Systems Analysis & Design I

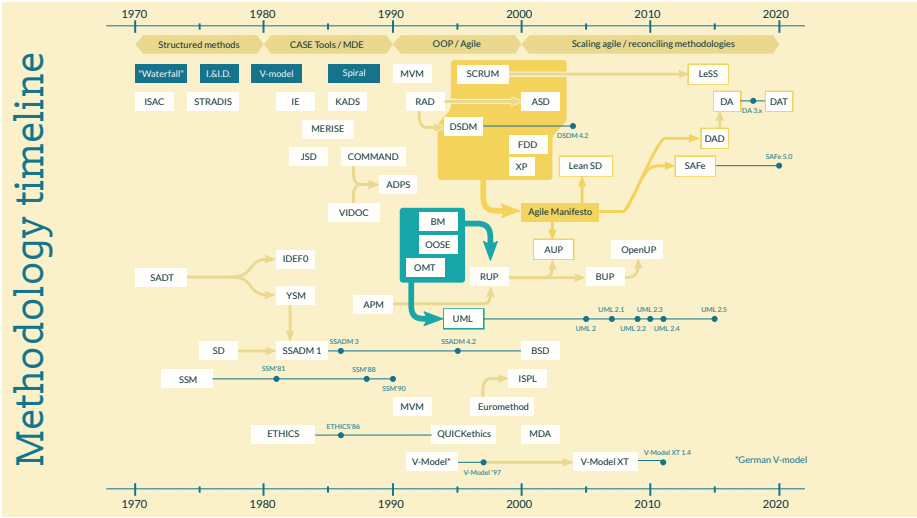
Week 3: Life cycle models

Life cycle models in the 50s and 60s

1

- Last week we discussed methods and methodologies
- and we learned about the "hard" systems view and what assumptions it makes about the world.
- For the next few sessions, we will simply accept this world view, but we will be more critical of it later. -
- We further saw how "hard" design methodologies follow a broad structure in order to arrive at a system design
- In this segment I would like talk about how these methodologies developed over time.

2



- See, this timeline is somewhat misleading, it suggests that all of these methodologies have the same goal. That is only true on a very broad level:
- some of these methodologies cover only small aspects of design and analysis, some are specifically designed for software development, and others are general project-management frameworks.
- In this segment we will focus on the so-called "software development life cycle" models and how they changed over time.
- Life cycle models divide the design process into distinct phases to guide the designer.
- On this slide you can see four of them, the dark blue boxes on the top left labeled "Waterfall", "I.I.D" (which stand for Iterative & Incremental Development), "V-model" and "Spiral".

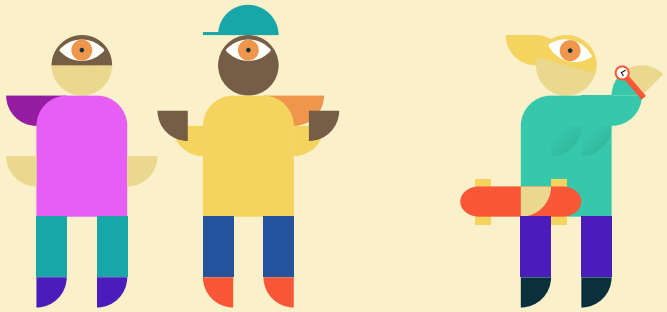
- Note that V-model appears twice, the box in 1992 refers to the German V-model which is, confusingly, not just a model but a whole methodology.

3

Life cycles

What should we do next?

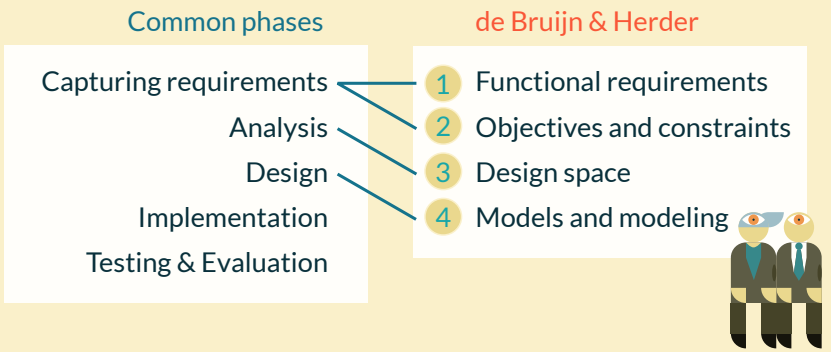
For how long should we do it?



- In general, life-cycle models provide answers to two simple questions: 1) What do we do next and 2) for how long should we do it?

4

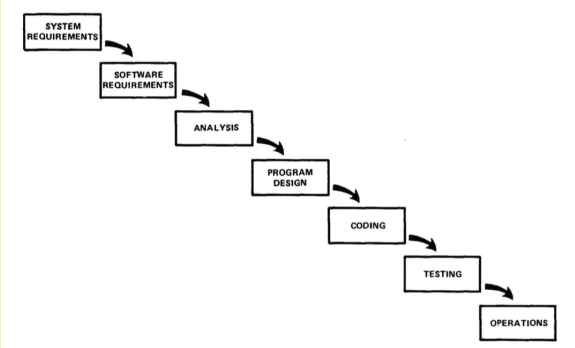
Life cycles



- Every methodology has some form of life cycle model and all of them have roughly the same phases: capturing requirements, analysis, design, implementation, testing and evaluation.
- Compare that to the four stages of De Bruijn and Herder for "hard system" design methodologies that we discussed last week:
- The phases "Functional requirements" and "Objectives and constraints" correspond to requirements capture,
- the phase to map out the "Design space" is part of the analysis,
- and the phase "Models and modeling" is the final design.
- This suggests that a "hard systems" life cycle must cover variants of these four phases (and the remaining phases cover the real-world implementation process)
- However, as we will see, the models put different emphasis on these phases and traverse them in different modes.

5

"Waterfall"

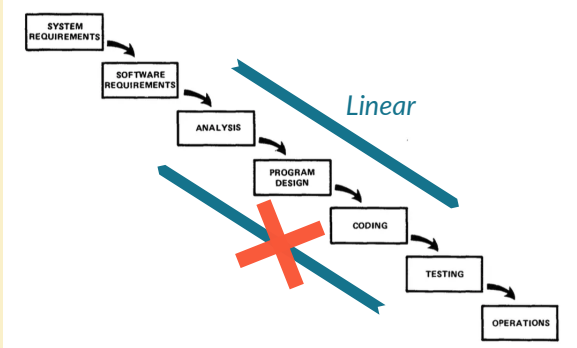


Royce WW. Managing the Development of Large Software Systems 1970. InProceedings, IEEE WESCON 2012 (pp. 1-9).

- There is an unwritten rule that every software engineering lecture has to introduce and then condemn the waterfall model.
- They usually show this diagram and say "this is the waterfal model by Winston W. Royce and it is bad because it is linear"

6

"Waterfall"

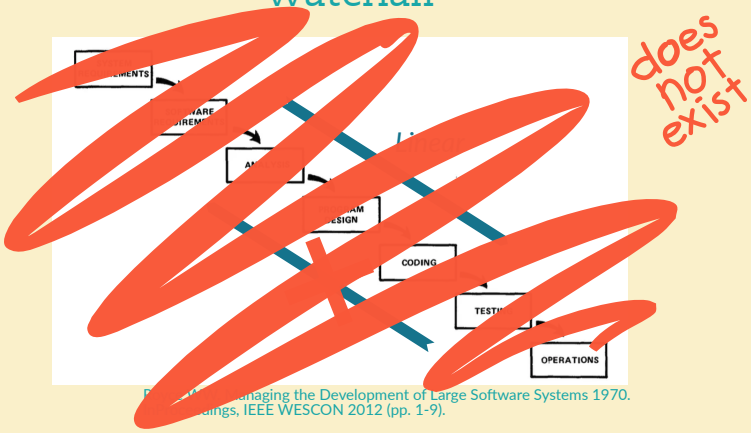


Royce WW. Managing the Development of Large Software Systems 1970. InProceedings, IEEE WESCON 2012 (pp. 1-9).

- "linear" here means that, like a waterfall, we go from one project phase to the next without ever looking back.

