Software Development Lifecycle Models
The Basic Types

Christina Wallin
ABB Corporate Research
Department of Industrial IT
721 78 Västerås
+46 (0)21 34 50 74
christina.wallin@se.abb.com

Rikard Land
Mälardalens Högskola
Department of Computer Engineering
Box 883
721 23 Västerås
+46 (0)21 10 70 35
rikard.land@mdh.se

ABSTRACT
Throughout the second half of the 20th century, the use of computers has grown enormous. As a consequence, software has become more and more diverse and complex. In addition, there are increasing demands on software – it must be cheaper, have more functionality, be delivered faster, and be of higher quality than previously.

In the ever-changing environment and society of software development, it is obvious that the processes and methods used when developing small programs are not sufficient when constructing large systems. As one answer to this, different development lifecycle models have been defined.

This paper describes the three basic types of software development lifecycle models, from the sequential models via incremental models to evolutionary models. The iterative development method is also discussed, and we also elaborate the connection of development lifecycle models to two emerging fields in software engineering: software architecture and component-based software development. The paper is biased towards computer science students.

Keywords
Software lifecycle, software development, software engineering, sequential model, waterfall model, “V” model, incremental development, iterative development, evolutionary development, spiral model.

1. INTRODUCTION
In very small software development tasks, such as assignments in university programming courses, it is often feasible to start writing program code and “make the program work”. It is possible to have an overview over the whole program during the short period of time until the assignment is approved – and after the assignment is approved, there is often no need to extend or maintain the code.

This “code-and-fix” development lifecycle model [5] may be appropriate for course assignments, but is clearly insufficient in practical applications. We know that virtually every commercial piece of software is too large and is developed for such a long time that it is impossible for everybody to have a complete overview.

One problem is that there are many related terms in software development that may be confused with each other. Terms as: software project, product and system. System, product and development lifecycles. Sequential, incremental or evolutionary lifecycles. Iterative development. Software development method, process and model etc. Architecture, components and platforms. Consensus of the interpretation of each term is not reached yet in the software community. We will provide our definitions of some of these terms in this paper, and focus on the definition of three basic types of software development lifecycle models: the sequential, incremental and evolutionary models. In sequential lifecycle models, the development phases are traversed in sequence, with an evolutionary lifecycle model, there is a cyclic approach, and incremental lifecycle models are something in between. We compare them to each other, and also describe their relation to the iterative method of working and to software architecture and component based software engineering.

In this paper we focus on the time covering the development of one particular version of a software product or system, from requirements definition until requirements validation. We believe we will give a more clear view of software development lifecycles than when models are presented unrelated.1

1 The figures of this paper are in general intended to serve as examples of the general issues discussed in the text, and
2. SEQUENTIAL MODELS

The sequential models were the first software development lifecycle models to address software specific issues, and they are still widely used especially in large organizations. These models stem from common practice; to our knowledge no specific person or group is considered the inventor.

The sequential models build on the assumption that the problem to be solved can be completely understood and described before a solution is designed. A design that satisfies all aspects of the problem can be specified before implementation. All implementation can be done before validation and delivery. This approach can be applicable if experienced developers’ work in a well-known problem domain, with well-known technology and with short development times as for example in software maintenance.

The main risk with the sequential approach is that requirements may change during the development time and that the delivered system or product in the end is not what the customer or market wants any more. The main opportunity is that the sequential approach makes it easy to plan and follow up progress.

2.1 Pure Waterfall

The most basic of all sequential models is the Waterfall model. It seems to be the first named software development lifecycle model to be described and used. It defines some basic tasks, which are carried out in sequence: requirements definition, architecture design, detailed design, implementation, component verification, integration verification and requirements validation. See Figure 1.

Each tasks result in documents or other artifacts that are used as specifications for the next task, e.g. the detailed design specifications forms the basis for implementation task.

In the “ideal” form, one task should be completed before the next starts. There is however variants with overlapping tasks [5], and these are probably used more in reality. Implementation starts when some of the detailed design is ready, component tests when components are implemented etc.

2.2 The “V” Model

The most commonly known variant of the waterfall model, the “V” model, explicitly includes the observation that the result of each specification task must be verified in a corresponding test task. See Figure 2.

Figure 2. The “V” Model – earlier tasks are verified later.

2.3 Advantages and Disadvantages

The sequential lifecycle models have the advantage of being easy to understand, and it is easy to plan a project according to a sequential approach.

