

# Software Process Iteration in Action

by Allan Baktoft Jakobsen

*The Waterfall model has dominated software development for many years, but iteration of processes is catching in. There is now a number of well-established iterative development process models that can be classified according to the levels where iteration is applied. Iteration can improve validation and verification by allowing earlier quality feedback. Moreover, there seems to be a secret marriage between teamwork and iteration. Altogether, from a SPI point of view, changing to an iterative development process model could very well raise your professional standards in software development.*

## Waterfall rules!

Don't *hate* the good old Waterfall model! There are very good reasons why it has become so common around the world. I believe it has to do with some fundamental intellectual activities that are related to doing just about any task. Consider the following *fundamental process* consisting of four basic activities:

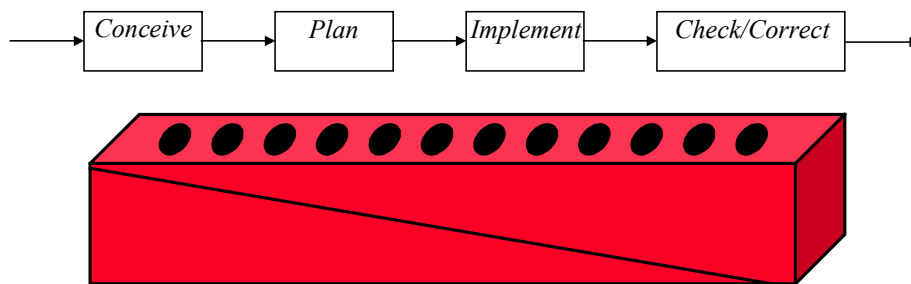


Figure 1: A fundamental process.

First, the task needs to be *conceived* i.e. you got to find out what it is all about and image how things will look like when the job is done.

When you have a clearer picture of what to do, you need to *plan* your activities. Planning is time spend *today* to save time *tomorrow*. Is the task feasible with the resources your have and are there any particular order things should be done in? Planning is pretending to do something and making mistakes in you mind is usually cheaper than making them in reality.

Now you can't sit there dreaming all the time. You need to *implement* your plans and produce something. This activity is very concrete since you can actually see the results of your work and more important: Other people can see the results, too.

Having implemented your plans you are hopefully very close to completing the job. But you are not perfect, so you may want to check what you have done and correct the mistakes. This final activity is where you check the quality, that is, how the expectations to the job are met. This includes at least two checks: *Validation* of the job conception and *verification* that the implementation was as planned.

The Waterfall model is an instance of this fundamental process: First, the developers find out what to do, then the design is planned, the code is implemented and finally tested. Each phase may take several months and should be completed before continuing. Otherwise, mistakes will occur in the next phase and rework is then urgent. Rework is considered waste of time and should be avoided. From a project management point of view, the Waterfall model is simple: It's is easy to explain since the phase-skeleton is rather non-complex.

These arguments seem to have persuaded a lot of people in the pre-historic time of software development. So when you ask people today why they use the waterfall model their best argument is "*because we always have*". But meanwhile the world has changed.

An alternative approach to software development is iteration. Before seeing the advantages, it is worthwhile explaining what it means.

### What is process iteration?

I have heard many people use the term *iterative* about their software development. It always takes some time before I understand what they really mean and when I do, it sometimes just means doing things as they come without much planning. I don't get *mad* anymore but that's not exactly what I mean by iteration.

Remember how mathematical equations are solved by iteration? First, you guess a candidate to the solution. Then you put it into an algorithm (i.e. a mathematical process) which produces a second approximate solution. Eventually, by using the *output* from one turn as *input* to the next, your algorithm brings you nearer and nearer to the exact solution (hopefully). When you are sufficiently close, the iteration can stop.

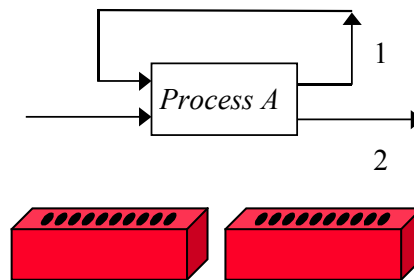


Figure 2: Iteration.

