

Process models for modern software development

a comparison of eXtreme Programming and the Spiral Model

by Niklas Saers, March 2002

The dot-com crisis exemplified that even IT projects are bound by constraints. Investors want to ensure that they will get a return on their investment and that their investment does not go down in the drain of a failed development effort. To ensure the success of a project, we need process models that provide a framework for leading a development effort to success. This article will define what a process model is in systems development and quickly summarize common process models. Two articles describing the spiral model and extreme programming will be summarized and contrasted using the ten principles of modern software development set up by Royce Walker. Based on this it will see on the different attitudes towards management the two models have, and explain why it is not possible to say if the one is better suited than the other.

Process models

The software process spans the entire life of a piece of software. There is no limit to the ways of organizing a software process. To simplify the thinking about software processes, there has been made different software process models. A model is different from a method in that it is an abstract framework on which a method can be based. A method is a concrete organization of tasks that need to be executed in order to reach a defined goal. Although process models vary widely, the following fundamental activities are common to them all: the specification, design, coding, validation and implementation in a production environment. In addition, since the process model spans the entire lifetime of a piece of software, many models include evolution, maintenance, and phasing out the software. [Sommerville, 2001: ch3]

Although this article will compare the spiral model with extreme programming, we must first have an understanding of the tradition that these two models come from. As said in the introduction, models are shaping methods. Each software project needs a method, as no project is developed in a void of context. We will therefore first outline the following process models: the code-and-fix model, the stagewise model, the waterfall model, the evolutionary model, the transform model, and the incremental model.

The code-and-fix model used in the earliest days of software development was an iterative process of coding and fixing the problems in the code. While this method is a favourite among computer science students, it has a number of problems on large scale projects: the numerous fixes lead to poorly structured code, very often the produced code does not solve the users problem, and the maintenance costs are high due to the lack of preparation for testing and modification. [Boehm, 1988:61]

With the introduction of assemblers and compilers for higher level programming languages, it became possible to write large systems. Problems of testing and documenting the systems led to the introduction of the stagewise model. This model has eight stages that software development needs to go through in a linear order. The model is heavily document-driven: Royce gives the example of 15.000 pages for a system consisting of 100.000 instructions. [Benington, 1987: 305-310]

The waterfall model is a refinement of the stagewise model. The two primary enhancements are feedback loops between the stages and the building of a prototype. The feedback loops is the realization that the current stage is an evaluation of the work done at the previous stage, and findings in this stage can force us to go back to a previous stage to do changes. The “build it twice”-philosophy of the waterfall model builds a prototype

to explore difficulties such as technical uncertainties or to gain understanding of how the user wants the interface. The main problem with the waterfall model is the criteria of the completion of fully elaborated documents. While this criterion is good for products that can easily be formalized, such as compilers and spacecraft controllers, this does not work well for end-user applications. Since the uses and interfaces of end-user applications are often poorly understood by the development staff in the beginning of a project, there is no purpose to writing the specifications in great detail. [Royce, 1970]

The evolutionary model assumes that there is software out there that with modifications can become the software the user wants. This approach matches well with in situations where the user knows he or she needs a product but is not certain about how this product should be. This kind of development has been made less costly with the introduction of fourth-generation languages. There are mainly three problems with evolutionary programming: (1) like the code-and-fix model, the code can easily turn into an unmanageable mess. This can lead to temporary fixes and features that the programmer assumes are nice but which are not used in the product. (2) The model assumes that the operational environment of the customer is flexible enough to follow the evolutionary path the process takes. (3) Like code-and-fix, the system can become poorly documented and structured, which will lead to problems when it has to be integrated with other applications or is going to be phased out and the data need to be transferred to another system. [Boehm, 1988]

The transform model approaches the problem of spaghetti code that easily results from the many of the before mentioned models. It assumes that formal specifications can be transformed into code. It is an iterative process where the first goal of the first iteration

is to make a formal specification with the best initial understanding of the system. The following development iterations are spent changing the specifications based on the customers input and optimizing the code by giving more detailed guidelines to the tool converting the specifications into code. When the program is deployed, further iterations are used to adjust the specifications based on operational experience. In this way the transform model bypasses the problem of poorly structured code that is hard to modify. While this sounds very promising, it has proved hard to build tools that will convert a specification into complete code for other than just small projects.

