Improving the software process guided by the EOS model

Wolfgang Hesse

c/o FB Mathematik/Informatik, Philipps-University Marburg/Germany
email: hesse@informatik.uni-marburg.de

Abstract

This article reports on an ongoing project which aims at introducing object-oriented technology in a large German user organisation working within the financial business area. The main part of the project is concerned with the complete new definition of the software process. It is intended to replace an older version based on structured techniques. Currently two proposals are under discussion which both try to capture key concepts like " process" itself, "view", "activity" and "development cycle". The first proposal tries to build a process model along the (technical) models, diagrams and results defined by the Unified Modelling Language (UML). The alternative proposal is based on some key ideas of the EOS model. These include to structure the software process according to the views of the concerned people and to the "objects" of the software being developed, i.e. its subsystems, components and modules.

1 Introduction: Scope and aims of the SPM project

A large German user organisation operating in the banking area (called FTO in this article) is presently running a project which aims at replacing 20-years old *structured* software development technology by modern, *object oriented* techniques and tools. A key part of this project is concerned with the definition of a new *software process model (SPM)* which is meant to form in some way the backbone of the new methodology. In the sequel, we concentrate on this part of the project and call it the *SPM project* for short.

The SPM project has been started with the following goals and aims:

- The new SPM is intended to replace a waterfall-like phase model which had acted as a backbone for the old methodology based on structured techniques,

- it shall form a solid basis for any kind of object-oriented software development projects,

- it shall provide a guideline for developers working with the UML language [UML 97],

- it shall go beyond the idea of sequential, waterfall-like development phases,

- it shall be used by software engineers of all levels (newcomers, experienced professionals and experts in the field) as well as by project leaders and software managers to organise projects according to the new methods and tools.

The project is conducted by a FTO-owned service organisation in co-operation with some consultants. The author entered the project when the SPM had already passed two reviews. Discussions concentrated on the concepts of "process" itself, of "views", "activities" and "development cycles". Original propositions were reviewed and some were confronted with alternative proposals based on key ideas of the EOS model [BHS 95, Hes 96a, Hes 96b, Hes 97]: This model is intended

(1) to be "object oriented" itself - basing all activities on the "objects" of software development (its building blocks, as e.g. classes and components),

(2) to be evolutionary [Leh 80] - considering dynamic, cyclic processes at all stages of development,

(3) to be systematic - defining all cycles in an analogous way to form a recursive "fractal" structure, and

(4) to be manageable - providing means for managers to create, plan, control, synchronise and conclude projects, processes and activities on any required level of detail.

The following sections reflect the discussions by applying the scheme (1) *Proposition:* formulating the first proposal (2) *Objection:* explaining my point of view and (3) *Conclusion:* offering an alternative solution.

2 Processes and views

The term of *process* is considered fundamental for the whole SPM. The software process is to be structured forming sub-processes which are composed of activities, sub-activities etc..

Proposition 1: There are four encompassing (sub-) processes forming the "columns" of the software development process. They are called

- Requirement,
- Modelling,
- Realisation,
- Introduction.

Objection 1: A process is a (logically) coherent, long-lasting chain of activities, happenings or events controlled by some actor, the "processor". It follows that processes are tightly linked to particular actors or groups of actors in the software development field. Sometimes a process may "sleep", i.e. no activities or events happen, but normally it may be resumed then, i.e. activities or events continue to happen. Further, we will try to view processes in an object-oriented way. This means we are looking on the object (or collection of objects) the process is concerned with. For example, a house construction process has a "house" as its object, a medical process has a person (or parts of his or her body) as its object etc.

How do the four terms listed above relate to that definition? In my opinion, they reflect a deep misunderstanding of the concept of "process" and its role in software development. "Requirement" is neither a process nor any kind of activity at all - it is some starting point or proposition typically triggered from outside and invoking some (internal) process or activity. The other three terms stand for activities or better: kinds of activities which roughly correspond to certain *stages* of a project or a development process. None of them behaves in the above mentioned manner. It is true that "modelling" or "realisation" and even "introduction" occur many times during a project but these occurrences are instances of *different* (sub-) processes concerned with *different objects* - be it a class C, a component or cluster or subsystem X or be it a complete application system S.

Conclusion 1: Activities like "modelling" and "realisation" are important parts of the software process but they are not coherent (sub-) processes themselves. We call them *activity types* the instances of which are concerned with the various objects of development.

On the other hand, in order to find the proper (sub-) processes of the software process we have to look at the *actors* in the software development field who bring in their particular *views* of the process.