Writing an essay is usually done in an iterative manner, too. The first input to the process is a blank paper and some ideas. Then you start writing something. At some point you'll stop and read it all over from the beginning. This is the first turn of the iteration. You're using what is just written as input to the next turn where you may write some more or make some modifications. This continues until you find yourself reading the essay again and again making less and less modifications. Then you should stop the iteration if you don't want to waste time.

Software process iteration works the same way. The same process uses output from one turn as input to the next. This means changing job assignments from *Make a system that can do all this* to *Modify the system so it can do a little more*.

Now, the big challenge of software development is *coordination*, that is, *communication of decisions*. If you want to make process iteration work within a group of people, you need more and more formality the larger the group is. Everybody needs the same picture of how the processes are iterated. The turns of the iteration should be marked strictly e.g. by some event. Moreover, effective iteration means optimizing the number of turns which requires the right *stop criteria*.

### Why iteration?

Now, why would you consider complicating your development process by using iteration? Well, there are several reasons for that.

In the *Antics* of software development ( $> 2^5$  years *Before Windows*) computers were only for the high-priests of technology. But eventually in the *Middle Ages* ( $2^3$ - $2^4$  *BW*) trading of software products evolved and ordinary customers and users started discussing the holy issues of software functionality.

A number of commandments were formed to control the software religion: *Thou shall not see Gods face* meaning *The implementation of the system is none of your business!* Requirement specifications were purged for everything indicating how the customers imagined the software to look like: Implementation independent specifications were the big thing. This was done in the *name of quality*, but in fact quality, meaning matching customer expectation, suffered.

With the introduction of *graphical user interfaces* (e.g. *Windows*) it got much easier to use the computer and the software religion grew faster and bigger than ever.

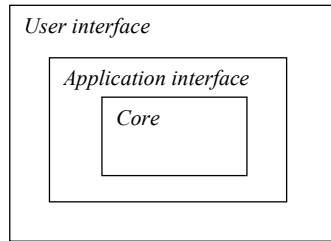


Figure 3: System levels.

Software systems got a human face and were no longer mysterious black boxes restricted to the few. The surface of the systems got much larger giving the customers and users much more to discuss. But that also meant more concrete expectations.

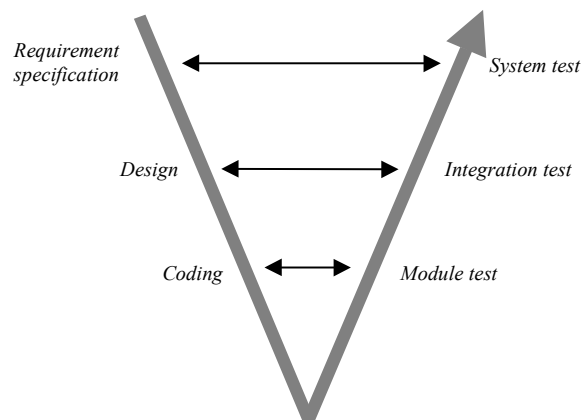


Figure 4: The V-model.

In the old days, software developers would say to the customers: Thank you for the requirement specification – see you in 9 months. They would then begin the journey down the valley of the V-model transforming the requirements to lines of code and up again checking the system.

It is important to understand that the V-model is a *Validation and Verification model* – not a development process model. Of course, the *vertical* direction looks like the Waterfall model but the big hint is the *horizontal* links in the V-model: That you can start preparing your module test simultaneous with the coding and that you can start planning your integration test simultaneous with your design, etc.

The V-model does not imply the Waterfall model and it does not imply that you should wait 9 months before you show the system to the customer and ask him if this was what he had in mind. On the contrary. The V-model says you can *validate* the system very early. It is just a matter of techniques.

Iteration is a generic technique that can give the early *quality feedback's* suggested in the V-model.

### High level iteration

High level, horizontal iteration in the V-model means earlier *validation* of the system. This minimizes the risk of building the wrong product, e.g. a table instead of a chair.

The challenge is to enhance the communication of expectations from external people (customers and users) to internal people (developers) in the software company. *Joint Requirement Planning* and *Joint Application Development* are techniques of bringing different worlds together.