7

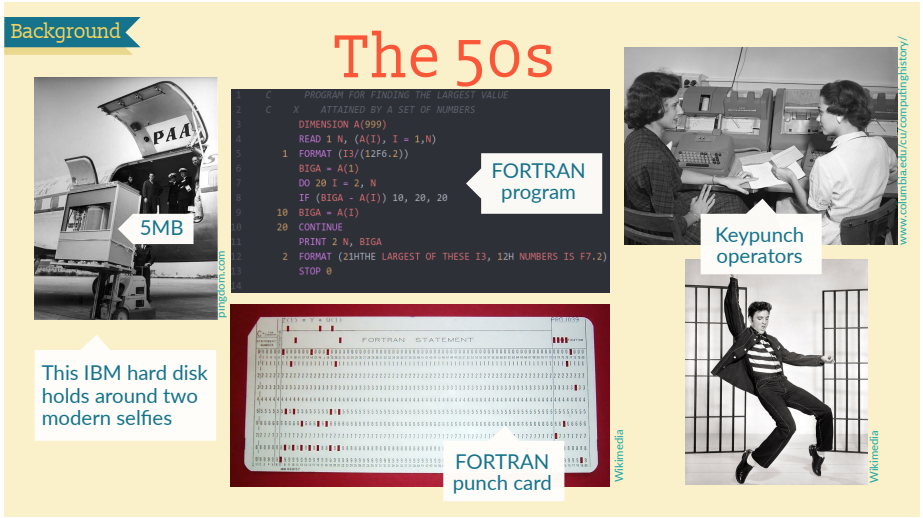
"Waterfall"



Royce WW. Managing the Development of Large Software Systems 1970. InProceedings, IEEE WESCON 2012 (pp. 1-9).

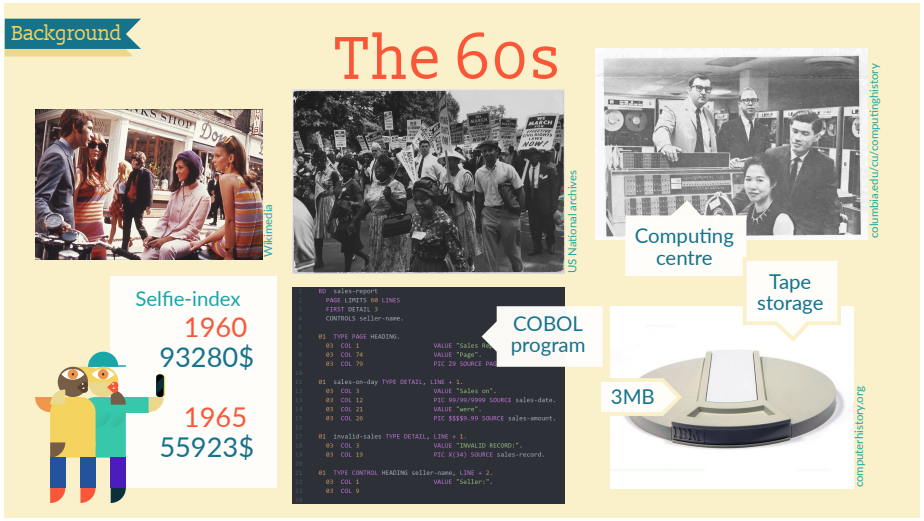
- However, the waterfall model does not exist, at least not this one! It has established itself as a myth in software engineering which says that "this is how 'bad' project management looks like", but you have to go back over 60 years to find examples of people implementing software like that.
- Winston W. Royce, the author of the paper that is usually cited for this model, did not actually espouse it. He essentially said that this kind of process is common in the engineering world and that for information systems, this process needs to be modified.
- Let's look at an actual example of a "linear" process model. For that, we have to go back to the 1950s!

8



- In order to put the following into context, let us briefly review how computing in the 50s and 60s looked like.
- In the early 50s, coding was done by writing assembler code which is just a small step up from writing machine instructions.
- Since this is a very tedious, slow and error-prone process, reseachers at IBM developed the FORTRAN language. FORTRAN is a compiled language, that means that the computer translates higher-level instructions into machine code.
- The FORTRAN languages already allows the use of variables, loops and branching statements.
- But as with assembler languages, the program had to be written on punch cards in order to run it. Key-punch machines sold alongside the computing hardware helped to write and verify the code.
- IBM also produced the first hard disk, you see a photo of it being loaded onto an airplane on the top left. The total storage capacity of this machine would be enough to store about 2 selfies taken with a modern smart phone.

9

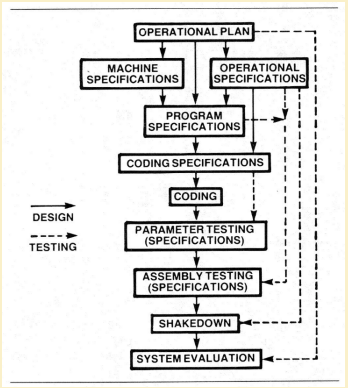


- In the 60s information technology is now cheap enough that computing centres become commonplace in large companies and universities

- Some large universities, like Columbia or Chapel Hill in the US or Manchester in the UK, now offer full-scale computer science courses.
- The COBOL language dominates the world of business programming.
- Due to the advent of magnetic tape storage, data storage costs are vastly lower than in the 50s, but still:
- The inflation adjusted cost for the storage amount needed for a single selfie is about 90 000$ in 1960 and 55 000$ in 1965.
- Now, this should give you a good impression of what people in software development had to work with. So let us look at a life cycle model from that time:

10

Benington's life cycle

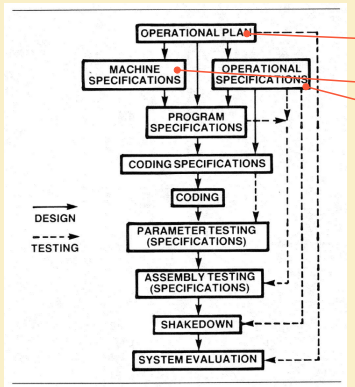


Benington HD. Production of large computer programs. InProceedings of the 9th international conference on Software Engineering 1987 Mar 1 (pp. 299-310).

- This model was developed by Herbert D. Benington in the context of a real-time air-defense system for the USA. He presented this model at a symposium on advanced programming methods in 1956.
- And indeed, this model shows a very linear structure.
- The terminology is a bit dated, so I will translate it to modern vocabulary.
- Going from top to bottom (so from start to finish):

11

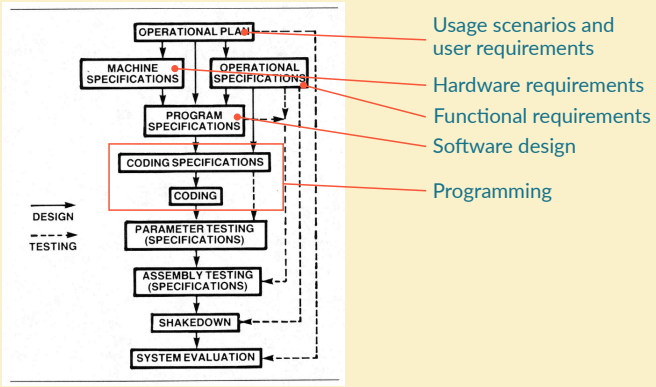
Benington's life cycle



Benington HD. Production of large computer programs. InProceedings of the 9th international conference on Software Engineering 1987 Mar 1 (pp. 299-310).

- "Operational plan" is the collection of usage scenarios and user requirements
- "Machine specifications" describe the hardware to be developed, back then no general-purpose computers existed. Instead, hard- and software was developed jointly.
- "Operational specification" correspond to capturing functional requirements and in particular the inrputs and outputs of the system

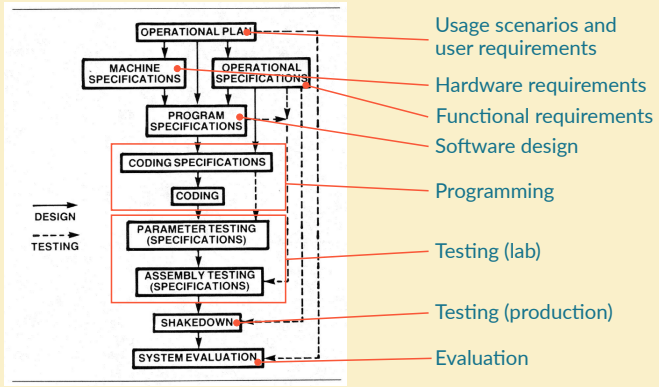
Benington's life cycle



Benington HD. Production of large computer programs. InProceedings of the 9th international conference on Software Engineering 1987 Mar 1 (pp. 299-310).

