# Evolutionary Software Development: An Experience Report on Technical and Strategic Requirements

Andreas Zamperoni Bart Gerritsen Bert Bril

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

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@iag.tno.nl

**Abstract**

Traditional software engineering approaches are no longer suitable when dealing with the development of innovative, complex software systems, such as e.g. applications of neural networks for geophysical subsurface modeling. At TNO, we succeeded to define and to establish a "hybrid" life cycle plan that integrates the developers' view of a creative, flexible and unrestricted development process on the basis of evolutionary prototyping with the management's needs for organization, controllability, and clearness of a software project.

We report the experiences we made when applying the evolutionary life cycle plan to multilateral software projects, and compare them with systems developed following the traditional approach.

In this paper, we focus on the most crucial technical and strategic requirements for (controlled) evolutionary software development. We discuss the issues of team structure, commitment of, and communication with users, frequent testing, integration & version control, component reusability, and management capabilities that become important considerations when replacing a traditional sequential by an evolutionary life cycle plan.

**Biographical:** *Andreas Zamperoni* received a master degree in computer science from the Technical University of Braunschweig (Germany). 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 geo-scientific software systems.

*Bart Gerritsen* received a master degree in mechanical engineering specializing in CAD from the Delft Technical University (The Netherlands). He has been, among other, 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 also been a project leader of the TNO participation at a 20+ man year international project funded by the EC.

*Bert Bril* received a master degree in geophysics from the Delft Technical University (The Netherlands). He has worked as field geophysicist, and been a senior system developer and at Cap Gemini and Jason Geosystems. Since 1992, he is senior system designer and technical project leader at TNO Institute of Applied Geoscience (Delft, The Netherlands).

# 1 Introduction

Most traditional software engineering approaches start from the ideal assumption that software development is a sequence of one-directional translations from an abstract problem description to a running program that meets all necessary quality criteria. The influence that these kind of **sequential - "waterfall" - life cycle models** [Boe76] have exercised on software development in recent years can not be neglected - they match the requirements of **project management** of a verifiable development process concerning time, budget, progress, and quality. Therefore, management usually seeks to keep control of the development process by imposing strict phasing to ensure clear organization of the production process.

On the other side, **developers** prefer to use their "unrestricted creativity" in their search for an "optimal" solution of the problem. They want to switch between the different development activities of analysis, design, and implementation, so that the system can evolve gradually and incrementally, following a **"creative chaos" life cycle model**.

This diversity leads to a **conflict** between the different groups of people committed to a project (cf. fig. 1), because though analysis, design, and implementation are distinct activities, they are also tightly related!

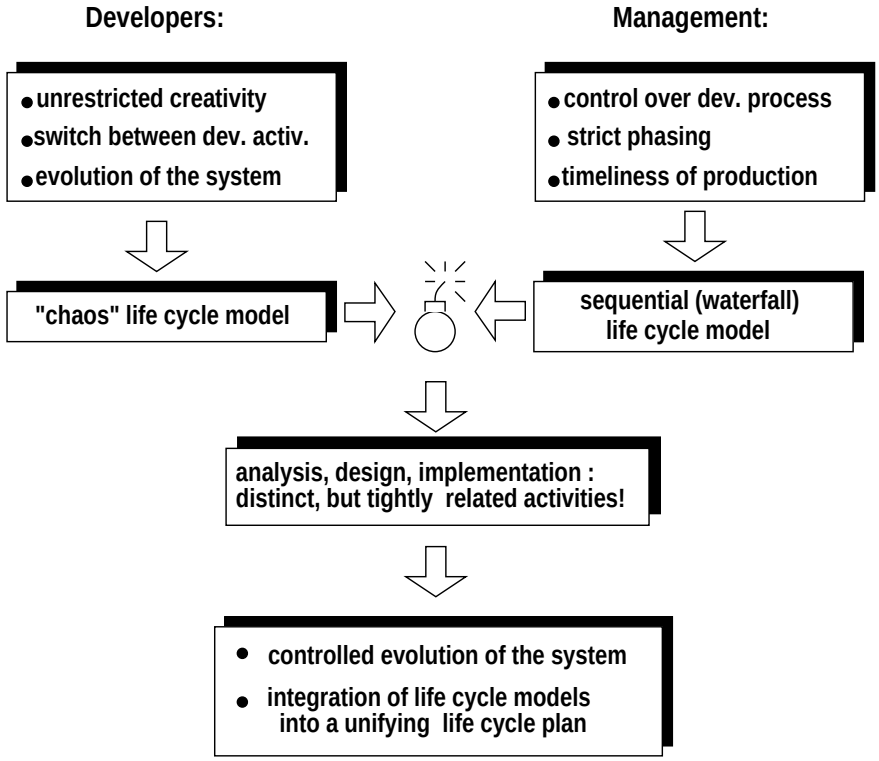


Figure 1: Developers' and management's view of software development

**Communication between users and developers** is another important issue, and is tightly related to the problem of **capturing the sound system requirements**. The reality of daily development work indicates that the traditional sequential process is in many cases not the best way to develop software. Sometimes users or customers have a certain, very specific perception of the future system, but are unable to formulate this perception adequately. Especially for innovative software projects, the requirements are not exactly known or can not be overlooked altogether at

project start. This problem is not solved optimally by waterfall life cycle models, where users are only involved at the very beginning (during requirements specification) and at the end (during $\beta$-testing). This relatively late feedback is a great risk to the system being developed of not meeting the expectations and wishes of the users. Even if an intermediate exchange of ideas is foreseen, i.e., the specification documents are discussed with users, the problem remains that formal, non-executable system specifications are unsuitable for evaluation by (non-computer scientist) users.