The goal of communication is to build a common idea and the fundament for this is sharing the *context*. Abstract ideas are usually extracted from specific examples and without this background knowledge they are hard to understand. Concrete ideas are easier communicated since they require less background knowledge.

This is how *prototypes* and *scenarios* work: A concrete and detailed model, representation or story trigger the domain conception much better than an abstract implementation independent specification.

The process to be iterated could now be: Consider domain, write scenario, make prototype, show it to user. Next turn: Reconsider domain, modify scenario, modify prototype, show modifications to user, etc. *Design for usability* is the issue here.

Prototypes look nice, are easy to build, and easy to modify. But usually the inside structure of a prototype is too weak to base further development on.

### **Low level iteration**

Low level, horizontal iteration in the V-model means earlier *verification* of the system. This minimizes the risk of building the product wrongly, e.g. a chair that collapses.

The process you are iterating could be this: Consider functionality, design, code, and test system. Next turn: Reconsider functionality, modify design, modify code, and retest system, etc. Typically, there are two kinds of activities: (1) introduction of new functionality and (2) modification of previous introduced functionality.

There are several advantages: Earlier quality feedback – if something is done incorrectly you can soon and quickly locate and solve the problem. Working with smaller parts of the system in smaller steps allow the developers to maintain *intellectual control*. The concrete experiences from the implementation help to better understand the job.

*Incremental development* is when a sequence of sub-processes (each turn of the above iteration for example) is spitting out compact parts of the product where some limited but closed family of features has been completed (i.e. designed, coded, and tested). The sequence of sub-products will hopefully converge toward the final product. In this case you could consider an external release of each increment and perhaps make some early money. Or you could just make it an internal release: partial deliverables are very visible milestones in a project. Thus, incremental development is a step-wise refinement of the system in such a way that the system at all steps appears as a whole although limited system.

To make this *system driven* or *feature driven* software development work, intelligent *vertical slicing* is needed and this requires careful *planning*. Beginning with the core functionality and then adding less important features is much smarter than coding in alphabetical order. You really need a *construction plan* to succeed with incremental development. This plan defines the contents and order of the increments by slicing the system architecture properly. If you are using high level iteration as well as low level iteration in a project, the construction plan is the bridge between the scenarios/prototypes and the increments.

Modelling the system using the proper building bricks is very important. This could be modules, components or objects. It is recommended to define the modules in such a way that external features can be incrementally added focusing on adding new modules instead of modifying too many old. This is an architectural challenge, especially during the initial construction of the core modules, but the reward is reduced coordination and less rework. Distributing the responsibility and tasks in terms of modules defined this way in fact make project management easier.

The sequence of subprocesses defining the incremental development is the iteration of building the modules, and the perfect construction plan is then integrating the prioritized features of the system, the system architecture, and the responsibility of the various human resources in an optimal way in terms of the defined modules.

### **Classification**

Several development process models use iteration either at the high level, at the low level or at both levels. The process that is iterated is in nature the fundamental process mentioned above. Using the fundamental process as the basic *building brick* you can build or classify most standard development process models by putting the bricks after each other (by iteration) or on top of each others (at various levels).

