Integrating the Developers' and the Management's Perspective of an Incremental Development Life Cycle

Andreas Zamperoni

Leiden University
Dept. of Computer Science
P.O. Box 9512, 2300 RA Leiden
The Netherlands
Tel.: ++31/71/27 7103
Fax.: ++31/71/27 6985
email: zamper@wi.leidenuniv.nl

Bart Gerritsen

TNO Institute of Applied Geoscience
Information Systems Oil&Gas
P.O. Box 6012, 2600 JA Delft
The Netherlands
Tel.: ++31/15/69 7196
Fax.: ++31/15/56 4800
email: gerritsen@igg.tno.nl

Abstract

Purely sequential life cycles can't cope with the developer's need for relatively unrestricted and "unordered" creativity when dealing with the development of innovative software systems. Nevertheless, project management usually favors sequential project phasing, because sequential processes are easier to plan, control and monitor.

In this paper, we present a life cycle plan that integrates an incremental and iterative development style, relying on evolutionary prototyping, with the management's perspective of clear and unambiguous project phasing. As consequence of applying this "hybrid" life cycle plan, the quality of innovative, complex software systems can be increased, while risks can be detected better and earlier.

We support validity of our approach by also reporting experiences we made when applying the life cycle plan to two large, multilateral software projects, and compare them to two systems developed following the traditional approach.

Keywords: incremental development, life cycle models, evolutionary prototyping, project management, risk minimization

Biographical: *Andreas Zamperoni* received a master degree in computer science from the Technical University of Braunschweig (Germany) in 1992. Since 1992, he works on a PhD on Software Engineering Methodologies at the Leiden University (The Netherlands), in the research group for Software Engineering and Information Systems. At the same time, he works at TNO Institute of Applied Geoscience (Delft, The Netherlands), where he investigates, evaluates, and enhances software development of geoscientific software systems. His research topics include object-oriented software engineering methodologies, software process modeling, and system architectures. He has also published in the field of quality enhancement for conceptual specifications. *Bart Gerritsen* received a master degree in mechanical engineering specializing in CAD from the Delft Technical University in 1986. He has been a senior system developer and project leader at Cap Gemini. Since 1991, he is project leader and software quality assurance officer at TNO Institute of Applied Geoscience (Delft, The Netherlands). He has been a project leader of the TNO participation at a 20+ man year international project funded by the EC. Currently, he is leading a project for the innovation of the entire information infrastructure of an oil company in South America. He is also working on a PhD on Subsurface Modeling Methodologies.

This article will appear at the Twelfth Annual Pacific Northwest Software Quality Conference, Portland, Oregon, October 17-19, 1994.

1 Introduction

When dealing with the development of innovative, complex software systems, such as e.g. neural networks for seismic simulation, traditional software engineering approaches are no longer suitable. *Prototyping* has emerged as one of the prime methods to facilitate, or even enable, the development of such innovative, experimental software products. Applying prototyping in software development consequently implies an adjustment of the software process model, too. Prototyping is strongly related to *incremental* or *evolutionary life cycle models* [Boe88], as opposed to the traditional sequential life cycle models [Boe76]. Nevertheless, project management often favors traditional sequential life cycles as the *waterfall model* and its variations because sequential processes are much easier to plan, control and monitor. Our paper describes how an incremental life cycle incorporating *evolutionary prototyping* as main concept can be integrated with the management's perspective of *sequential project phasing*, and reports the experiences we made applying such an integrated process plan during the development of two large geophysical software systems, compared to two similar systems developed earlier in the traditional manner.

At TNO[1], we succeeded to define and to establish a "hybrid" *software process plan* which bridges the gap between the developers' needs for relatively unrestricted and "unordered" creativity (*progress-by-experience*) and the management's need for organization, clearness and controllability of the development process (*progress-by-planning*). With this approach, we were able to increase the quality of our software development by capturing better the requirements of our customers, and at the same time limiting development time and technical risks of the final product.

Section 2 shows how an incremental and iterative life cycle as consequence of applying evolutionary prototyping looks like from the developer's perspective, and how it can be consistently integrated with the project management's sequential view of the software life cycle. In section 3, evolutionary prototyping as basis for a more sophisticated software engineering is motivated and described. Experiences with two recent, multilateral projects, carried out at TNO Institute of Applied Geoscience, are given and evaluated in regard of development time and risk management in section 4. Section 5 concludes the paper by summarizing assets and constraints of our approach.

2 An Evolutionary Life Cycle: Two Perspectives

The two basic components of a life cycle model are the *set of development phases*, characterized by the activities to be performed in them and their resulting deliverables, and the *execution order* in which the activities are combined to produce these deliverables [LRR93]. In practice, life cycle models are not only used to *describe* different forms of system development. The goal of using normative models is to have a validated *prescription* of the development process and the behavior of the people involved in it. Models used in this way to control a project are more adequately described by the term *life cycle plan* [BKKZ92].

Today, most of the large software development projects follow the line of a life cycle plan. The most common plan is the sequential *waterfall model* [Boe76]. In the waterfall model, the activities are fixed and arranged in a temporal sequence. The basic assumption behind this is that a system can be developed by a sequence of translations of well-defined specifications from an abstract problem description to a program that meets all the necessary quality criteria. The influence that these kind of sequential life cycle plans have exercised on software development in recent years can't be neglected: they match the project management's requirements of a clearly organized and

[1]At the TNO Institute for Applied Geoscience, Delft, The Netherlands, we develop experimental information systems and simulation programs for oil & gas exploration and production (cf. section 4).

verifiable development process. Nevertheless, there are important factors which indicate that this is not the (only) way how software is developed:

- By applying the waterfall model, software development is primarily seen as *technical problem* of stepwise transforming a problem description via a set of specifications into its program equivalent. The *communication problem* between users and developers is mostly totally neglected, as users are only involved at the beginning (requirements specification) and at the end (β-testing). The relatively late feedback can result in a serious mismatch between the users' expectations and wishes regarding the solution of their application-specific problem on the one hand, and the technical interpretation, i.e., the system, offered by the developers.
- Even if there is exchange of ideas, i.e., the specification documents are discussed with users, it has to be taken into account that *formal, non-executable system specifications* are unsuitable for communication between developers and users. As the software changes the working situation, the user needs "material" to visualize the new requirements, and on which to develop creativity and a feeling for the new possibilities. This is where *prototyping* comes into the game.
- Sequential life cycle plans do not meet the developing and the developer's reality: specification and implementation are distinct activities, but they are also tightly related in a *bidirectional* relationship. On the one hand, the implementation realizes the abstract specification of system functionality and structure. But on the other hand, the type of the potential implementation influences what aspects and parts of a system have to be specified [GS93]. The more is known about the projected implementation, the more precise and stable the specification can be, helping to avoid unrealistic or inconsistent requirements. But this kind of "change management" is often not implemented in sequential life cycle plans. Even more, in the case of research & development software (cf. section 4), the appropriate requirements are simply not sufficiently specifiable at the beginning of a project.
- Last, but not least, the gaining attention that is paid to *system maintenance* conceals the true problem concerned with *adapting a system to the application context*. Furthermore, the maintenance phase often falls outside the managed development cycle, resulting in unplanned, instant, bug-and-change-fixing activities.

Considering these factors, the traditional life cycle model has to be revised for projects expected to need more freedom in their development. Within this revision, the set of activities and resulting deliverables does not have to defer from sequential life cycles. But its execution order has to take the iterative and incremental nature of an evolutionary system development into account, without loosing the managerial control aspects out of sight.

At TNO, we established a life cycle plan that incorporates relative *unrestricted, but controlled evolution* of the software product, based on an *evolutionary life cycle model*. Note that the base document defining the life cycle plan is a software quality assurance plan [Ger93] that conforms entirely to the ANSI/IEEE Std. 730-1984[2]. Interpretation of this standard is in most cases consistent with the ANSI/IEEE Std. 983-1986[3] and other standards[4].

An overview of such an evolutionary development life cycle can be found in figure 1 to figure 3. In these figures, the iterative approach and its related sequential perspective can be depicted. The

[2]IEEE Guide for Software Quality Assurance Plans [IEE84]

[3]IEEE Guide for Software Quality Assurance Planning [IEE86]

[4]These standards are collected in [IEE89].

inner part of the cycle shows the development process from a technical, developer's perspective. The *network of nodes* indicates *tasks and activities* to be carried out, while *arrows* show their *order and interconnection*. In order to have the desired flexibility of an incremental approach, multiple transitions are possible at certain stages (cf. figure 2).

The *outer circle of deliverables* lists the desired *order of delivery* from the management perspective. Management, software quality assurance, etc. can be seen as taking along this temporal frame. For clarity, not all individual deliverables are shown in this picture. Configuration plan, verification & validation plan, user documentation, among others, are not depicted here, but mentioned in the following description of the life cycle.

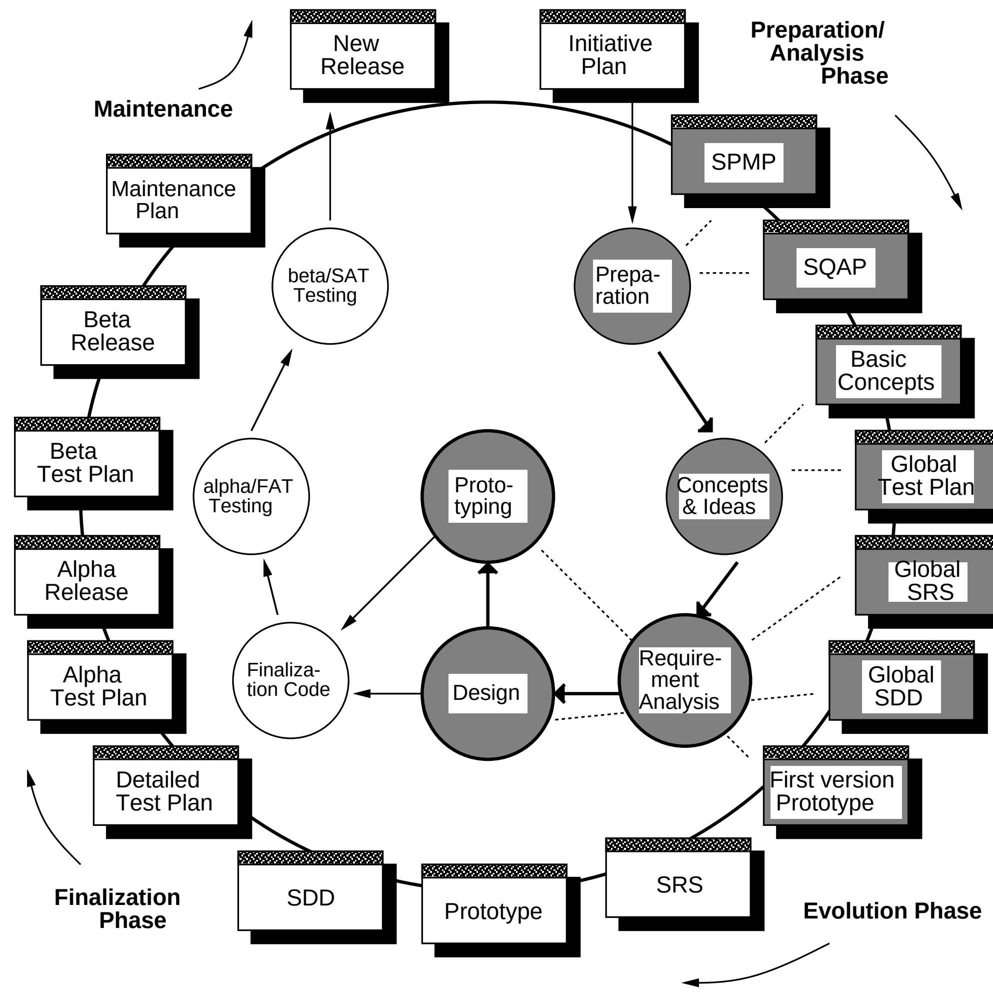

Figure 1: The evolutionary software life cycle, part 1: Preparation/Analysis Phase

The main phases reflect the subdivision of the life cycle into iterating and non-iterating parts: *preparation/analysis*, *evolution*, and *finalization* (plus operation/maintenance).

