Measurement and Estimation of Software and Software Processes

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Simon Moser

von Rüderswil

Leiter der Arbeit: Prof. Dr. O. Nierstrasz
Institut für Informatik und
angewandte Mathematik

Measurement and Estimation of Software and Software Processes

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Simon Moser

von Rüderswil

Leiter der Arbeit: Prof. Dr. O. Nierstrasz
Institut für Informatik und
angewandte Mathematik

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Der Dekan:
Bern, den 28. November 1996 Prof. Dr. H. Pfander

Foreword

From the very beginning of my studies in computer science in 1983/84, I have worked in the software industry as a part time employee. In 1989, when I was just finishing my masters thesis, my employer assigned me the task of consolidating and documenting all the estimation techniques used at my employer's site. We were a software service shop of about a hundred software professionals. You may imagine that I encountered as many techniques as project leaders. I finally decided to document 2 main methods, one for very early estimates, an adaptation of the Aaron method, and one for estimates after detailed system specification which was based on the ideas of COCOMO but counting screens and functions instead of lines of code.

Later in 1989 I was assigned to head a consulting project entitled "software economics". The client was a large retail company's internal software department. The project title suggested a wider perspective than just estimation but I did not yet know much about it. A few weeks before the project's kickoff meeting, I was reading DeMarco's book "Controlling Software Projects" and Symons' "Function Point Analysis Mk II" to get more up to date. In summer 1990, immediately following my masters exam, this consulting project was successfully completed. We implemented a software measurement and estimation programme based on the Function Point metric. In October of the same year, I attended a seminar by Tom DeMarco - and this was enough to definitively make me addicted to the topic.

For 3 years - from summer 1990 until summer 1993 - I was employed full-time in industry and was responsible for metrics and estimation as well as software engineering and modelling techniques for two employers (I switched to my current employer in 1992). During those years I have assessed over 50 projects using the Function Point metric. I developed guidelines for counting Function Points on domain analysis models according to Yourdon's "Modern Structured Analysis" and documented them in a "Metrics Manual".

In 1993, I started my Ph.D. on the topic, because I found that - besides of its great merits - the Function Points also had some serious deficiencies. These mainly lay in its counting ambiguity, old-fashionedness and lack of support for reuse modelling. Now, in autumn 1996, you may read the results on the following pages. The System Meter - "my" newly proposed metric - may be viewed as evolving from of the Function Point metric even though, to many readers, it may appear revolutionary.

During these seven years of being involved - and sometimes struggling - with this fascinating topic, I got to know many inspiring and motivating people. My thanks go to all of them, but especially to: Fritz Beck (thanks for the cartoons!), Stefan Brantschen, Pierre-André Briod, Annemarie Buess, Simon Christen, Dennis DeChampeaux, Adrian Fröhlich, Alfred Graber, Brian Henderson-Sellers, Georges Huber, Christian Hürlimann, Rolf Jufer, Michael Körsgen, Markus Lumpe, Theo Dirk Meijler, Hansjürg Mey, Thomas Moor, Robb Nebbe, Oscar Nierstrasz, Jürg Oehler, Bruno Russiniello, Hansjörg Scheitlin, Peter Schorn, Robert Siegenthaler, Peter Stöckli, Hung Dinh Truong, Åke Wallin, Marius Walliser and Gerhard Wanner. They all contributed to the presentation or contents of this work.

I am also greatly indebted to my parents Hannelore and Hans Rudolf Moser, without whom many things would have been impossible.

And last but not least, I wish to thank my wife Daniela and my children Severin and Maria. They saw me sitting behind books or at my computer more often than not.

This work was partially sponsored by Bedag Informatik, Berne, Switzerland

Table of Contents

1 Introduction

The work presented in this thesis is rooted in practical questions: "What does it cost?", "How long does it take?" Those questions are typically raised very early in the software process - in most cases even earlier than a precise definition of the software's functionality. The need for answers to these questions led to first estimation models for the prediction of work hour effort and project duration around 1974-76. The most widely used models in industry today are Boehm's COCOMO (Constructive Cost Model) and Albrecht's FPM (Function Point Method). Both models evolved out of the 1974-76 models and became popular in the early eighties. Their underlying perception of software and the software process, though, is traditional. Modern concepts like object-orientation, framework or component based development and incremental prototyping - to name a few but important new techniques - are not covered. The starting point of our work, therefore, was the analysis of the established techniques with respect to the new software development strategies: What parts of COCOMO or FPM have become invalid? What ideas are still useful? Based on this analysis we have developed new solutions for the invalidated parts and verified them in an extensive field study. The newly developed items, namely a new measure of software complexity called System Meter (SM), have also been used to propose solutions to other topics than estimation, e.g. productivity analysis and software quality assessment. Those applications, however, have only been tried out in a few sample tests.

Contents of the Introductory Sections

The following introductory sections will first present the underlying economical issues in more detail. Then, the main ideas behind estimation and measurement are presented. The next section is dedicated to metamodels, i.e. semi-formal models of software artefacts that allow us to precisely define measures, followed by a section about evaluation criteria. These evaluation criteria have also been used in the analysis of the field study results. The second from last section contains a summary assessment of existing estimation techniques (COCOMO and FPM) in the light of modern software development practices. The final two sections describe the structure and contents of the rest of the document and summarise some related topics.

Some Words about Technical Terms

The technical terms and abbreviations used in this document will - in general - be explained when they first appear. Please refer to appendix C, the glossary of terms and abbreviations, for an alphabetically sorted list. The ubiquitous and popular term "software metric" refers to the idea of a rule, algorithm or function that takes some "software" as input and delivers a numerical output. The term "software measure" is used as a synonym here, and is generally preferred, because it does not imply the mathematical properties of a metric (which a "software metric" generally does not possess). The plural form "software metrics" is used for the related area of research and practice - including estimation - as a whole.

1.1 Software Economics

Overview

This section presents some economical key concepts behind the questions "What does it cost?" and "How long does it take?". We discuss the importance of estimation and measurement for the software industry and for software science.

The 3 Economical Parameters of the Software Process

From the viewpoint of a software customer, investments into software only make sense when the purchasing cost is considerably lower than the earned value. Whereas the earned value is often hard to quantify, the costs are easily summed up as the payments to the software contractor, i.e. some software producing unit (SPU), be it internal or external to the software customer's company. We will, in order to discuss software economics for our purposes, take on the viewpoint of the SPU. "Costs" for the customer means "earned value" for the SPU. Its costs, on the other side, are mainly the costs for manpower effort of software development/maintenance[1]. To discuss software economics, we furthermore make the assumption that the earned value, i.e. what is paid for a piece of software, is in direct relation to the product, i.e. its functionality and quality[2]. The last parameter under consideration is time. As for every economic process, also for the software process, time to market is a very critical factor for success or failure. To resume, we have 3 parameters in the context of software economics: ❶ cost/effort, ❷ duration and ❸ the product.

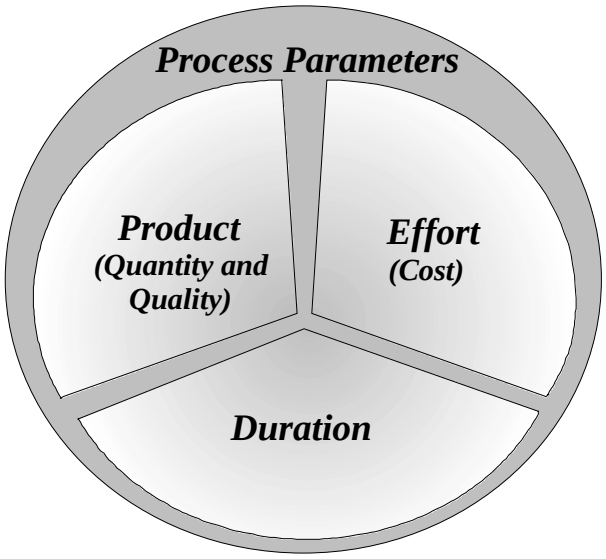

Figure 1: The 3 Parameters of Software Economics

An observation we can immediately make is that each of the parameters is heavily dependent on the others. We cannot, e.g., develop software in half the time with half the effort yielding the same product. Analogous statements can be made for the two other parameters. So we conclude that it is always important to have these 3 parameters and their trade-off dependencies in mind when managing software projects.

Controlling Software Processes

In practice, a manager of a software project is typically assigned the non-trivial task of controlling a software process. This controlling activity spans two basic areas (according to [DM82]) with respect to the 3 parameters:

1. The expectations on the future outcomes.
2. The knowledge of what has already been achieved.

[1] We intentionally omit considerations of investment costs for development hardware and software as well as overhead costs like sales, general management, marketing, financial controlling, etc. in order to stay focused on software issues.

[2] One might argue that functionality and quality are to be considered as two distinct economical parameters. For the sake of simplicity we unified (according to other authors) the two aspects in this introduction.

The first area is the starting point for estimation techniques. It is, however, also an area of negotiation and of compromises. All three economic parameters are usually part of those negotiations. The art of controlling the expectations is to have the estimated values for the three parameters in reasonable balance, in spite of all the pressure from the customers, salespeople and corporate managers.

The second area leads to measurement. Whereas measuring effort and elapsed time are of no theoretical difficulty, measurement of software products is still in its infancy [DU93]. However, besides delivering indispensable input to the management task of the project, it also improves estimation quality, because we will be better estimators when we know how we performed in the past. Only by measuring numerous past processes are we able to conclude statistically how the 3 parameters are related, i.e. what is a reasonable balance and what is not.

Surveys conducted in the eighties [DM83] demonstrated that estimation quality is one of the factors that correlates most with project success. This factor outperforms others like technology used, staff experience, etc. If we give this fact a second thought it becomes quite evident, as shown in the following example: Let us assume that organisation A produces 10 product units a day (PU/D) and organisation B produces 15 PU/D. Furthermore, the customers of organisation A are more satisfied. How come? It is because A based its schedules on its actual 10 PU/D productivity whereas B assumed 20 PU/D. Therefore, B grossly misscheduled its projects. Quintessence: It's better to be slow and know you are slow than to be acceptably fast but think you're Carl Lewis.

One might argue correctly that if the conclusion drawn above would be the truth and nothing but the truth, there would be no motivation in becoming a more productive, faster SPU. In fact, customers not only want high quality schedules and software, they also want them at a competitive cost and - usually most important - they want it fast. So, it is highly probable that B will get the contract when it competes with A, simply because B offers the same software for half the cost. Therefore, A will be forced to rise its productivity in order to persist in the market, while B will either somehow get out of the messy schedules or vanish because of too many angry customers. Quintessence: It's best to be fast and know about that!

We may conclude that estimation and measurement are crucial success factors for the industrial software process. Furthermore, academic computer science could also profit immensely from quantitative methods. The area of experimentation is - in its more rigid definition - only possible when using sound reproducible measurements. Just imagine traditional engineering sciences without kilograms, meters, volts, etc. and you know what is missing in software engineering.

Will Measurement and Estimation Raise Software Quality?

In this short paragraph we will discuss a topic that does not seem central to the thesis, and therefore requires a few words of positioning. The term *software quality* is traditionally viewed in a rather narrow sense. Software is viewed as an isolated thing, isolated from its development process. Quality is therefore restricted to notions like "error freeness", "usability", "extensibility", etc. which we would like to subsume under the term *product quality*. Product quality, however, ignores the 2 parameters of cost and time. Integral software quality, which we would like to call *software process quality*, on the other side, must encompass these aspects. So, the best software (with respect to software process quality) is often only good enough software (with respect to product quality). This thesis, therefore, is aimed to be a contribution to the integral concept of software quality and not directly to product quality. Indirectly though, product quality may profit from the more

reasonable schedules gained by improved estimation and from quality measurements that may be derived from the estimation metrics.

1.2 Estimation and Measurement: How are they Related?

Rational Estimation

Measurement of the 3 economical parameters of effort, time and product do not directly lead to better estimates. Even when we know about our productivity, i.e. the average ratio of product and effort, and about velocity, i.e. the ratio between product and time, we cannot give reasonable values for the parameters before we have estimated at least one.

The first key idea towards estimation is to pre-build a model of the final product. Once we have the model, we can measure its parameters: effort and time of modelling as well as the model's product metric. The second key idea is then to correlate the parameters of the pre-modelling process to those of the full-fledged production process, in hope the former predict the latter. These ideas are not restricted to software engineering. They are not even originated in software engineering, but emerged from the more traditional engineering domains of architecture, ship construction and car manufacturing.

The most prominent examples for rational estimation are the plans drawn by the architects before a house is built. Not only do the plans serve as a specification of what is to be built, not only may the plans be checked against "laws of consistency" (like static laws, minimum allowed distances to neighbour buildings, etc.), but - most important to our objectives - they serve as a predictor model for cost and time of the process of construction. This prediction or estimation is performed by (A) measuring the building's volume on the plan, e.g. in m^3, yielding the predictor value, by (B) multiplying this value with some "magic" factors (usually published by architects associations that derive them statistically), and by (C) adapting the resulting value to specific conditions of the project.

Estimation according to this ABC-scheme is also called measurement based estimation. This is virtually the only rational approach to estimation. It is explicitly used in Albrecht's FPM and implicitly in Boehm's COCOMO. We thus retained and refined this basic strategy in this thesis.

Psychological and Organisational Barriers to Rational Estimation

One question, however, treated in this single paragraph only because it is related to a non-technical topic, could probably fill pages in a thesis on psychology and management sciences: "Why, in practice, is nobody interested in a rational cost estimate?"

Let us analyse the question by having a look at the persons typically involved: 1) a manager (or sales representative) of the SPU, 2) the project customer and 3) the project manager. The SPU's manager is interested in selling the project to raise his bonus. Therefore, he will sell at any price the customer is willing to pay. The customer, in turn, is willing to pay a minimum only. The project manager, finally, when forced to accept the budgeted effort negotiated between his and the customer's managers, will strive to produce as much as he can, and will not waste time with useless re-estimation. Did we exaggerate? Maybe a little, but reality is not far from that.

As a consequence of this lack of interest, we may observe a lack of experience with estimation techniques and even a lack of understanding what estimates are. Very often cost and time estimates are considered to be the minimum values for which the probability is not zero to be reached. Reasonable estimates, however, should be symmetrically biased, i.e. there should be a 50% chance for over- and underestimation.