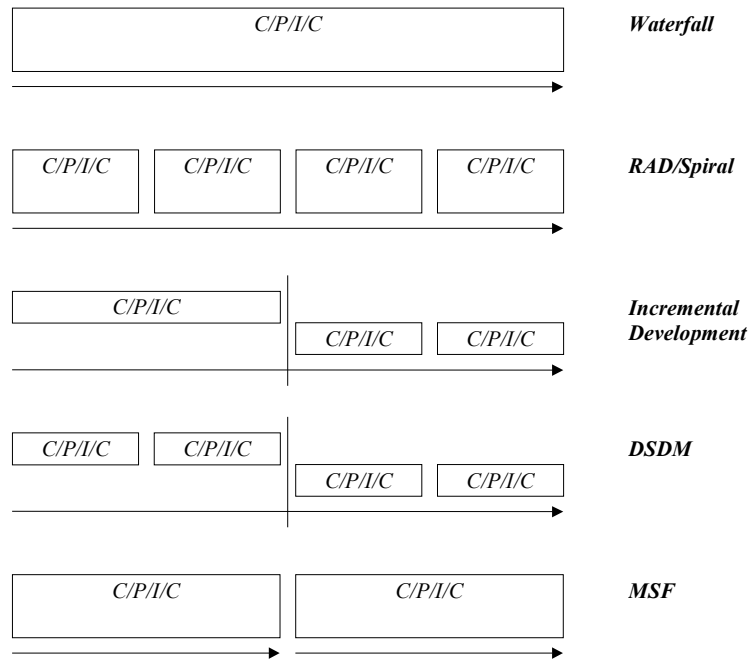


Figure 5: Classification of development process models.

RAD, *Rapid Application Development*, is pure iteration where you try to understand the requirements and try to produce code right from the beginning. You get code fast, but you must be willing to accept a lot of rework.

*Incremental development*, originally, is the old Waterfall model at the high level of the development, but iteration with partial deliverables at the low level. This is a half RAD based on a construction plan.

DSDM, *Dynamic Systems Development Method*, is two cycles of iteration: Prototyping at the high level and incremental development at the low level. DSDM is the modern, *customer friendly* development process model. (In fact, there is a third cycle where the final product is introduced to the users.)

MSF, *Microsoft Solution Framework*, can be described as a slow RAD where each version is stable enough to be sold. Making money of each version is obviously an important Microsoft strategy. MSF is perhaps even better described as *evolutionary* development based on the Waterfall model.

Of course, there is a lot more to say about the above models and the terms used here are sometimes used differently elsewhere. For example, the term RAD is often used in a much broader sense. The issue here is what the models have in common.

### Timeboxing

Timeboxing is a time planning technique, not a development process model. The idea is to go from *process-driven* activities to *time-driven*.

When the *driver* of some activity is the process, the activity is not complete before process is completed. And the process is not completed until the task is completed.

This means for example, that your design activities can be not considered completed until all of the functionality is described in the design documents. You finish one process before you start the next.

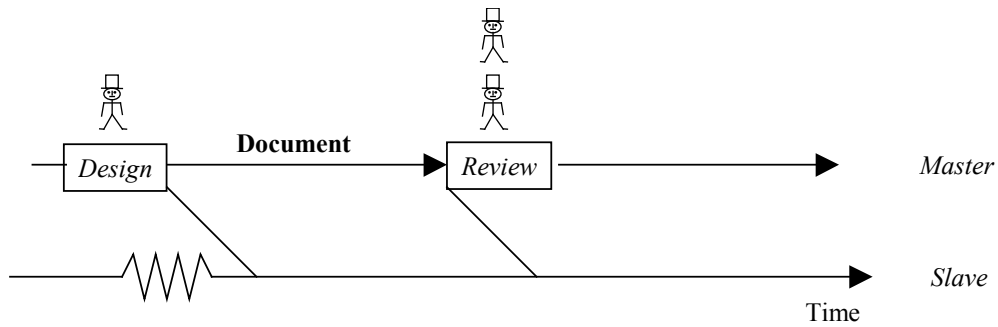


Figure 6: Process driven activities.

In consequence, if the activities are delayed, the plan starts sliding. You can easily recognize organizations where the activities are process driven: If the agreed task is not complete or *perfect enough*, it is accepted that meetings are delayed, deadlines are moved, etc.

In a timebox, the deadline is fixed, but the contents can be changed. This means that if you are working with a number of functionalities in a timebox and you can see that there is not enough time, you will skip some functionalities instead of sliding the deadline. This way, time boxing is a countermeasure to *creeping featurism*, but it requires a lot of discipline.