The developer's perspective

Triggered by the Initiative Plan, during *Preparation*, activities are carried out to produce all deliverables required at the project start. Principle deliverables are the Software Project Management Plan (SPMP) and the Software Quality Assurance Plan (SQAP). Also in this phase, the contractual and financial arrangements are settled.

In *Concepts & Ideas*, the Basic Concepts, considered fundamental for the application are gathered, worked out, and documented. Once established, they are not to be changed anymore, as they represent the core ideas of the project. In contrast to the basic concepts, design and implementation concepts may change during the project.

The *Requirement Analysis* activity concentrates on a description of how the resulting software has to look like, and will eventually result in a global Software Requirements Specification (global SRS), together with the result of the first *Design* activity, a global Software Design Description (global SDD). It is crucial that the requirements are *sufficiently* worked out (but do not have to be complete) and concise to start *Prototyping*. The construction of a first prototype concludes the first, non-iterating part of the development process. All these activities (and deliverables) are shown in figure 1 (the gray symbols).

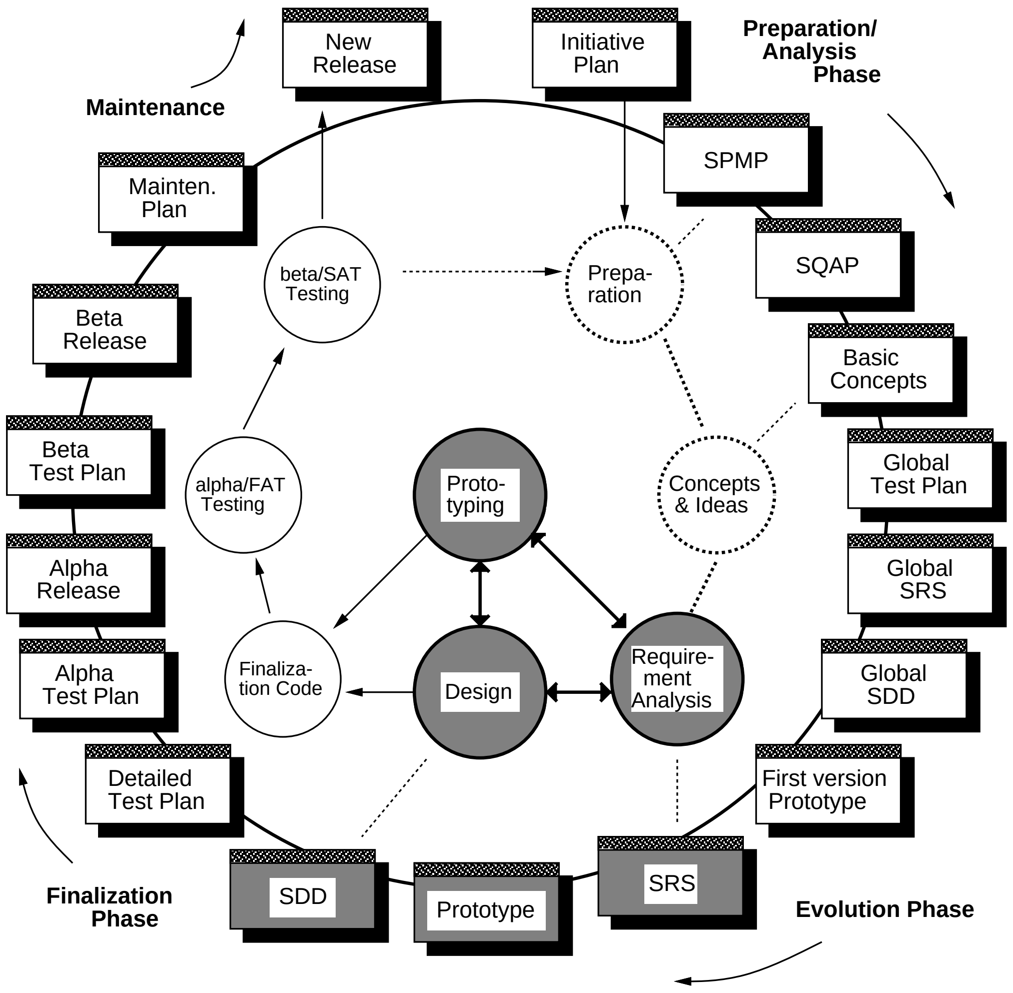

Figure 2: The evolutionary software life cycle, part 2: Evolution Phase

Subsequently, the global system layout, as laid out in the deliverables produced so far, is worked out in full detail. Focus is on those concepts and technical issues that are expected to be most critical for the future system. Attention may easily shift between *Requirements Analysis*, *Design* and *Prototyping*, as can be seen in figure 2, indicated by the gray process symbols. By this iterative, unrestricted approach, individual parts of the system can further be worked out in a number of cycles of *requirements-design-prototyping*. By using evolving versions of the same prototype as means of communication with the users, the discussion concerning the relative merits of alternative (design and implementation) concepts and solutions can be narrowed down. The special role of the prototype is further discussed in section 3.