The incremental model [Mills et al, 414-77] gives the customer the ability to delay decisions until he has some experience with the system on which he can make the decision. The customer identifies which parts of the project are the most important to them, and these parts get developed in early increments. Each increment is a delivery that the use customer can put into production. Each increment is a small waterfall model, and this way specification, development, verification and validation can be processes that come over and over. The customer will get the necessary documents to keep an overview of the project status. The use of increments, the system being put into production and the customer overview minimize the risk of the project failing due to miscommunication. One major problem with the incremental model is that it is difficult to estimate how large an increment should be. Also, since the requirements do not come all at once, it is difficult to identify and build general structures that will make it easier to code the system.

The Spiral Model

The goal of the spiral model is to decrease the risk of failure, where failure is either developing the wrong product, using a significant amount more money than the project budget or a significant amount more time. The spiral model has been evolving over many years, and the article that is best known for describing it is Barry W. Boehm's article from May, 1988. This sub-chapter will summarize his article with respect to explaining what the spiral model is.

The spiral model is an incremental model where a system is specified, built and implemented in cycles. Each cycle adds knowledge or functionality to the project. A round starts with determining what the objectives for the round are, what alternatives are available and what constraints the round has. The round then continues to evaluate the alternatives and identify and resolve the risks of the round. Boehm advocates the use of prototyping to evaluate the risks and identify the risks. Then comes the phase of developing the objective and verifying its completeness, and finally comes the planning of the next process cycle. A part of the verifying is staging simulations and benchmarking to see that the developed product indeed will meet its demands.

Compared to the other process models that have been mentioned in the previous section, the spiral model has a very visual representation. Boehm uses the following figure in his article:

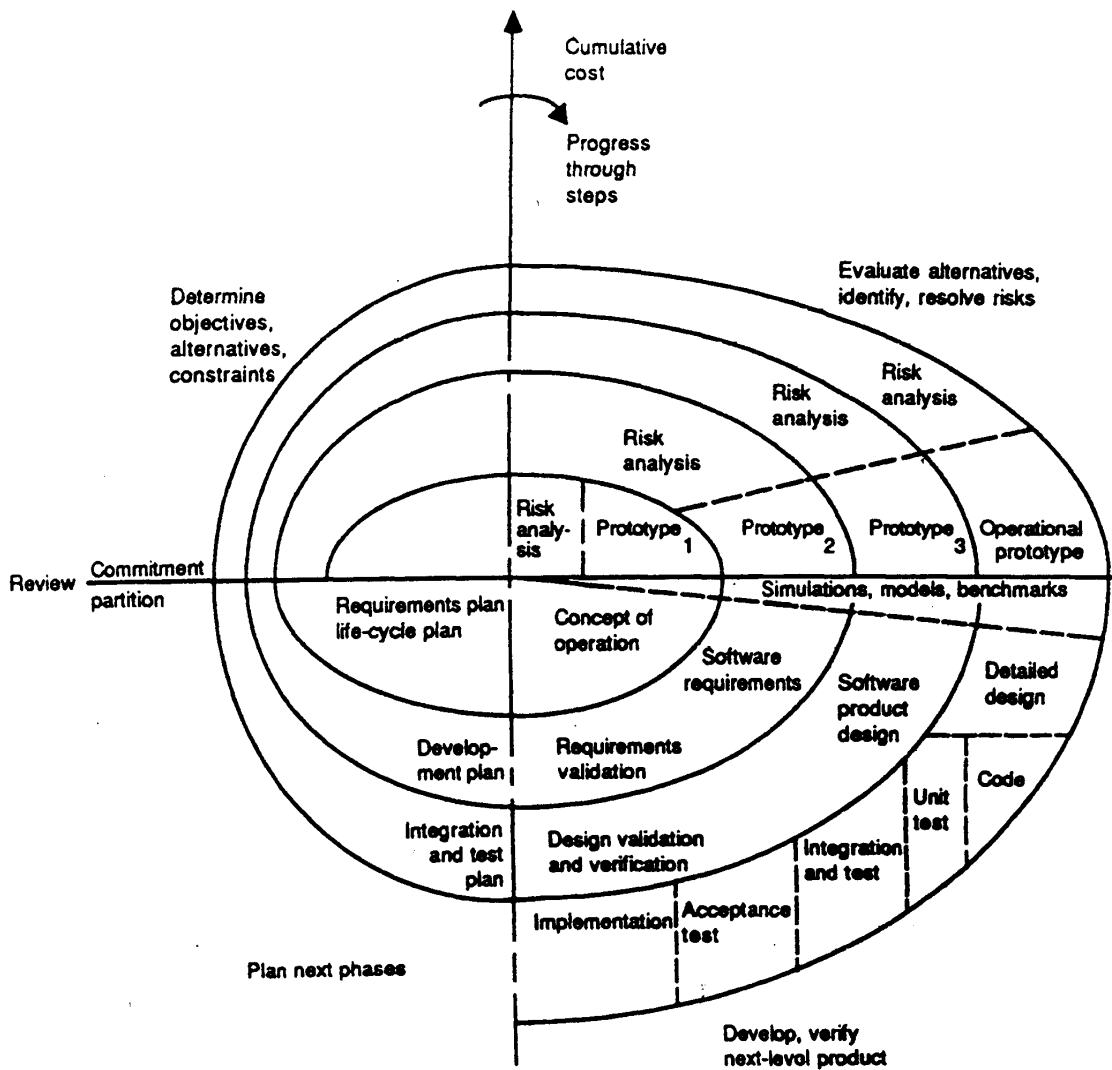


Figure 1: The Spiral Model [Boehm, 1988]

The angular dimension shows how far the current iteration has come. This figure shows the cumulative cost on the Y-axis and how much has been done on the project on its X-axis.

Prototyping is an approach on evolutionary programming where early versions through experimentation and feedback converge towards the system the customer wants

[Budde et. al, 1991: 6-9]. Boehm calls the final product of this process an operational prototype. For the customer, the operational prototype is easier to accept as this is a continuum of what he or she has been presented in the project status meetings and is therefore the most familiar with.

Note that within each cycle the project manager is free to use any process model he or she should wish to use. The development cycle will for instance for applications that require much user-interaction be evolutionary based, while a waterfall approach can be used for the development cycle when developing compilers [Boehm, 1988: 65].

Extreme Programming

eXtreme Programming (XP) reduces the cost of change through a number of practices, and exploits this to be reach the goal *to reduce project risk, improve responsiveness to business changes, improve productivity [...] and add fun to building software in teams – all at the same time* [Beck, 2000: xvi]. The article *Embracing Change With Extreme Programming* written by Kent Beck in 1999 is regarded as the introduction of XP into the business of software development. Appendix A contains a table of the 13 practices of XP with brief explanations of the individual practices. This sub-chapter will summarize how these practices together work as a good model for software development.

Extreme programming does design, implementing and reviewing at the same time and all the time. While the waterfall model has a stepwise path through the project, extreme programming designs pieces of software that a pair of programmers can

understand the full scope of and keep it clear in their mind. Pieces that do not fulfill this criterion should be broken up until the unit fulfills this requirement.

The project is planned through agreeing with the customer on stories that a successful project should implement. A story describes something the system needs to do. The different stories are given estimates in ideal implementation time by the programmers, and the customer chooses what stories are to be implemented in the different releases. If the business case for the customer changes, new opportunities are seen or the priorities are changed during the project, stories can be added or changed and priorities changed. Since the project management only has detailed plans for the current release (typically one to three weeks), the cost of change is very low and the response time of the project very low.

While each release should contain a limited number of stories, there should be frequent releases. The stories are then broken up into tasks that fulfill the requirement of scope. Each programmer then accepts responsibility for a number of tasks he or she is confident he or she can implement fully, and makes an estimate of how long it will take to implement the task. If it is discovered that some the tasks do not balance out well, they are reassigned. Then the final time estimate is given to the customer and implementation of that release starts.

The unit tests is one of the practices that to many programmers seems the most strange. In XP, a unit test is written before the code they will test. This ensures that the interface is well thought through, and that the implementation is consistent with this and behaves as expected. This also allows for others to make changes they think are right and use the tests to see that their improvements did not break anything. The tests reassure the

programmers of a job well done when they leave for the day, which in turn is a good motivation.