To cope with the conflicting views on software development and the communication issue with users, a **controlled evolution of the system** with frequent feedback to the users has to be defined, i.e., an integration of the two life cycle models into a unifying life cycle plan[1] that satisfies the needs of all groups of people involved.

At TNO[2], 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, capturing better the requirements of our customers, meeting the request for more flexibility of the development process by the developers, while at the same time still controlling development time and technical risks of the final product.

Yet, such a modification of the development strategy does not only require a solid re-definition of the life cycle model, but also implies some important **adjustment of the development infrastruture**, i.e., an adaption of the technical and human environment of the production. The aim of this paper is to discuss these infrastructural requirements in view of the experiences we made with evolutionary software development.

Section 2 gives a concise overview of how an incremental and iterative life cycle can be consistently integrated with the project management's sequential view of the software life cycle, using evolutionary prototyping as basis. A comprehensive discussion of the "hybrid" life cycle plan and its integration aspects can be found in [ZG94]. Experiences with several recent, multilateral projects, carried out at TNO Institute of Applied Geoscience, are reported in section 3, and evaluated concerning development time, risk management, and software quality. Technical and strategic requirements related to the evolutionary life cycle plan, as e.g., team structure, communication channels, system integration and version control (and other), are presented in section 4. Section 5 concludes this paper and gives an outlook on future work.

## 2 Evolutionary Prototyping and the Evolutionary Life Cycle Plan

As stated in the introduction, the communication between development team and (potential) users is a key factor to the quality of the software system developed. This is even more crucial when the aim of the development is not a standard application, but an innovative, complex, highly technical and specialized system, as e.g., the application of neural networks for geophysical subsurface modeling. At TNO, these kind of innovative systems are called **experimental systems**, as they interact with the domain-specific research and researchers of TNO.

[1] We distinguish the notion of the conceptual, *descriptive* life cycle *model*, and of the applied *prescriptive* life cycle *plan* [LRR93].

[2] At the TNO Institute for Applied Geoscience, Delft, The Netherlands, we develop information systems and simulation programs for oil & gas exploration and production (cf. section 3).

## Evolutionary Prototyping

Prototyping offers a practicable solution to the problem of validating and capturing better system requirements, hence constructing a more stable system [Bud84]. **Throwaway prototyping** can deliver useful insights about certain limited details of a system, but usually not much effort is taken to make those kind of prototypes qualitatively persistent in regard of software qualities as e.g., efficiency, completeness, etc. [GJM91].

Nowadays, in major projects, the investment in prototyping has become too expensive for the organization to afford to throw the prototypes away. Furthermore, in innovative software development, prototyping has to go beyond addressing single, isolated details of the projected system (e.g. the user interface), but has become an essential means to stimulate imagination and creativity of both the developers and the non-computer-scientific target users. In addition, **reusable components** are becoming more and more available, enabling a rapid production of reliable and functional prototypes.

So, at TNO, we decided to exploit the new potential of **evolutionary prototyping**. Prototyping has matured from offering very limited working models which helped to highlight or validate certain limited aspects of a system to **evolving, full- scale, and persistent repositories** of acquired knowledge, development solutions and decisions [BKKZ92].

As described above, the full set of requirements and possible solutions are often not known or can not be overlooked 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, highly risky parts to explore, the current prototype represents at any time the ideas and visions with respect to conceptual solutions and the system architecture.

Every evaluation cycle decreases the risks inherent to every software development, and enhances the quality of the software product concerning **technical risks** (feasibility, performance, etc.) and **user acceptance** (functionality, adequacy, comfort, etc.).

The new role of prototyping and the intention to achieve more flexibility in the choice of development activities by a more flexible software process have far-reaching consequences for a software development project. The experiences made at TNO reveal the key issues connected to this change of development strategy:

- a revised (evolutionary) life cycle plan and revised change management
- sophisticated (object-oriented) software engineering methodologies
- integration control and version control
- regular testing and evaluation, and short communication paths
- management of reusable components and standards

As the choice of adequate software engineering methodologies is abundant (and somehow an individual process), we only mention *Objectory* [JCJO92] as "our" choice for early development phases. **Object-orientation**, not only for the analysis which captures the intuitive organization of the application, but also for design and implementation, offer the concepts crucial for the construction of an evolutionary prototype (modularization, reusability, encapsulation), and hence also for the final system.

After briefly introducing the life cycle plan underlying the evolutionary development[3], we will dedicate our main attention in this paper to the last three items on the list above in section 4.

