Reprint from: *A. Smolyaninov, A. Shestialtynow (Eds.):Proc. WOON '96/WOON '97, 1st and 2nd International Conference on OO Technology, St. Petersburg, pp. 88-101 (1997)*

From WOON to EOS:
New development methods require a new software process model

Wolfgang Hesse, Philipps University, Marburg (Germany)

Author's address:

Prof. Dr. Wolfgang Hesse,
FB Mathematik/Informatik, Philipps-Universität
Hans Meerwein-Str.
D-35032 Marburg
Germany

Tel.: +49-641-281515, Fax: +49-641-285419, email: hesse@informatik.uni-marburg.de

Abstract:

Software development goals, guidelines and techniques have changed considerably during the last decade. New paradigms and ideas such as object orientation and software evolution have led to new system architectures, development strategies, methods and tools. Software process models, however, have not yet evolved sufficiently to support the changes in process management requirements resulting from the paradigmatic and technical changes. This was recently confirmed by an empirical study carried out by an interdisciplinary team of computer scientists, work psychologists and sociologists. Many object-oriented methodologies such as OOA by Shlaer & Mellor, OOAD by G. Booch, OMT by J. Rumbaugh et al. and OOSE by I. Jacobson have improved the technical standards of software development but stick more or less to traditional, phase-oriented process structures. Thus a considerable gap has emerged between modern development techniques and conventional process models and management techniques.

A new software process model for *evolutionary, object-oriented software development (short: EOS)* is presented to fill this gap and to support modern software development by adequate process structures and management techniques. It discards completely the traditional phase structure, being based instead on development cycles and activities associated with the „objects" of software construction: systems, subsystems, components and classes. Management procedures built upon this framework allow for multi-level, adaptive, differentiated planning and control. This implies new challenges for project managers but also enables them to plan and to act in a *more adequate, dynamic and situational* way.

This report is a contribution to the International Conference in Object-Oriented Technology WOON '96 in St.Petersburg/Russia. Parts of this research have been sponsored by the Bayerische Forschungsstiftung through FAST e.V., Munich/ Germany.

1 Introduction: The dominance of traditional software life cycle models

Since *Software Engineering* was first introduced as a new term and a new discipline more than 25 years ago, efforts to improve the software development process and project management have been dominated by the so-called *phase* or *waterfall models*. Traditionally, the software process is characterized by the following properties:

- A major piece of software development is declared a „project" which is formally initiated and (at least in those cases where it comes to a certain success) formally terminated. The project is decomposed into „phases" which are executed by teams (often different from phase to phase).

- Each phase starts with a set of requirements and (except the first phase) with specifying documents obtained from the results of previous phases. Its goal is to produce a set of results (sometimes called „products") which either become part of the final result or are used as specifying documents for successor phases.

- During the middle phases, large systems are hierarchicaly decomposed into smaller units (called e.g. subsystems, components, and modules) which after a separate development are (re-) integrated, tested, installed at their target environment, checked by a final acceptance test and then released to the users.

These phase-based models have been subject to a variety of criticisms:

- The phase structure is too rigid and unrealistic; in practice, development processes contain many overlaps and cycles.

- The linear development scheme is not flexible enough to cope with instable and changing requirements.

- The demand on software engineers to produce many documents implies the danger of „software burocracy" (cf. [Den 90]). An excessive sequence of products and documents to be created may lead to redundancies and inefficiencies. A pseudo-automated development style has mostly demotivating and quality-decreasing effects. Developers feel constrained, become bored or suffer from burocratic rules and guidelines.

- Waterfall models do not sufficiently reflect the demands of new development techniques. For example, object-oriented techniques influence the sequence of acti-vities and the structure of results. Guidelines often do not focus sufficiently on preparing products for modification, adaptation to new requirements and reuse.

- Users and developers represent different „worlds". This is reflected by distinct (and often incomptible) models dominating the analysis, design and implementation phases.

- Software projects are often still considered in isolation, as independent efforts with unrelated goals. Long-term goals and interdependencies with other projects, with general business strategies or social developments are given too little attention.

Recent empirical data raised at German software houses and software development departments have shown that the majority (over 60 %) of the investigated projects still followed a linear phase model with defined project phases - at least "officially". About one third of the projects reported an incremental work procedure, another third a prototyping style (multiple answers were admitted) [H-W 94].