Since sequential lifecycle models are easy to understand, they are still commonly referred to, while practice has shown that they present a too idealistic view of how software development is carried out. In [1] the “old way” of looking at software development is described: “Conceptually, the requirements document gets tossed over the wall into the designer’s cubicle, and the designer must come forth with a satisfactory design.”

Early errors are commonly regarded as the most serious, because the rest of the phases must be re-entered to correct them – the compiler finds syntax errors interactively: detailed design errors are found when verifying against the detailed design specification; an architectural design error is found during the integration verification, and forces you to step back to the architectural design task again, make the corrections, redo detailed design etc. Not until the very end is the system or product validated against the requirements specification, and if the requirements are found to be incomplete or wrong, the whole piece of software may be more or less wasted.

Therefore often contain details that can be omitted without loss of understanding.
In the pure sequential models, it is not possible to verify the early tasks (for example let the customer see parts of the system) until the very end of the development lifecycle. The unavoidable, and unpleasant, conclusion is thus that the early errors are extremely costly, since you “don’t find out until you get to system testing that one of the requirements was missing or wrong” [5]. We can however observe that the tasks identified are intrinsic to software development, as we shall see in other life cycle models; the problems arise “from the treatment of these activities as disjoint, sequential phases” [5].

These drawbacks have forced more elaborate lifecycle models to evolve addressing these problems. We will now describe the incremental and evolutionary lifecycle models.

### 3. INCREMENTAL MODELS

With incremental development lifecycle models, risk of developing the wrong thing is reduced “by breaking the project into a series of small subprojects [increments]” [5]. The total scope of work is decomposed to smaller chunks of work, increments, based on the risks, architecture and/or the requirements.

In incremental models, as in sequential models, the overall requirements of the final system or product are known at the start of the development. In incremental models however a limited set of requirements is allocated to each increment and with each successive (internal) release more requirements are addressed until the final (external) release satisfies all requirements.

One risk with the incremental approach is that the first releases addresses such a limited set of requirements that the customer could be dissatisfied, one opportunity on the other hand is that wrong or missing requirements can be corrected in time.

#### 3.1 The Staged Delivery Model

In the Staged Delivery lifecycle model [5] the development work is focused on the construction of one part (subsystem) of the final product or system at the time. See Figure 3. Each subsystem is finalized and released internally before the construction of the next subsystem is started. Integration of the different subsystems can either be done continuously or in the end.

![Figure 3. Staged Delivery Model.](image)

The order in which the subsystems are developed can be selected in different ways. Either the riskiest parts are developed up front, or the fundamentals of the architecture or the most important requirements.

Construction in Figure 3 and Figure 4 means the tasks of detailed design, implementation and component test for each subsystem.

#### 3.2 The Parallel Development Model

In parallel development the different subsystems are developed at the same time but by different teams instead. See Figure 4. Integration is done in the end. This approach can decrease the calendar time needed for the development, that is the Time To Market (TTM), if there are enough resources available, but the risk to get problems with integration is at the same time increased.

![Figure 4. Parallel Development Model.](image)

The parallel development model does not support changing requirements very well – once the requirements definition phase is left behind, the requirements specification is “frozen” (cannot be changed). In this respect there is no difference between parallel and sequential lifecycle models.
4. EVOLUTIONARY MODELS
Under some circumstances, an evolutionary approach to the software development lifecycle is chosen. This is often the case in research or technology development, where often only prototypes are developed, and where all the requirements cannot be known in advance.

More sequential More cyclic
Sequential Incremental Evolutionary
Type of development lifecycle model

Figure 5. The relationship between sequential, incremental and evolutionary development lifecycle models.

It is difficult to define the exact border between incremental and evolutionary models – McConnell [5] describes them together. Incremental models may be seen as something in between sequential and evolutionary models, as Figure 5 describes.

The major difference between the evolutionary models and the previously described sequential and incremental models is that in evolutionary models all requirements of the final system or product are not known when the development starts. Initially the requirements are partly defined and then refined and extended with each successive (internal) release as the requirements picture mature and until a satisfactory solution is reached. See Figure 6.

Figure 6. A cyclic approach for stepwise refinement.

### 4.1 The Spiral Model
The most commonly known evolutionary model is probably the Spiral model defined by Barry Boehm [2], which describes a model where a prototype is refined to a product or system in a series of cycles.