Proposition 2: In order to get a complete insight of the various aspects of the software process, different *views* are adopted. In the proposed SPM, the following five views are defined:

- Project management,
- Quality management,
- Methods,
- Reusability,
- Safety.

Objection 2: The objection does not refer to the concept of views in principle but to its - at least in my opinion - inconsistent use. *Views* should be associated with different groups or *kinds of actors*. While the first two examples conform to this interpretation, the other three do not. While project management and quality management are associated with kinds of actors specialised in the above mentioned sense this is not true for the rest of the above list.

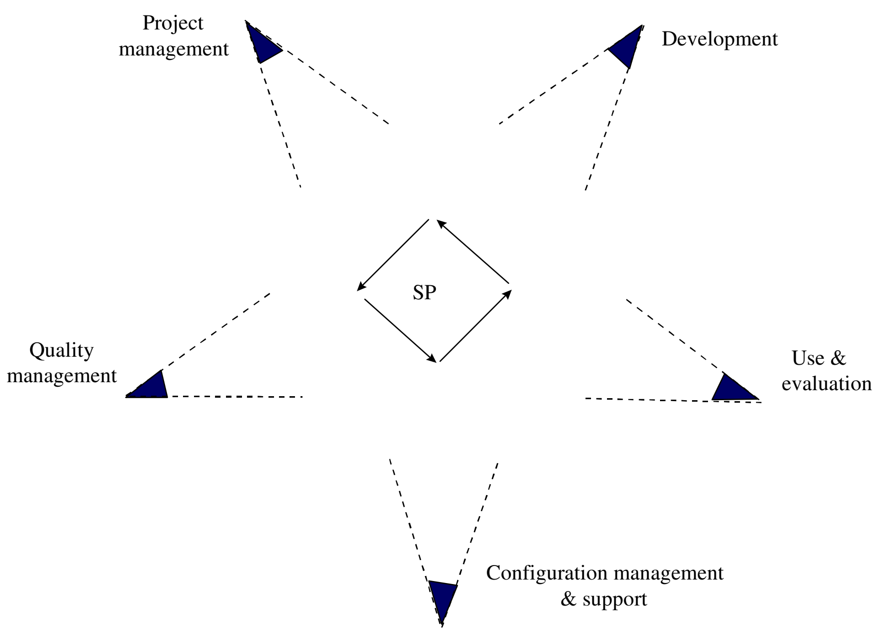

Fig. 1: Five views on the software process

"Methods" is not a *view* but *part of the support* offered or given to a software project. Other parts are tool preparation, documentation aid etc. A corresponding view might be named *"Project support"* ,*"Support"* or - more specifically - *"Development support"*. "Reusability" and "safety" are specific quality requirements which may influence many technical decisions but they are not primarily attributed to specific groups of actors. So we may consider them as sub-views of the general *development* view which (among other aspects) has to cover *all* quality requirements - not just the two selected in a rather unjustified way.

Conclusion 2: It is important to distinguish different views on the software process. Views are associated with corresponding groups of actors. This leads to the following list of (possible) views (cf. fig. 1):

- Development (the central and most important view, possible to be refined according to specialised subgroups of the development team),
- Project management,
- Quality management,
- Configuration management and support,
- Use and evaluation.

In particular, "use and evaluation" should not be forgotten. As we learned from Ch. Floyd's STEPS model, any kind of evolutionary, "learning" development process has to be paralleled by a use and evaluation process providing the necessary feedback for improving the main process in a dynamic, context-driven way [FRS 89].

With these views, we are now able to define the main processes of a software project. They correspond to the views in a unambiguous way and can thus be identified with them. This leads to the following

Corollary to conclusion 2: The software process can be described as a bunch of parallel, interacting (sub-) processes conducted by different groups of actors. Thus we can identify the five views listed in conclusion 2 with five parallel processes which make up the software process as a whole (cf. fig. 4 at the end of this paper).

3 Activity types: Distinguishing analysis and design

Although processes and activities have to do with each other they should be well distinguished. Processes are bound to a processor, may be long-lasting, are typically composed of different activities but are coherent with respect to the objects concerned. Hence we have proposed to define particular processes for developing a single class, building a component consisting of several classes, a subsystem or even a full application system. [Hes 96a, Hes 96b]. Activities, on the other hand, are the basic elements of processes, are limited in time, serve a certain purpose and may occur in several processes in equal or similar form.

Is it then possible to find a *typology of activities*, i.e. to identify certain types of activities which are common to several processes? Looking back to proposition 1, we find there at least a starting point for defining such *activity types*. This might lead to the following reformulation of proposition 1:

Proposition 3: The structure of software processes may be described in a generic way by identifying *activity types*. Main activity types occurring in most development processes are the above "columns" of software development (requirement, modelling, realisation, introduction).