## An Evolutionary Life Cycle Plan

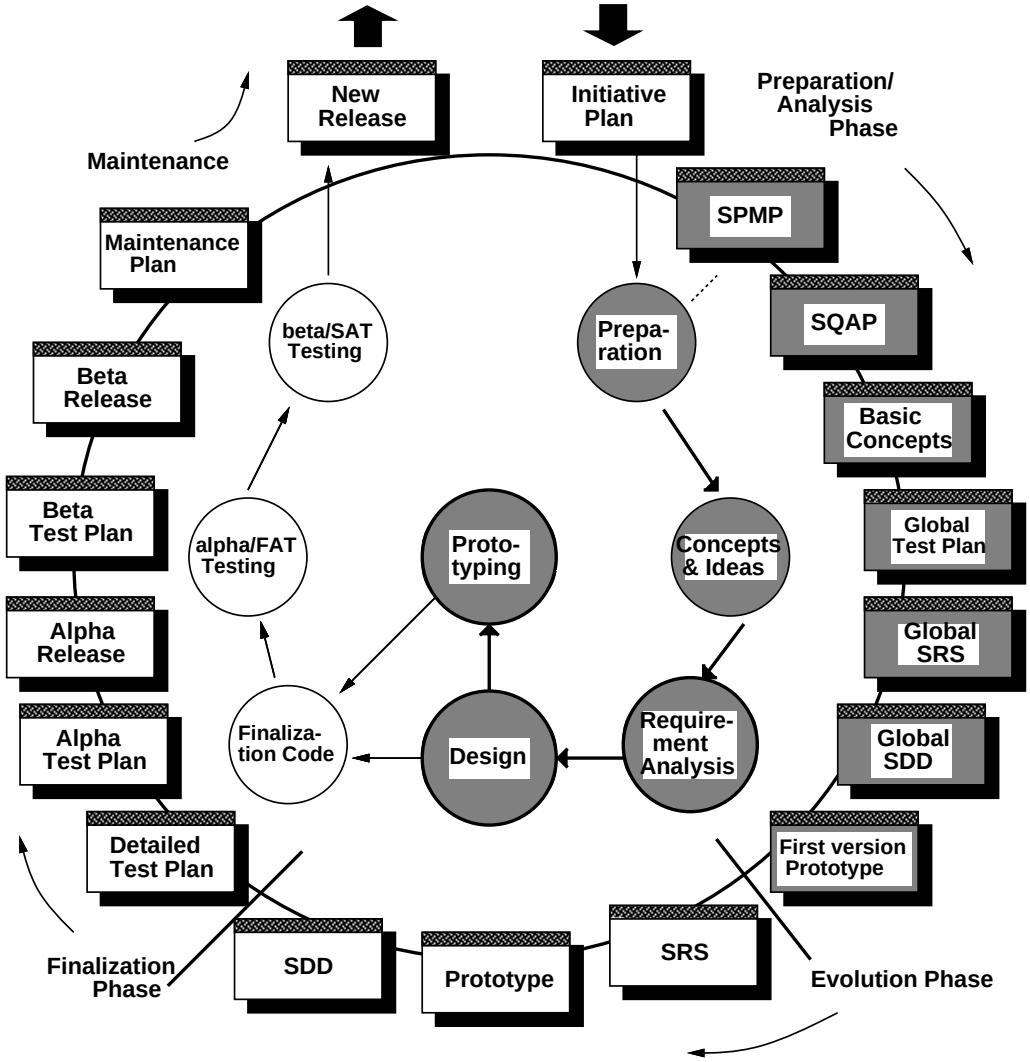


Figure 2: Phase 1 of the "hybrid" software life cycle: Preparation/Analysis[4]

At TNO, we established a life cycle plan that incorporates relatively unrestricted, but controlled evolution of the software product and of its main specification documents. Note that the base document defining the life cycle plan is a **software quality assurance plan** (SQAP) [Ger93] that conforms entirely to the ANSI/IEEE Std. 730-1984[5]. Interpretation of this standard is in

[3]For a detailed description of the integration of the developers' and the management's view of the software development process cf. [ZG94].

[4]`SPMP`: Software Project Management Plan; `SQAP`: Software Quality Assurance Plan; `SRS`: Software Requirements Specification; `SDD`: System Design Document

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

most cases consistent with the ANSI/IEEE Std. 983-1986[6] and other standards[7]. An overview of the evolutionary development life cycle can be found in the figures 2 and 3. In these figures, the iterative, incremental process and its related sequential management perspective are depicted.

The **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 achieve the desired flexibility of an incremental approach, multiple transitions are possible at certain stages (cf. figure 3).

The **outer circle** of *project deliverables* lists the desired order of delivery from the management perspective. Management, software quality assurance, etc. can be seen as taking along this temporal time line. For clarity, not all individual project deliverables are shown and explained in these pictures.

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

Triggered by the initiative plan, during **preparation**, the documents required at project start are produced. The *Basic* (technical) *Concepts*, considered fundamental for the application, are worked out. *(Global) Requirements Analysis* concentrates on the description of how the system should look like. Together with a first *(Global) System Design*, they are implemented in a *First Version of the Prototype* which reflects the global system layout. The construction of the first prototype concludes the first, *non-iterative* part of the software development process (cf. fig 2, gray symbols).

The **evolution phase** is the core of the development. In that phase, 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*, indicated by the dark-gray process symbols and the bidirectional arrows in figure 3. By this iterative, incremental proceeding, individual parts of the system can further be worked out in a **number of cycles** of which each may, *or may not*, include all the three activities of requirements analysis, design, and prototyping. This is an important difference to waterfall models *with iteration*, like the *spiral model* [Boe88], as they impose an unambiguous order of activities, and iteration is seen as a repetition of a well-defined sequence of activities done before.

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.

At the end of this evolution, the code of the incrementally developed prototype is completed and brought into a state of **finalization**, meeting the quality criteria defined in the software quality assurance plan (SQAP). Acceptance is ensured by ($\alpha$- & $\beta$-) *Testing*, and preparations for *Maintenance* are taken (cf. fig. 3, light-gray symbols).

Note that in this software process, **change management** is defined as a change of the "Basic Concepts" document *after* the preparation/analysis phase (phase 1), or a change of the "Software Requirements Specification" or the "System Design Document" *after* the evolution phase.

The purpose of this brief overview of the life cycle plan was to place its evolutionary part in the context of the whole software life cycle. In the next section, we will illustrate the impact of the evolutionary life cycle for projects developed at TNO.