Why are waterfall-like phase models still so popular - if their shortcomings are so well understood? Plausible reasons for their persistence include:

- They structure the development process in a simple and transparent way.
- Planning of the overall project seems to be rather simple, as the phase structure is very transparent, coarse and linear.
- The phase schema offers a simple pattern for defining milestones and using them to terminate activities, to urge the delivery and check of results.

In the early project stages, the phase model looks attractive for managers, as it facilitates planning of time, costs and resources for software development in a seemingly exact way. In later stages, however, these planning data often turn out to be unreliable or irrelevant. Our investigations have shown, that software development practice substantially deviates from the "theory" of linear, sequential phase models:

- Only 25 % of the "waterfall-like" projects actually followed the original phase plan throughout the project. 41 % of the projects involved long phase overlaps, 13 % of the projects jumped forth and back between the phases, 13 % were simply characterized as "anarchic"[W-O 92].
- Another investigation of 29 projects following waterfall-like project models (which were already calibrated to 4 main development stages) revealed 39 explicit loops back to earlier stages, in 11 cases spanning two or more stages [BHS 95].
- There is almost no correlation between formal planning and project success. The assumption that projects which are more formally and rigidly planned would be more successful has proven a myth. On the other hand, the most successful projects have been those which were managed in the most flexible and adaptive way [H-W 94].

As a first conclusion, we can notice that working after linear or sequential phase models often does not correspond to the practical needs, sometimes it is just fiction. The practical needs are often much better met by *evolutionary software development* as originally advocated by M.M. Lehman. This kind of development is primarily dedicated to *socially embedded systems* which are closely intertwined with the organisation, the work procedures and personal requirements of people directly working with or affected by the software. Typical for such systems is the continuous feedback from organizations and users which results in a long series of requirements for changes and enhancements and thus in continued cycles of adaptation and further development [Leh 80].

Our investigations have shown that the *evolutionary aspects* of long-term software development are often acknowledged by the developers (and reflected in particular provisions for program enhancements and changes) but that they are widely neglected in formal project planning and management [HBS 92, BHS 95].

Moreover, *object orientation* (*OO*) as a programming paradigm is now more than 15 years old (or even 25 years, if Simula is considered to be object oriented), it has been successfully extended to software design and analysis and it is now becoming the preferred software development paradigm for many industrial projects. This affects more than just single process steps with local technical consequences, it influences the whole development process. Traditional, waterfall-like life cycle models cannot cope with these new challenges.

Thus we need a new type of software life cycle model which reflects the ideas of object orientation and evolutionary development not only at the bottom level of activity descriptions and guidelines but throughout the whole process. The rest of this paper will show the way from the evening of traditional waterfall models through the white OO nights *(WOON)* of so-called *OO system life cycles* to the morning dawn of *EOS*, a new software process model for *evolutionary, object-oriented software development.*

2 Object-oriented software life cycle models

During the last decade, a wide variety of OO software development methods, techniques, languages and tools have been published. It is beyond the scope of this article to give a technical overview on these; instead, we will select a few prominent examples and concen trate on the life cycle aspects of the methods. The following paragraphs summarize a forthcoming study on the life cycle concepts of known OO methodologies [Hes 96b]. For a systematizing introduction to this field, we refer to [H-E 90].

OO Analysis (Shlaer & Mellor)

S. Shlaer and S. Mellor have presented their methodology for *Object Oriented Systems Analysis (OOSA)* in two books [SM 89], [SM 91]. They emphasize the role of the Information Model in system development (cf. Ch. 9 of [SM 89]). Their general software development process model follows a traditional phase structure approach and comprises four phases:

- Problem Analysis (including the formation of the information model),
- External Specification,
- Systems Design,
- Implementation & Integration.

With their separation of the analysis model into the three submodels *information model, state model, process model* they reflect the traditional point of view of complementary, but self-contained submodels. Particularly the process model (represented by data flow diagrams) maintains the idea of top-down design through functional decomposition.

The central position of the information model (containing the identified objects) is acknowledged but it is used primarily as a basis for the following data structure design steps rather than as a focus for a coherent object-oriented modelling covering both structural and behavioural aspects. Data abstraction and inheritance are hardly discussed and the identified objects and classes do not play a recognizable role in the solution space, i.e. in the design and implementation phases.

Evolutionary aspects of software development are even more marginal and hardly reflected in the life cycle model. Development cycles are not explicitly excluded but it is not shown how they might fit into the overall sequential phase structure.