- "Program specification" is the design of the software
- and "Coding specifications" are something like today's programs in a high-level language. Not in a computer, mind you, with pen and paper!
- "Coding" is then the translation of the high-level language into machine code. In the 50s this often enough still involved a translation by hand.

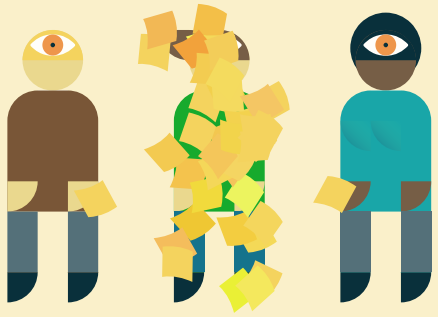
Benington's life cycle



Benington HD. Production of large computer programs. InProceedings of the 9th international conference on Software Engineering 1987 Mar 1 (pp. 299-310).

- In "Parameter testing" we test the individual sub-programs based on the coding specification (that's why there is a dashed arrow connecting parameter testing and coding specifications)
- During "Assembly testing" the system (hard- and software!) is put together and is tested according to operational and program specifications (again, that's why there is a dashed arrow)
- "Shakedown" is the test of the system in its production environment
- And finally "system evaluation" is the evaluation of the system once it is in production, meaning, it is actively used in the organisation
- Now, hopefully you picked up on a few things here: the software development process in the 50s is *incomparable* to the software development today, and even to what was possible in the 70s!
- It is much more of an engineering process—with hardware being built together with software and most of the programming done with pen and paper.

Hands-on!



- In the next segments we will see how life cycle models for information systems changed over the decades.
- But before that, I would like you to experience going through a linear design process like Benington's model.
- Sadly, it is impossible to simulate information system development in the classroom, but luckily many issues we face there are also present in other types of projects
- Therefore, I'd like you to simulate a project with a "linear lifetime" by working together in groups to perform more of an engineering task.
- It is important that you follow the instructions I uploaded to make this a proper learning experience! The task is _not
emph to present the best possible solution to the given problem,.
- Instead, the task is to go through the design process as described in order to experience it first-hand.

Systems Analysis & Design I

Week 3: Life cycle models

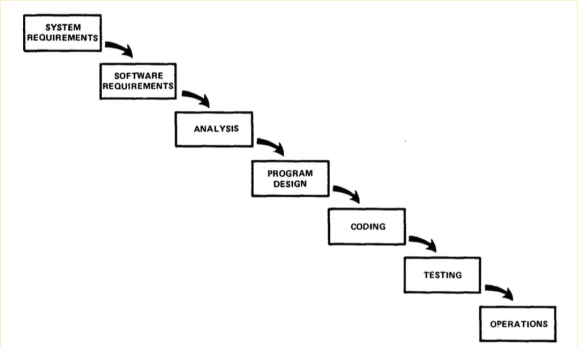
Life cycle models in the 70s and 80s

1

- In the 70s, the semiconductor memory replaces magnetic core memories in commerical computers, halfing the space requirement and paving the way for mass-production.
- Storage space is still quite expensive: storing a modern selfie would cost around 1700$ in 1970 and almost 1000$ in 1975 (as a sidenote: I have not found out why hard disk prices rose in the mid-70s or whether that is some bias in the data I have)
- In this decade, network technology matures and commercial computer networks like Telnet become commonplace. The picture on the top right is Queen Elizabeth sending an email via the military network ARPANET, which is predecessor of the modern internet.
- The first personal computers appear in the late 70s. Programming becomes available to the hobbyist with languages like BASIC or PASCAL and with the help of cheap floppy disks small programs are for the first time easy to distribute.
- For those that grew up after the era of floppy disks: you can see one in the bottom-right.
- Dennis Ritchie invents the C programming language which quickly becomes an industry standard and helps give birth to the UNIX operating system.
- The 70s mark the beginning of the end for computing centres: the shift towards smaller desktop computers has begun, but it will take until the 90s until this development completes

2

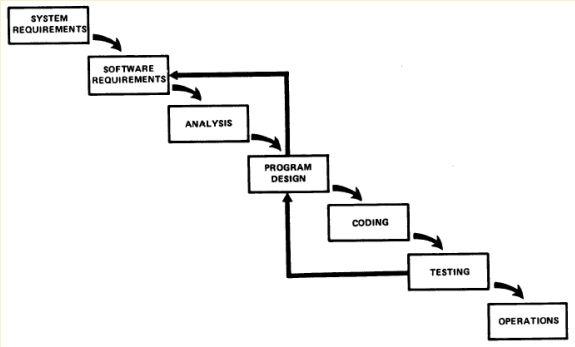
"Waterfall"



Royce WW. Managing the Development of Large Software Systems 1970. InProceedings, IEEE WESCON 2012 (pp. 1-9).

- So, in the 70s Royce writes about the "waterfall" model we discussed earlier.
- The phases he distinguishes are "System requirements", "software requirements", "analysis", "program design", "coding", "testing", and "operations"
- These phases are very similar to Benington's model, so I will not discuss them again. They also should sound familiar from last week's segment on hard systems design as described by de Bruijn and Herder.

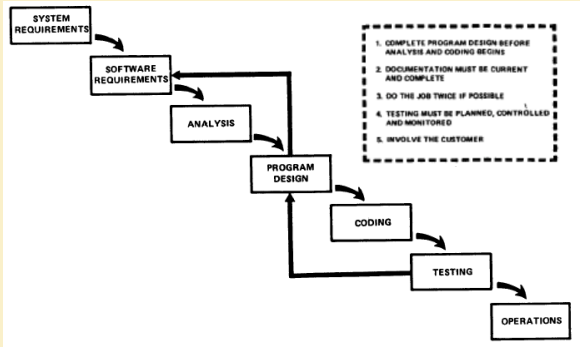
"Waterfall"



Royce WW. Managing the Development of Large Software Systems 1970. InProceedings, IEEE WESCON 2012 (pp. 1-9).

- But Royce only introduced this model to show how it can be improved. This here is one of his proposals: after the testing phase we actually need to go back to the programs design step, maybe even the functional requirements step and repeat the subsequent steps to arrive at the final system.
- So Royce's first proposal already moves away from the idea of a "linear" life cycle.
- Royce also added a few notes to the diagram, two of which I would like to highlight because it is usually glossed over:

"Waterfall"

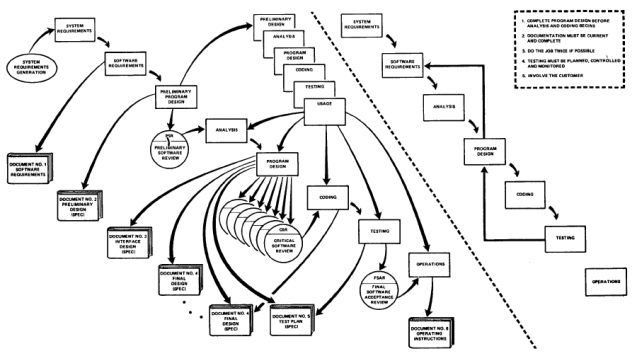


Royce WW. Managing the Development of Large Software Systems 1970. InProceedings, IEEE WESCON 2012 (pp. 1-9).

- Point 3 reads: "Do the job twice if possible"
- and point 5 reads: "Involve the customer"
- In software engineering, this advice from 1970 was only really taken seriously in the 1990 with the advent of the so-called "agile" methodologies.
- You will find a lot of texts that critize early methodologies for a) being linear and b) not involving the user enough. This was clearly no the case!
- In the 70s we find a lot of methodologies which more or less follow a linear structure, borrowing from the engineering projects, but we see here that people were clearly aware that iteration was possible and desireable.

