Controlling software for EMF laboratory studies

Author(s):
Oesch, Walter

Publication Date:
2006

Permanent Link:
https://doi.org/10.3929/ethz-a-005174613

Rights / License:
In Copyright - Non-Commercial Use Permitted
Controlling software for EMF laboratory studies

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH
for the degree of
Doctor of Technical Science

presented by
Walter Oesch
Dipl. Natw. ETH Zurich
Born February 25, 1974
From Oberlangenegg, Bern

Accepted on the recommendation of
Prof. Dr. W. Fichtner, examiner
Prof. Dr. N. Kuster, Dr. M. Burkhardt, co-examiners

2006
Contents

Summary v
Zusammenfassung ix
Acknowledgments xiii

1 Introduction 1
1.1 Non-ionization EMF exposure 2
1.2 ELF exposure 3
1.3 RF exposure 3
1.4 EMF laboratory studies 4
1.4.1 In vitro studies 4
1.4.2 In vivo studies 5
1.4.3 Human studies 5
1.4.4 Limits of EMF laboratory studies 6
1.4.5 Controlling software for EMF laboratory studies in this thesis 6
1.5 Software engineering 7
1.5.1 Software systems 7
1.5.2 Software engineering methodologies 8
1.5.3 Software engineering in this thesis 11

2 EMF laboratory studies 13
2.1 Exposure and evaluation system 13
2.2 Experiment conditions 15
2.2.1 Exposure conditions 15
2.3 Design of exposure setups 16
2.3.1 Dosimetry 16
2.3.2 Design concessions 17
2.3.3 Evolution of exposure systems 17
2.3.4 Controlling software 18
2.4 Requirements 19
2.4.1 Exposure uniformity 19
2.4.2 Exposure strength 19
2.4.3 Signals 20
2.4.4 Good exposure - sham isolation 20
2.4.5 Double blind study design 20
2.4.6 Intermittent exposure protocols 21
2.4.7 Resonance structure 21
2.4.8 Human safety 21
2.4.9 Data monitoring 22
2.4.10 Data logging and security 22
2.4.11 Easy handling for non-engineering personnel 23
2.4.12 Self-detection of malfunction 23
2.4.13 Highest possible system stability 24
2.5 Summary 24

3 Software engineering methodology 25
3.1 The system development life cycle 25
3.1.1 Planning 25
3.1.2 Analysis 26
3.1.3 Design 26
3.1.4 Implementation 26
3.2 Evolution of software engineering methodologies 27
3.3 Object-oriented analysis and design (OOAD) 28
3.3.1 Basic characteristics of object-oriented development 29
3.3.2 Encapsulation 29
3.3.3 Inheritance 30
3.3.4 Polymorphism 30
3.4 OOAD design principles 31
3.4.1 Use-case driven 31
3.4.2 Architecture centric 32
3.4.3 Iterative and incremental 33
3.5 The unified modeling language (UML) 33
3.5.1 Use-case diagram 34
3.5.2 Component diagram 35
3.5.3 Deployment diagram 36
3.5.4 Class diagram 36
3.5.5 Sequence diagram 38
3.5.6 Activity diagram 39
3.5.7 Conclusions about the UML 40
3.6 Object-oriented design quality criteria 40
3.6.1 High cohesion 40
3.6.2 Low coupling 41
3.6.3 Connascence 41
3.7 Object-oriented design patterns 41
3.8 Real-time software design 42
3.8.1 Data acquisition systems 44
3.9 Heavyweight software engineering methodologies 44
3.9.1 OOAD with UML 45
3.10 Lightweight or agile software engineering methodologies 47
3.10.1 Key features of agile methodologies 49
3.10.2 Extreme programming XP 53
3.10.3 Crystal family 55
3.10.4 Lean software development 59
3.10.5 Pragmatic Programming 62
Seite Leer / Blank leaf
Summary

Perfection is the worst enemy of the good.

This PhD thesis addresses two topics: 1) the controlling software for electromagnetic field (EMF) laboratory studies investigating possible health risks of EMF exposure and 2) the critical evaluation of the benefit of both established and novel software engineering methodologies and the assessment of their suitability for complex and real-world software systems based on the experience with the development of exposure systems.

Controlling software for EMF laboratory studies

The use of wireless communication systems in the range of a few MHz to a few GHz has increased exponentially in the last decade. Constantly increasing microprocessor clock rates and the growing market of mobile telecommunication equipment and services are the main driving forces of this trend. Customer groups and health agencies have been concerned about the increase of wireless communication systems and have requested the government and manufacturers to provide scientifically based compliance assessment for this technology.

Numerous experiments addressing the possible negative health effects of EMF have been conducted in recent years. Various research groups have performed animal, cell and human studies with different scientific approaches. Several experiments have turned out to be of limited value due to severe shortcomings in the exposure setup. A major point of criticism is the lack of mature controlling software of most setups, having some important drawbacks. Firstly, biomedical teams with generally little technical knowledge have to manipulate radio frequency (RF) equipment. Subsequent possible handling errors are not detectable, and the investigated items are exposed to an unpredictable radiation dose. A second point addresses the reconstruction of the experiments. Systems without controlling software are unable to collect and store data. Therefore, retrospectively asserting the accuracy of the experiments is not possible. Thirdly, simulating complex real-world signal shapes like GSM or UMTS is only possible with software. Finally, to obtain high quality results, biomedical studies have to be performed under blind conditions, i.e., the experimenter must not be aware whether an active field exposure or the sham control is applied. In this way, unconscious (or even conscious) manipulation of the experiment is not possible. Fully double-blinded study design is only possible with controlling software implementing a random decision maker and enabling the encoding of exposure data.

The first part of this PhD thesis describes the basic requirements of modern exposure systems (in vitro, in vivo and human). Analyzing these requirements reveals the impor-
tance of well-adapted controlling software. Only systems with sophisticated controlling software can meet all requirements. Up to the present, no exposure system with software tightly controlling the experiments has been developed nor described in the literature. The purpose of this topic is to close this gap and to provide designers of exposure setups with basic guidelines for the creation of controlling software.

An exposure setup's scientific core is the transfer function, combining RF power and/or electromagnetic field strength with the specific absorption rate (SAR) of the investigated subject. Usually, a combination of measurements and simulations reveals this connection. Depending on the setup category (in vivo, in vitro, human), EM coupling mechanism and setup design, the transfer function consists of different parameters. This thesis extracts elements common to various transfer functions and suggests a generic function valid for all setup types. The generic approach enables the transfer function's encapsulation and simplifies the implementation and exchangeability of power feedback algorithms. Moreover, maintaining a strict separation of data acquisition and data conversion enables the creation of powerful software design patterns occurring, which can be used in any data acquisition system.

The realized controlling software was applied in more than 20 different worldwide laboratory studies for cells, animals and humans. The controlling software enabled to perform animal studies under NTP-like (National Toxicology Program) GLP (Good Laboratory Practice) conditions. The derived mathematical GSM signal model and the implementation of this model became standard in BIOEM (bioelectromagnetic) research. This signal model represents the first exposure scenario that applies temporal changes of different modulation schemes as occurring in an actual phone conversation. This signal contains a cocktail of low-frequency modulation components. Studies in the past neglected the influence of these low-frequency components.

**Software engineering methodologies**

Based on the experienced gained during the implementation of several large-scale software projects as presented in the first part of this thesis, the importance of the application of suitable software engineering methodologies became clear and is analyzed in the second part.

Software engineering has constantly tried to systematize the development process. To this end, very different software engineering methodologies have been proposed. In the last decade, two main topics have ruled the discipline of software engineering: 1) the invention and enhancements of the Unified Modeling Language (UML) as a real standard for modeling object-oriented software systems (this fact has boosted traditional heavyweight methodologies) and 2) the emergence of agile software engineering methodologies.

As a result, today software engineering includes two sets of fundamentally different methodologies: heavyweight methodologies and agile methodologies. Heavyweight methodologies follow the traditional way of constructing software. The core of heavyweight methodologies constitutes the separation of design and implementation activities. Agile methodologies on the other hand focus on incremental software development,
close cooperation with the customer, simple solutions and fast reaction time to changing conditions.

The second part of this thesis describes the specific heavyweight and agile methodologies based on the methodologies' most important features. Similarities and varieties are identified and discussed. The description of the methodologies serves as the basis for the subsequent evaluation of the presented software engineering methodologies for practical use.

The evaluation has shown that no methodology is inherently suitable for all situations and for all organizations. Each methodology has its strengths and weaknesses, which have positive effects in different situations. Agile methodologies seem to be particularly suitable in situations where the requirements are likely to change in the future, whereas heavyweight methodologies deploy their full potential if future requirements (performance, exception handling, functionality) are known and can be specified in detail.
Zusammenfassung

Perfektion ist der ärgste Feind des Guten.


Steuerungssoftware für Expositionseinrichtungen


Softwareentwicklungstechniken

Seit jeher hat die Softwareentwicklung versucht, den Entwicklungsprozess zu systematisieren. Aus diesem Grund wurden im Verlaufe der Zeit sehr unterschiedliche Softwareentwicklungstechniken erfunden. Im letzten Jahrzehnt haben zwei Tendenzen die Disziplin Softwareentwicklung beherrscht: 1) die Entwicklung und die konstante Erweiterung der Unified Modeling Language (UML) als effektiver Standard zur Modellierung objektorientierter Softwaresysteme (was schwergewichtigen Entwicklungstechniken einen enormen Aufschwung bescherte) und 2) die Entstehung agiler Softwareentwicklungstechniken.

Der zweite Teil dieser Arbeit beschreibt die einzelnen schwergewichtigen und agilen Entwicklungsmethoden anhand der wichtigsten Eigenschaften dieser Methoden. Gemeinsamkeiten und Unterschiede der einzelnen Methoden werden aufgezeigt und besprochen. Die Beschreibung der Entwicklungsmethoden dient als Basis für die nachfolgende Untersuchung der Methoden auf ihre Praxistauglichkeit.

Die Untersuchung hat gezeigt, dass keine Entwicklungsmethode in allen Situationen und für alle Organisationen per se geeignet ist. Jede Methode hat ihre Stärken und Schwächen, welche sich positiv oder negativ in unterschiedlichen Situationen auswirken. Agile Entwicklungsmethoden scheinen besonders gut in Situationen geeignet zu sein, in denen die Anforderungen im Verlaufe des Entwicklungsvorganges mit großer Wahrscheinlichkeit ändern, wohingegen schwergewichtige Entwicklungsmethoden ihr volles Potential entfalten können, wenn die zukünftigen Anforderungen (Leistung, Ausnahmebehandlung, Funktionalität) bereits bekannt sind und im Detail spezifiziert werden können.

Seite Leer / Blank leaf
Acknowledgments

My deep gratitude goes first to the supervisor of my thesis, Prof. Dr. Wolfgang Fichtner. He gave me the opportunity to work within his excellent group. Additionally, he gave me the possibility to finalize my thesis according to my ideas by agreeing to transfer me to his institute.

I wish to express my gratitude to Micheal Burkhardt for immediately accepting to be co-examiner of my thesis.

I would like to express many thanks to Dölf Aemmer. This thesis would never have been possible without his mental support. With his calm and composed character, he helped me in difficult personal situations and was like a father for my thesis. Dölf perused and corrected my thesis and thereby revealed the secrets of scientific writing. My special thanks go to Christine Haller, who had to accomplish a lot of paperwork associated with my person. Thanks also to all the IIS members, especially to Norbert Felber, Hanspeter Mathys and my office mates Stephan Oetiker, Frank Gürkaynak, Matthias Brändli and Stefan Eberli, for the very pleasant atmosphere at the institute.

Many thanks go to Prof. Dr. Niels Kuster. Especially, I would like to thank Niels for having had all the benefits of his outstanding scientific competence and his ability to solve complicated technical problems in a short time. Niels always had an open ear for my questions and even had time to have a look into my source code. Still, I am at odds with him as to whether presentations should mainly impress the audience or impart knowledge.

At this point, I want to thank my former and present colleges, who helped my in one way or the other. For the hardware side of my work, I want to address my special thanks to Rainer Mertens, Urs Frauenknecht, Jean-Claude Gröbli, Nico Vetterli, Roy Hunziger, Peter Fleischmann, Fin Bomholt, Bruno Reumer and Manual Lindemann for their help with all kinds of hardware equipment, Jürg Fröhlich and Sven Ebert for discussions on all sorts of dosimetry issues.

I am also in dept to the people whit whom I had the opportunity to talk about programming topics. Particularly, discussing with Hans-Ueli Gerber was greatly inspiring to me and helped me to become a better programmer. His enthusiasm, his searching for simple solutions, his exact method of working (some might say his finickiness) combined with his pragmatism revealed the wide and beautiful world of programming to me. Emilio Cherubini, the decent programming wunderkind (He knows everything, from deeply covert compiler settings to special characteristics of the Unicode character set. At least I always won the ping-pong matches against him (ok, once I let him win :-), otherwise I most probably would have suffered inferiority complexes.), solved many pro-
gramming problems for me - thank you. Thanks also to Peter Sepan, Oliver Egger, Andreas Christ, Harald Songoro and David Perels for their valuable hints.

Here, I want to thank Michelle Stubbs, who had the “pleasure” of proofreading all my English texts. I also want to acknowledge the administrative efforts carried out by Monica Lewis, Jonathan Gubler, Sibyl Akrong and Jaqueline Piper.

Martin Loeser was a good ping-pong partner and an enjoyable conversation partner.

My very special thanks go to Jürgen Schuderer. Working together with him on the REFLEX project was the most exciting time of my work at IT'IS. His huge expertise, his great engagement together with his companionableness made him the ideal fellow. I would appreciate the opportunity to work together again with Juergen in a team at any time. Additionally, Jürgen was a serious ping-pong partner and an enjoyable conversational partner about fictional and adventure literature.

Verónica Berdiñás Torres was my office mate for one year. We had a good time together and laughed a lot.

I also would like to thank my close friends Sam Anderegg, Renato Nanni, Stefan Kneubühl, Oliver Rhas (I especially enjoyed our regular Kahuna board games at our local restaurant), Irene Keller, Daniela Hadorn, Sabine Zürcher, Edith Keller and Antonella Fortunato, who showed me that life is much more than just work.


In the end, I want to express my deepest thanks to my future wife Eva for her endless love and caring for me. She was always there when I needed comfort. Her beautiful eyes and her curious and open nature were the driving forces for my efforts. Eva also improved the quality of my work by carefully proofreading some parts of my thesis for simplicity and consistency.
1 Introduction

This PhD thesis covers two interwoven topics. One topic discusses controlling software for electromagnetic field (EMF) laboratory studies. The second topic introduces software engineering methodologies. It presents and critically evaluates the benefit of both established and novel software engineering methodologies and assesses their suitability for complex and real-world software systems. The two topics are closely connected and cannot be addressed separately. Therefore, some chapters discuss both topics in parallel.

The discussion of the software engineering methodologies composes the theoretical part of this PhD thesis. The specific methodologies are described based on the methodologies' most important features. Similarities and varieties are identified and discussed. The description of the controlling software for EMF laboratory studies is so to speak the application-oriented or the experimental part of this PhD thesis. The goal is to identify how useful the software engineering methodologies (= theory) are on the realization of controlling software for EMF laboratory studies (application). Afterwards, based on this consideration, the evaluation and validation of the software engineering methodologies becomes possible. Expressed differently, the theory is validated by the experiment.

While planning and implementing controlling software for EMF laboratory studies fulfilling a large set of requirements, I have faced various challenges. Some of these challenges concerned the concrete realization of the controlling software, whereas other challenges affected more general software engineering topics.

The challenges concerning the concrete realization were:

- specification of a prototype system
- determination of the functionality of the system
- implementation of a real-time software system
- simulation of synthetic exposure signals.

The challenges concerning general software engineering topics were:

- creation of flexible and extensible controlling software for different projects with the same core functionality
- versioning of the various controlling software projects
- switching between projects
- maintenance and support for various controlling software projects
work efficiently in a team

- assessment of different software technologies

- gathering of information floating somewhere in the development team or present at the customer

- simulation of synthetic exposure signals

- dealing with uncertain and changing requirements

- changing and unclear project responsibilities.

Over time, the realization has emerged that unclarity and project management related difficulties (changing requirements, unclear project responsibilities, gathering information, flexible software, maintenance, support) affect the project progress more seriously than technical problems. Since software engineering methodologies cover the entire development process, it was obvious to deal with software engineering methodologies in order to find solutions for these difficulties. Now, after having completed around 20 software projects, I feel able to critically evaluate software engineering methodologies and to assess their suitability for real-world projects.

1.1 Non-ionization EMF exposure

Extremely low frequency (ELF) electromagnetic fields (EMF) and radiofrequency (RF) fields are the two major sources of human non-ionization EMF exposure. A single photon's energy of non-ionization radiation does not suffice to break electron bounds.

The ELF dose of the public has remained more or less constant over past decades, whereas the RF dose has increased drastically during the last couple of years due to the exponentially increasing use of wireless communication systems. Constantly increasing microprocessor clock rates and the growing market of mobile telecommunication equipment and services have been the main driving forces of this trend. Customer groups and health agencies have been concerned about the exposure dose rise and have requested the government and manufacturers for scientifically based compliance assessment for this technology. In parallel to this trend, the question of ELF exposure has again become important.

The discipline of bioelectromagnetics copes with the interaction of EMF and biological tissue. The most relevant topic is the potential adverse health effects caused by EMF exposure. ELF and RF fields have different physical characteristics resulting in different biological interactions. Therefore, the two ranges are treated separately. In contrast to health effects caused by non-ionization EMF exposure, the effects from ionization radiation are well understood.

The International Commission of Non-Ionization Radiation Protection (ICNIRP) relies heavily on the findings of bioelectromagnetic research and publishes guidelines for limit-
1.2 ELF exposure

This range contains the frequencies up to 300 Hz. Generating and distributing electricity are the main technical application for ELF fields. Power lines and overhead traction wires emit these frequencies and are the sources for human ELF exposure. Electric fields outside the body induce a surface charge on the body, resulting in induced body currents. This is the only established interaction mechanism of ELF fields with the human body. The current density \( \text{A/m}^2 \) is the relevant dosimetric quantity. Various studies examining the cancer risk of people living close to power lines have not found a significant correlation. ICNIRP's basic restrictions intend to prevent established effects on the nervous system. Occupational and public exposure limits refer to fields inducing current densities of maximal 10 and 2 \( \text{A/m}^2 \), respectively. The corresponding maximum reverence levels for 50 Hz are E-fields of 20 kV/m (10 kV/m) and B-fields of 500 microT (100 microT).

1.3 RF exposure

This range covers the frequency band 100 kHz - 300 GHz. Most wireless communication systems, from a simple pager to large radiation beam systems, operate in the RF range. Today, in industrialized countries mobile phones and base station antennas are the major sources of human RF exposure. Biological tissues exposed to RF fields absorb energy. Therefore, the Specific Absorption Rate (SAR) is the relevant dosimetric quantity. SAR has the unit W/kg and defines the tissue-absorbed energy per unit mass and time. It can be expressed using the induced electric field strength or the temperature rise in tissue

\[
\text{SAR} = \frac{dP}{dm} = \frac{\sigma E^2}{\rho} = c \frac{dT}{dt},
\]

where \( E \) is the root-mean square value of the induced electric field strength, \( dT/\text{dt} \) the temperature rise, \( \rho \) the tissue density, \( \sigma \) the dielectric conductivity and \( c \) the specific heat capacity.

Numerous research laboratories have performed RF exposure studies addressing various potential health effects, the results of which have been contradictory. RF exposure haz-
ards can be divided into thermal and non-thermal effects. According to the formula above, temperature changes or heat added to a system cause thermal effects. In contrast, non-thermal effects occur at exposure levels neither challenging thermoregulation nor producing any significant change in body temperature. Interference problems of mobile phones with technical systems (e.g., pace makers) may also indirectly influence human health.

ICNIRP has standardized the SAR measurement procedure of tissues to obtain comparable results by defining the time averaging period to 6 minutes and the localized SAR averaging mass to 10 g of contiguous tissue.

A body temperature rise of more than 1°C produces established biological effects. This temperature increase results from a whole-body average SAR of 4 W/kg for about 30 minutes under moderate environmental conditions.

Consulting available research results, ICNIRP has defined exposure limits for three body parts for occupational and public exposure: (1) whole-body average SAR limit of 0.4 W/kg (0.08 W/kg) to avoid general thermal stress, (2) local head and trunk SAR limit of 10 W/kg (2 W/kg) to avoid local heating, (3) local limb SAR limit of 20 W/kg (4 W/kg).

Various research groups have performed animal, cell and human studies to establish non-thermal effects. The results of these studies have been extremely contradictory. Therefore, ICNIRP has not presented quantitative risk estimations for non-thermal effects.

1.4 EMF laboratory studies

EMF risk assessment involves field and laboratory studies. Field studies systematically evaluate the health data of people frequently exposed to EMF and compares them with statistical means. If an examined disease appears significantly more frequently to exposed people, EMF fields may activate or at least catalyze this disease. Having a surpassing exposure dose, people living close to power lines and workers in electrical occupations are prime test subjects. Such field studies focus mostly on the possibly enhanced cancer risk for these people. Linking EMF exposure to diseases, field studies may yield statistical trends, but they suffer from some disadvantages: (1) the mechanism of action remains unclear, (2) dose estimation is almost impossible or at least within a large range, resulting in an uncertain dose effect mechanism, (3) the focus is on certain diseases, ignoring more subtle effects. Thus, field studies provide qualitative results for selected diseases in real-world environments.

EMF laboratory studies on the other hand examine the effects of fields on biological tissues in controlled laboratory conditions and can be divided into in vitro, in vivo and human studies. Each study has advantages and disadvantages.

1.4.1 In vitro studies

Investigating the effects of EMF exposure on cellular and molecular levels as well as functional and structural changes in living cells, in vitro studies may illuminate basic mecha-
nisms of action. They are low-cost, allow fast experiments, require only standard laboratory facilities, can be replicated quite easily and serve as a basis for establishing dose effect mechanisms. Experiment parameters like dose, duration, signal scheme and investigated cells can be adapted quickly, enabling fast experiment evolution. In vitro studies rarely have ethical restrictions, and generally the dosimetry is accurate, since numerical modeling is generally straightforward and dosimetric quantities are measurable at almost all locations. However, in vitro studies have a serious drawback: Extrapolating scientific findings from the cellular level to human beings is almost impossible, limiting the relevance of these studies for human health.

1.4.2 In vivo studies

In vivo studies investigate EMF exposure on standardized animal species. The area of examination reaches from behavior studies to long-term hazards such as cancer studies. In Vivo studies benefit from well-established and proven toxicological procedures applied for various risk agents. They are expensive and require laboratories equipped for animal experiments. Dosimetric accuracy is at best average, since numerous parameters (e.g., animal weight and position) influence the dosimetry. Additionally, the relevant dosimetric quantities are not measurable in the living animal, and the exposure dose is mainly calculated by numerical simulations. In vivo studies underlie ethical restrictions, and performing replication experiments is difficult. The transferability of the results to humans is limited and relies on extrapolations. Planning and conducting an in vivo study is a complex task. Veterinarians, RF engineers, dosimetry experts and software engineers have to work closely together to achieve this goal.

1.4.3 Human studies

Human studies directly inspect the effects of EMF exposure on human beings. Some of the investigated biological issues in the past were electroencephalogram, sleep structure, cognitive functions, interactions with neurological diseases, evoked potentials, event related potentials and event related desynchronisation/synchronization [1]. Obviously, these studies are highly relevant to human health risk assessment, because their results are significant one-to-one and must not be extrapolated. Most countries apply tight ethical restrictions to human studies. Normally, the study protocol limits the number of test persons and sets upper boundaries for the applied exposure dose. Therefore, the statistical significance is poor, and only weak effects may be detected. Dosimetric quantities are not directly measurable in humans, but detailed human body numerical models enable relatively good dosimetric accuracy nonetheless. In contrast to test animals, test persons follow the tutors’ advice on how to position themselves relative to the emitter. These controlled exposure conditions allow good dosimetric predictions. The exposure setups used for human studies are generally relatively simple, and therefore the studies are reasonably inexpensive.

Laboratory studies supply quantitative results in artificial environments. Using different approaches, field and laboratory studies complement each other, both delivering important data for health agencies and governments to release scientifically based guidelines.
1.4.4 Limits of EMF laboratory studies

Numerous in vitro, in vivo and human studies addressing the possible negative health effects of EMF have been conducted in recent years. Various research groups have performed animal, cell and human studies with different scientific approaches. Several experiments have turned out to be of limited value due to severe shortcomings in the exposure setup. A major point of criticism is the lack of sophisticated controlling software of most setups, resulting in some important drawbacks. Firstly, biomedical teams with generally little technical knowledge have to manipulate RF equipment. Subsequent possible handling errors are not detectable, and the investigated items are exposed to an unpredictable radiation dose. A second point addresses the reconstruction of the experiments. Systems without controlling software are unable to collect and store data. Therefore, retrospectively asserting the accuracy of the experiments is not possible. Thirdly, simulating complex real-world signal types like GSM or UMTS is only possible with software. Finally, to obtain high quality results, biomedical studies have to be performed under blind conditions, i.e., the experimenter must not know whether an active field exposure or the sham exposure takes place. Therefore, unconscious (or even conscious) manipulation of the experiment is not possible. Fully double-blinded study design is only possible with controlling software implementing a random decision maker and enabling the encoding of exposure data.

1.4.5 Controlling software for EMF laboratory studies in this thesis

One part of this thesis covers the quality control improvement and significance of EMF laboratory studies. Analyzing the requirements of today's exposure setups reveals the importance of well-adapted controlling software. Controlling software fulfilling all requirements becomes complex. The complexity of large software projects is one of the main difficulties of modern software design. The subdivision of software packages into modular entities considerably reduces this complexity.

This thesis describes in detail each subdivision's purpose, responsibility, and potential pitfalls. The smooth interaction of subdivisions is crucial for fully functional and reliable software; however, assembling subdivisions is a delicate process. Unexpected compatibility problems can occur or timing troubles in real-time systems may happen. Software integration is a challenge in modern system design, and therefore the integration process is analyzed and discussed thoroughly.

Validating and testing large systems is an extremely complex task. Testing all possible scenarios is impossible within a reasonable time span. In many cases, even detecting all scenarios is impossible. Having a defined testing and validating strategy aids the determination of software bugs and makes the final system more reliable. One chapter outlines an applicable testing scheme useful for difficult environmental conditions such as high-purity laboratories and being under time pressure.

An exposure setup's scientific core is the transfer function, combining RF power and/or electromagnetic field strength with the specific absorption rate (SAR) of the investigated subject. Usually, a combination of measurements and simulations reveals this connec-
tion. Depending on the setup category (human, in vivo, in vitro), EM coupling mechanism and setup design, the transfer function consists of different parameters. A chapter extracts elements common to various transfer functions and suggests a generic function valid for all setup types. The generic approach enables the transfer function’s encapsulation and simplifies the implementation and exchangeability of power feedback algorithms. Moreover, maintaining a strict separation of data acquisition and data conversion enables the creation of powerful software design patterns occurring repeatedly in data acquisition and evaluation systems.

The second chapter of this thesis illustrates the requirements of modern controlling software for EMF laboratory studies.

The fourth chapter describes in detail the controlling software for the PERFORM A exposure system. PERFORM A is a large-scale in vivo project under Framework V of the European Union studying the long-term effects of electromagnetic exposure. This is the largest project I have realized and incorporates most aspects of the controlling software I have written.

The realized controlling software was applied in more than 20 different worldwide laboratory studies for cells, animals and humans. The controlling software enabled to perform animal studies under NTP-like GLP conditions. The derived mathematical GSM signal model and the implementation of this model became standard in BIOEM research. This signal model represents the first exposure scenario that applies temporal changes of different modulation schemes as occurring in a real-world mobile phone conversation. This signal contains a cocktail of low-frequency modulation components, which were neglected in the past.

1.5 Software engineering

Software engineering deals with the efficient realization of software systems. Since the software costs of a typical system are constantly increasing, efficiently building software is essential for the global economy. Software engineering addresses the functionality, the efficiency and the user-friendliness of a system as well as the economical handling of hardware resources. Additionally, software engineering deals with soft factors like project management, employee motivation and working efficiently in a team.

1.5.1 Software systems

Software systems come in all shapes and sizes. This ranges from systems processing e-mails, systems calculating payrolls, systems monitoring student grades, mobile phones and printing units to systems controlling air traffic or systems monitoring space missions. No matter how different these systems, they all have some features in common.

- Systems have end users. As the name assumes, end users apply the system’s functionality.
• Systems have functions (methods) and data. Data describe the system's state, whereas functions define how the system processes the data.

• Programmers create systems using programming languages. Depending on the system's nature and the requirements of the system, a programmer chooses the appropriate programming language. No programming language is inherently better than another. The context and the problems to solve define which programming language a programmer should choose. During the last decade, Object-Oriented Programming Languages (OOPL) have become more and more state-of-the-art. Software systems have evolved into large and complex structures. Offering good opportunities to handle complexity, OOPLs are today's most efficient programming languages for large systems.

• Systems contain hardware and software. On the one hand, systems contain hardware on which the software runs (CPU, memory, disk, i.e. computers). On the other hand, systems may also contain external hardware, which is controlled by the software (data acquisition devices, sensors, control units etc.).

• Software engineers design and implement a system using a software engineering methodology. At first glance, this sounds odd. Missing a structured proceeding, most software development seems to be a chaotic activity, consisting of the phases "coding" and "fixing bugs". However, this chaotic approach is completely inappropriate for larger projects. Over the course of time, various structured methodologies have appeared. Today, software engineering includes two sets of fundamentally different methodologies. One group is labeled heavyweight methodologies, whereas the other group is labeled lightweight or agile methodologies. Ironically, neither of them is very popular among programmers.

1.5.2 Software engineering methodologies

The average amount of source code within a software package has increased considerably in past years. Consequently, the size of a team realizing a typical software package increased as well. Concurrent to these changes, the discipline of software engineering has emerged, coping with the specification, development, management and evolution of large software systems. Recently, various software engineering methodologies have been presented to systematize the engineering process.

Software engineering has never been shy of introducing new methodologies. Over the last 30 years, very different approaches to software engineering have been proposed. However, most methodologies developed in academia have not survived until this day, and professional software engineers have never used these methodologies in the real world.

Managers of software engineering companies with little technical background have been consistently displeased about the unmethodical approach to software engineering used in industry. The ultimate goal of these managers has always been to develop a formal method for separating design and implementing processes. Given this separation, soft-
ware projects would no longer be dependent on the individual skills of programmers. Like other resources, programmers should become easily exchangeable. Up to now, this scenario is not achieved. Stable and robust software systems are still dependant on the individual skills of programmers.

**Heavyweight methodologies**

Traditional software engineering methodologies have focused on the separation of design and implementation and have tried to formulate a software development process following a relatively tight schedule. Thereby, software development should become predictive. Generally, predictability is a very desirable property. Predictive processes are easily measurable and controllable. For controllable processes, companies simply have to compare the actual project state with the target state to assess the overall project progress. This comparison actual-target state is one of the main occupations of business analysts. Therefore, they obviously prefer predictive processes to more chaotic ones.

Today, Object-Oriented Analysis and Design (OOAD) using the Unified Modeling Language (UML) is the most popular heavyweight methodology. OOAD applies object-oriented principles to the whole development lifecycle and uses a set of highly standardized diagrams to document the software system. At last, OOAD has seemed to offer the comprehensive software development methodology for which the world has long waited. Strongly focusing on planning, analyzing, designing and complete documentation, OOAD is a heavyweight methodology. Ideally, implementation starts only when the whole system is completely planned and specified. Civil or mechanical engineering disciplines have significantly inspired OOAD. Like other engineering disciplines, OOAD assumes the development process to be more or less predictive, allowing the separation of design and implementation activities. OOAD distinguishes four fundamental phases: planning, analysis, design and implementation. Each phase is well-defined and delivers output used as input for the following phase.

The planning phase evaluates why a system should be built. Before the project manager creates a work plan, selects the team staff and identifies appropriate management techniques, various feasibility studies are performed. As the name suggests, these studies assess the system's chance of realization. The analysis phase evaluates who will use the system, what the system will do and where it will be used. Gathering information is central in this phase. All information related in any way with the system is potentially useful and must be collected and bundled. The analysts weigh up the importance of all information and create a concept for the new system. The concept in turn serves as the basis for a set of UML analysis models. The design phase establishes how the system will operate. It defines architecture design, interface design, program design, and database and file specification. The design specifications should be comprehensive enough to enable straightforward implementation. In the implementation phase, the system is built, validated and tested as well as installed. In general, installation does not mark the end of a project. The development team must support the system and adapt it to current user needs.

UML is an abstract, programming language independent modeling language to specify, visualize, construct and document software. Today, UML is established in the industry as
a real standard for modeling object-oriented software systems. UML is not a methodology; however, it is best used together with a methodology. UML has an open standard and is under constant development. UML's most powerful features are a set of diagrams. These modeling diagrams are appropriate to describe a system in different levels of detail. Each diagram addresses a different aspect of the system and has a different scope. Some diagrams focus on the system-environment interaction, other diagrams describe the system's static structure and again other diagrams reveal the system's dynamic behavior at run-time.

After having worked with OOAD and UML in the practical world for some years, software engineers have realized that this methodology could not hold all promises. Some developers have argued that this methodology artificially complicates the development process and slows down development speed. The main points of criticism have addressed this methodology's document-centric approach, its focusing on formal processes, its slow reaction time to changing requirements as well as its neglect of people as an important factor in the development process. This negative attitude towards OOAD is addressed in various studies. According the field studies of Nandhakumar et al. [2] and Truex et al. [3], document-centric heavyweight software engineering methodologies like OOAD have never been really used in practice.