The governing idea is that each cycle starts with risk evaluation, and not until the risk is resolved does the cycle continue with implementation and testing. (These tasks can very well be carried out with any development model, for example the sequential.) In the end of each cycle, there is a usable prototype, in early cycles with limited functionality, which is evaluated – if more functionality is needed, another cycle starts. See Figure 7. Note that the prototype may very well be thrown away after each cycle, and a new prototype developed until the final loop where the final product or system is created. That is the Spiral model is not necessary incremental.

The type of prototype to be developed of course depends on the specific system, but can include:
- Illustrative user interface – a “stupid” user interface is implemented and used for further dialog with the customer.
- Functional prototype – an operational system with minimum functionality is implemented.
- Exploratory prototype (“hacking”) – a part of the system is implemented to gain knowledge about the requirements.

Figure 7. A detailed example of the Spiral Model with four cycles.

### 4.2 Advantages and Disadvantages
The evolutionary model’s advantage is that it reduces the risks – if too high a risk is found, you can terminate the development, and it will have cost less than with other development models. Furthermore, the development can be terminated after any loop, and there will still be a working system.

The evolutionary model’s disadvantage is that it is complicated [5]. It is difficult to plan and follow up and it might not be worth the effort using it if the development is manageable enough, with low risks.
Another disadvantage is that the architecture of the first version of the system must support the changes introduced in each cycle. Otherwise you will have to redesign your system completely, however the experience of earlier cycles is very valuable input when doing this.

5. RELATED ASPECTS OF SOFTWARE ENGINEERING

Within Software Engineering, many research fields are strongly related. In this section we will briefly present a couple of recent trends and one good practice, and give examples on their relationship to the above described development lifecycle models. The point here is that a development lifecycle model has to be combined with other methods and principles to cover all aspects of software engineering. Our choice of subjects is due to our own research interests.

5.1 Component-Based Software Engineering

One recent trend in software development is component-based software engineering (CBSE) [9]. This research field addresses problems arising with today's large and complex systems. The key observation is that it is practically impossible for any company to build a large enterprise software system entirely by itself. A design of one system is likely to include specifications of components implemented in other systems, thus being potential candidates of reuse.

The advantages of reusing software are obvious. If it is possible to find and use a software component that fulfils the requirements at hand, it will in general be cheaper and include more functionality than an in-house component would; furthermore its quality is known and it is immediately available. In some areas there are already today products and well-developed standards that make the use of such a component an obvious choice (this is the case e.g. with operating systems or databases).

However, the CBSE community pictures a future component market embracing many more types of products. There will be accepted standards, and a variety of commercially available components conforming to this standard, each with competitive advantages.

This scenario affects the software development in two major ways:

1. System development will tend towards assembling pre-existing components, and
2. There will be a market for developing standard components.

The development of components will be a process where the requirements are well specified in the standard, and of course any development model and method can be chosen.

The CBSE community therefore pictures a development process where you either are a “System Assembler” – the former “implementation” task will then be replaced by an “assembly” task, or you are a “Component Developer”, and the former “requirements” task will partly be stated by the standards. Each of these development “lines” can use any development lifecycle model.

In Figure 8, a waterfall sequential representation of the main tasks in CBSE is sketched. The “System Assembling” line starts at the top left, and the “Component Development” line starts at the top right. The ellipse denotes the component marketplace, where these lines meet.

5.2 Architecture Tradeoff Analysis

The notion of Software Architecture appeared in the nineties as a separate field of Computer Science [1] [6]. The research in this field has given birth to formal approaches to high-level (architectural) design, but has also affected development methodologies.

In 1998, the Architecture Tradeoff Analysis Method (ATAM) was first described [3]. This informal analysis method introduces iterations between the requirements specification and architectural design tasks, see Figure 9. It focuses on so-called non-functional properties of software, and describes how scenarios affect these. A software development team will thus give a higher attention to non-functional properties when the requirements specification task is re-entered; the ATAM is explicitly said to have “both social and technical aspects” [3]. This means for example that the members of a software project will give higher attention to non-functional properties and how these are affected by the architectural design.
For our purpose, we can also note that future changes of a system are anticipated already during the requirements specification and architectural design, given that the change scenarios used are representative of actual future change scenarios. We have said that continuous changes during the development may be one problem with development lifecycle models, so if such changes are anticipated, with ATAM the development will run more smoothly.

