers. In the current environment computing resources are so cheap and plentiful that the human costs of developers, project staff, and related personal far outweigh the cost of any computing resources that are consumed.

However, the situation was once the exact opposite. As Barry Boehm relates, “On my first day on the job [in the 1950’s], my supervisor showed me a GD ERA 1103 computer, which filled a large room. He said, 'Now listen. We are paying $600 an hour for this computer and $2 an hour for you, and I want you to act accordingly'.”

Bennington provides similar information in his paper in which he states that the cost for a full-time engineering man-year (with all overhead) came to roughly $1.50 an hour, whereas computer time was billed at a cost of $500 an hour. [1]

While the cost structure eventually reached its current state of human costs being the vast majority, I would guess that computing resources were more expensive at least into the mid-1980’s.

What this cost structure meant was that there was an effort to do as much analysis, coding, and quality testing offline as possible. Computing time was simply too expensive to be used for things that were done much more cheaply by humans without the use of a computer.

Project Scales

The last contextual factor to consider in the development of Waterfall was the project scales that were being undertaken (relative to other work of the era). As Royce states in the first paragraph of his paper, he had spent the previous nine years “mostly concerned with the development of software packages for spacecraft mission planning, commanding and post-flight analysis”. Like Barry Boehm and many of other major figures in software development at the time, he had spent most of his time working on major software development projects for the U.S. government. Mainly the Department of Defense and NASA.

This means nearly all of his work involved the massive complexities of government bidding and contracting, the frequent use of many sub-contractors, and often massive systems integration challenges. Many (or most) of these initiatives would be made up multiple (or a great many) brand new non-computing hardware components (radars, rockets, etc.), integrated with cutting edge computer hardware and software, with dozens or hundreds of sub-systems that had to communicate together properly. It seems not uncommon for there to have been thousands of individual developers working on different aspects of these efforts. In Royce’s case he had the additional complexity of working on systems that would be going into space where the margin for error is essentially zero.

Conclusions

Given the situation context provided above, it should be no surprise to readers that Royce would discuss a process that was focused on the following:

- Cost Control – Doing as much without computing resources as possible

- Quality Management – Using as many processes as possible to ensure that the solution delivered is of the highest quality possible (thus documentation, documentation, documentation). One particularly important aspect of this was the use of documentation as a coordination artifact.
- Project Management – As can be seen later in this document, Royce was very concerned with the project management, support, and maintenance aspects of the software lifecycle, not just the actual coding work. Thus, his ideas were focused more on supporting those efforts than the focus of someone who was mostly concerned with development would be.

Origins

The origins of the Waterfall model are usually ascribed to papers by Herbert D. Bennington in 1956 [1] and Winston W. Royce in 1970 [2] and the blame for the wide-spread adoption of the 'frozen' Waterfall methodology is usually attributed to its adoption by the US Department of Defense with DOD Standard 2167, the MILITARY STANDARD: DEFENSE SYSTEM SOFTWARE DEVELOPMENT, published on June 4, 1985. [33]

However, neither author uses the term Waterfall in their papers anywhere, nor are the processes they discuss nearly as rigid as the 'frozen' interpretation of waterfall.

The Bennington Paper

Herbert D. Bennington presented a paper at a symposium on advanced programming methods sponsored by the Navy Mathematical Computing Advisory Panel in June 1956 in which he described the techniques used to produce the programs for the Semi-Automatic Ground Environment (SAGE) system that was developed for the U.S. government. This was the first U.S. air defense system, and its scope was the entire North American continent. It's cost, both in funding and number of personnel involved, exceeded the Manhatten Project. [40] The MIT Lincoln Laboratory was established to do the systems engineering work (and would eventually be spun out by MIT to become the MITRE corporation [41]), and the System Development Division (later System Development Corporation) established by the RAND corporation to do the software development (h/t to Dennis Asaka for pointing out the role that the Systems Development Division played to me).

To give some additional context on the pioneering work involved in the SAGE project, the following quote is taken from the SAGE page in the history section of the Lincoln Laboratory web site at MIT.

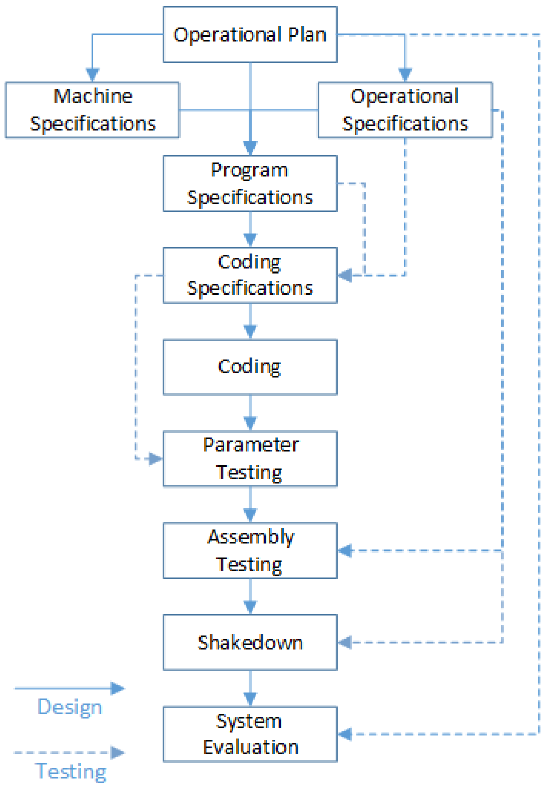
> *In addition to the computer hardware, a large part of the air defense project effort was devoted to software development. The software task developed quickly into the largest real-time control program ever coded, and all the coding had to be done in machine language since higher-order languages did not yet exist. Furthermore, the code had to be assembled, checked, and realistically tested on a one-of-a-kind computer that was a shared test bed for software development, hardware development, demonstrations for visiting officials, and training of the first crew of Air Force operators.*
>
> *Computer software was in an embryonic state at the beginning of the SAGE effort. In fact, the art of computer programming was essentially invented for SAGE. Among the innovations*

was more efficient programming (both in terms of computer run time and memory utilization), which was achieved through the use of generalized subroutines and which allowed the elimination of a one-to-one correspondence between the functions being carried out and the computer code performing these functions. A new concept, the central service (or bookkeeping) subprogram, was introduced.

Documentation procedures provided a detailed record of system operations and demonstrated the importance of system documentation. Checkout was made immensely faster and easier with utility subprograms that helped locate program errors. These general-purpose subprograms served, in effect, as the first computer compiler. The size of the program — 25,000 instructions — was extraordinary for 1955; it was the first large, fully integrated digital computer program developed for any application. Whirlwind was equal to the task: between June and November 1955, the computer operated on a 24-hour, 7-day schedule with 97.8 percent reliability.

Because of the complexity of the software, Lincoln Laboratory became one of the first institutions to enforce rigid documentation procedures. The software creation process included flow charts, program listings, parameter and assembly test specifications, system and program operating manuals, and operational, program, and coding specifications. About one-quarter of the instructions supported operational air defense missions. The remainder of the code was used to help generate programs, to test systems, to document the process, and to support the managerial and analytic chores essential to good software.

Bennington's paper was the first that I can find that described a phased development process that is similar to what eventually became the Waterfall model. And Barry Boehm seems to agree when he called Bennington's process the 'Stagewise' model [26] and said that it was the first to say that software should be implemented in successive stages. Boehm then went on to say that the 'Waterfall' model was a "highly influential refinement" of the Stagewise model. [26] The following is a re-creation of a diagram Bennington provided in his paper that described their overall process.



However, before you make the same mistake that others made with Royce's diagrams and think that Bennington was advocating for a rigidly sequential process model, consider the additional information that Bennington provides in his paper.

In describing how they were successful Bennington says: [1]

- "We were all engineers and had been trained to organize our efforts along engineering lines."
- "… to define a system of documentation so that others would know what was being done;"
- "… to define interfaces and police them carefully;"
- **"… to recognize that things would not work well the first, second, or third time"**
- "… to keep a record of everything that really went wrong and to see whether it really got fixed"
- "… we undertook the programming **only after we had assembled an experimental prototype** of 35,000 instructions that performed all of the bare-bones functions of air defense."
- "… producing large computer programs is like raising a family… you have to start out and do it on your own, learn the unique options you have, see what unexpected problems arise, and, with reasonable luck, perform about as well as those who have been doing it forever."

What you see above is the foundations of a structured, planned approach to software development. But you also see that ideas such as building prototypes were being used as far back as the 1950's. And just because they may not be explicitly shown in diagrams does not mean they were not being done.

One interesting point of the Bennington paper is that it points at why so much effort was placed on documentation by those who successfully worked on large computer programs (at least at that time). He said,

> *Finally, there is the problem of documentation. In the early days of programming you could call up the programmer if the machine stopped. You seldom modified another's program – you wrote your own. Although present automatic programming technology has done much to make programs more communicable among programmers, there is a long way to go before we can take an integrated program of 100,000 instructions and make it "public property" for the user, the coder, the tester, the evaluator, and the on-site maintenance programmer. The only answer seems to be the documentation of the system on every level from sales brochures for management to instruction listings for maintenance engineers.*

This emphasis that documentation serves a purpose far beyond use by the coder is an important one that Royce will repeat and which continues to be overlooked in many discussions today.

The Royce Paper

The second paper, and the one most frequently cited as the source for the Waterfall methodology, is the paper by Winston W. Royce in 1970. Royce starts the paper off by declaring that it "describes my personal views about managing larger software developments" and that "I have become prejudiced by my experiences and I am going to relate some of these prejudices in this presentation."

So it should be clear right off the bat that Royce was not saying all programming efforts should be done the way he discusses. He says his discussion is focused on "larger software developments". He also makes clear that he is presenting his prejudiced opinions.

The 'Common' Activities of Any Programming Effort

Royce starts out by saying that there are two steps common to all programming efforts, regardless of size or complexity. These are the Analysis and Coding steps, which are shown Figure 1 of his paper (recreated below):

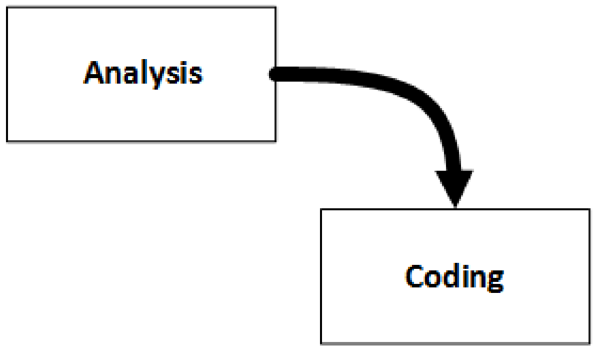


Figure 1

Royce says that these two steps are all that is needed if the effort is sufficiently small and if the final product is to be operated by those who built it. [2] However, he then goes on to say that:

> *"An implementation plan to manufacture larger software systems, and keyed only to these steps, however, is doomed to failure. Many additional development steps are required, none contribute as directly to the final product as analysis and coding, and all drive up the development costs. Customer personnel would rather not pay for them and development personnel would rather not implement them.* ***The prime function of management is to sell these concepts to both groups and then enforce compliance on the part of development personnel****."* [2] *[Emphasis mine]*

Given the statement bolded above, it should be immediately clear that Royce is discussing a software development model that has a [project] management perspective. He's not trying to optimize developer happiness, customer happiness, or drive-down costs. His focus is on what he considers critical for the success of large system development efforts.