**Agile methodologies**

As a reaction, a group of new methodologies has appeared in the last few years: agile methodologies. The term "agile" expresses these methodologies' fast reaction time to changing conditions. Developed from professional software engineers for professional software engineers, agile methods are fast and simple. In 2001, a group of software developers published the Agile Software Development Manifesto. This manifesto's four key statements excellently express the core ideas. The first statement emphasizes the individual skills of programmers. Programmers should not only have technical skills, but also have social competence. Social competence is important to work efficiently in a team. The second statement stresses the central importance of working software. Working software releases, instead of written documents, should serve as primary milestones for measuring the project's progress. The third statement addresses a software company's relationship to its customers. Developers and clients should cooperate as closely as possible. However, this is only possible with new business models. Finally, the fourth statement addresses the agile methodologies' readiness for motion and changes. Quick reaction time is achieved through short release cycles. Customers obtain new releases in short intervals and can test new features quickly. The reactions of the customers in turn are incorporated in future releases. Thereby, software evolves which truly meets customer needs.

Today, Extreme Programming (XP) is the best-known and documented agile methodology. Covering all phases from planning to implementation, XP is a comprehensive methodology. The Crystal Family of methodologies, Lean Software Development and Pragmatic Programming are other popular methodologies. The Crystal Family is also a comprehensive methodology, whereas Lean Software Development focuses more on development strategy. Offering a set of guidelines, Pragmatic Programming is not a meth-
odology in the stricter sense of the word. However, Pragmatic Programming has strongly influenced the agile movement.

1.5.3 Software engineering in this thesis

The third chapter of this thesis introduces software engineering. The first part covers the fundamental concepts of object-oriented software engineering. It introduces the software development life cycle with its planning, analysis, design, and implementation phases, as well as the basic characteristics of object-oriented development. Additionally, it describes the UML's most important diagrams and outlines the characteristics of real-time software systems. The second part presents software engineering methodologies. These are distinguished in two categories: heavyweight and agile methodologies. In my opinion, presenting software engineering methodologies without adding individual comments of the software engineer is not a good approach, since developing software is a highly individual activity. Some programmers value one technique as extremely important, whereas other programmers find the same technique negligible. Simply, different aspects are important for different people. Therefore, one's experience is important. For this reason, I have commented on the methodologies and have used my own examples to clarify some statements.

Finally, the fifth and last chapter evaluates the presented software engineering methodology for practical use. Again, the evaluation is not completely objective. Different methodologies have different scopes. Therefore, the evaluation also shows which methodology is useful for which project type.

This thesis uses UML diagrams compatible with UML version 1.5. All presented source code snippets apply standard C++ syntax and use standard C++ data types like std::string and standard C++ container classes like std::vector<T> or std::map<Key, Type>. Source code is formatted using LetterGothic font.

For example, ClassName::MethodName() marks a method of a class, whereas someObject.MethodName() marks a method of a specific object. To enhance the code's readability, this thesis utilizes some basic coding standards:

- class names begin with a capital letter (SomeClass)
- method names begin with a capital letter (SomeClass::SomeMethod())
- a capital letter connects composed names for classes, methods, variables
- class member variables start with m_(m_MemberVariable)
- non-member variables start with a small letter (tempVariable)
- slanting style for open and closed brackets, where open brackets appear at the end of the line and closed brackets appear at the beginning of a new line.
Seite Leer / Blank leaf
2 EMF laboratory studies

This chapter discusses the importance of well-defined conditions for biological experiments addressing the health risk concern of EMF. Subsequently, it describes the design of exposure setups. The focus lies on the main difficulties when planning and constructing exposure setups fulfilling specific conditions. Based on the investigations, the chapter’s third part lists and discusses the core requirements for exposure setups.

Several authors have published detailed characterizations of specific exposure setups, e.g., [4], [5], [6]. Still other authors have extracted and evaluated the requirements for a specific setup category, e.g., [7], [8], [9]. Choosing a generic approach, this reflection does not distinguish between in vitro, in vivo and human setups. It integrates the established knowledge and proposes general guidelines. Considering the integration results, the relevant requirements can only be completely fulfilled using a computer-controlled exposure system. Therefore, controlling software is an indispensable part of modern exposure systems. Consequently, the discussion emphasizes the software part to fulfill the individual requirements and suggests basic implementation guidelines. Up to now, no exposure systems having complex controlling software with tight exposure and environmental control have been realized. Therefore, the realization and description of software-controlled exposure systems is a real innovation. Software-controlled exposure systems enhance considerably the credibility of the EMF experiments.

2.1 Exposure and evaluation system

A computer-controlled exposure and evaluation system consists of the following parts:

- exposure chambers. In the exposure chambers, the test subjects are exposed to electromagnetic fields. Examples for exposure chambers are waveguides, in which cell cultures are exposed to a standing electromagnetic wave and Ferris wheels, in which test animals are exposed to a rotation-symmetric electromagnetic field.

- hardware and sensors. The hardware mainly consists of a signal unit and a data acquisition device. The signal unit generates the electromagnetic field and feeds the exposure chambers. Field sensors and environmental sensors measure the field strength and environmental parameters such as temperature, humidity, oxygen concentration and the airflow in the exposure chambers. These sensors are connected to the data acquisition device. The data acquisition device collects the data from the sensors.

- controlling software. The controlling software manages the signal unit and thus regulates the field strength in the exposure chambers. Additionally, it collects the sensor data recorded by the data acquisition device. Depending on the measured
sensor data, the controlling software applies control commands. The controlling software stores all measurement data, the conversion parameters as well as the applied hardware commands in a log file. For each exposure, a new file is created. The controlling software is real-time software. Depending on the type of the exposure system, the controlling software can become very complex. Controlling software for small-sized human exposure systems typically collects data from 2-4 sensors and manages 1-2 control parameters. Software for large-sized *in vitro* exposure systems in contrast collects data from up to 250 sensors and manages up to 16 control parameters.

• evaluation software. The evaluation software analyzes the log files created by the controlling software. The log files are evaluated after the EMF experiments are finished. The data evaluation software runs on a different computer than the controlling software. It runs on a computer in the system designer facilities. Therefore, the log files must be transferred from the place of the data creation (bio-laboratory) to the place where the data are evaluated (system designer facilities). This transfer can be accomplished automatically (when the controlling software is connected to the internet and automatic data transfer is possible) or manually (operators send the log files manually by email). The evaluation software extracts the data of the log file, creates data charts and generates a detailed exposure report. This exposure report serves to verify retrospectively if the exposure was conducted according to the specifications.

![Complete exposure and evaluation system diagram](image)

**Figure 2.1:** Schematic description of a complete exposure and evaluation system. The real time exposure system (controlling software, hardware, sensors, exposure chambers) controls the EMF experiment. The controlling software stores the measurement data, the conversions parameters as well as the applied hardware commands in a log file. After an experiment is finished, the log file is transferred to the system designer facilities. There, the data are evaluated using the evaluation software.

The exposure chambers, the hardware (signal unit, data acquisition device, sensors) and the controlling software compose a real-time control system. This system is called expo-
2.2 Experiment conditions

Well-defined biological experiments are indispensable to provide relevant data for risk evaluation. Relevant EMF data quantify the subject’s exposure dose as well as the uncertainty of the exposure dose and estimate the influence of environmental parameters. Only well-characterized setup designs provide the necessary data relevance. These setups quantify the exposure and the uncertainty, minimize intersubject exposure variability, reduce artifacts or at least quantify them and generate the signal properties of the tested technology.

EMF exposure may cause negative effects on human health. If this hypothesis should come true, the impact on the global economy would be huge. Therefore, the results of biological experiments clearly suggesting negative health effects caused by EMF must be handled very carefully. Consequently, regulation agencies demand clear scientific evidence of negative health effects caused by EMF before releasing strict safety guidelines. Whenever an EMF experiment detects clear negative effects, regulation agencies normally ask different biological laboratories to conduct replication studies. Only if the results of the replication studies are coherent with the results of the initial study, might the guidelines be adapted. Therefore, sound exposure systems must be well characterized to enable replication studies. These systems must always provide the same exposure conditions, thereby allowing different research laboratories to crosscheck their results.

2.2.1 Exposure conditions

In the past, most research groups conducting EMF experiments have underestimated the difficulties of obtaining the exposure conditions (homogenous field distribution, well-quantified exposure dose etc.) imposed by the requirements of biological experiments. The realization of high-quality exposure setups requires profound knowledge in near-field measurement techniques, anatomy, tissue physiology and thermoregulation, dosimetry, material science, numerical simulation and software engineering.

Obviously, no single scientist is an expert in all of these disciplines. Therefore, excellent teamwork among scientists from these varied fields is crucial. However, different standards of knowledge or different sights can complicate efficient teamwork. A first step towards good cooperation is to establish a common vocabulary. All people involved in the project must speak the same language. Therefore, a glossary containing all the technical terms of the different disciplines is indispensable. This glossary must be accessible via internet.
Clearly, the importance of a good teamwork for the overall project success has been mis-calculated in the past. Competent project leaders should be aware of these potential pitfalls and must care for an efficient organization and a good cooperation in the team.

2.3 Design of exposure setups

Setup designers must master two main challenges: (1) identify and minimize the influence of environmental parameters and (2) generate homogenous and quantifiable exposure conditions.

 Depending on the setup category, diverse environmental parameters may influence the experiments. For example
- animals of in vivo experiments exposed in a fixed position should receive a constant airflow.
- the relative animal positions in the exposure chamber influence the exposure dose a single animal obtains.
- tissue heating masks the results of in vitro studies.
- electrodes attached to the heads of human volunteers in human studies cause local dose variations.

Various publications address the influence of these unwanted effects. Critics of studies finding negative effects often claim the effects to be artifacts of poorly considered environmental parameters. Therefore, knowing the parameters influencing an experiment is the first step for high-quality results. In the second step, setup designers have to minimize the effects of the identified parameters.

2.3.1 Dosimetry

The dosimetry allows the calculation of the dosimetric relevant quantity (usually SAR [W/Kg]) out of the characteristics of the electromagnetic field, the tissue parameters of the exposed subjects and the setup parameters

\[ SAR = f(EMF(f_s, f, f_c), TissueParameters(\varepsilon, \sigma), SetupParameters(sp)) \]

The mainly relevant electromagnetic field parameters are field strength \( f_s \), carrier frequency \( f \) and the low frequency components \( f_c \). The mainly relevant tissue parameters are the electric permittivity \( \varepsilon \) and the electric conductivity \( \sigma \). The setup parameters \( sp \) are dependent of the concrete setup type.

The dosimetry is the scientific core of an exposure setup. Generally, the dosimetric relevant quantity (usually SAR) cannot be measured directly in the exposed subject. Especially when humans and animals are exposed, the measuring of the dosimetric relevant
quantity in the humans or in the animals is not possible. Therefore, indirect methods are used to calculate the dosimetric relevant quantity. Two indirect methods exist to calculate the dosimetric relevant quantity. 1) Direct measurements in phantoms. A phantom is a physical model that has essentially the same electrical parameters as the exposed subject. In general, a phantom consists of a hollow structure filled with a liquid that has the same electrical properties as the exposed subjects. 2) Numerical simulations. Numerical simulations calculate the field distribution in numerical models of the exposed subject. The field distribution combined with the tissue parameters allows then the calculation of the dosimetric relevant quantity. Direct measurements as well as numerical simulations do not measure the actual dosimetric quantity in the exposed subjects. Consequently, both methods do not calculate per se the correct values. However, it is not evident which method provides values that are more accurate. Therefore, the goal is that both methods deliver the same values. Then the likelihood is high that the correct dosimetric value is obtained.

Having an accurate dosimetry is crucial for quantifiable exposure doses. For that reason, a large amount of the effort and money from laboratory studies is spent acquiring high-quality dosimetry.

Normally, setups expose several subjects at the same time in the same exposure chamber. This experiment design drastically reduces the overall experiment duration and costs. Every subject must receive the same exposure dose. This requires a homogenous EM field distribution in the exposure chamber.

However, generating a homogenous field distribution in a multi-subject exposure chamber is a complex task. The level of difficulty increases drastically with the number of simultaneously exposed subjects. As a result, a fundamental conflict between exposure setup costs and homogenous exposure conditions exists. Depending on the budget at hand and the requirements imposed by the biological experiment, biologists and engineers must come to a compromise. The goal is to find the cheapest setup design that fulfills all core requirements.

### 2.3.2 Design concessions

Each setup design allows individual concessions without severely degrading the quality of the setups. Experienced setup designers intuitively know the influence of various parameters and can make compromises that only slightly influence the overall setup performance.

A standard exposure system suitable for most experiments does not exist. A particular biological protocol requires selected exposure conditions and therefore an individual system. Nevertheless, basic setup designs for different setup categories exist.

### 2.3.3 Evolution of exposure systems

EMF laboratory studies have evolved over time. The requirements have become more and more extensive and tight. On one hand, the biological protocols have become more
comprehensive and biologists have wanted to investigate more subtle effects. On the other hand, knowledge about environmental parameters causing artifacts has increased considerably. Consequently, exposure setups fulfilling these requirements have become increasingly complex.

As one of the first bioelectromagnetic research groups, IT'IS has realized the need of comprehensive controlling software to fulfill biological protocols and has begun planning and implementing software-controlled systems. IT'IS had to develop the software from scratch and could no rely on established knowledge, since no similar software had been written or discussed in the literature. Additionally, the fact that every biological experiment follows unique protocols complicated the realization of the controlling software. Only custom-made exposure setups can fulfill the particular requirements the biological experiment imposes. Therefore, each setup is a product made to specification, obviously requiring adapted controlling software.

2.3.4 Controlling software

Since biological experiments (and thus exposure setups) are generally unique, programming a single controlling software which functions for all setup types is hardly possible. Therefore, the goal must be to implement flexible and extensible controlling software quickly adaptable to changing systems. However, programming flexible software is more complex than creating single-use software. In return, the amount of maintenance required for flexible software is considerably smaller than for numerous programs. Most software engineers are familiar with this dilemma of flexibility versus simplicity.

Initially, the ambition of IT'IS has been to create a single software design suitable for all exposure systems. Later, the insight of the virtual impossibility to fulfill this assignment in a reasonable time span emerged. Instead, a pragmatic approach was chosen. A single program skeleton has been created which has served as the basis for all specific programs. The temporal conditions for the software engineering have heavily pressured his approach. Over the course of time, planning and implementing the controlling software has turned out to be a considerable challenge requiring profound software engineering knowledge.

Controlling software implements the results of the engineering and research efforts. Therefore, programmers of software for exposure setups need a good knowledge of the exposure system and detailed information about the experiment procedures. The programmers are system integrators combining and integrating the specialists' knowledge. Additionally, they deal with special duties like real-time programming and automatic control engineering.

Given their awareness of the technical feasibility, setup programmers have to participate in the planning stage of biological experiments and contribute their points of view. In the past, setup programmers have often not participated in experiment planning. They simply obtained a set of tasks the controlling software has to complete. Study directors coordinating the work of the engineers and biologists did not notice the importance of the controlling software in the past. Additionally, the programmers often did not want to
participate at planning meetings. The change of the attitudes of the coordinators as well as the programmers is a prerequisite to improve the controlling software and thus the whole experiment in the future.

2.4 Requirements

The following chapters discuss in detail the individual requirements for exposure setups. They strongly focus on the software's part to fulfill these requirements.

2.4.1 Exposure uniformity

This is a crucial requirement for systems exposing several subjects simultaneously in the same chamber. Non-uniform field distributions result in different induced dosimetric quantities. The effects caused by the exposure are thus not objectively comparable.

In the past, most exposure systems have measured the input EMF power of an exposure system and used this power to calculate the field strength inside the exposure chamber. The field power in turn has allowed basic dosimetric calculations. However, this indirect measurement method has a serious drawback. The input power does not guarantee the desired field strength in the exposure chamber. Badly tuned chambers might reflect large part of the incoming power and weaken the field in the chamber. A common turnaround has been to measure the reflected power as well. Nevertheless, this also does not guarantee the desired field strength. The relative subject size and position might result in non-uniform field distributions, which cannot be observed from outside the exposure chamber.

Therefore, systems which measure the actual field strength close to the subjects are more reliable. However, the field sensors themselves must not influence the field. Large multi-subject exposure chambers like Ferris wheels frequently used for *in vivo* studies should measure the field at least at two different locations and thereby permit uniformity verification. Controlling software must periodically read the sensor data and calculate the differences. Comparing the difference with reference values, the exposure system decides whether the uniformity is achieved. Given only one source feeding the exposure chamber, the exposure system cannot actively influence the homogeneity; however, it can survey the homogeneity and guarantee the exposure conditions for all test subjects.

2.4.2 Exposure strength

Precise dosimetry is the heart of well-defined exposure systems. Given a certain field strength inside an exposure chamber, the dosimetry must predict the induced dosimetric quantity in the test subject as accurately as possible.

Measurements and numerical simulations calculate the dosimetry before the experiment actually starts. In the running experiment, technical influences such as amplifier drift and cable quality loss may generate field strength variations without changing the amplifier's output power. Consequently, assuring permanent field strength and exposure dose
requires periodic power feedback control, which is only possible with a software-controlled exposure system. The system collects the sensor data, compares them to reference values and adjusts the power according to the difference.

2.4.3 Signals

Exposure systems must generate the same signal properties as the investigated technology. If, for example, the effects of mobile phones on human health are examined, the exposure system’s signal unit must generate a realistic GSM signal.

However, generating a synthetic GSM signal is complex. The signal shape changes depending on talking (GSM Basic) and listening (GSM DTX) mode, and the power adapts according to the connection quality (power control). A realistic synthetic GSM signal must contain these features. Further it must contain the features yielding low frequency signal components, since these components are considered to cause health effects.

Initially, an accurate mathematical model describing the signal properties must be compiled. Software in turn implements the model, applies the appropriate commands to the signal unit and thus generates the desired signal. The software has to be flexible and allow the generation of different signal schemes.

2.4.4 Good exposure - sham isolation

Biological EMF experiments want to examine the influence of EMF fields only. Therefore, exposure systems must consist of exposure and sham chamber(s). The conditions in the chambers should be identical with the exception of the field strength. The field inside the sham chambers should be as low as possible. Therefore, different effects on subjects in exposure and sham chambers are likely the result of EMF exposure.

The controlling software has to measure continuously the field in the exposure and sham chambers. Thereby, the controlling software can detect unusually high field strengths in sham chambers and respond appropriately. Especially for studies looking for effects of low power fields, good exposure sham isolation is vital.

2.4.5 Double blind study design

Before an experiment is finished, study experimenters and test subjects must not know which subject has obtained real or sham exposure. Studies in which neither the experimenter nor the test subject know the experimental conditions are called double blind studies. The term double blind is somewhat misleading for in vivo and in vitro studies; the term blind study would be more appropriate, because animals and cell cultures cannot distinguish cognitively between sham and real exposure. Animals of course may feel the exposure. However, the experiment is not influenced if they know in advance (should that be possible) whether they will be exposed or not.

For experiments exposing test subjects to changing dose levels (for example real exposure and sham exposure), the assignment exposure chamber – dose level must be kept secret
until the end of the study. This study design prevents two effects: (1) Test subjects knowing they will be exposed may show psychosomatic effects; effects that are not caused actually by the exposure but by the sheer knowledge of the exposure. Psychosomatic effects may in fact play an important role in human studies. Exposed and sham exposed persons may show similar effects, whereas reference persons may not show any of these effects at all. Therefore, controlling software of “classical” exposure systems consisting of two chambers (exposure and sham) must switch the power randomly. As a result, the test subjects do not know if they are exposed or not. (2) Study experimenters may favor, consciously or unconsciously, a certain experiment result. This may influence the analysis of the experiment. The chance to publish a study demonstrating negative effects of EMF exposure to human health in a renowned journal like *Nature* is a good deal larger than that of a study showing no effects. Hence, the chance to publish in a renewed journal may influence the work of the experimenter.

### 2.4.6 Intermittent exposure protocols

Some authors, e.g., [4], have suggested that constantly switching the exposure on and off could induce effects. With controlling software that allows flexible on and off exposure time it is possible to test this hypothesis.

### 2.4.7 Resonance structure

Operating at resonance frequency, some exposure chambers achieve strong electromagnetic fields with moderate amplifier power. Depending of the loading volume or animal size, the resonance frequency slightly shifts. Yet small deviations from the resonance frequency weaken the field strength considerably. Therefore, before the exposure starts, controlling software has to perform a frequency scan and detect the exact resonance frequency.

### 2.4.8 Human safety

An exposure system must provide maximal safety for the people operating the system (The safety of the human test subjects is a different matter and mainly relies on ethical restrictions.). Protecting human study supervisors, molecular biologists examining cell structures, and especially animal keeper loading and unloading the animals into the exposure chambers from unintended EMF exposure is essential.

Large-scale *in vivo* studies usually take place in high purity rooms. Animal keepers should only enter the room when the exposure is off. When loading or unloading the animals, they must reach into the exposure chamber's field maxima. The field strength at this spot can lead to high local SAR lying way above the security limits for the hands.

Controlling software caring for human safety should indicate optically and/or visually the experiment state. Particularly, it should indicate the time when safe manipulation of the exposure chambers is possible. Software should only indicate the most important information and avoid fancy human interfaces.
Electromagnetic fields and their danger cannot be seen. Therefore, system designers knowing the danger of strong electromagnetic fields must carefully advise the system operators what to do at a certain experiment state. Standard Operation Protocols (SOPs) are a good possibility to document the handling of an exposure system.

2.4.9 Data monitoring

Dependable exposure systems monitor all relevant technical and environmental parameters. Technical parameters provide information about the exposure setup's condition. Controlling software continuously evaluating technical parameters assures smoothly running systems. Estimating the influence of any detected irregularities, software can respond appropriately.

Environmental parameters provide information about parametric quantities influencing the condition of the exposed subjects. Sensors placed as close as possible to the subjects without influencing the field distribution measure the relevant parameters. Software should collect and evaluate the data according to the specifications, thereby ensuring stable exposure conditions.

2.4.10 Data logging and security

Controlling software has to log measuring data, meta data (data describing how to interpret the measuring data), executed hardware commands and event data.

Some key data such as induced SAR cannot be measured directly. Instead, the controlling software dynamically converts the measuring data into the desired quantity. Software should not log the converted values directly. Instead, it should log the initial measuring values and the conversion parameters (or the meta data), because this guarantees flexible error tracing. Converted values may fall short of expectations for two reasons: the value may indeed be out of specifications or the conversion parameters may be wrong. Having the initial sensor values and the conversion factors, precise and sensitive data analysis is possible. In the case of a system failure, error tracing is very important and simpler when original data are available. Completely abnormal sensor values, for example, may indicate a broken sensor.

Most controlling software are multithreading applications. Several commands must be executed at virtually the same time. However, testing and debugging real-time multithreading applications is tedious. Setting breakpoints at any desired location in the source code is not possible without influencing the temporal program behavior. The combination of recording all hardware commands and analyzing the data may sometimes be the only efficient debugging strategy.

Event data mark the time at which special actions occur. User inputs or dynamic signal modification are typical event data.

With this complete data set (measuring data, meta data, hardware commands and event data) at hand, the reconstruction of every experiment is possible. Obviously, data files
recording these data become large, and manually analyzing the content is no longer possible. Therefore, data evaluation software is needed for proper analysis. This software should summarize the important parameters, calculate statistics, display graphics and automatically generate reports. Especially important are the evaluations of the generated synthetic signals. Complex real-world signals such as GSM signals cannot be created directly. Instead, first a mathematical model that describes the signal must be generated. Afterwards, the controlling software must simulate the signal according to the mathematical model. Comparing the applied synthetic signal statistics with the key parameters of the mathematical signal model, the conclusion of whether the signal was generated according to specifications becomes possible.

Evaluation software must reverse engineer some data. Only a combination of measuring data and meta data reveals some important data. At first glance, data reconstruction out of original data may seem dispensable. Of course, the controlling software could log only the converted data. Having already converted values in the log-file, the evaluation software could directly use them, and the program would become much simpler. However, when both programs (controlling software and evaluation software) calculate important data independently, the comparison of the results is possible. Thereby both programs verify each other and deviations indicate errors in either program. This concept of strict data separation has turned out to be a very powerful test and validation strategy.

The EMF experiment data must be decrypted. The decryption prevents the manipulation of the data by the system operators. Therefore, the controlling software has the decrypt the data. Biological laboratories must not be allowed to decrypt the data themselves. They should send the data to the setup supporting team for decryption. The supporting team decrypts and evaluates the data and sends an experiment report back to the biological laboratory.

2.4.11 Easy handling for non-engineering personnel

Biomedical teams with generally little technical knowledge regarding EMF measurements should not have to manipulate RF equipment. Subsequent possible handling errors are not detectable, and the investigated items could be exposed to an unpredictable radiation dose. Software should control all electronic devices and offer the operator a simple interface. It should display only the relevant information.

2.4.12 Self-detection of malfunction

As mentioned above, biomedical teams rarely have detailed knowledge about exposure systems. Whenever the system fails to work properly, they cannot perform on-site error tracing and analysis, not to mention fixing complicated defects.

Controlling software able to self-detect malfunction assesses the level of the problem and undertakes appropriate measures. Some problems can be bypassed without interventions. For example, systems having exposure chambers equipped with more than one field sensor are not dependent on all sensors working properly. Smart software detects broken field sensors, ignores them and uses the intact sensors to calculate the field strength. For
more severe problems, the software preferably informs the user how to fix a defect than to display a detailed error descriptions.

Depending on the laboratory studies, controlling software implements different error scenarios. Biologists conducting in vitro studies first expose, then evaluate and afterwards dispose of the cell cultures. Normally, they do not use the same cell cultures for more than one experiment. Typical exposure duration is between hours and several days, and the cost of the cell cultures is comparatively small. Therefore, the cost for a single in vitro exposure is quite small. This in turn influences the controlling software's error tolerance and field-power feedback algorithms.

Generally, power feedback algorithms suffer from a trade-off between reaction time and stability: the faster the reaction time, the lower the stability. For in vitro experiments, the error tolerance is small and the feedback algorithms react quickly. Therefore, in vitro experiments are aborted whenever slight anomalies are detected.

Long-term in vivo cancer studies in contrast need a more tolerant error handling strategy and more stable field-power feedback algorithms. The system only aborts experiments when it detects large anomalies. Cancer study test animals are exposed daily for months or even years. Thereby, they are very valuable towards the end of the study. Therefore, in vivo system failures occurring while animals are in the exposure chambers must be handled very carefully in order to preserve the animals' health. Before checking the system and bypassing defects, the software must maintain the climate regulation functionality and advise the operator to unload the animals immediately.

2.4.13 Highest possible system stability

System stability and reliability is often taken for granted; nevertheless, it is a core requirement of an exposure system. The overall experiment duration influences the quality and reliability of the system components used and sets the norms for the controlling software. Software for long-term studies must be very stable and robust. A next step to increase the system stability used for long-term studies is to establish a parallel system that supervises the real system.

2.5 Summary

Mature controlling software considerably improves the quality and significance of EMF laboratory studies. It guarantees well-defined and stable exposure conditions and provides original experiment data. These data allow the complete reconstruction of every experiment. Study founders normally demand assurance about the study's quality. However, they cannot supervise the study and must rely on the study conductor's conclusions. Having access to all original experiment data, they can reconstruct experiments and verify whether the experiments were conducted within the specifications. This data controlling improves the study's credibility.
3 Software engineering methodology

Following the classical system development life cycle (SDLC), the first section of this chapter outlines the fundamental concepts of object-oriented software engineering (OOSE). In parallel, today's most important modeling language, the unified modeling language (UML), is introduced. The second section summarizes the comprehending object-oriented (OO) development methodologies which have recently appeared in the literature.

This chapter serves as the basis for the two remaining chapters, which describe in detail the controlling software of a large in vitro exposure system and evaluate the benefits of both established and novel OO software engineering methodologies.

3.1 The system development life cycle

SDLC describes the development process of a software product and consists of a set of four fundamental phases [10]: planning, analysis, design and implementation. Originally used for traditional functional system design, SDLC has been transformed for OO system development. Each phase delivers output used as input for the following phase. The phases are well defined, and they recur cyclically, resulting in incremental work progress. The individual increments gradually refine the system. Each phase refines and elaborates the work done previously. Consequently, system development becomes an evolutionary process.

3.1.1 Planning

The planning phase evaluates why a system should be built. In the commercial world, a new system must provide value to the organization, whereas in the research world a new system must address new scientific questions or considerably refine and improve existing experiments. Before the actual development process starts, development teams must conduct a feasibility analysis. A system might not be feasible for many reasons: too expensive, technology constraints, lack of experience to create the system or organization structure too small for system size. Once analysts approve the system's feasibility, the project manager creates a work plan, staffs the team and selects appropriate management techniques. I have little experience planning a project, because I have never actively participated in the planning phase. Therefore, this thesis does not cover the planning phase more deeply.
3.1.2 Analysis

The OO system development process names the analysis phase object-oriented analysis (OOA). The fundamental outputs of this phase are: 1) who will use the system, 2) what the system will do and 3) where it will be used. Conform to other authors [11] [12], I think the analysis phase should already be user centric. The user's needs and wishes must be the driving forces throughout the whole development process.

This phase's first and most important step is gathering information. Various strategies for gathering information exist, e.g., analysis of existing systems, end-user interviews, creation and evaluation of questionnaires, working together with end-users and joint application design (JAD) [13]. Some proceedings are highly formal and structuralized, whereas others are more diffuse and rely deeply on the analyst's personal skills and intuition. Generally, formal methods alone do not provide all the essential information to realize a large system. The last chapter demonstrates why an analyst's social-emotional skills are particularly important to collect knowledge floating in the project team and to extract real user needs.

Evaluating all information enables the creation of the new system's concept. This concept afterwards serves as the basis for a set of UML analysis models. The different models explain different aspects of the systems and have different levels of abstraction. Use-case models describe interactions with the system's environment and with other systems. Structural models (UML: package diagrams, class responsibility cards (CRC) and class diagrams) describe the system's static data structure. Behavioral models (UML: sequence diagrams, activity diagrams, state chart diagrams) describe the system's internal chronological or dynamic activities.

3.1.3 Design

The OO system development process names the design phase object-oriented design (OOD). Describing how the system will operate, the OOD delivers information about architectural design, interface design, program design, and database and file specification. The architectural design describes the hardware, software and networks used for the systems. Interface design specifies how the user will use the system. Program design defines each program's purpose in the system. Large systems are composed of several independent programs. Frictionless interaction of these programs is essential for stable systems. The database and file specification eventually describes how and where the system will store the relevant data. In theory, all design information should enable smooth implementation of the system. However, mostly some major problems only first appear during the implementation process. In this case, the programmer's experience and creativity are necessary to complete the system.

3.1.4 Implementation

The implementation phase terminates the SDLC. In this phase, the system is actually built, validated, tested and installed. The installation does not mark the final activity.
The development team must support the running system in the case of failure and possibly implement new user needs. It must guarantee unobstructed running of the system.

### 3.2 Evolution of software engineering methodologies

Today's modern society depends completely on computer-based systems. Software plays an important role in virtually all realms of our life. Systems with very different scales contain software. The range reaches from large-scale power systems to home toasters. More and more products embed computers and hence software and the software's part of the overall system costs is constantly increasing. Therefore, efficiently building software is vital for global economy.

Over the course of time, software systems have become more and more complex. Various reasons have led to this trend, including:

- increased microprocessor performance and more memory allowing extensive codes to run faster
- natural evolution of a new technology (Software engineering is a young science)
- application of user-friendly graphical interfaces
- development of high level programming languages
- improvement of integrated development environments enabling simple project handling and effective code debugging
- development of robust software deployment frameworks
- availability of binary software libraries for various purposes
- increasing competition and the appearance of the open source software movement.

Software is abstract and intangible, and consequently not bound to physical laws and not constrained by materials. Incorporating these properties, software is extremely flexible, allowing almost infinite solutions to a single problem. The intangibility and flexibility is both a blessing and a curse. On the one hand, the flexibility simplifies the engineering process and allows fast modifications. On the other hand, lacking natural constraints, software can become very complex and hard to control. Carelessly built large software can become very difficult to understand and can even become not maintainable.

Large software packages are considerably more difficult to plan, manage and implement than small ones. Thus, various system developers have tried to formalize the development process for large software packages. These first formalization attempts mark the hour of birth of the discipline software engineering (SE). Over time, as people have learned more about developing software systems, numerous system development strategies have been developed.
In the early days of computing, programmers moved directly from limited system comprehension into writing code. This approach is completely reasonable for small programs, but may fail for large software systems. Requirements may change late in the implementation process, with disastrous results for badly planned programs. In some cases, when the software is very inflexible, the whole program must be rewritten from scratch. This just-jump-into-writing-code approach has another severe drawback: Long-term planning of a software system becomes almost impossible. However, planning is necessary if more than one programmer works on a system. Consequently, the just-jump-into-writing-code approach makes good teamwork unfeasible.

Emerging in the early 1980s, structured design was the first comprehensive category of software engineering methodology. This method's key feature is waterfall development: analysts and users sequentially move from one phase to the next, using the output of one phase as the input for the next. The waterfall approach is a one-way development strategy. Once moved into the next phase, going backwards to the previous phase is not possible any more. Structured design has introduced two sets of diagrams: one for process and one data modeling. Software engineers have debated for a long time about which diagram should play the dominant role.