Object-Oriented Analysis and Design (Booch)

Grady Booch distinguishes two levels of the development process and has designed two intertwined kinds of processes: the *micro and macro process* [Boo 94]. The micro process is similar to Boehm's *spiral model* [Boe 88]. It serves as the framework for an iterative and incremental approach to development. The macro process is more closely related to the traditional waterfall life cycle. It serves as the controlling framework for the micro process. Both processes have to be reconciled in order to obtain a fully rational development process. Following Booch, this is a necessary prerequisite for achieving greater process maturity, i.e. the level "defined" according to the Software Capability Maturity (SCM) Model [Hum 88].

The overall structure of the *macro development process* is given by phases resembling those of traditional waterfall models:

- Conceptualization,
- Analysis,
- Design,
- Evolution,
- Maintenance.

Declaring *conceptualization* a separate phase, Booch emphasizes the importance of establishing core requirements for the software and developing a vision for some application. This includes experimenting with prototypes which are meant to be thrown away. Another specialty is Booch's naming *evolution* for the usual implementation phase. The evolutionary aspects are also stressed by the description of the purpose of that phase: „To grow and change the implementation through successive refinement, ultimately leading to the production system. To try out (through successive attempts) to satisfy a number of competing constraints and achieve an optimal compromise.“

The macro process ends with *maintenance* - as is usual for many traditional models. But Booch equates maintenance with „postdelivery evolution“ and explicitly mentions repetitions of the macro process after major product releases. Its basic idea is that of incremental development [Boo 94, p. 249].

The macro process represents the activities of the entire development team on the scale of weeks to months. It includes configuration management, quality assurance, code walk-throughs, and documentation. It focuses on risk and architectural vision. It serves as controlling framework for the micro process.

The *micro development process* consists of the following four steps:

- Identify classes and objects,
- Identify class and object semantics,
- Identify class and object relationships,
- Implement classes and objects.

All micro development processes are embedded in the macro development process. That means: macro process activities such as analysis, design and implementation follow a similar scheme of subactivities which are explained in the micro process.

The idea of intertwined macro and micro processes supports the objective of software evolution and lays a good basis for OO development. The two-level model illuminates the fact that various kinds of classes and objects are identified and implemented during the macro phases of analysis, design and evolution. But the two processes are not completely orthogonal to each other: analysis, design and implementation activities occur in both processes (even if „intentionally blurred“ at the micro level [Boo 94, p. 235]) and it is, for example, not clear what „evolution“ means in contrast to „maintenance“ and how long-term evolution is to be treated.

As a conclusion we can state that Booch presents a *hybrid approach* which combines traditional life cycle concepts with new process structures as required for object-oriented development processes.

Object-Oriented Modelling and Design (Rumbaugh et al.)

In their *Object-Oriented Modelling and Design technique* (short: *OMT*) J. Rumbaugh et al. distinguish 6 phases of software development [Rum 91]:

- Initial formulation of the problem
- Analysis
- Design (later decomposed into *system design* and *object design*)
- Implementation
- Testing
- Operational phase (incl. maintenace and enhancement)

The general structure of the OMT life cycle is a traditional, waterfall-like one. Object orientation comes in on the second (lower) abstraction level, where phases are decomposed into steps and activities. There we observe an emphasis on data structure before function: The object concept is considered the leading, single, unifying software concept throughout the process. All other concepts (functions, relationships, events, ..) are organized around objects.

The analysis model reflects this point of view. It consists of three submodels which cover the three essential aspects of the system:

- Objects and their relationships (object model, represented by object diagrams)
- Dynamic flow of control (dynamic model, based on state charts)
- Functional transformation of data subject to constraints (functional model, depicted by DFD's)

Life cycle phases are no longer considered as important and distinct as in traditional function-oriented models. The object structure is defined early in the project and maintained, enhanced and refined throughout the development cycle. This is to guarantee a seamless development process. Discontinuities (for example involving a change of notation) are avoided.

The authors claim the whole development process to be iterative rather than sequential. Development steps can easily be repeated at progressively finer levels of detail. Encapsulation of data structures into objects (with public interfaces that hide their private internal implementation) helps to localize the effects of changes. Inheritance supports the condensation of functionality into superclasses and facilitates evolutionary development starting from already defined (super-) classes.

Whereas the method is fully object-oriented and supports the ideas of software evolution, this has not led to a new type of life cycle model. Instead, the authors have chosen a hybrid approach building the steps of OO development into the well-known framework of an overall waterfall model.