The additional steps Royce feels are necessary for a large effort are then shown in Figure 2 of his paper. And this also seems to be where most of the confusion over what Waterfall means comes from. That diagram [Figure 2] is recreated [with color added] below:

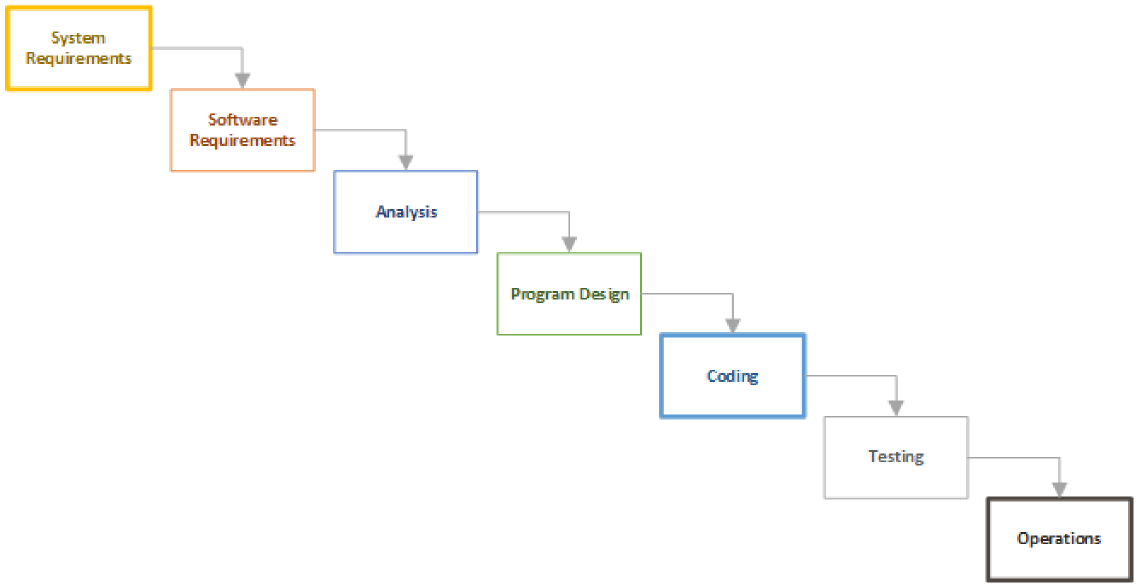


Figure 2

Royce is using his figure 2 (above) as an expansion of the simple two-step process he showed in Figure 1 by adding the additional steps he states are necessary. He describes it as:

> *"A more grandiose approach to software development is illustrated in Figure 2. The analysis and coding steps are still in the picture, but they are preceded by two levels of requirements analysis, are separated by a program design step, and followed by a testing step. These additions are treated separately from analysis and coding because they are distinctly different in the way they are executed. They must be planned and staffed differently for best utilization of program resources."* [2]

So far this still sounds like the commonly assigned characteristics of "Waterfall". But that idea completely ignores the very next line after the quote above and its accompanying figure. The next paragraph immediately after the quote above is:

> *"Figure 3 portrays the iterative relationship between successive development phases for this scheme. The ordering of steps is based on the following concept: that as each step progresses and the design is further detailed, there is an iteration with the preceding and succeeding steps but rarely with the more remote steps in the sequence. The virtue of all of this is that as the design proceeds the change process is scoped down to manageable limits. At any point in the design process after the requirements analysis is completed there exists a firm and closeup, moving baseline to which to return in the event of unforeseen design difficulties. What we have is an effective fallback position that tends to maximize the extent of early work that is salvageable and preserved."* [2] [emphasis mine]

So Royce was saying that the process he is discussing is an iterative process and, by context, that the first diagram (Figure 2) is just a representation of the overall process flow. He then goes on to show Figure 3 (as indicated in the quote) which does include the iterative nature in the diagram. That diagram (his Figure 3) is recreated below:

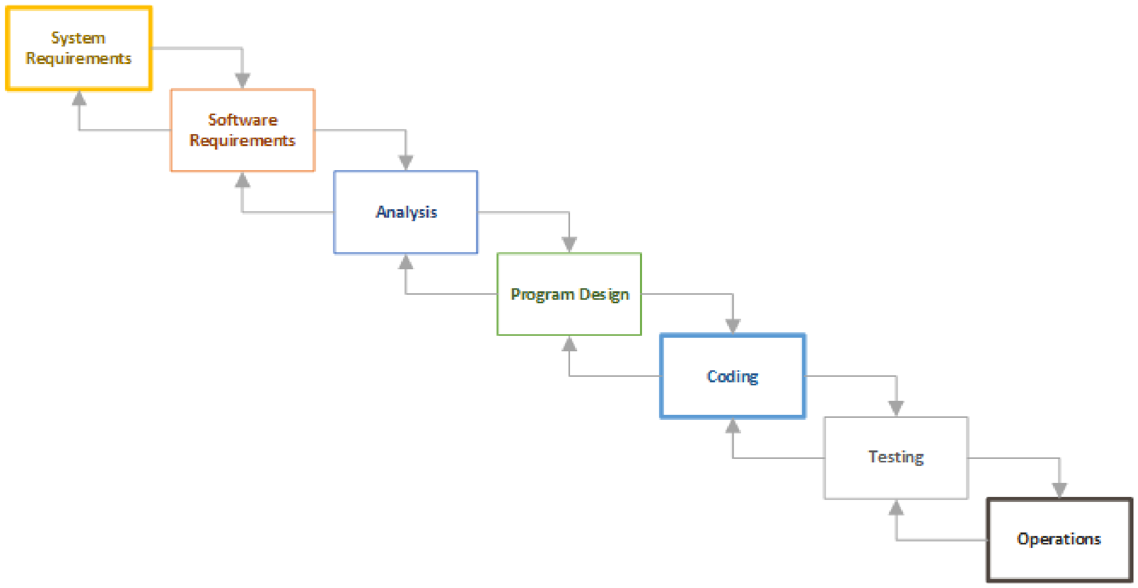


Figure 3

Figure 3 above is probably the closest definition to a ‘traditional development model’ that I can find, if you consider ‘traditional development’ to be whatever Royce was building upon. Or if you consider ‘Waterfall’ to be what Royce was criticizing, Figure 3 above would be the correct model to use, because it is this figure [Figure 3] that forms the baseline process that Royce wants to improve.

However, it should be made clear that Royce is NOT saying this model [Figure 3] is not a good one to use, or should never be used. This is shown later on the same page of his paper when Royce states:

> *However, **I believe the illustrated approach to be fundamentally sound**. The remainder of this discussion presents five additional features that must be added to this basic approach to eliminate most of the development risks.* [2] *[emphasis mine]*

So Royce is stating that the basic process he has described to this point [which is what is shown in Figure 3, NOT what is shown in Figure 2] is fundamentally sound. But that it can be improved by the additional steps he is recommending.

Royce begins his criticisms by stating:

> *I believe in this concept, but the implementation described above is risky and invites failure. The problem is illustrated in Figure 4.* [2]

I have re-created Figure 4 from the Royce paper below.

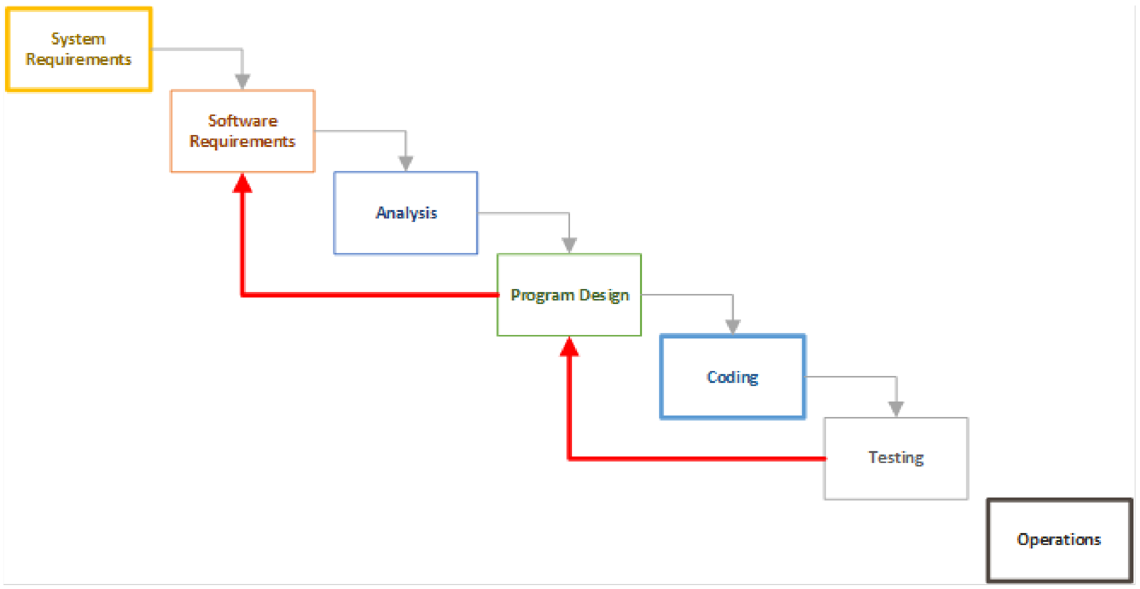


Figure 4

After referring to Figure 4, Royce goes on to say:

> *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 as distinguished from analyzed. These phenomena are not precisely analyzable. They are not the solutions to the standard partial differential equations of mathematical physics for instance. Yet if these phenomena fail to satisfy the various external constraints, then invariably a major redesign is required.** A simple octal patch or redo of some isolated code will not fix these kinds of difficulties. The required design changes are likely to be so disruptive that the software requirements upon which the design is based and which provides the rationale for everything are violated. Either the requirements must be modified, or a substantial change in the design is required. In effect the development process has returned to the origin and one can expect up to a l00-percent overrun in schedule and/or costs.* [2] *[emphasis mine]*

So Royce is NOT saying that it is impossible to define functional requirements ahead of time, as some have posited. Rather, he is pointing to the difficulties of getting the functional needs of the customer to operate in an acceptable manner within the system constraints as they were originally defined in the system requirements. I think in part this needs to be evaluated based on the more constrained computing resources available at the time and trying to get large and complex software to execute with stability and speed within those constrained system environments. This was no doubt further influenced by the fact that Royce was operating on space systems where issues such as power use, memory, and system reliability were all major constraints.