The rapid application development (RAD) methodology emerged in the 1990s. Focusing on quick development, RAD quickly hands out parts of the system to the users. They test the prototype of the system and give feedback. Suggesting changes and enhancements early in the development process, the users bring the system closer to their actual needs. Yet the users' roles are somewhat passive. Their real needs are not analyzed, and the realized system may not necessarily be the best one for them. Classical RAD still separates processes and data.

3.3 Object-oriented analysis and design (OOAD)

The final years of the last century saw the triumphal procession of object-oriented programming languages (OOPL). Actually, the first OOPL were developed back in the sixties. Nevertheless, the procedural programming language C was the lingua franca for large systems for a long time. Brian Stroustrup added object-oriented (OO) features to C and created the OOPL C++ [14]. The large C programming community incorporated the new features and leveraged C++ as the first widely used OOP. In 1995, Sun Microsystems released the platform independent OOPL Java. Java's script version has become very popular among internet programmers. The rapidly growing internet and the concept of platform independency have spectacularly spread Java's popularity. Nowadays, the majority of software source code is object-oriented.

Based on excellent experience with OOPL, developers have begun to adopt OO principles in earlier phases of the software engineering process. The resulting OO software engineering methodology is therefore a logical enlargement of OOPLs. Today, OO is by far the most popular software engineering methodology used for any kind of systems, from web-based to software-embedded systems. The principles of object-orientation are the fundamentals for a deep understanding of OO systems. The next chapter outlines these principles and adds comments and hints from my own experience.
3.3 Object-oriented analysis and design (OOAD)

3.3.1 Basic characteristics of object-oriented development

The central object-oriented paradigm is the object. Objects contain data and methods. Data describe the object's state, whereas methods define the object's behavior. In OO systems, everything is organized around objects. Objects have an own life cycle and are responsible for themselves. This responsibility describes the object's behavior. Seen conceptually, an object is an entity with responsibility. This definition focuses on what the objects are supposed to do. Thinking in terms of responsibilities facilitates object selection and definition. Additionally, this aids defining the object's public interface. Each responsibility roughly maps to a public method. Finding proper object definitions and designing the public interfaces is the main point of good design.

Objects can communicate with each other. For that purpose, an object sends a message to another object. The receiver executes the requested action and returns the result to the caller. From the perspective of implementation, a message conforms to a method call from one object to another.

A class is the template used to define objects. A class describes the object's data elements, the methods an object can execute and the way the data and elements can be accessed. Program code is organized around classes. The containing data elements of an object can vary; therefore, objects of the same type may have different data but will still have the same functionality. Objects are identical when they are of the same type and contain exactly the same data.

Creating an object from a class template is called instantiation. Objects are instances of a class or the concrete manifestations of an abstract construction plan (= the class).

3.3.2 Encapsulation

Encapsulation is probably the most powerful OO concept. Encapsulation implies more than just bundle and hide data. Encapsulation also denotes information shielding. An entity receives only minimal information from another to perform its task. Shalloway et al. [15] lists four levels of encapsulation:

- encapsulation of data. Simple data types describing the object's state are not visible to a client.
- encapsulation of methods. Polymorph method calls (see later): the caller does not know which method will be executed.
- encapsulation of subdasses. In an inheritance hierarchy (see later), clients do not see the concrete type.
- encapsulation of other objects. Objects may contain other objects and delegate tasks to them.

However, why is this closefisted information management so crucial? The less an entity knows of another, the less it must care and the less it can destroy. Entities can rely on
each other without knowing their internal activities. Entities are therefore similar to black boxes, where a certain input generates a corresponding output. Thereby, the user of a black box does not know what actually happens inside the box. Only the result is relevant. Internal changes of an encapsulated entity do not affect the outside system. This enables better system management, because the outside system does not notice any changes at all.

Handling all information properly and mastering the resulting complexity is the most complicated task of large software systems. Good encapsulation decreases the information needed to perform an action and reduces complexity. Shalloway et al. [15] has stated “Find what varies and encapsulate it”. This sentence describes the heart of encapsulation very accurately. For example, consider the controlling software of the PERFORM A project. There, the RF generator generating the RF signal may change from one system to the other. The software encapsulates all RF generator specific functionality in the base class RfGenerator. Concrete RF generator classes are derived from RfGenerator. This enables flexible implementations of new RF generators.

3.3.3 Inheritance

A class (derived class) can be a specialization of another class (base class). The derived class inherits the data and methods of the base class.

Inheritance simplifies defining classes. Data and methods common in classes are placed in a base class. The specialized classes will inherit the common data and methods from the base class.

Therefore, inheritance prevents duplicating data and behavior and helps accomplishing the “once and only once rule”, stated by Shalloway et al. [15]. Implementing one idea (or program code) at only one place is the heart of this rule. Following this rule eliminates code duplication. Duplication is bad not only because of the extra work typing the same code multiple times, but because of the likelihood of something changing in the future and then forgetting to change it in all of the required places.

However, inheritance denotes more than just avoiding duplication. Inheritance expresses a fundamental relationship between the derived and the base class, called an “is a” relationship. The derived class is essentially the base class plus some extra features. Whenever a system part expects a base class, the system also accepts a derived class of the base class, because the derived class actually is a base class. Inheritance simplifies the logical grouping of classes and makes designs more readable.

3.3.4 Polymorphism

Polymorphism literally stands for the ability to take several forms. Object-oriented systems support polymorphism. A message sent to a set of different objects may be interpreted differently by different classes of objects. Mostly, programmers use polymorphism in the context of class inheritance.
Consider the base class $B$ and the derived classes $D_1$ and $D_2$. $B$ defines and implements the method $B::Oo()$. $D_1$ and $D_2$ inherit the method $Oo()$ and both overwrite it. $Oo()$ receives different semantics in $D_1$ and $D_2$. Activating $Oo()$ in an object of $D_1$ and $D_2$ results in different object behavior.

Dynamic binding refers to this ability of OOPL to defer the object's data type at program run time. Depending on the objects' type, given the same message (method call), objects interpret the same message differently. Having this ability, individual classes are easier to understand.

However, polymorphism realized through dynamic binding is a double-edged sword. Using dynamic binding, the system knows only at run time which specific object will interpret a certain message and makes a decision not specifically coded anywhere.

Polymorphism leverages the system's abstraction level and flexibility but complicates the tracking of the system's program flow. Additionally, highly polymorph systems are more difficult to debug. The programmer does not know in advance which object's method will be called. Consequently, he cannot set breakpoints before the debug session starts and often has to step through several method calls to find the desired one.

### 3.4 OOAD design principles

Object-oriented methodologies strongly try to map the real world into a set of corresponding models. This improves the system's understandability and maintainability and is one of the strengths of this methodology. The real world does not separate data and process. For example, consider a beetle. It has six legs and has the ability to crawl and fly. Its attributes combined with its behavior completely describe the beetle. Like the real world, OO methodologies do not separate data and methods but combine them in the shape of an object. Following the example of the beetle, a beetle's object model would contain an integer data type describing the number of legs as well as the methods `Crawl()` and `Fly()`.

Encapsulating data and processes, objects are potentially reusable components. New systems can use components developed for old systems, resulting in reduced programming and validation efforts and hence reduced costs.

Three proceedings characterize OOAD development: Use-case driven, architecture centric as well as iterative and incremental. Good OOAD development includes all three proceedings equally.

#### 3.4.1 Use-case driven

Use-cases are the primary modeling tool to define the system's behavior with the environment or with other systems. Generally, the environment comprises everything outside the system, but mostly refers to a user of the system. A use-case is simply an abstraction of a system's reaction to special events. Understanding the system-to-environment and system-to-system relationships helps to provide the required system functionality.
Additionally, knowing these relationships helps to establish an effective communication structure between the system and its environment. Focusing on only one activity at a time, use-cases are simple and communicate the high-level functions of the system as well as the system's scope. In standard OOAD, the system's details are derived from the system's use-cases. Use cases serve as the building block for the whole system. Detailed models refine the initial use-cases. This approach may overemphasize the relevance of use-cases. It fails to document properly complicated system interactions. The absent information will then be missing in the detailed models as well.

3.4.2 Architecture centric

Large systems are composed of several sub-systems. The system architecture describes these sub-systems and their interactions with each other. The architectural design identifies and establishes a control and communication framework for the sub-systems. The smooth interaction of sub-systems is crucial for fully functional and reliable systems; however assembling sub-systems is a delicate process. Unexpected compatibility problems can occur or timing troubles in real time systems may happen. Therefore, software integration is a challenge in modern system design. Especially, system designers have realized the complexity of integrating existing stand-alone systems into a larger system.

The system architecture establishes a basic framework and defines clear interfaces. The individual sub-systems' designs are orientated toward this architecture. Following a global design strategy, the sub-systems become compatible and therefore can communicate effectively. Focusing on the system architecture and defining clear interfaces before the sub-systems are designed circumvents a complex integration process.

Bass et al. [16] discuss three advantages to designing and documenting software architecture:

- stakeholder communication. The architecture is a high-level and therefore simple presentation of the system. This simplicity allows a broad range of stakeholders to obtain a general plan of the system. Possible vulnerabilities at different levels may be identified and solved in an early project state.

- system analysis. Defining the system architecture early in the project state enables further analysis. A particular architectural design influences critical system parameters like performance and reliability. After having selected a particular architectural design, assessing the critical system parameters is possible. When the investigated architecture does not meet all critical system requirements, then a new architecture must be found. Analysis out of the system architecture can be completed early in the project state. Therefore, possible shortcomings are detected before much effort and money is put in the project.

- large-scale reuse. Successful architectural designs may be transferred across systems with similar requirements. If two systems have similar architectural designs, the copying of independent sub-systems from one system to another becomes possible.
Architecture centric implies that the underlying architecture of the evolving system drives its specification, construction and documentation. OOAD supports this approach and offers several architectural views of a system. The functional view describes the system's behavior from a user's perspective. Using classes, relationships and messages, the static view describes the system's structure or static model. The dynamic view reveals the system's behavior at run-time. It describes the program flow in terms of messages passed between objects.

3.4.3 Iterative and incremental

OOAO is iterative and incremental. The SDLC's phases planning, analysis, design and implementation are not completed sequentially. In fact, cycling through the phases is typical for OOAD. Actions performed in one phase influence the progressions in the other phases. Iterative and incremental development undergoes continuous testing and refinement throughout the life of the project. The iterations bring the system closer and closer to the real needs of the users.

System developers have realized that completely planning and analyzing a system is not possible. Some details appear only in the implementation phase. This iterative approach considers the implementation not simply as an operating phase, but emphasizes its importance for the whole development process.

3.5 The unified modeling language (UML)

UML is an abstract language to specify, visualize, construct and document software. Originally compiled by the three modeling gurus Booch, Jacobsen and Rumbaugh, UML has become a real industry standard for modeling object-oriented software systems. Its popularity is such that today most software engineering job offers require profound UML knowledge.

UML is programming language independent. This fact has boosted UML's acceptance as a standard modeling language. UML is simply a modeling language and not a methodology; however, it is best used together with a methodology. Using UML diagrams together with a methodology increases the understanding of an application under development. UML is an open standard and is constantly enhanced. Its current version (November 2004) is 1.5; major revisions will be incorporated in version 2.0, which will be published in the beginning of 2005.

Although UML is very popular in the industry, universities have difficulties teaching and using it. Some university professors favor use of their own notation techniques and do not want to abandon them.

Every modeling technique has advantages and disadvantages. None of them is perfect. Perhaps there exists an even more powerful modeling technique than the UML. However, UML is standardized and state-of-the-art, and that is its main advantage over other modeling techniques. Having a standardized modeling language significantly simplifies the circulation of new ideas. People familiar with UML can quickly understand the high-
level architecture of complex systems without first learning a specific modeling technique.

If education and industry standards diverge, no professional modeling technique can emerge. Therefore, software modeling profits enormous if a unified modeling language is used. Since UML is an open and evolving standard, people with good ideas may formulate their contributions to make UML even more powerful.

UML's most powerful features are a set of six main diagrams: use-case, component, deployment, class, sequence, statechart and activity diagram. The UML includes more than these diagrams, but the diagrams offer a good tool for the understanding of the system design presented in the next chapter of this thesis.

Each diagram addresses a different aspect of the system and has a different level of detail. Some diagrams focus on the system-environment interaction, other diagrams describe the system's static structure and again other diagrams reveal the system's dynamic behavior at run-time. Therefore, UML covers the three fundamental views (functional, static, dynamic) of an architectural system design.

3.5.1 Use-case diagram

A use-case illustrates a functionality the system provides and reveals the system's functional requirements. Actors mark human beings or other systems interacting with the initial system.

Use-cases as well as actors can have relationships with other actors and use-cases. The prominent relationship among actors is generalization (a more specific actors inherits the features of its related more general actor). Use-case relationships include generalizations (a more specific use-case inherits the features of its related general use-case), include relationships (a base use-case contains or includes the behavior of another use-case) and extend relationships (given a condition, an extension use-case augments a base use-case).

To show a use-case on a use-case diagram, draw an oval and put the name of the use-case in the center. To draw an actor on the use-case diagram, draw a stick person on the left or right of the diagram. Simple lines depict the actor – use-case relationship.
3.5 The unified modeling language (UML)

### 3.5.2 Component diagram

A component diagram shows the system's dependencies on external software components (libraries, COM components). It is a high-level diagram describing the organization of the software. A component is a modular, deployable and replaceable physical implementation having an identity at run-time. A rectangle with two small rectangles protruding at its side depicts a component. Dashed arrowed lines show the dependency among components.

![Component diagram](image)

Figure 3.2: Simplified component diagram for the controlling software of an in vivo exposure system. Rectangles with two small rectangles protruding at its side depict components. The diagram reveals the controlling software's dependency on three third party components: the Visa32 library to communicate over the General Purpose Interface Bus (GPIB) with hardware, the ProEssentials -32 library to create data charts, and the PDF Report Writer library to generate PDF-documents using simple html-like commands.
3.5.3 Deployment diagram

The deployment diagram shows the system’s deployment in the hardware environment. It illustrates where the different software components physically run and how they communicate with each other. It models the physical runtime. The deployment diagram uses the same elements as the component diagram with some additions, including the concept of a node. A node represents a physical or a virtual machine. A three-dimensional node with the name at the top depicts a node. Nodes may contain components.

Figure 3.3: Simplified deployment diagram of an *in vitro* exposure system. The three-dimensional nodes represent physical computers. The experiment computer sends the collected data to the storage computer using its samba interface. The evaluation computer fetches the data from the storage computer using the samba interface too.

3.5.4 Class diagram

Class diagrams are fundamentally important for programmers. System analysts also consider them as indispensable to describe OO systems. UML classes often map one-to-one to classes used in an OOPL. However, this is not necessarily the case. Sometimes UML classes are more general to simplify the system structure.

The class diagram shows the relation of different entities (people, data, algorithms, objects). It explains the system's static structures at compile time. A class diagram can display logical real-world classes to show the business processes or explicitly show implementation classes programmers typically deal with. In addition to logical class diagrams, implementation class diagrams use programming language typical attributes like vectors or maps.

A rectangle with three horizontal sections describes a class. The upper section shows the class's name, the middle section contains the class's attributes, and the lower section holds the class's methods.

The real power of class diagrams is the relationships they show among classes. These relationships help to understand the concepts used to solve a problem. Two types of relationships exist: inheritance and association. UML inheritance follows the principles of OO: the derived class inherits the identical functionality of its base class and adds additional functionality. To model an inheritance relationship, use a solid line drawn from the derived class with a closed arrowhead pointing to the base class.

Class association indicates a general relation between two classes. UML supports five types of associations: bi-directional, uni-directional, basic aggregation, composition ag-
3.5 The unified modeling language (UML)

Aggregation and reflexive associations. To model an association, draw a solid line between the related classes. Depending on the association's type, the endings of the line vary as well as the cardinality. The cardinality is a multiplicity value next to the related classes and reveals the number of instances involved in the relationship. Additionally, a role name describes the part a class plays in the relationship.

Bi-directional associations are the standard associations and indicate the classes' awareness of their relationships. Bi-directional associations do not have special line endings.

Uni-directional associations show that two classes are related, but only one class is aware of this relationship. To indicate a uni-directional association, draw a solid line with an open arrowhead pointing to the known class.

Basic aggregation is a relationship denoting that one class is a subordinate class of another class. The subordinate class is a part or a data member of a larger class. In a basic aggregation, the child class can outlive its parent class. To symbolize a basic aggregation, draw a solid line from the parent class to the subordinate class, draw an unfilled diamond shape on the parent class's association end and draw an open arrowhead on the child class's association end.

The composite aggregation is a restricted version of the basic aggregation, where the child class's lifecycle is dependant on the parent class's lifecycle. The notation is the same as for the basic aggregation, but the diamond shape is filled this time.

Reflexive associations show a recursive relationship between a class and itself. To show a reflexive association, draw a line from a class to itself.

Figure 3.4: Class diagram of the signal unit structure of in vivo controlling software. DigitalSignalUnit and AnalogSignalUnit inherit from the base class SignalUnit. They overwrite the SetPower() and Init() methods. DigitalSignalUnit aggregates a DigitalRfGenerator, whereas AnalogSignalUnit aggregates a FunctionGenerator and an RfGenerator.
3.5.5 Sequence diagram

Sequence diagrams show the order of method calls from one object to another. They explain the dynamic behavior of system parts. Sequence diagrams are best used to illustrate sequences of complex method calls at the implementation level.

A sequence diagram has two dimensions: the vertical dimension displays the time order of occurring message calls and the horizontal dimension presents the objects and their types executing the methods. To draw a sequence diagram, horizontally spread the objects (OBJ) involved in the entire sequence. Put each object in a box with the syntax name: type, whereby the type is optional. Draw a line with an open arrowhead pointing to the receiving object to express a method call from one object to another. Place the name of the method above the line (METHOD). Optionally draw a dotted line with an arrowhead pointing back to the calling instance and label the return value above the dotted line (RETVAL).

Figure 3.5: Sequence diagram of SAR calculation in an exposure chamber. The involved objects are spread horizontally. Firstly, experiment calls chamber.GetSar(). The chamber in turn calls rfSensor1.GetField(). This method returns a double data type representing the measured field. Afterwards, chamber calls dosimetry.GetSar(field) method with the just measured field as parameter. Dosimetry converts the field value in SAR and returns the converted value. Next, chamber repeats this procedure for the second RF-Sensor. Finally, chamber calls its CalcMean() method, which returns the mean of both SAR values, and returns the mean SAR to the caller.
3.5.6 Activity diagram

An activity diagram is an enhanced flow chart diagram adapted for OO systems. It shows the procedural control flow while processing an activity. Frequently, an activity diagram models a specific use-case at a more detailed level. Activity diagrams are best used to model high-level system workflows.

Activity diagrams are very simple and have a familiar appearance, making them the ideal tool to display processes to customers and non-programming team members. The starting point of an activity diagram is a solid circle connected to the initial activity. To model an activity, draw a rectangle with rounded edges, enclosing the activity's name. Transition lines connect activities or an activity with a decision point (symbol: diamond). Guard conditions regulate the transition from a decision point to the next activity. Merge points join transition lines. Use the same symbol with two or more input paths but only one output path to show a merge point. Synch states allow modeling of parallel processes. They fork program execution. Use a bold solid line with input and output transition lines to show a synch state. The activity's flow terminates when the transition line of the last action in the sequence connects to the final state, symbolized by a bull's eye.

Figure 3.6: Simplified activity diagram of the program flow of an in vivo EMF bio-experiment. At first, the program initializes the system. After having loaded the animals, the operator starts the experiment and switches the system into the run experiment state. At this state, the program flow splits in two threads, simulate signal and collect & evaluate data. If data are out of range, the system switches to error state. After the experiment is finished, the program flow again splits into two threads, idle run and data backup. When the data backup is finished, only idle run remains active.
3.5.7 Conclusions about the UML

UML provides developers with a stable and common design language enabling them to talk about software engineering problems in an abstract way using a set of standardized diagrams. With dependable UML diagrams, new project staff members can understand the structure of an existing software system more quickly. UML also simplifies the spreading of new ideas in the development community. UML has been developed especially to fulfill the needs of object-oriented system engineering. Therefore, it explicitly supports object-oriented principles like encapsulation, inheritance and polymorphism in most diagrams. The building block of UML is the object, and it can be applied in all phases of the SDLC. Following the same principles, the transitions from top-level diagrams to more detailed diagrams is relatively smooth. No new paradigms have to be used for each diagram. Additionally, UML serves as an interface between system analysts and programmers and may help to bring them closer together and therefore to coordinate work more effectively.

3.6 Object-oriented design quality criteria

Three criteria indicate the quality of an object-oriented design: high cohesion, low coupling and connascence.

3.6.1 High cohesion

Cohesion expresses how single minded a module (class, component) in a software system is [17]. Generally, the higher the modules’ cohesion is, the better the system’s understandability. My personal maxim to achieve high cohesion is “Put together what belongs together.” For example, consider a class capable of generating a written report of its internal structure. This class should not care about the report’s formatting, because the formatting is not the class’s core responsibility. Separating the formatting and the data allows exchanging the formatting style without changing the internal structure of the class. The example class’s declaration could look like this:

```csharp
class SomeClass {
    bool GenerateReport(ReportFormatter& formatter);
};
```

Coud and Yourdon [18] have identified three types of cohesions for OO systems: method-, class- and generalization/specialization cohesion. Method cohesion addresses the cohesion within an individual method and assesses how single minded a method is. A method performing multiple actions is more difficult to understand than a single-minded one. Class cohesion assesses the cohesion of attributes and methods of a class and assesses how single minded a class is. A class should only represent one thing, and all attributes and methods should contribute to the proper representation of this thing. Having a well-defined class automatically results in high-class cohesion. Generalization/specialization cohesion addresses the sensibility of the inheritance hierarchy and
assesses the classes’ relations involved in the inheritance hierarchy. Highly cohesive inheritance hierarchies only support the generalization semantics (“is a” relationships).

### 3.6.2 Low coupling

Coupling describes the modules’ interdependency level in a system [19]. The lower the interdependency or the coupling, the less the changes in one design part influence the design of other parts. The expandability and maintainability of large systems strongly depend on low coupling. Low coupling enables new functionalities to be added in a system part without having to perform complete system tests afterwards. Coud and Yourdon [18] have identified two types of coupling for OO systems: interaction and inheritance. Interaction coupling describes the coupling of methods and objects via message passing. Inheritance coupling describes the classes’ coupling degree in an inheritance hierarchy.

### 3.6.3 Connascence

Connascence combines the ideas of cohesion and coupling with the paradigms of encapsulation [20]. Maximizing the cohesion (connascence) within an encapsulation boundary (method, class, component) and minimizing the coupling (connascence) between the encapsulation boundaries are the characteristics of good software design.

### 3.7 Object-oriented design patterns

Published in 1995, “Design Patterns, Elements of Reusable Object-Oriented Software” [19] by Gamma, Helm, Johnson and Vlissides, has applied the idea of design patterns to software design. This book has deeply influenced OO design. Today, it is one of the most often cited publications in the field of software engineering.

A software pattern is a solution to a problem in a given context. The same solution or pattern can be applied to solve recurring problems without ever doing the same work repeatedly. Gamma et al. have reviewed the software design of running stable systems and extracted, catalogued and published solutions to recurring problems: the patterns. The patterns have been inherent knowledge of the developing community. This is the basic strength of software patterns: They have proven to work in real-world software and are not simply an abstract academic concept, which may fail in commercial software. Gamma et al. have applied the reverse engineering method to discover the patterns. According to Shalloway [15], patterns serve three purposes:

- **reuse solutions.** Commonly recurring problems can be solved reusing established designs.

- **establish common terminology.** Design patterns provide a common base of vocabulary and a common problem viewpoint. They simplify project communication and teamwork and offer reference points during the SDLC.
Software engineering methodology

- high level perspective. Enabling the high-level modeling of a software system, patterns express and model complicated processes using simple formulations and well-known diagrams.

Some patterns like the observer pattern exist in almost every software system. To demonstrate the design pattern's power, consider an OO data acquisition system. A data acquisition class (DAC) typically collects data, whereas sensor classes evaluate the data. Apparently, the data must be moved from one class (the DAC) to a collection of classes (the sensors). Several mechanisms handling this kind of one-to-many data transfer exist. A simple but nevertheless powerful solution is offered by the observer pattern [19]. The DAC plays the subject, whereas the sensors act as observers. The sensors (observers) register themselves with the DAC (subject). The DAC notifies the registered sensors when new data are available. Having a unique identifier and implementing an \( \text{Update()} \) method, the sensors grab their data autonomously. Developers familiar with design patterns will understand the data distribution from the DAC to the sensors quickly when they learn that the DAC and the sensors constitute an observer pattern. The simple mention of a concept suffices to explain a complicated chain of events.

3.8 Real-time software design

A real-time system must react adequately and in real time to events from the system’s environment. Typically, a real-time system consists of hardware devices, a computer communicating with these devices and controlling software. The hardware devices provide the computer with information about the environment. The software evaluates the information and issues a control signal depending on the data. A real-time system's correct implementation depends on responding to events within a certain time interval. Software engineering differentiates between hard- and soft real-time systems. A hard real-time system must react within a given time interval to guarantee correct results. If the system extends the time interval, the behavior becomes incorrect. In contrast, a soft real-time system's operation is merely degraded if the timing interval is extended.

Figure 3.7 shows a simple stimulation/response model of a real-time system. Sensors collect information about the external environment and provide them to the real-time control system in the form of stimulations. The system evaluates the stimulations and directs the response to actuators. The actuators in turn operate hardware units, which influence the system’s environment. Simplified, the system receives information about the environment and takes appropriate actions to influence the environment. Sensors, control system and actuators constitute a closed loop system. Naturally, should it be necessary, the controlling system can trigger actuators independently from the attached sensor stimulations.
3.8 Real-time software design

Figure 3.7: General model of a real-time system. Sensors provide the real-time system with information. The system evaluates the information and directs the response to the actuators.

Real-time systems contain two types of stimulations:

- **Periodic stimulations.** Typically used for systematically monitoring, polling or sampling information from sensors over a long time interval, periodic stimulations occur on a regular time basis between fixed time intervals. For example, consider the controlling software of EMF bio-experiments. The software reads temperature and field sensors every ten seconds.

- **Sporadic stimulations.** They occur irregularly and are event driven. Usually, an external signal such as a hardware failure activates sporadic stimulations. However, the controlling system itself may also trigger stimulations. In this case, actuators stimulate the system. For example, consider a synthetic signal generation mechanism used in software for EMF bio-experiments. The controlling system simulates a signal in the time domain and triggers actuators at appropriate times to control a signal generator. That way the signal generator creates the simulated signal.

A real-time system must transfer received stimulations as soon as possible to the appropriate handler. Sequential programs support this mechanism only limitedly. Therefore, real-time systems normally have a concurrent and cooperating architecture. The standard process model [21] is the most common model for this architecture. Software implementing states, modes and transitions consist of a set of interacting processes. Processes send messages to each other, share hardware and software, and compete for resources. The standard process model can be implemented using threads and/or timer functions.

The run-time behavior of real-time computer hardware must be predictable, so that programmers in turn can predict application behavior. Hardware must be reliable and fault tolerant to prevent costly errors or at least handle faults predictably. Redundancy can significantly enhance hardware reliability. Whenever a hardware part fails, a replacement part takes care of the functionality. For example, systems for EMF bio-experiments should have more than one field sensor per exposure chamber. If software detects broken field sensors, it ignores them and uses only the intact sensors to calculate the field.
strength. Both software and hardware can implement some system capabilities. Therefore, system designers must carefully coordinate the hardware and software design to achieve smooth interactions.

The real-time software design process differs from other software design processes. In addition to the identification of objects and their relationships, events (stimulations) and associated responses must be identified. Early identification of these events enables assessment of whether the system can meet the timing constraints. Events are the building blocks of real-time systems. The software is organized around these events. Each event transforms the software into another state. Therefore, the state pattern [19] is ideally suitable for real-time software.

### 3.8.1 Data acquisition systems

Data acquisition systems periodically collect and analyze data from sensors. Data acquisition time and data processing time may vary. Most systems for data acquisition buffer the input data to compensate speed differences or to temporally separate the collection and analysis of the data. The process collecting the data (producer) adds information in the buffer, and the process analyzing the data (consumer) takes information from the buffer.

Frequently, data acquisition systems apply the cyclic executive (CE) approach. This method handles only periodic processes; sporadic processes are either translated into equivalent periodic ones or handled in separate threads. The principle aim is to define tasks that meet their deadlines. This goal is accomplished using a predefined, interleaving executions schedule. A CE program controls the process executions according to this pre-run-time schedule. The CE method is simple, generates efficient systems and generates predictable system behavior. However, some authors [22] claim this method to be somewhat inflexible. The next chapter describes a modified CE approach used in all realized controlling software for EMF bio-experiments.

### 3.9 Heavyweight software engineering methodologies

Following the classical software development life cycle, OOAD using UML notation is today's most frequently applied and probably best described heavyweight software engineering methodology. Heavyweight methodologies strongly focus on planning, analyzing and designing software systems. Ideally, implementation starts only when the whole system is completely planned and specified. The term "heavyweight" expresses these methodologies' slow reaction time to changing conditions.

Civil or mechanical engineering disciplines have significantly inspired these methodologies. Such disciplines place emphasis on planning before the actual building process starts. Engineers generate drawings precisely indicating the parts involved and how these parts fit together. Most design decisions are made at the time the drawings are generated. Afterwards, the engineers pass the drawings to a different team to perform the actual building.
Like other engineering disciplines, heavyweight software engineering methodologies assume that design- and construction/implementation activities can be more or less separated. Indeed, most modern heavyweight methodologies consider iterative and incremental development. However, iteration is performed merely to adjust the various diagrams to the current state of the system. At any time, the diagrams should completely describe the system. Heavyweight methodologies do not change the overall system design once the implementation has started. The iterations simply bring the documentation and the system closer together.

The initial aim of heavyweight methodologies has been to make software engineering more predictable and efficient. This thesis portrays only one heavyweight software engineering methodology: OOAD with UML. Other, older methodologies are hardly used in the industry any more and are therefore omitted.

3.9.1 OOAD with UML

Using UML notations and diagrams in all phases of the SDCL, this methodology strongly focuses on modeling. It assumes that a project stands and falls with its specification. The better the specification and modeling of a project, the more likely it can be realized in time and in budget. Additionally, detailed modeling simplifies the extensions of existing programs and reduces the period of vocational adjustments for new employees. New employees can study the models and can get a basic idea of the system before looking at the source code.

Economical considerations are the driving force to use this development methodology. Having a set of standardized diagrams enables fast circulation of new ideas and may improve the project teamwork. Besides the modeling of a system, these comparatively simple diagrams have also an important psychological aspect. Project managers without programming background tend to feel inferior to programmers when technical details are discussed. Some programmers even try to profit from this situation and intentionally use technical terms to mark who really does the hard work. Obviously, such hidden power struggles affect effective teamwork. Establishing a common vocabulary, UML modeling overcomes the knowledge discrepancy to a certain degree and permits all team members to contribute their ideas. However, programmers must be ready to abandon their hidden power and accept managers as equivalent partners. Only then, the full power of UML modeling diagrams is exploited. The hidden conflict between managers and programmers is so popular that a separate cartoon series covers this topic: the Dilbert cartoon [23].
The development processes start with exactly written descriptions of concrete scenarios. Scenarios are the ancestors of use-cases and are established together with end-users. They constitute the initial point of all types of specification. Scenarios are examples out of the application area and serve as starting points for model abstraction. For example, the concrete generation of an RF signal composes a scenario. Several similar scenarios allow the derivation of a more abstract description. Analysts group comparable scenarios and extract common issues. Following the example, all concrete scenarios for the generation of an RF signal constitute the group RF signal generation. The group of scenarios represents a use-case. Expressed more abstractly, the sum of all scenarios in a given context composes a use-case; a use-case is the set of all possible scenarios.

Starting from these use-cases, other more detailed diagrams model the system's interaction with the environment, the system's static structure and the system's internal activities. The subsequent diagrams should be congruent with the initial use-cases and with each other. Consequently, the system is consistently modeled in different granularity. To maintain congruency, changes in one diagram possibly require changes in another. UML-modeling software like Microsoft's Visio or IBM's Rational Rose support compatible sets of UML diagrams. Changes in one diagram type are automatically assigned to the affected diagrams.

Having a set of congruent UML diagrams and textual description at hand, a team of programmers should be capable of implementing a system entirely. The basic idea behind this approach is that a system can be modeled completely and that it only makes sense to start implementation after specification is complete. The programmer's job is simply to implement the modeled system.

Totally adhering to the model and implementing what the diagrams require, the programmer has very limited liberties. He is not allowed to change the model. The programmer's work is the one of a simple implementer.

Following this approach, a system engineering company can separate the modeling from the implementation process. Therefore, it can outsource the implementation processes to a contractor company located in a low salary country. The contractor company implements the system according to the set of UML diagrams and delivers the whole system to the contracting party. The company tests the system's raw implementation and adds
additional minor changes or generates a new model to be again implemented through the contractor company.

This scenario may sound like dreams of the future. However, some companies are working towards its realization. Oliver Rhas, a programmer at Winterthur Versicherung, informed me how the system development process proceeds in his company. The business analysts identify the new system's business needs. Afterwards, they hand a textual description of the desired needs to the system analysts. Collaborating with end-users, the system analysts identify the requirements, produce feasibility studies and generate detailed UML diagrams as well as requirement documents. Next, the diagrams and documents are proposed to the programming team. When the programming team considers the information as sufficient, the chief developer signs a contract. He assures the information to be sufficient for a complete system implementation. Winterthur Versicherung's long-term objective is to generate comprehensive diagrams and outsource the implementation to an Indian contractor company.