One design goal of extreme programming is that one piece of logic is written only once. When implementing it, if we can see a clean way to make it run, we implement that. If we think we will need an ugly way to implement it but can see changes that could be made in the system that would make it possible to implement it cleanly, we make those changes and implement it cleanly. If we can only see an ugly way that will make it work, we implement it the ugly way. If we later on, for instance when running the tests, can see a way to make the ugly code clean, we clean it up. The tests will at all times tell the programmer if the system still works as expected after having completed the changes.

Discussion – Principles of Software Management

Walker Royce published in July/August 2000 an article on the Software Management Renaissance in which he showed that software development management was having its renaissance. He listed what he considered the top ten principles of conventional software management, and went on to list his top ten principles of modern software management. While it can be discussed whether these principles are indeed fit, we will assume they are and use them to compare the spiral model to extreme programming.

Principle no. 1: Base the process on an architecture-first approach

Both the spiral model and extreme programming have an architecture first-approach. In the spiral model you are encouraged to do an analysis of the architecture you will be

basing your project upon (a so-called feasibility study). Extreme programming chooses architecture when implementing the small releases. Since releases are released early, the choice of architecture has to be made early. Early releases also lead to an evaluation of how well the chosen architecture works in the network of the customer.

Principle no. 2: Establish an iterative life-cycle process that confronts risk early

The spiral model is an iterative model, and extreme programming has made iterations so fine-grained that they happen all the time. The involvement of the customer, the analysis of stories and use of tests easily effectively confront risk in an effective way. The spiral model is itself a risk-driven model, and thus strives towards identifying and minimizing the project risk through early risk identification and resolution. The areas that contain the most risk will determine in what way the implementation phases are done.

Principle no. 3: Transition design methods to emphasize component-based development

Neither Boehm nor Beck discuss component-based development in their articles. However, as said in the discussion of principle 2, if the risk distribution suggests that it can be minimized through use of component-based development in the implementation stage, it will be. Equally the programmers in extreme programming have the obligation to think re-factoring. If it seems that the project would benefit from being broken down to components, it is their duty to make it component-based.

Principle no. 4: Establish a change-management environment

Extreme programming is built around change. Each story is a change to the product, and the task of the pair is to implement this change. In the spiral model, every iteration is a round of change, and every round the risks (and thus the chance of change being imposed) are evaluated and measures to resolve these risks are taken.

Principle no. 5: Enhance change freedom through tools that support round-trip engineering

A code-management tool often inhibits programmers through a rigid implementation. This does not necessarily enhance the freedom of change. Bannick [Bannick, 2001] defines in his article round-trip engineering as an iterative process of 8 steps that loop around generating a code skeleton, implementing and extending the code, re-reading it into the model and enhancing the model. Although XP does not mention modeling explicitly, this can be done and each implementation and extension can be seen as a little step. In the same way, larger implementation steps can make it fit nicely into the spiral model. Thus, assuming that the use of such a tool can indeed enhance the freedom of change, this works nice for both models. I have not been able to find articles that investigate whether round-trip engineering actually enhances or reduces the freedom of change for either of the two models.

Principle no. 6: Capture design artifacts in rigorous, model-based notation

I understand this principle to be referring to how CASE tools can change the specifications into code, and thus being something that should be applied both to the design and implementation phase. Neither extreme programming nor the spiral model

suggest how the design should be done nor what kind of programming languages that should be used.

Principle no. 7: Instrument the process for objective quality control and progress assessment

Extreme programming approaches quality control through rigorous testing and validation of releases by the customer. The use of frequent releases makes it easy for the customer to follow the progress. As each of the releases fulfills a number of stories provided by the user, there is a finite end in sight for the user to compare the releases to. In the spiral model, however, the customer does not know how many iterations there should be and does not know what an iteration will contain before the iteration has completed determining its objectives and constraints. Quality control is handled through reducing identified risks, the validation step in the development phase and the use of active prototyping.