This emphasis on the system design requirements is further revealed in yet another quote from page 2 of his paper, in which Royce says:

> *In my experience there are whole departments consumed with the analysis of orbit mechanics, spacecraft attitude determination, mathematical optimization of payload activity and so forth, but when these departments have completed their difficult and complex work, the resultant program steps involve a few lines of serial arithmetic code. If in the execution of their difficult and complex work the analysts have made a mistake, the*

> *correction is invariably implemented by a minor change in the code with no disruptive feedback into the other developmental bases.* [2]

So while Royce acknowledges that getting the customer (or functional) requirements correct is a difficult and complex task (so much so he says it twice), his focus in this aspect is on getting system design requirements correct. I don't necessarily agree that his statements about the relative ease of implementing (functional) requirements changes are still true, but I am trying to show the context in which he made his statements.

His concern with system requirements is further shown when he discusses the magnitude of potential disruption caused by missed system requirements by stating:

> *The required design changes are likely to be so disruptive that the software requirements upon which the design is based and which provides the rationale for everything are violated. Either the requirements must be modified, or a substantial change in the design is required. In effect the development process has returned to the origin and one can expect up to a lO0-percent overrun in schedule and/or costs.* [2]

So the point he seems to be making is that if the system design (the system requirements) isn't sufficient to meet the software requirements (the customer or functional requirements AND non-functional requirements), that either the entire system design must be re-architected or the software capabilities must be changed (reduced) so as to enable the system design to run the software.

While I personally agree that it's probably not possible to define requirements perfectly ahead of time for large software products, using the quote above to criticize the creation of functional requirements misses the point that Royce was raising.

More broadly, Royce is also making his first major criticism of the process he has described up to this point by pointing out that its scope for iteration is too narrow. With iterations only occurring (generally) between each immediately prior and successive phase, he thinks this is too narrow and risky. He feels this may limit the processes ability to react to issues discovered in one phase may require a larger change than to either the immediately prior or successor phases.

Royce's Five Fixes #1 – Program Design Comes First

At this point Royce goes about offering his suggestions for how to fix the original process through adding 5 additional steps. The first step was Program Design Comes First, about which he says:

> *A preliminary program design phase has been inserted between the software requirements generation phase and the analysis phase. This procedure can be criticized on the basis that the program designer is forced to design in the relative vacuum of initial software requirements without any existing analysis. As a result, his preliminary design may be substantially in error as compared to his design if he were to wait until the analysis was complete. This criticism is correct but it misses the point. By this technique the program designer assures that the software will not fail because of storage, timing, and data flux reasons. As the analysis proceeds in the succeeding phase the program designer must impose on the analyst the storage, timing, and operational constraints in such a way that he senses the consequence….. If the total resources to be applied are insufficient or if the embryo operational design is wrong it will be recognized at this earlier stage and the*

> *iteration with requirements and preliminary design can be redone before final design, coding and test commences.* [2]] *[emphasis mine]*

He provides Figure 5 in his document (recreated below) to show this change.

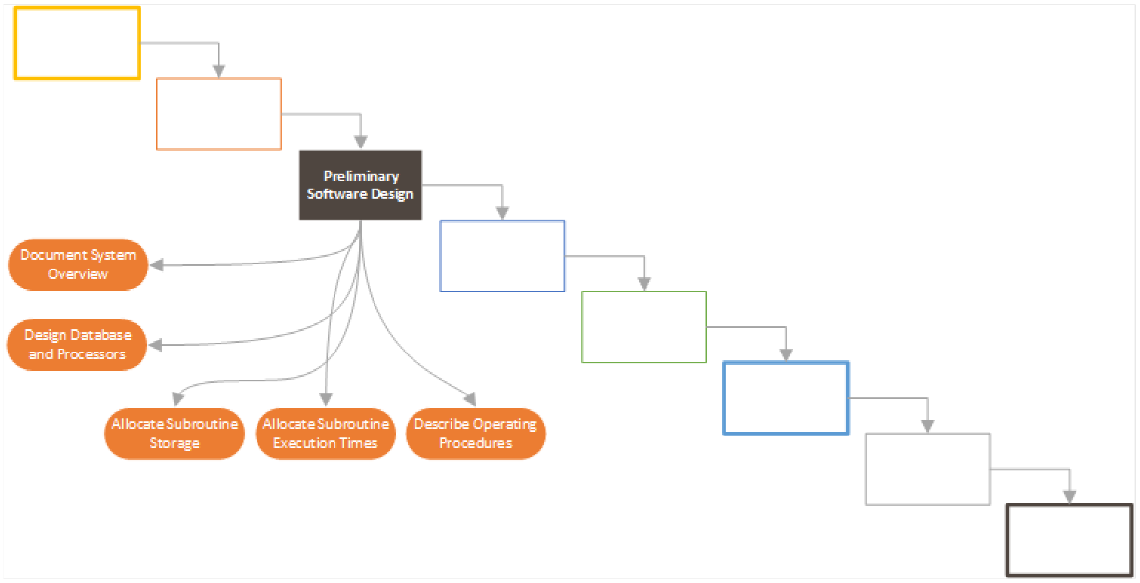

Preliminary Software Design
Document System Overview
Design Database and Processors
Allocate Subroutine Storage
Allocate Subroutine Execution Times
Describe Operating Procedures

Figure 5

Again, it is important to note that even with his fix, Royce is staying with essentially the same Waterfall model that he started with.

Royce's Five Fixes #2 – Document the Design

The second step Royce recommends as part of his fix is to **Document the Design**. A focus on documentation is often flagged as one of the major faults of Waterfall, so it might be useful to understand why Royce was such an ardent fan of documentation. In his words:

> *"At this point it is appropriate to raise the issue of – 'how much documentation'? My own view is 'quite a lot', certainly more than most programmers, analysts, or program designers are willing to do if left up to their own devices.* ***The first rule of managing software development is ruthless enforcement of documentation requirements****.*
>
> *Occasionally I am called upon to review the progress of other software design efforts. My first step is to investigate the state of the documentation.* ***If the documentation is in serious default my first recommendation is simple. Replace project management****. Stop all activities not related to documentation. Bring the documentation up to reasonable standards.* ***Management of software is simply impossible without a very high degree of documentation"****.* [2] *[All emphasis mine]*

Notice that the emphasis Royce is placing on documentation is not for the purpose of helping the developers. He comes right out and says that the level of documentation that should be present is more than most programmers, analysts, or program designers are willing to do if left up to their own devices. The purpose of documentation is for the management of the software lifecycle.

Royce then discusses why so much documentation should be provided and why it is so important during the design phase of a software development effort. As he says:

> *Each designer must communicate with interfacing designers, with his management and possibly with the customer.* ***A verbal record is too intangible to provide an adequate basis for an interface or management decision****. An acceptable written description forces the designer to take an unequivocal position and provide tangible evidence of completion****. It prevents the designer from hiding behind the – 'I am 90 percent finished' - syndrome month after month.*** *[2]*
>
> *During the early phase of software development the documentation is the specification and is the design.* ***Until coding begins these three nouns (documentation, specification, design) denote a single thing. If the documentation is bad, the design is bad. If the documentation does not yet exist there is as yet no design, only people thinking and talking about the design*** *which is of value, but not much. [2]*

Royce also makes the point that documentation serves important purposes far beyond the design stage. Indeed, he says:

> *"****The real monetary value of good documentation begins downstream of the development process during the testing phase and continues through operations and redesign****. The value of documentation can be described in terms of three concrete, tangible situations that every program manager faces. [2]*
>
> *During the testing phase,* ***with good documentation the manager can concentrate personnel on the mistakes in the program. Without good documentation, every mistake … is analyzed by one man who probably made the mistake in the first place*** *as he is the only man who understands the program area.*
>
> *During the operational phase, with good documentation the manager can use operation-oriented personnel to operate the program and to do a better job, cheaper. Without good documentation the software must be operated by those who built it. … It should be pointed out that in this connection that in an operational situation****, if there is a hangup the software is always blamed first. In order to either absolve the software or fix the blame, the software documentation must speak clearly****.*
>
> *Following initial operations, when system improvements are in order****, good documentation permits effective redesign, updating and retrofitting*** *in the field. If documentation does not exist, generally the entire existing framework of operating software must be junked, even for relatively modest changes".*

Royce's Five Fixes #3 – Do It Twice

Royce's third fix was to **Do It Twice**. Or essentially, start off with prototype when building entirely new software. Royce states:

> *If the computer program in question is being developed for the first time, arrange matters so that the version finally delivered to the customer for operational deployment is actually the second version* ***insofar as critical design / operational areas are concerned. …*** *The point of all of this, as with a simulation, is that questions of timing, storage, etc. which are otherwise*

> *matters of judgment, can now be studied with precision. Without this simulation the project manager is at the mercy of human judgment.* [2]

Note that even with this change Royce does not seem to be saying that every last detail of the program must be finalized and irrevocable. The purpose of this simulation is to validate what are commonly called non-functional requirements. Does the designed architecture support the levels of performance necessary? Does the system have enough memory, enough storage, and enough processing power? Can the system design support the operational needs of the customer?

Royce's Five Fixes #4 – Plan, Control and Monitor Testing

Royce's fourth fix was focused on the testing phase. This was the phase of greatest risk and greatest effort. He states, Without question the biggest user of project resources, whether it be manpower, computer time, or management judgment, is the test phase. He even acknowledges that the prior three recommendations are all aimed at uncovering and solving problems before entering the test phase.

Royce has several suggestions for testing, most of which may no longer be true with modern test automation capabilities. For example, one suggestion Royce made was to review every line of code by human eye because using computer time for that is too expensive. However, he does emphasize the benefits of good documentation in this effort when he says:

> *Many parts of the test process are best handled by test specialists who did not necessarily contribute to the original design. … **With good documentation it is feasible to use specialists in software product assurance who will, in my judgment, do a better job of testing than the designer**.* [2]

So again we see Royce's perspective as a software program manager who is trying to control costs AND find ways to ensure the software delivered is of high quality as possible.

Royce's Five Fixes #5 – Involve the Customer

Royce's fifth fix is one that may surprise those who think customers in the Waterfall process are only involved in the requirements and operations phases, with some involvement in the testing phase. As Royce says:

> *For some reason what a software design is going to do is subject to wide interpretation even after previous agreement. It is important to involve the customer in a formal way so that he has committed himself at earlier points before final delivery. **To give the contractor free rein between requirement definition and operation is inviting trouble**.* [2]

Royce then lists specific places in the program effort where the customer should be involved**, at a minimum.** These are:

- During the requirements phase, with customer sign-off of the requirements
- During the preliminary design phase, with customer review and approval of the initial system design
- During the detailed design phase, with the customer review and approval of the detailed design of every system component

- During the final software acceptance reviews

However, it should also be noted that the caption under the diagram that shows the formal recommendations above [Figure 9 in the Royce paper] says ***Involve the customer – the involvement should be formal, in-depth, and continuing***.