Present efforts aim at unifying Booch's and Rumbaugh's methods to a new, joint method called the "unified" one so far. Less successful or old-fashioned features will not be included in the new method. For example, DFD's will not be maintained as they were not considered very helpful in practice (cf. [Rum 96]). As far as the life cycle model is concerned, no decision has been met so far about a future "unified process". At the moment, Rumbaugh does not give this issue highest priority: "... Many users tailor the process to their organizations anyway. ... It is not essential that everybody uses the same process, since the process is not a product that people exchange. It is important to standardize *artifacts* that people exchange, regardless of how they produce them. ... We will defer until a later time development of a new unified process - existing processes can be used with the unified notation."

Object-oriented Software Engineering (OOSE, I. Jacobson)

Jacobson distinguishes two kinds of life cycle models: The *System Life Cycle (SLC)* and the *project model (PM)* [Jac 93]. The former (also called *software engineering life cycle* or *process model*) consists of one pre-phase (Project selection and preparation) and four main phases:

- Requirements Modelling (RM)
- Object Modelling (OM)
- System construction (SC)
- System test (ST)

The managerial aspects are grasped by a *project model* containing phases and milestones. The phases are named as follows:

- Pre-study
- Feasability study
- Establishment
- Execution
- Conclusion

From the managerial point of view, the OOSE process model can be embedded into the project model by identifying conclusion of particular subprocesses with PM milestones. More exactly: The pre-study yields the requirements model (result of RM), and the execution phase yields the analysis model (result of OM), the design and implementation models (results of SC), and the test model (result of ST).

Systems development is seen as a process of change. Instead of new development, revision of existing systems and system changes are the primary focus. Therefore, the starting point of development is called *delta requirements specification*. Incremental development is particularly favoured: Starting with a couple of *use cases* having highest priority, further increments of reasonable size (about 5-20 use cases) are successively to be defined, implemented and integrated.

In OOSE, a modern, advanced object-oriented methodology is also reflected in the somewhat unconventional process model. Software development is rather seen as a long-term process of change than as an isolated project resulting in a particular product. The development processes are embedded in a project model which covers the managerial aspects. However, the relation of (and the necessity for) these two life cycles is not quite clear. For example, the cross-references between the system life cycle phases and the PM milestones are not well defined and it is not clear where the scope of one model ends and of the other starts.

The cluster model of B. Meyer

We conclude this section with a model which does not belong to a fully elaborated methodology and which is not embedded in a complete software life cycle. However, it gives a useful hint on how to organize the development of components or "clusters" of an object-oriented system in a (relatively) independent manner. Typically, a cluster is a set of logically or technically interrelated modules or classes. As is shown in fig. 1, the development of each of the clusters 1 to n starts independently from each other and can be timed according to the specific requirements of delivery or use by other components. In the sequel, we will show how this idea can be extended to any kind of building block (including single classes or modules) and be used to form a complete software process model.