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

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

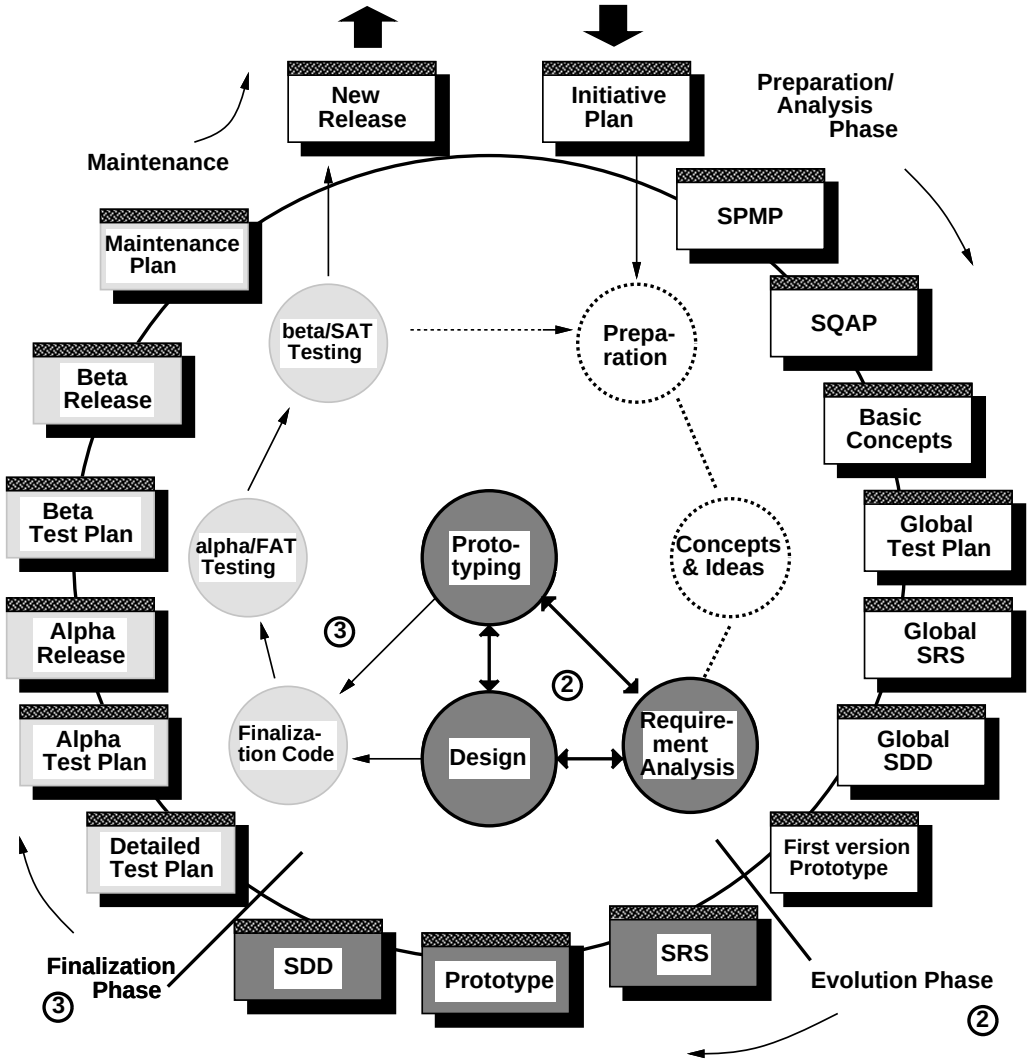


Figure 3: Phase 2 and 3 of the "hybrid" software life cycle: Evolution Phase and Finalization Phase[7]

# 3 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 geo-scientific research. These kind of projects lead to so-called **experimental software systems**, a term that reflects that these systems implement the results of geo-scientific research (performed e.g. at TNO), and at the same time support and enable that research. As the success and the impact of the resulting systems

usually can not be predicted, these projects 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, combining research with the production of operative software systems, that triggered the elaboration of a different life cycle plan.

To evaluate the impact of the new development strategy, we compared TNO research & development projects carried out with the evolutionary approach presented so far, with TNO (pure production) projects developed according to the traditional waterfall life cycle model. All the projects included distributed, i.e., multi-site, software development and dealt with either **information systems for reservoir data management**, or with **reservoir characterization by seismic interpretation**. Table 1 lists a summary of some of the most important project parameters for all projects examined[8].

| Project parameter: | Value: |
|---|---|
| total duration | 18 - 30 months |
| total man-hours | 3100 - 9600 |
| overall costs (k US$) | $\leq$ 1300 |
| team size | 4 - 15 |
| platform, tools | SUN, DEC, SGI, Cray YMP-EL, POSIX, OSF/Motif, XFaceMaker, Uniface, Oracle DBMS, Lotus 1-2-3, XV (image display), SU (Seismic Unix), WingZ (spreadsheet), Asprine/Migraine (PD neural network) |
| languages | C, C++, Uniface4GL, WingZ-Hyperscript, SQL |
| development tools | Uniface, X-NIAM, (ER tool), Software trough Pictures (StP), XFaceMaker, Centerline, RCS/CVS, proprietary tools |

Table 1: Project parameters for the projects examined

Because of the particular development situation at TNO, sketched above, no reference or "standard" projects or systems were available. Furthermore, it was very difficult to "measure" *risk decrease* otherwise than to judge on the statements and assessments of users and developers.

As a first approach, we examined the amount of time allocated to the different development activities. The absolute numbers were "normalized" to give the percentages of total project man-hours dedicated to the different activities, because though projects were comparable in size and complexity, they were not exactly the same. Usually, the research & development projects were larger, more complex, and more innovative.

Figure 4 gives an overview of the relative amount of time used for the various development activities for the different types of projects. The exact numbers for the different projects have been published in [ZG94].

Management activity hours are not explicitly stated, but included in the numbers for the respective development phases. Furthermore, for the projects following an evolutionary life cycle plan, only the main evolution cycles between so-called *sponsor meetings*, where the (prototypes of

[8] Although the range of "values" in table 1 is rather large, we actually compared pairs of very similar sequential and evolutionary projects.