Royce's Recommendations – The Final Result

All of the recommendations above are captured in the final diagram of Royce's paper [Figure 10], which show a far more complex process than is usually attributed to Waterfall. This is what I call 'Royce's' Waterfall when I need to separate it from 'Frozen' Waterfall, and is what Royce actually recommends and proposes in his paper:

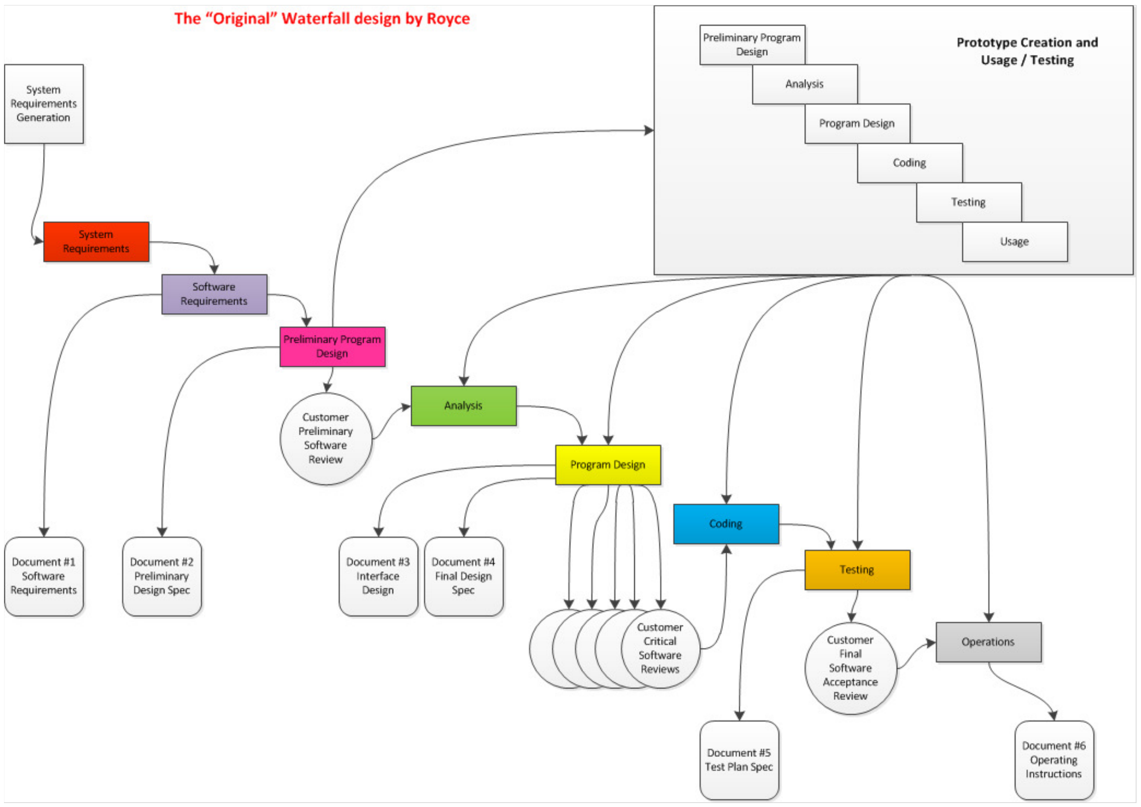

The "Original" Waterfall design by Royce
System Requirements Generation
System Requirements
Software Requirements
Preliminary Program Design
Customer Preliminary Software Review
Analysis
Program Design
Coding
Testing
Operations
Customer Critical Software Reviews
Customer Final Software Acceptance Review
Document #1 Software Requirements
Document #2 Preliminary Design Spec
Document #3 Interface Design
Document #4 Final Design Spec
Document #5 Test Plan Spec
Document #6 Operating Instructions
Prototype Creation and Usage / Testing
Preliminary Program Design
Analysis
Program Design
Coding
Testing
Usage

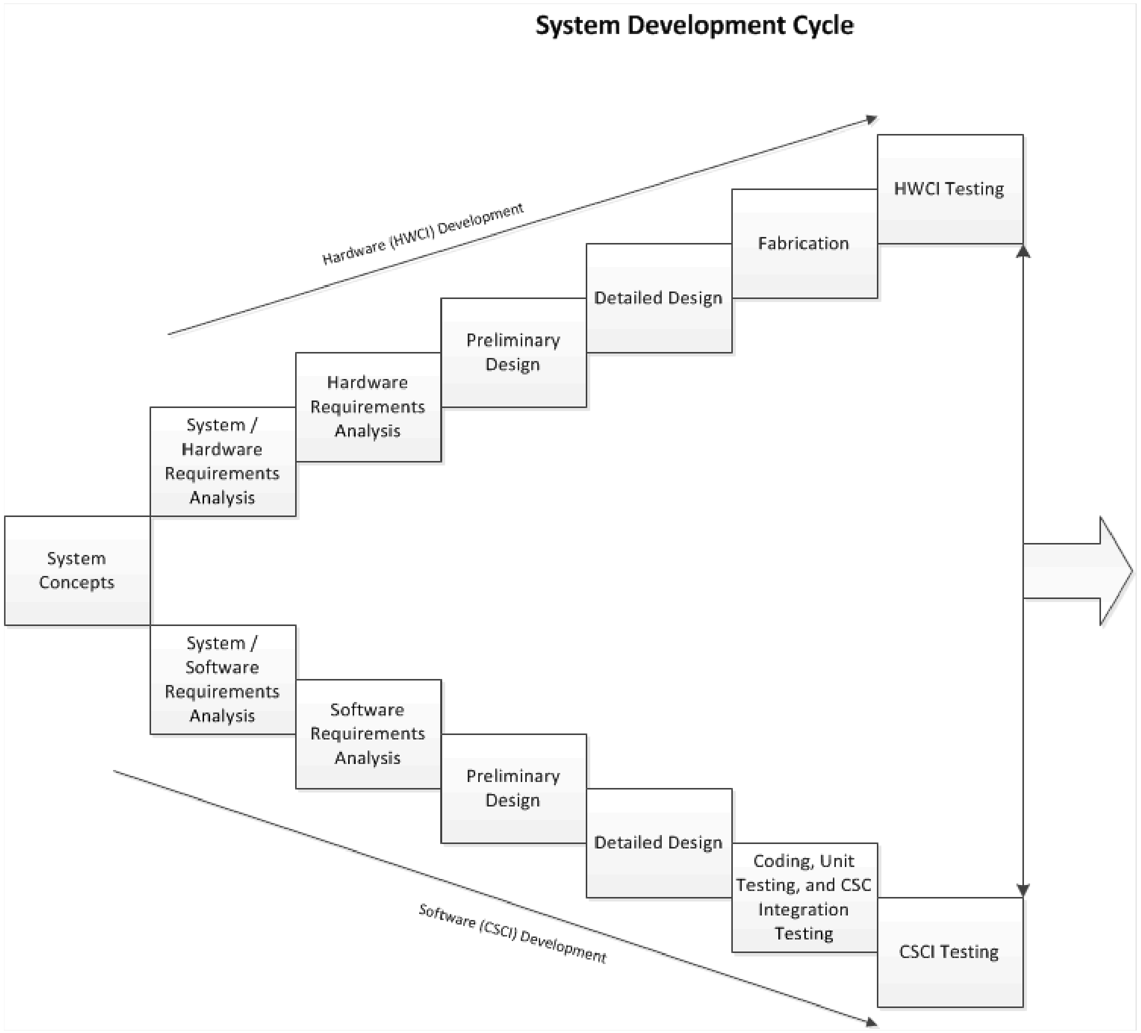
Figure 10

DOD Standard 2167

So if Royce did not describe the 'frozen' Waterfall methodology, where did it come from? The next culprit named is usually the U.S. Department of Defense, which supposedly adopted the 'Waterfall' methodology in DOD Standard 2167 (published 4 June 1985) for the acquisition of Mission Critical Computer Systems. In doing so, it is argued that the DOD forced the 'frozen' Waterfall model upon the rest of the world. [29]

The argument is that the DOD made some slight accommodations to less rigid structure when they adopted the 2167A revision of the standard, but that even that fell short.

And if you look at page 2 of the standard document, it sure looks like the military has adopted the traditional Waterfall model when you see this diagram:


System Development Cycle
Hardware (HWCI) Development
System Concepts
System / Hardware Requirements Analysis
Hardware Requirements Analysis
Preliminary Design
Detailed Design
Fabrication
HWCI Testing
System / Software Requirements Analysis
Software Requirements Analysis
Preliminary Design
Detailed Design
Coding, Unit Testing, and CSC Integration Testing
CSCI Testing
Software (CSCI) Development

The diagram seems to be implying a linear, sequential process, with no iteration and revisiting prior stages.

This potential understanding is not helped if the reader skims through the document looking for the description of the Software Development Cycle, which proscribes the following steps:

1. Software Requirements Analysis. The purpose of Software Requirements Analysis is to completely define and analyze the requirements for the software. These requirements include the functions the software is required to accomplish as part of the system, segment, or prime item. Additionally, the functional interfaces and the necessary design constraints are defined. During Full Scale Development, and Production and Deployment, this phase typically begins with the release of the SSS [System/Segment Specification], Prime Item Specification(s), Critical Item Specification(s), or Preliminary SRS(s) [Software Requirements Specification] and IRS(s) (Interface Requirements Specification), and terminates with the successful accomplishment of the SSR. During this phase, analyses and trade-off studies are performed, and requirements are made definitive. The results of this phase are documented and approved requirements for the software. At the initiation of

Software Requirements Analysis, plans for developing the software are prepared or reviewed (as applicable).

2. Preliminary Design. The purpose of Preliminary Design is to develop a design approach which includes mathematical models, functional flows, and data flows. During this phase various design approaches are considered, analysis and trade-off studies are performed, and design approaches selected. Preliminary Design allocates software requirements to TLCSCs, describes the processing that takes place within each TLCSC [Top-Level Computer Software Component], and establishes the interface relationship between TLCSCs. Design of critical lower-level elements of each CSCI may also be performed. The result of this phase is a documented and approved top-level design of the software. The top-level design is reviewed against the requirements prior to initiating the detailed design phase.
3. Detailed Design. The purpose of Detailed Design is to refine the design approach so that each TLCSC is decomposed into a complete structure of LLCSCs [Lower-Level Computer Software Components] and Units. The detailed design approach is provided in detailed design documents and reviewed against the requirements and top-level design prior to initiating the coding phase.
4. Coding and Unit Testing. The purpose of Coding and Unit Testing is to code and test each Unit of code described in the detailed design documentation. Each Unit of code is reviewed for compliance with the corresponding detailed design description and applicable coding standards prior to establishing internal control of the Unit and releasing it for integration.
5. CSC Integration and Testing. The purpose of CSC [Computer Software Component] Integration Testing is to integrate and test aggregates of coded Units. Integration tests should be performed based on documented integration test plans, test descriptions, and test procedures. CSC Integration test results, and CSCI test plans, descriptions, and procedures for testing the fully implemented software are reviewed prior to the next phase of testing.
6. CSCI Testing. The purpose of CSCI testing is to test the fully implemented CSCI [Computer Software Configuration Item]. Testing during this phase concentrates on showing that the software satisfies its specified requirements. Test results should be reviewed to determine whether the software satisfies its specified requirements.