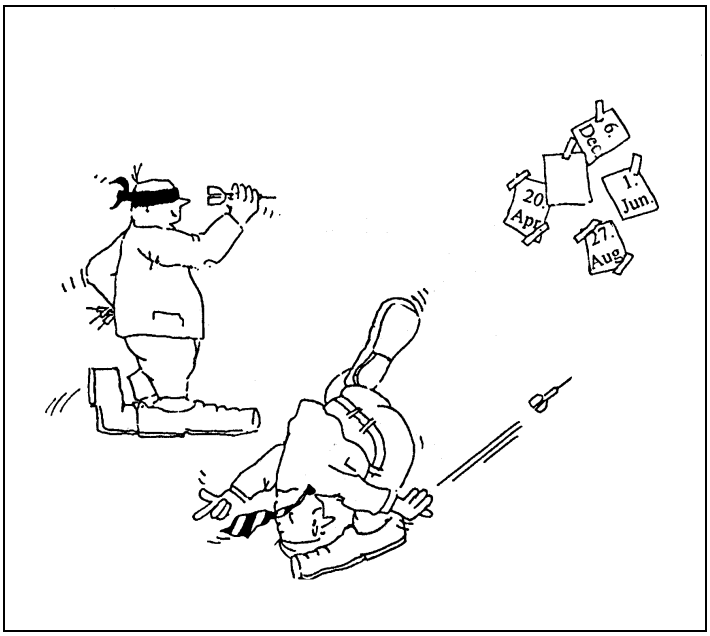

Figure 2: The Day-to-day Software Estimation Process

Another psychological topic is that the knowledge of "truth" about the cost at an early stage may substantially inhibit the further deployment of projects. We postulate that if for every project one had known about the cost as early as rationally possible, roughly every second project would have been stopped in its early stages. Therefore rational estimation can be viewed as a project killer. However, we believe that it does not kill projects but only strips off unnecessary parts, thus helping to assign limited resources to the areas where they are most needed.

This thesis is aimed to be a contribution to one of the many remedies necessary to improve practice. We hope that by stabilising and standardising the technical means of estimation - which is this work's attempt - managers and project leaders will be more routinely willing to apply rational estimation processes due to their reduced efforts. Other non-technical measures, however, must support this approach. DeMarco [DM82] e.g. has proposed the establishment of a metrics team within an SPU that is assigned the single task of producing high quality estimates. We do, however, not cover such managerial aspects in our work.

Some Consequences of the Absence of Rational Estimation

To conclude this section we would like to present the following - rather ironical - scenario of dialogues between A) the manager of an SPU, B) the manager of its customer in project x and C) the project manager to illustrate some of the undesired consequences of non-measurement based estimation:

The estimation process (relaxed version): A and B are talking while playing golf:

B: When will this billing system be operational? Within 6 months?

A: Sure!

B: How much does it cost? We have $100'000 left in our budget.

A: We make it for $90'000.

Tracking the productivity (a quickie): Three months later, same place, same persons:

B: Are you on schedule?

A: Yes!

Revising the estimate and typical means to rise the productivity: another 3 months later, manager A in his office with project manager C:

A: Why didn't you work harder? We missed the deadline and still don't have that system ready. Thanks to me (!) the customer will accept to postpone the deadline by 1 month, but not a single day more! This means overtime for you and your team!!

C: Yes, but ...

A: You wanna risk our jobs?!

The bottom line: bad product quality, high maintenance costs, dissatisfied staff: 4 months later, same place, same persons:

A: @!☠... bugs, bugs, nothing but bugs! Can't you build reliable systems? That customer is really angry with us, he doesn't even invite me golfing anymore. We've now spent $180'000 on this project for which we were only paid $90'000. I'll tell you something: Your bonus for the year is far below freezing and if that system isn't stable within one month you're fired!

C: Uh huh. Interesting. Oh, and by the way, I forgot to tell you that me and two senior engineers will quit our jobs by the end of next week.

A: (Shocked and surprised) How disloyal of you! You can't rely on anybody these days...!

1.3 Measurement and Metamodels: How are they Related?

Metamodels = Models of Models

The measurement of the economical parameters 1 and 2, the effort and elapsed time, are well understood in theory (practical problems remain). On the other hand, the measurement of the 3rd parameter, the product, i.e. the software system, is both theoretically and practically non trivial. The very first problem is raised by the impossibility to measure a thing whose definition is formally (or intuitively) ambiguous. The notion of a software system, however, is such a non rigidly and non intuitively defined thing: to many people a system is a set of functions, to some it is a set of rules, to others a database and to still others a set of objects, etc. Furthermore, there is some confusion about the difference between the system and the system description. For some people there is a difference; for others there is none.

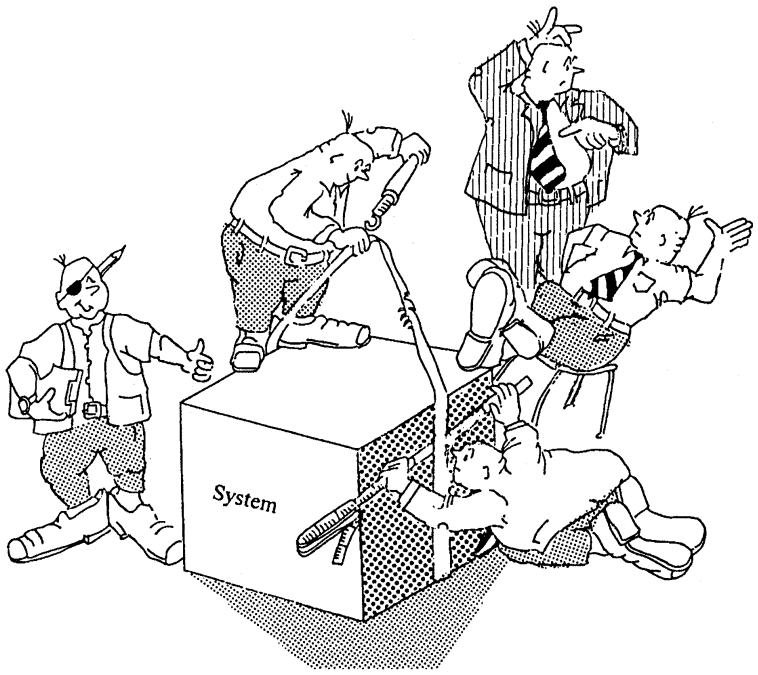

Figure 3: What is a System?

No wonder that the state-of-the-art in measuring and consequently developing and maintaining systems is dominated by "art and chaos" rather than by stable engineering skills. Of course, you might say, we all like art and chaos - but is this still the case when our standard of life or even life itself depends on it?

One of the main reasons for the relative instability inherent in software engineering (SE) is its high degree of abstraction and therefore high need for formalisation. Not all engineering fields are that abstract, e.g. building houses is far more intuitive to human beings and therefore not much formalisation was needed. We observe, however, that the needed SE formalisation is far behind SE applications. As Y. Wand and R. Weber have stated in [WA90] "... developments in the CS and IS disciplines have been inhibited by inadequate formalisation of basic constructs". The topic is complicated by the intriguing fact that software is itself a formalisation, i.e. a model or description of a machine process. Therefore, we first had to develop models of models, i.e. metamodels, to be able to tackle the measurement problem afterwards.

Software engineering, however, may also directly profit from metamodels by enabling it to describe methodologies in terms of the metamodel's object types and by implementing tools that may act upon the metamodel's object types, amongst others CASE tools that may graphically display and edit, for example the two object types "class" and "association". Those applications, though, are not further discussed in our work.

Metamodels in the Context of Estimation

The ABC-strategy of estimation not only requires a metamodel of the final product, i.e. coded software, but also a metamodel of the model of software, i.e. of analysis and design artefacts. We therefore also had to elaborate metamodels of several layers of software artefacts.

In the course of all this metamodelling, the term "software" appeared to be too narrow: we therefore established the new term "system description" (SD). The term "system" was chosen because it does not limit us - especially at the higher layers - to implemented software, but opens the field for real world processes, e.g. business processes. The term "description" was chosen because it should always be remembered that in software engineering we do not deal with systems directly, but rather with man-made artefacts that describe systems.

Furthermore, estimation's step C, the interpretation of the initial estimate with respect to the project structure plan (PSP), requires a model of the production process. For this reason and in order to rigidly define the metrics of effort and duration for the software process we also had to define a metamodel for the software process, i.e. a standard process template. This process metamodel answers often asked questions like "Are project management tasks (and efforts) included?", "What is an experimental or evolutionary or incremental prototyping project?", "What if we leave testing to the customer?", "What if we have to accept the analysis results of another contractor?", ...

Again, in practice, it is often observed that neither the products nor the process of the project are well defined, i.e. they do not obey some metamodel. This fact makes the application of sound metrics more difficult and thus diminishes the metrics' profits. Obeying to a metamodel, however, is not a hard thing to achieve. It just requires some discipline. It does not mean that everything should be planned into the last detail but that the substantial landmarks of the project landscape are recognised and reflected. A software engineering project should not be considered a wasteful journey to nowhere as cartooned below:

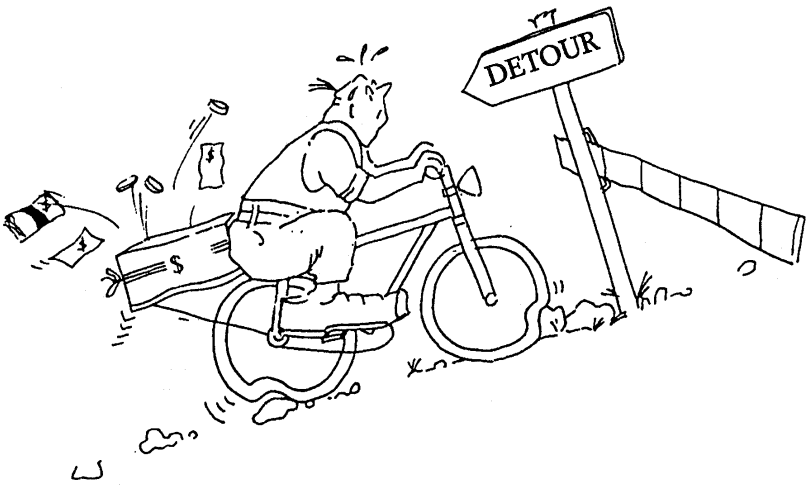

Figure 4: The Day-to-day Software Process

1.4 Distinguishing Good and Bad: Evaluation Criteria for Estimation Techniques

Criteria for software estimation evaluation may easily be derived from the fact that it is an economic process, too. The key parameters for estimation are therefore: effort, time and the product, i.e. the number and quality of the estimates. In order to simplify the situation, we assume that estimation is performed by a single person, so effort and elapsed time will be unified. We furthermore assume that only one value is estimated, so the number of estimates yielded is irrelevant. Thus two criteria remain:

1. Estimation effort
2. Estimation quality

Put simply: estimates should be produced quickly and be close to the actual outcome that is later observed.

According to the ABC-strategy of estimation, we must take the pre-modelling effort into account for the overall estimation effort. This pre-modelling effort is usually the dominant component when adding up estimation efforts. We may conclude that good estimation methods with respect to that criterion, therefore are those that require a minimum of pre-modelling activities. Our work focuses on that requirement.

On the other hand, better values for the second criterion, estimation quality, will be achieved when the underlying models are more detailed. This trade-off between criteria 1 and 2 is inherent to estimation and cannot be overcome by any technique. Another problem with this second criterion is its measurability (note the recursion of problems). We therefore had to develop, based on ideas by DeMarco [DM83] a measure of estimation quality, the estimation quality factor (EQF), which - to our belief - should belong to every management metric suite in the context of software processes. The EQF, however, has the drawback of being measurable only after the completion of the process and it can only be used to assess the estimation technique that was actually applied in the early days of the project. If we have to assess a new estimation technique, however, - as we had to - other paths have to be followed: whereas the EQF can be applied regardless of the rationale behind the estimation process, we strictly relied on measurement based techniques. This allowed to substitute the EQF with the bias of the correlation. In the case of estimation of the effort out of some metric of software complexity, this means that both metrics are

measurable entirely after project completion, thus allowing the assessment of a new measurement based technique, with no need to ever actually apply it.

Besides the two economically derived criteria there also exist several subordinate criteria for estimation techniques [KN91], e.g. 1) repeatability, 2) understandability, 3) adaptability, 4) documentability, 5) independence of human estimator, etc. However, we do not believe that these contribute to the ultimate clarification of the topic. They either are trivial (e.g. repeatability which must be given for everything that calls itself a "technique") or they are not objectively measurable themselves (e.g. understandability). We therefore focused entirely on the two criteria previously outlined.

1.5 New Trends in Software Development: Are Existing Solutions Still Valid?

One might wonder what is wrong - in the context of modern software development techniques like object technology (OT) - with the two most widely used methods, COCOMO and FPM, for software sizing and estimating. In order to analyse the two methods, we first quickly summarise, and second, comment them with respect to general criteria. Then we recapture the dominant new trends in software engineering 1) object orientation, 3) framework or component reuse and 3) prototyping with respect to the methods.

COCOMO (B. Boehm 1981, [BOE81])

When using COCOMO, you have to start right away with the most unsound step of this method, the heuristical estimation of the number of lines of code (LOC) your project will encompass. Then you calculate an initial effort using Boehm's statistically derived exponential function. Finally you adjust that effort by multiplication factors that are determined by attributes of the project (so called "cost drivers"). The multiplication factors are again statistically taken from Boehm's empirical database. The cost drivers mainly are: 1) product type, 2) run-time environment, 3) development team, 4) development tools. Several refinements were made to the original method by Thebaut and Jacote that did not change, however, the basics just outlined [DC96].

COCOMO's pre-modelling step is reduced to a heuristic and merely mental process wherein one imagines the future software solution. This is fast and cost efficient, but inherently unreliable. The next two steps then are bound to Boehm's empirical database which may substantially differ from a database valid for a concrete situation (situative variations range in the order of a magnitude, i.e. a factor of 10, according to [DM88]). Besides yielding good results for some SPUs, COCOMO is - if observed without emotion - mainly used as an alibi: without generating to much effort, it may yield virtually any desired value and at the same time may be called a method. This critique is severe, but not new [DC96]. The main idea we retained from COCOMO is its classification of cost drivers that may be used to categorise empirical data.