The automatic code generation out of UML diagrams goes even a step further. The modeler creates a set of congruent diagrams. A code generating software interprets the diagrams and automatically generates source code in a specified programming language. Changes in the model automatically lead to source code changes and vice versa. High-level web publishing tools like Macromedia's Dreamweaver or Microsoft's FrontPage use a similar mechanism. HTML source code is generated automatically out of a visual description. Up to now, no software generating source code out of UML diagrams provides useful results. However, new programming languages tightly coupled to UML may be developed, and some day not all too far in the future automatic code generation will become possible. One of the obvious advantages of this process would be that code and documentation would be congruent.

3.10 Lightweight or agile software engineering methodologies

Firstly, the appearance of UML was something like the Holy Grail for software engineers. Ultimately, they had a tool to formalize the development process. The diagrams offered by UML should become equally important for software engineers as blueprints for civil engineers. At last, the separation of design and implementation activities should become possible and software engineering has been progressing toward becoming a mature engineering discipline, which no longer has to hide behind other engineering disciplines. The software industry no longer has to be dependent on the individual skills of hard to control and headstrong programmers. Programmers should become easily exchangeable.

However, after working for some years with UML, the great disillusionment emerged. UML could not hold all promised advantages. Mainly the separation of design and implementation proved to be far from feasible. Over the years, software engineers have started to criticize the inflated bureaucracy associated with UML. They argued that following this methodology slows down the whole development speed.
As a reaction to the disillusion of UML's power, a new group of methodologies has appeared in the last few years. First known as lightweight methodologies, they are now called agile methodologies. Once again, agile methodologies attempt to offer an answer to the eager business community for efficient development processes. The study of Nandhakumar and Avison [21] substantiates the importance of new approaches. They argue that traditional development methodologies "are treated primarily as a necessary fiction to present an image of control or to provide symbolic status." Truex et al. [3] take an even extremer position and state that possibly traditional methods are "merely unattainable ideals and hypothetical straws that provide normative guidance to utopian development situations". Over time, commercial software developers have become skeptical whenever new and difficult to grasp methodologies have been presented. Normally they think that these methodologies are too academic and not useful for daily work. As a result, some developers refuse to accept any methodology at all and even do not want to look at new methodologies. Developed from practitioners for practitioners, agile methodologies explicitly address these skeptical developers.

Agile methods are fast and simple. Following the statement "the key part of the documentation is the source code", they are rather code-oriented than document-oriented. In his outstanding publication [25], Martin Fowler mentions two fundamental differences between heavyweight and agile methodologies:

- agile methods are adaptive rather than predictive. Heavyweight methodologies try to plan large parts of the software in detail and for a long time span ahead. Obviously, this procedure works well in stable situations, but fails when requirements or technologies change. Therefore, Fowler concludes that their nature is to resist changes. This is evident. Consider the development of a complex UML based system. If, during implementation, a programmer detects a simpler solution to a problem than the one proposed by the UML diagram, he may anyhow not use the simpler solution. Using another solution would impose him to change the diagrams, which he probably does not want to. Some major changes would even question the congruency of the whole set of diagrams. Agile methods, however, accept changes as natural and essential in the software development process and embrace them. One of their goals is to adapt quickly to these changes.

- agile methods are people-oriented rather than process oriented. Heavyweight methodologies try to abstract people from the processes they use. Asserting that no process will ever be as excellent as the skills of a development team, agile methodologies merely define processes supporting the development team in their work.

The influence of civil and mechanical engineering on most heavyweight methodologies is dangerous. Software engineering and civil or mechanical engineering are not offhand comparable. The traditional engineering disciplines have two fundamentally different activities: design and construction. Design diagrams allow the generation of a detailed construction plan and enable the compilation of a reasonably predictable schedule and budget for construction. A construction plan tells the people performing the construction how to do their work.
Design activities are generally difficult to predict and require well-trained and therefore expensive people. Construction activities on the other hand are comparably easy to predict and generally require less qualified and therefore less expensive people. Assigned to software engineering, the separation of design and implementation is the crucial task. Once achieved, implementation will become straightforward after the planning is finished and a predictable schedule can be generated. However, predictable schedules are only realistic if the requirements remain stable. And this is exactly the deciding argument.

Most software development processes do not have stable requirements and are therefore not predictable. After all, software is supposed to be soft and to be able to change easily. An exception, for example, is NASA’s space satellite programs. In these projects, the requirements stay fixed and the organization has virtually no financial limitations. (This constellation does not exclude fatal software errors, however: Remember the crash of an unmanned shuttle because of an unit conversion error in the controlling software.) However, most software projects belong to a different category, where predictability is not possible. Following a predictable process when it actually is not is also dangerous. Designers frequently have the temptation to identify predictable processes, because they are very desirable, since they allow long-term planning activities.

Using a predictive methodology (and most heavyweight methodologies are predictive) is not adequate in non-predictive situations. However, abandoning predictability does not denote reverting to uncontrollable chaos. Instead, another methodology is required enabling the control of unpredictable processes. Being adaptive, agile methodologies explicitly address unpredictable processes.

3.10.1 Key features of agile methodologies

Published by a group of software developers in 2001, the Agile Software Development Manifesto marks the cornerstone of the agile movement in the software industry. The manifesto contains four key statements [26]:

- individuals and interactions over processes and tools
- working software over comprehensive documentation
- customer collaboration over contract negotiation
- responding to change over following a plan

The authors of the manifesto indeed admit that there is value in the items on the right, but they value the items on the left more.

The first statement emphasizes the role of individuals in the software development process. Traditional heavyweight methodologies do not treat the developers as separate persons. Rather, they consider developers as exchangeable resources. Agile methods reject this assumption. In his paper “Characterizing People as Non-Linear, First-Order Components in Software Development” [27], Alistair Cockburn, one of the spiritual leaders of
the agile community, concludes that the people are the most important factor in software development.

Skilled programmers act creatively and professionally and are generally sensible personalities. They identify with their work and tend to take criticism on their work personally. Treating them as individuals raises morale and consequently improves productivity.

The agile method's people first approach contradicts classical business thinking, which treats people merely as resources. This thinking expresses the modern term “human resources” for personnel management. The paradigm shift to people first requires a fundamental change in management and needs new models for cooperation. Identifying with their work, experienced programmers generally are competent professionals and have a high work ethic. Excellent programmers must have this work attitude to realize complex and large-scale systems. They must have the longing to realize the system. A good developer team only gets along with few simple followers that do not really care if the final system will run. Such simple followers might even lower the team's performance.

According to Fowler [25], accepting an agile process requires the involvement of the whole team. That means managers should not force the developers to accept an agile methodology. It is important that the developers themselves can choose to follow an agile methodology. Forcing them to move to the new processes would undermine the very central aspect of agile methodologies. Having the technical leadership, developers should be able to make all technical decisions themselves. However, fearing a loss of power and questioning their position, confessed managers may not welcome new ideas. They may resist new methodologies and favor traditional management techniques. The developers in turn must accept the role of the manager, and they must not feel secretly superior.

Such attitudes affected by personal ego have no space in agile processes. Management and developers must have equal project leadership. The team members must respect each other as individuals and should appreciate the work of other members. For example, consider the paperwork in a project. Traditionally, developers do not like to do paperwork and even claim that managers to generate useless paperwork. Paradoxically, the managers' work enables the developers to focus on the work they like: programming. That is why developers should appreciate the work of managers instead of criticizing it.

The word interaction in the first statement of the agile manifesto expresses the importance of good and frequent communication among the programmers. The rate of technology change in the software industry is very fast and without parallel in any other technology. For this reason, the amount of knowledge available about software engineering is huge. Obviously, nobody can master all technologies. The amount of information is simply too large. Although it is not desirable for a team to master all technologies, it must be able to master the most important ones.

Communication among the programmers helps to grasp new important technologies and to separate them from useless ones. Therefore, communication is essential for a team's long-term success. Personal communication is the fastest way to spread new ideas and technologies and to assess their potential. Certainly, a programmer can read books to
3.10 Lightweight or agile software engineering methodologies

stay up to date. However, this self-teaching is not especially efficient. Additionally, programmers tend to be lethargic in adopting fundamental new principles. They grasp at their existing knowledge and try to elaborate it. Therefore, they often read books about technologies they already master. Eventually, they could miss important changes. Agile methods are aware of the importance of frequent team communication and emphasize this fact repeatedly.

An especially desirable result of increased communication is the boost of team spirit. The more team members speak together, the better their motivation and the better the team's efficiency. Additionally, good team spirit keeps personnel fluctuation small and can even attract new skilled developers, since news about a company's extraordinary team spirit spreads fast. Most developers prefer working in an agreeable environment and valuate these criteria almost as high as the salary. Good team spirit also enhances we-feelings. These feelings help to consider a software package as the result of the whole team's efforts. Developers no longer just want their code to work fine but want the whole system to run smoothly. In teams with low team spirit, some developers hope in private that another developer makes serious errors; they thereby feel valorized. Such attitudes may have catastrophic effects. In an agile development environment, tracing an error back to the programmer who made it is useless and even considered to be fatal for the team spirit. The main point is that the problem is fixed and that the system is improved.

The agile manifesto's second statement stresses the central importance of working software. Agile methodologies consider working software as the primary measure of progress. Therefore, they propose short release intervals, because the releases serve as primary feedback for the developers and for measuring the team's performance.

In contrast to heavyweight methodologies, the agile procedure puts comparably little importance on comprehensive documentation. Detailed documentation is considered to cumber the project progress. Documentation should not be abandoned altogether, but only used when real benefit results. The attitude is that no however complete documentation can ever substitute source code.

Furthermore, developers should keep the source code simple and straightforward. Agile methodologies pay little importance on ultra flexible and extensible solutions. They strongly focus on finding solutions for the active problems at hand. Predicting the application potential of generic solutions is very difficult and afflicted with high incertitude. Therefore, spending a lot of time and money for a flexible solution which will probably never be used again in the future is bootless. Certainly, a good code design is important; however there is the danger that analysts become "paralyzed by analyse". This expresses the fact that sometimes analysts study a problem too long and get lost in insignificant details. The German language has a fitting proverb for the straightforward agile approach: "Probieren geht über studieren". The English equivalent is "The proof of the pudding is in the eating".

Once a solution is found, the source code must be refactored to keep it simple. Refactoring denotes the rearranging of source code without adding or removing functionality. The only purpose is to enhance the internal code structure. Therefore, agile developers
constantly refactor their code, always trying to find simpler solutions and to improve code quality.

The third statement addresses a software company's relationship to the customers. Cooperation between developers and clients is given preference over strict contracts. However, agile developers admit the increasing importance of contracts with increasing project size. Agile methodologies consider negotiations as a means of achieving and maintaining a viable relationship. Focusing on delivering business value immediately as the project starts, agile cooperation reduces the risk of non-fulfillment of the contract. Agile processes also aim at a new price contract model. In the past, the usual form was a fixed price contract. However, a fixed price contract requires stable requirements and therefore a predictive process. Agile methodologies do not consider software development as a predictive but rather as an adaptive process, since requirements may change in the course of the development. In principle, a fixed price contract cannot work for adaptive processes fulfilling changing requirements. Trying to fit a fixed price model to adaptive processes has a high potential to hurt the customer as well as the software development company.

According to Fowler [25], fixing the time, price and scope of software is not possible. The usual agile approach is to fix the time and price and to allow the scope to vary according to the customer's current business needs. In such a business model, the customer has much more control over the software development process. However, this business model assumes mutual confidence. It can lead, finally, to true business partnership.

The fourth statement appeals to the agile methodologies' readiness for motion and changes. Considering the software development process to be adaptive, agile methodologies welcome changing requirements even late in the development process. This flexibility offers the customers a high degree of freedom of decision. The customer can adapt the software to business needs quickly and obtains competitive advantage. Short release cycles guarantee customer satisfaction and enable customers to test new features immediately. Nevertheless, agile methodologies honor the values of plans, but only plans accommodating the adaptive nature of software development.

Ambler [28] has suggested the following set of commonsense approaches existing in all agile software development methodologies:

- people matter
- less documentation is possible
- communication is a critical issue
- modeling tools are not as useful as usually thought
- big and detailed up-front design is not required.

Abrahamsson et al. [29] have asked the question, what features make a development methodology an agile one. Their answer is as follows: An agile methodology is incremental (small software releases, with rapid cycles), cooperative (customer and developers working constantly together with close communication), straightforward (methodology
is easy to learn and to modify, well documented, and adaptive (able to make last moment changes).

This chapter's last part describes the most popular agile methodologies: Extreme Programming (XP), Crystal Family, Lean Development and Pragmatic Programming.

3.10.2 Extreme programming XP

Kent Beck's publication "Embracing Change With Extreme Programming" [30] marks the genesis of XP. Today, XP is by far the most popular agile methodology. In fact, XP's appearance strongly influenced the foundation of the agile movement. Without XP, probably, the world would not have agile methodologies at all. XP is a comprehensive methodology and covers all phases from planning to implementation.

XP begins with four fundamental values: simplicity, feedback, communication and courage [31]. Anyone adopting these values can learn the XP practices. XP aims to find the simplest possible solution in a given context. Simple solutions are quicker, more cost effective to produce, easier to explain, maintain and develop than extensible and flexible solutions. Following these guidelines, XP itself is kept very simple. XP's primary quality control mechanism is feedback. The feedback happens at various levels: developers receive feedback from automatic unit tests, users give feedback about the usefulness of the software and developers get feedback from colleagues. Intensive communication among all team members is considered fundamental for a team's long-term success. XP highlights personal communications, since this communication form exchanges information very quickly.

Simple solutions, honest feedback and open communication require courage. Particularly, open communication requires a great deal of courage. Developers must be able to criticize colleagues without hurting them and must accept in turn their critics. Fears, conceit and rivalry among programmers are often responsible for the failure of software development projects. Therefore, open and honest communication is a crucial task. Having worked in projects without open communication, some developers have difficulties adopting the new communication style. Therefore, establishing open communication is a very demanding task and requires casual manners and a great deal of soft skills or social competence on the part of the programmers.

From the four values simplicity, feedback, communication and courage Beck [31] derives practices an XP project should follow:

- Planning game. Close collaboration of business people and developers is needed to estimate the time to implement short tasks. (In the XP jargon called "customer stories"). Customers prioritize the requirements and the developers are responsible for estimating the cost.

- Metaphors. Clear metaphors between developers and customers describe the core ideas and support the ongoing system development.
• small/short releases. Customers get modified system components within a short period. They can quickly profit from the system's new features and at the same time, they evaluate the modifications. The evaluations in turn may reveal shortcomings and influence further development activities.

• testing. XP follows the test-first programming maxim and uses intensively unit tests. Developers write test classes and the actual classes at the same time. Tests for public method calls are implemented before the operations themselves are implemented. At the end of each development cycle, customers perform acceptance tests.

• simple design. Only considering known and omitting possible requirements, developers design the system as simply as possible. The code provides no explicit flexibility and extensibility for future changes.

• refactoring. Developers constantly refactor the source code to make it simpler and clearer.

• pair programming. Two people write the code at one computer to improve its quality. Changing pairs quickly promotes the spreading of system knowledge.

• collective ownership. All developers are responsible for the whole system. Any developer can change any source code at any time. Because of this organization, individual developers are not indispensable. The last chapter of this thesis covers the risk when large software projects are dependent on key developers.

• continuous integration. Developers integrate a new piece of code into the code base as soon as it is ready, enabling the whole team to test the changes straight away. The changes have to pass all automatic test runs before they are accepted.

• coding standards. The team defines pragmatic coding standards before a project starts. For example, coding standards rule code indents or brackets positions. If all programmers follow these standards, the source code becomes uniform. This uniformity enables the programmers to quickly understand foreign code.

• sustainable pace. Developers work maximally 40 hours a week. No two overtime weeks in a row are accepted. An upper working hour limit ensures the programmers' long-term creativity and dedication.

• on-site customer. A customer business person is dedicated to the project and is available full-time for the project. This leads to a close relationship between the developer team and customer. The customer person guarantees that the system in development will meet the critical requirements.

• open workspace. Workspace configuration should boost personal communication.

Sometimes, developers and managers claim that XP offers nothing new and that they have used XP practices for years. However, such people equate XP methodology often with developing software without an underlying plan. XP offers a planned and structured procedure to handle unpredictable processes. It is not simply a pretext to legitimize a
3.10 Lightweight or agile software engineering methodologies

chaotic approach to software engineering. According to Beck [31], a project using 80\% of the XP practices only achieves 20\% of the effects. These figures show a tremendous difference between a project using just some practices and a true XP project.

![Figure 3.9: XP process life cycle](image)

Relying on refactoring and existing systems, XP follows an evolutionary design process consisting of several iterations. All design is centered on the current iteration. No work is done for future needs or for the next iteration. At the beginning of the iteration, a planning meeting discusses the next user stories to be implemented. Generally, the customer selects the sequence of the user stories. The developer team breaks the selected stories into discrete programming tasks. Each developer pair estimates the time to implement the task based on ideal programming days. Such days are the number of days to program a task without distractions or interruptions. At the end of the iteration, the team checks the system for code duplication. Possible duplication will be consequently removed. The resulting design process is a combination of discipline with adaptivity.

3.10.3 Crystal family

The Crystal Family of methodologies includes a number of different methodologies with common ideas. Depending on the project (size, criticality), a development team has to choose the adequate methodology. Developed by Alistair Cockburn, a long-run IBM methodology observer, the Crystal family of methodologies supplements his direct experience with actively seeking to interview projects to see how they work. The Crystal approach differs in this point from most other agile methodologies, which build solely on personal experience.

Crystal suggests choosing the appropriate methodologies for a project based on two criteria: size and criticality. Larger projects require more coordination and heavier methodologies than smaller ones. If many people work on a project, at least some people have to
know who is doing what. This is important to eliminate redundancy. In large projects, it is no longer possible for every person to know the work of all the other team members. Simply, it is not possible that all people talk together. Therefore, large projects need people who coordinate in the work fulltime.

Systems that are more critical require more rigorous tests and formal methods. This is obvious. Systems controlling dangerous processes like controlling software for nuclear power plants must be tested very thoroughly to assure error-free functionality. Figure 3.10 shows a matrix containing the horizontal axis “project size” (people involved) and the vertical axis “system criticality”. This matrix allows selection of the appropriate methodology depending on size and criticality. Cockburn [12] distinguishes four criticality levels based on potential system failure: Comfort (C), Discretionary money (D), Essential money (E) and Life (L). For example, a system crash at criticality level D causes the loss of a limited amount of money, whereas a crash at criticality level L causes a loss of life. Given its size and criticality, the matrix allows the assignment of a project to a project category symbol. For example, the symbol E20 describes a project with 20 persons working on a system with the criticality level “Essential money” (system crash causes loss of large amount of money).

Currently the Crystal Family consists of three methodologies: Crystal Clear, for small projects up to six developers; Crystal Orange for medium sized projects with 10 to 40 project members; Crystal Web, a Cockburn’s tailored methodology for the company eBucks.com. The following discussion considers only the similarities of all three methods. A single methodology’s detailed description is offered in [32].

Crystal’s philosophy is to offer a family of human-powered and adaptive, ultra light, “shrink-to-fit” software development methodologies [32]. In contrast to architecture centric methodologies like OOAD, Crystal emphasizes enhancing the work of the people involved. In other words, it is people centric. Ultra light expresses the reduction of paperwork, overhead and bureaucracy that results using an appropriate Crystal Family
methodology for a given project size and given priorities. "Shrink-to-fit" means that the development process starts with a light methodology. Then the aim is to make the methodology even smaller and better fitting. Crystal is open for incorporating new ideas from other methodologies.

The adaptivity and the incorporation of new ideas is the very strength of the Crystal Family of methodologies. Every team is different and needs a methodology suited to the corporate culture. Additionally, the methodology's acceptanee among the team members increases when the methodology considers the team's peculiarities. Ideally, the whole team selects and adapts the methodology democratically. People are more motivated when they follow a methodology they have chosen themselves than when following a methodology that only the upper management has selected.

The Crystal family incorporates two common rules:

- projects follow incremental development cycles with increments of four months or preferably less.
- the team discusses their experiences after every cycle. These discussions help to detect weaknesses and to improve performance. Additionally, they level the knowledge about the system among the developers.

These rules are followed using two techniques:

- methodology adaptation. The team adapts a basic methodology according to the current project needs.
- reflection workshops. These workshops evaluate the benefit of the chosen methodology and help to spread knowledge.

Cockburn [12] suggests the application of a set of polices during the development process:

- each increment's output is a new software release
- software releases instead of written documents serve as milestones and measure the project's progress
- users are closely involved in the development process
- automatic unit and integration tests check the code for correctness
- two user viewings per release
- the team holds workshops for product- and methodology-tuning at the beginning and in the middle of each increment
- adjacent activities only start when the foregoing ones are finished and tested
Speaking from my own experience, the last item is fundamentally important. When I started planning and implementing controlling software for EMF bio-experiments, I could concentrate on a single project. In this phase, I was very productive and the realized systems did not have serious errors. In the course of time, I had to perform several tasks in parallel (programming new systems, finding bugs in old systems, tracing hardware failures, performing dosimetry measurements, installing new systems all over the world). This parallelism affected my work quality as well as my personal well-being. People who have never programmed large systems do not know from their own experience how difficult and exhausting it is to switch between several projects. It takes a considerable amount of time to become acquainted again with a system one has realized in the past. Therefore, good project planning assures that a single developer works only on one task at a time. This is good work organization for the company as well as for the developer.

Considering the policies, Crystal methodologies follow an incremental development process. Each increment contains a number of practices including staging, monitoring, parallelism and flux, and reviewing.

**Staging**
Staging describes the planning of the system's next increment. The team selects the requirements to implement in the increment and schedules the period.

**Revision and review**
Each increment consists of iterations. A single iteration contains construction, demonstration and review.

**Monitoring**
Milestones (start, review 1, review 2, test, deliver) and stability stages (wildly fluctuating, fluctuating and stable enough to review) measure the development progress.

**Parallelism and flux**
Once the stability stage reaches "stable enough for review", the next task can start. This organization allows parallel proceeding of multiple tasks.

**Methodology-tuning technique**
Project interviews and team workshops help to work out a specific Crystal methodology for an individual project. This allows fixing or improving the development process. The team gains knowledge in each increment and can use the new information in the next increment.

**User viewings**
Crystal suggests two user viewings per one release.
Reflection workshops

The team holds reflection workshops at the beginning and in the middle of each increment.

Figure 3.10 summarizes the development process proposed by the Crystal Family of methodologies and shows one generic Crystal increment.

![Figure 3.10: Development process proposed by the Crystal Family of methodologies](image)

3.10.4 Lean software development

Lean Software Development (LSD) has roots in Lean Production (LP) and Totally Quality Management (TQM). Both, LM and TQM are business strategies to improve production processes.

LP evolved from Taiichi Ohno’s efficient system for creating cheap and high-quality automobiles. Ohno was the chief architect of the Toyota Production System. This system leveraged Toyota from a spinning and weaving company to a competitive car production company. Ohno eliminated waste in the production process wherever possible, which led to rapid product flow and built-in quality. Over time, Ohno realized that these values lead to the highest quality, lowest costs and shortest lead time for products.

TQM traces back to the studies of Dr. W. Edwards Deming [33]. According to Deming, quality is a basic management responsibility, and poor quality mostly results from badly planned systems. Demming assumes that the employees have the immanent desire to do high quality work. The production system must offer the basic conditions so that the
employees can do high quality work. Demming recognized that teamwork and trust-based relationships with suppliers boost the production process.

LP and TQM merged in the 1980's and formed a revolutionary production process, which sustainably influenced the industry. Ten simple rules summarize the new process [34]:

1. eliminate waste
2. minimize inventory
3. maximize flow
4. pull from demand
5. empower workers
6. meet customer requirements
7. do it right the first time
8. abolish local optimization
9. partner with suppliers
10. create a culture of continuous improvement

Numerous industries including logistics, finance and engineering have successfully adapted these rules. Obviously, a specific industry has to modify these rules to tailor them to the special needs of the industry. Nevertheless, the basic principles have remained constant.

Lean Software Development is now the application of lean principles to software development. Compared with other agile methodologies, LSD focuses more on strategy than on the development processes. Offering no structuralized development process, it is not a comprehending methodology, but provides principles to improve software development quality.

Mary Poppendieck was the first person to apply lean principles to software development. She is considered the founder of the LSD movement. She has summarized her findings in the book "Lean Software Development: An Agile Toolkit" [35]. Poppendieck emphasizes that lean development is not equivalent to shoddy work. On the contrary, lean development results in high quality products. She argues that the measure of an organization’s maturity is the speed with which it can respond reliably and repeatedly to customer requests. She declines the idea that the quality of a system can be measured by the comprehensiveness of its documentation [34]. In [35], Poppendieck lists seven principles of LSD. These principles should not be regarded as cookbook recipes for software development but could guide devising appropriate practices in a given environment.
3.10 Lightweight or agile software engineering methodologies

Eliminate waste

Anything adding no value to the customer is waste. Poppendieck has identified the following wastes in software development: partially done work (the inventory of a development process), extra processes (common in document centric development), extra features (develop only what customers want right now), task switching (do one and only one thing at a time), waiting (for instructions, for information), handoffs (lost knowledge) and defects (defects residing for a long time in the code). Reducing waste automatically results in the improvement of the flow of value from requested to delivered results.

Amplify learning

Programmers developing a system delivering business value have to learn many things. Obviously, they have to learn technical skills (programming language, framework, remote communication, etc.). However, they have to learn business processes, requirements gathering techniques and soft skills as well. LSD focuses on short iteration cycles delivering fast feedback information. The programmers get information very quickly about their work and can learn from successes or failures.

Decide as late as possible

Delay decisions as long as possible until more information is available. Especially, delay fundamentally irreversible decisions until known events are available, rather than insecure forecasts. Several possibilities exist to keep options open in software development, e.g.: sharing of partially complete design information, organizing worker-to-worker collaboration, developing a sense of when decisions must be made, developing a sense of how to absorb changes, performing refactoring and the use of automated Test Suits.

Deliver as fast as possible

This is the foundation of iterative development. Requirements change disproportionately with the amount of time. Therefore, once the customers decide what they want, the developing company should create the requested value as fast as possible. Thereby, the customers will probably not change their minds in the short time span between request and delivery.

Empower the team

The people factor is the most important element for a successful software project. People have to take responsibility for their work, must be motivated and must work efficiently with each other. However, this is only possible when the people are responsible for the outcome (the software release) and when they can respond directly to urgent requests without asking the management beforehand.

Build-in integrity

Poppendieck distinguishes perceived and conceptual integrity. Perceived integrity is achieved when the customer is pleased with the software. Detailed information flow from the users to the developers automatically results in perceived integrity. Conceptual
integrity describes the quality of the software architecture. The continuous and detailed information flow between the people working on a system yields conceptual integrity. A document-centric approach tends to undermine the vital personal communication flow. Unit and integration tests in contrast help to achieve integrity.

**See the whole**

Most software engineering theories recommend breaking the whole into individual parts and optimizing each one. Lean thinking assumes that this approach leads to a sub-optimized overall system. LSD suggests measuring the team’s performance instead of the individual’s performance. A team responsible for its own processes can improve the whole system and not just some parts of it.

### 3.10.5 Pragmatic Programming

Pragmatic programming is not a methodology in the stricter sense of the word. In 2000, Andrew Hunt and David Thomas have published the book “The Pragmatic Programmer: From Journeyman to Master” [11]. The ideas in this book have deeply influenced the agile movement. Hunt and Thomas were both among the originators of the Agile Manifesto. Following the approaches to software development suggested in this book is called Pragmatic Programming.

In the preface of the book, the authors write the following statement: “This book will help you become a better programmer.” This is a bold statement, and numerous books make similar claims to push sales. However, if a programmer is receptive, open and willing to change, this book will effectively enhance his or her skills considerably. The Pragmatic Programmer has obtained thoroughly positive feedback. The book was vaunted by most leading software developers.

Pragmatic Programming offers no distinct processes, phases or roles. Rather, it makes a programmer think about what he is already doing and looks at aspects from a different point of view. Covering all aspects of software engineering, Pragmatic Programming is a collection of common sense, good ideas and sound techniques. Each chapter of the book illuminates a part of the software development process and offers suggestions how to improve the process in day-to-day work. However, the philosophy behind the presented solutions is general and not limited to a certain situation. Stressing their main points in short tips (a total of 70), the authors distil the current part of the topic down to a single sentence.

These concise and simple sentences helped me to become a better developer. The tips are not abstract but can be applied in daily work. These tips do not simply express the experience of Hunt and Thomas. In fact, they concentrate the knowledge that has already been present in the developer community. This is actually the power of the tips.

Abrahamsson et al. [29] have extracted six main points of Pragmatic Programming:

- take responsibility for your work. Think of solutions instead of excuses.
• do not go along with bad code design. Fix bad design and inconsistency as you see them.

• take an active role in introducing change.

• your software should satisfy the customer, but know when to stop.

• constantly update your knowledge.

• improve communication skills.

Perhaps, this book’s key concept is to encourage a programmer to expand his own personal horizons to become aware of the available options. The broader and more diverse a programmer’s experience, the better he can choose the most convenient approach and suitable tools to solve a given problem in a given context. Programmers benefit more when they learn new programming languages, technologies and new software than when they perfect their existing knowledge.

Pragmatic Programming focuses on incremental iterative development, careful testing (using of automatic tests whenever possible), flexible solutions to withstand changes, user-centric design, actively searching for requirements and keeping specification and documentation at a reasonable level.
This chapter comprehensively describes the controlling software for the PERFORM A project. This software is the largest controlling software package I have realized and incorporates all major features of other software for bio-experiments I have programmed. Therefore, it serves as a standard model of software for bio-experiments. Before planning and implementing the PERFORM A software, I realized the controlling software for other bio-experiments including the European framework projects REFLEX and SLEEP. The experience I gained in these projects enabled me to create the large and complex PERFORM A software. If I had had to start the PERFORM A software without having experience with other controlling software beforehand, I would not have been able to complete such a large, robust and reliable software program.

Describing a large controlling software package is difficult. One really does not know where to start and stop. The large amount of source code contains an enormous amount of information. Presenting this information coherently in written form is not a simple task. Before the controlling software is described, the basic properties of the project as well as its scope must be described. A sound project overview containing the abstract concepts and describing the concrete realization lays the foundation for the understanding of the software system.

Therefore, to present an overview, the first section of this chapter introduces the basic properties of the project (goal of the experiment, framework, involved institutes, time period, budget).

The second section introduces the PERFORM A's non-software elements. It describes the exposure design, the field source and the sensors, the electronic equipment, the dosimetry and the toxicological biology.

Starting with the software architecture, the large third section discusses the controlling software. It presents the static as well as the dynamic software design, the interaction of the two designs, the GSM signal model as well as the implementation and the definition of a generic transfer function.

Throughout the chapter, the applied software engineering methodologies and principles are compared with the established software engineering techniques presented in the previous chapter. As a result, this chapter contains numerous cross references to the previous chapter.
4.1 Properties and scope

The PERFORM A project consists of seven animal experiments that should resolve whether long-term and low level RF exposure to mobile phone radiation cause a carcinogenic or co-carcinogenic response in mice and rats. The carcinogenic experiments clarify whether mobile phone radiation can induce cancer in healthy test animals. The co-carcinogenic experiments on the other hand examine if mobile phone radiation can amplify the growth of existing tumors in test animals.

Effectively, PERFORM A is the direct answer to the population’s growing concern about the health risk of mobile communication and the WHO’s demand for better scientific data. PERFORM A is the largest toxicological study ever performed in the context of the health risk assessment of mobile communication. The PERFORM A project was sponsored by the fifth Framework Program of the European Union. One of the fifth Framework Program’s (FP5) thematic is the “quality of life and management of living resources”. The objective is to support sustainable research and development for individuals, society and the environment. This program uses three specific approaches to achieve these objectives: meeting socio-economic needs, increasing European benefit and supporting European competitiveness. One key action of FP5 is to support research on reducing the harmful effects of environmental factors like electromagnetic radiation. The full title of the PERFORM A project is “PERFORM A – In vivo Research on possible Health Effects related to mobile Telephones and Base Stations”.

The following institutes are involved in the PERFORM A consortium: FhG-ITA (Hanover, Germany), RCC (Ittingen, Switzerland), ARCS (Selbersdorf, Austria), RBM (Colleretto Giacosa, Italy), IT'IS (Zurich, Switzerland), RCL/AUTH (Thessalonica, Greece) and Zhejiang University (China). The part of IT'IS in the PERFORM A project is to plan, realize and support the exposure systems. FhG-ITA, RCC, ARCS, RBM and the Zhejiang University perform the actual animal experiments. RCL/AUTH supports IT'IS in dosimetric issues.

FhG-ITA conducted two carcinogenic studies with B6C3F1 hybrid mice. The test animals were exposed for two hours a day, five days a week, for a period of two years to high frequency EMF at 902 MHz and 1747 MHz respectively. RCC conducted the same experiment scheme with Wistar rats. The studies at ARC and at Zhejiang University used DMBA (7,12 Dimethylbenz(a)anthracen) induced mammary tumor bioassays in female Sprague-Dawley rats to examine the influence of daily four hour exposure to 902 MHz for six months on tumor growth. The study at RBM was a replication of the Repacholi study of 1997. Repacholi et al. [36] found a three times higher lymph tumor rate in pim1-transgene mice after 18 months daily one hour exposure to 902 MHz.