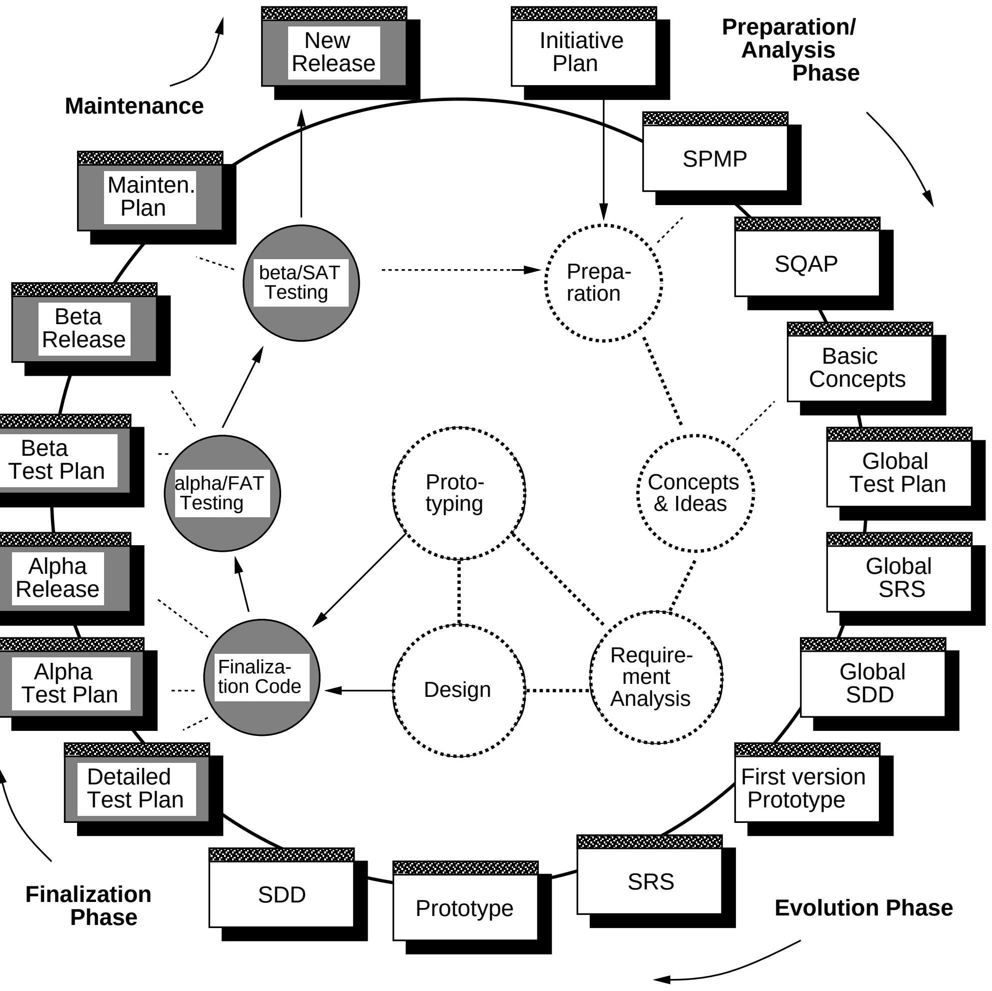

Figure 3: The evolutionary software life cycle, part 3: Finalization/Testing Phase

At the end of this evolution, the code of the incrementally developed prototype is completed and brought into a state of *finalization*, meeting the software quality standards defined in the SQAP.

Acceptance is ensured by α- (factory acceptance) and β- (site acceptance) *testing* (FAT, SAT), and preparations for carrying out maintenance are taken. The tasks and deliverables of this - again sequential - part of the life cycle are marked gray in figure 3.

Operation and maintenance are not shown in figure 3, but close the software life cycle, and can trigger further evolution of the project.

The project management perspective

Whereas the flexibility of the incremental approach of tasks and activities presented above may appear convenient from a technical perspective, the evolutionary approach is generally to fuzzy and too far beyond control from a managerial perspective. Having such a complex item as the prototype as reference "documentation" may also blur the exact taxation of the current project status. Therefore, for the management, tasks and activities as well as deliverables need to be put in a *clear and unambiguous time frame* in order to avoid misplanning.

This can be best accomplished by "unfolding" the network of activities in figure 1 to 3, resulting in distinct *project phases* and a clear sequence of deliverables that each can be attributed to a project phase (cf. figure 1 to 3, the outer circle).

The *Preparation/Analysis Phase* emphasizes on a global work out of the concepts, and a global layout of system requirements and system architecture. In figure 1, the shaded deliverables indicate the deliverables completed during this phase. Again, this phase is not intended to be carried out iteratively.

As can be read from figure 1, deliverables include:

• a description of the basic concepts applied • a global SRS • a global SDD • a global Test Plan • a first version of the Prototype

These deliverables typically mirror the global layout of the system so far. In addition, deliverables serving as preparation for further development will be produced:

• an SPMP • a SQAP • Programmer's Guidelines • a Configuration Plan • a Verification & Validation Plan

The *Evolution Phase* hosts the cyclic part of the system development. During this phase, individual parts of the system are analyzed in detail, implemented by, and added to the prototype. SRS and SDD may *no longer be anchor points*, i.e., validated baseline documents, for supervising and controlling development. As consequence, the prototype incorporates the crucial role of representing the state-of-the-art with respect to conceptual solutions and the system architecture. The prototype is used to discuss and validate decisions taken. Checkpoints for the progress of the project are when a "cycle" is finished by evaluation of, and feedback about the last prototype version by the user. Implications for the project managment which arise from this new role of the prototype within the project are discussed at the end of section 4. In figure 2, the deliverables marked gray represent the deliverables of this project phase. At the end of this phase, the

• SRS • SDD

will have reached their final form. Moreover, the code of the

• Prototype

needs to be in such shape that *Finalization*, the next phase, can be started. To work with the prototypes, (short) User Instructions are written, but the final User Manual is not supposed to be completed before the finalization phase, as the prototype might change extensively.

During the *Finalization/Testing Phase*, all products and deliverables will be brought in the form prescribed in the SQAP, and performing as described in the SRS. Project management's focus is on the deliverance to and acceptance by the customer. Also in this project phase, the preparation for carrying out maintenance takes place. The gray deliverables in figure 3 indicate the products and documents completed in this project phase:

• User Manual • Detailed Test Plan • α-Test Plan • α-Release • β Test Plan • β-Release • Maintenance Plan • New Release

The delivery of the New Release usually marks the end of this - sequential - project phase, and of the software development. The *Operation/Maintenance Phase* is covered by the Maintenance Plan, and may evolve in such a manner that the development of a further release or the development of a related system is envisioned.