5

"Waterfall"



Royce WW. Managing the Development of Large Software Systems 1970. InProceedings, IEEE WESCON 2012 (pp. 1-9).

- Just to do Royce justice, here is by the way the full model which is already half-way to a methodology. Part of this diagram actually refers to building a prototype, something that we will find again in the following methodologies.

6

Iterative & incremental

> Even if the implementors have previously undertaken a similar project, it is still difficult to achieve a good design for a new system on the first try. Furthermore, design flaws often do not show up until the implementation is well under way so that correcting the problems can require major effort.

V. Basili and J. Turner, "Iterative Enhancement: A Practical Technique for Software Development," IEEE Trans. Software Eng., Dec. 1975, pp. 390-396

- To give you further evidence against the idea that "linear" life cycles were somehow uncritically accepted back then, Basili and Turner already in 1975 wrote the following: (Quote)

7

Iterative & incremental

> The basic idea behind iterative enhancement is to develop a software system incrementally, allowing the developer to take advantage of what was being learned during the development of earlier, incremental, deliverable versions of the system. Learning comes from both the development and use of the system, where possible.
> Key steps in the process were to start with a simple implementation of a subset of the software requirements and iteratively enhance the evolving sequence of versions until the full system is implemented. At each iteration, design modifications are made along with adding new functional capabilities.

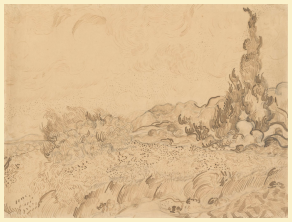
V. Basili and J. Turner, "Iterative Enhancement: A Practical Technique for Software Development," IEEE Trans. Software Eng., Dec. 1975, pp. 390-396

- They propose to work in an iterative and incremental fashion as a remedy: (Quote)
- So *iterative* means that we cycle through the design process more than once
- and *incremental* means that we start out with a scaled down version of the product and in each iteration we add further features.
- Today, you will find the term "iterative and incremental development" or IID in almost all modern software engineering methodologies
- But as you can see, the idea is over 45 years old.

8

Background

Iterative & incremental



Various sketches of Vincent van Gogh during his stay at the Saint-Paul asylum in Saint-Rémy-de-Provence

Van Gogh
Wheat Field with Cypresses
Saint-Rémy, September, 1889

- I think it makes sense to compare this approach to other creative efforts. A painter does not simple start out painting, they usually start with a rough sketch which they then turn into a rough painting.
- They then refine the painting by adding details and correcting mistakes.
- The painting on the right here is "Wheat Field with Cypresses" by Vincent Van Gogh.
- Van Gogh spent the summer of 1889 in an ayslum in Saint-Rémy-de-Provence. During this time, he studied the landscape and created a learge number of sketches and paintings.
- Somewhat simplified, we can call this process *iterative* because the works all show similar motives, and we can call it *incremental* because each version adds more details.
- This is the same in almost all creative endeavours, whether it is writing, painting, music or design. Instead of aiming for perfection with the first version, we should see it as a learning opportunity.
- This also means that true skill does *not* mean doing great on the first try, but to be able to recognize flaws and fix them.
- In my mind, this is a great way of approaching many different types of problems.

9

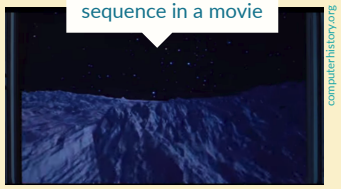
Background

The 80s

Wikimedia

The Guardian

The first CGI sequence in a movie



computerhistory.org

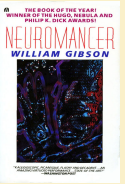
Selfie-index
1980
618$
1985
14$

computerhistory.org

The birth of Cyberpunk

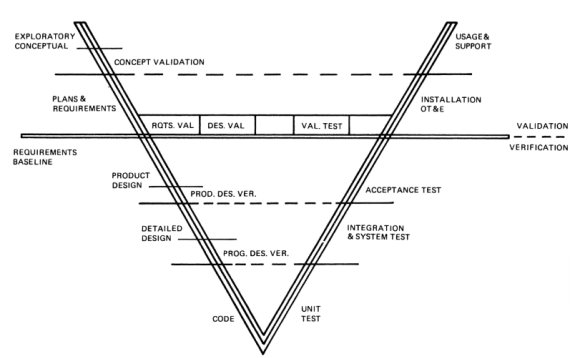
1.44MB

Wikimedia



- In the 80s we see a lot of the trends that started in the 70s to come into full force and enter the mainstream.
- Computing enters the creative industry with special effects and computer-generated imagery. The second star trek movie featured the first full CGI sequence.
- The increasing connectivity through computer networks inspires gernes like cyberpunk, which imagines virtual realities and widespread hacking in a corporatist future.
- The home computing revolution continues with the highly succesful Apple II model and we see storage and computing costs go down at a fast pace.
- While a modern selfie would still cost hundreds of dollars in the beginning of the 80s, towards the end it is below ten dollars.
- Floppy disks become smaller and can now hold around 1.5 Megabyte.
- The programming languages C++ and objective-C extend the C language with object-oriented capabilities.

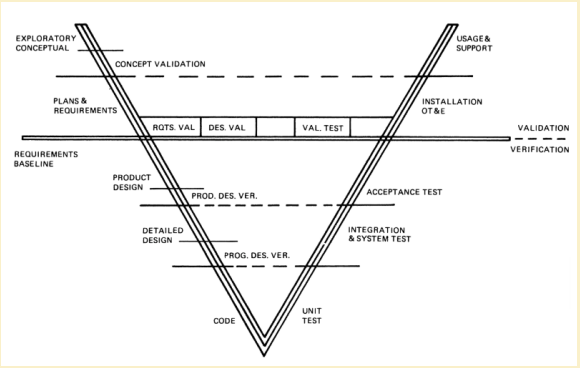
Boehm's V-model



B.W. Boehm, "Guidelines for Verifying and Validating Software Requirements and Design Specifications," Euro IFIP 79, P.A. Samet, ed., North-Holland Publishing Company, 1979, pp. 711–719.

- One way in which people tried to improve "linear" life cycle models was to highlight how the different steps reside on different conceptual levels.
- This is highlighted by drawing the phases in a V-shape: we begin with high-level plans and requirements, then move down to design, and then end at the bottom where actual code is written.
- When we move back up the V-shape through different testing phases, we traverse these conceptual levels in reverse.
- So we first test the code itself, then its integration into the larger system, and then the interaction with the user.
- Higher up, we install the system and finally provide support during its operation.
- These last two parts are marked as "validation" in the diagram and the preceeding steps as "verification". This is a useful distinction, so let us discuss it in more detail:

Boehm's V-model



Validation

External process to test system against needs of stakeholders (customer)

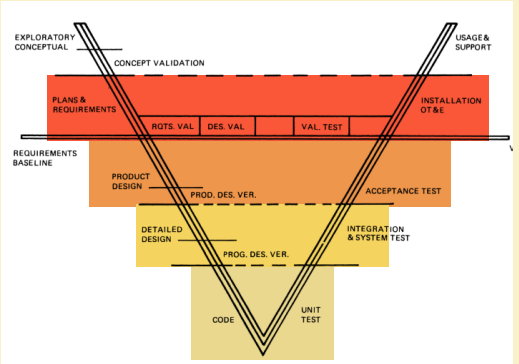
Verification

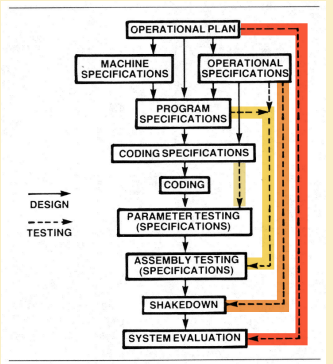
Internal process to test system against specifications and requirements

B.W. Boehm, "Guidelines for Verifying and Validating Software Requirements and Design Specifications," Euro IFIP 79, P.A. Samet, ed., North-Holland Publishing Company, 1979, pp. 711–719.