As presented above, the PERFORM A project consists of two principal animal experiments: Experiments with mice and rats. These two experiment types require different exposure systems and hence require different controlling software. To keep things as simple as possible, this chapter only presents the rat exposure system and its controlling software. Naturally, the two systems have a lot in common; nevertheless, some significant differences exist, mainly in the number of exposure chambers and how the RF signal is regulated. No comparison between the mouse and rat setups will be accomplished. Therefore, the similarities and differences of both setup types will not be discussed.
Whenever this thesis mentions the PERFORM A setup or PERFORM A software, implicitly PERFORM A rat is meant.

4.1.1 Hardware components

One exposure setup consists of sixteen identical exposure units of cylindrical shape, below referred to as wheels, sixteen ventilation systems, four amplifiers, an amplifier water cooling unit, four monitoring units, a controlling computer, and a signal generation system. Each wheel allows whole-body exposure of up to 17 rats. The setup allows the simultaneous exposure of four rat groups: a high exposure group, a medium exposure group, a low exposure group and a sham group. A fifth rat group acting as a control group is kept in a separate room. The exposure dose ratios between the high, middle, and low exposure groups is 9:3:1. Figure 4.1 shows an overview picture of the PERFORM A setup installed at RCC, Füllinsdorf.

Figure 4.1: PERFORMA rat setup installed at RCC, Füllinsdorf. At the lower part to the right and left, the picture shows the exposure wheels. They are placed in a cleanroom especially designed for animal experiments. The air conditioning system regulates the temperature and humidity to guarantee stable conditions.

The wheel design allows the exposure of a high number of rats in the same setup with compact dimensions, provides a high efficiency of absorbed RF-energy in the animals and offers reasonable exposure homogeneity. Exposure homogeneity refers to the fact
that all rats in an exposure wheel obtain essentially the same exposure dose, independent of the rats' size and weight. This is only possible when rats in adjacent wheel positions do not influence the field strength in a given position. Exposure wheels based on a novel concept using H-field polarization have turned out to provide the best homogeneity.

An exposure wheel consists of a circular cascade of 17 sectoral waveguides. Two parallel circular metallic plates with 120cm radius at a distance of 30cm encircle the waveguides. In the middle of the wheel is a quarter loop antenna, uniformly irradiating the sectoral waveguides. The wheel constitutes a resonance structure for the emitted electromagnetic waves. A reflective closing at the outer radius reflects the waves back to the centre of the wheel; a fence of thin stainless steel cables serves as the reflecting closing. This open closing provides the test animals with laboratory light. 28 cables with 10 mm spacing constitute the closing fence. This construction enables access to the sectoral waveguides and tenses the cables for good electrical contact and exact planes. Figure 4.2 shows a close-up view of a wheel's front and rear sides. Figure 4.3 shows the ventilation system of two of the wheels (left) and the reflecting closing consisting of stainless steel cables (right). Figure 4.4 shows the amplifier water cooling system.

Figure 4.2: Front (left) and rear (right) sides of an exposure wheel. The front side shows the seventeen sectoral waveguides, aligned perpendicularly and between the two circular metallic plates. Pulling the white plastic bars activates the sliding mechanism for loading and unloading the animals. The sectoral waveguides have metallic caps to avoid electromagnetic leaking. The rear side shows the three air hoses and the black box in the middle of the wheel. Inside the black box is the electronics for the field and environmental sensors. The blue wire is the RF cable. The RF signal goes into the black box, where a tuner tunes the signal to the wheel's impedance. The black cable at the bottom of the box contains the sensor signal cables and the current feed for the sensor electronics. The exposure chamber and the mobile supporting construction constitute a 2° angle.

Restraining tubes commonly used for inhalation studies keep the animals in a stable position. Stoppers at one end fix the animals. The waveguide cavity holds the tubes. A
comfortable sliding mechanism allows easy access to the sectoral waveguides and permits effortless loading and unloading of the animals.

A ventilation system consisting of a ventilator box, three air hoses and a ring channel for each wheel supplies each animal with fresh air. This construction guides airflow from the ventilator to the front side of each wheel's 17 tubes and ensures continuous airflow passing through the breathing zone of the animals.

The exposure chamber and the supporting construction constitute a 2° angle, avoiding the inflow of excrements toward the animal's nose. The supporting construction itself consists of a metallic frame with plastic rollers for easy mobility. All metallic material used for the wheel is either stainless steel or gold-plated copper. The synthetic materials are either polycarbonate, PVC or POM. The animals have direct contact only with polycarbonate material.

Figure 4.3 Close up of the ventilation system of two wheels (left). Three air hoses per wheel route fresh air in the ventilation ring, where the air is equally distributed and flows to the animals' noses. Stainless steel cables (right) form a fence acting as a reflecting closing for the electromagnetic field.
4.1.2 Electronic equipment

The PERFORM A exposure setup consists of four essentially independent exposure groups for the four exposure dose levels high, medium and low. Each group has designated electronic equipment and four exposure wheels. Some of the electronic equipment such as the signal unit, the amplifier water cooling system and controlling computer are the same for the entire setup. Below is a listing of the parts included at the three levels wheel, exposure group and setup.

Equipment per wheel:

- two E-field sensors (ACC ACSP 2663NZC15), controlling electronic and current feed
- two Schottky diodes (minimum sensitivity: K-Factor: 2000 mV/mW, flatness: +0.3 dB)
- O₂ sensor (Pewatron FCX-MV-CH, accuracy: +/- 0.5% O₂), controlling electronic and current feed
- temperature + humidity sensor (E + E Series Ee06, accuracy temperature: +/- 0.5° C, accuracy humidity: +/- 5 %), controlling electronic and current feed
- tuner (Maury Microwave Triple Stup 1878B, frequency range: 0.8 ... 4.0 GHz)
- quarter loop antenna (homogeneity: 0.2 dB)
- ventilation system consisting of three air hoses and a controlling electronic
4.1 Properties and scope

Equipment per exposure group:

- datalogger and switching unit (Agilent 34970A)
- digital control unit DCU (Schmid & Partner Engineering AG)
- high power RF amplifier and a high speed, high power switch (LS Elektronik)
- power distribution boxes for amplifier and ventilators

Equipment per complete setup:

- digital RF generator (Rhode & Schwarz SMIQ02B) or arbitrary function generator (Agilent 33120A) and RF signal generator (Rhode & Schwarz SML02), depending on signal generation mechanism
- RF divider
- water cooling system, composing a closed circuit through all amplifiers
- room light indicating the system's state
- controlling computer (Dell, P3 1000 MHz, Riser card Gx100, 128 MB SDRAM, 20 GB IDE 7200 hard disk, 48x IDE CD-Rom, MS Intellimouse 2 BTN PS, MS Windows 2000 Service Pack 1)
- 12.1" LCD touch monitor (ELO 1226L)
- controlling software PerformRat Version 4.01 (IT'IS)

Figure 4.8 shows graphically the compact PERFORM A setup scheme.

4.1.3 Signal generation mechanism

Depending on the signal generation mechanism, the digital GSM/DCS signal generator Rhode & Schwarz SMIQ02B or a combination of the or arbitrary function generator Agilent 33120A and the RF signal generator Rhode & Schwarz SML02 generate the GSM exposure signal. The SMIQ02B is able to independently generate the desired GSM signal. For the alternative signal generation mechanism, the modulation signal of the function generator 33120A modulates the carrier RF signal of the RF generator. This modulated RF signal corresponds to the desired GSM signal as well.

Figure 4.5 shows a schematic description of the RF signal distribution. The RF divider splits the signal into four power-equivalent signal parts. Each part is designated to one exposure group. Each exposure group has an amplifier, which amplifies the signal. The DCU blanks the amplifier input at regular time intervals to suppress undesired signal frames and to trigger the high-power switch. This blanking mechanism generates four GSM signals out of a simple frame structure signal. The RF signal generator thereby acts as a frame trigger. The GSM signal will be discussed later in detail.
A GSM signal contains eight timeslots, whereby only one timeslot actually contains signal components. The initial RF signal contains signal components in every second timeslot (T 0, 2, 4, 6). However, a real GSM signal only has signal components in one timeslot. Therefore, the signal at the amplifier's input has signal components for four real GSM signals. A high-speed, high-power switch allocates the different timeslot signals to the four amplifier outputs, so each wheel gets the signal component of one timeslot. This technique generates four GSM signals out of a single frame structure signal and allows power timesharing of a single amplifier and therefore a reduction of the number of amplifiers by a factor of three.

The DCU implements the power distribution algorithm and controls the amplifier switch via two digital address lines. Switching the RF switch is only possible in cold state (i.e., no power on the input line). Therefore, to guarantee cold switching, the DCU checks the amplifier output power and blanks the amplifier input while switching.

### 4.1.4 Data acquisition

Figure 4.6 shows the controlling electronic for two exposure groups. The datalogger (Agilent 34970A) collects the data from the sensors, the DCU, the air ventilation systems and the amplifier. The environmental sensors (oxygen, humidity, temperature) measure the conditions inside the ventilation ring caps on the backside of the wheels. The measured values represent the actual conditions for the animals. The water resistant black box on the back of the wheels houses the controlling electronic of the environmental sensors. Two field sensors per exposure wheel located in two different sectoral waveguides measure the field strength during the exposure. The DCU, the air ventilation systems and the amplifier send controlling signals to the datalogger.
The datalogger not only collects data, but also applies controlling signals to the electronic devices. Two digital signals manage the DCU, one analogue (power regulation) and one digital signal control the amplifier, three digital signals manage the room light, and one digital signal switches the main power for the air ventilation system and for the amplifier.

The controlling software on the Dell PC collects the data from each datalogger and applies controlling commands. Running continuously, the software collects oxygen, humidity and temperature data at all times, even when no exposure takes place. During the actual exposure, the software collects field strength data quasi-continuously. In the case of a mismatch between the actual field strength and the target field strength, the exposure system regulates the field via the output power of each amplifier and thus maintains a stable electromagnetic field inside each exposure group. All software/hardware communications run via GPIB bus.

A watchdog mechanism implemented in all DCUs checks if the controlling software is working properly. During proper operation, the software sends an alternating digital signal to the DCUs. In the case of a software hang-up, the DCUs no longer receive alternating digital signals, raise an alarm (seen as a red light emitting diode (LED) on the
DCU) and blank the amplifier input. The mechanism prevents that the wheels obtain uncontrolled RF power.

4.1.5 Field sensors

Monopole field sensors measure the field strength in the exposure wheels. The monopoles consist of a silver plated 0.9 mm copper-clad wire as the center conductor of a semi-rigid RF coax cable and a plastic shell to protect the sensors form deformation during cleaning activities (Figure 4.7). The optimal radial location for a sensor is in a local E-field maximum. Simulations and measurements revealed the optimal sensor positions to be in the sectors A0 and A2 (Each wheel is divided into 17 sectors; each sector obtains a label from A0 to A16, in the E-field maxima at radius $y=318$ mm.

Figure 4.7: Uncovered E-field monopole sensor with Schottky diode and BNC connector

Figure 4.8 shows a compact PERFORM A setup scheme containing all described elements. Red lines indicate the RF path, green lines the GPIB connections, blue lines digital signals and black analog signals.
Figure 4.8: Compact PERFORM A setup scheme. The complete exposure system contains four exposure groups (the corresponding software object is ExposureUnit). Each exposure group in turn contains four exposure wheels (the corresponding software object is Wheel). Special system configurations may contain less exposure groups and less exposure wheels. The fundamental unit is the exposure group or exposure unit. Each exposure is an independent measurement and regulation unit.
4.2 Controlling software

4.2.1 Applied software engineering methodology

No single development methodology was used during the entire development process. The parts of every methodology especially suited to develop a prototype measurement system were applied. As a result, the applied software engineering methodology was a mixture of heavyweight and agile methodologies. Heavyweight as well as agile methodologies provided valuable procedures to develop the prototype measurement system. The consequent use of UML modeling diagrams has proven to be valuable to discuss the software design with various software engineers. Indeed, the software system was not completely modeled before the implementation started. This would have been a bad strategy, since the design of a prototype system is likely to change in the development process. The central strategy to develop the prototype system was to find simple solutions for the given problems. Only known requirements were considered and only possible requirements were omitted. This approach was conforming to the Extreme Programming maxim to find simple solutions. The development process was strongly iterative with short iteration cycles. This was typical for a prototype system, since some requirements emerged only in the development process. At the end of every iteration cycles, the source code was refactored to make it simpler and clearer. This was very important. Only a consequent refactoring kept the software system maintainable.

4.2.2 Computer platform & programming language

Before starting to implement a system, the computer platform and the programming language must be selected. These two decisions are extremely important, because they affect the system’s performance, robustness, distributability and maintainability. Therefore, careful examination of the advantages and disadvantages of computer platforms and programming languages is vital for the subsequent implementation process. Before comparing and assessing computer platforms and programming languages, non-functional system requirements must be identified. Based on these requirements, the different configurations can be evaluated, whereby every configuration must fulfill the non-functional requirements. A closer look of the PERFORM A exposure system revealed the following non-functional requirements:

- system stability. The software has to run for 2 two years, possibly without interruptions.
- hardware communication. Support for GBIB communication.
- storage capacity. 20 gigabyte of disk storage capacity for experiment data.
- multithreading functionality
- internet access. Easy internet access for sending data files
- secure and encrypted data format
In addition to the non-functional system requirements, the skills and preferences of the programmer have to be considered as well to guarantee a smooth implementation process. Actually, the skills of the programmer are probably even the most important factor. Learning a new computer platform and a new programming language takes a considerable amount of time. However, in the case of the PERFORM A project, this time was not available, because the whole project had to be realized under extreme time pressure. If a programmer has to implement a new system with unclear and changing requirements from scratch and under great time pressure, he is at least glad to use a familiar operating system and programming language he can already master.

After evaluating the non-functional system requirements as well as the skills of the designated programmer, Microsoft's Windows operating system turned out to be the optimal computer platform. In the beginning, the project leader was skeptical about using the Windows operating system. He had serious stability concerns, because he thought that a Windows computer would not run for two years without a system crash. However, the switching off of unneeded services normally running in the background and the removal of unused components considerably improves the stability of the Windows operating system. At last, the PERFORM A team decided to use a slimmed version of Windows 2000 as the operating system.

After selecting the operating system, Microsoft's Visual C++ and National Instrument's LabView were the two programming languages suitable for the implementation of the PERFORM A software. As the designated programmer, I was skeptical about the suitability of LabView as a programming language for large systems. I was afraid that a large LabView project with hundreds of visual icons would become at least as complicated to manage as a corresponding C++ program. Additionally, I thought that some requirements such as an encrypted data format or the generation of a synthetic GSM signal would be very complicated or even impossible to implement with LabView. Thus, LabView did not offer me the flexibility I wanted. Therefore, I decided to use C++ instead of LabView as the programming language.

### 4.2.3 Software architecture

The computer hardware and software for the PERFORM A project not only includes the experiment computer and the controlling software, but also file servers, mail servers, an http server with a mail script and an evaluation software running on a particular evaluation computer. Figure 4.9 shows a simple Visio diagram of the system architecture of PERFORM A.

The controlling software PerformRat.exe runs on the experiment PC, which is located in the bio laboratory. Operating day and night, this software controls the animal experiment. PerformRat.exe collects and evaluates experiment data and monitors and adjusts the field strength during actual exposure. Whenever a system error is detected (parameter
out of tolerance, hardware communication failure, etc.), PerformRat.exe triggers an http request on an IT'IS web server. This request activates the Perl script Mail.pl on the web server and passes basic error description parameters to the script. Mail.pl extracts the error description parameters, generates an error mail message and sends the mail to a set of e-mail addresses. The notification mechanism operates like an e-mail newsletter. People involved in the maintenance of the PEFORM A setup register their e-mail address in the script. Thereafter, they get an error e-mail whenever the controlling software detects a system failure.

Naturally, the controlling software PerformRat.exe could send directly error e-mails without using an http request. However, such a solution would be inflexible, because adding and removing e-mail addresses would only be possible with direct access to the experiment computer. Since the Experiment PC in the bio laboratory is hidden behind a firewall, no direct access from the IT'IS facilities to the Experiment PC is possible. Additionally, the strict experiment protocol permits people working in the bio laboratory to make changes in the controlling software. Consequently, to add or remove e-mail addresses, a person from IT'IS would have to travel to the bio laboratory. However, some laboratories are far away from the IT'IS facility in Zurich, and the resulting costs and expenditure of time to add or remove addresses would be disproportional. Once at the bio laboratory, an additional difficulty would be to access the experiment PC to make the required setting changes, because the experiment PC is placed in a cleanroom. Working in a cleanroom demands high hygiene standards, and one must follow a strict protocol. Therefore, people working in a cleanroom generally do not appreciate when other people enter the clean area.
In contrast, the error notification solution using an http request trigger mechanism is much more flexible. Anyone who knows the password of the PERFORM A account can add or remove e-mail addresses from any PC with an internet connection. This system flexibility has proved to be very valuable. Over the course of the PERFORM A project, the
The controlling software continuously saves the measured data and the applied hardware commands in a data file. Every night, PerformRat.exe closes the current data file and creates a new file. Therefore, for every day a new data file is generated. PerformRat.exe makes two backup copies of the just closed data file: one on a local file server in the bio laboratory and one on an IT'IS file server. To perform the backups, PerformRat.exe establishes FTP connections with the remote servers and sends the data file. The data file for IT'IS has to be sent over the public internet. To guarantee data security, this data file is encrypted using a standard RC algorithm.

Running on an IT'IS PC, the evaluation software PerformRatAnalyze.exe analyzes the experiment data and automatically generates experiment reports. PerformRatAnalyze.exe regularly checks the file server housing the experiment data files for new data files. If the software finds a new unprocessed data file, the software decrypts the data file, opens it, extracts all exposure settings and data and generates a report in pdf format containing all relevant exposure data and graphs. Afterwards, PerformRatAnalyze.exe automatically sends the report to the study director of the corresponding bio laboratory. Finally, PerformRatAnalyze.exe prints out the report on an IT'IS printer. All printed reports are then archived in ring binders for preservation.

Figure 4.10 again shows the PERFORM A system architecture, but this time as a UML deployment diagram. Having both diagrams side by side, the comparison of their significance is possible and can be further examined.

The three-dimensional cube with the node's name at the top of the cube represents a node. A node symbolizes either a physical computer or a hardware device. Each rectangle with two small rectangles protruding from its side centered inside the nodes represents a software component. The arrowed lines show the dependencies of the components. The controlling software for example is dependent on the Visa32.dll library. This library offers basic GPIB commands. The evaluation software PerformRatAnalyze.exe depends on four additional dlls: zip.dll for easy file compression, PEGRP32a.dll for chart generation, Visualsoftmail.dll for the e-mail handling and pdfrw.dll for pdf formatting. Arrowed lines connecting components of different nodes illustrate how the components communicate with each other. The text in between the lines describes the nature of the communication.
Clearly, the deployment diagram shows a lot of information within a small space. Presenting the physical computers and the software components running on the corresponding computers as well as the dependencies of the software components, the deployment diagram offers a compact way to obtain an overview of the system architecture. However, the observer must have a basic understanding of software architecture and must be familiar with the semantics of the deployment diagram to obtain an impression of the architecture. People with little knowledge about software architecture will most probably have difficulties reading and interpreting a deployment diagram. Most symbols in the deployment diagram are very abstract. For example, consider the symbol of a node. The association that a three dimensional cube symbolizes a computer is not evident. Additionally, some symbols are pretty squiggled and bother the optical coherency, like for example the symbol for a component. Considering its purpose to offer an overview of a system for a broad public, the deployment diagram is almost too complicated. This diagram should offer a quick overview of the system architecture not only for experts, but also for managers. However, the deployment diagram cannot meet this claim. Nevertheless, the deployment diagram is an excellent tool for software engineers, mainly because of its standardized form.

In contrast to the deployment diagram, the Visio diagram of the PERFORM A system architecture contains less information. Showing familiar symbols and pictures, the Visio diagram is easier to understand. No modeling language has to be learned to comprehend the diagram. Everyone looking at the diagram obtains a first expression of the system. Therefore, this diagram style is more suitable to present a system to a wider audience. Particularly in project meetings with participants with different backgrounds, deployment diagrams should not be used, instead, more simple diagrams should be preferred.

4.3 Design of the controlling software PerformRat.exe

After presenting the complete PERFORM A system architecture, this chapter thoroughly describes the PERFORM A system's core, the controlling software PerformRat.exe. Running on the experiment PC in the bio laboratory, PerformRat.exe controls the exposure system.

PerformRat.exe consists of two main parts. One part remains unchanged at runtime, whereas the other part continuously changes. The steady part describes the exposure system's static structure; the unsteady part expresses the system's behavior. This chapter's first section describes the static structure. Afterwards, the second section illustrates the dynamic behavior. Finally, the two sections are put together and their interaction mechanism is presented.

4.3.1 Static design

Modeling the exposure system's static structure is straightforward. A clear mapping between instruments, hardware and physical setup structure exists. Each object of the real world is assigned to a software object. The software class Setup represents the complete physical exposure system. Setup contains all parts of the exposure system as data mem-
The organization of the Setup members reflects the organization of the physical exposure system.

**Setup**

The fundamental static software object is Setup. Containing all parts of the exposure system as data members, Setup represents the complete physical exposure system. Setup has a very similar structure as the physical exposure system.

The physical exposure system consists of numerous items, which are arranged in a whole-part pattern. The whole-part pattern contains parent and child items. A parent item can contain other items, the child items. Each of these child items in turn can contain their own child items. Each item has exactly one parent item and can have several child items. The item on the top of the hierarchy does not have a parent item, whereas the items at the bottom of the hierarchy do not have child items.

The PERFORM A exposure system has the following hierarchical structure: An exposure wheel contains several sensors and a ventilation system. An exposure unit contains four wheels, a datalogger, a DCU, an amplifier and power distribution boxes. The whole exposure system contains four exposure units, a signal unit, an amplifier cooling unit and a room light.

This whole-part physical structure can be mapped one-to-one into an equivalent software model. As in the real world, parent and child objects exist in the software model. In this child-parent model, the objects compose a "has-a" relationship. Each object can have a parent object and several child objects. Parent objects contain their child objects as data members. Setup is on top of the hierarchy and does not have a parent object. However, Setup has several child objects; Setup contains these child objects as data members. The child objects in turn can have their own child objects as data members and so forth, until the child objects do not have child objects of their own. This data arrangement is commonly known as a tree structure.

Actually, the composite software design pattern [19] assembles objects to tree structures to represent whole-part hierarchies. Allowing client objects to treat single objects and compositions of objects equally, this pattern offers great flexibility. However, the composite pattern is only suitable if the objects in the data structure have a similar semantic. Since in PerformRat.exe the various static objects do not have similar semantics, the composite pattern is not very suitable. Additionally, using the composite pattern for large and complex data structures enhances the complexity of the code.

Therefore, following the Extreme Programming approach to use simple designs whenever possible, a conventional software model was used to describe the exposure system. Objects simply contain other objects as data members, and the objects do not have a common base class.

The following listing shows the simplified C++ class declaration of Setup. The declaration shows the context relevant methods and data members. Setup is not complete and would fail to compile.
Like most class representing hardware in PerformRat.exe, Setup is derived from Streamable. Dr. Hansueli Gerber, senior programmer at SPEAG, is the original author of Streamable. He kindly developed this class for the demands of PerformRat.exe. Streamable manages the persistent saving and loading of object data. Following the object-oriented maxim that every object is responsible for itself, every object must be able to save its content to disc and to retrieve its data from disc. Every class derived from Streamable has to implement Serialize(std::ostream& out) and Deserialize(std::istream& in). Serialize() writes the object's data into the output data stream, Deserialize() reads the object's data from the input data stream.

Essentially, Setup has two data members: m_SignalUnit and m_ExposureUnits. m_SignalUnit is a pointer to a SignalUnit and m_ExposureUnits is a map the of type ExposureUnit. m_SignalUnit represents the signal generation unit and m_ExposureUnits represents the exposure units.

The standard PERFORM A exposure system consists of four exposure units. However, special pre-studies have only used one, two or three exposure units. The software model must be capable of handling a changing number of exposure units. Therefore, Setup cannot just contain four instances of ExposureUnit, but must be prepared to hold a changing number of ExposureUnits. To make this possible, the ExposureUnits are stored in a map. The map allows the adding and the removing of ExposureUnits, according to the actual need of the experiment. Before a certain experiment starts, m_ExposureUnit is filled with the number of ExposureUnits used for this experiment.

SetSignalUnit(SignalUnit* signalUnit) assigns the signalUnit to m_SignalUnit.

The PERFORM A setup has two different signal generation mechanisms. Depending on the signal generation mechanism, a digital RF signal generator alone or a combination of an arbitrary function generator and an RF signal generator generate the GSM exposure signal. The software model must be able to support both signal generation mechanisms,
without making the source code too complicated. The programmer should not have to
care at multiple source code locations which signal unit actually must be used. To
achieve this goal, a facade hides the real signal generation mechanism. SignalUnit acts as
this facade. The facade encapsulates and thus hides the real signal generation mecha­
nism. Client code just can use SignalUnit without bothering whether SignalUnit actually
represents a digital signal generator alone or a combination of an arbitrary function gen­
erator and an RF signal generator. Before using it for the first time, the programmer must
configure SignalUnit with the corresponding signal generator objects. Whenever a
method of SignalUnit is called, SignalUnit forwards the request to its generator objects.
The detailed structure of SignalUnit will be discussed later.

As the name suggests, Initialize() initializes the basic data members of Setup and in turn
calls the same method for its child objects. Each object initializes itself. Setup does not
take care of the initialization process of its child objects.

```cpp
bool Setup::Initialize()
{
    bool ok = true;
    ok &= m_SignalUnit->Initialize();

    // iterate through all exposure units and call their Initialize() methods
    //
    for (std::map<int, ExposureUnit, std::less<int> >::iterator unit = m_ExposureUnits.begin();
        unit != m_ExposureUnits.end(); ++unit) {
        ok &= unit->second->Initialize();
    }
    return ok;
}
```

CheckHardware() tests the exposure system's hardware. It returns true if all connections are
established and all hardware devices are working properly. The implementation is similar
to Initialize(). CheckHardware() calls the Check() methods of the child objects.

```cpp
bool Setup::CheckHardware()
{
    bool ok = true;
    ok &= m_SignalUnit->Check();

    // iterate through all exposure units and call their Check() methods
    //
    for (std::map<int, ExposureUnit, std::less<int> >::iterator unit = m_ExposureUnits.begin();
        unit != m_ExposureUnits.end(); ++unit) {
        ok &= unit->second->Check();
    }
    return ok;
}
```
Log(Log& log) writes the values of the basic data members of Setup into a log file. Next, Log() calls the Log() method of the child objects. The child objects log their basic data members and call the Log() methods of their child objects and so forth.

Edit() allows the graphical editing of the parameters of Setup. Setup is responsible for its parameter editing. Edit() creates a setup dialog class and displays the parameters of Setup. SetupDialog is a combination of a Setup and a simple Windows dialog. Therefore, SetupDialog is derived from Setup and CDialog. CDialog is part of the Microsoft Foundation Classes (MFC) and describes a dialog window. Edit() passes the this pointer to the copy constructor of SetupDialog. The copy constructor of SetupDialog in turn calls its Setup copy constructor. This mechanism creates a dialog class containing a copy of the original data of Setup. SetupDialog::DoModal() creates the dialog and displays it on the screen. If the user closes the dialog with the ok button, SetupDialog is assigned to the dereferenced this pointer. Since SetupDialog is derived from Setup, SetupDialog has an "is-a" semantic with Setup and the assignment is possible. After the assignment, the data members of Setup have the same values as the corresponding SetupDialog data members. In short, Edit() creates a SetupDialog containing a deep copy of the Setup data. If the user quits the dialog with ok, Setup adopts the changed data. This mechanism allows simple data transfer from an object to a dialog and vice versa.

All objects in PerformRat.exe that have to be editable implement the same mechanism. To configure PerformRat.exe properly, the software user has to enter about 2000 parameters in dialog windows. This editing mechanism allows a compact, localized and efficient handling of the data input. It is optimal for editing parameters of tree structures, since every object of the tree structure implements its own Edit() method. Therefore, new objects can be added very easily into visual tree structures. The following listing illustrates the editing mechanism.

class SetupDialog : public Setup, public CDialog
{
public:
    SetupDialog(const Setup& setup, const std::string& title);
    ...;
};

SetupDialog::SetupDialog(const Setup& setup, const std::string& title) :
    Setup(setup)
{}

bool Setup::Edit()
{
    bool ok = false;

    // Create SetupDialog and pass the this pointer to the constructor
    // SetupDialog setupDialog(*this);

    // create the dialog and display it on the screen

4.3 Design of the controlling software PerformRat.exe

Perhaps the reader will miss a room light and an amplifier water-cooling object in Setup. The physical setup has exactly one room light, one amplifier water cooling system, and four exposure units, whereby each exposure unit contains one datalogger. The room light and the cooling system are connected to one datalogger. The software model has to map this structure; however, there is a problem modeling this structure. In the software, the room light and the cooling system are closely related to one datalogger. Therefore, the datalogger, the room light and the cooling system have to be together in an exposure unit. This results in the fact that although in reality only one of the four exposure units has a room light and a cooling system, in the software every exposure unit has a room light and a cooling system. To solve this conflict between reality and the software model, one datalogger acts as the master datalogger. Only the master datalogger communicates with the room light and the cooling system. The software model therefore contains four room lights and four cooling systems, but effectively used are only one room light and one cooling system.

ExposureUnit

ExposureUnit represents a physical exposure unit. The next listing shows the class declaration of ExposureUnit.

```cpp
// ExposureUnit models a physical exposure group or an exposure unit. Like in reality,
// each exposure unit is an independent measurement and regulation unit
//
// class ExposureUnit : public CStreamable
//
//   public:
//     bool Initialize();
//     bool Check();
//     bool Log(Log& log);
//     bool Edit();
//     virtual bool Serialize(std::ostream& out);
//     virtual bool Deserialize(std::istream& in);
//
//   Result Evaluate(Id::Evaluate::Id type);
//   bool SetSwitchPosition(bool on);
//   bool SetLightColour(Light::LightColour lightColour);
//   bool SetAmplifierPower(bool on);
//   bool TriggerWatchDog();
```
ExposureUnit contains child objects representing hardware devices (m_Amplifier, m_DataLogger, m_Dcu, m_CoolingSystem, m_Light, m_Fan), m_Wheels representing the exposure wheels and m_AverageFieldPowerDeviation collecting the field power deviations.

Principally, an exposure unit can contain a variable number of wheels. For example, the exposure units at RCC (Basel) have four wheels; the exposure units at ARCS (Seibersdorf) in contrast have only three wheels. This implementation using a variable number of wheels can handle the different exposure systems.

Similar to Setup, ExposureUnit implements basic administration and status methods.

Evaluate(Id::Evaluate::Id type) is the central data evaluation method. The method uses an id argument to indicate what exactly should be evaluated. Evaluate() returns an object of type Result holding the evaluation’s result. Result is a basic type used at numerous locations in the code. Result is especially designed to accept multiple return values of a method. Result implements operator= and operator+= and thus allows natural semantics (Objects of type Result can be combined in a natural way.). The following listing shows the relevant section of Evaluate().

Result ExposureUnit::Evaluate(Id::Evaluate::Id type)
{
    Result res(true, true);
    switch (type) {
        case Id::Evaluate::Field: {
            res = EvaluateField();
            break;
        }
        case Id::Evaluate::All: {
            res = EvaluateAll();
            break;
        }
    }
    return res;
}
SetSwitchPosition(bool on), SetLightColour(Light::LightColour lightColour), SetAmplifierPower(bool on) and TriggerWatchDog() are sample methods for hardware communication. ExposureUnit forwards the requests to the corresponding hardware object.

GetFieldPowerDeviationFromTargetdB() returns the ratio measured field power / target field power in dB of the whole exposure unit. This value is used to adjust the amplifier gain. ExposureUnit stores the deviations of each measurement in m_AverageFieldPowerDeviation. m_AverageFieldPowerDeviation allows several averaging and smoothing operations depending on the desired regulation mechanism.

ExposureUnit is a large and complex class. The presented class declaration shows only the very most important methods.

Wheel

Wheel represents a physical exposure wheel. The following listing shows the class declaration of Wheel.

```cpp
// Wheel represents a physical exposure wheel. Like in reality, Wheel houses field and
// environment sensors.
//
class Wheel : public CStreamable
{
public:
    bool Initialize();
    bool Check();
    bool Log(Log& log);
    bool Edit();
    virtual bool Serialize(std::ostream& out);
    virtual bool Deserialize(std::istream& in);

    Result Evaluate(Id::Evaluate::Id type);
    double GetFieldPowerDeviationFromTargetdB();

    double GetUpperSensorFieldPower();
    double GetLowerSensorFieldPower();
    double GetTemperature();
    double GetOxygene();
    double GetHumidity();

private:
    FieldSensor m_UpperFieldSensor;
    FieldSensor m_LowerFieldSensor;
    LinearVoltDcSensor m_TemperatureSensor;
    LinearVoltDcSensor m_HumiditySensor;
    LinearVoltDcSensor m_OxygeneSensor;

    FilterMovingAverage m_AverageFieldPowerDeviation;
    Result m_State;
};
```
Wheel contains child objects representing sensors placed in the exposure wheel (m_UpperFieldSensors, m_LowerFieldSensor, m_TemperatureSensor, m_HumiditySensors, m_OxygenSensors), m_AverageFieldPowerDeviation collecting the field power deviations and m_State representing the state.