3 Evolutionary Prototyping

Prototyping offers a practicable solution to the problem of validating and capturing better requirements, hence constructing a more stable system [Bud84]. While *throwaway prototyping* can deliver useful insights about certain limited details of a system, usually not much effort is taken to make those throwaway prototypes qualitatively persistent in regard of software qualities as e.g., efficiency, completeness, etc. [GJM91]. But nowadays, in major projects, the investment in prototyping is often too expensive for the organization to afford any longer to throw the prototypes away. Furthermore, in experimental software development, prototyping has to go beyond addressing single, isolated details of the projected system, but has become an essential means to stimulate imagination and creativity of the non computer-scientific target users. Assisted by new technologies, such as object-orientation, prototyping has matured from offering very limited working models which helped to highlight or validate certain limited aspects of a system (e.g., the user interface) to *evolving, full- scale, and persistent repositories* of acquired knowledge, development solutions and decisions [BKKZ92]. Sophisticated prototyping tools and reuse of (object-oriented) system specifications and components support a rapid construction of systems.

The full set of requirements and possible solutions are often not known or can't be overseen before implementation begins. With a prototype that is constructed early and evolves during the whole development, software concepts can be worked out and implemented in subsequent cycles of realization, testing, and feedback from (early) users to the developers. Concentrating on core functionality and on the new, experimental parts to explore, the current prototype represents at any time the ideas and visions with respect to conceptual solutions and the system architecture.

This new role for prototyping puts additional requirements and constraints on the prototype and on the activity of "prototyping", and hence makes it a principle target for *configuration management* and *software quality assurance*. Important prerequisites of evolutionary prototyping are:

1. *Sophisticated software engineering methodologies* to facilitate a comprehensible and thorough requirements specification (e.g., Use Cases [JCJO92]), and to assist the definition of a solid, but extensible system design (e.g., OMT [RBP$^+$91]).

2. *Object-orientation* - not only for the development methodology which captures the intuitive organization of the embedded knowledge, but also for the implementation. Object-orientation offers the concepts and the components crucial for the construction of the evolutionary prototype, and hence also for the final system. Modularization, reusability, and encapsulation support focusing on the relevant parts of the specification and the system at any time, and to filter out the unimportant parts. Clear implementation of the interfaces between objects is also well supported by object-orientation. Through reuse of existing components, an easier construction of full-scale prototypes is fostered.

3. The evolutionary prototype incorporates much more than the actual version of the implementation. To be able to trace back decisions and to reconstruct previous stages of implementation, *version control* and *careful documentation of the code* has to be supported to bridge the gap in time and understanding between successive development cycles.

4. Users have to be more tightly committed to the development of the system, as regular *evaluation and testing* of the prototype versions are necessary to steer system evolution into the right direction.

As benefit, costs and implications of decisions taken become more visible, risk decreases *earlier* and *faster*. Another important benefit of this form of prototyping is a smoother transition from the design phase to the implementation. The prototype is constructed, kept and evolved throughout the whole design (now: evolution) phase. Thus, the available code on which the implementation is based, is already rather complete and sound, i.e., fulfills required software standards.

4 Experiences With the Evolutionary Approach

In recent years, TNO Institute of Applied Geoscience has developed a number of software systems to manage exploration & production (E&P) data. Many of these applications are unique regarding their innovative, scientific nature, and the knowledge TNO has gathered in this field is reflected by the numerous cooperations with international petroleum and software companies, universities, and standardization organizations.

TNO's activities fall into three categories: research & development projects, consultancy & (pure) production projects, and knowledge transfer. While the latter two pose no, or only low risk of not succeeding, research & development projects have a higher risk of failure, due to their innovative nature of realizing in a software system the results of geoscientific research. These kind of projects lead to so-called *experimental software* (because the success of the resulting system can't be predicted), and are therefore often contracted to TNO. Our institute combines the application specific, i.e., geophysical, knowledge with the information technological skills to cope with the high risks of these projects. It was the particularity of the research & development projects that triggered the elaboration of a different life cycle plan.

This section will compare the development efforts for two "pure production" projects, realized with a traditional waterfall life cycle plan, with two experimental software projects, developed using the evolutionary approach introduced in the previous sections. As not only technical data about the products, but also confidential development information is presented, the real names of the projects are omitted and referred to using capital letters.

Project A and B: Information Systems for Reservoir Characterization

The projects presented in this subsection are both information systems that aim at managing information needed for the process of oil & gas reservoir characterization. Both systems run on workstations and offer graphical display functionality on top of a relational database. In both cases, information analysis was carried out with semantic data modeling techniques (cf. table 1).

The fundamental difference between the two systems lies in the specific application area. Unlike the conventional systems for well information, hydrocarbon production and geology (the category of *System A*), *System B* focusses on outcrop information and includes viewing functions in the form of e.g. maps, cross sections and scanned images. As such a system didn't exist in the past, there was no reference system to rely on, and therefore System B's technical risks were much higher than for System A.

Figure 4 gives an overview of the distribution of man-hours for the various development activities of the two projects.

Management activity hours are not explicitly stated, but attributed to the respective development phases. Although both projects are about the same size (duration, man-hours, platform), it is

Characteristics:	Project A:	Project B:
project type	pure production	research & development
total duration	18 months	18 months
total man-hours	3100	3600
team size	6	4
platform/ tools	workstation, Unix, OSF/Motif, Uniface, Oracle DBMS, Lotus 1-2-3	workstation, Unix, OSF/Motif, Uniface, Oracle DBMS, XV (image display)
languages	Uniface 4GL	Uniface 4GL
development tools	(ER tool), Uniface	X-NIAM, Uniface
product size	40 entities 249 forms 2.9 mb executable + 19.7 mb Uniface 4GL code	180 entities 197 forms 3.5 mb executable + 14.4 mb Uniface 4GL code

Table 1: Technical data for Project A and Project B.

difficult to compare the absolute numbers directly. Developers, who participated at both projects, subjectively categorized Project B as "2–3 times more complex" than Project A. This appraisal is supported by the complexity of the data model (cf. table 1, no. of entities). So instead of the absolute numbers, the amount of man-hours is given in percentage of the total man-hours of the respective project, which reveals a very interesting picture, too.