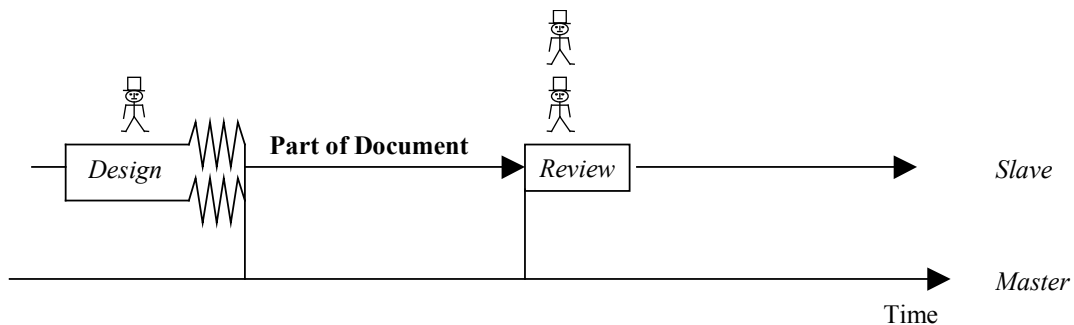


Figure 7: Time driven activities.

The timebox deadlines are the *pillars* of the project, and as you are *not* allowing them to move, you are stabilizing your time planning. If the activities in this way are time-driven, a developer can not get a review delayed but will be requested to meet with what he have which hopefully is the most important.

*Prioritizing* is obviously essential when timeboxing. The less important functionalities are the ones to be skipped if necessary. And who is the better to do this prioritizing: The *customer*, of course! That's why time boxing is tighten to high level iteration techniques where the customers and users are encouraged to make early validations. If the developers understand the users in a usability test over some scenario this knowledge is valuable when prioritizing later. In DSDM for example, timeboxing is tightly integrated with JAD sessions.

A rule of thumb seems to be that timeboxes should be relatively short when you are doing low level development (3 weeks) and a little longer when doing high level development (6 weeks).

Not all activities are suited for timeboxing. Typically, safety critical activities and to some extent knowledge transfer are not to be time driven. A solution is to insert process-driven periods between the timeboxes. These periods are sometimes called *plasma periods* (or *inter-incremental slots* in incremental development) and are used for baselining, knowledge search, and tool setup.

Furthermore, timeboxing may increase productivity, but many deadlines could stress your developers to a degree where any productivity benefit is lost. Again, it should be considered to insert relaxation plasma periods between the timeboxes, or even to add a *relaxing/cleaning/stabilizing* timebox at some point.

If several timeboxes are running in parallel, they should be in phase, that is, the deadlines should be synchronized. Here, a *bus* of plasma periods are suitable for some total project coordination, which might otherwise be lost during the pursuit of individual time box goals.

Timeboxing and iteration obviously goes well together: A timebox is a perfect way to mark a turn of an iteration, e.g. an increment.

### Teamwork fits

Al though processes are important, people is still the essential ingredient in a software project. So how does people fit into an iterative development process. Very well, in fact.

First, let us praise the Lord for diversity: Various people have various talents. Some are very good at getting ideas, some at planning. Some love producing and some are good at following-up. The fundamental process above requires at least four quite different types of personality in order to work effectively. Luckily, most people can play more than one role, but few can play all roles equally well. This is a matter of *process competence*, an important concept not to be confused with *technical competence*. A person can be an excellent designer by nature without knowing anything about the actual system.

The few but long phases of the Waterfall model may cause problems. If a number of people are producers who like working towards concrete goals, they will be frustrated in the initial half the project where *nothing* happens except understanding the requirements. We could be talking about 4 months – plenty of time to find another job. And visa versa: Good administrators sometimes detest the coding phase.

If you iterate your processes, you may turn the frustration into synergy. How?