Setup, ExposureUnit and Wheel are organizational units and do not implement much functionality. Rather, they group logical units and distribute tasks. They forward requests they cannot handle themselves to one of their child objects that actually can perform the task. In other words, most methods of these objects call methods of their child objects. To perform a task, these objects have to know the functionality of the child objects very well. Consequently, they have to know the type of their child objects.

Setup, ExposureUnit and Wheel all implement a set of methods with the same signature. They offer the interface but forward the request to one of their child objects. This mechanism distributes requests from the top of the hierarchy to the place which can really handle the requests without skipping a hierarchy level. This is a common principle in nested data structures. The interfaces should be defined as high as possible in the hierarchy level. The implementation in contrast should be as low in the hierarchy as possible. The same principle is also used in inheritance structures.

The next section describes the functionality of some core objects representing hardware devices and the interactions of these objects.

**SignalUnit**

SignalUnit represents and encapsulates the signal generation mechanism. Figure 4.11 shows an UML class diagram of SignalUnit.

SignalUnit declares basic methods to initialize hardware and to generate signals. Signal encapsulates the signal settings such as power, frequency and modulation type. These methods of SignalUnit are purely virtual. Therefore, SignalUnit cannot be instantiated. AnalogSignalUnit and DigitalSignalUnit are both derived from SignalUnit. They implement all methods declared in SignalUnit. Consequently, they can be instantiated.

DigitalSignalUnit holds a pointer to a DigitalRFGenerator and forwards all requests to this pointer. This structure represents the case when a single digital RF generator creates the signal.
Figure 4.11: UML class diagram of SignalUnit’s static structure

AnalogSignalUnit holds a pointer to an RfGenerator (m_RfGenerator) and to a FunctionGenerator (m_FunctionGenerator). AnalogSignalUnit forwards requests to generate a signal to these two pointers. This structure represents the case when a combination of RF generator and function generator creates the signal. For example, consider the creation of a GSM signal. At first, AnalogSignalUnit commands the FunctionGenerator to generate the modulation signal. For that purpose, AnalogSignalUnit calls the appropriate functions of m_FunctionGenerator. Next, AnalogSignalUnit creates the RF signals and modulates them with
the modulation signal. The following listing shows a simplified code snippet from
AnalogSignalUnit::SetSignal() :

```cpp
bool AnalogSignalUnit::SetSignal(const Signal& signal) {
    bool ok = true;
    ok &= m_FunctionGenerator->SetModSignal(signal.m_FilePath);
    ok &= m_FunctionGenerator->SetFrequencyHz(signal.m_FrequencyHz);
    ok &= m_FunctionGenerator->SetPowerVolt(signal.m_PowerVolt);
    ok &= m_FunctionGenerator->SetOffsetVolt(signal.m_OffsetVolt);
    ok &= m_RFGenerator->SetPowerDbm(signal.m_DbM);
    switch (signal.m_ModulationType) {
        case signal.PULS: {
            ok &= m_RFGenerator->SetPulsModulationExtern(true);
            break;
        }
        case signal.AMDC: {
            ok &= m_RFGenerator->SetAmDcModulationExtern(true);
            break;
        }
    }
    return ok;
}
```

RfGenerator is an abstract base class for concrete RF generators. Concrete RF generators are
derived from RfGenerator; they represent a real RF generator. For example, RfGeneratorRsSml02
represents the generator Rhode & Schwarz RML02. AnalogSignalUnit holds a
pointer to an abstract RfGenerator. Therefore, AnalogSignalUnit can operate with various
concrete RfGenerators and does not know the concrete type of its RfGenerator. This imple-
mentation is very flexible. If a new RfGenerator is used in the exposure system, the pro-
grammer simply has to create a new class derived from RfGenerator representing the new
generator. In the initialization phase, this new RfGenerator has to be assigned to the RfGen-
erator pointer of AnalogSignalUnit. That is all to be done to add a new RF generator. No
other adaptations have to be made, and no client code has to be changed. Client code
using the functionality of AnalogSignalUnit does not have to know which RfGenerator is
used. This modeling principle is also used to represent function generators.

Now, consider again the Setup object. Setup has a data member of the type SignalUnit*.
This pointer can point to an AnalogSignalUnit or a DigitalSignalUnit. Consequently, Setup
and clients of Setup do not know what kind of signal unit is used. Additionally, they do
not know which concrete generators will actually generate the signal.

The design of SignalUnit is a result of the Extreme Programming's maxim to use simple
designs and to refactor source code. SignalUnit's structure did not emerge from a single
design activity, but evolved over the course of time. The first implementation of Per-
formRat.exe was only able to generate the GSM signal with a specific digital signal gen-
erator. However, suddenly the desire emerged to use a different generator type. Thereby I
was forced to change the software design. Later, the requirement appeared to use an ana-
log signal generation mechanism as well. Again, the software design had to be changed,
and the presented design arose. I am convinced that most designs evolving from real needs are more stable and more intuitively comprehensible than complex designs developed detached from the implementation process.

**GPIB bus**

Encapsulating the functionality of the C-programming language library Visa32.dll, Gpiib handles the hardware communication over the GPIB bus. Visa32.dll offers basic functions to send and receive data. Gpiib acts as a wrapper class around Visa32.dll to simplify communication. Objects representing hardware devices connected to the computer over GPIB have a Gpiib object as the data member and forward communication requests to this object. This Gpiib object is responsible for sending and receiving data. The next listing shows the class declaration of Gpiib.

```cpp
class Gpiib : public Bus
{
public:
    virtual bool Open(int address, int timeoutMsec);
    virtual bool Send(const string& cmd);
    virtual bool Receive(string& txt, int timeoutMsec);
    virtual bool Transmit(const string& cmd, string& txt, int timeoutMsec);
    virtual bool Close();

private:
    int m_Address;
    unsigned char m_DataBuffer[3000];
    ViStatus m_Status;
    ViUInt32 m_ReceiveCount;
    ViUInt16 m_StatusByte;
    ViSession m_Instrument;

    static int m_SessionCount;
    static ViSession m_DefaultRM;
};
```

Open() checks the value of m_SessionCount. If m_SessionCount is zero, the bus is not yet initialized and Open() creates a default connection. Next, Open() creates and configures a concrete connection and assigns this connection to VI Session m_Instrument. If the creation was successful, the m_SessionCount is incremented. Send(const string& cmd) extracts cmd, creates the control command, and calls the Visa32.dll function viWrite(). viWrite takes m_Instrument as an argument and thus knows to which hardware device it must send the command. Receive(std::string& txt) calls viRead(). viRead() also takes m_Instrument and collects data from the device attached to m_Instrument and writes the result into txt. Finally, Close() closes the connection and decrements m_SessionCount.
Datalogger & sensors

Collecting data from the sensors and controlling amplifier, DCU, cooling system and ventilation system, the datalogger is the central data gathering place and the essential switching unit. Considering the crucial importance of the datalogger, the datalogger's software model must be flexible, robust and well conceived.

The physical datalogger in the exposure systems performs various well-defined tasks. For example, the datalogger switches on the amplifier or triggers the watchdog in the DCU. If the datalogger object (Datalogger) in the software directly maps all functionality, it will become very large, complex, inflexible, and semantically incorrect. Therefore, it must only implement standard, device dependent as well as system independent commands and delegate specific tasks to other objects. Following the example above, the datalogger object does not offer methods like SwitchAmplifier() or TriggerWatchDog(). Rather, Datalogger delegates these tasks to the amplifier (Amplifier) and DCU (Dcu) objects. These objects are themselves responsible for device manipulations. However, Amplifier and Dcu are not capable of performing real hardware communication. (Remember, the physical amplifier and DCU are not directly connected to the computer; they can only be accessed via the datalogger.) Amplifier and Dcu know exactly what they have to do; however, they do not have the means to perform the tasks. That is where Datalogger comes into play again. Datalogger is able to do hardware communication. Allowing the Datalogger to act as a visitor object overcomes this dilemma between knowing what to do and being able to do it. Amplifier and Dcu offer methods for device manipulation. These methods obtain a reference to Datalogger as an argument. The methods subsequently call methods of datalogger with the proper configurations. Amplifier and Dcu offer methods for device manipulation; however, the hard work is done by Datalogger:

```cpp
bool Amplifier::SetPower(bool on, DataloggerAgilent& logger) {
    return logger.SetBitDigitalOutput(on, m_OnOffBit, m_DiglnputChannel);
}
```

SetPower() obtains a reference to DataloggerAgilent34970A (logger). Amplifier knows at which channel (m_DigInputChannel), and at which bit position (m_OnOffBit) the physical amplifier in connected to the physical datalogger. However, Amplifier is not able to do real communication. Datalogger in contrast does not know which device is connected at which channel and position, but knows how to change the binary switches. Therefore, SetPower() calls Logger.SetBitDigitalOutput() with the channel and bit position as arguments.

This principle has two main advantages:

- flexibility. New objects representing hardware devices connected to the datalogger can be implemented without changing the interface of Datalogger.

- natural semantic. For example, clients objects expect the amplifier to offer a SetPower() method. They do not care how the actual communication is performed.
4.3 Design of the controlling software PerformRat.exe

The following listing shows the class declaration of Datalogger.

class Datalogger : public CStreamable
{
public:
    // basic measurement methods
    //
    bool Scan();
    bool IsScanning();
    bool Fetch(int timeout, std::string& data);

    // Channel configuration methods
    //
    virtual bool ConfigChannelDigitalOutput(int bitValue, int channel);
    virtual bool ConfigChannelDigitalInput(int channel);
    virtual bool ConfigChannelVoltDC(int channel, double range,
        double resolution, double integrationTimeSec,
        bool autoDCInputImpedance = false);
    virtual bool ConfigChannelVoltAC(int channel, double range, int filterHz);
    virtual bool ConfigChannelFrequency(int channel, int timeout = 20);
    virtual bool ConfigChannelTemperature(int channel, double resolution,
        std::string wire="FRTO", double rtdtype = 85);
    virtual bool SetBitOigitalOutput(bool on, int index, int channel);
    SetChannelOutputVolt(int channel, double volt);

    // methods for observer pattern. Sensors register to the datalogger and
    // will be informed when new data are available
    //
    void RegisterSensor(Sensor* observer);
    void UnregisterSensor(Sensor* observer);
    void UnregisterAllSensor();
    double GetRawData(int channel);

private:
    // notifies the sensors when new data are available
    //
    virtual void NotifySensors();

    // insert the raw data in the data map
    //
    void InsertData(int channel, double rawData) const;

    // Registered sensors
    //
    std::vector<DataloggerObserver*> m_Observers;

    // raw data map
    //
    mutable MultiValueMap<int, double> m_Data;
Gpib m_Gpib;

Scan() starts a measurement over selected channels, IsScanning() checks whether a scan is currently running, and Fetch() collects the data. Depending on the number of channels to measure, a full scan lasts 0.1 – 15 seconds. Since a scan can last up to 15 seconds, Scan()'s implementation is asynchronous, meaning that Scan() just triggers the measurements and then immediately returns, without waiting until the scan is finished. If clients request data, Datalogger first calls IsScanning(). If no scan is running, Datalogger can safely call Fetch() and returns the data; if a scan is still running, Datalogger returns no data.

Datalogger offers various functions to configure the input and output channels. ConfigChannelVoltAC(int channel, double range, int filterHz) for example prepares the channel at position channel for AC volt measurement, specifies the measurement ranges and applies a filter. SetChannelOutputVolt(int channel, double volt) sets a voltage at a certain channel.

Just collecting data, Datalogger does neither convert nor evaluate the data. If Datalogger should be able to convert and evaluate all data, Datalogger would become complex and inflexible. Converting and evaluating is the sphere of competence of the Sensor. A Sensor cannot collect data itself. However, it knows what to do with the data. Therefore, the data must be transferred from the place of data collection (Datalogger) to the place of data evaluation (Sensor). Consequently, the data flow in the software is exactly opposite to the real data flow. In reality, the data flow from the sensors to the datalogger. In the software, the data flow from Datalogger to Sensor.

The observer software design pattern is appropriate to handle the data transfer Datalogger – Sensor, whereby Datalogger is the subject and Sensors are the observers. Datalogger offers methods for registration and un-registration of Sensors. Datalogger knows all registered Sensors and notifies them when new raw data are available. Sensor implements the actualization method Update(Datalogger& logger), calls logger.GetRawData(m_Channel) and retrieves and stores its raw data. Later, Sensor can convert and evaluate the raw data. Figure 4.12 shows the UML sequence diagram of the data transfer Datalogger - Sensor.

Datalogger stores the raw measurement data in an instance of MultiValueMap<int, double>. MultiValueMap<> is a template class that is able to assign multiple values to a given key. MultiValueMap is an extension of the std::map<>, which allows only the assignment of one value to a given key. The first template parameter (int) represents the channel number and the second template parameter (double) represents the data collected at this channel. This assignment allows the identification of all measurement data.

The complete program logic depends on the clean implementation of Datalogger. Therefore, this class had to be tested very thoroughly. After the tests, no changes in the implementation were done. Datalogger’s flexibility allows the adding and removing of Sensors as well as the controlling of new hardware devices without changing the implementation and without performing additional tests.
4.3 Design of the controlling software PerformRat.exe

`MultiValueMap<>` turned out to be a very handy, flexible and fast helper class used at numerous locations in the program. The complete source code can be found at the end of the thesis.

![Sequence diagram of the data transfer Datalogger - Sensor](image)

**Figure 4.12:** Sequence diagram of the data transfer Datalogger - Sensor. Datalogger and Sensors form an observer pattern.

4.3.2 Dynamic design

PerformRat.exe is a classic "soft" real-time controlling software. "Soft" real-time describes systems with no stringent timing requirements. Like all real-time systems, PerformRat.exe has to react dynamically and predictably to external stimulations. Therefore, the program must have a dependable dynamic design.

**Timer**

In PerformRat.exe, some actions have to occur at a well-defined time or for a distinct time. Consequently, the program must have access to a timer service offering operations to set and retrieve time. Since multiple timers have to run at the same time, encapsulation of the timer functionality in a class is a good idea. Each timer instance then tracks its own time. The next listing shows the complete implementation of the timer class used in PerformRat.exe.

```cpp
class Timer
{
public:
    void SetMillis(int64 seconds);
    int64 GetRemainingMillis() const;
    int64 GetElapsedMillis() const;
    bool IsRunning() const;
}
```
private:
    static unsigned __int64 Int64FromFiletime(const FILETIME& ft) const;
    static unsigned __int64 GetSystemTimeAsInt64() const;
    unsigned __int64 m_EndTime;
    unsigned __int64 m_StartTime;
    static const TIMER::m_FactorSystemTimeUnitToMilliSeconds
        m_FactorSystemTimeUnitToMilliSeconds = 10000;

unsigned __int64 Int64FromFiletime(const FILETIME& ft) {
    __int64 i = ft.dwHighDateTime;
    i = i « 32;
    i = i + ft.dwLowDateTime;
    return i;
}

unsigned __int64 GetSystemTimeAsInt64() {
    FILETIME currentTime;
    ::GetSystemTimeAsFileTime(&currentTime);
    return Int64FromFiletime(currentTime);
}

void Timer::SetMilliSeconds(unsigned __int64 milliSeconds) {
    m_StartTime = GetSystemTimeAsInt64();
    unsigned __int64 duration = milliSeconds* m_FactorSystemTimeUnitToMilliSeconds;
    m_EndTime = m_StartTime + duration;
}

bool Timer::IsRunning() const {
    unsigned __int64 currentTime = GetSystemTimeAsInt64();
    return (currentTime <= m_EndTime) ? true : false;
}

unsigned __int64 Timer::GetRemainingMilliSeconds() const {
    unsigned __int64 remainingMilliSeconds = 0;
    unsigned __int64 currentTime = GetSystemTimeAsInt64();
    if (m_EndTime > currentTime) {
        remainingMilliSeconds = (m_EndTime - currentTime) /
                                m_FactorSystemTimeUnitToMilliSeconds;
    }
    return remainingMilliSeconds;
}
4.3 Design of the controlling software PerformRat.exe

unsigned __int64 Timer::GetElapsedMilliseconds() const
{
    unsigned __int64 currentTime = GetSystemTimeAsInt64();
    unsigned __int64 elapsedMilliseconds = (currentTime - m_StartTime) /
    m_FactorSystemTimeUnitToMilliseconds;
    return elapsedMilliseconds;
}

Timer provides one function for setting the timer and three functions for polling its state. SetSeconds() first calls the Windows API function : :GetSystemTimeAsFileTime(). This function returns the system time as a FILETIME type. FILETIME stores the time in two double word (DWORD, 32 bit) variables with a resolution of \(10^{-7}\) seconds. However, FILETIME does not support basic calculating operators. Therefore, it is not possible to add or subtract two FILETIME variables to calculate time.

SetSeconds() transfers the information of FILETIME into an unsigned __int64 variable and stores the value as the start time. Unsigned __int64 is a standard C++ data type supporting all calculating operators. Next, SetSeconds() converts the time period from milliseconds to \(10^{-7}\) seconds and adds this value to the start time to calculate the end time. IsRunning() retrieves the current time, checks whether it is smaller than the end time and returns the result. GetRemainingMilliseconds() retrieves the current time and calculates end time - current time. GetElapsedMilliseconds() retrieves the current time and calculates current time - start time. Using large unsigned __int64 data types as basic time storing variables, Timer has a high dynamic range. It can handle time periods between 0.1 millisecond and years with a resolution of 0.1 millisecond without suffering from wrap-around effects.

The following code snippet shows a typical application of Timer.

Timer timer;
timer.SetMilliseconds(10000);
while (timer.IsRunning) {
    // do some work
}

CyclicExecutor

PerformRat.exe mainly uses the cyclic executive (CE) approach to organize real-time data acquisition and evaluation. Handling only periodic processes, a CE program controls process execution according to its pre-run-time schedule. Since the controlled tasks are periodic, CE is also periodic or cyclic. The CyclicExecutor class implements the CE mechanism used in PerformRat.exe.

CyclicExecutor is derived from CWnd, the basic window class of the MFC. In Microsoft Windows, periodic processes are closely connected to window classes. For the purpose of PerformRat.exe, periodic processes must be detached from windows. Undertaking this task, CyclicExecutor uses the functionality of CWnd without actually creating a visible window and provides clients with a cyclic executive mechanism.
CWnd implements the methods OnTimer(UINT nIDEvent), SetTimer(UINT nIDEvent, UINT milliseconds) and KillTimer(UINT nIDEvent). SetTimer() starts a timer with a certain ID and a certain time period. After each time period, CWnd autonomously calls OnTimer() with the timer ID activating the function as an argument. CWnd can handle up to 20 timers simultaneously and queues timer requests. If OnTimer() is running while CWnd should accomplish another timer request, CWnd waits until program execution leaves OnTimer() and then immediately triggers OnTimer() again with the new timer ID. Consequently, this mechanism does not guarantee chronologically exact program execution. Clients must make sure that the different timer requests do not block each other. CyclicExecutor overrides CWnd::OnTimer() and provides a simple interface to start and stop timer requests.

Offering only a cyclic executive mechanism, CyclicExecutor does not implement real program functionality. Therefore, CyclicExecutor holds a pointer to the Experiment object (see below) and passes requests to this pointer. The following listing shows the implementation of CyclicExecutor.

```cpp
class CyclicExecutor : public CWnd
{
public:
    static CExecutor* Get();
    void SetExperiment(Experiment* experiment);
public:
    enum TimerId {
        all = 0,
        scan = 1,
        fetch = 2,
        gsmBasic = 3,
        gsmDtx = 4,
        handoverDuration = 5,
        signalStep = 6,
        targetLevelChange = 7,
        watchdog = 8,
        mailbox = 9
    };
    void SetTimer(TimerId id, UINT msec);
    void StopTimer(TimerId id);

protected:
    //afx_msg(CExecutor)
    afx_msg void OnTimer(UINT nIDEvent);
    //afx_msg
declare_message_map()

private:
    Experiment* m_Experiment;
    CyclicExecutor();
    static CyclicExecutor* m_The;
};
```
4.3 Design of the controlling software PerformRat.exe

CyclicExecutor* CyclicExecutor::m_The = NULL;

CyclicExecutor::CyclicExecutor():
{
    ASSERT(m_The == NULL);
    // Create the Windows window object upon which this executor is based
    // (so that it can receive WM_TIMER messages).
    // Make the window invisible
    // VERIFY(CreateEx(O, AfxRegisterWndClass(0),
        //_T("CyclicExecutor Notification Sink"),
        //WS_OVERLAPPED, 0, 0, 0, 0, NULL, NULL));
    m_The = this;
}

CyclicExecutor* CyclicExecutor::Get()
{
    if (m_The == NULL) {
        m_The = new CyclicExecutor;
    }
    return m_The;
}

BEGIN_MESSAGE_MAP(CyclicExecutor, CWnd)
    //CAFX_MSG_MAP(CyclicExecutor)
    ON_WM_TIMER() 
    //])AFX_MSG_MAP
END_MESSAGE_MAP()

void CyclicExecutor::OnTimer(UINT nIDEvent)
{
    TimerId id = static_cast<TimerId>(nIDEvent) 
    switch (id) {
        case TimerId::all: {
            ... 
            break;
            
            case TimerId::scan: {
                m_Experiment->Scan();
                break;
            }
            ... 
        }
    }
}

void CyclicExecutor::SetTimer(TimerId id, UINT msec)
{
UINT uintTimerId = static_cast<UINT>(id);
SetTimer(uintTimerId, msec, NULL);
}

void CyclicExecutor::StopTimer(TimerId id)
{
    UINT uintTimerId = static_cast<UINT>(id);
    KillTimer(uintTimerId);
}

To guarantee controlled program execution, it is important that in PerformRat.exe only one copy of CyclicExecutor exists. Therefore, CyclicExecutor is implemented as a singleton [19]. The singleton pattern assures the creation of one and only one CyclicExecutor. CyclicExecutor is not thread safe. It should only be used in the main thread. Using CyclicExecutor in multiple threads may result in unpredictable behavior.

CyclicExecutor is derived from CWnd. Therefore, CyclicExecutor inherits the timer mechanism. The constructor of CyclicExecutor creates the associated window and thereby activates the timer mechanism. CyclicExecutor creates an invisible window. Therefore, this window cannot be seen on the screen. SetTimer() activates a particular timer. After the time is up, OnTimer() is triggered. OnTimer() identifies the particular timer and takes the appropriate action. Eventually, StopTimer() stops a particular timer from execution.

PerformRat.exe uses mainly the cyclic executive (CE) approach to organize real-time data acquisition and evaluation, because CE provides highly predictable behavior. Considering that the program must collect, evaluate and average data from about 200 sensors, the program execution should be as predictable as possible. If the execution is not highly predictable, the large number of sensor data would likely result in disordered program behavior. Additionally, depending on the data collected, the program must initiate appropriate actions, which must not be interrupted. CE guarantees these uninterrupted executions.

Another possibility to organize data collection and evaluation is using threads. A worker thread constantly polls the sensors and collects data. This thread then transfers the data to the main thread. The main thread evaluates the data and executes appropriate commands. Each command in turn is executed in a separate worker thread. Such an approach requires heavy cross-thread communication. However, programs with many threads that must communicate with each other tend to become complex and difficult to debug.

In PerformRat.exe, the thread approach is not suitable for another reason. The datalogger used in the PERFORM A setup is able to set switches while performing a scan. Using a thread based structure to organize data collection, scan and switch commands would have to be set from different threads. However, accessing the same hardware device from multiple parallel threads is very error-prone. This should be avoided whenever possible.

Using parallel threads is perfectly suitable for simple monitoring software, where the sensors just detect binary states and do not deliver additional data that must be further evaluated.
4.3 Design of the controlling software PerformRat.exe

The CE implemented in PerformRat.exe allows the collection and evaluation of data as well as the execution of controlling commands in a single thread. If a controlling command is still running while the CE should perform a scan, the CE simply waits until the command has terminated its task. Obviously, such architecture does not assure exact timing. However, this design has a huge advantage: it is relatively simple and straightforward. Since the timing constraints for the PERFORM A experiment have not been extremely tight, CE has been chosen as the primary data collection mechanism.

PerformRat.exe sources out only large tasks taking a long time to complete in separate threads. Examples of such tasks are the data transfer over the internet, the sending of error e-mails and the downloading of dynamically calculated waveforms to the function generator.

4.3.3 Program flow logic

The user interface maps the internal program flow logic. Therefore, looking at the user interface first simplifies the understanding of the program flow.

The message field shows the system's state and informs the user what to do next. Clicking the next button performs the action displayed in the message field. Clicking the abort button interrupts the current task and transfers the system into a safe sleep mode. In the lower half of the figure at the right and left sides, the four exposure groups are shown. A Ferris wheel symbol shows the overall wheel status (green: everything ok, no exposure running; blue: animals can be loaded / unloaded. It is completely safe to step in the animal room; claret (color of red wine): exposure is running; red: system failure in the wheel detected). Clicking on a Ferris wheel symbol selects a wheel. Thereby the serial number's color changes from black to red. The middle section displays the parameters as well as the tolerance intervals of the selected wheel (deviation: only during exposure, deviation measured field strength / target field strength; temperature: temperature in the ring channel; oxygen: oxygen concentration the ring channel; humidity: humidity concentration in the ring channel). In each exposure group, an icon displays the amplifier's state and another icon displays the ventilation system's state. In the lower left corner, the coolant icon indicates the amplifier cooling system's state. On top left, the user enters the experiment settings. He or she selects the animal group and the average weight of all animals. These controls are only enabled when the text in the message field informs the user to input the settings for the next exposure. The software needs to know the animal weight to calculate the field strength corresponding to the desired whole-body SAR of the animals.

Only two buttons are crucial for the understanding of the program flow logic: “Next” and “Abort”. At any time, the exposure system is in a well-defined state. The system can change its state in two ways: autonomously or by user request. An example of an autonomous change is after the occurrence of an error. As soon as the system detects the error, it changes its state from the current state into an error state. Users on the other hand can click “Next” or “Abort” and force the system to change its state.
The System is ready.
Press "Next" to initialize the system.

Figure 4.13: PerformRat.exe front-end

An adequate software model of this system behavior provides the state pattern [19]. This pattern allows an object to change its behavior at run-time if its internal state changes. A context object manages a concrete state object and forwards state specific requests to this state object.

4.3.4 Experiment and ExperimentState

The fundamental object describing the dynamic program logic is the Experiment. Managing a pointer to an ExperimentState object, the two objects together compose a state pattern. Experiment delegates state specific requests to ExperimentState. Experiment implements the required functionality to conduct the exposure; it does not know, however, when to use which functionality, simply because it does not know its current state. ExperimentState on the other hand implements no functionality. However, each state knows when to use which functionality of Experiment. Figure 4.14 shows a simplified UML class diagram of the state pattern Experiment - ExperimentState and other involved classes. Additionally, the diagram describes the state change from ExposureState to ErrorState.
4.3 Design of the controlling software PerformRat.exe

Experiment

+OnButtonNext() void
+OnButtonAbort() void
+Scan(in on : bool) bool
+Fetch(in on : bool) bool
+SetSwitch(in on : bool) bool
+SetLight(in colour) bool
+StartSignal() bool
+Notify() void
+ChangeState(in newState : ExperimentState) void

ErrorState

+The ErrorState
+OnButtonNext(in exp : Experiment) bool
+OnButtonAbort(in exp : Experiment) bool
+Eval(in exp : Experiment) bool
+Get() ErrorState

ExposureState

+The ExposureState
+OnButtonNext(in exp : Experiment) bool
+OnButtonAbort(in exp : Experiment) bool
+Eval(in exp : Experiment) bool
+Get() ExposureState

CyclicExecutor

+TimerId
+The CyclicExecutor
+CyclicExecutor(in)
+SetTimer(in Id : TimerId, in msec : unsigned int) void
+StopTimer(in Id : TimerId) void
+OnTimer(in Id : TimerId, in msec : unsigned int) void
+Get() CyclicExecutor

Figure 4.14: State pattern Experiment - ExperimentState. The comments show code snippets of the state change from ExposureState to ErrorState.
Experiment

Experiment has three concrete child objects (Setup m_Setup, Signals m_Signals and Settings m_Settings) and a pointer to the currently active ExperimentState object (ExperimentState* m_State). Representing the exposure system, Setup implements the hardware communication. Signals encapsulates the signal settings and the synthetic signal generation algorithm. Finally, Settings contains different program settings, such as network settings, timing settings, etc.

Experiment provides basic methods to control and operate hardware (Scan(), Fetch(), SetRf(), SetSwitch(), ResetDcu()) and delegates the real implementation to its Setup object. Experiment also presents a method to start the signal generation mechanism.

Whenever the user clicks the “Next” button on the graphical control panel, the panel calls Experiment::OnButtonNext(). Experiment::OnButtonNext() is therefore the event handler of the event the user clicks the “Next” button. Experiment::OnButtonAbort() is the corresponding event handler if the user clicks the “Abort” button.

Experiment::Notify() informs the graphical display that data have changed and that it must redraw.

Experiment::ChangeState(ExperimentState* newState) assigns newState to m_State. Thereby another ExperimentState becomes active. However, Experiment does not know which one.

ExperimentState

Declaring the purely virtual methods OnButtonNext(), OnButtonAbort(), Enter() and Eval(), ExperimentState is an abstract base class. Additionally, ExperimentState has basic state specific attributes. m_EnableButtonStart declares if in a concrete state the "Next" button is enabled or not. m_Text stores the text that will be displayed in the message field when the software is in this state.

Concrete states (for example ErrorState, ExposureState) are derived from ExperimentState. They implement the purely virtual methods of ExperimentState. Each concrete state is a singleton to assure that only one instance is created. Concrete ExperimentState objects know their successor state and their "abort state". For example, consider ErrorState::OnButtonNext(Experiment* exp). ErrorState knows its successor state (CheckState). ErrorState::OnButtonNext() calls exp->ChangeState(CheckState::Get()). Thereby, Experiment changes its state. The implementation of ErrorState::OnButtonAbort(Experiment* exp) is similar. Enter() of a concrete state implements the commands that must be executed when the program enters into this state. Each state knows what it has to do. Finally, Eval() implements the state specific evaluations.

The overall behavior of Experiment is divided into different ExperimentStates. A concrete ExperimentState localizes state specific behavior. Concrete ExperimentStates know exactly what they have to do. Since all state specific code resides in a class derived from ExperimentState, new states and state transitions can be added simply by defining new subclasses of ExperimentState.
CyclicExecutor
This class has been described above. Each concrete state starts several timers.

4.3.5 Example: User clicks the next button
This paragraph illustrates the program flow logic when the user clicks the "Next" button and thereby forces the software to change its state. Figure 4.15 shows the corresponding UML activity diagram.

Figure 4.15: Simplified UML Sequence diagram of state change ReadyState to ExposureState
The click event triggers `Experiment::OnButtonNext()`. Experiment is not able to handle the event itself and delegates it to its active state `m_State`. To do this, Experiment calls `m_State->OnButtonNext(this)`. The active state is able to handle the event, because it knows its successor state. Next, the active state calls `Experiment::ChangeState(ExperimentState* newState)` with its successor state as an argument. `ChangeState(ExperimentState* newState)` assigns `newState` to `m_State`. Thus, the state change is accomplished. After the assignment, `ChangeState` calls `m_State->Enter()`. The new state is asked to initialize itself. Depending on the concrete state, `Enter()` sets the display control variables, forces the display to redraw, stops all cyclic timers that might still be running from the previous state, initializes hardware, starts the signal statistic, prepares the hardware and sets the state specific cyclic timers. If a command should fail to complete properly, the concrete state gets a pointer to its corresponding error state and calls `Experiment::ChangeState()`.

### 4.3.6 Compact program description

`PerformRat.exe` consists of three modules. Each module has to perform well-defined tasks. 1) Representing the physical exposure system, `Setup` handles all hardware communication, collects, converts and evaluates data, checks the status of all devices and regulates the field strength. 2) `CyclicExecutor` triggers regularly recurring actions like starting a scan, fetching data and triggering the watchdog, as well as irregularly recurring actions used to generate the GSM signal. Implemented as a singleton with a global interface, `CyclicExecutor` can be accessed from every location in the program. The task `CyclicExecutor` executes must be implemented asynchronously and lasts only for a short time to complete in order to prevent `CyclicExecutor` from being blocked. 3) The state pattern `Experiment` (acting as a context) – `ExperimentState` (acting as a state) describes the dynamic program behavior. At any given time, the program is in a well-defined state. Each state is responsible for itself and knows which actions to perform. States do not implement the functionalities to perform the actions. Rather, they indirectly call methods of `Setup` and start recurring tasks by calling methods of `CyclicExecutor`. After becoming the active state, a certain `ExperimentState` arranges its environment. With self-contained objects to represent states, the state changes of `Experiment` are explicit. The complete state change is accomplished by a single assignment. Since the state changes from the point of view of the `Experiment` are atomic, the `ExperimentState` objects prevent the `Experiment` from adopting inconsistent internal states.