Every process starts with a *requirement*. In particular, *modelling* encompasses both analysis and design activities. The argument to subsume these under one common activity type was that in OO development analysis and design are intertwined in time, mostly using the same model and being supported by the same tools. *Realisation* means to put the model(s) in an operational form and *introduction* means to do this at the user's side.

Objection 3: I appreciate the idea of four main activity types but we advocate for another classification. As I argued above, "requirement" is no activity or activity type since "requirements" come from outside and do not incorporate an *own* activity. But of course, *requirements analysis* is such an activity.

As far as the *modelling* activity is concerned, I strongly plead for maintaining the distinction of *analysis* and *design*. I do not believe that 30 years of Software Engineering experience which have confirmed the usefulness of such fundamental principles as information hiding and data abstraction [Par 72], which have shown the advantages of distinguishing the "what" and the "how", separating the specification from the construction for any piece of software to be built [Hes 84] can be obliterated by some superficial temporal coincidences. Even if we admit that a cyclic, integrated view of the software process brings them closer to each other, there remains one principal logical difference: *Analysis* is always oriented to the *problem space* whereas *design* is oriented to the *solution space* (cf. [L-M 81], cited from [H-E 90]).

It is true that the OO paradigm suggests a structural and terminological analogy between these two spaces - but their principle difference cannot be neglected. For didactic reasons, I would even consider fatal any kind of mixing up the "piece of real world" we observe during analysis with the artefact we are going to build during design . By the way - almost all authors of known OO methodologies maintain the analysis/design dichotomy - and I think for good reasons (cf. [Boo 94], [C-Y 90], [Jac 93], [Rum 91], [S-M 88])!

Conclusion 3: I prefer to replace the above list of activity types by the structure of a development cycle depicted in fig. 2 (cf. also [Hes 96a, Hes 96b]):

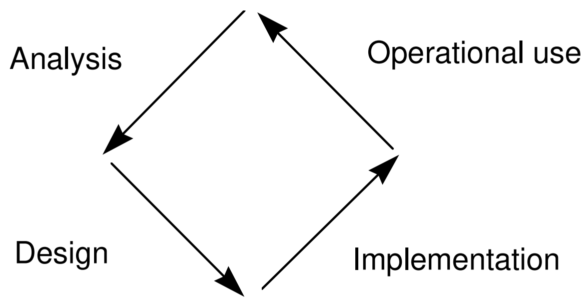

Fig. 2: Activity types in a development cycle

"Requirement" has been replaced by *"analysis"* - including any kind of analysis to be done when a new piece of software is to be developed: Requirements have to be analysed, but also constraints, quality plans, possibilities for reusing existing components, repositories etc. *"Modelling"* has been replaced by *"Analysis"* and *"Design"* for the reasons given above. Further, I prefer *"implementation"* to *"realisation"* just for terminological polishing (since for me, it sounds more focused on the programming, testing and integrating activities). On the other hand, I suggest that the last activity type *"introduction"* is extended to *"Operational use"*. Beyond the technical and organisational introduction of a product or piece of product this implies its use, evaluation, enhancement and, possibly, further generalisation and

proliferation activities enabling and supporting future development cycles and any kind of re-development and re-use.

4 Activities and their structure

Besides the concepts of *process* and *view*, the proposed SPM extensively builds on the concept of *activity. Activities* are considered to be the essential components of processes. They are normally assigned to particular responsible actors (persons), may be subject to project planning and quality assuring actions and may be decomposed into sub-activities forming a (tree-like) activity hierarchy.

Proposition 4: The complex structure of software systems is reflected by a deep activity hierarchy. This leads to many activity levels (up to 7 or 8) and to an encompassing, broadly ramified compendium of activity descriptions (up to 250).

Objection 4: Using the powerful concepts of recursion and orthogonality, we can save a lot of description work and produce a much simpler, more transparent description of activities - without loosing any power or detail. For reasons of clearness and transparency, activity hierarchies should not contain more than, say, three levels and the total number of activity descriptions should not exceed, say, 100.

Conclusion 4: Processes and activities can be defined in an orthogonal way profiting from the recursion principle. This is used to produce a refined view on the development process: Starting as a single stream during the early stages of system analysis, it is split into several concurrent processes according to the evolving structure of the software system being designed (cf. fig. 3 and 4). This means, whenever new components, sub-components or classes are defined, a corresponding development cycle is created and enacted. Each piece of software to be developed has its own (sub-) process, all processes follow one general scheme given by the above sequence of main activities. Of course, activities have to be specialised according to the kind of object they are belonging to: system analysis is by an order of magnitude different from analysis done for a new class to be developed or reused. But such specialisations can be given as subsections or appendices - without inflating and obscuring the clear overall structure.