However, actually reading through the standard begins to immediately show that this assumption is not correct, as the following statements from just the first 4 pages of the standard indicate:

1. This standard is intended to be dynamic and responsive to the rapidly evolving software technology field. As such, this standard should be selectively applied and tailored to fit the unique characteristics of each software acquisition program. (Page iii, item 2)
2. 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]. (Page 1, section 1.2)
3. This standard, or portions thereof, may not apply to small applications which perform a fixed function that is not expected to change for the life of the system. (Page 1, section 1.2.2)
4. Software shall be developed in accordance with this standard to the extent specified in the contract clauses, SOW, and the Contract Data Requirements List. … The contracting agency

will tailor this standard to require only what is needed for each individual acquisition. (Page 4, section. 1.3)

Like Royce, the DOD is looking at the full lifecycle of a piece of software. They envisage the need for developers other than the contractor(s) to maintain and enhance the application being developed, and they see documentation as being a critical support tool for that process.

This is further confirmed later in the document when there are explicit mandates for the contractor to provide standard user instructions (section 5.2.1.8), error identification and diagnostics instructions (section 5.2.1.9), and system life cycle support documentation (5.2.1.10).

But even in this standard, the DOD is not mandating the use of this approach all of the time, even for critical software. Item 4.8 on page 16 makes this clear when it states:

> *The contractor shall use a top-down approach to design, code, integrate and test all CSCIs [Computer Software Configuration Item], **unless specific alternative methodologies have been proposed** in either the SSPM [Software Standards and Procedures Manual] or SDP [Software Development Plan] **and received contracting agency approval**.*

And when it states:

> *The contractor may depart from a top-down approach to: (1) address critical lower-level elements or (2) incorporate commercially available, reusable, and Government furnished software. (Page 28, item 5.3.1.4)*

A more detailed examination would also show that the Software Development Cycle is described as being part of a larger System Development Cycle that is not captured in the diagram shown above. The description the System Development Cycle stages is enlightening. It states:

- 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**. (Page 61, section 20.4)
- **Concept Exploration**. The Concept Exploration Phase is the initial planning period when the technical, strategic, and economic bases are established through comprehensive studies, experimental development, and concept evaluation. This initial planning may be directed toward refining proposed solutions or developing alternative concepts to satisfy a required operational capability.
    - During this phase, proposed solutions are refined or alternative concepts are developed using feasibility assessments, estimates (cost and schedule, intelligence, logistics, etc.), trade-off studies, and analyses. The SSA [Software Support Agency] and user should be involved in these activities.
    - For computer resources, the software development cycle should be tailored for use during this phase and may result in demonstration of critical algorithms, breadboards, etc.
- **Demonstration and Validation**. The Demonstration and Validation Phase is the period **when major system characteristics are refined through studies, system engineering, development**

of preliminary equipment and prototype computer software, and test and evaluation. The objectives are to validate the choice of alternatives and to provide the basis for determining whether or not to proceed into the next phase.

    - During this phase, system requirements, including requirements for computer resources, are further defined, and preferred development methodologies for computer software and data bases are selected. The results of validation activities are used to define the system characteristics (performance, cost, and schedule) and to provide confidence that risks have been resolved or minimized.
    - For computer resources, the software development cycle should be tailored for use during this phase, resulting in prototype software items.

- **Full Scale Development**. The Full Scale Development phase is the period when the system, equipment, computer software, facilities, personnel subsystems, training, and the principal equipment and software items necessary for support are designed, fabricated, tested, and evaluated. **It includes one or more major iterations of the software development cycle**. The intended outputs are a system which closely approximates the production item, the documentation necessary to enter the system's Production and Deployment phase, and the test results that demonstrate that the system to be produced will meet the stated requirements. **During this phase the requirements for additional software items embedded in or associated with the equipment items may be identified**. These requirements may encompass firmware, test equipment, environment simulation, mission support, development support, and many other kinds of software.
    - Software requirements analysis is performed in conjunction with system engineering activities related to equipment preliminary design. SRSs and IRSs for each CSCI are completed and authenticated at the SSR, establishing the Allocated Baseline for each CSCI. Requirements for software that is part of an HWCI [Hardware Configuration Item] may be authenticated during HWCI design reviews. The OCD [Operational Concept Document] is completed and reviewed at the SSR as well.
    - A preliminary design effort is accomplished and results in a design approach. For computer software, preliminary design includes the definition of TLCSCs in terms of functions, external and internal interfaces, storage and timing allocation, operating sequences, and data base design. Detailed design of critical lower-level elements of the CSCI may be performed as well.
- **Production and Deployment**. The Production and Deployment Phase is the combination of two overlapping periods. The production period is from production approval until the last system item is delivered and accepted. The objective is to efficiently produce and deliver effective and supported systems to the user(s). The deployment period commences with delivery of the first operational system item and terminates when the last system items are removed from the operational inventory.
    - **After a system is in operational use, there are a variety of changes that may take place on the hardware items, software items, or both hardware and software items. Changes to software items may be necessary to remove latent errors, enhance operations, further system evolution, adapt to changes in mission requirements, or incorporate knowledge gained from operational use**. Based upon complexity and other factors such as system interfaces, constraints, and priorities, control of the changes may vary from on-site management to complex checks and balances with mandatory security keys and access codes. …… The same six phases of

the software development cycle are utilized for each change during the Production and Deployment phase (see Figure 4)."

Signs that the DOD was not mandating the 'frozen' Waterfall structure are further shown on page 69 of the standard, in section 20.4.5, Software Development Cycle Application and Documentation. This section includes such statements as:

- The software development cycle may span more than one system life cycle phase, or may occur in any one phase.
- The phases in the software development cycle may involve iterations back to previous phases. For example, design may reveal problems which lead to the revision of requirements and re-institution of certain analyses; checkout may reveal errors in design, which in turn may lead to redesign or requirements revision; etc.

So by reading the specification it would seem that what the DOD actually proscribed was a hybrid of incremental and iterative processes, implemented within a version of the more process-controlled Waterfall structure that Royce proposed. It was closer to the strict Waterfall methodology in the design and coding of each phase of a software component, but not of the entire solution.

Note that the DOD standard specifically supports such activities as:

- Multiple iterations
- Prototyping
- User involvement in solution design and specification
- The discovery of new requirements during and after a software cycle
- Testing from very early phases
- And even alternative development methodologies

In the end, I suspect a similar issue occurred with the DOD Standard as with the Royce paper. But rather than people just looking at the pictures (or even just the first picture) and not bothering to read the text; you had a lot of DOD personnel and contractors who looked at the pictures, skimmed a bit of the text, and following the tradition of bureaucratic processes everywhere, implemented it in a rigid and detailed manner as possible in order to CYA (Cover Your Ass for those unfamiliar with the acronym).

The 'Freezing' of the Waterfall

Given the above, the exact origin of the 'frozen' Waterfall interpretation is a bit of a mystery to me [I may not have come across the right reference yet, I fully admit]. I see this methodology described in many places, but I can't find an origin for what is commonly described as Waterfall in any of the papers that are cited as the 'source' of the Waterfall concept.

I would go so far as to say that the highly rigid process commonly described as traditional development, or what I call 'frozen' waterfall, has probably never been practiced since the 1960's or before. Even in 1970 Royce was describing the common software development process of the time as being at least a partially iterative process. I don't know a single Business Analyst who has ever been told to use this exact, highly rigid, methodology [although again, I will admit there is certainly that possibility].

However, I would argue that such rigidness should not be blamed on a methodology that does not seem to exist as more than a theoretical straw man,[9] and that instead the problem in those cases is the people advocating such rigidness.

So that raises the question, how did the common perception of the Waterfall Model as being so incredibly rigid and silo'd come about? Unfortunately, I can't tell you. I do know that by 1988 you can find quotes like this one from academic literature:

> *The waterfall model makes the assumption that all activity of a certain type occurs during the phase of that same name and that phases do not overlap. Thus all requirements for a project occur during the requirements phase; all design activity during the design phase.* [28]

And by 1994 you can find quotes like the one below in academic papers:

> *The so-called Waterfall model has been, until recently, the most frequently used model for controlling and guiding complex software development projects.* ***The basic idea underlying this model is that development proceeds in stages. Each stage must or phase must be finished in its entirety before a new phase can start. Just as water in a waterfall cannot flow back, phases that are finished should not be started again.*** *[27] [Emphasis mine]*

Unfortunately, these just point to time periods where the perception of 'Waterfall' had shifted to the 'Frozen' interpretation, but not the source or reason for that perception. What's funny is that neither of the two papers above cites the Royce paper when talking about 'Waterfall'. Indeed, neither paper cites ANY source for their statements, which indicates to me that they thought their understanding of Waterfall was so commonly understood and agreed upon that no citation was necessary.

The best reason I can find for the emergence of the 'Frozen' Waterfall concept comes from a blog post by Professor David Dischave, in which he recites the following story:

> *My first encounter with the mythical Waterfall methodology was in the early 1980's. As the director of a systems development department at a fortune 100 company I received a visit from a salesman from Coopers & Lybrand (C&L.) As the director of a multi-million dollar IT department you can imagine how many people wanted to sell me stuff. The C&L sales rep, attempting to sell me a methodology called Summit-D, asked me what systems development method we used. So I shared with him that our shop standard was the Systems Development Lifecycle which entails doing a bunch of tasks and activities and we grouped in five phases: planning, analysis, design, implementation and maintenance & support. I went on to say; you know, we subscribe to Winston Royce's work.*
>
> *Before I could finish my description, the C&L sales rep quipped right back with 'ahhh, you are using that old obsolete Waterfall model. Oh, by the way who is Winston Royce?' Not waiting for an answer, he added, nobody should use the Waterfall model anymore. 'You see, Dave, once you complete a phase it is frozen.' I asked, what is this Waterfall thing that you keep referencing? The C&L sales rep said it is the method almost every company uses, where the phases are worked sequentially i.e. in lock step and no phase can start until the previous one finishes). He went on to say that all deliverables produced in a phase were frozen once that phase ended. He said each phase cascades down into the next, you know, like a waterfall. 'See, it is called waterfall because water just can't flow up hill.'*
>
> *You can imagine what I must have been thinking.* ***At that time, I'd been in IT application development for 20 years with five different major corporations and I was introduced to***

waterfall - by a salesman. I had never heard of a waterfall method. In all of the years and all of the places I worked and all of the conferences I had been to, I didn't know of any organization that built systems this way*. Yes, I did try to dissuade the sales rep that we were not using any waterfall, watering hole, water table, water can or water cooler methods* ***but as you can suspect sales folks can't sell you a solution if you don't have a problem and he was really trying to create a problem.****" [10]*

The idea that the common perception of 'Waterfall' as being of the 'Frozen' interpretation could be due to nothing more than sales folks needing a straw-man to compare their 'new and improved' product against doesn't really seem that outrageous to me. But if anyone can provide alternative explanations please let me know.