Principle no. 8: Use a demonstration-based approach to assess intermediate artifacts

Extreme programming requires the project to deliver versions often. Similarly, it is not uncommon [Boehm, 1988: 65] for projects run with the spiral model to use prototyping heavily and deliver operational prototypes to the customer or use them for demonstration processes. This way, the end user gets to give his feedback early. All the artifacts that are produced in the intermediate stages are therefore addresses quickly.

Principle no. 9: Plan intermediate releases in groups of usage scenarios with evolving levels of detail

Extreme programming requires the end-user to produce stories that will be implemented. As the end-user becomes familiar with writing such stories, they will contain greater detail. The spiral model does not in it self contain user feedback on scenarios, but these are measures that should be taken when developing a user-dependent system, for instance complementing an evolutionary approach.

Principle no. 10: Establish an economically scalable, configurable process

Neither of the two articles say anything about how well these models scale to larger projects, but give examples small projects and of projects that have had 17 project members and lasted for six years in the case of extreme programming [Beck, 1999: 76] and projects that had 12 project members for 18 months in the case of the spiral model [Boehm, 1988: 66-68]. While XP claims to be well suited for small- to medium-sized projects, I have not found articles on neither XP nor the spiral model being used for very large projects.

Discussion - View on management

After having explained the two models and seen the differences in comparison with the 10 principles set up by Royce, we need to see if the two models have different views on management. Management consists of planning, leading, organizing and controlling [Robbins et. al, 2000].

Extreme programming plans the entire progress through stories submitted and prioritized by the users and estimates on the stories by the programmers. The current increment is planned in detail by breaking the stories selected for that round into tasks that programmers take the responsibility for. If the customer changes his or her mind, making changes in the plans requires a minimal effort.

The spiral model has a planning phase that determines the objective for the current cycle. It has no overall plan for the number of cycles in it self, so if there is to be a plan about the number of iterations, this must be external to the model. It is worth noting that the uncertainty of the number of iterations in the spiral model and the loose planning of extreme programming makes both models little suited for fixed-price projects. In fixed-price project you need to know what you are going to do at a low risk of exceeding budget, or your company may go out of business. Therefore, both models require that they have been used much in the organization before a fixed-price project can be tried.

In his book “eXtreme Programming explained”, Beck writes a great deal about leading projects that use extreme programming. Most of his suggestions go on motivation and keeping people fresh, such as discussing issues while eating something together, having a snack after all tests for a particular task run correctly, sharing a common metaphor, having the programmers self choose what tasks they wish to accept responsibility for and having a 40 hour week. While these are not abstract ideas about leadership, they are ideas to how motivating the employees and thus saying something about how the atmosphere in a project that uses extreme programming should be. In this regard it can be said that extreme programming interferes with the organizational culture

while the spiral model, that doesn't even mention leading, can be adapted to the company culture that is in place.

Both extreme programming and the spiral model are all about organizing and controlling. Both models take a risk-driven approach. The spiral model organizes for risk to be low by spending a large amount of resources in evaluating the alternatives and identifying and resolving the associated risks. Extreme programming, on the other hand, assumes that the risks involved in coding when you code just enough to make the tests work is low. Making radical changes in the program, i.e. while refactoring, is also low-risk as the tests will indicate if any inconsistencies have occurred. The frequent releases make the risk of the customer not agreeing to the direction the development is going small and the impact if it should happen small. Tests and the verification of the user stories are thus the essential controlling mechanisms of extreme programming. The spiral model uses a verification, validation and testing part of the development stage as its controlling mechanism.

Conclusion

This essay has put process model in the context of management. When contrasting the spiral model and extreme programming in the context of the ten principles of modern software management, there were surprisingly few differences. The articles describing the models talked about the same principles, and did not explicitly mention the three same principles: principle four, five and six. Only in principle nine was there a difference in that extreme programming mentions the evolving level of detail in scenarios explicitly while the spiral model does not. But inexplicitly it is clear that this principle is not incompatible with the spiral model.

In discussing management, the differences are clearer. While the spiral model does not come with a culture itself, and is thus easily adoptable to most companies, extreme programming brings along a way of planning, a way of organizing, a way of leading and a way of controlling. Boehm only mentions what is required to do in his article; Kent gives a recipe on how to do it.