5 Development cycles

At least since the discussions on prototyping and on Boehm's spiral model we know that software development processes are rather cyclic than linear as suggested by the waterfall models. In object-oriented projects, this tendency is even enforced: Re-development and re-use require a cyclic approach and are hardly compatible to a waterfall-like phase model. Most authors of modern OO methodologies acknowledge cyclic development on a macro level (concerning full systems or applications) while only a few extend the concept of cycles to a micro level - as, for example, G. Booch has done in his OOAD process model [Boo 94]. Thus, many of these methodologies are still rather "waterfall-like" than evolutionary - at least as far as their process models are concerned (for a detailed discussion cf. [Hes 96b, Hes 97]).

In the first SPM proposal, a development cycle is defined as a sequence of "kernel activities" which form together the "steps" of a "staircase" leading from one "floor" of a software engineering "building" to another. This staircase is to be trespassed as often as necessary.

Proposition 5: A development cycle is described as a sequence of eight kernel activities called

- Collect requirements
- Build use case model
- Define user interface
- Define scenarios
- Build logical architecture
- Build persistence model
- Determine component architecture
- Determine distribution structure

Objection 5: This kernel activity list might be appropriate to capture the *modelling steps* but it is not covering the *software process as a whole*. It contains too much and not much enough at the same time. Too much - since it goes into detail already at the highest level of the activity hierarchy, since it does not profit from recursive and orthogonal concepts, since it does not conform to the activity types identified in section 3. It contains not much enough, since it covers only one part of the system development cycle - its analysis and design steps - where the later stages (implementation, installation, use and revision) are missing. Furthermore, the idea of cycles is restricted to the macro level while cycles on the micro level (for example, concerning components, single classes or subsystems) are not addressed.

Conclusion 5: A generic, multi-cyclic process model defines processes on all levels of development in an analogous way. In the EOS model this has been done taking the four activity types of conclusion 3 as a general scheme [H-W 94, Hes 96a]. The 8 "kernel activities" listed above fit very well into such a scheme: the first two activities represent some steps of analysis on the system level *("system analysis")* while the other six detail the design activities on that level *("system design")*. Corresponding explanations have to be added for *system implementation* and *system operational use* to complete the system development cycle.

This approach proves particularly useful if it is extended to the other (the component and class) levels. This results in a complete re-classification of the activities into a general scheme of 12 main activities according to the 4 activity types on 3 levels (cf. fig. 3). Among other advantages, this approach stresses the particular role of components and classes in OO development and encourages an "object-oriented" view on the activities.

Activity type/ Development cycle	*Analysis (.A)*	*Design (.D)*	*Implementation (.I)*	*Operational Use (.O)*
System (S.)	SA	SD	SI	SO
Component (X..)	XA	XD	XI	XO
Class (C.)	CA	CD	CI	CO

Fig. 3: Development cycles and activity types

6 Management processes and activities

A process model for OO software development has to offer particular support for project management. A new methodology and new tools are hardly effective when they are not joined with corresponding management strategies. The SPM proposal acknowledges the role of project management by defining a particular "view" and some basic activities for it:

Proposition 6: The kernel (development) activities listed above are paralleled by three management activities called

- Start project,
- Control project,
- Conclude project.

Objection 6: This is an obvious approach, well-approved in the world of waterfall models but it is not sufficient in an object-oriented world. It does not support what we call *dynamic management* [H-W 94, Hes 96a], i.e. the ability to make and adjust plans on many levels, according to the current events and requirements raised by the (dynamically evolving) project.

Conclusion 6: Each *development process* has to be accompanied by management activities. Thus we have to modify the list of management activities to:

- Create and start process,
- Guide and control process
- Conclude process.

Since the development process branches into many sub-processes (according to the building blocks of the software architecture), management activities apply to manifold processes on various levels (cf. fig. 4). Such a structure offers the detail and flexibility required for dynamic management. Although this structure is more complicated than a waterfall model (and, in fact, makes the present challenges on project management visible), it is manageable and may be synchronised by means of so-called *revision points* (cf. [Hes 96a], [Hes 96b]).