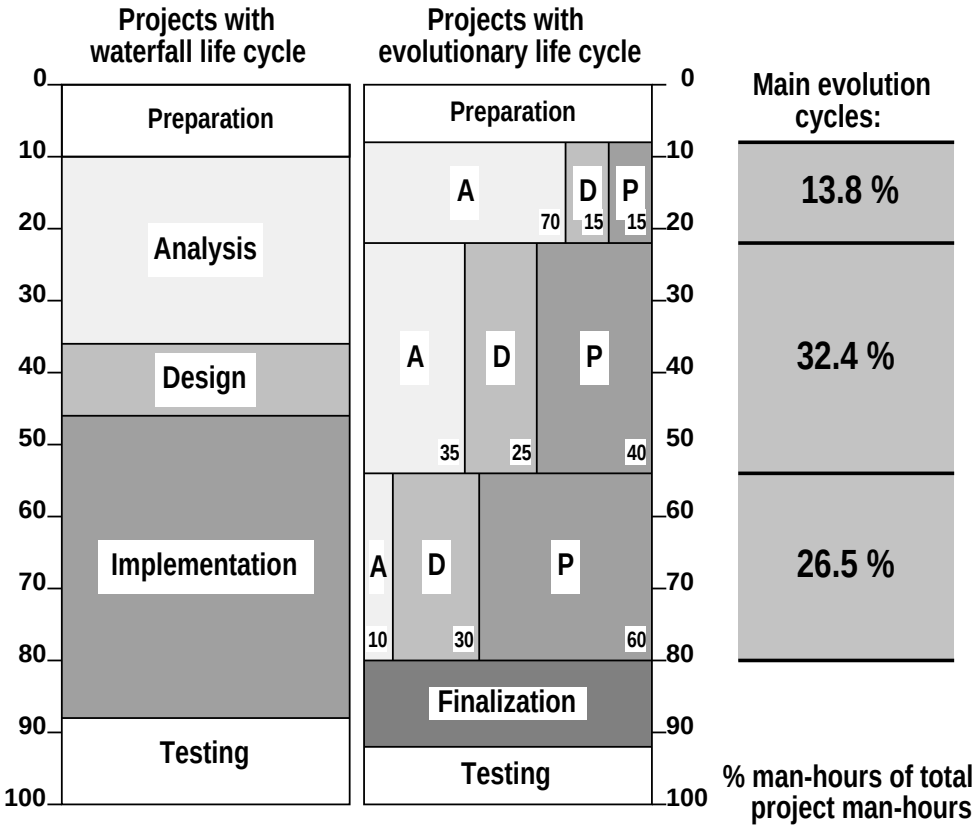


Figure 4: Overview of development activities for projects with "waterfall" and evolutionary life cycle

the) system were presented to a larger public, are depicted. The fine-grain two-weekly cycles are not shown.

In projects with the sequential life cycle, a first version of the system was available only in the implementation phase, after about 80% of the total project time, with the first complete version being ready only after more than 87% of the total project time. As consequence, feedback about adequacy and performance of the system occured only at a very late project stage, while the risk of the project was still relatively high. Once, this lead even to a complete redesign and reimplementation after $\beta$-testing had revealed major shortcommings of the system developed.

At a comparable point in time (at approx. 80% of project time), projects following the evolutionary life cycle model were still in their finalization phase, but feedback had been given already several times, i.e., at least three times (at 21.4%, at 53.8%, and at 80.3% of the total project time) if only the main cycles between sponsor meetings are taken into consideration.

Figure 4 shows also very clearly how the percentage of activities shifts during the three evolution cycles from mainly requirements analysis (`A`) to mainly prototyping (`P`). The small numbers at the bottom of each box give the percentage of the activity within the respective main evolution cycle.

Table 2 gives an overview of the total percentage of time allocated for requirements analysis, design, and coding activities. Design activities have a considerably bigger share of the total project man-hours in projects with the evolutionary approach (18.1% compared to 10.8% for sequential projects). This isn't primarily an indication for more complex systems, but mainly due to the fact that evolutionary prototyping requires a *better-designed system architecture* which does not only

| Sequential life cycle | | Evolutionary life cycle |
|---|---|---|
| 9.2% | Preparation | 7.6% |
| 25.9% | Analysis | 24.2% |
| 10.8% | Design | 18.1% |
| | Prototyping | 31.2% |
| 41.2% | Coding (total) | 42.7% |
| 12.9% | Testing | 7.4% |

Table 2: Overall distribution of relative man-hours used for development activities

aim at an optimal final product, but also facilitates reconstruction and change of its subcomponents during development. On the other side, due to prototyping, $\alpha$- and $\beta$-testing of the final product require much less time (7.4%, compared to 12.9% for sequential projects).

The overall distribution of project man-hours (table 2) shows also that the effort put into the main development activities (analysis, design, coding) is approximately equal for the two types of projects. The big difference is how these efforts are distributed, i.e., how the total effort put into the core development activities is "fragmented" in the evolutionary software process.

### Assets

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 that can not be drawn up against other.

What comes closest to an absolute estimation is a cautious statement that with the new approach "*we were able to develop more complex systems with the same order of magnitude of resources we used before*". 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 5, where the risk curves for traditional, sequentially phased projects and for the evolutionary development strategy 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 and design activities are not as detailed as for the sequential approach. But this changes with every prototype that is introduced. 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.