Function Point Method (A. J. Albrecht 1979 [AL79])

The FPM assumes the existence of a domain analysis model. First (A), the function points are counted assuming the following underlying concept (or metamodel) of systems: there are database structures (relations, persistent classes if you like) on one side and, on the other side, functions that access those structures with the 4 basic database operations create, read, update and delete (CRUD). Both parts, i.e. the database and the functions, get historically founded fixed points per data element and per access according to a complexity rating (easy, medium, complex) associated with the element (e.g. a create access of a complex data element yields 6 FP). In case your system does more (or less) than CRUD,

you are offered a set of a limited number of so called influence factors. These influence factors are also historically founded and vary the unadjusted function point count by ±30%. Second (B), you access an empirical database from which you take approximation factors appropriate for your situation. Then you can calculate an initial effort estimate. Finally (C), you adapt the initial effort to project specifics. Again there were several refinements published on top of Albrecht's original method. They range from minor revisions by Albrecht himself, over different versions of counting guidelines from the International Function Point User Group [IFPUG] to major changes in Symons' FPA Mk II [SY88] [SY93], Capers Jones' Feature Points [JO85], Harry Sneed's Object Points [SN94] and the newly developed Task Points [SBC95]. The basic assumptions of the FPM were not touched. Besides terminological adaptations to new software engineering concepts and refinements (and complications) of the categorisation guidelines there is only one major enhancement in Jones' Feature Points. He allows for function specific adjustments for complexity instead of only one system wide correction factor.

Like the COCOMO method, the FPM has several drawbacks - not as severe as COCOMO's - that are not related to OT:

1) the unnecessary historical factors (a recent Finnish study [KI93] showed, that correlation quality to the effort remains the same, even when they replaced all historical factors with a constant value of 1)

2) the old-fashioned terminology, inhibiting interpretation of new concepts

3) high interrater[3] and intermodel biases (cf. [KE93]) due to the "non-standardness" of the method

4) its inherent limitation to database or IS (information system) applications [SY88].

The enhancements mentioned were not always positive, too: Symons, for example, proposes that the historically fixed points per element be periodically re-calibrated. Besides being a tedious and labour-intensive task which - as a side effect - prohibits or at least complicates the comparison of old projects with new ones, it is also indifferent to estimation quality.

Assessment with Respect to Object-Orientation

Object-orientation is a buzzword. Nevertheless, we try to define it as a 4-tuple of the concepts of 1) information hiding, 2) data abstraction, 3) inheritance and 4) polymorphism[4]. COCOMO's lines of code metric does not reflect any of these concepts, neither do the original function points. Therefore, the complexity of systems described using the concepts of object-orientation cannot be captured by these measures. Some of the newer Function Point counting guidelines, however, offer means to cope with inheritance of data attributes by including them in the complexity ratings of the inheriting classes.

[3] This term is used in [KE93] and means "between different human measurers".

[4] We mean by these 4 concepts:

1) Information hiding = restricting access to implementation details for "clients"

2) Data abstraction = bundling data structures and functions into classes

3) Inheritance = the relationship between two classes where one "inherits" the data structures and functions from the other (actually we should distinguish between subtyping and inheritance, refer to 3.2.2.3 for details)

4) Polymorphism = the fact that the same function signature may be differently implemented down the inheritance relationship, thus leading to different behaviour patterns of objects depending on their class, i.e. polymorphic behaviour.

Classes, though, are viewed as data holders only. We therefore have only a minimal support for object-orientation. This assessment, of course, is no surprise due to the age of the methods.

Support for object-orientation as an assessment criterion for estimation methods on the other hand, is questionable, because the impact for estimation is low. The pre-modelling needed for estimation cannot ever yield full-fledged object-oriented system descriptions and hence object-oriented concepts need not be fully supported by the estimation predictor metric. A metric's applicability, however, should not be limited to estimation, just as the "normal" meter to measure distances not only serves to estimate the duration of journeys, but also it should help to assess the quality of (object-oriented) system descriptions. This is definitely never possible with the LOC or FP metrics and we therefore had to investigate new approaches.

Assessment with Respect to Framework and Component Reuse

General purpose and application specific frameworks and component suites are becoming more and more important these days. Software construction can profit immensely from the reuse of well-tested components with respect to product quality and cost of production. The question is "What is a reused component?". We distinguish between active and passive reuse according to [KA92]: 1) active reuse will propagate changes of the reused component to the reuser, 2) passive reuse will not. Typically, copying and modifying a template is a form of passive reuse, whereas inheriting and method or function calling are active reuses. For our purposes we make the following categorisation: 1) any system description created or added to within the software process is not reused, i.e. project specific, 2) any other system description that is part of the resulting product is a reused element. Reused system description elements are further classified as language or library elements, according to the somewhat fuzzy definition that all highly standardised components are language components (e.g. all C++ language elements - which are also part of every final C++-written product). The links of the newly developed parts to the reused parts are also system description elements and therefore, of course, part of the newly developed ones.

This very flexible attitude to reuse, where almost 0% or almost 100% of a resulting product may consist of reused elements, is - superficially though - supported by one of COCOMOS's cost drivers. The FPM, on the other hand, offers no means to model reuse situations. Reuse interpretation is left to the heuristical step C. One notable exception are the Feature Point enhancements of Jones [JO85]. The complexity adjustment per function allows one to model various degrees of reuse. We tried to implement a similar approach in our proposed new method. However, we did not merge the concepts of complexity and reuse but introduced a distinct modelling instrument because, in reality, complexity and reuse are distinct concepts, too.

Finally, the recently emerged concept of design patterns [GA94] is nothing but reuse of system descriptions at a higher level of abstraction, i.e. of one of the pre-models. The key idea behind design patterns is that repeated elements of design - and if possible even analysis - are identified, named and described. We may then reuse patterns to model our software system at design level. Because this idea is completely analogous - using the notion of system descriptions - to reuse at implementation level, the same comments as above apply.

Assessment with Respect to Prototyping

Prototyping processes may be viewed as reduced versions of the full-fledged development process. They are often instantiated as sub-processes within a regular development process: 1) experimental prototyping in the early phases to support decision making, 2) evolutionary

prototyping later to test the behaviour in simulated production situations and finally 3) incremental prototyping to verify small but growing versions in production. The latter term actually is a little bit misleading because the resulting system is never a tinkered prototype, but a fully working, high quality and documented product. The functionality however, according to the old motto "act small - think big", is only a subset of the ultimately planned functionality.

In COCOMO and the FPM - as well as in our newly proposed method - the support for prototyping lies in step C. The prototyping process templates are compared to the complete process model and the initially estimated efforts reduced accordingly. While the FPM and COCOMO leave the underlying process metamodel open, we tried to have ours defined. This should make our step C more reliable.

Furthermore, one aspect of the FPM is increasingly becoming a disadvantage: it requires a domain analysis pre-model, for which typically around 15% of the overall development effort are already spent. For the reduced cycles of prototyping this may be too late to come up with a sound estimate. And - for some categories of prototypes - no such model is ever established. Therefore, we tried to keep our new method, especially the required pre-modelling step, as simple and effortless as possible.

Summary of the COCOMO and FPM Analysis

This brief review of the existing solutions to the estimation problem showed that investigation into a new method for estimation was - and probably still is - desirable. Not only the newly emerged development techniques of OT required this work, because both established major estimation techniques, COCOMO and the FPM, have other inherent flaws. COCOMO's deficiencies - besides its merits - lie in the metric it uses, which is not available as a measurement early enough. Its mechanics are applicable only after the main estimation step (i.e. the derivation of the LOC value) is done. Reuse is only scarcely supported by COCOMO and the FPM. The FPM - which is considered to be superior to COCOMO - in turn requires a non-negligible pre-modelling effort and is inherently limited to the database metaphor. This motivated the design of a new general purpose, historically unbound and reuse based estimation method that requires a minimum of pre-modelling.

1.6 Our Contributions: Contents and Structure of this Document

Overview and Analysis of Existing Solutions (contents of chapter 2)

Our work started with a thorough analysis of established techniques of estimation and measurement. In section 2.1 we start with some in-depth discussions of the measurement based estimation strategy (as outlined in introductory section 1.2) followed by its two by-products: ① the ability to chain estimates, i.e. to use estimated values as predictors for still other estimated values, and ② the backward application of the correlation, i.e. to answer questions like "How much software can we produce given a certain cost budget?". We may then leave estimation and shift our focus to measurement. All characteristics of an estimation technique depend on the underlying metric, provided it follows the measurement based strategy, which is assumed to be a 'conditio sine qua non'. After briefly discussing basic concepts and terms of measurement, we will summarise the "Goal-Question-Metric" (GQM) approach [BA88] for introducing metrics within an organisation. This widely known approach is positioned as a superimposed framework for our more technical work. We then briefly analyse evaluation criteria for measures of complexity. We argue that some of the well-known criteria - Weyuker's criteria [WEY88], to name the most prominent suite - are inconsistent themselves (as do [ZU91] and [FE91]) and are mainly focused on formal aspects. For our purposes the semantic quality of a metric with respect

to product measurement estimation (besides some indispensable formal legitimity) is most important. We therefore propose a new metric evaluation strategy that is documented within the "New Approaches" chapter. The last topic of section 2.1 is dedicated to a framework [HS93] that enables researchers and practitioners to get information about usage, applicable model layer and granularity of (object-oriented) metrics. This framework may be used to support the GQM/AMI process. In particular it enabled us to systematically identify areas of neglect.

Section 2.2 is entirely dedicated to an analysis of the most prominent complexity metrics currently used in industry and academia. According to our newly developed evaluation technique of "metamodel analysis" (cf. 3.1.2), the metric evaluation is closely tied to the analysis of the corresponding metamodels. We start with the simple and well-known lines of code (LOC) metric, which is used in COCOMO and many other estimation schemes. The LOC metric- even though formally sound - suffers from an overly simplistic software metamodel. According to the historical chronology of metric invention, we then present the cyclomatic complexity metric of McCabe [MC76]. It is mainly used as a quality threshold metric but also as a predictor for maintenance efforts [OM92]. Its underlying metamodel is flowchart oriented, thus lacking many modern aspects of software engineering. The next metric suite analysed is Halstead's so called software science metrics [HA77]. Some of Halstead's ideas are unsound formally and with respect to human cognition. Nevertheless, the metamodel analysis showed the best coverage of engineering concepts among the implementation layer metrics. The function point metric, used in the FPM and invented by Albrecht, is the only metric widely used on a non-implementation layer. Behind mostly sound and useful ideas, our assessment revealed a few formal and some more serious semantical deficiencies which were already outlined in introductory section 1.5. An assessment of the newly proposed Chidamer/Kemerer metrics suite for object-oriented design [CHI94] is also given, even though those metrics were not originally intended - and therefore are not used, to our knowledge - for estimation purposes. In spite of this fact, the analysis was fruitful, because we could use the underlying metamodel, first presented by Wand and Weber [WA90], as a starting point for refining our metamodel of object-oriented systems. This section concludes with some remarks about other approaches.

The last section of chapter 2 focuses on the state-of-the-art in software process science. The more or less well understood metrics like effort, duration, etc. and the corresponding metamodels are made explicit, commented, analysed and rigorously defined. The known concepts of process dynamics, i.e. consciously setting a tighter schedule than estimated, are then discussed with respect to estimation. The metric impacts of the different metamodels for development processes and subprocesses, i.e. the waterfall, spiral and fountain approaches as well as the subprocess and span activity metaphor, are analysed in the last three sections.

New Approaches (contents of chapter 3)

We continue our work by proposing and applying new approaches to the neglected areas identified in chapter 2. Our main contributions are: ① the adaptation of metamodelling to metrication in order to semantically analyse and derive metrics, ② the derivation of a completed metamodel of object-oriented system descriptions at several modelling layers on the basis of the Wand/Weber metamodel, as well as the corresponding derivation of a new metric, the System Meter, and ③ in the context of software processes, the definition of a new metric for estimation quality and of the restrictively cyclic BIO (Bedag Informatik/Object) process [MO96b]. This last contribution had to be made to support sound measurement and interpretation of effort, cost and time values.

In section 3.1 we introduce the notion of a metric's metamodel, which can be roughly described as a metric's minimal domain model; minimal with respect to the formal definition of the metric. This notion can be used for two primary purposes: 1) When a metamodel of some domain is elaborated, we can use this metamodel to derive a metric of complexity, i.e. quantity and quality, for the domain. A metric of complexity should encompass all of the metamodel's object and association types. 2) When - the other way round - a proposed metric of complexity for a certain domain is given, we can compare the metric's metamodel to the independently elaborated domain metamodel which usually is a superset of the former. The more object and association types that are covered, the more semantically complete the metric is with respect to the "real" domain complexity. Besides the detailed description of these two metamodel applications, metric derivation and metric assessment, we also describe a newly elaborated short-cut to derive metrics with related metamodels, the metamodel mapping.

Section 3.2 contains the main part of this thesis. Based on a completed metamodel of object-oriented system descriptions a new measure of software complexity is introduced, the System Meter (SM). The metamodel encompasses

① objects, i.e. instances, variables, constants, etc.,

② classes, i.e. abstractions for types of objects,

③ features, i.e. the attribute abstractions that make up one part of a class,

④ methods, i.e. the functional abstractions that make up the other part of a class,

⑤ formal parameters, that make up the formal signature of a method,

⑥ messages, that usually make up a method's body by calling other methods,

⑦ actual parameters, that reference objects in messages.