It seems thus that both models are appropriate to use in modern software management. To the authors knowledge, there has not been published any hard facts that prove that the one is better suited than the other. Indeed, there has been very little quantitative research on contrasting process models at all. Until more quantitative research has been done, choose according to your taste and see what fits best in your company culture. Extreme programming sells itself as a fun way of programming, and if your company can handle its culture, enjoy. If not, the spiral model is a good alternative.

Appendix A - The 13 principles of eXtreme Programming

Collective Ownership – Everyone takes responsibility for all the code and can modify any part of the code at any time it makes sense to the programmer. This requires that simple coding standards must be kept

Continuous Integration – Whenever a piece of code passes all the tests on the computer of the programmers, the code is integrated into the code repository and all the tests are run again to verify that the integration has not broken the system. If it should break the system, it is either fixed straight away or taken back to the computer of the programmers after restoring the integration machine to the original, running state

Just Rules – XP is just a set of rules, and while projects run quite well on these rules, it is folly to integrate them all into your organization right away. Identify the major problem in your project, replace the problem with XP rules and evaluate the usefulness. Then go on to the new major problem.

Metaphor – Each project must have a common metaphor, a common understanding of what the project is.

On-site Customer – XP requires a user representative to be on-site so that questions can be answered quickly and solutions can be evaluated on the fly. The remaining time is spent writing stories and functional tests. If the customer cannot afford to have one of its staff on-site, chances are that the project is not worth the time of more than a person to the customer, and the project should not be running in the first place

Open Workplace – Instead of each their own desk, programmers work in a large room to easily be able to make questions or help others. Programmers work in pairs on each task

Pair Programming – All the programming is done in front of one computer. One person is writing tests or code at a time while the other oversees the overall structure of the project is preserved, that the design is kept simple and thinks of tests that would be able to break the current code

Refactoring – Whenever the design is not simple, the code is not nice or code is duplicated, it is the responsibility of the programmer to refactor the code in such a way that it becomes nice and simple. The collective ownership makes a programmer able to do this, and the tests ensures that no part of the system fails

Simple Design – Logic is written only once. The code has the fewest possible classes and methods, does what the programmer wants it to do and runs all the tests.

Small Releases – A new release, adding valuable functionality, is implemented at the customer between every three weeks to a few months. Internal releases are made on every one to three weeks. The system is built every time code is integrated and all the tests run.

Tests – The customer writes functional tests that the program should complete successfully when a story is completed. Programmers write tests that the code should complete successfully once a task is completed.

The Planning Game – The customer provides stories about how the system should be usable. On this basis, stories that can be implemented within a couple of days are created. Estimations of the stories are made, and the customer prioritizes what stories should be integrated in the current release. Only the present release is planned in great detail. Programmers accept responsibility for tasks and make estimates for the implementation time in ideal programming hours (time with programming without any interruptions).

40-hour Week – No one works more than 40 hours two weeks in a row. Too much use over overtime signals a problem in the project that must be handled

[Beck, 1999] [Beck, 2000]

References

Barry W. Boehm (1988) *A Spiral Model of Software Development and Enhancement*, IEEE Computer

Budde et. al (1991) *Prototyping – An Approach to Evolutionary System Development*, Springer Verlag

Dr. Winston W. Royce (1970) *Managing the Development of Large Software Systems*, Place: Publisher

Herbert D. Benington (1987) *Production of Large Computer Programs in 9th International Conference on Software Engineering*, Washington: Computer Society Press of the IEEE, page 299-310

Ian Sommerville (2001) *Software Engineering 6th Edition*, Essex: Pearson Education Limited

John H. Bannick, Jr (2001) *Java – Round Trip Engineering*, <http://www.tiac.net/users/jbannick/java/roundtrip/roundtrip.htm>

Kent Beck (1999) *Embracing Change with Extreme Programming*, Computer vol 32, no.10

Kent Beck (2000) *eXtreme Programming eXplained - Embrace change*, New Jersey: Addison-Wesley

Mills et al, 1980, The management of software engineering (414-77, ch3)

Robbins, S., Bergman, R., Stagg, I. and Coulter, M. (2000) *Management*, Sydney: Prentice Hall.

Walker Royce (2000) *Software Management Renaissance*, IEEE-Software Vol 17, No.4