## 4 Technical and Strategic Requirements

The most significant difference between the traditional sequential life cycle models and the evolutionary approach are that the **blurred boundaries between the development activities**

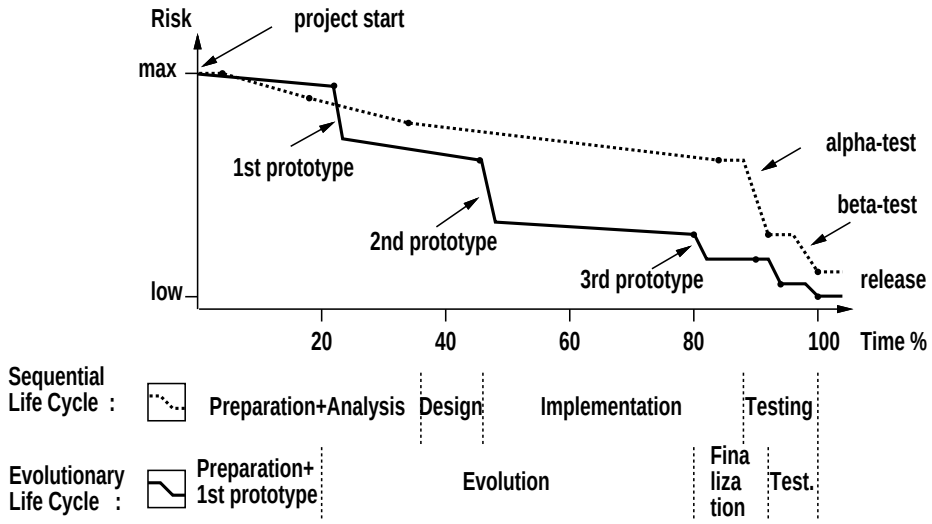


Figure 5: The risk curves for sequential and evolutionary development compared.

analysis, design, and implementation. In the evolution phase, the decision which activity to perform, i.e., on which kind of specification document to work, becomes flexible, even if none of the relevant deliverables has been worked out in full detail.

The direct consequence of the undetermined order of specification activities in the evolutionary software process is that the traditional specification documents software requirements specification (SRS) and system design document (SDD) are **no longer stable anchor points of development** prior to the beginning of a consecutive activity. Requirements specification and design document **evolve together with the prototype**, and reach stability only at the end of the evolution phase. Hence, the evolving prototype acquires two crucial roles, becoming:

- the **means of communication** with users and domain experts, i.e., the mediator to reach agreement with the users and meet their expectations, and
- the **central project repository** about acquired specification knowledge, design decisions, and development solutions, and therefore also the primary source for evaluation of development progress.

This fundamental change of requirements to the project documents and the system to be built has to be compensated by an adaption of the **technical** (integration & version control, test plan, reusable components) and **human** (team structure, communication channels) **environment**, as discussed in the following.

## 4.1 Integration Control: Integration Manager and Controlled Environment

Aim of integration control is to coordinate the prototyping activities of the developers at the different sites, and to regularly and frequently provide running versions of the system, i.e., prototypes, that can be tested and evaluated. Therefore, the special task of an **integration manager** is defined (cf. fig. 6). This task has to be filled by a *senior programmer*, as one of his roles within the development team, or as full-time job, depending on the project complexity.

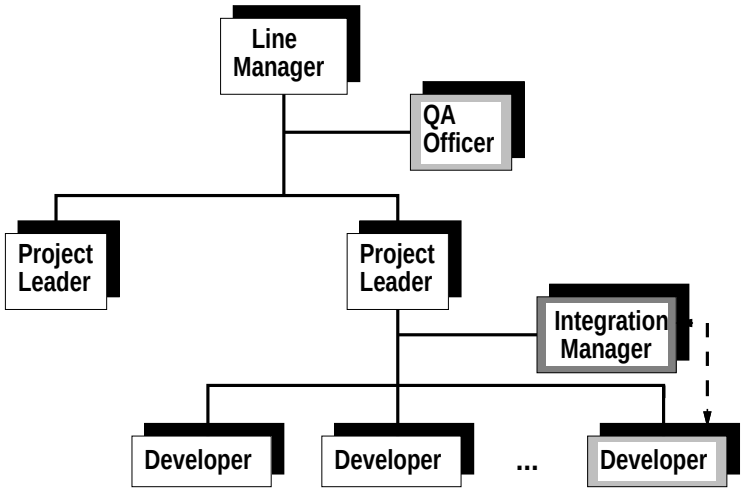


Figure 6: The position of integration manager within a development team

The integration manager is responsible for regularly "collecting" the different system "patches" (modules, classes, etc.) into an integration environment, exclusively administered by him. At TNO, this integration environment is called **controlled environment**, because it acts as filter between those team members committed to development and those in charge of evaluation of the prototypes.

As the integration manager constructs the running versions of the prototype, he is an active **decision maker about design decisions**, but not responsible or in charge of correcting the components he receives. In case of errors that prevent a proper integration, the integration manager gives direct feedback to the respective developer. In case of conflicts, the project manager might have to be called in.

The task of an integration manager is *product-oriented*, i.e., to ensure the technical quality of the system prototypes, including the management of "meta"-information about the procedure to construct the system. This profile discriminates him from the Quality Assurance (QA) Officer (cf. fig. 6), whose responsibilities are *procedure-oriented*, assuring the quality of the production processes.

Figure 7 gives an overview of the course of events and the **communication during the evolution phase**. The integration manager collects the different parts, produced individually by the developers, and integrates them to a running version of the prototype. This version of the prototype is then tested and evaluated by an appropriate group of users and/or domain experts (cf. sec. 4.3 for a discussion about appropriateness of "testers"). Feedback is given to the developers, via the integration manager, if not possible otherwise. The developers change the parts for which they are responsible, thereby initiating a new "integration-evaluation-feedback" loop. A change of the code is eventually related to a change of parts of the specification documents, but this issue is not further discussed here, as responsibility for integration and consistency of the specification documents can be planned individually (per document and per project).