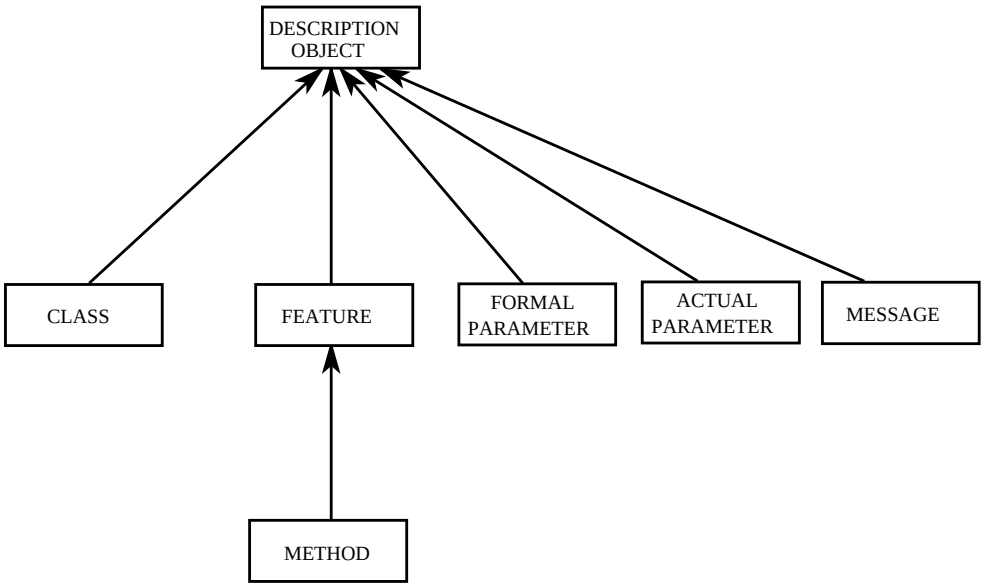

Figure 5: Type Hierarchy of the System Metamodel

We may see from the type hierarchy above that everything ultimately is an object, i.e. a description object. We chose this term to avoid confusion with active (or passive) objects in a running system. Note that every description object, be it a class, a method, a "normal" object or whatever, may be passed as an actual parameter to messages. We may therefore describe every system "from scratch" using these metatypes. We first have to define our (programming) language in terms of classes and methods. Next, we build up our reused components and form the library by sending message calls to language methods and using instances of language classes. Finally, on top of the reused parts, we describe the newly

developed elements which, again, are built by sending messages to the already defined methods. When starting a project, the language and library descriptions are by definition already finished. Note that the links between newly developed and reused parts are established through messages which themselves are description objects. Those links therefore are also part of our metamodel. Of course, they always belong to the newly developed elements. Also note that some of the language methods - and maybe even the library or project methods, in case the initial language supports a metaobject protocol (MOP) - are meta-methods. Meta-methods are those that accept abstractions, i.e. methods, features or classes, as their parameters. Meta-methods are typically used to describe new abstractions, e.g. "from scratch" or using some composition mechanism like inheritance.

In order to stay compatible with more conventional approaches to programming, we do not prescribe that every meta-association[5] should be actually used in a language. For example we distinguish between a type graph and an inheritance graph between classes. For concrete languages like Smalltalk this distinction and the possibility of a network, i.e. multiple inheritance, is not used. In BASIC we have even more restricted class-like concepts: there is no possibility of inheritance or subtyping at all, but still our metamodel can be applied.

The new metric, we called it System Meter, consists of two parts: ① the external size of software objects, and ② the internal size of software objects. The external size of a software object, for example a class, is determined by the complexity of the object name or signature. We propose to count the tokens within the name to capture this aspect. The internal size makes use of the dependencies that exist between software objects. It sums up the external sizes of all the other objects an object is dependent on. In order to take into account for reusable parts, reused objects only count for their external size when summing up the overall size of a system. Project specific objects count for both, the external and internal sizes. With this definition, we can capture virtually every degree of reuse at the finest possible granularity. Other modes of reuse, for example copying and modifying templates of classes and methods, as well as writing a modification of an existing system are conceptually supported by the System Meter, too.

While the System Meter's metamodel is rather complex and detailed, the metric itself is essentially a simple token count. This has two tremendous formal advantages: ① The System Meter is a metric of absolute scale, i.e. virtually every mathematical and statistical operation may be applied to it (cf. 3.2.1.3). ② The System Meter can be uniformly applied to all various types of artefacts: classes, methods, single messages or - at the finest level - actual parameters. This dimensional uniformity allows the safe definition of derived measures, for example the ratio of the total message size per total class size, in a system incorporating all possible combinations of description object types.

Besides a detailed description of the ideas just summarised, section 3.2 also contains the definition of the System Meter for the various modelling layers. In order to use the new metric as a predictor in estimation - our initial goal - we are even more interested in those definitions than in the one for a complete, implemented object-oriented system description. We used the metamodel mapping technique (cf. 3.1.3) to derive those higher order versions of the System Meter. The layers correspond to the layers of the BIO process (cf. 3.3.3):

[5] The meta-associations are not shown here in figure 4 for the sake of understandability of the basic ideas. Please refer to the detailed descriptions in section 3.2.

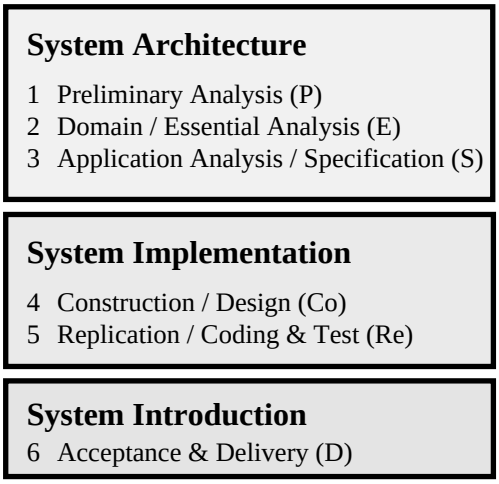

Figure 6: The Layers of the BIO Software Process

The strategy of measuring on different layers is similar to the following methods of measuring the distance between, for example, Seattle and San Diego: To get a first rough estimate, you take a very high scale map and simply measure the direct (or flight) distance. Then you might be able to take a somewhat more detailed map and measure the sum of the major highway sections. Then you use several even more detailed maps to measure the road distance as drawn on the map. Finally you take your car and drive every bend and eventual unexpected deviation resulting in an ultimately accurate measurement on your distance meter. Note that all those distances are measurable using the same metric, the "normal" meter (or some equivalent like the British mile). We have attempted and achieved an analogous behaviour for the System Meter.

In the final section 3.3 of the chapter we first present the definition and some empirical data on the EQF, the estimation quality factor. The main idea behind this metric is that the factor takes on a maximum value of 100% that is best reached when you estimate as early as possible the effective value. Re-estimates are honoured as soon as they officially appear in the project plans. Then the concepts of restrictively cyclic process models are discussed. The main idea here is to open up the current and later layers to the current process phase but to prohibit changes to layers already released. This will lead to steadily smaller cycles, e.g. the domain analysis results will, once released, not be permanently re-questioned (except for defect-removal). Such a process metamodel can be instantiated as a pure waterfall to a fully iterative model. A concrete metamodel based restrictively cyclic process, the BIO process, is then described in order to give the reader a reference to what activities are included in the effort values of the field study.

Results of a Field Study (contents of chapter 4)

Finally, we tested our new approaches in a field study that encompassed 36 projects. While we could evaluate the System Meter estimation method on preliminary analysis results and domain analysis results for all projects, this was not the case for the reuse analysis because we only had 4 framework projects in our sample base.

In section 4.1 we first briefly characterise the sample projects with respect to cost drivers like tools used, application domain, development and customer team, method used, etc. Then we present the instruments we developed and used for measuring the values. One utility - built in C++ - was used to scan system descriptions and count the System Meter values (as well as some of the function point values). The other toolset consists in several EXCEL spreadsheets that regress pairs of predictor and result values, e.g. the System Meter at domain analysis and the effort of the software process. Afterwards we present the

detailed procedure - including two formal languages, PRE and DOME, to denote preliminary and domain analysis results - of assessing the estimation power of the new approach. Then the procedure of measuring the effects of reuse on productivity is outlined. The section finishes with an empirical analysis - however based on only four samples - of the effects of project dynamics. This last analysis has nothing to do with the System Meter method but was done to verify the "mythical man month" rule first formulated by Brooks [BR75].

Sections 4.2 and 4.3 present the findings at the preliminary level and domain level, respectively. At the preliminary level the new method performs excellently with respect to low pre-modelling and estimation effort. Overall estimation quality, however, is slightly poorer than that of the traditional FPM which - as its drawback - may only be applied after the more costly pre-modelling level of domain analysis. With respect to non-database or framework development projects (a subset of 7 samples), the new method performed better with respect to estimation quality. The 7 samples, though, do not allow statistically significant conclusions to be drawn. The System Meter after domain modelling - thus being equally costly as the FPM - yields the most precise estimates with respect to the overall sample base. It also yields substantially better results than the FPM for the database subset (29 projects). All these findings are very promising. We will try to expand our sample base in the nearer future to be able to come up with results that are more statistically significant.

Conclusions and Outlook (contents of chapter 5)

Besides summarising our findings, we propose in the outlook section the definition of a tentative suite of System Meter derived quality measures. The most prominent and useful will be the coupling and cohesion measures that seem to correspond well to psychological findings about human cognition of textual information like code or any of the system descriptions [HS96]. We will certainly try to continue research in this direction, thus leaving the realms of estimation for which a satisfying solution seems to have been found.

1.7 Related Topics

Productivity Tracking

Productivity tracking is the task of telling about what has been achieved by some software development process. Typically this is done by weekly or monthly progress reports wherein cost expended, manpower consumed, time elapsed, problems encountered and measures taken are documented. The problem usually comes when we are asked about the state of the 3rd parameter, the developed product: "Ooh, well, ..., well,.. we're doing fine!" or "60% is completed because 60% of the effort is spent, isn't that logical?". These are two of the less satisfying possible answers we can give.

Estimates of the product size X, however, or measurements of its pre-model size Y, could yield more sound statements of progress. Using X, we can calculate the ratio of the already finished product part X'. Thus, percent finished equals to 100% x X' / X and productivity is X' / effort spent. Using Y, we can measure the part Y' of the pre-model for which our product is already finished. Thus, percent finished equals to 100% x Y' / Y and productivity is Y' / effort spent. All this is straightforward provided you have a sound and easy way of measuring products and models as a whole and partly. One such way is the newly proposed measurement rules of the System Meter.

Measurement and estimation of software and software processes might therefore enable project managers to give statements of progress like: "Our design is 60% completed (for

600 out of the estimated 1000 classes we have a validated design), and we've spent 55% of our estimated design effort.".

A drawback of this very direct way of measuring progress and productivity is that it neglects the actual product quality. But as long as the production process is running we cannot empirically measure the final product quality. For final productivity measurement, though, there are ways to include product quality aspects. One might argue that the production process includes reviews, module tests, integration and system tests, etc. Thus quality would already be accounted for in the effort. But all those steps may 1) inherently not find every bug, 2) be executed poorly or 3) be omitted under time pressure. Productivity as defined above will rise and may support bad practices when the second and third variant apply. However, we can define another kind of productivity, usually called productivity II. It is collectable after a certain period of product usage and maintenance and measures the unchanged, i.e. bug free, part of the product only:

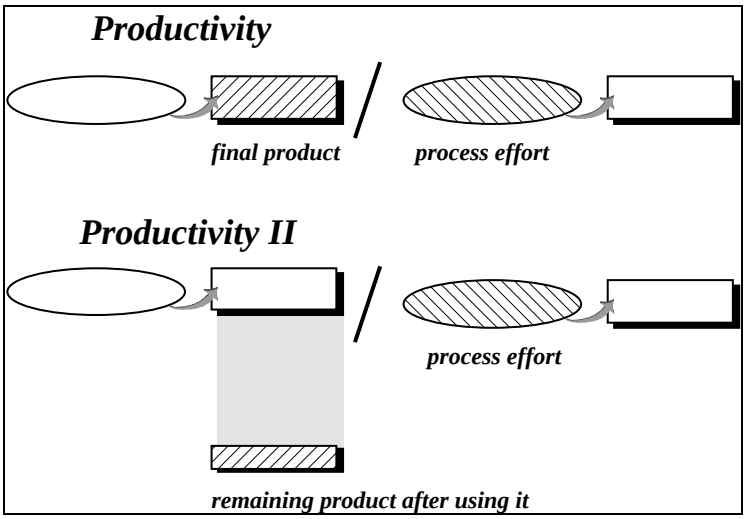

Figure 7: Productivity and Productivity II

Things get more complicated when requirements change, but we will not discuss that issue here (cf. last paragraph of 2.1.1 for some generic strategy to cope with changes). We conclude that measurement and estimation are the only sound basis to unbiased productivity control.

Quality Management Systems

Measurements are needed as feedback data for self correcting software processes as defined by quality management standards (like ISO 9000ff) or by Humphrey's "Software Process Maturity Levels" 4 and 5 [HUM89].