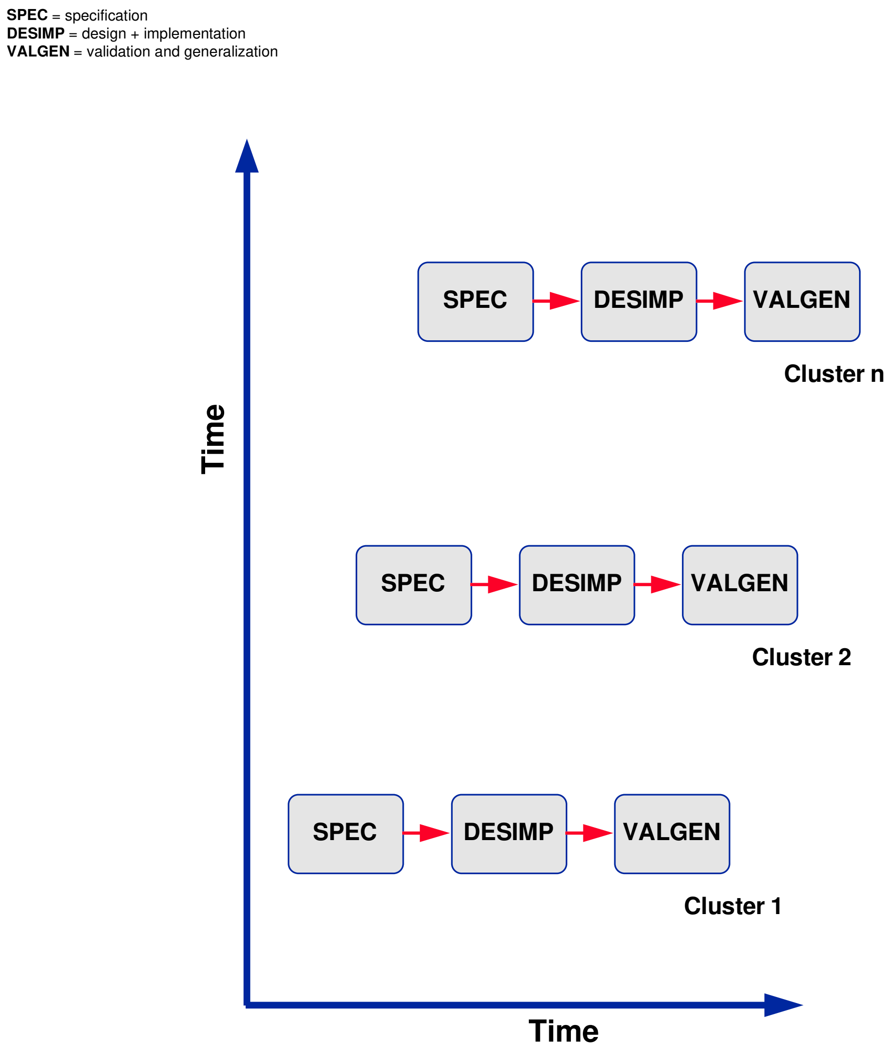

Fig. 1: Software life cycle for module/class design: The cluster model (cited from [H-E 90])

3 EOS: A model for evolutionary, object oriented software development

Three main arguments have inspired us to define a completely new software process model:

- Theory and practice of software development deviate too much from each other and need to be „re-harmonized“.

- Object-orientation is not well enough reflected in most of the existing models: mostly the focus is on „system life cycles“ rather than on „component life cycles“ (for any kind of OO-components including full systems).

- The evolutionary character of software development is not yet satisfactorily acknowledged. For example, bending a traditional waterfall structure into a cycle is not enough to build a truly evolutionary model.

The EOS model (for *Evolutionary Object oriented Software development*) has been designed to overcome the drawbacks of previous models. Its most important guiding principles can be summarized as follows (for details cf. [H-W 94] and [Hes 96a]):

- *Object orientation* as overall development paradigm:

Each piece of software development is based on uniform, evolving object structures. *Application objects* are identified during analysis of the application space and transformed to *objects* or *classes* of the solution space. They are complemented by other *solution-specific objects and classes* and may be grouped to *components*. Objects, classes, components and even full systems are addressed by the generic term *building block*. A building block is considered as a data encapsulation and is designed and implemented following the principles of data abstraction, inheritance and polymorphism (cf. [Mey 88]). Building blocks are treated as autonomous units and can thus easily be tested, changed, further developed or replaced.

- *Hierarchical system structure*

Statically, a system is composed of *components* and *classes* (cf. above). Components are disjoint from each other and form a hierarchical (tree) structure. This way, a three-level hierarchy of software building blocks is formed:

(S.) System
(X.) Components (or class complexes)
(C.) Classes

Classes (and objects) are interconnected by the static relationships of inheritance, aggregation and association. Dynamically, components, classes and objects are connected by message passsing and delegation mechanisms. Classes and objects may be linked together for testing, integration, prototyping or other purposes to form (typically non-disjoint)

(SS.) Subsystems.

- *Cyclic development*

An outstanding charcteristic of EOS is the treatment of each software building block (irrespective of its position in the system hierarchy) as carrier of its own life cycle. All life cycles (on all levels) have the same basic structure of *activities* (cf. fig. 2):

- (.A) Analysis
- (.D) Design
- (.I) Implementation
- (.O) Operational use (installation, operation and revision - incl. maintenance activities)

In particlar, s*oftware use and revision* are considered integral parts of each cycle. This replaces the often very dubious, sometimes even missing "maintenance phase" of previous models and makes it an explicit, clearly defined activity occuring on all levels.