- "Validation" is the assurance that the system meets the needs of the customer and other stakeholders, so it is an external process (meaning that people from external organizations are involved).
- "Verification" is an internal process (meaning it happens inside the developing organization) where the system is tested against specifications and requirements.
- You can see how the verification and validation steps conceptually match up with their counterparts on the left side of the V: unit tests happen on the level of actual software code at the bottom of the V, integration concerns the design specifications, user acceptance tests concern the final product in its entirety.
- For validation, we see that installation, operational tests and evaluation match up with plans & requirements: essentially, we are testing the customer's requirements by evaluation the system in its final environment.
- And finally, "usage and support" matches up with with the conceptual phase at the top: the system that was envisioned in the very beginning must be compared to the live system.
- So why is this perspective useful? One way of thinking about this correspondence is that each "level" of the V should be executed by the same roles.
- The top levels are where project managers work, in the middle system designers or software architects, and the bottom programmers.
- You will often read that the V-model is "linear", but really that is not the point of the model: instead, it is the correspondence between the different phases.
- And this correspondence was already known much earlier, but not captured with this type of diagram.

Boehm's V-model



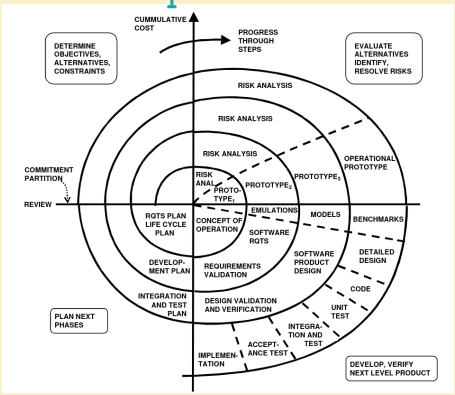


B.W. Boehm, "Guidelines for Verifying and Validating Software Requirements and Design Specifications," Euro IFIP 79, P.A. Samet, ed., North-Holland Publishing Company, 1979, pp. 711–719.

- Already in Benignton's model we can see the correspondence between the conceptual levels: look at how the outermost activities are matched up, then the second activity and the second-to-last, and so on.

- This matching describes exactly the V-correspondence that we just discussed.

The spiral model



Boehm, B.W., 1988. A spiral model of software development and enhancement. Computer, 21(5), pp.61-72.Boehm, B.W., 1988. A spiral model of software development and enhancement. Computer, 21(5), pp. 61-72.

- The final model from the 70s/80s era that we discuss today is the so-called Spiral model by Barry Boehm.
- We earlier heard the terms iterative and incremental development. The spiral model is a more concrete methodology that helps to follow these principles.

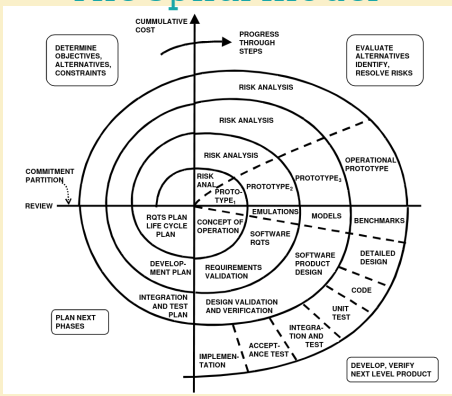
The spiral model

Determine objectives

Identify and resolve risks

Plan next iteration

Development and testing



Boehm, B.W., 1988. A spiral model of software development and enhancement. Computer, 21(5), pp.61-72.Boehm, B.W., 1988. A spiral model of software development and enhancement. Computer, 21(5), pp. 61-72.

- The model consist of four phases. We have:
- Every iteration goes through these four steps, at the end of step 3 (bottom right) we end up with the next version of the software.
- If we visualize the process as moving around the four phases in each iteration and moving "outwards" for each increment of the software, both movements together form a spiral from which the model takes its name
- Now, it is important to note that not every iteration looks the same: the first iteration should, for example, not include any actual coding, instead, we want to end up with something like a conceptual prototype.
- In the example provided in the figure, the spiral model works like a run-up to a linear development process at the very end: we see the phases 'detailed design', 'code', 'unit test', 'integration and test', 'acceptance test' and 'implementation'.
- This last bit looks very much like a variant of the "waterfall" or Benignton's model, but of course with a lot of preparation and insight

generated by the earlier iterations.

- The author, Boehm, in later texts clarified that the spiral model should be seen as a *risk mitigation strategy* and emphasises that its is a general methodology in which other methodologies can be included.

15

The spiral model

> The spiral development model is a risk-driven process model generator. It is used to guide multi-stakeholder concurrent engineering of software-intensive systems. It has two main distinguishing features.
> One is an acyclic approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk. The other is a set of anchor point milestones for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions.

Boehm, B. and Hansen, W.J., 2000. Spiral development: Experience, principles, and refinements (No. CMU/SEI-2000-SR-008). CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST.

- (Quote)
- The term "stakeholder" refers to people that are involved or affected by the system. Who is and who is not a stakeholder is actually a deeper topic that we will revisit later, for now, simply imagine those people that are either part of the development process or people who are part of the organisation for which the development is done as the stakeholders.
- In the quote we just read we see that Boehm puts a great emphasis on risk mitigation which is often glossed over when it comes to the spiral model.
- Many texts simplify it as "waterfall, but iterative and incremental". As we can see from both the diagram and the quote, this description is inaccurate.
- Instead, the spiral model can be seen as a process that *supports* another life cycle model by adding incremental and iterative phases to it.

16

Hands-on!

- So, let us see whether iteration can help in your design process.
- Your task is very similar to before, but please read the second set of instructions carefully: our goal is to *iterate* the design process, which means that you first need to *reflect* on how the first iteration went.
- That means that you are asked to start off with a discussion to figure out what went well and what did not. I strongly suggest to listen to the person who built the last design! Practical experience usually trumps theory!

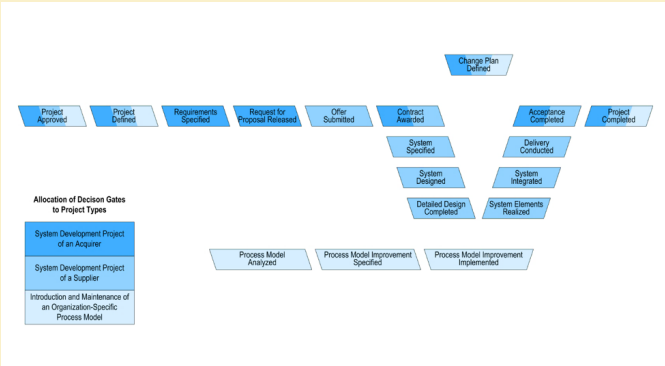
Systems Analysis & Design I

Week 3: Life cycle models

Life cycle models in the 90s and now

Background

The "German V-model"



http://ftp.uni-kl.de/pub/v-modell-xt/Release-1.1-eng/Dokumentation/pdf/V-Modell-XT-eng-Teil1.pdf

- Before we move forward into the 90s, we need to briefly mention one methodology which feels more like a relic from the 80s but whose development spans into the 90s.
- This methodology is called the "German V-model".
- It was developed in 1986 on an initiave of the German Federal Office for Defense Technology in order to improve software development practices.
- It was completed in '92 and updated in '97 (the so-called "V-Model '97"),'
- The German V-Model goes far beyond a life cycle model and is really better called a methodology. The introduction document is almost 500 pages long, so this diagram here is only one small aspect of a very big methodology.
- We will not talk about it further, I included it here so you are aware of the two different V-models out there since it is often thrown together with life cycle models.
- For us, only the V-model from the 70s is relevant.