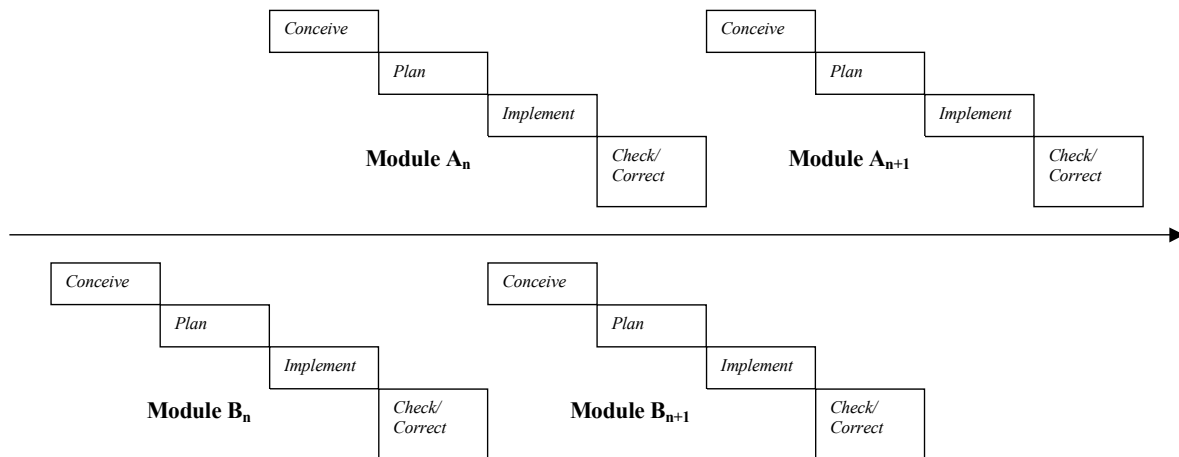


Figure 8: Parallel iteration and teamwork.

Consider the *parallel iteration* suggested above. A team of 4 persons are developing two modules ( $A_n$  and  $B_n$ ) every 4 weeks. This is done in parallel by iterating two *out-of-phase* Waterfall processes. Now by allocating the tasks properly, your entrepreneurs and planners as well as your producers and checkers could be doing what they are best at regularly (every two weeks in this example). This is done by sharing the responsibility for the modules. Moreover, the producers and checkers will be dependent of the input from the entrepreneurs and planners, who by iteration will be dependent of the output of each iteration turn. Suppose that this happens every 4 weeks and that the interdependency is crystallized at common review meeting.

You begin with a construction plan combining the right people with the right timeboxes and what you eventually get (with a little luck) is a *success machine* at the individual level and at the group level. This is *planned synergy* at it's best and believe me: It works! This *secret marriage* between iteration of processes and teamwork is an important clue to optimizing software projects. And that could be the essential factor for your company success. A well functioning team is a tremendously strong organizational structure. In fact so strong, that has been seen to cause problems in relation to the surrounding organization. Rotation of employees from team to team may be crucial.

If the project consists of several teams running timeboxes in parallel, the overall project leader can relax during the timebox periods. Here, he can leave most decisions to the almost autonomous teams, just checking progress now and then, for example at a midterm meeting. Inside the team, typically, a formal team leader is not really

needed but the processes should be stabilized by a process manager. When the deadline approaches, the project leader suddenly find himself very busy following up on the timebox planning.

Altogether from a SPI point of view, changing the development process model is fundamental. You'll soon find yourself engaged in improving your *specifications*, your *system architecturing*, your *teams*, your *reviews*, and your *testing*. You will harvest the mere benefits of increased *process attention*. In a sense, the development process model is like a devil: Give it a finger and it'll grab the entire arm raising your professional standards in software development.

#### **References:**

1. Jennifer Stapleton: DSDM, Addison-Wesley 1997.
2. DSDM Manual version 3, DSDM Consortium 1998.
3. Microsoft Solution Framework: Reference Guide, Microsoft 1996.
4. K. Van Camp: Why RAD is BAD, Object Expert, August 1996, pp.68-70, SIGS Publishing 1996.
5. Steve McConnell: Rapid Development: Taming Wild Software Schedules, Microsoft Press 1996.
6. James Martin: Rapid Application Development, 1991.
7. Barry W. Boehm: A Spiral Model of Software Development and Enhancement, ACM Software Engineering Notes, Aug. 1986, pp. 14-24.
8. Charles L. Biggs: Managing the Systems Development Process, Touche Ross & Co, 1980.
9. John M. Carroll: Scenario-Based Design: Envisioning Work and Technology in System Development. John Wiley & Sonds, 1995.
10. Allan Jakobsen: Bottom-Up Process Improvement Tricks, IEEE Software, Jan./Feb. 1998, pp. 64-68.

*Allan Jakobsen has an educational background in mathematics and physics from Copenhagen University and Dartmouth College. He has been working in the software business with maintenance, design, coding, test, quality assurance, and as an SEPG member. He is now working for the Danish company DELTA as a consultant in best software practices and process improvements.*