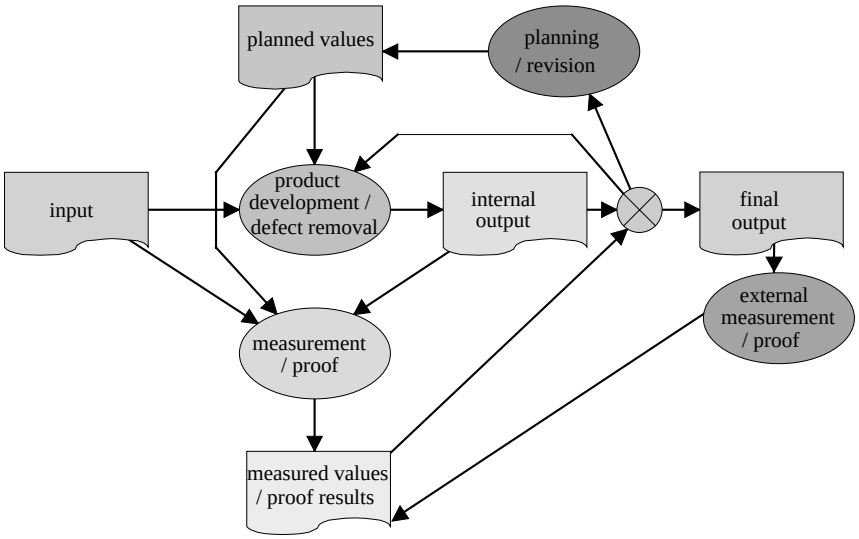

Figure 8: A Single Process using Feedback from Measurements

Even for a single project, i.e. without any superimposed quality management processes, one can profit from measurable process or product goals. Thus, instead of saying "I want to be faster than last time!" we can state: "I want to produce 2 FP/elapsed day which is 10% better than last year's average.". The preconditions of such feedback cycles are however: 1) standards with respect to product and process metamodels and 2) standards with respect to metrics and test procedures. Those standards are typically set by a superimposed quality management process:

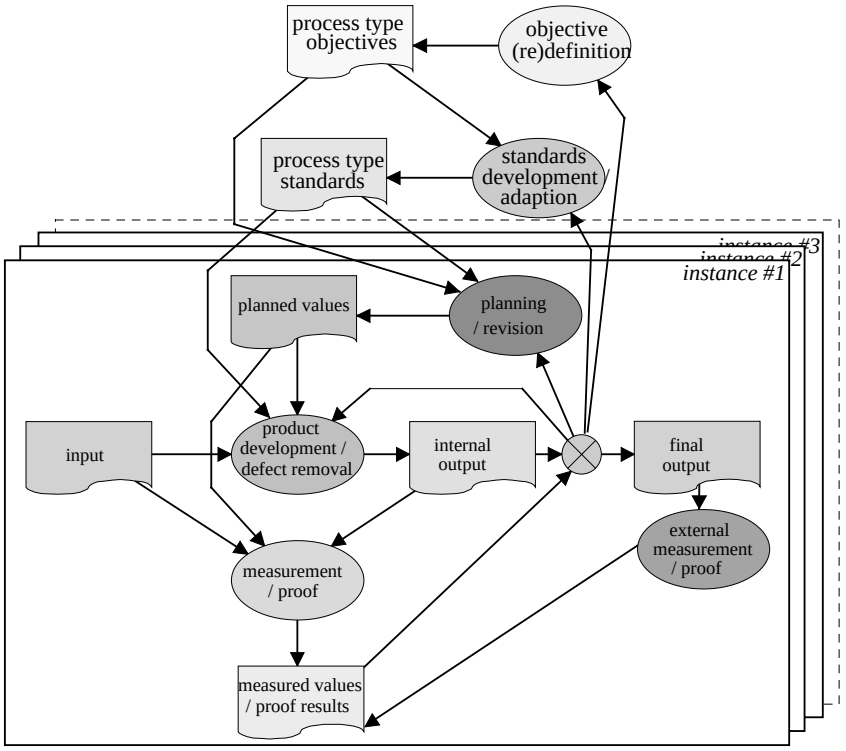

Figure 9: Multiple Instances of a Process Type, Controlled by a Quality Management Process

Only by permanently setting goals and measuring outcomes on enough instances of a certain type of processes (for example prototype processes, full development processes, maintenance processes) one may take full advantage of measurement and estimation techniques. The potential to improve productivity and quality is still vast.

Product Quality Assessment

The measures used for this purpose are usually called quality measures as for example the ratio of comment lines to the total lines of code. To be precise, they are called *inherent quality* measures, because they may be measured on the product isolated from the producing process. In general, it is not easy for inherent quality measures to distinguish between good and bad values, because there are always unproved hypotheses behind those measures. In our example: The more comment lines in the code, the more understandable it will be. This may, or may not be true. Another annoying property of inherent quality metrics is their tendency to proliferate. Everybody can invent lots of metrics - and everybody did so.

In contrast, *empirical measures* of quality directly measure the facts, e.g. the ratio of the effort for a new programmer to "learn" a coded system to size for capturing the aspect of understandability. Those measures, however, have their drawback, too: they are measurable after the fact only.

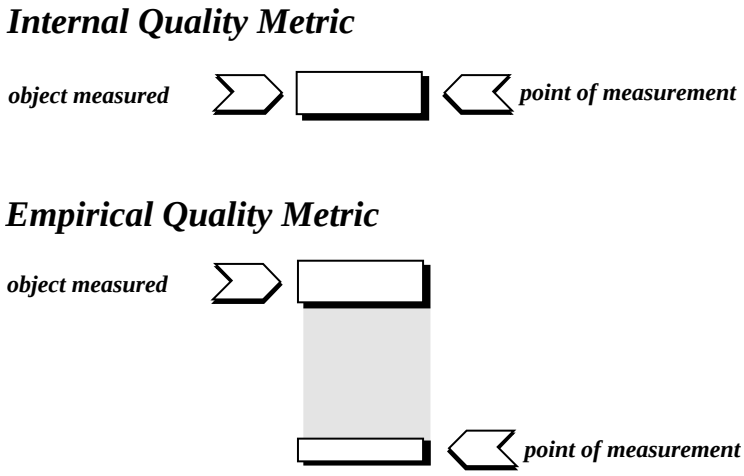

Figure 10: Inherent Quality Measures versus Empirical Quality Measures

Empirical quality metrics may usually be defined as some ratio of measures of size for the product or process. A typical empirical metric is the error-freeness of a product, i.e. the ratio of the unchanged, i.e. error-free, part after some period of use to the original size. Typical inherent metrics of software quality are coupling and cohesion (cf. 5.2). In order to confuse things, many inherent metrics of quality have names with the ending "-ability", e.g. understandability, maintainability, etc. The "-ability" idea, however, is ultimately empirical. So one should be careful unifying the empirical idea with the inherent metric because they might correlate poorly.

The use of empirical measures allows to quantify the empirical facts ex post. The use of inherent quality measures, on the other hand, should be limited to two areas:

1 Setting alarms with empirically derived thresholds, e.g. when the comment ratio in a module sinks below 5%, one has to have a look at that module.

2 Estimating the empirical quality measures before they are measurable.

We conclude this introductory chapter with a citation from Brian Henderson-Sellers: "Beware, software metrics are deep and muddy waters!" [HS96].

2 Overview and Analysis of Existing Solutions

In this chapter we will summarise existing solutions for estimation and measurement of software and the software processes. They are also analysed and commented with respect to our new approaches, with respect to underlying concepts and finally with respect to our practical experiences with them.

2.1 Generic Concepts of Estimation and Measurement

This section is about fundamental established techniques of estimation and measurement. Absolutely central is the idea of measurement based estimation in three steps (ABC-steps). After presenting this technique and some add-ons (calibration, estimation chaining, backward estimation) we leave the field of estimation and focus on measurement with presenting basic terms, usages and categories of metrics. The GQM/AMI approach, evaluation criteria for metrics and metamodelling techniques are finally outlined and commented with respect to our approaches.

2.1.1 The Basic Estimation Process (ABC-Steps)

The generic estimation strategy (adapted from [DM82]) is a procedure of the three steps A, B and C:

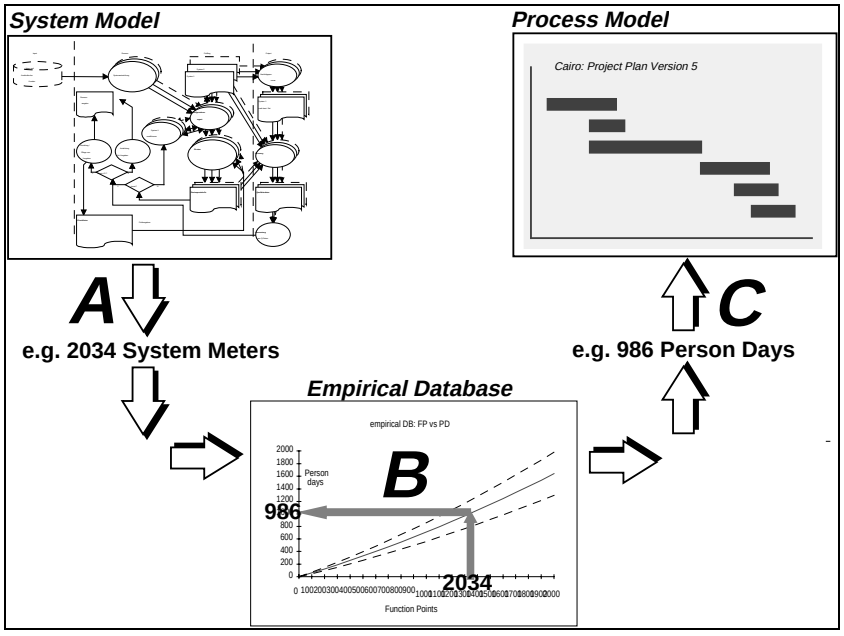

Figure 11: The Generic ABC Estimation Strategy

Step A: Quantification

In step A, we make a quantification of the complexity or size of the system model at hand. We accomplish this by applying a software complexity metric, e.g. the newly proposed System Meter, on the system model. Often the expression "metric of size" is used for those types of software metrics. It should, however, rather be called "effort inducing property", because the notion of size, adopted from non-software domains, is not necessarily related to effort (e.g. it requires more effort to build a small watch than to construct a simple wooden doghouse). However, we will also use the term size because of its popularity but will keep the remarks above in mind.

The ABC estimation strategy - as already stated in the introduction - is not only suited to estimate software process efforts, it is a generic procedure. Virtually any quantitative

parameter of any domain can be estimated this way. Thus, instead of the terms 'system model' and 'metric of size', we may also use the more abstract terms *predictor model* and *predictor metric* to express what those concepts are for, namely to predict the outcome of some future result. For the explanation of the following steps, let the measured value be called ***p***, measured in any predictor metric unit PMU.

Step B: Estimation

The second step is estimation in its more narrow sense. We apply an *approximation function* A (usually but not necessarily a polynomial) to the predictor value ***p***. Function A is sometimes also called the *estimation function*. The estimated value ***e*** may thus be derived as:

$$\boldsymbol{e} = A(\boldsymbol{p})$$

This value is also called the *result value* and is measured in some result metric unit RMU. The effective outcome - measurable later - should be as near as possible to ***e***. However, in general we observe an error or bias. The estimation model therefore not only consists of the function A but also of the *mean approximation bias* dA, commonly expressed as a relative percentage. For the sake of simplicity we assume[6] in our work a symmetric 95%-confidence model (which equals 3 times the standard deviation) to express biases. For example the indication dA = 0.5 (equivalent apparently to 50%) means that the effective outcome of the estimated parameter will lie with a probability of 95% in the interval $[\boldsymbol{e} \cdot (1-(0.5/2)), \boldsymbol{e} \cdot (1+(0.5/2))]$. The mean approximation bias is one of the two crucial empirical evaluation criteria for estimation models (cf. chapter 4, the field study).

Step C: Adaptation

When we have, after step B, a statistically derived estimation ***e*** for e.g. the effort of the software process, we are, however, not finished. The *process model*, i.e. the activities we want to undergo, is usually non-standard. The last step, thus, consists of an adaptation and interpretation of the result value - to abstract again from the software specific terms - to the *result model*. The typically raised topics are:

- the risk we can and/or want to take, that the effective outcome will be higher (or lower) than ***e***
- interference from other parameters of possibly higher priority (e.g. with time constraints as dealt with in dynamic cost models, cf. 2.3.4)
- the effects of omissions or additions of the result model with respect to some template result model, e.g. the tailoring of the software process model

The first aspect of risk may be covered by not taking ***e*** as the estimate but rather some other value out of the range $[\boldsymbol{e} \cdot (1-(dA/2)), \boldsymbol{e} \cdot (1+(dA/2))]$. If we would take ***e***, the risk of a higher actual outcome will be 50%, if we take the maximum $\boldsymbol{e} \cdot (1+(dA/2))$ this risk will be reduced to a mere 2.5%. Any value in between yields risk rates according to the characteristics of the Gaussian distribution [RI78].

The second aspect of interference is typically solved with applying additional estimation models (again according to the ABC-strategy) that estimate the amount of interference. A sample of such an interference are the so called project dynamics (i.e. tight delivery

[6] This assumption is supported, but not strictly proven, by visual analysis of the distribution of the samples taken in our industrial work and field study. More rigid statements might be needed by theory, for practical purposes, although our pragmatic approach seems sufficient. For the distribution of the estimation errors using the new PRE System Meter method we conducted a Chi-Square-Test in chapter 4. This test supported the hypothesis of an underlying Gaussian distribution.

deadlines required by the customer and/or management). They are treated with so called "dynamic cost models" (cf. 2.3.4) which are nothing but adapted instances of the ABC-strategy to the parameters of process dynamics.

The adaptation to the tailored result model is the last mentioned here, but nevertheless the most common one. In general, to give an example, we do not have to undergo every activity of the template software process[7]. Thus, the result value ***e*** is to be distributed by percentage onto the remaining parts of the result model. These distributions are conducted by a chained application of the ABC-estimation strategy (cf. 2.1.3).

This last step of interpretation is not to be confused with aspects of pricing, negotiating and selling software development projects. The estimates - even after step C - are reasonable and rational. The conversion of those rational estimates of effort and time into deadlines and prices is - and should be treated as - a separate process yielding results of another quality. We do not further comment on this rather non-technical process.

Prerequisites to the ABC-steps

Several prerequisites are implicitly assumed in the descriptions of the three steps above:

① *the system model (predictor model)*

It is not possible to yield accurate estimates when only the project's name is known. Even though this is self-evident when stated as above, it is what a lot of people expect from the estimation process. Rough requirements, though, must at least be described or modelled, i.e. we must elaborate a *system model* beforehand, in analogy to architects' plans before building a house. Furthermore, the system model should be described in a consistent way, i.e. conforming to a model of modelling (metamodel). If the metamodel varies, i.e. shows a bias, the models will show a multiplied bias. Thus, they will be less useful for the estimation process as shown in [KE93]. This is the first reason why metamodelling (cf. 2.1.7) is an important technique in the context of estimation. Further reasons will be given below.

The more refined and detailed the system models are, the more precise the estimates, e.g. for effort, will be. But, the modelling is not for free, i.e. it requires some effort itself. Therefore, when starting a project, we first elaborate a coarse model, e.g. the preliminary system description (cf. 3.3.3), to minimise this initial effort. Later in the software process we will stepwise elaborate more refined system models, i.e. system descriptions, that allow refined estimates. This is one reason for the partitioning of the software process into phases[8] (cf. 3.3.2). Other equally important aspects of modelling - even though not further elaborated in our work - are the verification and documentation of customer requirements by means of these models.

Furthermore, the second crucial empirical evaluation criterion for estimation techniques (cf. chapter 4, the field study) is tightly connected to this pre-modelling.