In Project A, with the sequential life cycle, a first version of the system was available only in the implementation phase, after more than 70% of the total project time, with the first complete version being ready only after more than 90% of the total project time. This means that feedback about adequacy and performance of the system could occur only at a very late project stage, while the risk of the project was still relatively high. At the same (relative) point of time, Project B was still in the finalization phase, but feedback by the users had already occurred (at least) *three times* (at 17.2%, 50.8%, and 77.8%), not taking into account the α-release finished shortly after (at 94.6%).

Note also, how the percentage of activities during the three evolution cycles of Project B shifts from mainly requirements analysis (`R`) to mainly prototyping (`P`). The small box on the right side of figure 4 gives the total percentage of requirements analysis, design and prototyping for Project B. Design activities in Project B have a considerably bigger share of the total project time than in Project A (15.8%, comp. to 8.1%). This is not only an indication for a more complex system, but is also due to the fact that evolutionary prototyping requires a better *thought-out system architecture* which does not only aim at an optimal final product, but also facilitates reconstruction and change of its subcomponents. Furthermore, Project B already incorporates the principle of reusability, i.e., the aim of constructing, components that would be easier to reuse in future projects - a characteristics that requires a good components design.

Interesting are also the numbers for the requirements analysis, which was shorter for Project B (27.5%, comp. to 35.5%). In Project B, requirements analysis was done by a project partner without computer-scientific specification and abstraction background. As consequence, the specifications were not so clear and complete, and had to be adapted and revised, by showing their implications to the users via the prototypes.

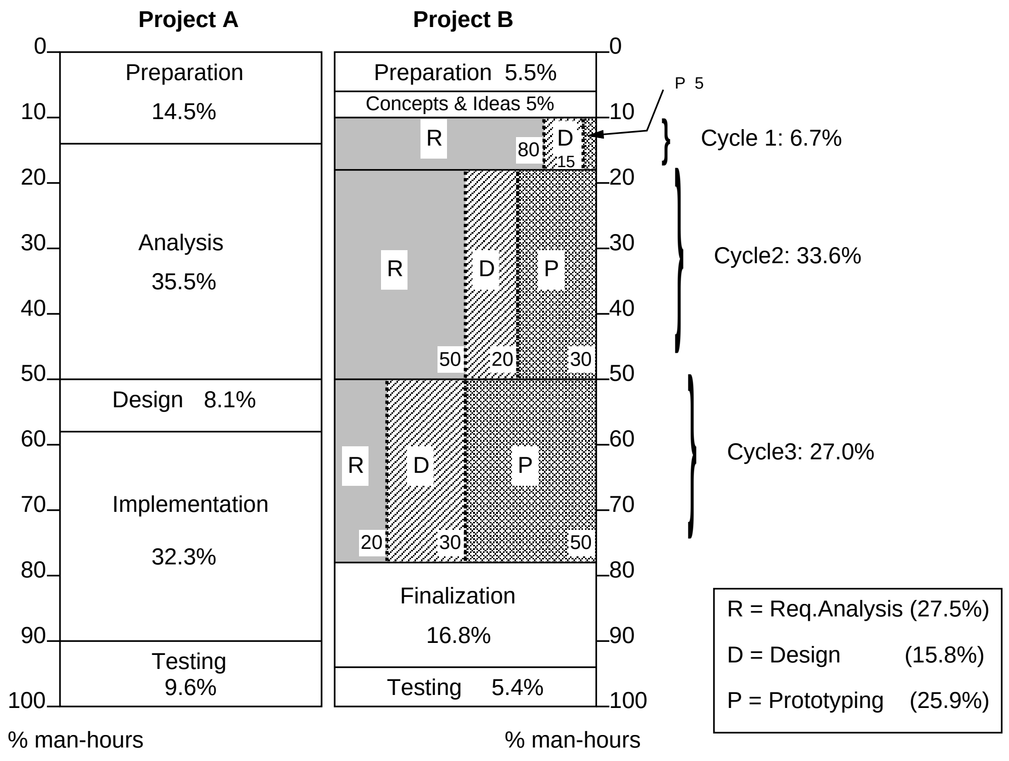

Figure 4: Overview of development activities for Project A and B

As expected, the finalization phase of Project B is much shorter than the implementation phase of Project A (16.8%, comp. to 32.3%). Finalization consisted mainly of "cleaning up the code". Nevertheless, the total amount of coding activities is higher for Project B, as the prototyping has to be considered as coding, too. This high percentage of coding activities (P + finalization: 40.7%, comp. to 32,3% for Project A), but even more the fact that the α-release of the system was preceded by (at least) three versions of a prototype, required a good version and code management tool. In the case of Project B, this task was sufficiently realized by *Uniface* itself, which was used also as development environment.

Project C and D: Reservoir Characterization by Seismic Data Interpretation

Both projects aim at developing a software system which enables the geoscientist to extract structural and reservoir characteristics from seismic response around an interpreted (artificial) seismic event. Goal is to transform the seismic data into an accurate depth image of the subsurface including the greatest possible detail on stratigraphic, structural features, and above all, rock and pore parameters. While both systems incorporate innovative parts and are comparable in terms of project size (cf. table 2), an innovative combination of stochastic modeling and artificial intelligence, i.e., artificial neural networks, is applied in *System D* to achieve its goal. The approach of using trained neural networks is new and made it necessary to prototype extensively with real data in proprietary case studies, in order to feedback the results of these studies to optimize the software product.

A situation similar to Project A and B holds here: while both projects had about the same duration and costs, and similar platforms, the complexity of Project D, indicated by the number

Characteristics:	Project C:	Project D:
project type	production	research & development
total duration	30 months	26 months
costs (overall)	US $ 1.25 M	US $ 1.30 M
total man-hours	7400	9600
team size	5	15
platform/ tools	workstation (UI) + Cray Unix, OSF/Motif, XFaceMaker SU (Seismic Unix)	workstation (UI) + Cray Unix, OSF/Motif, XFaceMaker, *System C* (reuse), SU, WingZ (spreadsheet), Aspirin/Migraine (PD neural network)
languages	C	C, C++, WingZ-Hyperscript
development tools	StP[5], XFaceMaker	StP, XFaceMaker, RCS/CVS, proprietary tools
product size: subsystems (subject of research)	 2 (1) 48 mb executable (116 kloc)	 5 (4) 38 mb executable (110 kloc) + 2.44 mb worksheets (WingZ) (35 kloc)

Table 2: Technical data for Project C and Project D.

of subsystems that were subject to research, classifies it as *experimental development*.