Identifiers for building blocks are combined with activity identifiers in a fully orthogonal way (examples: "SA" for system analysis, "CI" for class implementation)

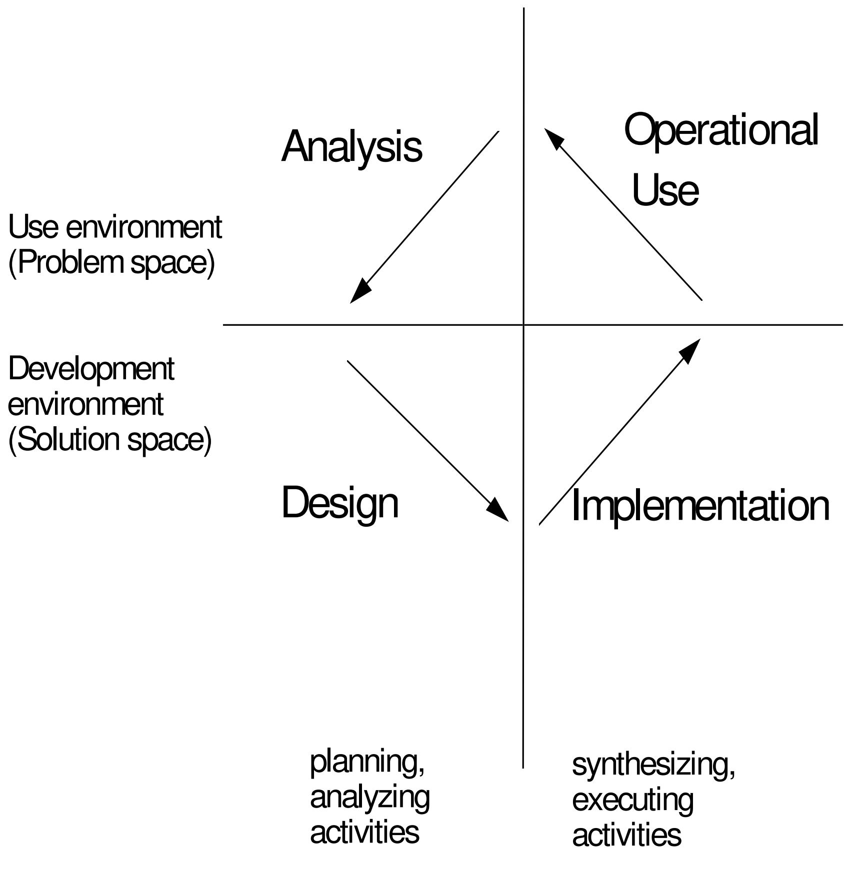

Fig. 2: The structure of an EOS development cycle

- Provisions for *further development* and *reuse*

One reason for emphasizing the autonomy of building blocks is to meet the leading goals of all object oriented techniques, i.e. the *reusability* of software. This is to be supported by a *building block library (BBL)* containing all building blocks from previous work and those defined so far within the project. Work with the BBL is particularly suggested during the analysis and the usage and revision activities of a cycle: first in order to search for reusable blocks and later to complement the library by new or derived blocks.

- *Systematic composition* of activities and developmemt cycles

Another outstanding feature of the EOS model is its *orthogonality:* All development cycles have a uniform structure and can be combined in a most flexible way. A rather traditional way to build a full *system life cycle* from the given cycles follows the well-established ideas of top-down analysis & design and bottom up implementation & integration: The initial steps of system analysis (SA) and design (SD) are followed by the corresponding steps on the component (XA, XD) and class level (CA, CD). The (bottom-up) way back leads from class implementation (CI) and use (CO) through the corresponding activities on the subsystem level (SSI, SSO) to those on the system level (SI, SO).