[7] In our work we used the BIO [MO95c] software process as a reference to which the various concrete processes were mapped. This template process follows a phase-like (not strictly waterfall, though) procedure with 6 main steps (partially derived from OMT [RU91]): ❶ preliminary analysis, ❷ domain analysis, ❸ application specification, ❹ technical design, ❺ implementation and test, ❻ installation and system introduction. The process also covers all the management (project, quality, configuration) and documentation activities (for more details cf. 3.3.3).

[8] The effort spent for the first phase, usually a preliminary system description, remains to be intuitively estimated. It is recommended [HS94] to appoint experts from both worlds, the application domain and software engineering areas, and from all participating organisations, the customer(s) and contractor(s), with this preliminary modelling in a workshop like manner. The time available should be rigidly limited in so-called time boxes. The effort is thus usually calculated as number of persons times length of the time box.

The estimation effort, which is used as the criterion, mainly consists of the pre-modelling efforts. The ABC steps are usually of marginal influence because they are automatable at large. As we will see under prerequisite ③ below, there is also some effort needed to elaborate a process model. This, however, is also a minor effort, because we usually can take a complete template model and only minimally adapt it. The efforts for maintaining prerequisite ②, the empirical database, finally does not contribute at all to the effort directly attributable to an estimation process. Those efforts contribute to the overhead costs of an SPU and are not considered in our evaluation.

② *empirical databases*

For the second step it is necessary to know how we have performed in the past, in order to achieve estimates of acceptable accuracy. This knowledge is present in the form of a regressed function **A** (usually a polynomial). **A**'s coefficients are calculated out of so called *empirical databases* wherein pairs of measured parameters (e.g. 1. the system size after preliminary study and 2. the effort of the rest of the software process) are inserted after each software process termination. Thus, those databases contain empirical data. The process of entering data pairs and regressing approximation functions is called the *model calibration* process and is described in the following subsection 2.1.2. Note, however, that for any SPU it is in general unwise to blindly adopt empirical databases from other SPUs or from industry surveys. Productivity rates (i.e. the coefficients of A) vary within an order of magnitude, i.e. with variance factors from 1 to 10, from SPU to SPU [DM88]. Estimates gained with the wrong empirical database, thus, are most likely to be useless and even dangerous to project success.

③ *the process model (result model)*

In order to successfully execute step C, the adaptation of the initial estimate, at least a rough process model - in analogy to the system model - must be defined, i.e. we must elaborate a *result model* beforehand. The result model defines what we are going to do, i.e. what exactly we are estimating. As the system model, the result model should also be described in a consistent way, i.e. conforming to a metamodel. This is the second reason why metamodelling (cf. 2.1.7) is an important technique in the context of estimation.

Based on process or result modelling alone, another more heuristic approach to estimation may be defined: the bottom-up approach. Even though we do not further elaborate this approach, it is important to know about it. The key idea is to model the subject of estimation in such detail that each part is tiny enough for an ad hoc estimation. The total estimate is then a simple sum. This approach has its advantages, i.e. it is fast and can be adapted without further refinement to virtually any domain. However, we do not consider this a sound approach for several reasons: 1) it entirely depends on heuristic modelling, 2) it entirely depends on heuristic estimates and 3) it cannot be improved by systematically collecting empirical data and using statistical means.

Estimation Models

After the discussions above we conclude that an estimation model may be viewed as a 5-tuple:

Definition of an Estimation Model:
Estimation Model ≡ (PMM, PM, eDB, RM, RMM)

wherein PMM denotes the predictor metamodel, PM the predictor metric, eDB the empirical database, RM the result metric and RMM the result metamodel.

Dealing with Changing Predictor and Result Models

We all know that systems evolve. When the change substantially affects the predictor model we should re-estimate using the following strategy:

1 Mark the outdated parts of the old predictor model as "deleted", measure them yielding $\boldsymbol{p}^-$.

2 Make a copy of the old model and delete the obsolete parts. Add the newly wanted system features to the model and mark them with "added". Measure them yielding $\boldsymbol{p}^+$.

3 Enter a "two predictor single result" empirical database that yields a result value.

This conceptual approach to changing systems [DM82] was, however, never tested in our work because it would have required the construction of additional measurement and regression tools. Instead, in practice, we either relied on $\boldsymbol{p}^+$ only and used it as a single predictor, or - more often - we measured the new model entirely, yielded the estimate $\boldsymbol{e}$ and then adapted it for the remaining process parts. Both of these pragmatic approaches delivered useful results.

2.1.2 Calibration

Calibration means establishing and maintaining empirical databases. Before we go into the details of the process of calibration, two statements about empirical databases and their biases have to be made:

1 The empirical databases contain - made explicit in form of A's coefficients and the bias dA - all information that contribute to the correlation of predictor and result but are not covered by the metrics. Therefore, the less rigid those two metrics are defined and the less aspects of the corresponding metamodels they encompass, the bigger will be the *inherent bias* of any empirical database. Knowledge about this fact immediately led to adaptations of the metamodelling technique for metrics (details cf. 3.1), i.e.:

- the derivation of predictive metrics for any domain by first elaborating that domain's metamodel and then defining the metric in such a way that every metamodel entity and association is covered.
- the assessment of a metric for its predictive power by comparing the coverage of its metamodel with respect to the model's metamodel.

2 Many other factors (the cost drivers from Boehm's COCOMO [BOE81]) that may not be covered by the predictor or result models and metrics, will also determine the quality, i.e. biases, of the empirical databases. The main factors in the software engineering area are, ordered by increasing influence [DM88]:

① the tools (programming language, compiler, CASE tools, etc.)

② the methodology (SA/SD, Jackson, OMT, etc.)

③ the infrastructure (rooms, technical and non-technical equipment)

④ the people (developers and (!) customers with individual and team skills)

Those factors are virtually unique at every site and in every project, which is the reason we cannot capture them in a sound quantitative way. The more those factors vary in an empirical database the less probable it is that the coefficients of A reflect a situation close to the one given in the project to estimate. Therefore each SPU should ideally build its own empirical database. On the other hand, there are the statisticians

who reject any statistic based on less than 30 independently measured samples. In practice it is unlikely, though, to observe more than 30 projects with identical tools, methodology, infrastructure, organisational conventions and people involved. Therefore a variance in those factors will always remain, leading to the *environmental bias* of the empirical database.

Establishing an Empirical Database

The first step when establishing an empirical database is to chose the two metrics to correlate, e.g. 1) the system size at preliminary analysis level, measured using the System Meter and 2) the effort of the rest of the software process, measured in Person Days.

Second, we determine heuristically the shape of the approximation polynomial. In our work we restrict ourselves to two shapes:

1. *The purely linear model*

   This model is chosen for predictor value pairs of immediate connection, e.g. to estimate the effort of the whole development process out of the measured effort of some already completed phase. Note that in the example just given, an effort metric is used in both roles, predictor and result. Be aware that in general any metric may appear in any role depending on the purpose of the estimation model. The purely linear approximation function has the form:

   $$\boldsymbol{e} = c_1 \cdot \boldsymbol{p}$$

   The linear coefficient c_1 entirely determines the approximation function A in this case.

2. *The quadratic (increasing) model*

   This model is chosen for predictor value pairs of presumably loose connection, e.g. to estimate the effort out of the measured size of some system model. The quadratic approximation function has the form:

   $$\boldsymbol{e} = c_0 + c_1 \cdot \boldsymbol{p} + c_2 \cdot \boldsymbol{p}^2$$

   The approximation function A is thus determined by the constant coefficient c_0, the linear coefficient c_1 and the quadratic coefficient c_2.

We decided to use these two models mainly for practical reasons. From a theoretical point of view this decision is reasonable because every other function (exponential, logarithmic, ...) is calculated using a Taylor-polynomial (or some more sophisticated but equivalent technique) whose most dominant parts are typically the constant, linear and quadratic summands.

In case we have to model a decreasing relationship (e.g. between effort and time), we do so by reversely applying the quadratic shape. Backfiring (cf. 2.1.3) is then the normal way of using the estimation model.

As the next step, we decide which of the environmental factors, i.e. cost drivers, to keep constant. As already stated, the more factors we can keep constant, the less bias we will observe. In an advanced SPU it is typical to maintain separate empirical databases for every major programming environment used, for each development team and each customer segment. It was already reported [HU89] that empirical databases per individual developer were used. Note, however, that the more specific your database is, the less value pairs you will be able to enter. Theoretically at least 30 independent value pairs are needed for sound regression. In practice you may start - carefully though - with 5 to 10 measurements.

Next, we have to enter measured value pairs (cf. next paragraph on maintaining empirical databases) of projects completed in our SPU into the database. In the case where we have

too few values, i.e. less than 30 pairs, we may take values from surveys, partner SPUs, or we may randomly generate pairs that fit the (A, dA) behaviour gained with the few "true" values.

Approximation, finally, is achieved through polynomial regression techniques. Refer to any statistics textbook (e.g. [RI78]) for the formulae. Because small projects should have small estimation biases we used relative biases and also regression with respect to relative biases. The constant coefficient was always set to `0,` because when there is no system to be built, there may be no effort spent. Early industrial experiences with non-`0` coefficients led to unsatisfactory results for medium to small projects. The idea of fixed costs per software process instantiation therefore had to be abandoned. Purely financial models might yield other results but were not the topic of this research.

Maintaining an Empirical Database

Maintaining an empirical database consists of entering and removing value pairs (and subsequent re-regression). When measuring predictor models one should either measure the historical one, i.e. that was used for the estimate, or the current one. The corresponding result value is - to switch to the most important example of the effort metric: 1) either the effort as if there were no changes to the original model, or 2) the effort as if we always worked on the current model. One other important aspect is that when measuring result metrics, e.g. the effort, the effects of step C have to be eliminated before the data are put into the empirical database. This is called the *normalisation step* of the calibration and can be viewed as step C^{-1}, i.e. the inverse of step C.

Removing data pairs is another important activity. Most important is to remove old data. Old in this context means outdated with respect to the cost drivers, i.e. tools, techniques,

A last note for the practitioner: While the predictor model is usually documented somewhere (watch out for changing models, though) or can be re-established from the running system, this is usually not true for the result model, i.e. project plans and measured efforts. Be very careful when being given some value for effort and always ask questions like "Are the project management efforts included?", "Are analysis efforts included?", "Are quality assurance efforts included?", "Are documentation efforts included?", etc. Only after being sure of what kind of effort you were given, can you soundly accomplish the normalisation step C^{-1}.

2.1.3 Chaining and Backfiring

Two add-on techniques to the basic ABC-steps of estimation are "chaining" and "backfiring". The first technique reduces the number of estimation models and - as a consequence - the number of empirical databases to maintain by composing two or more basic models to form a combined model. The second technique is useful for some special situations where the result value is known before the predictor value, e.g. when a strict effort budget is given but the size of the system to be built is not yet known. Backfiring will then enable us to plan according to the "fit-to-cost" [DM82] paradigm, i.e. the maximum size of the product is estimated out of some fixed effort budget.

Chaining

The need for chaining arises when a first estimate, e.g. the effort of the total software process, is correlated to another estimate, e.g. the effort of the technical design phase. Instead of correlating each predictor metric, say n metrics, to each of the phases, say m phases, thus yielding $n \cdot m$ estimation models, we may reduce this number by only correlating each predictor to the total effort and the total effort to each phase, thus, yielding n+m models. We may estimate using two approximation functions A_1 and A_2 as follows:

$$\boldsymbol{e} = A_2 (A_1 (\boldsymbol{p}))$$

The chaining, denoted as operator ⊗, of two estimation models EM_1 and EM_2 as 5-tuples (PMM_1, PM_1, eDB_1, RM_1, RMM_1) and (PMM_2, PM_2, eDB_2, RM_2, RMM_2) may be defined, under the condition of RM_1=PM_2 and RMM_1=PMM_2, as

Definition of Estimation Model Chaining ⊗:

$$EM_3 \equiv EM_1 \otimes EM_2$$

with (PMM_3=PMM_1, PM_3=PM_1, eDB_3=$eDB_1 \otimes eDB_2$, RM_3=RM_2, RMM_3=RMM_2)

The chaining is thus mainly reduced to a chaining of the empirical databases, which in turn is a chaining (denoted ○) of the approximation functions and biases according to:

$$A_3 = A_1 \circ A_2$$
$$dA_3 = dA_1 + dA_2$$

Most important is the negative fact that the biases of chained models are summed up! We will, nevertheless, use chaining mainly for estimation of 1) elapsed time out of estimated effort, and 2) efforts of phases out of total effort.

Backfire Method

The empirical database may also be accessed backwards by using the reverse approximation function A^{-1}. This is useful when the result metric is given, for example an effort budget, but not the predictor metric, for example the system size. The reversely estimated predictor value can then be used, for example, as an upper limit of system size that may be built within the effort budget. This technique is simple but especially useful in negotiation processes with the customers. If they tell us to build a stock management system for $1'000'000 but do not specify what exactly to build, we can estimate the maximum system size, for example 1'800 Function Points, and put this upper limit in the contract. When system size runs above 1'800 FP, the customer either has to reduce functionality or pay more.

Summary

By applying the backfire method together with chaining we can control the three economic parameters of effort, time and product:

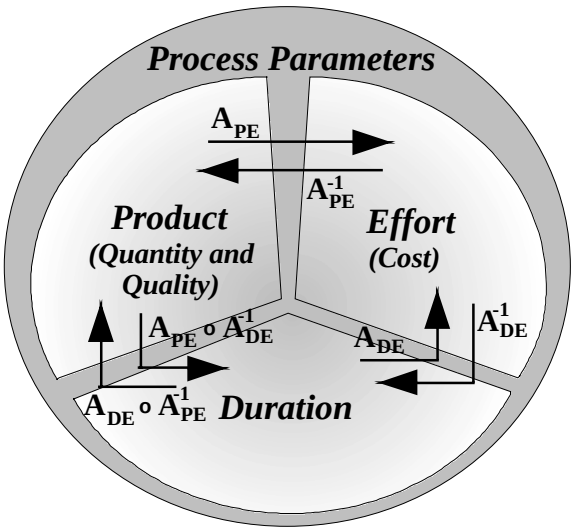

Figure 12: The 3 Parameters of Software Economics and their Relationships

This balance is accomplished with only two empirical databases PE (product ⇨ effort) and DE (duration ⇨ effort) and, therefore, two regressed approximation functions A_{PE} and A_{DE}.