Figure 5 shows an overview of the development activities for both projects. The same technique as for the other two projects is applied to compare development efforts.

In *Project C*, with sequential life cycle, implementation was available for α-testing after 83.8% of the total project time. The functional tests at this relatively late stage revealed serious performance problems related to the access and processing of the large amount of seismic data. These problems explain the extensive α-test phase (13.5%), in which it was tried to locate the pitfalls that threatened the success of the whole project. After β-testing soon confirmed the shortcomings of the system, it was decided to totally redesign and re-implement System A (this is not indicated in figure 5). With the experiences of Project A in mind, management and developers chose for *extensive prototyping* of System B (44.3% of the total project hours), concentrating especially on the technical part, the performance of the "unknown" neural network. At a comparable point of time (at 82.3% of the total project hours), three prototypes had been tested and evaluated. Thus, finalization and α-testing, like for System B, took much less time than in the sequential reference projects (8.3% finalization, 3.1% α-testing).

The same *shift in activities* from requirements specification (`R`) to prototyping (`P`) during the evolution phase is observable here for Project D. The percentage of design activities is again higher than for System C, and the total amount of coding activities, i.e., prototyping + finalization, for System D matches the implementation activities of System C. The difference is, of course, that in System D these coding activities are more "distributed" and intertwined with the requirements analysis, design and feedback by the user. Initially, in Project D, the familiar SCCS-tool for configuration management & control was used. Later, it was switched to the use of *RCS/CVS*, as this tool offers a richer set of features and more appropriately allows a team to do concurrent

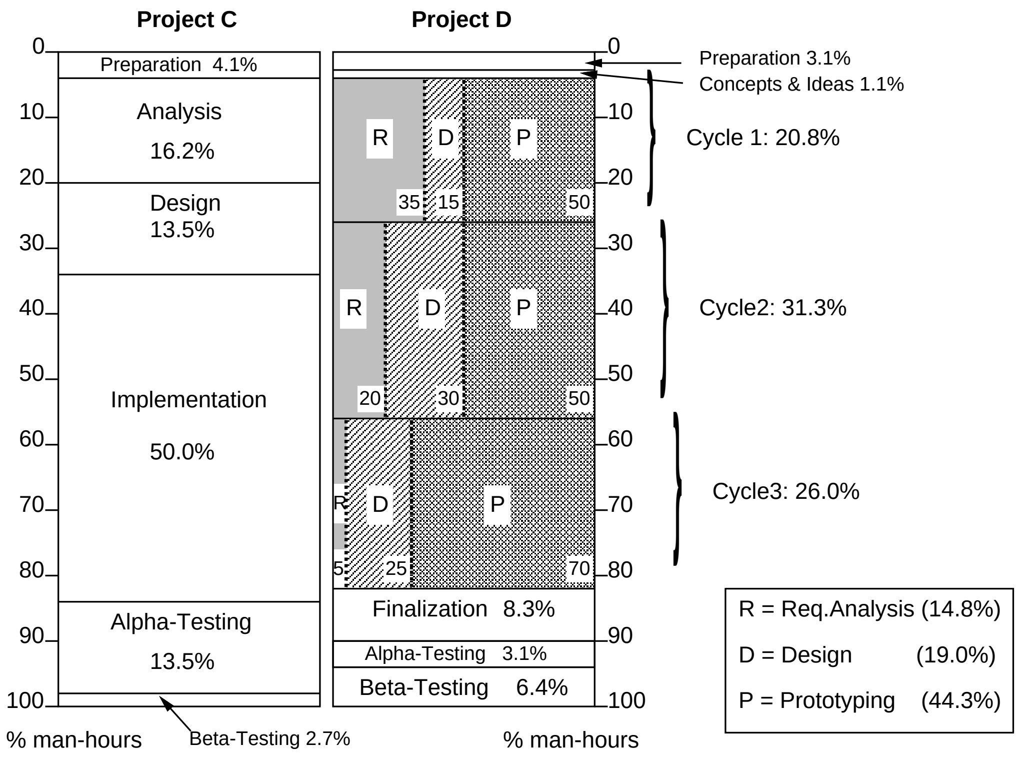

Figure 5: Overview of development activities for Project C and D

software development. Apart from this tool, a proprietary tool, implemented for a former TNO-project, was used for documentation, extraction of headers, annotating, etc..

5 Assets, Constraints and Conclusions

It truly is difficult to compare the absolute numbers of the projects to come to a conclusion like "we saved *this* amount of time and *that* amount of money with our evolutionary approach." After all, the new life cycle plan was established because we had to cope with new, experimental software systems, for which no reference systems exist.

Assets

What comes closest to an absolute estimation is a cautious statement that "we were able to develop more complex systems with the same order of magnitude of resources." Much more important though, is that the evolutionary life cycle plan helped us to decrease technical risks and the risk of not meeting the users' requirements and wishes earlier and faster.

Each version of the prototype and the feedback from testing and users translates directly in a decrease of risk, as sketched in figure 6, where the risk curves for traditional, sequentially phased projects and for the evolutionary development are compared. As consequence, this approach to system development can increase the confidence of users and developers in the quality of the future product.