Two aspects concerning this proceeding are very important. Firstly, integration and evaluation have to take place *as often as possible* in order to guarantee **continuity of the evolution process**. At TNO, a period of 14-20 days between two consecutive versions of a prototype and their evaluation is scheduled (and maintained). This might look like an over-proportional increase of additional tasks during development, but the more often a system is being evaluated, the smaller the (new)

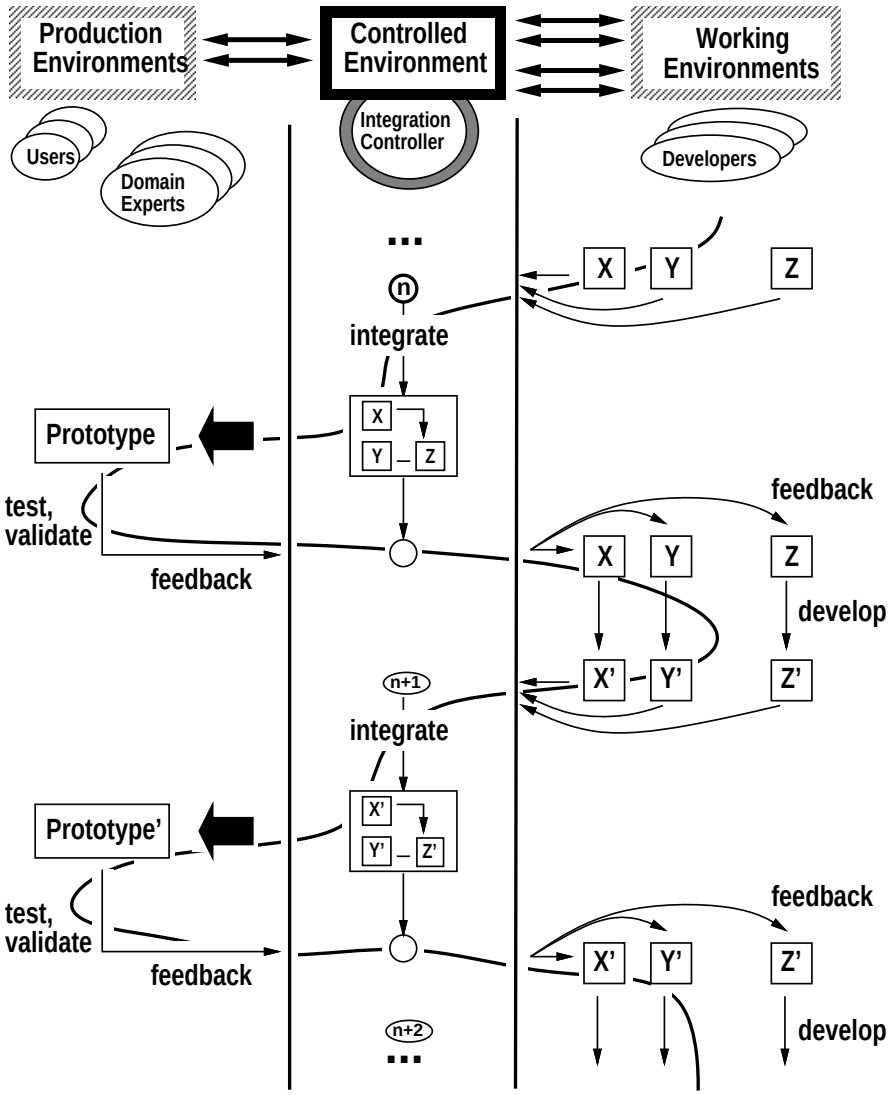


Figure 7: Overview of the evaluation cycles

increments really to be tested are. This means that the total amount of testing activity doesn't increase when the frequency of the tests augments.Committing people to test a new system during development has even more advantages, as discussed in section 4.3.

Secondly, strict physical **separation of the environments**, i.e., of the (distributed) working environments of the developers, the controlled environment, and the (distributed) environment(s) in which the prototypes are tested (e.g., the production environment(s)), has to be maintained. This ensures that the current status of the evolving system is always unambiguously determinable, and that construction and evaluation of the system can take place independently.

## 4.2 Technical Support: Version Control and Early Test Plans

As the evolutionary prototype is the core and repository of development knowledge, the controlled environment, where the prototypes are integrated and administrated, requires sophisticated tech-

nical support and management. Main task over time is **version control** to be able to discuss and compare different versions of the prototype, and to be able to roll back to previous versions of the prototype and to follow other branches of the prototype version tree if that is regarded to be necessary.

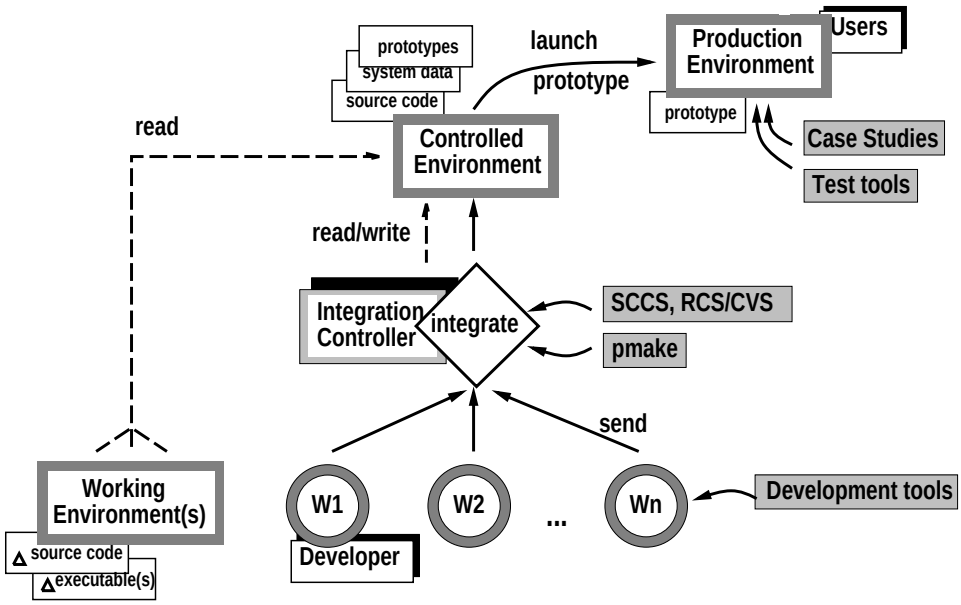


Figure 8: The different environments and their tools

Developers within their development environment have to *read-access* the prototype(s) in the controlled environment to be able to harmonize their work with the complete system. In the production (testing) environment, **sophisticated testing tools** have to support the evaluation of the prototypes. The situation, together with the types of tools and of code is depicted in figure 8.