Background

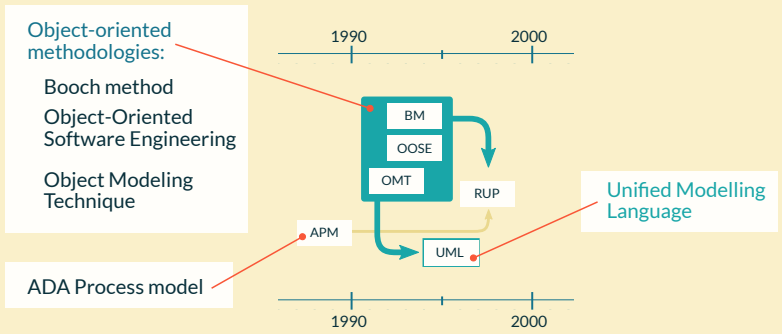
The 90s

- Ah, the 90s. Personal computers are everywhere and the first clunky laptops make their way into offices.
- Hard-disk space is cheap and quickly becomes cheaper: storing a modern selfie would cost around a dollar in the 90s and that price quickly comes down to mere cents in the mid-90s.
- Programmers can now rely on high-powered integrated development environments (IDEs) like visual studio.
- The world-wide web arrives and with it comes the first wave of internet businesses which boom and crash in the dot-com bubble.
- The "Year 2K" bug haunts the computing landscape and a huge global effort of updating crucial software is undertaken.
- The Soviet union collapsed and Fukuyama proclaims that the "end of history" has arrived. From now on, it is just business business business.

- Culturally, not everyone trusts this new techno-capitalism. Media like the "Matrix" movie express the unease that many feel with both the technology and the social order which make up the new status quo
- One aspect of this unease are the strict managerial hierarchies inside companies which many feel are counterproductive.

3

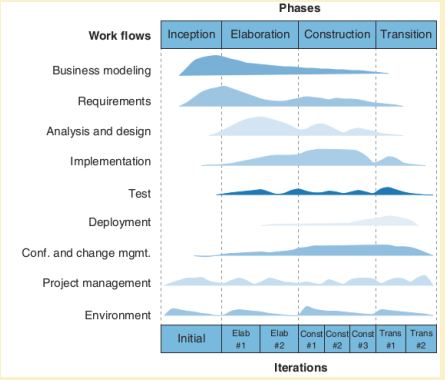
Object-oriented design



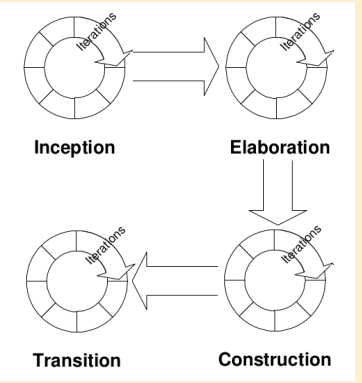
- In the mid-90s we see a proliferation of new approaches to software development. This is related to two important developments in software engineering: the rise of object-oriented design and the "agile revolution".
- While object-oriented programming goes back to the 60s and 70s with languages like SIMULA and SMALLTALK, it really only became a mainstream technology in the 90s.
- And this "new" style of programming sparked new ideas of how analysis and design could be performed.
- The Rational Unified Process (RUP) incorporated methods from three earlier methodologies centered on object oriented programming plus the "ADA process model" which is a variant of the Spiral model developed for the ADA programming language
- Around the same time the first version of the Unified Modeling Language (UML) was created, we will take a closer look at that in the upcoming weeks.
- Two properties of RUP are important from our perspective:

4

Rational Unified Process (RUP)



Kneuper R. Sixty years of software development life cycle models. IEEE Annals of the History of Computing. 2017 Sep 20;39(3):41-54.Kneuper R. Sixty years of software development life cycle models. IEEE Annals of the History of Computing. 2017 Sep 20;39(3):41-54.



Abrahamsson, P., Salo, O., Ronkainen, J. & Warsta, J. (2002) Agile software development methods: Review and analysis, VTT publication

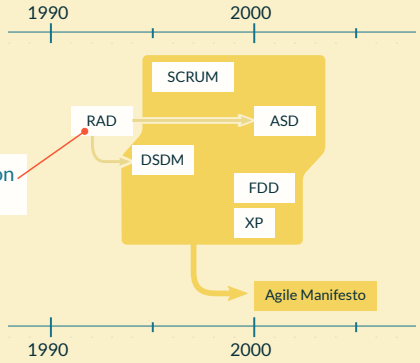
- First, it models the life cycle as a continuum: instead of tasks starting and stopping, each task continues for a certain time span, overlapping with other tasks. RUP uses the term "workflow" for these tasks.
- A workflow might wind up and wind down multiple times, as you can see in the diagram: the blue plots are supposed to indicate how much work is done in a specific workflow.
- Inside a workflow, RUP prescribes four phases: inception, elaboriation, construction and transition.

- We are not going into further details of how exactly these phases are supposed to work, but note that each phase is modelled in an iterative fashion.

5

Agile methodologies

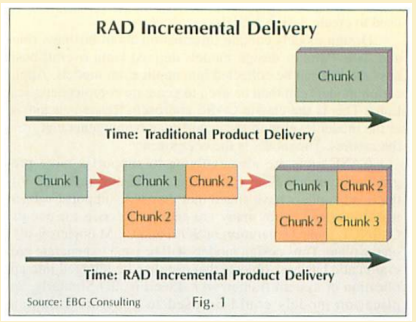
Office Space (1999, Twentieth Century Fox)

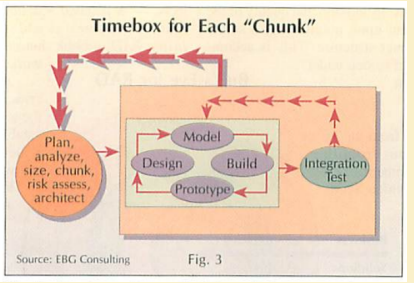


- The other big development besides object-oriented methodologies was a reaction to calls for "leaner" or more "agile" project management.
- Developers felt that strict hierachical management had a stifling effect on their productivity as it filled up their time with wasteful meetings and unproductive administration
- Another reason for this demand was the advent of the world wide web which remodeled the software landscape considerably.
- Faster computers and better networking capabilities alongside the emergent online marketplace also meant that customers had higher demands for software engineering:
- Software needed to be developed and deployed faster, with less time for analysis or thorough up-front design.
- On the developer's side, faster computers ment that build times for software went down and frequent changes to the design where becoming more feasible.

6

Rapid Application Development (RAD)

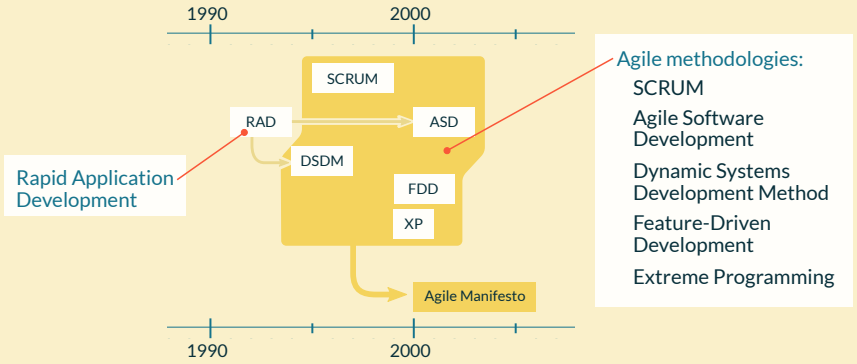




Gottesdiener E. RAD realities: Beyond the hype to how RAD really works. Application Development Trends. 1995 Aug;2(8):28-38.

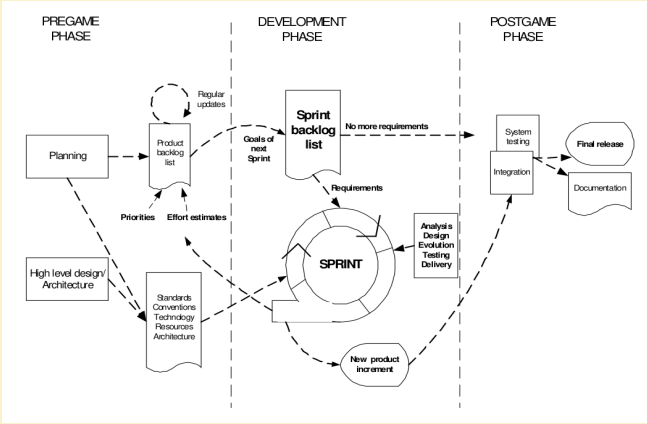
- We can see this reflected in the Rapid Application Development (RAD) methodology. In the figures shown here, we see that the software is build incrementally in "chunks" and that the life cycle for each chunk is iterative.