Apart from the general, more advantageous risk curve for the evolutionary approach, two details concerning this curve are noticeable. In the beginning, risk decreases slower because analysis

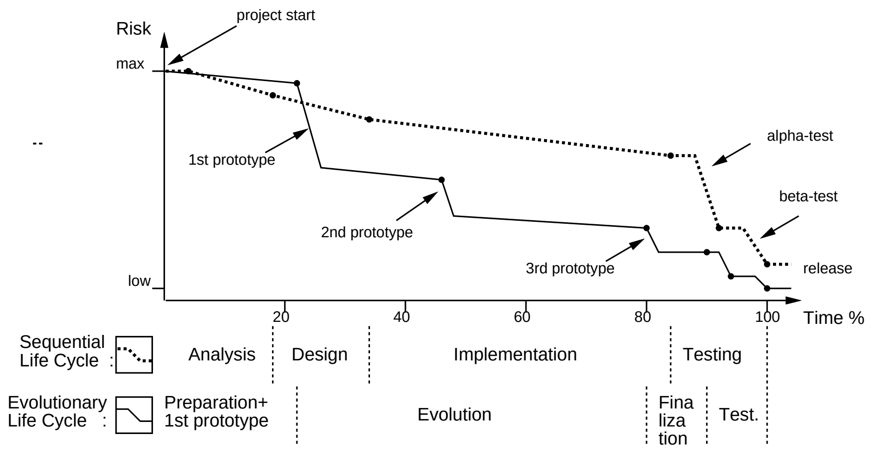

Figure 6: The risk curves for sequential and evolutionary development compared.

and design activities are not as detailed as for the sequential approach. But this changes with every prototype that is evaluated. At the end, the hazard of failure generally is lower for systems developed via the evolutionary approach, simply because, although spread over the whole life cycle, the total of feedback and evaluation of the current development efforts is higher than in traditional projects.

Constraints

Some considerations with respect to the presented evolutionary approach deserve particular attention, as they are vital for the success of the "hybrid" life cycle plan:

- Project management has to pay constant attention to measure precisely enough the *progress of development* during the cyclic evolution phase.
- Deliverables are not necessarily produced in the order presented in figure 1, and specification documents like the SRS and SDD may no longer be anchor points for controlling development prior to the start of prototyping. But concepts, overall requirements, and overall design have to be *sufficiently complete and concise* when coding starts, in order not to hamper the evolution of the system at later stages. To decide about this is again a crucial task for project management and (software) quality assurance.
- The prototype has an additional role as central repository of knowledge concerning concepts, design decisions and their implementation. As consequence, the code has to be *documented and annotated carefully*, and a sophisticated development environment has to *support the different versions and stages* the different parts of the prototype may be in.
- Due to early prototyping, a (global) *test plan* needs to be available in a relatively early stage. Users and non-computer scientists, as e.g., application domain experts have to be *involved in the activities of the evolution phase*. This might not always be so easy to achieve for usual software development sites. At TNO, which is a large organization[6], it is possible and supported that software constructors and domain experts (geophysicists) work together in one team.

[6]about 4500 employees in 15 institutes

Conclusions

The benefits of our approach to integrate the developer's and the management's perspectives of the software life cycle and of evolutionary prototyping have been confirmed by the outcome of the project described above. This holds for positive results as the sophisticated capturing of requirements and wishes of the users, as well as for the early detection of risks and shortcomings. Still, it is difficult to *measure precisely* the impact of our life cycle plan. The two reasons for this point to the areas of future work concerning our approach:

- *Metrics* have to be provided to describe more concretely the decrease of risk and increase of confidence, which up to know can only be estimated, as sketched in figure 6. The time diagrams (figure 4 and 5) give a good picture of the different stages of the project, but another significant indication would be comparing the number of *change proposals* and their impact on the projects.
- The experimental nature of the systems likely to be developed via the evolutionary approach makes it difficult to *find reference systems*. Uncapturable factors as the experience of the development team and the knowledge of the users, and the complexity of the research part always play an important, individualizing role that can only be excluded by developing *more projects* according to the evolutionary approach.

Furthermore, work is done to formalize and to integrate the approach presented here into a more general methodology which integrates also other aspects of software engineering, such as system architecture and specification techniques [Zam94].

References

[BKKZ92] Reinhard Budde, Karlheinz Kautz, Karin Kuhlenkamp, and Heinz Züllighoven, editors. *Prototyping: An Apporach to Evolutionary System Development.* Springer-Verlag, 1992.

[Boe76] B. W. Boehm. Software Engineering. *IEEE Transactions on Computers*, C25(12):1226–1241, 1976.

[Boe88] B. W. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 12(5):61–72, 1988.

[Bud84] R. Budde, editor. *Apporaches to Prototyping.* Springer-Verlag, 1984.

[Ger93] Bart Gerritsen. Software Quality Assurance Plan. Technical Report OS 93-52-C, TNO Institute for Applied Geoscience, 1993.

[GJM91] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering.* Prentice-Hall, Inc., 1991.

[GS93] David Garlan and Mary Shaw. Architectures for Software Systems. Tutorial Notes, 15th International Conference on Software Engineering, Baltimore, June 1993.

[IEE84] IEEE. *ANSI/IEEE Standard for Software QUality Assurance Plans*, 1984. ANSI/IEEE Std 730-1984.

[IEE86] IEEE. *ANSI/IEEE Guide for Software QUality Assurance Planning*, 1986. ANSI/IEEE Std 983-1986.

[IEE89] *IEEE: The Software Engineering Standards - Third Edition*, 1989.

[JCJO92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering.* Addison-Wesley, 1992.

[LRR93] Perdita Löhr-Richter and Georg Reichwein. Object-Oriented Life Cycle Models. Technical Report 93-05, Technical University of Braunschweig, 1993.

[RBP+91] James Rumbaugh, Michael Blaha, William Premberlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, Inc., 1991.

[Zam94] Andreas Zamperoni. Integration of the Different Elements of Object-Oriented Software Engineering into a Conceptual Framework: The 3D-model. Technical Report 94-18, Leiden University, Dept. of Comp. Science, 1994. (also available by *anonymous ftp* from `ftp.wi.leidenuniv.nl` in `/pub/cs-techreports` as `tr94-18.ps.gz`.

Acknowledgements:

We would like to thank the project members and leaders Paul de Groot, Simon Pen and Ipo Ritsema for providing us the useful statistics about the four projects mentioned. Their meticulous man-hour counting won't be forgotten.