Figure 4.16 summarizes the program flow. The diagram is a slightly modified UML activity diagram. It shows a principle program state as a rectangular object with semicircular left and right ends. The rectangles connected in sequence with a state show the order of main tasks a state performs. Decision points between states show conditions that must be fulfilled so that the program can change states.
4.3 Design of the controlling software PerformRat.exe

![UML activity diagram]

Figure 4.16: Simplified UML activity diagram of the program flow
After the initial experiment start, the program is in InitState. In this state, the software is configured according to the program settings saved in the file config.dat. For example, before starting the experiment, the program configurator selects the number of exposure units and the number of wheels in each unit. The software checks these settings and prepares the static structures according to these settings. Next, the software creates a log file and saves all configuration settings. Afterwards, it initializes the hardware. If no error is detected, the software changes into IdleState. In this state, the software collects and evaluates the values of the environment sensors. If an error is detected, the software changes into ErrorState. This state determines the error level and if necessary sends an e-mail to ITIS containing the error cause as body text. When the user presses “Next” in IdleState, the software changes into the Warm-UpState. This state switches the amplifiers on and waits until they reach operating temperature. As soon as the amplifiers have operating temperature, the software changes autonomously into CheckState. This state tests all devices, calculates sensor offsets and checks the remote connections. If all checks are successful, the software changes into SettingsState. In this state, the user enters the average weight of all rats and selects the animal group. Next, SettingsState calculates the target field out of the average rat weight to achieve the desired whole-body SAR. Afterwards, the software changes into Start-UpState. This state calculates and applies the start RF power, performs a scan, reads the field sensors, and compares the measured field with the target field strength. If the measured and target fields differ significantly, the correction is calculated and the power is adjusted. After having found the start power, the software changes into ExposureState. This state splits the program execution in three parts. The main part collects and evaluates data from the sensors and checks the state of the hardware devices. Additionally, it compares the measured field with the target field and adjusts the power if necessary. The target power is coupled with the signal statistic and changes during the exposure. Therefore, the main program part must regularly check for target power changes. The second part generates the GSM signal according to the statistic. Finally, the third part continuously sends the watchdog signal. After finishing the exposure, the software changes into ExposureCompletedState.

Consider that the program flow described above is a rough simplification. Some states have been omitted completely and still others are incomplete. Nevertheless, Figure 4.16 shows a good overview of the PerformRat.exe program flow.

4.4 Dosimetry

The dosimetry is the exposure system’s scientific core and for this reason of central importance for the controlling software. However, what exactly does dosimetry mean in the context of EMF laboratory studies? To answer this question, the first consideration is what an EMF laboratory study should investigate. Normally, a study’s aim is to expose a test subject (human being, animal, cell culture) with a well-quantified biological exposure dose and to observe how the test subjects respond to this dose. What exactly is a biological exposure dose in the context of EMF laboratory studies? A biological exposure dose is a variable inside the test subject that is only influenced by the outer electromagnetic field and that is considered relevant for biological processes. Different test subjects exposed to different frequencies have different biological exposure doses. For example,
the relevant biological exposure dose for all test subjects exposed to RF electromagnetic fields is the SAR inside the test subject. The primary relevant unit is therefore the SAR, whereas the electromagnetic field is, though the fundamental unit, the indirect unit. It is very difficult or sometimes even impossible to measure the SAR inside the test subject. For example, consider the exposure of animals or human beings. It is not possible to plug a measuring probe into living biological tissue. Whenever the direct measurement of the SAR in the test subjects is not possible, indirect methods try to reveal the SAR. One such method is to calculate the SAR in test subject out of the incident electromagnetic field strength. This indirect assessment of a unit is exactly what dosimetry describes: to calculate the primary relevant unit (SAR) out of an indirect unit (field strength). Obviously, the connection between the relevant and the indirect units has to be known to calculate the relevant unit out of the indirect unit. This connection is called the transfer function \( f \). Expressed differently, the relevant unit \( R \) is a function of the indirect unit \( I \) and other parameters \( P_s \):

\[
R = f(I, P_s).
\]

It is not a priori obvious which parameters \( P_s \) influence the transfer function. As a result, the first step to derive the transfer function for a given experiment is to unravel all parameters \( P_s \) influencing the transfer function. In the next second step, the relationship among the indirect unit and these parameters \( P_s \) must be identified.

In vivo, in vitro and human studies have different transfer functions. The field strength is not always the indirect unit, which is used to calculate the relevant unit, although the field strength is always the fundamental indirect unit. However, sometimes it is not possible to measure the field strength, and only a derived unit can be measured. For example, consider an exposure chamber where the measurement of the field strength inside the chamber is not possible. In such a case, only the input RF power can be measured and the field strength inside the chamber is estimated from the input power. Consequently, the relevant unit SAR must be calculated from the RF power. Normally, measuring the fundamental unit (field strength) close to the test subject is more precise and less error-prone than measuring a derived unit.

These considerations can be transferred to the controlling software of EMF laboratory studies. The software must make sure that the target exposure dose is reached. However, it cannot measure the exposure dose directly. Rather, the software must calculate the exposure dose (SAR) out of the indirect unit (field strength). After that, it can compare the converted value with the target value. In the PERFORM A experiment, the controlling software measures the field strength and converts the values into corresponding whole-body SAR values of the exposed rats. Therefore, the software must be able at any time to convert measurement data into exposure dose data and vice versa.

Preferably, a class encapsulates these conversions. The listing below shows the declaration of such a conversion class. Dosimetry is an abstract class serving as a base for concrete dosimetry classes. Dosimetry declares only two methods. One method calculates the relevant unit out of the indirect unit. The other method converts the relevant unit into the
indirect unit. The term Si at the end of the method names as well as at the end of the parameter names expresses that the methods return and expect SI units.

class Dosimetry
{
public:
    Dosimetry();
    virtual double GetRelevantUnitSi(double indirectUnitSi) const = 0;
    virtual double GetIndirectUnitSi(double relevantUnitSi) const = 0;
};

Concrete dosimetry classes implement the two methods and provide a constructor taking the conversion parameters as arguments:

class PerformRatDosimetryVersion1 : public Dosimetry
{
public:
    PerformRatDosimetry(double meanRatWeightKg, double frequencyHz, double fitParameter1, double fitParameter2, double fitParameter3);
};

This mechanism allows the easy exchanging of different concrete dosimetry classes without changing client code. Client code just calls the methods of a generic Dosimetry pointer without having to know the concrete implementation. Whenever a new dosimetry is requested, a new Dosimetry class is created. Then this new object is assigned to a generic pointer of the type Dosimetry.

4.5 Definition and implementation of optimized GSM exposure signal

Most exposure signals used in the past to assess the effects of GSM signals on biological tissues did not consider low frequency power variations nor overall signal power variations. As a result, most exposure signals did not reflect the actual human exposure. Obviously, studies using signals that differ significantly from real GSM signals have limited significance.

To simulate a GSM signal properly, a mathematical model describing the signal properties accurately must be derived first. Until now, no mathematical GSM signal model has been developed and not even proposed. Consequently, no exposure system was capable to simulate a real GSM signal.

This thesis presents a mathematical GSM signal model as well as the implementation of this model. This signal model represents the first exposure scenario that applies temporal changes of different modulation schemes (mainly a cocktail of low-frequency modulation components and power variations) as occurring in an actual phone conversation. Therefore, the mathematical GSM signal model and its implementation are real innovations. This GSM model was presented at 24th Annual Meeting of the Bioelectromagnetics
4.5 Definition and implementation of optimized GSM exposure signal

The rationale behind the GSM signal model was to find functions and parameters describing the actual human exposure, especially with respect to low frequency power variations, since they are considered to have a potential for evoking biological responses. Therefore, the signal must contain the entire ELF spectral content of the different GSM modulation modes but also typical power variation as occurring due to environmental changes while using a handset within a GSM network. To define the functions and parameters simulating the environmental events, measurement data as reported by Wiart et al. [38] has been analyzed. Using a test mobile system at 900 MHz, Wiart et al. have recorded the power control level along routes in Paris and its vicinity.

The proposed GSM signal model contains the following features: handover, power control (PWC), discontinuous transmission mode (DTX), and time division multiple access (TDMA).

4.5.1 GSM burst specifications

GSM systems use TDMA. Eight users share one physical radio channel. This is achieved by dividing the channel into eight time-slots. Therefore, the mobile phone only emits power during one eighth of the time, sending an amplitude-modulated burst signal. The European Telecommunications Standards Institute (ETSI) has defined the limits for the power ramping of these burst [39]. Pedersen et al. [40] present a comprehensive description of the burst specification.

IT'IS has developed exposure systems for several bio experiments. The different exposure systems generate the GSM bursts differently. PERFORM A's main toxicological studies use an expensive digital generator to create the GSM signal. The ramping of the GSM burst generated by this generator has a cosine shape. This signal includes simulated speech using pseudo-random binary sequences with a length of 23 bits, multiframes (one MF = 26 frames) and intermediate-multiframes (one IMF = 104 frames, duration 480 ms). These signal parts lead to an enhanced spectrum including 8 Hz and 2 Hz components.

The in vitro studies of the REFLEX project as well as the PERFORM A replication studies of the Repacholi experiment [36] performed at RBM (Ivrea, Italy) and ARCS (Seibersdorf, Austria) apply another signal. There, a combination of a conventional generator with an arbitrary function generator creates the GSM signal. Only lacking the simulated speech feature but including multiframe (MF) and intermediate multiframe (IMF) structures, the burst meets the ETSI specifications. This solution to generate the signal out of a combination of a conventional generator and an arbitrary function generator is cheaper than using an expensive digital generator.

Figure 4.17 shows the applied GSM bursts for the different studies.
Figure 4.17: Details of the synthetic GSM power burst in comparison with the ETSI standard. Note the break in the time axis. The white area indicates the ETSI standard.

4.5.2 GSM features

Handover
Whenever a GSM phone receives a stronger signal from another base station than the currently active one or if the current base station is overloaded, the active base station hands over the communication to another base station. During this process, the mobile phone changes frequency and adjusts the power output to the maximum level. In the following seconds, the mobile phone stepwise reduces the output power until it reaches a level still strong enough for reliable communication. This system behavior is called power control.

Power control
Whenever connection quality allows power reduction, the base station and mobile phone can reduce the output power. However, the mobile phone does not reduce the power autonomously. The base station commands the mobile phone to reduce its power and thus controls in fact the power level of the mobile phone. This dynamic power regulation significantly reduces the mobile phone's output power. This has two advantages: 1) reduction of interference problems 2) saving of the mobile phone's battery power. Power control does not reduce the output power continuously but in steps of 2 dB. Depending on the concrete network implementation, the number of PWC levels varies. The ETSI standard simply defines maximal 15 levels for GSM 900 MHz and DSC 1800 MHz [41].
Discontinuous transmission mode

Reducing a mobile phone's power consumption during a phone conversation, DTX is an advanced feature of the GSM standard. The mobile phone reduces transmission during silence periods from 104 to 12 frames in an intermediate multiframe, decreasing the power by 88% (1 - 12/104). If a mobile phone's speech detection works perfectly, DTX reduces power consumption by 44% (standard dialog, both partners talk for 50% of the overall call duration). The standard GSM modulation with 104 frames in an intermediate multiframe is called GSM Basic, and the DTX modulation with only 12 frames in an intermediate multiframe is called GSM DTX.

Because of the power reduction in the DTX period, the SAR in a mobile phone user's brain drops as well. The power and SAR reduction results only due to the fact that the signal contains fewer frames, whereas the peak power (= maximal frame power) remains the same. The GSM DTX signal contains more frequency components than the GSM Basic signal. This fact is especially important for epidemiological studies, since low frequency power variations may cause negative health effects.

Figure 4.18 shows the modulation sequence of a GSM Basic (A) and a GSM DTX signal (B) in an intermediate multiframe. Figure 4.19 displays the spectra of both signals derived by numerical calculations using 16 intermediate multiframes with 8320 sampling points.

Figure 4.18: Modulation in an intermediate multiframe for a GSM Basic (A) and for a GSM DTX signal (B)
Figure 4.19: Spectra for a GSM Basic signal (top) and a GSM DTX signal (bottom). The GSM Basic signal consists only of 8 Hz and 217 Hz components. The GSM DTX signal on the other hand contains many more frequency components. Note the different amplitude ranges.

4.5.3 Signal model

The signal model contains six independent random distributions:

- **handover.** This distribution determines the time between two handover events. After a handover, the signal power instantly increases to maximum power level.

- **environment event.** An environment event indicates a change in connection quality (e.g., turning of the head, entering a house, changing rooms, increasing or decreasing the distance to the base station). This distribution determines the time between two environment events. Handovers and environment events are completely independent. In the model, an environment event can occur just after a handover. This may be a signal model's shortcoming, because in a real GSM system, the connection quality does not change immediately after a handover.
4.5 Definition and implementation of optimized GSM exposure signal

- target level change. After an environment event, the signal model calculates a new target power level. This new level indicates the minimum power needed to ensure good connection quality. The target level change distribution calculates with a resolution of 2 dB the difference between the actual target level and the new target level.

- power ramp. The power ramp comes into operation whenever the present power level does not correspond to the target power level. The signal model approaches the target level in steps of 2 dB. The power ramp distribution determines how long the signal power remains at the new power level. The duration is usually around four seconds, with the exception of the first step after a handover. This step lasts approximately half as long (due to measurement perceptions).

- GSM DTX. This distribution calculates the duration of GSM DTX signal periods.

- GSM Basic. This distribution calculates the duration of GSM Basic signal periods.

Figure 4.20 visualizes the different distributions and the duration of the different events.

![Figure 4.20: Visualization of the GSM signal features](image-url)
**Measurement data**

Using a test mobile system at 900 MHz, Wiart et al. [38] have recorded the output power of the system along routes in Paris and surrounding areas. Wiart has analyzed three cases: driving on a highway, driving in the city and walking in the city. Figure 4.21 shows the measured mobile phone’s output power for the three scenarios. These measurements have been analyzed with respect to the model described above. The goal was to derive the statistical information for the six model distributions out of the measurement data. The next section describes the results of the data analysis.

![Graphs showing measured power levels for different scenarios](image)

Figure 4.21: Measured power level of a mobile phone for a pedestrian in the city (duration: 16 minutes), a car driver in the city (duration: 80 minutes) and a car driver on the highway (duration: 50 minutes) (From [38]).

**Handover statistics**

No correlation between the power variation of a mobile phone and health risk has yet been determined. Nevertheless, the PERFORM A and REFLEX studies have applied high power dynamics. Obviously, a highway driver using a mobile phone experiences far more handovers than a city driver or a pedestrian, since the highway driver drives with a large velocity through the GSM micro-cellular system. Consequently, the power variations for a highway driver are higher than for the two other user types. Considering the fact that high power variations are desired, the data of the highway driver has served as the basis for the handover statistics. The two other cases have not been analyzed for the handover statistics. The time distribution between handovers was analyzed using the chi-squared test of fit goodness for the following distributions:

- normal
4.5 Definition and implementation of optimized GSM exposure signal

- lognormal
- exponential
- extreme Value type I
- gamma
- weibull.

Likelihood-Ratio-Chi-Square was used to calculate the relation between the distributions

\[ \chi^2 = -2n \sum_{i=1}^{n} \ln \left( \frac{t_i}{r_i} \right), \]

where \( t_i \) represents empirical data and \( r_i \) expected data.

Next, the Maximum Likelihood method was applied to calculate the mean value \( \bar{t} \) and the variance \( \text{var} \) of the tested distributions

\[ \bar{t} = \frac{1}{n} \sum_{i=1}^{n} t_i, \quad \text{var} = \frac{1}{n^2} \left[ n \sum_{i=1}^{n} t_i^2 - \left( \sum_{i=1}^{n} t_i \right)^2 \right]. \]

The analysis has shown that the data can be best fitted with an exponential distribution

\[ y = Ae^{-\lambda t}, \]

where \( y \) is the calculated time between two handovers, \( A \) is a normalization factor and \( \lambda \) the shape parameter. The shape parameter turned out to be \( \lambda = 0.0233. \)

The Gamma and Weibull distributions have revealed first order shape parameters close to one. For this value, the two distributions reduce to the exponential distribution. Therefore, they have not been analyzed further.

Figure 4.22 shows the histogram of the time between handovers and the corresponding exponential distribution.
Environment events statistic

Two definitions have provided the basis for extracting environment events out of the measurement data: the changing of power trend direction (from increasing to decreasing or vice versa) and a constant power level for more than five seconds (according to power ramp analysis). The rationale for the latter case comes from the fact that the longest period of constant power in a power ramp lasts 5 seconds.

The distribution analysis has revealed very similar statistics for all three data samples (city walk, city driving, and highway driving). To increase the sample size, all samples have been merged. Next, the same statistical analysis as for the handover statistic was applied. The result was that an exponential distribution \( \lambda = 0.0867 \) describes the data adequately. Figure 4.23 shows the shape distribution and the histogram of the measured data.

Figure 4.22: Histogram of the time between handovers and the resulting distribution

Figure 4.23: Histogram of the time between environment events for all three user groups and the resulting distribution
4.5 Definition and implementation of optimized GSM exposure signal

Target level change statistic

Only step durations of more than five seconds in the data between two successive environment events have been considered as target level changes. Additionally, the target level change is not valid when a handover has occurred between two environment events.

To increase the sample size, again the data of all three samples have been merged. Avoiding a random walk in a favoured direction has been achieved by centring the distributions.

Analysing all target level changes did not lead to an obvious solution. None of the applied distributions generated passable fits. However, the normal distribution fits the data slightly better than the other distributions. Additionally, the normal distribution showed a broader distribution for restricted power ranges (e.g., +/- 10 dB) and hence leads to more target level changes, which in turn means a higher dynamic. Considering that high dynamics is desirable, the normal distribution was chosen to model the target level changes. The resulting distribution has the form

\[ y = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{\sigma^2}{2\sigma^2}}, \]

where \( y \) is the time between two target level changes and \( \sigma \) the standard deviation. The best fit was achieved with \( \sigma = 5.83 \) [dB].

Figure 4.24 shows the histogram of the detected target level changes and the corresponding centred normal distribution.

![Figure 4.24: Target level change statistics: data of all user groups and corresponding centred normal distribution](image)
Power ramp statistic

The power step durations were extracted out of the data for a pedestrian walking in the city. Analysing the data revealed a normally distributed step duration

\[ y = \frac{1}{\sqrt{2\pi} \sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \]

where \( y \) is the step duration, \( \mu \) the mean value and \( \sigma \) the standard deviation. The best fit was achieved with \( \mu = 4 \) s and \( \sigma = 0.414 \) s. The duration of the maximum power level after a handover is shorter than the other ones. Again, normal distribution (mean 2.3 s, standard deviation 0.2 s) describes this time interval adequately.

Figure 4.25 shows a typical power ramp after a handover.

![Figure 4.25: Mobile phone power near a base station after a handover](image)

GSM Basic - GSM DTX

For real systems assuming a speech rate of 50% is reasonable (both conversation participants talk and listen for 50% of the overall call duration). For systems using DTX, Zimmerman [42] assumes a power reduction of 30% for the above scenario. This low power reduction rate results from imperfect speech detection. A power reduction of 30% corresponds to a no speech detection rate of 34%.

The literature shows no experimental evaluation of pause intervals while using a mobile phone. Therefore, to obtain a basic idea of the interval lengths, very rudimentary quantifications were done measuring the duration of pauses people make while phoning. Figure 4.26 shows the histogram of the collected data.
4.5 Definition and implementation of optimized GSM exposure signal

Supposing that ending a pause has the same probability in each second leads to an exponential distribution. The Maximum-Likelihood method of the data yields a shape parameter of $\lambda = 0.122$. Assuming a speech rate in the measured phone calls of 50%, the detected GSM DTX phases and the GSM Basic phases are exponentially distributed as well. They do not have a minimum length, and the shape parameters are $\lambda = 0.180$ and $\lambda = 0.0924$, respectively. These parameters yield to speech detection rate of 34%.

As already stated, the model for the synthetic GSM signal contains six independent distributions. They describe an adequate model for simulating a real-world GSM signal. Table 1 summarizes the results and lists the distributions with the corresponding shape parameters.

<table>
<thead>
<tr>
<th>Event</th>
<th>Sample</th>
<th>Distribution</th>
<th>Shape parameter</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handover highway</td>
<td></td>
<td>exponential</td>
<td>$\lambda = 0.0233$</td>
<td>43 s</td>
</tr>
<tr>
<td>Environment all</td>
<td>all</td>
<td>exponential</td>
<td>$\lambda = 0.0867$</td>
<td>11.5 s</td>
</tr>
<tr>
<td>Target level change</td>
<td>all</td>
<td>normal</td>
<td>$\sigma = 5.83$</td>
<td>0 dB</td>
</tr>
<tr>
<td>Power ramp city</td>
<td></td>
<td>normal</td>
<td>$\sigma = 0.414$</td>
<td>4 s</td>
</tr>
<tr>
<td>GSM DTX office</td>
<td>office</td>
<td>exponential</td>
<td>$\lambda = 0.180$</td>
<td>5.6 s</td>
</tr>
<tr>
<td>GSM Basic office</td>
<td></td>
<td>exponential</td>
<td>$\lambda = 0.0924$</td>
<td>10.8 s</td>
</tr>
</tbody>
</table>

Table 1: Distribution and shape parameters for the model events

Figure 4.26 shows a computer simulation of a GSM signal using the model presented above.
4.5.4 Implementation

The first consideration is how the hardware generates the GSM signal. The digital generator as well as the combination of a conventional generator with an arbitrary function generator cannot generate a real GSM Basic signal. They are merely able to generate a TDMA main frame signal. In this signal, only the GSM main frame is active with main frequency components of 217 and 1733 Hz. The TDMA main frame and the GSM Basic signal differ slightly. In the GSM Basic signal, an associated control channel appears in every 26th frame in which no burst is transmitted (idle frame to receive and analyze signals emitted by other base stations). This missing burst in the GSM Basic signal yields an additional frequency component of 8 Hz. Thus, to obtain a GSM Basic signal out of a TDMA main frame signal, every 26th frame has to be blanked. In the PERFORM A exposure system, the digital control unit (DCU) is responsible for the blanking of every 26th frame. The DCU blanks the amplifier for the time duration of every 26th frame. This GSM Basic blanking transforms the amplifier input signal (TDMA main frame signal) into a GSM Basic output signal.

The same mechanism is used to generate a GSM DTX signal out of a TDMA main frame signal. Again, the DCU blanks the corresponding frames. This GSM DTX blanking transforms the amplifier input signal (TDMA main frame signal) into a GSM DTX output signal. The alternate blanking (GSM Basic – GSM DTX) of the amplifier allows the creation of a GSM Talk signal out of a TDMA main frame signal. This signal consists of temporal changes between the GSM Basic and GSM DTX signals and simulates a typical conversation without moving. The DCU is only able to generate GSM Basic and GSM DTX signals.
out of a DTMA signal. However, it does not implement the control logic to switch be­
tween the two signal types in order to generate the GSM Talk signal. This is the task of
the controlling software. The software commands the DCU to alternately apply the two
blanking types according to the signal statistic. Therefore, the software implements the
GSM Basic – GSM DTX signal statistic.

The model for the GSM signal contains power variations as well. Basically, the hardware
can change the output power in two ways: 1) by changing the amplifier power and 2) by
changing the RF generator power. Power changes due to the signal statistics have to act
on all four exposure groups of the PERFORM A exposure system. Remember, each expo­
sure group has its own amplifier, but the whole setup has only one generator. Changing
the generator power changes the power in all exposure groups accordingly. Therefore,
since it is easier to change the power at one central place than at four different locations,
the controlling software changes the generator power and not the amplifier power to
generate the varying output power imposed by the GSM signal statistic.

The signal consisting of the modulation changes GSM Basic – GSM DTX (= GSM Talk) as
well as the power variations is called a GSM Environment signal. The following enumera­
tion summarizes the different signal types:

- TDMA Main Frame. Only the GSM main frame is active in this signal (main fre­
quency components: 217, 1733 Hz).

- GSM Basic. Active during speaking (main frequency components: 8, 217, 1733 Hz).
  8 Hz components result from the slow associated control channel, which appears in
every 26th frame in which no burst is transmitted (idle frame to receive and analyze
  signals emitted by other base stations).

- GSM DTX. Active during non-speaking (main frequency components: 2, 8, 217,
  1733 Hz). Transmission during the silence period is reduced to 12 frames in an in­
termediate multiframe (104 frames, repetition rate 2 Hz).

- GSM Talk. Simulates a typical conversation without moving the handset. It consists
  of temporal changes between the GSM Basic and GSM DTX signal.

- GSM Environment. Simulates a typical phone call while moving and turning
  around in a GSM cellular network. The modulation consists of GSM Talk with addi­
tional varying output power of the phone (SAR) due to handover and power control
  events. The events are subdivided into environmental, target level change and
  power level stay events.

Considering the remarks above, the software must perform two tasks to generate the
GSM Environment signal out of the standard DTMA main frame signal:

- Temporally change the amplifier blanking (via DCU) to switch between the GSM
  Basic and the GSM DTX signals in order to generate the GSM Talk signal. This in­
volves the GSM DTX and GSM Basic statistics.
• Change the output power of the generator to simulate handovers and power control events. This involves the handover, environment, target level change and power ramp statistics.

Therefore, all the software has to do is blank the amplifier and changing the generator power at the time intervals imposed by the signal model. Changing the signal (modulation and power) has important consequences for the data evaluation. For example, every change in the signal modulation results in a change of the duty cycle of the RF signal. This must be considered when the raw data of the field sensors are converted into corresponding field values. Therefore, the program part performing the conversions raw data – field value must know at any time the current duty cycle of the signal. The regulation of the field strength is also affected by the signal statistics. The software collects the field data form the exposure wheels, compares them with the target field strength, and changes the signal power depending on this comparison. Whenever the signal power changes due to an environmental event, the target field strength changes as well. Consequently, all software parts involved in the evaluation of field data must be informed when the signal power changes.

The software must implement the statistical distribution for the six model events. The class SignalStatistic encapsulates the statistical shape parameters and provides methods to obtain the statistically distributed model data.

class GsmSignalStatistic
{
public:
    // Methods to get statistically distributed model data
    //
    int GetTimeTillNextHandoverMsec() const;
    int GetTimeTillNextTargetLevelChangeMsec() const;
    int GetNewTargetPowerLevelIndex(int oldTargetIndex) const;
    int GetFirstPowerStepDurationMsec() const;
    int GetPowerStepDurationMsec() const;
    int GetGsmBasicDuration() const;
    int GetGsmDtxDurationMsec() const;
    bool Notify();
private:
    // Statistical shape parameters
    //
    double m_HandoverDurationLambda;
    ...};

GetTimeTillNextHandoverMsec() returns the time period until the next handover. GetTimeTillNextTargetLevelChangeMsec() provides the time period until the next target level change. GetNewTargetPowerLevelIndex() returns the new target level index. The number of power levels and the power step size define the power range of the GSM signal. Concrete network implementations may have a varying number of power levels and a fixed step size of two dB. This signal model in contrast is more flexible. The number of power levels
as well as the step size may vary. GetFirstPowerStepDurationMsec() returns the first power step duration after a handover whereas GetPowerStepDurationMsec() provides the power step duration for all other steps. GetGsmBasicDuration() and GetGsmDtxDurationMsec() return the duration of the GSM Basic and the GSM DTX modulations respectively.

To provide this model data, GsmSignalStatistic must generate the statistically distributed data. GsmSignalStatistic does not implement the statistics itself, but uses the methods of StatisticalDistribution. This class encapsulates core statistical functionalities.

Below is an example of a method that returns the time until the next handover.

```cpp
int GsmStatistic::GetTimeTillNextHandoverMsec() const
{
    return static_cast<int>(
        ::floor(StatisticalDistribution::
            GetExponentialDistributedNumber(0.1, m_HandoverDurationLambda)));
}
```

Providing just the statistically distributed model data, GsmSignalStatistic implements no triggering mechanism and no real hardware communication. (Signal modulation and generator power have to be changed according to the statistic.) These are the tasks of CyclicExecutor and Experiment. Experiment starts and stops signal model timers of CyclicExecutor using the calculated times of GsmSignalStatistic. The sequence of timer starts and stops to generate the GSM signal is not described here in detail. However, the principle is rather straightforward. Basically, each signal event corresponds to the execution of a timer of CyclicExecutor. Whenever a timer event occurs, the appropriate hardware commands are performed and the time of the next event of this same type is calculated and the timer is started again. Consequently, the timer events sustain each other and generate the synthetic signal.

4.6 Programmatic calibration of monopole field sensors

The PERFORM A exposure system contains 16 exposure wheels, and each wheel contains two monopole E-field sensors. These 32 E-field sensors have to be calibrated before an experiment starts. Calibrating E-field sensors without a software program is a cumbersome and time-consuming task. Therefore, to simplify the calibration process, software has been developed which performs the calibration process programatically.
The goal of the calibration is to establish a relationship between the sensor output $U$ and reference field strength $E^2$. This relationship then allows the calculation of the field strength out of the sensor output.

The E-field sensors are passive and are connected to a detector diode. The detector unit converts the RF power into DC voltage. This signal is the sensor output. The DASY3 near-field scanning system (Schmid & Partner Engineering AG) with a high precision free space E-field probe provides the reference field strength. The software collects the sensor output and the reference field strength. Out of this dataset output voltage – reference field strength, the calibration function can be derived.

This calibration method is suitable until the limit (2 A/m) of the SPEAG probe is reached. Higher fields can be calculated using the change of the input power and considering the input power to be proportional to the square of the field. Considering that the square of the field determines the SAR, the following relationship allows the calculation of $E^2$ out of the sensor voltage $U$ and the sensor offset $U_{offset}$.

$$E^2 = a_0 \left( U - U_{offset} \right) + \frac{(U - U_{offset})^2}{DCP},$$

where $a_0$ is a sensitivity factor [A/m/V] and $DCP$ is the diode compression point [V]. Applying a least square fit on the calibration data enables the calculation of $a_0$ and $DCP$ out of the sensor output $U$ and the reference field strength $E^2$.

The calibration software collects data from the DASY3 near-field scanning system, the E-field sensors, and the power meter, which measures the input power of the wheel. These datasets have to be collected at multiple input power levels. Therefore, the software systematically changes the power of the RF generator. At every power level, the software collects a dataset. At the end of the calibration procedure, the software automatically transfers the data into a Microsoft Excel worksheet. There, a least square fit on the calibration data can be applied to calculate $a_0$ and $DCP$.

The calibration software uses Microsoft's Object Linking and Embedding (OLE) technology to access program interfaces. Thus, the calibration software does not need to implement hardware communication itself but can use the hardware communication of other programs. Figure 4.28 shows a UML component diagram of the calibration software.
4.7 Testing and validation

Large and complex systems like the PERFORM A exposure system must be tested very thoroughly. Since the outcome of the PERFORM A project could significantly influence the policy of WHO concerning the danger of non-ionization radiation, testing of the PERFORM A exposures system required extreme care.

System testing distinguishes between software and overall system tests. Software tests simply try to assess the correctness of the software. Overall system tests on the other hand evaluate whether the system behaves according to the specifications.

4.7.1 Software tests

Most agile methodologies emphasize the importance of automatic tests such as unit tests and integration tests. The central argument is that in the end writing these tests saves time, because errors can be detected much faster with automatic tests than without them. However, this statement is not true for every project. Writing automatic tests for the PERFORM A software would have slowed down the development speed, because the project organization has been very particular in two ways: 1) one single programmer realized the complete software and 2) the determined requirements stayed more or less constant.

Since only one developer programmed the system, he was responsible for the complete source code. Nobody else changed the code. As a result, this developer always had to consider whether changes in one part would influence the other parts. Obviously, automatic tests could help detect bugs faster. However, if only one programmer is working on a project, the amount of time required to write automatic tests is larger than the saving of time that results when automatic tests are used. When more people work on a software project, automatic tests help to keep the source code error free, because a developer can impossibly estimate the effects of changes he makes on foreign program parts; in such a case, automatic tests facilitate the finding of bugs.

If the requirements of a system remain constant and if only few new features have to be implemented in the future, then the use of unit tests does not speed-up the development
process. If no new features are added to a software system, then the system has to be tested only once, and automatic tests cannot deploy their full potential.

In summary, unit tests are more important the more programmers work on a system and the more the requirements change.

4.7.2 System tests

The key system test was to check whether the system generates the desired field strength inside the exposure wheels. To test this, a desired SAR was entered in the controlling software. This SAR value was manually converted using the transfer functions into the corresponding field strength value. Afterwards, a test experiment was started. As soon as a constant power level was reached, the field strength was measured externally using the DASY3 near-field scanning system. The measured field strength was then compared with the calculated field strength. All tests showed excellent agreement (less than 10% deviation) between calculated and measured values.

After these basic tests had been passed satisfactorily, redundancy tests were performed. Selected field sensors were disconnected, so that some exposure wheels had only one intact field sensor. The goal was to see whether the controlling software can detect and ignore the disconnected sensors and still calculate the field strength correctly. The tests showed that the software can detect disconnected sensors very reliably.

This validation method of comparing measured with calculated values may seem too simple. However, it is has to be considered that the controlling software averages and smoothes the field sensor data. Therefore, the software does not simply regulate the power according to the currently measured data.

To assure stable field strength inside the wheels, the software compares the measured field strength with the target field strength. In the case of a deviation, the software adjusts the amplifiers. An exposure unit has eight field sensors (four wheels, two sensors per wheel) and an amplifier. The field sensors provide the measured field strength and the amplifier controls the field strength. Therefore, a mismatch between field data (measurement variable) and amplifier power (control variable) exists. The eight measurement variables must be averaged into one single value. This average field strength then serves a reference value for the controlling of the amplifier power. The software uses a two step moving average algorithm to distill the average field strength. First, the software averages the data of the two sensors of a wheel (this is the averaging at the wheel level). Then, the software averages the data of all four wheels (this is the averaging at the unit level).

The measured and the expected field values may differ because of three reasons:

- incorrect dosimetry. If the dosimetry that converts SAR into field strength is incorrect, the expected field strength will be incorrect as well.