The effects of changes to any parameter can be analysed using these two, plus the combined models in figure 12.

2.1.4 Metric Basics

As we have seen from the presentation of the generic estimation process in 2.1.1, the notion of *metric* or *measure* in the sense of a function that yields a numeric value out of some model is crucial. In this subsection we therefore present some of the basic principles behind software measurement.

A first misunderstanding may mislead the reader who is familiar with metrics in a mathematical sense. Even though such metrics exist in the software domain too (cf. 2.1.6), we generally don't deal with metrics in that sense, but rather with ordinary mapping functions. The synonymous term "measure" is therefore preferred in this text even though not strictly applied.

A software measure's input domain is the model domain, i.e. the *metamodel* of what we want to measure. The metamodel usually is a multi-dimensional structure. The measure's output domain is often called its *scale* and is always single-dimensional. Whereas much effort has been spent on the formal analysis of the scales (cf. 2.1.6 as well as [FE91][ZU91]), which admittedly is of importance, the equally important area of the input domains, i.e. the metamodels, has traditionally been neglected (cf. our new approaches in 3.1). To sum up we may formally define a metric as

Definition of a Measure:
Measure: {metamodel} $\Rightarrow$ {scale}

Measure Categories

The very first distinction we can make in the software area is between *product metrics* and *process metrics*. The first category measures attributes of software, e.g. the total lines of code, or of system descriptions at higher levels, e.g. the number of classes in a class diagram. The latter category measures attributes of the process that led to the software or system description, e.g. the total of elapsed days spent on the process from start to end. The distinction therefore is made between kinds of metamodels: if the metamodel is a model of software, we have a product metric; if it is a metamodel or template of a software process, we have a process metric.

Because there is a one-to-one relationship between product and process, however, we can also attribute a process metric to the product, e.g. the testing effort percentage as a measure of product quality, and vice versa. Another special case are mixed measures like productivity, i.e. the ratio of product size and process effort, which are in general attributed to the process. Some synonyms like *result metric* or *software metric* (in a more narrow sense) are used for product metric and the synonym *project metric* [LO94] is used for process metric. Because this categorisation is easy to use and understand, we used it in our work as well as in the structure of this document, i.e. chapters 2.2 and 3.2 contain information about product metrics, whereas chapters 2.3 and 3.3 are about process metrics.

The next categorisation is derived from the *layering* of software artefacts into 3 main layers, requirements, design and code, plus a fourth maintenance and modification layer. With the process aspect in focus, one often speaks also of the *phases* or the *life-cycle* of software, i.e. requirements analysis, design, implementation and maintenance:

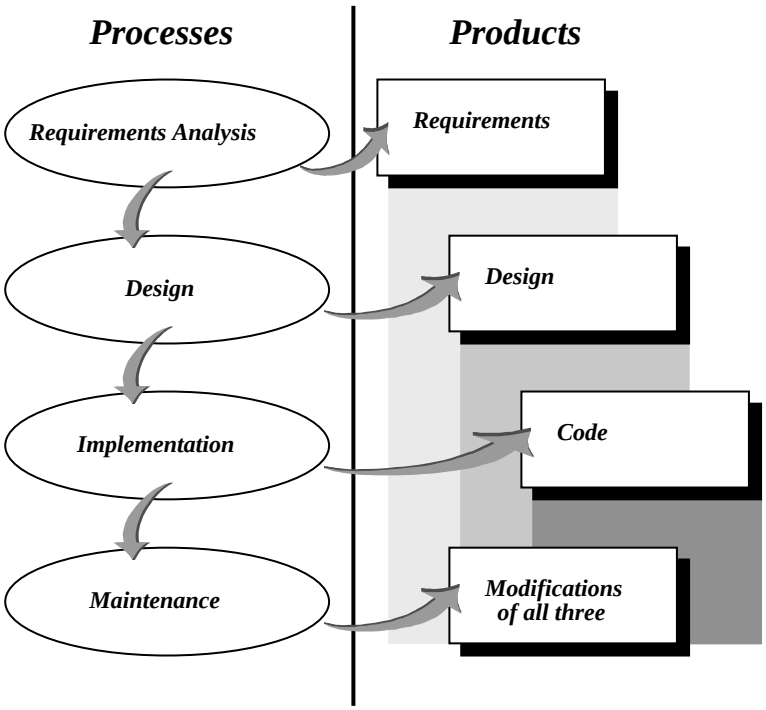

Figure 13: Layered Software Processes and Software Products

All these terms, however, are subject to interpretation. It is not generally decided for example whether a design is only technical design or also user-driven GUI-design, whether implementation also contains testing and documentation, whether project management activities are included or not, etc. As a consequence of this fuzziness we had to elaborate a new reference model, called the *BIO* software process, for this layering and phasing (cf. 3.3). The distinction with respect to layers is essential for the earliness of the availability of a metric. This earliness, in turn, is crucial for estimation. Layering of processes and products, therefore, is important for our purposes.

The next categorisation is also related to this layering. We distinguish between *functionality metrics* and *implementation metrics*. Functionality metrics may be understood in two ways: 1) referring to some size metric (details see below) of a previous layer, e.g. the size of the (functional) requirements, or 2) referring to functionality, i.e. function signatures in contrast to function bodies, within the same layer[9]. Implementation metrics may accordingly be understood in two ways. Synonyms for functionality and implementation metrics of the same layer are *external* and *internal metrics of size*.

Next, we may distinguish between *predictive metrics* and *descriptive metrics*. The first category of measures is used in an estimation process to predict values of metrics that are not yet measurable. The term predictive, however, does not make assumption on the quality of prediction. The total weight of all project team members, to give a rather weird example, may also be considered a predictive metric, e.g. to estimate the effort to implement a system. Metrics of the latter category, in contrast, are used for ex post analysis of results only. Our focus is clearly on predictive metrics.

The last categorisation presented here is the one of *metrics of size* and *metrics of quality*. A metric of size may be used to distinguish between the tiny and the big. These two real world concepts of tiny and big must, however, be taken 'cum grano salis', i.e. be interpreted the right way. A big system description is often a complex one, i.e. one that takes much

[9] This concept of intra-layer functionality primarily makes sense for code, because in layers above, usually all artefacts are external, thus making the distinction of signature and body obsolete.

effort to write and understand. This concept is therefore not only related to sheer size, i.e. number of tokens written, but also to how the tokens are linked and dependent on each other. Quality metrics, on the other hand, may be used to distinguish between the good and the bad. Many metrics are attributed as quality metrics, though, for which no indication of good and bad is given. The metric "depth within inheritance tree" for a class is for example said to be a quality metric, but the question of what's good and bad remains open. Our experience told us to mistrust any quality metric that is not a ratio or some kind of average, because otherwise the interpretation of what is good and bad is not constant but dependent on size. Metrics of size, furthermore, may be used to assess quality: small descriptions are usually preferred over functionally equivalent voluminous descriptions. Synonyms for metrics of size are metrics of weight [DM82], volume [HA77] and - in some cases - complexity [HS96]. The metrics of quality may be further subdivided into *inherent* and *empirical quality* measures. The inherent measures may be measured on the isolated product. In general, however, it is not easy for inherent quality measures to tell what are good and bad values, because there are always unproved hypotheses behind those measures: e.g. "the more comment lines in the code, the more understandable it will be". This may - or may not - be true depending on the comments' readability and consistency with the code. In contrast, *empirical measures* of quality directly measure the facts, e.g. the ratio of the effort of learning a coded system to the size in order to measure "understandability". Those measures also have their drawback, however: they are measurable after the fact only. The use of empirical measures is to quantify the empirical facts ex post. The use of inherent quality measures, on the other hand, should be limited to two areas: 1) setting empirically derived thresholds as alarms, e.g. when the comment ratio in a module sinks below 5%, and 2) estimating the outcome of corresponding empirical quality measures.

A Prototypical Multi-Dimensional Metrics Categorisation Framework

The categorisation presented above has led, in a joint research with Prof. Brian Henderson-Sellers, to a prototype of a metrics framework that serves two main purposes: 1) to assess the diverse existing metrics for object-oriented and conventional systems, and 2) to identify areas of neglect, i.e. poor or missing support by existing metrics. Furthermore, this framework can be a basis for step 3 in the analysis activity of the AMI metrication methodology (cf. 2.1.5 and [AMI92]). The framework is restricted to product metrics only.

The key idea of the framework is to elaborate one matrix per software life-cycle phase, wherein the columns represent the main usage aspects, i.e. quality and size, the rows represent the various metrics and the cells contain information on 1) the presence and direction of the metric's assessment of an aspect, 2) the granularity (e.g. class level, method level) on which the metric may be applied and 3) the binary information of whether the metric predicts or describes / measures the aspect. This last information about prediction or description is an enhancement over the original framework presented in [HS93]. Another enhancement, however only planned, would be to include the average correlation bias from industry surveys into the cell information.

The following diagram is an excerpt from the currently elaborated framework matrix for the implementation layer:

Aspect	***Quality***					***Size***		
Metric (?=desired)	***completeness of abstraction***	***reusability***	***understandability***	***maintainability***	***testability***	***total***	***internal***	***external***
LOC (cf. 2.2.1)			*-all*	*-all*	*-all*	all	all	
η (cf. 2.2.2)			*-all*	*-all*	*-all*	all	all	
ν (cf. 2.2.3)			*-M*	*-MC*	*-MC*		*all*	
FP (cf. 2.2.4)								all
WMC (cf. 2.2.5)			*-C*	*-C*	*-C*	CPS		
DIT (cf. 2.2.5)			*-PS*	*-PS*	*-PS*			
NOC (cf. 2.2.5)				*-P*	*P*			
RFC (cf. 2.2.5)			*-C*	*-C*	-C			
CBO (cf. 2.2.5)		*-C*		*-C*	-C			
LCOM (cf. 2.2.5)			*-C*					
S/C (cf. 2.2.6)			*-all*	*-all*	*-all*	all	all	
Task Pts (cf. 2.2.6)								all
?write coupling		*-all*		*-all*	*-all*			
?read coupling	*all*	*all*	*all*					
?cohesion		*all*	*all*					

Table 1: An Excerpt from the Metrics MDF for the Implementation Phase

We distinguished 5 different sub-aspects of quality: 1) the completeness of abstraction (i.e. no more need to enhance a class), 2) reusability, 3) understandability, 4) maintainability and 5) testability. The signatures of the cell entries are:

no entry	=	metric does not apply
g	=	metric applies for granularity g
-g	=	metric inversely applies (i.e. big values = low aspect)
italics	=	predictive metric
upright	=	descriptive metric (i.e. measurement of the aspect)

The granularities g are indicated as:

S	=	total system
P	=	part / subsystem
C	=	class
V	=	services / public methods
M	=	methods
all	=	all levels of granularity

An analysis of such a matrix has shown that metrics for measuring (not predicting) quality are missing to a large extent, as well as metrics for coupling and cohesion. Our future research interest (cf. 5.3) is therefore directed towards these kinds of metrics.

2.1.5 A Goal Driven Metrication Process (AMI/GQM)

When having to introduce software metrics into the standards and practices of an SPU, we are faced with an enormous variety of metrics to chose from, as well as an enormous variety of possible uses of metrics. In order to really profit from quantitative techniques for estimating or controlling the software process, the choice of the right measures is critical. Metrics based on counting code lines, for example, are proven to be of minor use (some

people even claim they are of harm) in the object-oriented context. It is of great use when firstly the aim of the metric is well elaborated. The use of so-called metrication frameworks like the Goal Question Metric method (GQM, proposed by Basili [BA88]) or its further evolved successor the AMI (Application of Metrics in Industry) method [AMI92] is recommended for this. We briefly present the 4 AMI steps of metrication, 1) the assessment, 2) the analysis, 3) the metrication and 4) the improvement step, and put them into relation to our work. Each of the steps is divided into several substeps as follows:

Assessment:

1. ###Find answers to the following questions: Where do we stand? Are we following standards (i.e. metamodels) so that the use of metrics makes sense? Do we first have to establish standards? The answers may be found by applying the SEI capability model [HUM89] or other methods of SPU-self-assessment like Bootstrap.
2. Define and validate primary goals with the SPU's top-level managers. Samples of primary goals may be: "improve productivity", "improve quality", "lower the error rate after final tests", etc.

Analysis:

1. Break down the primary goals into sub-goals ("monitor productivity of development projects", "monitor productivity of maintenance projects", etc.) together with lower-level managers and project leaders.
2. Identify questions that support the goals ("What is of value to the customer - delivery on schedule? What is of value to our organisation - few errors? What will reduce the use of resources - reuse?", etc.).
3. Identify metrics, for example lines of code produced per person day, %components reused, %rework effort, out of a list of existing metrics that seem to fit the purpose (this step is supported by our multi-dimensional metrics framework, cf. 2.1.4, last subsection). Sometimes the invention of new metrics is needed. This non-trivial task is supported by our metric derivation technique based on metamodels, presented in 3.1.1. Inform all involved people about the chosen metrics and the means to measure them.

Metrication:

1. Write a measurement plan, i.e. identify the projects you want to measure. Define the frequency of measurement and inform the key players in the project about it.
2. Collect and verify the data.

Improvement:

1. Distribute, analyse and review the measured data. Get feedback from all involved people on the values, measurement methods, effort consumed by the measurement process and possible precautions.
2. Validate the metrics, i.e. remove unsound values and outliers. Totally remove metrics that systematically did not receive positive feedback with respect to soundness.
3. ###Relate the data to the goals and implement actions after getting the management's support. Iterate previous steps as needed (i.e. when the metrics support the goals iterate step 3, the metrication, when metric support is insufficient iterate step 2, the analysis, and -finally - when new goals emerge, iterate step 1, the assessment step).

2.1.6 Soundness Criteria for Metrics of Complexity

According to [HS96] measurement should be based on sound theoretical foundations. Formal aspects of some 100 different metrics were carefully analysed by [ZU91]. A less

formal but also very sound approach was chosen by Fenton [FE91] and Ejiogu [EJ91]. A more pragmatic, but - as proven by Zuse - self-contradictory set of so-called "metric axioms" was proposed by Weyuker [WEY88]. We will not repeat all the valuable work accomplished in this area, but rather will summarise the most important aspects for our work. The first topic are scales, i.e. the measures' value domains. Scales may be categorised and restrict the kinds of arithmetic operations that may be applied. The second topic are metrics in their mathematical sense based on comparison operators between system descriptions. Then, we briefly discuss dimensional analysis, i.e. - put simply - the fact that we should not sum up apples and oranges. Finally, we present our own critique of Weyuker's axioms. In each of the 4 topics, the requirements we retained in our analytical (cf. section 2.2) and constructive work (cf. chapter 3) are indicated separately. Note that these requirements are only valid for metrics of complexity, i.e. of the combination of the aspects of size and difficulty.