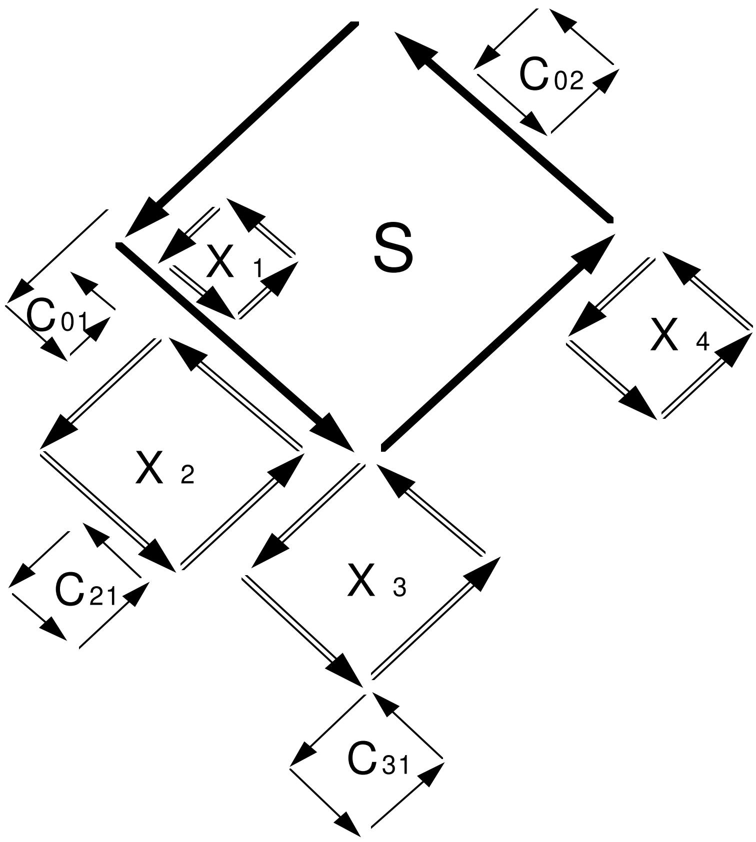

Fig. 3: Combining development cycles in an evolutionary way

Another combination of activities, which much better reflects the requirements of object-oriented development and system evolution, is depicted in figure 3. The varying starting points of arrows suggest that a new development cycle of a component or class can start at any arbitrary time. For project mangement this means: A certain developmemt cycle can be started at the moment *most appropriate* to obtain the intended result *when it is to be used*. Particular activities, starting and finishing dates of cycles are no longer bound to „phases“ but can be planned and performed exclusively triggered by the actual requirements of the project.

4 Project management based on the EOS model

For project managers, the EOS model has some important consequenes. First of all, project management does not become easier with EOS. It is not by chance that traditional waterfall models were so popular: They offered a seemingly simple and transparent structure of the development process. Milestones could easily be defined by identifying them with the termination of single phases. In practice, however, management of phase-structured projects often led to unexpected surprises, cost and time excesses or even to project failure. The phase structure turns out too rigid to scale activities or to react appropriately to new or changed requirements. On the other hand, it is too coarse to give sufficient support for detailed activity planning and control.

An important result of the IPAS investigations shows that formal planning and project success are *not correlated* to each other [H-W 94]. It is not a formal plan and the insistance of managers on planning data and stated requirements which makes a project successful but the ability of all people concerned (and particularly: of the managers) to react flexibly to unexpected situations accepting the challenges of new constellations and changed requirements.

Project management for *evolutionary system development* looks more complex in the very beginning, as it does not start with a complete, formal plan but requires that the preliminary plan be continually refined and adapted throughout the whole project in response to the current project status and the (remaining) requirements. This means: goals and requirements have to be checked, re-adjusted or re-defined continuously during the project life cycle. Software producer and user organizations cannot just negotiate a contract and proceed to execute it; they must regularly cooperate in order to keep their plans adjusted to the facts.

This kind of *dynamic project management* is not only supported but demanded by the EOS model:

- The phase structure is replaced by the much more flexible and adaptive (but also more complex) cycle & activity structure. Planning units correspond to the cycles and activities of the development schema: An overall system development (S-) cycle is refined by more detailed development cycles on the component level (X-cycles) and on the class level (C-cycles). The refined activity structure of the EOS model is a sound basis for both rough and detailed planning - according to the current situation at each moment during the project.

- Work distribution and responsibilities are based on products, i.e. on EOS building blocks rather than on types of activities. For example, a particular group of classes may be assigned to a single software engineer for development (including analysis, design, implementation and use & revision), a component or subsystem to a single team. For larger components and for the whole application system, long-term *service teams* may be defined. They are responsible for (further) development work as well as for operational support. A clear determination of responsibilities, as given by these rules, gives developers and users the individual freedom they need. Self-organization is not restricted by rigid top-down planning directives but motivated by enforcing personal responsibilities.

- In the EOS model, traditional milestones are replaced by more differentiated *revision points*. A revision point is used as checkpoint to determine and verify the status of a project at a given point of time. An example would be: "Components A, B, C finished, component D and classes E, F implemented, components G und H designed, component J analysed".

Revision points cannot all be defined in advance. They are roughly planned at project start, determined and reviewed whenever this is necessary, i.e. several times during project progress. This way, they can be adapted on demand to the current project situation and to the actual requirements. EOS revision points have some similarities to the „reference lines" of Ch. Floyd's STEPS model [FRS 89] - at least as far as their dynamics are concerned. In the EOS model, the dynamic creation of revision points is rooted in the generic cycle and activity structure. This facilitates their definition as the above example shows.