Agile methodologies



- RAD inspiried two other methodologies called "Agile software development" (ASD) and "Dynamic systems development method" (DSDM). Togther with SCRUM, Extreme Programming and Feature driven development they form the "classic" agile methodologies.
- Now, we are not going into detail of how any of these methodologies work, that alone would take a whole lecture.
- Today, we are only interested in the life cycle aspect of these methodologies and nothing else.
- While you should have a look at the following diagrams in your own time, the main point I want to make here is that all agile methodologies have a life cycle model that is iterative and incremental

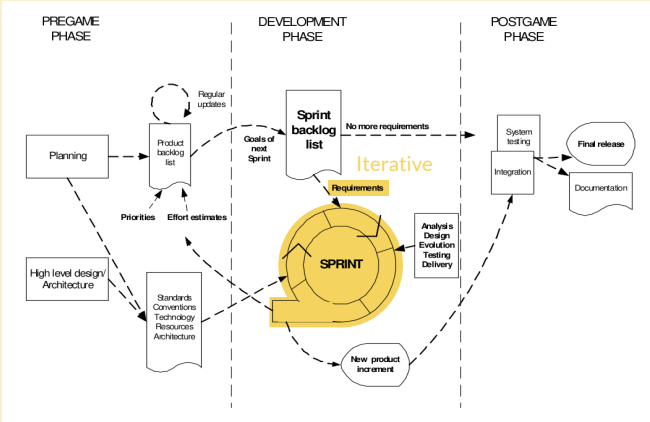
SCRUM



Abrahamsson, P., Salo, O., Ronkainen, J. & Warsta, J. (2002) Agile software development methods: Review and analysis, VTT publication

- The first agile life cycle we look at is from SCRUM, you can see a diagram of it here. SCRUM takes its name from rugby of which I know very little, but a lot of vocabulary is clearly taken from that sport.
- The overall project is divided into three phases, of main interest to us is the middle one, the development phase.

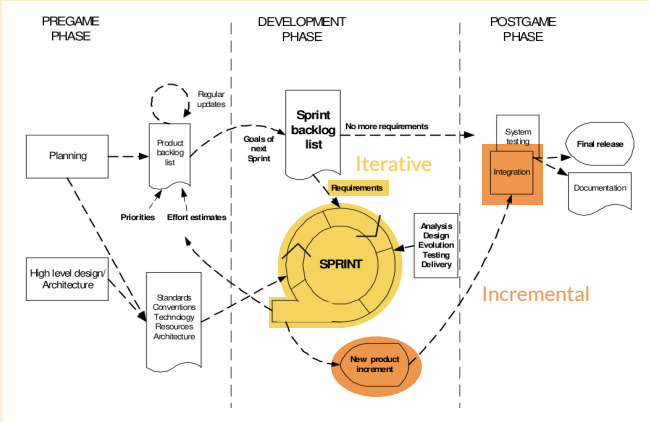
SCRUM



Abrahamsson, P., Salo, O., Ronkainen, J. & Warsta, J. (2002) Agile software development methods: Review and analysis, VTT publication

- Note how the central workflow (called a "sprint") is an iteration guided by requirements which goes through the phases Analysis, Design, Evolution, Testing, and Delivery.
- Here, "Evolution" simply refers to implementing the design into the existing software.
- Note that the SCRUM literature uses slightly different terminology for these phases, I have chosen to use diagrams from a source that tries to use consistent vocabulary across the different methodologies.
- In any case, we see that SCRUM works iteratively.

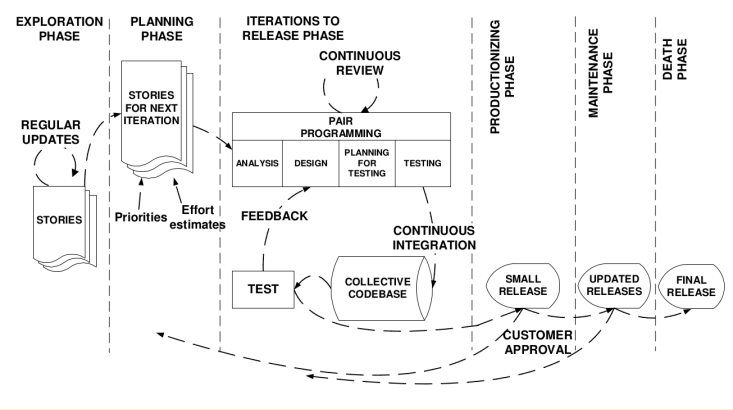
SCRUM



Abrahamsson, P., Salo, O., Ronkainen, J. & Warsta, J. (2002) Agile software development methods: Review and analysis, VTT publication

- To see that it works incrementally, note the boxes that say "New product increment" and "integration".
- One core idea of SCRUM is that versions of the software are frequently deployed and integrated.
- We can conclude: the life cycle model for SCRUM is fundamentally based on the idea of iterative and incremental development.

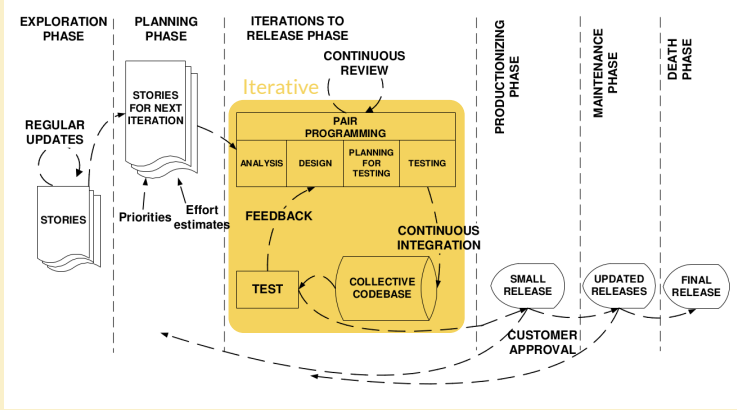
Extreme Programming (XP)



Abrahamsson, P., Salo, O., Ronkainen, J. & Warsta, J. (2002) Agile software development methods: Review and analysis, VTT publication

- Here we see the life cycle for Extreme Programming or XP, another agile methodology.
- As a sidenote from someone who grew up in the 90s: "extreme" was just one of these words that was hip and using the letter X was as well, so please do not take this name too seriously.
- XP divides the life cycle into six phases
-

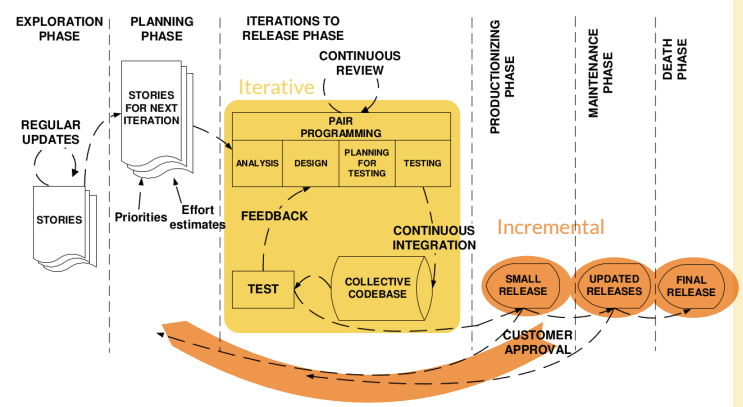
Extreme Programming (XP)



Abrahamsson, P., Salo, O., Ronkainen, J. & Warsta, J. (2002) Agile software development methods: Review and analysis, VTT publication

- The central phase is called "iterations to release" and in it we see a major cycle from programming, to integration, to testing and back again.
- This tells us that XP works iteratively, as claimed.