We can summarize the impact the ATAM has on development lifecycle models as follows:

1. There are iterations between the requirements analysis and architectural design tasks.
2. The members of a development team will focus on non-functional properties to a higher degree.
3. Future changes are anticipated, making any development lifecycle model run more smoothly.

5.3 The Iterative Development Method

So what about iterative development then? This is today regarded to be a best practice in software development [4]. Yes, iterative development is a good practice but it is a development method and not a (part of a) development lifecycle model. The iterative development method is often combined with incremental development lifecycle model, but it is important to differ between the two. Iterative development is the principle of refining and reworking software components over time, while incremental development is the principle of working on one part of the problem during one pass of the development. Iterative development can be combined with any of the three types of development lifecycle models described above, and all development lifecycle models can be applied without using the iterative method.

The number of iterations needed to solve a problem cannot be planned in advance. Neither can an iteration be planned in time or scope. It is not feasible to say that analysis of an unsolved problem will take two days, and then the design will take three days, the implementation three more days and then test two days. This type of planning can only be done if the solution to the problem is known in advance. What can be said is for example that a problem has to be solved within a certain timeframe, or else the plan will slip.

The Quality-Attribute Based Architecting process description (see Figure 10) is one typical example when the iterative method is applicable. This process description fits into the architectural design task in any of the development lifecycle models described above. It should be observed that our point here is not the details of this particular process, but that it is iterative. It is iterative in that a number of activities are re-entered until a solution is found, and the process can be used inside one task of any lifecycle model (in this particular case, inside the architectural design task). That is, even a sequential lifecycle model can be combined with an iterative way of working.
Finally, we have scrutinized two recent trends, the design and development of Software Architecture through the ATAM, and Component-Based Software Engineering. We found that the ATAM method is not a complete development method or lifecycle models in itself, but complements other development methods and lifecycle models. Neither is Component-Based Software Engineering pictured in the paper; instead it also adds another dimension to the field.

6.1 Contribution
The paper is biased towards an audience consisting of students in computer science and engineering. This in mind, we have presented an overview of three types of software development lifecycles in an organized form, emphasizing their major differences, as well as their relationships.

We also presented a definition of what we believe is the distinction between incremental and iterative development.

Furthermore, we related two new research trends, architectural analysis and component-based software engineering, to the established software lifecycle models.

6.2 Related Work
The lifecycle models described are often referred to as “common practice”, the exception being the Spiral Model, which was invented by Boehm [2]. There are however a number of overviews: Sommerville surveys the field of Software Engineering, including a classification of lifecycle models [7], slightly different than ours. McConnell provides a good overview over software lifecycle models and related areas, including many “good practices” [5].

The Dynamic Systems Development Method (DSDM) [8] is a process description for software development, defining a detailed lifecycle model. Rational Unified Process (RUP) [4] builds on the same type of process description as DSDM, but is more widely known and spread.

The notion of Software Architecture has been explored by Bass et al [1] and by Garlan and Shaw [6]. The ATAM was described by Kazman et al [3]. Szyperski [9] defines and describes Component-Based Software Engineering.

6.3 Future Work
The software community is today too immature to be able to come to a consensus about how the lifecycle of software development should be described. Thus, the software community still faces a major task in standardizing the terminology, the lifecycle models, development methodologies, and not least to make the knowledge widely spread and used – and taught.

6.4 Conclusion
There are a great number of development lifecycle models and methodologies around, each of them more or less claiming to be the best. This paper has only presented a brief overview.

In industry today, the described lifecycle models are used, but often much more detailed than described in this paper, often somewhat altered, and often not fully understood and thus used improperly. Since the development lifecycle models and development methodologies are complicated and there are no absolute standards, successful software development still relies heavily on individual developers’ experience. Quality software can only be achieved “the hard way”, i.e. through careful consideration of all aspects regarding software engineering. Software development organizations of today have to mature in this area and to start to take these issues seriously.

There are indeed University courses teaching Software Engineering explicitly, including software lifecycle models, and in some other courses the assignments are complex enough to force the students to consider engineering issues. Knowledge of the available development methodologies and lifecycle models is essential to a computer science student.

6.5 Acknowledgements
We owe credit to Kurt Wallnau and Judith Stafford at the Software Engineering Institute for the ideas to Figure 8 and Figure 10, and to Stig Larsson at ABB for his valuable comments on the contents of this paper.

REFERENCES