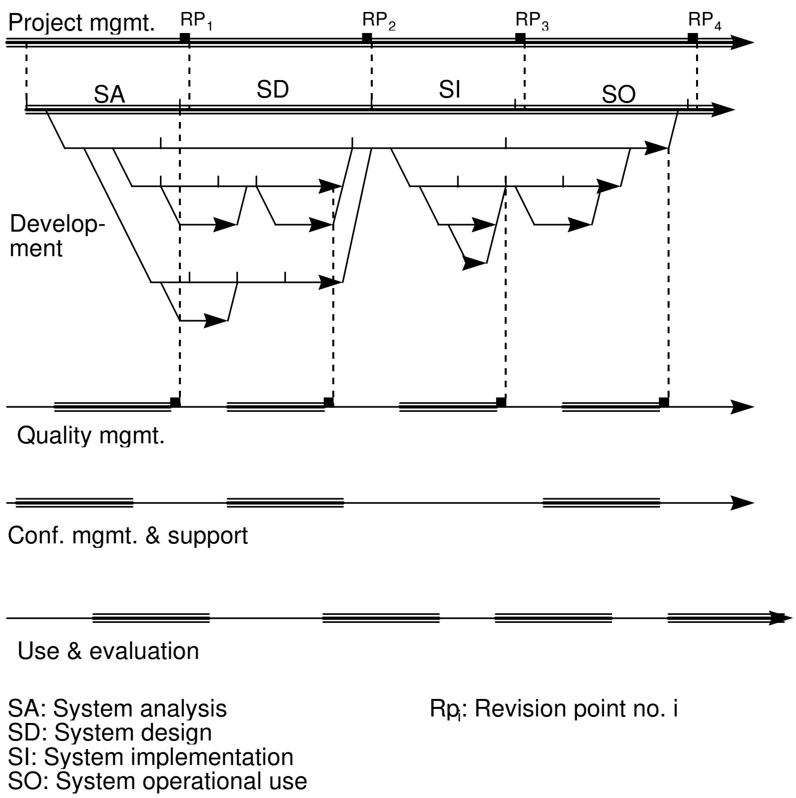

Fig. 4: The structure of the software process

7 Final remarks and outlook

The propositions and arguments given in this paper are now under discussion at FTO staff responsible for the SPM project. First reviews have shown that the original SPM approach (defining processes, views and activities, building an activity hierarchy) can be well complemented by basic ideas of EOS model such as "object orientation" of the development process, its recursive structure, common activity types and dynamic management. Current discussions are concerned with a consolidation of the SPM based on these ideas.

Further plans aim at extending the SPM by a project management component. Besides a deeper elaboration of the sketched management activity structure this should contain instruments for dynamic planning, simulation and control, the definition and evaluation of revision points and adequate means for cost estimation and calculation.

References:

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

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

[C-Y 90] P. Coad, E. Yourdon: *Object-oriented analysis*. 2nd edition, Yourdon Press 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

[H-E 90] B. Henderson-Sellers, J.M. Edwards: Objects oriented software systems life cycle; *Communications of the ACM* **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 84] W. Hesse: A systematics of Software Engineering: Structure, terminology and classification of techniques in: *P. Pepper (Ed.): Program Transformations and Programming Environments*. pp. 97-125, Springer 1984

[Hes 96a] W. Hesse: Theory and practice of the software process - a field study and its implications for project management; in: *C. Montangero (Ed.): Software Process Technology, 5th European Workshop, EWSPT 96*. Springer LNCS 1149, pp. 241-256 (1996)

[Hes 96b] W. Hesse: From WOON to EOS: New development methods require a new software processs model; *Proc. WOON Conference, St. Petersburg* (to appear) and: Technical Report no. 12/96, Fachbereich Mathematik, Philipps-Universität Marburg (1996)

[Hes 97] W. Hesse: Life cycle models of object-oriented software development methodologies. In: A. Zendler et al.: *Advanced concepts, life cycle models and tools for object-oriented software development.* Reihe Softwaretechnik 7, Tectum Verlag Marburg 1997

[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)

[L-M 81] H. Ledgard, M. Marcotty: *The programming language landscape.* Science Research Associates, Chicago 1981

[Par 72] D.L. Parnas: A technique for software module specification with examples. *Communications of the ACM,* **Vol. 15.5**, pp. 330-336 (1972)

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

[S-M 88] S. Shlaer, S.J. Mellor: *Object-Oriented Systems Analysis: Modelling the World in Data.* Yourdon Press 1989

[UML 97] G. Booch, J. Rumbaugh, I. Jacobson: *Unified Modeling Language (UML) for Object-Oriented Development (Documentation Set - Version 1.0).* Rational Software Corp., Santa Clara, CA 1997

Speaker' address:
Prof. Dr. Wolfgang Hesse, FB Mathematik/Informatik, Universität Marburg,
Hans Meerwein-Str., D-35032 Marburg

Tel.: +49-6421-281515, Fax: +49-6421-285419,
email: hesse@informatik.uni-marburg.de