The Many Misconceptions of Waterfall

IMPORTANT NOTE: This section attempts to address to some of specific misconceptions about Waterfall model that I see out there. Importantly, I equate the Waterfall model with what Royce actually described in his paper, with the possibility of expanding that definition to include what is in DOD-2167. At its most restrictive interpretation, you might equate this to Figure 3 in Royce's paper, or at the wider end you may equate it to Figure 10 in his paper or to the full scope of DOD-2167.

I am going to assume that you have not read the historical information above, and will try to summarize that information where appropriate. However, in some cases where there is too much information to easily summarize, I will simply refer to the information above.

Royce Only Described Waterfall as an Example of What Not to Do

One myth about Waterfall that seems arise frequently in one form or another is that Royce discussed 'Waterfall' only as an example of what not to do, or as a straw-man that he could then tear down. [35] This myth seems to have multiple ideas behind it. Among the most common quotes from Royce cited for this belief are the following: [2]

… *a more grandiose approach to software development is illustrated in Figure 2*

… *the implementation described above is risk and invites failure*

And then you get people who misunderstand [I think] Royce and who state things such as the following:

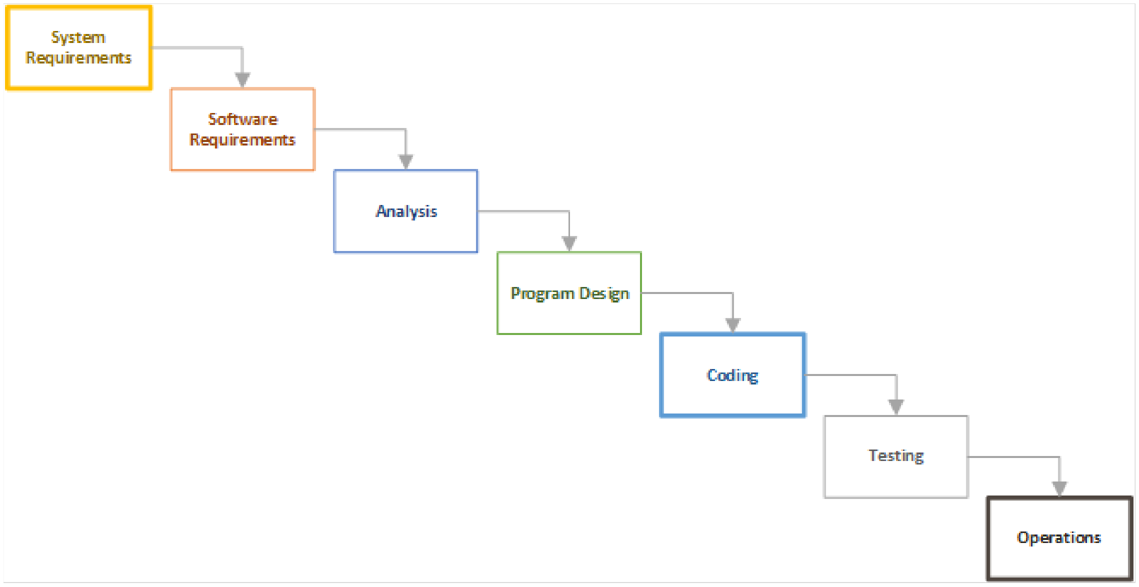
… there's no empirical evidence to back the claim that a linear process works, and actually there's not even a fluffy "i think so" claim to that effect, **but rather Royce considers the idea unworkable**. [31] [emphasis mine]

There are a couple of problems here.

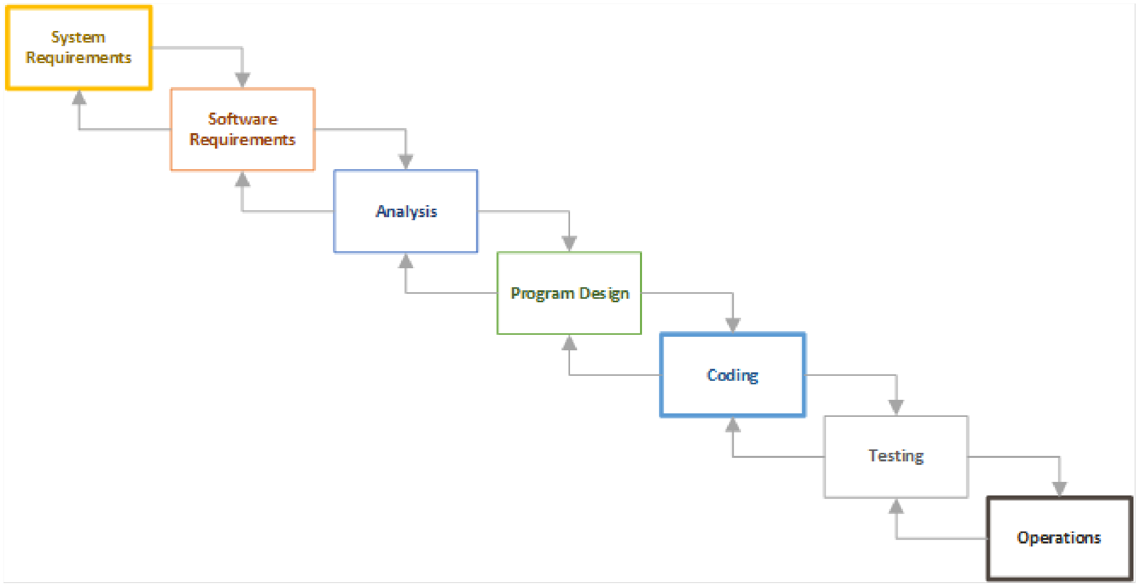
The first problem is the misunderstanding that what many people think of as 'the Waterfall model' isn't what he talked about even in the early part of his paper. It is only an incomplete diagram that is used to provide an overview. This is especially common for people who take the statement *a more grandiose approach to software development is illustrated in Figure 2*, and equate Figure 2 with the Waterfall model that his paper was discussing.

The second problem is the belief that Royce was describing a linear process at all.

Part of the problem is that Figure 2 IS the model that most people think was what Royce was describing in his overall paper. Here is Figure 2 for reference:



But that's not the specific process Royce is discussing for most of the paper. It's not even the process he criticizes in the paper. It's just a high-level overview (a 'grandiose' view). This is made clear by the start of the paragraph directly after the quote above, in which he says Figure 3 portrays the iterative relationship between successive development phases for this scheme. Note that this scheme refers to Figure 2, and Figure 3 is showing the iterative nature of Figure 2. And here is Figure 3 for reference:



It is about Figure 3 that Royce states I believe in this process, *but the implementation described above is risky and invites failure*. However, he then goes on to discuss the testing phase exposing issues such as timing, storage, input/output transfers, etc. [2] You'll note that these aren't standard 'software requirements'. Rather they fall into what we commonly call 'non-functional requirements' and which are highly-dependent on the system design. As Royce says, *These phenomena are not precisely analyzable*.

So right there I think we can dismiss the idea that Royce was only describing something he thought would fail or that was unworkable. This is further supported by the start of the second paragraph after the sentence quoted above, which states:

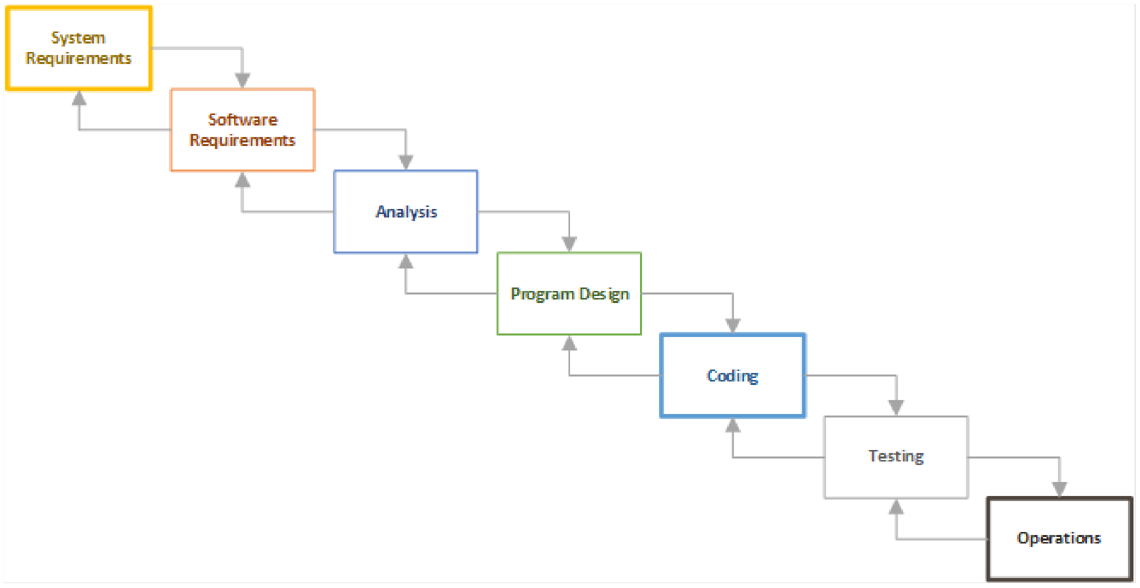
> *However, I believe the illustrated approach to be fundamentally sound. The reminder of this discussion presents five additional features that must be added to this basic approach to eliminate most of the development risks.* [2]

Royce then goes on to suggest 5 enhancements for the basic process that his describes in Figure 3, and which are all documented above.

Royce / Boehm Was Responsible for Adding Iterative Features to the Base Waterfall Model

Another misconception I've seen in a few places is that either Royce [32] or Boehm added iterative features to the standard Waterfall model. This misconception usually refers to a figure similar to Figure 3 from Royce's paper, which is displayed below.

However, as stated above, Royce says Figure 3 portrays the iterative relationship between successive development phases for this scheme. Note that this scheme refers to Figure 2 which was most people think of as the Waterfall model, and Figure 3 is showing the iterative nature of Figure 2.



So the 'Waterfall Model' included these features from the beginning and they actually pre-date Royce.

Royce Only Intended the Waterfall Model to Be Used For The Simplest of Efforts

This misconception is directly contradicted in the very first line of the Royce paper when he states that he is providing his personal views on managing large software developments. Indeed, he clearly states that for the simplest of efforts, you only need the Analysis and Coding steps that are shown in

Figure 1 of his paper. The whole point of both the Royce and Bennington papers was that they were targeted at very complex, very large development efforts.

Waterfall Means Never Working Iteratively

As stated in DOD 2167, 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. [3]

The DOD's system life cycle fully envisions multiple tracks of concurrent and iterative development. Royce's Waterfall was definitely less explicitly iterative, but its specific inclusion of prototypes and iterations back to prior steps (both the immediately prior stage and larger steps back) would indicate that iterative development was at least minimally supported, if not fully expected.

Waterfall Means Doing All Development at Once

Royce's Waterfall model included the option of iterating back to prior steps in the process. Most commonly the immediately-prior step, but also further back. So while in general most development would be done in one large stage, it allowed for options to do it later. Also, the use of a fully developed prototype by Royce allows for early development to be done, especially in the area of confirming system design options. Indeed, this was the main point of the prototype step envisioned by Royce.