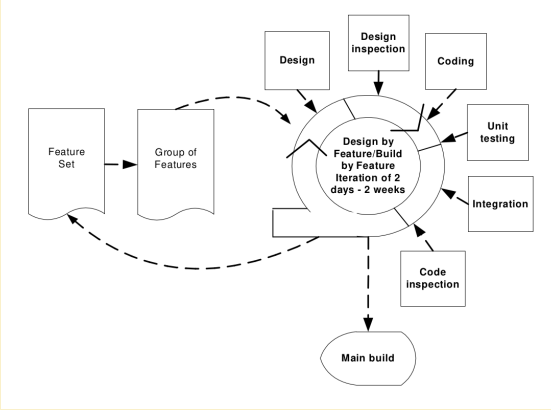
Extreme Programming (XP)



Abrahamsson, P., Salo, O., Ronkainen, J. & Warsta, J. (2002) Agile software development methods: Review and analysis, VTT publication

- Around that, there is a bigger cycle from the Planning phase to the iterations phase to the "productionizing phase" and back.
- This tells us that XP works incrementally: small releases of the software are implemented and reviewed by the customer

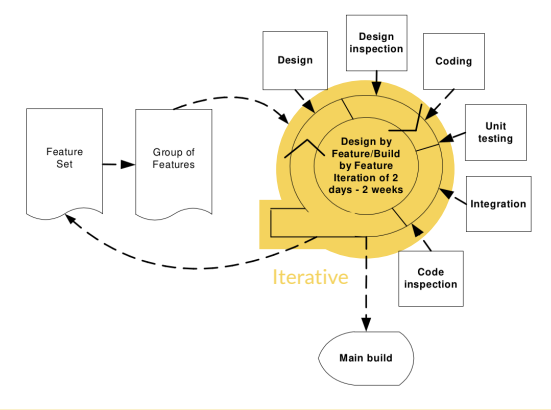
Feature-Driven Development (FDD)



Abrahamsson, P., Salo, O., Ronkainen, J. & Warsta, J. (2002) Agile software development methods: Review and analysis, VTT publication

- Finally, this is life cycle for FDD or Feature Driven Development. Here "features" roughly means the same thing as "requirements"—an annoying habit of new methodologies is to invent new vocabulary for existing concepts.

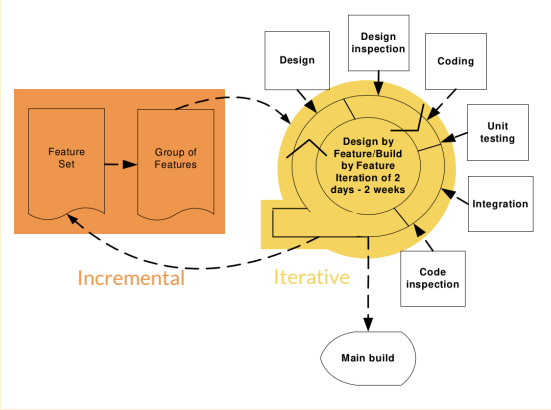
Feature-Driven Development (FDD)



Abrahamsson, P., Salo, O., Ronkainen, J. & Warsta, J. (2002) Agile software development methods: Review and analysis, VTT publication

- Again, we find an iterative procedure at the heart of the life cycle model with the phases Design, Design inspection, Coding, Unit Testing, Integration, and Code inspection.
- By now you should see that there is a certain similarity to all these life cycles; and let me remind you that this is exactly what De Bruijn and Herder tell us: if one follows a hard systems approach to design, there is only so much possibility for variation.
- In any case, we see that FDD works iteratively.

Feature-Driven Development (FDD)



Abrahamsson, P., Salo, O., Ronkainen, J. & Warsta, J. (2002) Agile software development methods: Review and analysis, VTT publication

- It is maybe less obvious that it also works incrementally: the idea is that every iteration implements a new set of features (requirements) as seen on the left here.
- If each iteration adds new features to the software, clearly we are working in an incremental fashion, so FDD is both iterative and incremental.
- What I find interesting here is the explicit timeframe of at most 2 weeks per iteration. Here we see very clearly why iterative approaches were impossible in, say, the 80s: without the added computing power available in the 90s, going through an iteration *that* quickly would not be feasible.

17

The agile manifesto

- This concludes our excursion into agile methodologies.
- The development of these "agile" approaches culminated in the publication of the "agile manifesto" by a group of developers and managers
- The screenshot here is from the original 2001 website agilemanifesto.org. I have some thoughts on the iconography of that background picture, but sadly we cannot really go into that—I just included some famous paintings as a comparison.
- The agile manifesto consists of four "values" and twelve principles. I uploaded those to moodle, here it will be enough to look at the four values:
- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan
- These values are very much a break with earlier methodologies and there is an implicit accusation here that these values are *not* part of those methodologies.
- So the manifesto claims that project management valued processes and tools over individuals and interactions, documentation over working sofwtare, contracts over collaboration, and that they follow a plan stringently instead of being flexible and open to change.
- Put like that, you can sense how stifling developers found the late 90s managment approach which the agile movement tried to reform

18

> "To some extent, this can be seen as a conflict between management, which tries to keep control of the projects and processes, and developers, who derive job satisfaction from self-development and dislike those very controls but try to manage the complexity through self-organization within a wide framework. Process models mostly take the management view, describing expectations of how the work is to be performed (which is not necessarily the way the process is performed in reality)"
>
> Kneuper R. Sixty years of software development life cycle models. IEEE Annals of the History of Computing. 2017 Sep 20;39(3):41-54, p. 50.

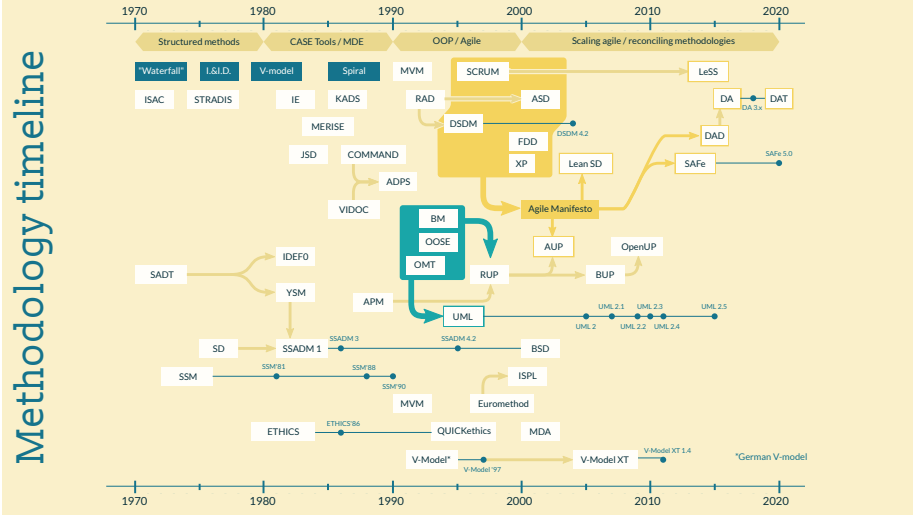
- According to Kneuper, the agile movement was as much a re-thinking of the software development process as it was a pushback on rigid management:
- (Quote)
- The highlighted part relates to what we discussed in the first week, the mismatch between map and territory, or model and reality:

- Developers felt that their management had an outdated model of how their work looked like and accordingly management decisions were seen as conflicting with the day-to-day activities.

19

- In terms of life cycle models, not much has changed in the last 20 years, altough there were major technological developments.
- We have seen a proliferation of different computing devices, starting with smart phones, tablets, and recently also wearable devices like smart watches
- Programming has become even more diverse with many new languages and ecosystems for specific areas, like Rust for safe high-performance computing or Julia for scientifict applications.
- Storage space, computing hardware and network access is now so cheap that much of our computing and storage needs has moved into data centers, the famous "cloud"
- Despite all of these developments, no new life cycle models have arisen. The consensus right now is that iterative and incremental development is the "best" way for design projects.
- Methodologies are still being developed, however. The current big issues seems to be an attempt at reconciling Agile methodologies with the more stringent methodologies of the past.
- The driving factor here is that Agile methodologies do not work well for very big projects, so "agile at scale" has become the new frontier.

20



- This concludes our tour through various life cycle models and methodologies.
- One of the important take-aways here should be that past life cycle models were not "bad" or inferior to modern methods: they simply were made in a time where software development was a very different type of activity.
- I like the concept of "chronological snobbery" for situations like this:

DON'T BE A CHRONO-SNOB!

- (Quote)
- So object-oriented programming and agile methodologies were not revolutions that everyone in the software world had been waiting on for decades.
- As we have seen, the core ideas in these paradigms are actually quite old.
- However, these ideas were only practical once the technology was advanced enough and could only be implemented once an appropriate societal context (here: certain attitude shifts in management) existed.