Apart from technical support, another prerequisite for evolutionary prototyping is the **early elaboration of a test plan** (cf. fig 2). It is not sufficient to collect only potential sets of synthetic test data. It is also necessary to define larger-scale, "real-life" **test scenarios** and even **case studies**. During the course of evolution, it will gradually become possible to test not only isolated components or aspects, but soon also the whole system and its functionality. The size and complexity of the test cases have to match this growth of the system.

### 4.3 Team Structure: Developers, Management, Users and Experts

In figure 7, one group of people mentioned before can be detected that normally interferes with the project only at the beginning, when requirements are collected, and at the end, when the system is $\beta$-tested: the *users* and the *domain experts* (which often are the same persons for technical, scientific applications). **Commitment of users** and **short communication channels** between users and the development team are important prerequisites for an effective application of evolutionary prototyping, too.

The communication channels between users and developers are *bi-directional*. Of course, feedback about the current version of the prototype has to be given to the developers by the users. But it is also necessary that the integration manager prepares **testing information** for the designated

testers about which increment of the system has been added or changed in the current prototype to direct their testing in the right direction, minimizing and focusing their efforts.

Another aspect of user involvement is the **selection of the appropriate types of users** for evaluation of an actual version of the prototype. This selection depends from the state of completion of the system. Obviously, not every potential user should be burdened with testing partial or unstable prototypes. Especially in the early stages of evolution, when the technically most critical parts are prototyped, "prototypical users" may be recruited within the development team, e.g., the integration manager himself. Eventually, as functionality and stability of the prototype increase, the circle of testers can be broadened.

As consequence of committing users to test, hence to "use" the future system already *during development*, not only the risk of acceptance decreases. If future users are identical to the test users, this causes a significant **decrease of the learning expenses** concerning the introduction of the final system at the users' site, because the users have already been accustomed to the system during its development. Even if the future users are different from the prototypical ones, the experience of the test users during the evaluation cycles can be used when training other users. Consequence is an **abridged overall "development+introduction" cycle**. An additional edge arises from the fact that the case studies performed during development (for testing) can immediately be used to illustrate and promote the final system.

**Management profile** requires also adjustment. (At least some) **technical and domain-specific knowledge** is necessary to cope with the important tasks of the:

- **identification of the most risky parts of the application** (technically, or regarding user acceptance) in the preparation phase, as these are the starting point of prototyping,
- **monitoring of the progress** in the evolution cycles, as completion of specification documents can't be used for measurement and change management has changed considerably (cf. sec. 2).

### 4.4 Reusable Components and Standards

Although reusability is a general quality of software development, the use of **reusable components** is an especially important prerequisite for effective, meaningful prototyping. As described in section 2, main concern of the prototyping efforts can be technical aspects as well as user acceptance (e.g., user interface). Hence, reusable components should be available for all areas of system specification and construction. We consider product-oriented **standards** in the same scope as reusable components, because they represent *reusable conceptual components*. Both facilitate development efforts by avoiding redundant (specification or production) work.

We distinguish (and apply) three different groups of reusable components:

1. **generic components**, as e.g. help facilities, error handling, image displaying, user interface widgets
2. **domain-specific components**, as e.g. domain-specific class libraries (e.g., oil exploration & production (E& P) widgets), POSC[9] standard data bases and data definitions, standard data access libraries
3. **application-specific components**, as e.g., reused proprietary components, Seismic Unix (SU) for seismic data processing, 3D-modeling tools (e.g., GoCad)

[9] **POSC** is the acronym of the standardization organization **P**etrotechnical **O**pen **S**oftware **C**orporation.

These groups cover the whole range of reusable conceptual, design, and implementation components (and product-oriented standards). Although the most useful components for a straightforward construction of prototypes are the design components (classes), because of their implementation specificity they are also unfortunately the most problematic ones to reuse.

# 5 Conclusions

The benefits of our approach to integrate the developers' and the management's perspective of the software life cycle based on evolutionary prototyping have been confirmed by the outcome of the projects surveyed. This holds for the positive results as the sophisticated capturing of requirements and wishes of the users, as well as for the early detection of risks and shortcomings.

In this paper, we focussed on presenting the most crucial technical and strategic issues to support (controlled) evolutionary software development. Team structure, commitment of, and communication with users, integration & version control, management capabilities, and component reuse become important issues when replacing a traditional sequential by an evolutionary life cycle plan.

To be able to measure quantitatively and more precisely the impact of the evolutionary life cycle model, two steps have to be taken, indicating the areas of our future work. **Other metrics** (as e.g. comparing the number of *change proposals*) as indication for the success of the evolutionary strategy have to be defined and applied, and **more projects** have to be evaluated to achieve more statistical certainty and to be able to neglect the individuality of each (experimental) software development.

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

# References


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

[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.

[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`).

[ZG94] Andreas Zamperoni and Bart Gerritsen. Integrating the Developers' and the Managerial Perspective of an Incremental Development Life Cycle. In *Proc. of the 12th Annual Pacific Northwest Software Quality Conference, Portland, USA*, pages 227–242, October 1994. Also as Technical Report 94-22, Leiden University, Dept. of Comp. Science, The Netherlands (Available by *anonymous ftp* from `ftp.wi.leidenuniv.nl` in `/pub/cs-techreports` as `tr94-22.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.