This was further emphasized by the DOD in standard 2167 when it said, 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. [3]

Waterfall Means No Customer Involvement After Requirements

A common misconception is something like Customers don't provide any input until the solution is delivered or in testing. [4] This simply isn't true.

According to Royce customers must provide input to the requirements; sign off on the baseline requirements (which are fully documented); review and approve the preliminary technical design (also fully documented); review and approve the interface design (again, fully documented and which would presumably include wireframes and similar visual models); review and approve the detailed program designs (multiple reviews per Royce's model) before ANY code is generated; interact with and review a prototype (if created); review the test plans; and finally, test and approve the working code. This is hardly consistent with the idea of no involvement after requirements.

But that is just customer involvement in documentation and design plans. Royce espouses the use of an initial prototype in his document. But modern prototyping and simulation tools (such as iRise as an example) didn't exist then. With tools like those and the use of Model-View-Controller (MVC) design methods, there is nothing in Waterfall that prevents users from being heavily involved in the design and implementation of the user interface long before the rest of the software is 'done'.

Waterfall Means Setting Impossible Deadlines

This is a myth I've found associated to Waterfall that baffles me. Nowhere in any of the 'source' documents for Waterfall is there any discussion of setting deadlines, rigid schedules, or even specifying the exact work that has to be done to implement a software program (the detailed design) early in the process. Not in Bennington, not in Royce, not in DOD-2167. But I see references to Waterfall including some assumption of an ability to reliably predict when we are going to deliver a system [36] or something to that effect.

Yes, Waterfall is a 'planned' methodology. But planning does not mean setting deadlines. All of the planning discussed in the sources I reference above is about planning features, requirements, designs, and system resources. It's about understanding what you need, as much as possible, and then creating a plan for software that fulfills those needs.

All three main sources indicate that there will be changes needed even after a software program goes into operations. All three state that there is significant unpredictability in developing software.

Deadlines are no more an aspect of Waterfall than they are of 'Agile' or any other development methodology. They are a factor of management and markets. Nothing more.

Waterfall Means that Time from Specification to Delivery is Great

Another misconception is that Waterfall is that it often took a long time between the requirement phase and the user feedback phase, [33] entailing months or more of delay from specification until working software is delivered. Or as Scott Ambler put it:

> *The period between the time that the requirements are 'finalized' to the time that the system is actually deployed will often span months, if not years. During this timeframe changes will occur in the marketplace, legislation will change, and your organization's strategy will change. All of these changes in turn will motivate true changes to your requirements.* [40]

This is an issue of scope, rather than methodology. There is absolutely nothing in the overall Waterfall concept that prevents chunking the business needs into smaller pieces, documenting them as discreet requirements, and delivering those smaller chunks rapidly as multiple, sequential Waterfall efforts. That is literally what the original iterative methodology concept was. Essentially using the Program of multiple Projects approach to deliver multiple chunks of an overall solution in smaller, more rapid phases. Indeed, this is exactly what DOD-2167 does with its larger System Lifecycle and smaller Software Lifecycle.

There is also nothing in Waterfall that prevents changes to the requirements or design as the solution moves forward. Discover a new requirement? Evaluate it and if it's worth adding, start the change management process to make the appropriate changes.

The Author(s) of DOD-2167 Had Never Heard of Iterative Development

A fairly common misconception is that the author(s) of the DOD-2167 standard had never heard of, or weren't familiar with, Iterative Development. In its more prosaic form this myth gets stated like this:

Source: http://www.bawiki.com/wiki/Waterfall.html Downloaded 1st February 2024

> *But, things change as a busy engineer in the US defense organization is asked to come up with a standard for military grade software projects. He doesn't know what a good process would be, and he's told that 'Hey, Royce already came up with the correct method: the waterfall. Use it.' So the waterfall becomes the US military standard DoD-2167.* [15]

The main source of this seems to be a book by Craig Larson in which he states that the primary author of the standard told him:

> *He was not familiar with the practice of timeboxed iterative development and evolutionary requirements at the time.* [30]

I can't identify the author(s) from the specification, and Mr. Larson does not identify his source in his book, so I have no way to further research this.

However, in looking at the standard itself this seems extremely implausible. Consider that the 3rd sentence (and start of the 2nd paragraph) of DOD-2167 says the following:

> *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.*

I will grant that the time-boxed part that statement is probably true. And possibly even the evolutionary part, if not under that name. However, given the actual text of DOD-2167 it seems less likely that the author(s) were not familiar with the concept of evolutionary development. Although again, perhaps not under that name.

Add in that it seems unlikely that the DOD would rely on a single person as the main author of such a standard when they would presumably have had access to a wealth of highly-knowledgeable resources such Barry Boehm and others who had years of experience (if not decades) in building large systems for government contractors like TRW. Or that the military (and US Government in general) tends to turn towards think-tanks like Rand or government contractors like IBM to provide model standards for things like software development.

In the end, I simply can't find any plausible information that would suggest this myth is true.

Slow Reaction and High Cost of Change

Another misconception is that Waterfall must be slow to react to change and that the cost of change is great. While a formal change control process means that Waterfall may not react to change immediately, it does mean that change is at least evaluated before it is made. Because change can be made at any point, even before coding has begun, I would argue that change is potentially cheaper in Waterfall (in some instances) because change can be made before any development (with its high modern cost) has been done.

With agile, change is usually only identified after some coding has been done. While this might make some changes easier to identify (as there is working code), it also means you have already paid development staff to create that software. And in the modern cost structure of software development, development work is usually among the highest-cost activities that are undertaken.

Yes, the change may be smaller since in theory the sprint is working on a smaller functional base, but that does not mean that a change will not require significant architecture changes that mandate the re-work of major portions of prior sprints. Indeed, in my experience the need for significant architectural change (and its associated delays and costs) is more common among large-scale 'agile' projects than among large-scale 'waterfall' projects.

Is the 'Waterfall' Name Permanently Tainted?

Despite all the information above, I'm sure that the term 'Waterfall' will continue to be associated with the 'Frozen' interpretation that has now become the common understanding in software circles of what 'Waterfall' means. This is regrettable because Winston Royce's name is still commonly associated with that interpretation, despite it having virtually no connection to what his paper described.

It also seems increasingly anathema to many in the software world to speak positively of planned development methodologies like Royce described, or even later evolutions of the concept. Trying to say anything good about 'Waterfall' on an internet discussion forum will generally make you a target for derision, scorn, or outright hostility. So what do you do if you want to talk about planned development but find the term 'Waterfall' has become too toxic to have meaningful conversations about?

Some might use the term Big Development Up Front,[38] but that seems to have become just a synonym for 'Waterfall' because it assumes the programs design is to be completed and perfected before that programs implementation is started. [39]

The best option I have found are terms like Planned Development which the IIBA seems to have adopted for all non-agile methodologies.

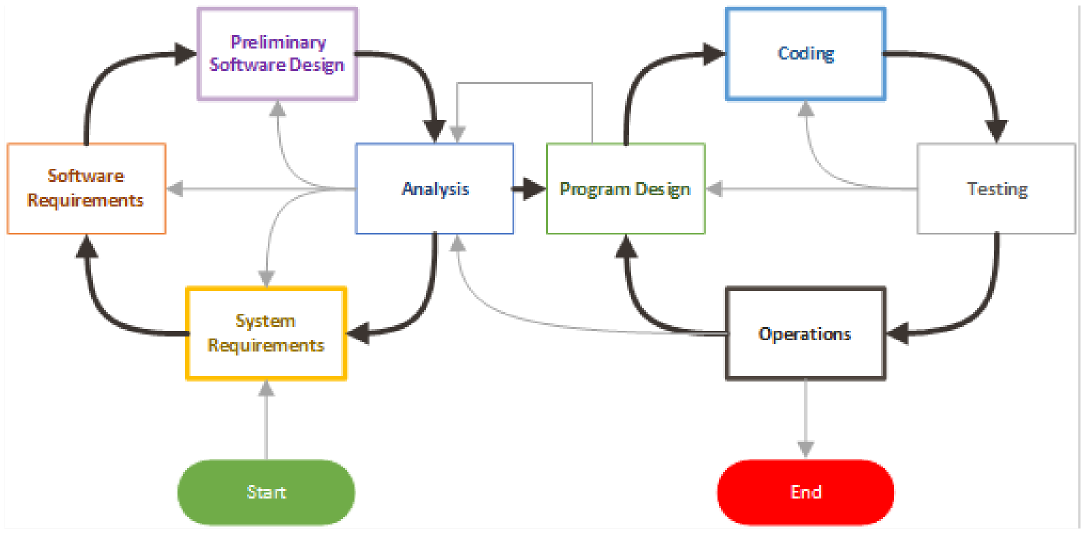
But even that will run into those who feel that any attempt to plan is a mistake. I have my own disagreements with that point of view, but they are outside the scope of this article. All I can say is that no matter what the original authors proposed, the term 'Waterfall' seems to be inextricably linked with the 'Frozen' interpretation and so your use of the term should take that into account.

'Modern' Interpretations

So if the term 'Waterfall' is tainted, and you are looking for a more project-management focused planned development approach [setting aside whether you think that is a good idea or not], what do you do?

From the reading I have done, it seems likely that at least one reason for the misunderstanding of what Royce was describing was due to the diagrams he chose to use. So the following are attempts to re-configure the diagrams Royce used into something that is more likely (I hope) to show what he was proposing, or which provide a more 'modern' take on what 'Waterfall' or similar planned development approaches might look like.

The first attempt I made results in the following double-loop design. The primary flows follow the darker lines, with the grey lines being opportunities to 'jump' the flow to other locations.