- incorrect Field sensor calibration. If the calibration of the field sensors is incorrect, the sensors will indicate incorrect field strengths.
The last section of this chapter lists basic characteristics of the entire exposure and evaluation software.

- Number of lines of source code: approx. 150,000
- Number of classes: approx. 300
- Number of editable configuration parameters: approx. 2000
- Number of different error messages: approx. 250
5 Evaluation

This chapter’s first part investigates how the quality of EMF bio experiments can be improved when a software-controlled exposure system is used. The second part evaluates the different software engineering methodologies with respect to suitability for daily use and benefit for the software engineering company based on the experience with the development of exposure systems.

5.1 Software for EMF exposure setups

The use of software-controlled exposure systems has heralded a new era of EMF bioexperiments. Software-controlled systems have a large set of advantages over simple manually manipulated systems. Probably the greatest advantage is that software-controlled systems enhance the creditability of the experiments, since software-controlled systems guarantee systematic and reproducible experiments. Additionally, software-controlled systems permit to conduct more experiments in a given period, which in turn enables statistical analyses and thus enhances the significance of the data and decreases the influence of mavericks.

In the past, critics of studies finding negative health effects have often claimed that the exposure conditions had not been according to the requirements imposed by the experiment protocol. Particularly, critics have doubted the SAR dose, the influence of neglected artifacts or they simply have questioned the design of the exposure setup. Software-controlled systems are more immune to overall criticism, since software-controlled systems collect all relevant data and thus augment the significance of the experiments. Systematic evaluation of these data enables the reconstruction of every experiment. Thereby, the exposure conditions can be inspected for every single experiment. Hence, the subsequent control of dubious results becomes possible. Overall criticisms of studies are no longer easily possible with software-controlled systems. Critics must have more profound arguments to discredit a study.

The following paragraph lists the core benefits of software-controlled systems over manually manipulated systems.

- increased creditability and trust. Software-controlled systems increase the creditability at various levels. 1) Biologists or veterinarians conducting the studies are assured that the exposure system acts according to the specifications. For example, they could unplug an RF cable and see if the exposure really stops. Therefore, the people performing the experiments no longer have to rely completely on the statements of the company or research group building the setups. Consequently, the confidence in the entire exposure system increases. 2) Study founders have the assurance that the biological laboratory cannot, be it intentionally or unintentionally, change the
exposure conditions. The laboratory is bound to the experiment specifications. 3) The scientific community can trust the findings of software-controlled experiments. Experiments can be replicated using the same controlling software. 4) The mobile phone industry is assured that high-quality research is performed. No manipulation of data is possible. 5) Consumer organizations and health agencies can formulate quantitative guidelines based on well-quantified exposure conditions.

- flexible signals and exposure protocols. Low frequency signal components are presumed to cause negative health effects. Flexible controlling software allows the generation of RF signals with varying low frequency components. Thus, the influence of the individual components can be systematically examined. Although no correlation between RF power variation and health risk has yet been determined, software allowing flexible on/off exposure time can test whether constantly switching the exposure on and off induces any effects.

- easy handling. Software-controlled systems do not have to be manipulated manually. Therefore, biomedical teams conducting the experiments do not have to manipulate technical equipment and can concentrate on their real work. No possible handling errors falsify the results.

- controlled exposure strength. Controlling software implementing periodical power feedback control guarantees stable exposure conditions.

- monitoring of environmental parameters. Environmental parameters may influence the condition of the exposed subject. Controlling software measuring the relevant parameters can assess the influence of these parameters.

- data logging. Controlling software logging all relevant data enables the reconstruction of the individual experiments out of the stored data.

- human safety. An exposure system must promise maximal safety for the people operating the system. Electromagnetic fields and their danger cannot be seen. People operating an exposure system usually cannot assess the danger of electromagnetic fields. Therefore, controlling software caring for human safety must indicate optically and/or visually when electromagnetic fields are present in the exposure chambers.

Since controlling software plays a central part in modern exposure systems, the software must be tested very thoroughly before experiments are performed. Serious software errors detected at the end of a study could question the complete output of a study.

5.2 Software engineering methodologies

This chapter evaluates the software engineering methodologies presented in the third chapter based on the experience with the development of exposure systems (chapter 4). Generally, software engineering distinguishes two sets of methodologies: heavyweight
5.2 Software engineering methodologies

and lightweight or agile methodologies. These two types have a different approach to the process of software engineering.

Strongly focusing on planning and modeling of a software system, heavyweight methodologies consider the software development process to be predictive. The entire system development process is planned in detail, and the implementation does not start until the planning and design is completed. Based on prediction, heavyweight methodologies define a predictive schedule for the entire development process. This focus on planning assumes that it does not matter who in the end accomplishes the processes. Particularly, the work of the individual programmer should become exchangeable. The plan dictates to the programmers exactly what they have to do and leaves no clearance for individual decisions. Heavyweight methodologies work fine until the requirement specifications of a system change. In this case, the entire development process as well as the schedule has to be updated. However, the adaptation of a process to the current needs is cumbersome and time consuming. Therefore, the nature of heavyweight methodologies is to resist changes. As soon as the process has to be changed, the complete schedule crashes like a house of cards. In today’s rapid economy with the “just in time” method of production, hardly any requirements remain stable for a long time. Therefore, heavyweight methodologies seem to be a relict of a passed time. Nevertheless, heavyweight methodologies still play an important role in today's software engineering business. Not all features of heavyweight methodologies are inapplicable or out of date. Particularly, the focusing on modeling, especially of large systems, is still an important part of modern software engineering. Today, object-oriented analysis and design (OOAD) is by far the most popular heavyweight methodology. Therefore, this evaluation considers only this heavyweight methodology.

In contrast to heavyweight, agile methodologies are adaptive. They welcome changes even late in the development process. However, agile methodologies are not completely non-predictive. The prediction simply refers to a short time span or a so-called iteration. Within a single iteration, agile methodologies follow a well-defined plan. Agile methodologies are the result of an accelerated evolution of the economy and of changed methods of production. Agile methodologies are fast and simple, less document-oriented but more code-oriented and are more people than process-oriented. It seems that agile methodologies are more appropriate in today's economy. Still, most software engineering companies have difficulties using agile methodologies. The four agile methodologies covered in this thesis are Extreme Programming (XP), Crystal Family of methodologies (Crystal), Lean Software Development (LSD) and Pragmatic Programming (PP).

Figure 5.1 shows the visualization of typical heavyweight and agile development processes. Heavyweight methodologies define a fixed goal and try to achieve this goal in as straight-line as possible. Agile methodologies on the other hand do not define a fixed goal. Over the development process, a more improved goal may appear. Being agile and therefore able to react quickly to changing circumstances, these methodologies change the direction of the development process and try to reach the new goal.

The comparison and evaluation of any particular methodology with another is difficult, and the result is often based upon the subjective experiences and intuition of the author. This thesis evaluates the methodologies from the point of view of a professional software
engineer. It explicitly does not try to compare and evaluate the methodologies objectively. Comparison often implies valuing one methodology over another. This is not the case here. The goal is to discover in which phase of the software development lifecycle and under which circumstances the specific methodologies have their advantages and disadvantages.

![Diagram](image)

**Figure 5.1:** Visualization of the process path for heavyweight and agile methodologies. Heavyweight methodologies define a fixed goal and try to achieve this goal in as straight-line as possible. In contrast, agile methodologies do not define a fixed goal. Over the development process, a more improved goal may appear. Being agile and therefore able to react quickly to changing circumstances, these methodologies change the direction of the development process and try to reach the new goal.

Each methodology approaches the problems faced in software engineering from a different angle. In a given situation, a certain strategy provides good a solution, whereas in another situation a different strategy is best suited to solve a problem. No software engineering methodology can be evaluated objectively. The opinion of the author always plays an important part. If an objective evaluation would be possible, the results would most probably not be really useful, since the evaluation criteria would then be hardly significant. Having no objective approach to compare and evaluate the methodology does not imply that comparison and evaluation must be chaotic or completely rely on personal experiences. On the contrary, a formal analysis method is still applicable. Below, the applied analysis is briefly introduced.

The methodologies are compared and evaluated with respect to different criteria of the development process. The selection is based on two considerations. A methodology must be suitable for daily (minimizing the effort to reach the goal) and must generate benefit for the company. These are the two key parameters. Employees will only use a methodology if the methodology helps to create excellent software or if the methodology decreases the work. Which of the two arguments is more important depends on the individual character of the employee. Companies on the other hand will only use a methodology if the methodology promises higher profits or reduced costs. Suitability for daily
use and benefit for the company need not exclude each other. However, it is not a priori evident that they coincide.

The assessment criteria selected based on suitability for daily use and generation of benefit for the company are as follows: Simplicity and low migration costs, requirements engineering process and information gathering, teamwork, documentation and modeling and code reuse.

5.2.1 Simplicity and low migration costs

Similar to business processes, a software engineering methodology only yields benefit if it is actually used in the production process. A new methodology must be relatively easy to learn, adapt and perform to have a chance to be used in practice. Generally, companies cannot afford to slow down or even stop the production process to apply a new way of doing business.

Resistance to change is common in most organizations. What is good for the organization is not necessarily good for the people working there and vice versa. Learning the new methodology and work processes requires more effort than continuing to use existing processes. Generally, people change if the benefits associated with the change outweigh the costs of the change. People tend to overestimate the costs and to underestimate the benefits. People act upon what they believe to be true, not what is true.

Ultimately, all change is made by individuals. If the changes make current skills less valuable, individuals may resist the change because they have invested a lot of time and energy in acquiring those skills and anything that diminishes those skills may be perceived as diminishing the individual (because important skills bring respect and power). This may be a real problem when a company decides to use an agile methodology. A specific programmer no longer has his own sphere of competence where he is the absolute boss. If a company is willing to change the production process, the company must necessarily give convincing evidence for the need of change.

Learning and applying the principles of OOAD takes some time and requires great discipline. However, generally the principles are close to the traditional idea of how to organize work. Therefore, companies may not have fear of contact with this methodology. Since OOAD seems to offer control over the entire development process, the management may favor this method over agile ones.

Agile methodologies should be per se simple to learn and to apply. However, great varieties among agile methodologies exist. XP is a strict methodology and is not open to incorporate business processes typical for a given company. If a company wants to migrate their software production process to XP, the old established organization structures must be abandoned completely. This may be a reason why companies are not willing to migrate to XP. Additionally, XP requires a lot of discipline on the part of the developers. I doubt that a company can get along for a long time with XP. XP simply requires too much discipline. Over time, people will most probably become neglectful and disesteem core principles of XP. The Crystal methodology accounts for company specific production process and allows tailoring depending on project size and criticality. Less discipline
is required to follow the Crystal methodology. LSD does not require large changes in the production process. Rather, the attitudes with which the production process is performed must change. The principles of PP do not interfere with the established production process. PP can be applied in every environment.

Conclusion

The application of a comprehending agile methodology (XP, Crystal) requires a paradigm shift of the company. This is not easy to achieve. If the paradigm shift is too large and supporting adoption scenarios are missing, then it is likely that organizations and developers will not adopt agile methodologies at all. Compared to traditional software development, agile methodologies do not value some topics very highly (planning, flexible code design for future projects, documentation, detailed contracts). Therefore, it is no wonder that the organization as well as the software engineers can get confused when adapting an agile methodology. A completely intuitive and therefore highly subjective simplicity and low migration cost ranking looks like this: PP, OOAD, LSD, Crystal, XP.

5.2.2 Requirements engineering process and information gathering

Software development processes are based on requirements the system must fulfill. If the system requirements are not clear or conflicting, then it is important to gather all relevant information associated in one way or the other with the system. Since the requirements engineering process and information gathering are fundamentally important, every software engineering methodology must address these topics.

OOAD tries to plan the entire development process in advance, whereas agile methodologies consider the planning of the entire development process as impossible. Therefore, agile methodologies are especially designed to handle changing situations. Considering their different purposes, it is not astonishing that requirements engineering and information gathering have different positions in OOAD and in agile methodologies.

Traditionally, OOAD relies completely on requirements. OOAD considers the initial requirements as the basis for the complete development process. Given the central importance of detailed and stable requirements, OOAD strongly focuses on exact analysis and elaborated requirements engineering techniques. OOAD is not based on short iteration cycles, resulting in a long time span until feedback from the customers is available. That is the reason why the initial requirements must be extraordinarily good. OOAD tries to find the requirements and to gather the information in a very technical manner (interviewing, scenarios, prototyping, workplace observations). However, the techniques do not sufficiently cover of the customers' wishes. Certainly, collaboration with the customers is accentuated; however, the manner in which this collaboration is realized seems to be improvable. Particularly, OOAD does not sufficiently consider the psychology of the customers and the analysts. In my opinion, the technique used to obtain information from the customer is less important than how the analyst behaves towards the customer. If, for example, the analyst acts arrogantly, the customer most probably will not try hard to cooperate closely. Another issue is the selection of people for interviews for political reasons. Normally, the management wants to be interviewed. Obviously, this may not
always be the best approach, because people from the management who will not use the system daily may have different preferences than employees who will use the system regularly.

Requirements engineering and information gathering play a less important role in agile methodologies, since these methodologies do not expect to find all requirements in advance. Rather, they are able to adapt quickly to changing situations. In XP, "user stories" acquire the requirements. The customer writes these stories himself. Generally, the stories are simple and must be comprehensible for the customer as well as for the programmer. This requirements analysis is consciously simple, since it is only done to assess the duration until the next software release. Details of the "user stories" are discussed during the implementation phase. XP has a very simple and open requirements engineering process. However, the process only works fine when a customer is on site. The requirements engineering process in XP has some shortcomings. Therefore, another methodology, Extreme Requirements (XR), has been developed. XR is an extension of XP, especially developed to handle the requirements engineering process. The Crystal methodology adjusts the requirements engineering process according to the project size. The definition of requirements in advance is more important than in XP. Therefore, the information gathering process is more formal. Crystal suggests a requirements document, which is continuously updated. Additionally, Crystal proposes the role of a requirements gatherer. LSD does not value requirements engineering very highly. Short iteration and release cycles provide fast feedback. The feedback in turn helps to extract new requirements. PP offers two very valuable tips in the context of requirement engineering. Tip 52 says: "Work with a User to Think like a User" [11]. This is a very good starting point to extract the real needs of the customer. Tip 54 declares: "Use a Project Glossary" [11]. Customers and developers must speak the same language to cooperate closely.

Conclusion

OOAD strongly focuses on formal techniques to find the requirements. However, these techniques often neglect the human component. Agile methodologies emphasize the proximity to customers and try to meet the wishes of the customers. Short and regular iterations enable fast learning and the enhancement of the requirements. However, the agile methodologies do not actively support the person who actually must gather the information. They provide no technique how to gather information efficiently and how to prioritize the information. Expressed simply, OOAD is all the more suitable the more stable the requirements are, whereas agile methodologies deploy their full potential when the requirements change.

Out of my personal experience having worked with very different people, I believe that the personality of the person gathering the information is extremely important. A good information gatherer must have three skills: intellectual, intuitive and emotional skills. He needs intellectual skills to prioritize and assess the importance of the information. Additionally, he needs good intuition. This is difficult to describe and may lack a real scientific background. Over time, good system designers develop a feeling of what will be important in the future. This feeling may be based on experience of the past. However, the experience cannot be grasped deliberately. Nevertheless, a good designer should carefully sense this feeling and not suppress it. I think it is not possible to assess the impor-
tance of certain aspects of a large software system purely systematically and intellectually. Generally, the consideration of personal feelings in the development process can be expressed as follows: "work together with your intuition, not against it". Most of all, a good information gatherer must have emotional skills. He must be very companionable and be able to get along with people with different professional backgrounds (managers, engineers, employees, workers etc.), since each of these people can provide valuable information. Mainly, a successful information gatherer must absolutely not be arrogant. Unfortunately, in the IT environment, quite many arrogant people exist. Sayings like "the silliest expected user" express this arrogance. People have to feel that their concerns are important. Only then will they cooperate openly.

5.2.3 Teamwork

The people working in a software company are the company’s most valuable resources. Their knowledge represents the company’s intellectual capital. The more motivated the people are and the more they are respected by the company, the better their efficiency and hence the more money the company makes. Since the extent of software projects has increased enormously in the past, today a single programmer rarely develops a software system alone. Rather, most professional software is developed in a team. Therefore, forming a project team working effectively together is critical for the overall project success. A successful project team has team spirit. "We-feelings" expresses this team spirit (we created this software together). On the other hand, personal goals must not be neglected. A team has a good team spirit when the team thinking and the personal goals are well balanced.

Today, OOAD emphasizes the importance of good teamwork and considers the people to be an important factor in the production process. This focus on people has most probably been catalyzed by the appearance of agile methodologies, which are very people-centric. However, originally the structures of OOAD have not been designed to put people in perspective. Particularly, OOAD offers no proposals to boost team spirit. In my judgment, OOAD does not really involve people to enhance the team spirit. OOAD restricts the provision of a fertile environment in which team spirit can flourish. People are still regarded as resources which must be handled optimally. The power ought not to be shifted away from the management towards the team.

Agile methodologies in contrast consider the peculiarities of the people and provide suggestions to improve team spirit. Thereby, XP goes the furthest. XP suggests the concept of collective code ownership to enhance "we-feelings". All developers are responsible for the whole system. Any developer can change any source code at any time. Since all developers are responsible for the whole system, the backtracing of an error to its originator is obsolete. In practice, collective code ownership is very difficult to achieve. The bigger a project, the more complicated collective code ownership is. However, collective code ownership is also difficult for some software engineers for personal reasons. They are scared that another software engineer will ruin their work or it may emerge that they are not as outstanding as they themselves believe they are. Additionally, most software engineers will agree, when they are really honest, that it is easy to shift the blame onto a colleague whenever a bug is found. XP also emphasizes the importance of oral communi-
cation and proposes working environments and information structures that cultivate oral communication. The Crystal family of methodologies mainly focuses on regular team discussions to build team spirit. These discussions also help to spread knowledge about the system among the team member. Crystal proposes office facilities stimulating personal communication. Compared to XP, Crystal provides less concrete suggestions to enhance team spirit. LSD considers the people factor as the most important element for a successful software project. LSD accents the responsibility of the whole team for the outcome. This should automatically result in good team spirit. PP pronounces that not only pragmatic programmers exist, but also pragmatic teams. The whole team must be responsible for the software quality. PP suggests that the team should communicate as one. To achieve this, a brand describing the team is proposed. This simple trick, to give a team an identity, may suffice to improve team spirit.

**Conclusion**

OOAD as well as agile methodologies emphasize the importance of good teamwork. However, they use a fundamentally different approach to achieve this goal. OOAD limits the provision of a good environment for the team. Agile methodologies in contrast suggest concrete proposals to improve team spirit. OOAD follows a control-based management style, whereas agile methodologies pursue a management style based on personal responsibility. Agile methodologies try to balance the company hierarchy with the knowledge hierarchy.

Good teamwork automatically results in small truck factors. The truck factor describes the likelihood of a project failure when a single key software engineer is run over by a truck. A truck factor equal to 1 means that the project will fail with a likelihood of hundred percent, whereas a truck factor equal to 0 states that the project performance will not be affected at all. In my opinion, a high truck factor yields to a lose-lose situation for the company and the software engineer. The company becomes hooked on an individual person and has to make concessions to satisfy this person. However, a high truck factor is also bad for the software engineer. He has more pressure and is constantly busy holding his position. Thereby, he does not advance personally and professionally. He burrows into his area of expertise and is no longer open to change.

Agile methodologies make very high demands on individual social competence. Particularly, software engineering must be able to handle criticism constructively. Individuals must be able to criticize others as well as accept critics of others. Agile methodologies follow the ideal that no person in the team is better than any other team member. Additionally they emphasize that working with trust and without disagreements makes work easier for everyone. These are very noble ideals that sound good but that may be very difficult to achieve in the practice.

### 5.2.4 Documentation and modeling

Since the introduction of the UML, modeling a software system has become more important. The importance will most probably even increase in the future. A reasonable amount of documentation and modeling diagrams enhances the understandability of a
large system. Extensive modeling diagrams are especially convenient for software engineers which have to be acquainted with an existing system.

OOAD regards modeling and documentation as very important. The entire development process is based on a set of congruent diagrams. The technical evolution of modeling software favors the use of modeling diagrams. Such software automatically assigns changes in one diagram type to all affected diagrams. Thereby, a congruent set of diagrams emerges.

Agile methodologies are skeptical of too detailed documentation and too extensive diagrams. XP almost completely dispenses with documentation and modeling. Crystal assumes that with increasing project size the documentation becomes more important as well. LSD provides no concrete guidelines about system documentation; however, in terms of lean development, the amount of documentation should be as small as possible. PP articulates that the documentation is an integral part of the overall development process. Generally, software engineers consider documentation at best as an unfortunate necessity and hope that the management will forget about it at the end of the project. Pragmatic Programmers in contrast embrace and welcome documentation. PP explicitly addresses the problem of synchronizing source code and documentation and provides strategies to overcome the synchronization problems.

Conclusion

Personally, I consider system documentation as important. However, comprehensive documentation may lead to static structures which cannot handle changing requirements fast enough. Therefore, a good balance between documentation and small reaction time to changing situations must be found. XP and Crystal possibly weight documentation slightly too lightly, whereas OOAD may rely too much on documentation. PP offers applicable strategies how to document a system efficiently. This focus on concrete suggestions is the major strength of PP. PP not only beats about the bush and emphasizes what must be done, but PP proposes how an activity can be best performed.

5.2.5 Code reuse

Especially for large software companies it is important that source code can be reused in different projects. Thereby, the same code not only does not have to be rewritten, but more importantly, the amount of maintenance is reduced.

OOAD tries to find an elegant solution to a problem. The idea behind this approach is that the same source code may be used in a future project. However, there is the danger that software engineers "play brilliant and lose". In searching for the perfect code, software engineers can forget easily the priorities. They lose themselves in endless design improvements and discussions about possible future requirements.

Agile methodologies do not accentuate finding good and reusable code in the first place. Rather, excellent code is the result of several refactoring cycles. XP explicitly emphasizes that no code should be written for future requirements or projects. Crystal suggests that the design is discussed after every increment cycle. Good design then evolves naturally
out of the discussions. LSD focuses on the current work and does not estimate first-hand reusable code as important. PP provides basic coding rules that may make the code more reusable.

**Conclusion**

Heavyweight and agile methodologies have different strategies to obtain high-quality source code. Heavyweight methodologies emphasize the initial design of a software system, whereas agile methodologies consider excellent source code as the result of an evolutionary process.

From my experience, I know how difficult the creation of reusable code is. Generally, encapsulating and reusing functionalities with little connection to the software system is easier than to reuse building blocks of the system. For example, reusing a class encapsulating statistical calculations in another software system is easy and straightforward. On the other hand, reusing a class encapsulating an exposure chamber for an EMF laboratory study in another software system is very difficult to achieve. Particularly, four features that a building block of a software system must have make the building block difficult to reuse: serialization, error handling, parameter editing and logging functionality. These features are often neglected when talking about class reuse. Thereby, exactly these features regularly make code reuse impossible. I did not find a solution to solve this code reuse problem. However, I think that only the use of a framework overcomes this problem.

**5.2.6 General considerations**

Most agile methodologies are only applicable for a certain team and project size. They are mostly restricted to small or medium sized projects. The Crystal Family is the only methodology that explicitly allows tailoring depending on project size and criticality. Therefore, methodology scalability is a major topic that agile methodologies must address in the future.

In recent years, many new agile methodologies have been developed. Unfortunately, each methodology uses its own vocabulary and terminology. This results in confusion rather than clarity in the development community as well as in academia. The sheer quantity of agile methodologies brings more damage than benefit to the agile movement. In my opinion, many software engineers and academics have used the gold-rush mood around agile methodologies to publish their own methodology. Guidelines enabling a company to select the appropriate agile methodology under the prevailing circumstances are more urgently needed than yet another agile methodology.

Up to now, no large-scale empirical study systematically examining the benefits of using an agile methodology exist. Mostly, the proposers and devotees of an agile methodology effusively describe the benefits of a certain methodology. However, these personal reports must be handled carefully. Often, these reports wrongly conclude that if the methodology is successful under certain circumstances, then the methodology must be successful under other circumstances as well.
No methodology is inherently suitable for all situations and for all organizations. Each methodology has its strengths and weaknesses. Agile methodologies seem to be particularly suitable in situations where the requirements are likely to change in the future [43]. Highsmith [44] believes that the adoption of an agile methodology yields especially good results when the project is likely to fail. It is important that the methodology suit the corporate culture. Companies that emphasize teamwork, open communication and close cooperation may have little difficulties adopting an agile methodology. Companies that rely on clear hierarchical structures and controlling on the other hand may have severe problems adopting an agile methodology, since the established control mechanisms become less important. Particularly, the management may have great difficulties adopting agile ideas.

In my opinion, it is not that important which methodology an organization uses. However, it is important that the organization actually uses a methodology at all. Basic development guidelines help to establish a smooth development process.
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM</td>
<td>Amplitude Modulation</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>BIOEM</td>
<td>Bioelectromagnetics</td>
</tr>
<tr>
<td>CE</td>
<td>Cyclic Executive</td>
</tr>
<tr>
<td>COM</td>
<td>Component Object Model</td>
</tr>
<tr>
<td>DCP</td>
<td>Diode Compression Point</td>
</tr>
<tr>
<td>DCU</td>
<td>Digital Control Unit</td>
</tr>
<tr>
<td>DL</td>
<td>Data Logger</td>
</tr>
<tr>
<td>DMBA</td>
<td>7,12 Dimethylbenz(a)anthracen</td>
</tr>
<tr>
<td>DTX</td>
<td>Discontinuous Transmission Mode</td>
</tr>
<tr>
<td>ELF</td>
<td>Extremely Low Frequency</td>
</tr>
<tr>
<td>EMC</td>
<td>Electromagnetic Compatibility</td>
</tr>
<tr>
<td>EMF</td>
<td>Electro Magnetic Field</td>
</tr>
<tr>
<td>ETH</td>
<td>Eidgenössische Technische Hochschule</td>
</tr>
<tr>
<td>ETSI</td>
<td>European Telecommunications Standards Institute</td>
</tr>
<tr>
<td>EU</td>
<td>European Union</td>
</tr>
<tr>
<td>FPS</td>
<td>European Union Framework Program No. 5</td>
</tr>
<tr>
<td>FTP</td>
<td>File Transfer Protocol</td>
</tr>
<tr>
<td>GLP</td>
<td>Good Laboratory Practice</td>
</tr>
<tr>
<td>GPIB</td>
<td>General Purpose Interface Bus</td>
</tr>
<tr>
<td>GSM</td>
<td>Global System for Mobile Communication</td>
</tr>
<tr>
<td>HF</td>
<td>High Frequency</td>
</tr>
<tr>
<td>ICES</td>
<td>International Committee on Electromagnetic Safety</td>
</tr>
<tr>
<td>ICNIRP</td>
<td>International Committee on Non-Ionizing Radiation Protection</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electric an Electronics Engineers Standards Committee</td>
</tr>
<tr>
<td>Acronym</td>
<td>Definition</td>
</tr>
<tr>
<td>---------</td>
<td>------------</td>
</tr>
<tr>
<td>IIS</td>
<td>Insitut für Integrierte Systeme</td>
</tr>
<tr>
<td>IMF</td>
<td>Intermediate Multiframe</td>
</tr>
<tr>
<td>IT'S</td>
<td>Foundation for Research on Information Technologies in Society</td>
</tr>
<tr>
<td>JAD</td>
<td>Joint Application Design</td>
</tr>
<tr>
<td>LP</td>
<td>Lean Production</td>
</tr>
<tr>
<td>LSD</td>
<td>Lean Software Development</td>
</tr>
<tr>
<td>MF</td>
<td>Multiframe</td>
</tr>
<tr>
<td>MS</td>
<td>Microsoft</td>
</tr>
<tr>
<td>NTP</td>
<td>National Toxicology Program</td>
</tr>
<tr>
<td>OLE</td>
<td>Object Linking and Embedding</td>
</tr>
<tr>
<td>OOAD</td>
<td>Object Oriented Analysis and Design</td>
</tr>
<tr>
<td>OOD</td>
<td>Object Oriented Design</td>
</tr>
<tr>
<td>OOPL</td>
<td>Object Oriented Programming Language</td>
</tr>
<tr>
<td>POM</td>
<td>Polyoxymethylene</td>
</tr>
<tr>
<td>PP</td>
<td>Pragmatic Programming</td>
</tr>
<tr>
<td>PVC</td>
<td>Polyvinylchlorid</td>
</tr>
<tr>
<td>PWC</td>
<td>Power Control</td>
</tr>
<tr>
<td>RAD</td>
<td>Rapid Application Development</td>
</tr>
<tr>
<td>REFLEX</td>
<td>Risk Evaluation of Potential Environmental Hazards from Low Energy Electromagnetic Fields (EMF) Exposure using Sensitive In Vitro Methods</td>
</tr>
<tr>
<td>RF</td>
<td>Radio Frequency</td>
</tr>
<tr>
<td>SAR</td>
<td>Specific Absorption Rate</td>
</tr>
<tr>
<td>SD</td>
<td>Standard Deviation</td>
</tr>
<tr>
<td>SDLC</td>
<td>System Development Life Cycle</td>
</tr>
<tr>
<td>SE</td>
<td>Software Engineering</td>
</tr>
<tr>
<td>SPEAG</td>
<td>SPEAG Schmid &amp; Partner Engineering AG</td>
</tr>
<tr>
<td>TS</td>
<td>Time Slot</td>
</tr>
<tr>
<td>TDMA</td>
<td>Time Division Multiple Access</td>
</tr>
<tr>
<td>TQM</td>
<td>Totally Quality Management</td>
</tr>
<tr>
<td>WHO</td>
<td>World Health Organization</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>XP</td>
<td>Extreme Programming</td>
</tr>
<tr>
<td>XR</td>
<td>Extreme Requirements</td>
</tr>
</tbody>
</table>
Seite Leer / Blank leaf
The following listing shows the complete implementation of the C++ template class MultiValueMap<>. MultiValueMap<> is a template class that is able to store multiple values to a given key. MultiValueMap<> is flexible and fast. The code is executable and can be used directly.

```cpp
#include <map>
#define TEMPLATE template <typename key, typename value>
#define MAP std::multimap<key, std::vector<value>, std::less<key> >

TEMPLATE class MultiValueMap {

public:
    MultiValueMap();
    void Insert(const key& k, const value& v);
    bool Find(const key& k, int pos, value& v) const;
    bool FindFirst(const key& k, value& v) const;
    bool FindAll(const key& k, std::vector<value>& vec) const;
    bool Erase(const key& k);
    bool Erase(const key& k, int pos);
    void Clear();
    int Size() const;

private:
    MAP m_Map;
};

TEMPLATE MultiValueMap<key, value>::MultiValueMap()
{ }

TEMPLATE void MultiValueMap<key, value>::Insert(const key& k, const value& v)
{ 
    MAP::iterator where = m_Map.find(k);
    if (where != m_Map.end()){
        where->second.push_back(v);
    } else {
        std::vector<value> vec(0);
        vec.push_back(v);
        m_Map.insert(MAP::value_type(k, vec));
    }
}
```
TEMPLATE bool MultiValueMap<key, value>::Find(const key& k, int pos, value& v) const
{
    bool found = false;
    MAP::const_iterator where = m_Map.find(k);
    ASSERT(where != m_Map.end());
    if (where != m_Map.end())[
        int vecSize = where->second.size();
        ASSERT((pos >= 0) && (pos < vecSize));
        if ((pos >= 0) && (pos < vecSize))[
            v = where->second[pos];
            found = true;
        ]
    ]
    return found;
}

TEMPLATE bool MultiValueMap<key, value>::FindFirst(const key& k, value& v) const
{
    bool found = Find(k, 0, v);
    return found;
}

TEMPLATE bool MultiValueMap<key, value>::FindAll(const key& k, std::vector<value>& vec) const
{
    bool found = false;
    MAP::const_iterator where = m_Map.find(k);
    ASSERT(where != m_Map.end());
    if (where != m_Map.end())[
        vec = where->second;
        found = true;
    ]
    return found;
}

TEMPLATE bool MultiValueMap<key, value>::Erase(const key& k) {
    return (m_Map.erase(k) == 1);
}
TEMPLATE bool MultiValueMap<key, value>::Erase(const key& k, int pos)
{
    bool itemDeleted = false;
    MAP::iterator where = m_Map.find(k);
    if (where != m_Map.end()) {
        int vecSize = where->second.size();
        if ((pos >= 0) && (pos < vecSize)) {
            where->second.erase(where->second.begin() + pos);
            itemDeleted = true;
        }
    }
    return itemDeleted;
}

TEMPLATE void MultiValueMap<key, value>::Clear()
{
    m_Map.clear();
}

TEMPLATE int MultiValueMap<key, value>::Size() const
{
    return m_Map.size();
}

#undef TEMPLATE
#undef MAP
Seite Leer / Blank leaf
Bibliography


Seite Leer / Blank leaf
Curriculum vitae

Education

Dipl. Natw. degree 4/2000
Eidgenössische Technische Hochschule Zurich ETH, Switzerland
Major: Geophysics

Matura Typus C degree 6/1994
Gymnasium Thun, Switzerland

Professional experience

Graduate Research Assistant 9/2000 – present
Laboratory of Integrated Systems
Faculty of Electrical Engineering, ETH Zurich, Switzerland

Awards


Personal data

Birth: 25 February 1974 in Oberdiessbach, Switzerland
Citizenship: Swiss
Parents: Vreni & Fritz Oesch