- Software development is viewed as a *continuous evolutionary process* which often lasts longer than the deadlines of particular projects. Development does not end with the installation of a "finished" software product. Ongoing development activities (as known as "maintenance") are both common and unavoidable demanding a knowledge transfer beyond the limits of a particular project. The EOS model prepares the ground for such transfer processes: from activity to activity, from cycle to cycle, from project to project.

5 Conclusion

The EOS model answers the challenges to project management raised by the ideas of object-oriented and evolutionary software development. It completes the way opened by the traditional life cycle models and continued by the spiral model and models for incremental development and prototyping. It goes beyond the approaches of OO development methodologies as it reflects their paradigmatic goals and requirements also in the software life cycle. This goal is reached by an orthogonal cycle and activity structure which builds upon software building blocks of any size rather than on the full system only.

Since its first publication, the EOS model has found much resonance, particularly in the software practice field. It is now being implemented and evaluated by small- and medium-scale software producers. Further work will focus on starting and supporting more pilot projects using the EOS method. Feedback from these practical projects will be used to improve techniques and instruments for project management, quality assurance and for object oriented analysis and modelling.

References:

[BHS 95] U. Bittner, W. Hesse, J. Schnath: Praxis der Software-Entwicklung, Methoden, Werkzeuge, Projektmanagement - eine Bestandsaufnahme; Oldenbourg 1995

[Boe 88] B.W. Boehm: A spiral model of software development and enhancement; *Computer,* May 1988, pp. 61-72

[Boo 94] G. Booch: Object-Oriented Analysis and Design with Applications; Second Edition, Benjamin/Cummings Publ. Comp. 1994

[Den 90] E. Denert: Software Engineering - Methodische Projektabwicklung; Springer 1990

[FRS 89] Ch. Floyd, F.-M. Reisin, G. Schmidt: STEPS to software development with users; in: C. Ghezzi, J. McDermid (eds.): ESEC '89, Second European Software Engineering Conference, LNCS 387, pp. 48-64; Springer 1989

[HBS 92] W. Hesse, U. Bittner, J. Schnath: Results from the IPAS Project: Influences of methods and tools, quality requirements and project management on the work situation of software developers; *in: P. Elzer, V. Haase (Eds.): Proc. Fourth IFAC/IFIP Workshop on Experience with the Management of Software Projects;* Annual Review in Automatic Programming, Vol. 16, Part II, Pergamon Press 1992

[H-E 90] B. Henderson-Sellers, J.M. Edwards: Objects oriented software systems life cycle; *CACM* Vol. 33, No. 9 (1990)

[H-W 94] W. Hesse, F. Weltz: Projektmanagement für evolutionäre Software-Entwicklung; *Information Management* 3/94, pp. 20-33, (1994)

[Hes 96a] W. Hesse: Theory and practice of the software process - a field study and its implications for project management; submitted for publication (1996)

[Hes 96b] W. Hesse: Life cycle models of object-oriented software development methodologies (forthcoming, 1996)

[Hum 88] W. Humphrey: Characterizing the software development process: A maturity framework; *IEEE Software*, Vol 5.2 (1988)

[Jac 93] I. Jacobson: Object-Oriented Software Engineering - A Use Case Driven Approach; Revised Printing, Addison- Wesley 1993

[Leh 80] M.M. Lehman: Programs, life cycles, and laws of software evolution; *Proceedings of the IEEE,* Vol. 68, No. 9, pp. 1060-1076 (1980)

[Mey 88] B. Meyer: Object-oriented software construction; Prentice Hall 1988

[Rum 91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen: Object-Oriented Modelling and Design; Prentice Hall 1991

[Rum 96] J. Rumbaugh: To form a more perfect union: Unifying the OMT and Booch methods, JOOP Vol 8, No. 8 , pp. 14-18 (Jan. 1996)

[S-M 89] S. Shlaer, S.J. Mellor: Object-Oriented Systems Analysis: Modelling the World in Data; Yourdon Press 1989

[S-M 91] S. Shlaer, S.J. Mellor: Object Lifecycles. Modelling the World in States; Yourdon Press 1991

[Ste 93] W. Stein: Objektorientierte Analysemethoden - ein Vergleich; *Informatik-Spektrum* 16, S. 317-332 (1993)

[W-O 92] F. Weltz, R. Ortmann: Das Softwareprojekt - Projektmanagement in der Praxis; Campus-Verlag 1992