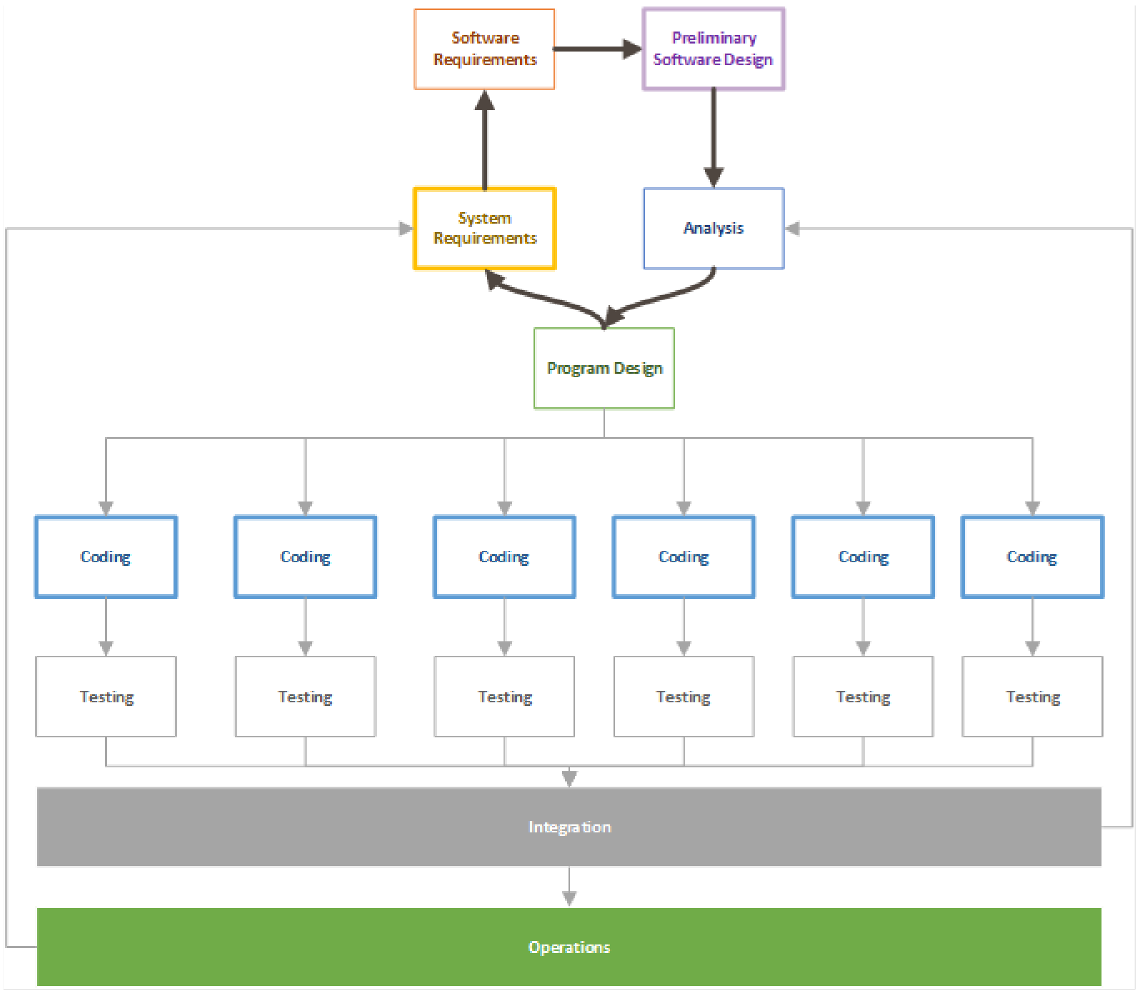
The other idea was to show a version that is closer to what is in DOD-2167 and which is design for the definition of an overall set of requirements, but with the phased implementation of those requirements.

This model is simplified slightly, but the idea is essentially to leverage concepts from both worlds. The idea is something like this:

1. Define the requirements, and especially the entire system architecture to as great a degree as possible first.
2. From that system design and requirements, start carving out functional sub-sets that can be built and implemented on their own.
3. As these are implemented, integrated, and in operation (if possible); feedback is gathered and changes to the requirements are made.
4. Those changes in requirements are incorporated into the ongoing design phase, assigned to one of the development streams, and integrated and deployed to operations when appropriate.

This results in an 'alternative' Waterfall that looks like the model below.

Source: http://www.bawiki.com/wiki/Waterfall.html Downloaded 1st February 2024



This is the current end of this article. Any feedback, additional citations, and alternative ideas would be appreciated.

References

1. Research Paper - **Production of Large Computer Programs**, by Herbert D. Bennington. From a presentation at a symposium on advanced programming methods for digital computers sponsored by the Navy Mathematical Computing Advisory Panel and the Office of Naval Research in June 1956. Forward with additional material added in 1987. https://mosaicprojects.com.au/PDF-Gen/Benington_-_Production_of_Large_Computer_Programs.pdf
2. Research Paper: **Managing the Development of Large Software Systems**, by Dr. Winston W. Royce, 1970. https://mosaicprojects.com.au/PDF-Gen/Royce_-_Managing_the_development_of_large_software_systems.pdf
3. Military Standard: **DOD-STD-2167: Defense System Software Development**, 4 June 1985. https://mosaicprojects.com.au/PDF-Gen/DOD-STD-2167.pdf
4. Research Paper: **Toxic Concepts in Systems Analysis and Design: The Systems Development Lifecycle**. By Paul Ralph. May 2010.

https://eprints.lancs.ac.uk/id/eprint/47395/1/Toxic_Concepts_in_Systems_Analysis_and_Design_The_Systems_Development_Lifecycle.pdf

5. Research Paper: **A Review of Risk Management in Different Software Development Methodologies**. By Hijazi, Khdour, and Alarabeyyat. May 2012. https://research.ijcaonline.org/volume45/number7/pxc3879113.pdf
6. <deleted>
7. Wikipedia Entry: **Waterfall Model**. On Wikipedia. Various Authors. Accessed on 15 October 2014. https://en.wikipedia.org/wiki/Waterfall_model
8. Article: **There's no such thing as the Waterfall Approach! (and there never was)**. By Conrad Weisert. Information Disciplines, Inc. 8 February 2003.
9. Wikipedia Entry: **Straw man**. On Wikipedia. Various Authors. Accessed on 16 October 2014. https://en.wikipedia.org/wiki/Straw_man
10. Article: **A Waterfall Systems Development Methodology … Seriously?** By David Dischave. On the Global Enterprise Technology web site. 17 September 2012.
11. Using this formula: (50 * (236.2 / 27.2)) = 434.19. Calculated based on information found at the The Federal Bank of Minneapolis
12. Research Paper: **Software Requirements - Are They Really a Problem?** By T.E. Bell and T. A. Thayer. Proceedings of the 2nd international conference on Software engineering. IEEE Computer Society Press, 1976. https://mosaicprojects.com.au/PDF-Gen/software_requirements_are_they_really_a_problem.pdf
13. Blog Post: **The Waterfall Accident**. By Pascal Gugenberger. On his personal web site. 2011.
14. Research Paper: **The Impact of DoD-Std-2167A on Iterative Design Methodologies – Help or Hinder?** By Scott P. Overmyer. ACM SIGSOFT Software Engineering Notes. 1990. 15(5), 50-59.
15. Blog Post: **Don't draw diagrams of wrong practices – or Why people still believe in the Waterfall model**. By Tarmo Toikkanen. On his personal blog. 9 September 2005. https://www.tarmo.fi/2005/09/09/dont-draw-diagrams-of-wrong-practices-or-why-people-still-believe-in-the-waterfall-model/
16. Article: **Department of Defense MIL-STD-2167 Waterfall is Iterative**. By Kenneth Shafer. On the Legacy Guild web site. 27 December 2013. https://legacyguild.org/index.php?page=downloads&type=entry&id=object-oriented-and%2Fdepartment-of-defense
17. Blog Post: **Waterfall Model Probably the Most Costly Mistake in the World**. By Rolf H. On the Value@Work blog. 18 April 2013. http://valueatwork.se/waterfall-model-probably-the-most-costly-mistake-in-the-world/?lang=en
18. Technical Report: **Evolutionary Software Development**. By NATO Task Group IST-026. August 2008.
19. Working Paper: **Beyond the Waterfall – Software Development at Microsoft**. By Michael A. Cusumano and Stanley Smith. MIT Sloan School of Business. 16 August 1995. https://dspace.mit.edu/bitstream/handle/1721.1/2593/SWP-3844-33836288.pdf?sequence=1

20. Wikipedia Entry: **Source Lines of Code**. By various authors. Accessed on November 16, 2014. https://en.wikipedia.org/wiki/Source_lines_of_code

21. Research Paper: **A View of 20th and 21st Century Software Engineering**. By Barry Boehm. Presented at ICSE'06 in Shanghai, China. 2006.

22. Wikipedia Entry: **Structured Programming**. Various Authors. Accessed on 17 November 2014. https://en.wikipedia.org/wiki/Structured_programming

23. Wikipedia Entry: **Spaghetti Code**. Various Authors. Accessed on 17 November 2014. https://en.wikipedia.org/wiki/Spaghetti_code

24. Blog Post: **There is No Such Thing as Waterfall.** By Erik Dietrich. On the DaedTech blog. September 19, 2012. https://daedtech.com/there-is-no-such-thing-as-waterfall/

25. White Paper: **Methodology for Rapid Development of C2 Planning Systems**. By Sheena Kelsey and Simon Snell. Of QinetiQ. 2003.

26. Research Paper: **A Spiral Model of Software Development and Enhancement**. By Barry Boehm. Computer 21, no. 5 (1988). 61-72.

27. Research Paper: **Constraint-Driven Software Design - An Escape From the Waterfall Model.** By R Hoog, T Jong, F Vries. Performance Improvement Quarterly, 1994. https://research.utwente.nl/en/publications/constraint-driven-software-design-an-escape-from-the-waterfall-mo

28. Research Paper: **Resource Utilization during Software Development**. By Marvin V. Zelkowitz. The Journal of Systems and Software. 1988. https://www.cs.umd.edu/users/mvz/pub/zelkowitz-jss-1988.pdf

29. Blog Post: **The Rise and Fall of Waterfall Development**. By Richard Banks. On his Agile and .Net blog. 8 January 2009. https://www.richard-banks.org/2009/01/rise-and-fall-of-waterfall-development.html

30. Book: **Agile & Iterative Development – A Manager's Guide**. By Craig Larman. 2003. Pearson Education, Inc.

31. Blog Post: **Failure of the scientific method and the waterfall method (again).** By Tarmo Toikkanen. On his personal blog. 18 October 2007. https://www.tarmo.fi/2007/10/18/failures-of-the-scientific-method-and-the-waterfall-method-again/

32. Research Paper: **Software Development Lifecycle Models.** By Nayan B. Ruparella. ACM SIGSOFT Software Engineering Notes. May 2010.

33. Blog Post: **The waterfall – A Historical View to Organizing Software Development**. By Patrik Malmquist. 5 September 2011.

34. Article: **The seductive and dangerous V Model.** By James Christie. On the Claro Testing web site. An expanded version of a December 2008 article that appeared in Testing Magazine.

35. Book Chapter: **Why the Waterfall Model Doesn't Work**. Scaling Software Agility.

36. Research Paper: **Anchoring the Software Process**. By Barry Boehm. Software, IEEE 13.4 (1996). https://www.kiv.zcu.cz/~brada/files/aswi/cteni/boehm1996anchoring.pdf

37. Research Paper: **Iterative Software Development – from Theory to Practice**. By Amir Torer, Boaz Shani, and Ely Bonne. RAFAEL, Israel.

https://www.stickyminds.com/sites/default/files/article/file/2014/Iterative%20Software%20Development-%20from%20Theory%20to%20Practice.pdf

38. Wikipedia entry: **Big Design Up Front**. Accessed on February 1, 2015. https://en.wikipedia.org/wiki/Big_design_up_front

39. Article: **Examining the ‘Big Requirements Up Front (BRUF) Approach**. By Scott Ambler. On his Agile Modeling web site. Undated. https://agilemodeling.com/essays/examiningBRUF.htm

40. Article: **Semi-Automatic Ground Environment Air Defense System Part of the Lincoln Laboratory 'History' section**. Accessed on 21 August 2022. https://www.ll.mit.edu/about/history/sage-semi-automatic-ground-environment-air-defense-system

41. Web Page: **A Brief History of MITRE**. Accessed on 22 August 2022.