Scale types

We distinguish between 5 types of scales: 1) nominal, 2) ordinal, 3) interval, 4) ratio and 5) absolute. The types refer to algebraic properties of the scale sets with respect to atomic operations [ZU91]. Since the types may, however, also directly be derived from the measurement procedure, we will not discuss the mathematical apparatus here. The measurement procedures may be described as 1) labelling, i.e. the immediate and arbitrary assignment of values of the scale to objects of the metric's domain, 2) ranking, i.e. the mapping of objects into some ordered scale equivalent to 1st, 2nd, 3rd, etc., 3) interval measurements, i.e. the measurement of distances to some arbitrary reference point, 4) ratio measurements, i.e. the measurement of distances to a fixed reference point called 0 and 5) counting items. The five types and corresponding measurement procedures, as well as allowed arithmetics and examples are given in the table below:

Type	*Measurement*	*Arithmetics*	*Example*
nominal	labelling	simple count	male, female red, blue, green, yellow
ordinal	ranking	simple count, median	agree, neutral, disagree
interval	distance to reference point	simple count, median, mean, variance	temperature Celsius
ratio	distance to 0-point	simple count, median, mean, variance, percentages	temperature Kelvin
absolute	counting	all of the above	lines of code, number of tokens

Table 2: Scale Types

The restriction to allowed arithmetics prohibits typical senseless statements like "the average human gender is half male half female", but also more subtle violations like "today it is 10°C, twice as warm as yesterday when it was 5°C". Since software descriptions are discrete, i.e. distances (in their traditional sense) are not measurable, we adopted the counting paradigm when designing our new metrics. When basing a metric on counting, we assure that statistics, a must for empirically based estimation, may be applied. We therefore retain the first criterion of soundness for software metrics:

(requirement 1) A software metric should be entirely based on counting.

We will complete our discussion about scales with a remark that reduces the importance of this often heavily debated topic. Consider the sides of a die. This very same object may be interpreted as 1) nominal scale, when labelled with colours, 2) ordinal scale, when labelled with 1st, 2nd, etc. and 3) even absolute scale, when labelled with 1, 2, 3, 4, 5, 6. Therefore, we have to remember that the same thing may be interpreted differently. We would even postulate that one may deliberately define the scale interpretation of any observation (even though there are "natural" interpretations and rather "unnatural" ones) as long as it is done in awareness. The same objection can be made to the mathematical analysis of scales. The properties of the scale do not depend only on the scale but also on the choice of the so called atomic operations on the domain objects. Those operations, however, immediately lead to discussions about metamodels of software (cf. 2.1.7).

Mathematical Properties of a Mathematical Integer Metric of Size

Metrics as defined by mathematicians are real valued functions d for the Cartesian product R x R of a set R. R is usually called a room. The function d corresponds to the real world notion of distance. It obeys 3 laws in order to be called a metric:

1) $d(x, x) = 0$

2) $d(x, y) = d(y, x)$ "symmetry"

3) $d(x, y) + d(y, z) \geq d(x, z)$ "triangle's condition"

A room for which a metric exists is usually called an Euclidean room. We extend these laws to integer metrics, because with requirement 1, i.e. software metrics should be counted measures, we imply that the function yields integer values:

4) $d(x, y) \in \mathbb{N}$ "positiveness"

5) $\text{if } d(x, y) = D \Rightarrow \exists z, d(x, z) > D$ "completeness"

The question remains how we extend the unary software metrics to the binary metric form needed here. We have to introduce a difference operator "diff"[10] to software. Usually this operator is line-based, token-based or based on other easily comparable entities. It yields a set of two kinds of items: 1) the deleted items and 2) the added items. We may then apply a unary software metric m to yield the mathematical metric d's value. Using the notation of functional composition ($\circ$) we may write:

$$d = m \circ diff$$

Software metrics, therefore, must be measurable for "scattered", i.e. non-sequential subsets of system descriptions as well as for integral system descriptions. This in turn implies two additional requirements for software:

(requirement 2) a software metric must be defined for single items

(requirement 3) a software metric must be defined for sets of items

In order to allow reasonable difference operators to work, the metric should honour items of finest granularity:

(requirement 4) a software metric should be defined for the most elemental items

A good example fulfilling these requirements is the lines of code metric. It is easily measurable for a single item (yielding value 1) and for sets. Its granularity, however, is only reasonably fine, because on one line of code several more elemental description items may be placed. The set of all possible programs and the LOC metric, nevertheless, form an

[10] Such as the UNIX diff utility.

Euclidean room. From this example we would formally conclude that the LOC metric is sufficient. But this formal validation qualifies the metric of size in the context of a metamodel of system descriptions only. It does not say anything about the meaningfulness of the metamodel. Thus, when viewing system descriptions as sets of LOCs that are modified and mentally understood line by line, number of LOCs is the right metric. However, we state that this is a very simplistic view ignoring almost all of the structural characteristics of system descriptions. Furthermore, this is an indication that formal software metrics requirements are not enough to assess metrics. We therefore developed semantically founded requirements based on the metamodelling paradigm (cf. 2.1.7 and 3.1).

Dimensional analysis

Having attended many lessons of high school physics we all know a simple way, dimensional analysis[11], to validate newly derived formulae: when the result should be a velocity and the formula yields kilograms per square meters it is definitely wrong. Similar analysis can be made for definition formulae of software metrics [HS96]. If we encounter a new metric that is defined like:

new_metric = #classes + #methods + #arguments passed

then it can be safely ignored because it violates dimensional analysis. We therefore state another requirement:

(requirement 5) software metrics should be dimensionally safe

Furthermore, a software metric of complexity should also be useful for the derivation of secondary metrics like "percent reused items". To achieve this, a metric should yield dimensionally equal values for all kinds of software items. This is no problem for the lines of code metric, which only knows one kind of software item. It may be a problem, however, for more sophisticated metrics defined for single statements, variables, functional abstractions and data abstractions (classes). An analogy to the 3-dimensional room is the cubic meter. It is uniformly defined for all kinds of objects[12].

(requirement 6) a software metric should be uniformly defined for all kinds of items

Weyuker's 9 "axiomatic" properties

To conclude our list of formal requirements, we summarise and comment the most widely used (but not widely accepted) criteria suite, Weyuker's axiomatic properties of software complexity metrics [WEY88]:

1 A metric should not rate every system description equal.

Comment: This is an absolute requirement for any good metric and we would like to retain it as a basic requirement. The requirement, however, is implied by the forthcoming requirement 7 so we do not mention it as an extra requirement.

2 The metric rates only a finite number of system descriptions equal.

Comment: We reject this criterion because there are good measures that violate it. Let us assume a C++ program containing the single statement `cout << 1;`. There are infinite variations of this simple (and rather useless) program, i.e. by replacing the constant `1` with any other integral number (assuming no limits on numbers for

[11] Dimensional analysis in measurement theory is analogous to type checking in software engineering.

[12] In contrast to square-meters and meters for which it is not clear for all kinds of objects what exactly to measure: 1) the length, 2) the height, 3) the depth.

simplicity's sake). The LOC metric, however, will yield 1 for any of the infinite variations of the program.

3 The metric is not different for every different system description.

Comment: This criterion is valid in our eyes. It is, however, so fundamental and easy to fulfil that we do not view it as an extra requirement.

4 If two system descriptions of the same layer (e.g. the functional specification) are equal, the metric values for the same system in another layer need not be equal.

Comment: This is a good property, however obvious in the context of layering (cf. 2.1.4) and hence we do not retain this criterion as a separate requirement.

5 If a system description Y is concatenated to a given base system description X then the rating for X alone must be smaller than or equal to that of X+Y.

Comment: This must not always be true for every kind of system description and concatenation (e.g. adding the interface description of a class to a piece of code that heavily uses this class reduces its (psychological) complexity), so it is an undesirable property in general.

6 Concatenating the same system description to two different pieces of identical rating need not result in the same rating in both cases.

Comment: This is surely a valid property (BTW: and contradicts property 5 as shown by Zuse [ZU91]). It is implied, however, by requirement 7, below.

7 Changing the mere sequence within a system description may change the rating.

Comment: The sequence is an important structural characteristic of system descriptions, so resequenced system descriptions are not to be considered equal. We retain this property as

(requirement 7) A software metric may yield different values for differently sequenced system descriptions.

8 Changing names must not change the rating.

Comment: Here we heavily disagree, because e.g. a string class named "Date" is much less understandable than one named "String". Names are important parts of system descriptions. In total contrast to Weyuker, we would like to formulate our requirement as

(requirement 8) A software metric should yield different values for differently named system description items.

9 Given two system descriptions X and Y, the rating of X+Y may be bigger than the sum of the individual ratings.

Comment: I can't think of any reason for this property and none of the knowledgeable metrics do satisfy it. Zuse "proves" that this property is needed for a metric to be a ratio scale, but there must be a big "misunderstanding/confusion" because e.g. LOC is even absolute scale and does not meet this property. In accordance to Ejiogu [EI91] and to the mathematical "triangle's condition" for metrics, we propose, in contrast, our last (semi-)formal requirement

(requirement 9) Given two system descriptions X and Y, the rating of X+Y must not be bigger than the sum of the individual ratings

A summary of the retained requirements is given in the following table:

#	***Requirement***
1	counting

2	single items
3	sets of items
4	elemental items
5	dimensionally safe
6	dimensionally uniform
7	sequencing
8	naming
9	triangle's condition

Table 3: Requirements for Software Metrics of Complexity

2.1.7 Metamodelling

Metamodelling is a rather modern approach to software engineering. Its roots go back to philosophical work on ontology by Bunge [BU77] and on general systems theory by Mesarovic and Takahara [ME75]. Metamodels formally show the structure of models, e.g. software or other higher level system descriptions. They play the same role as entity relationship or object models for application domains, but for the domain of software development itself. The practical purposes of metamodels thus are analogous to those of domain object models: they are a sound starting point for the analysis of the domain's processes, i.e. software processes, and for implementing automated tools that support those processes. Metamodels were used in the following fields recently:

1 Understanding software engineering techniques (e.g. system decomposition by Paulson and Wand [PA92], grammar evaluation by Weber and Zhang [WEB92]).

2 Building automated systems and software engineering tools (e.g. metamodels in CASE environments by Smolander [SM91], CDIF Interim Standard EIA/IS-83, Part 1 [CDIF91], Texel-SF [TE94], COMMA [OPEN96]).

3 Defining software measures which in turn may be a foundation for control mechanisms and quality systems of development processes (e.g. a metrics suite for OOD by Chidamber and Kemerer [CHI94]).

4 Evaluating and/or defining development methodologies (e.g. the frameworks by Österle/Gutzwiller [ÖS92] and OMG [OMG92], the methods Texel [TE94], BIO (cf. 3.3.3), UML [RA96], OPEN [OPEN96]).

Existing Metamodels Overview

Takagaki and Wand [TA91] have introduced an object based metamodel for information systems derived from an ontological point of view. To denote their metamodel they used an axiomatic mathematical system. Their metamodel was used to analyse the semantics of modelling concepts like the relationship (Wand, Storey, Weber [WA93]) like decomposition (Paulson, Wand [PA92] or like grammars (Wand Weber [WA92], Weber Zhang [WEB92]). The so called "Wand Weber Metamodel" (WWM) reflects the structure of a system from a rather theoretical point of view. Maybe this theoretical touch is partly caused by the notation used to describe the WWM. It is not a metamodel of a system description but rather of a system at work. We adopted some notions of it for our metamodel of system descriptions (cf. 3.2.1).

2 Other metamodels were described by Smolander [SM91] to denote system modelling techniques. He also presented (as a logical step) a meta-metamodel that shows the structure of metamodels (OPRR, Object Property Relationship Role Metamodel adopted from Welke [WEL89]). The essence behind OPRR is equivalent to Chen's modelling technique (ER, Entity Relationship Model first presented in [CH76]) enhanced by the explicit concept of a role (which is implicitly also present in Chen's original technique). Smolander

then described metamodels of some SE methodologies (like DeMarco's Structured Analysis [DM79] or Booch's Class Diagram Technique [BO91]) and built a prototype tool that interprets metamodels and lets systems engineers graphically edit the objects imposed by any methodology (as long as the metamodel is formulated in OPRR). His metamodels were used as one basis for our metamodelling of the domain analysis layer (cf. 3.2.2).

3 Yet other metamodels are defined in a standard of the EIA called CDIF (CASE Data Interchange Format [CDIF91]). The EIA also describes a meta-metamodel (EIA/IS-81, July 1991). Like Smolander's OPRR, it is also very closely related to Chen's ER modelling approach (except for an abbreviated notation for relationships). The interim standard EIA/IS-83, July 1991, then defines the semantic metamodels of the ERA (Entity Relationship Attribute, cf. [CH76]) and DFD (Data Flow Diagramming cf. [DM79]) techniques. Those metamodels are used to support the data transfer between different CASE tools without losing semantic information. CDIF is enhancable to denote system metamodels of other techniques and lifecycles. The CDIF meta-metamodel was adopted to denote the metamodels in our work. Furthermore, some parts of the ERA and DFD metamodels were used as parts of the domain analysis layer metamodel (cf. 3.2.2).

4 Based on the CDIF meta-metamodel, Carmichael recently presented COOMM, a metamodel for object oriented systems in [CA94]. It is a metamodel used to implement software development workbenches and is mainly a formalisation of the OMG Object Analysis and Design Reference Model.

5 Predecessors of the metamodels presented in this document were introduced by Moser et al. as they defined the SE methodology BIO (BI-CASE/*OBJECT)* in [MO93]. The metamodelling within this methodology was mainly done by the author. It was then used to define strategies to elaborate the object types identified and to define corresponding documentation and naming standards.

6 More recently - at the time of completing this work - OMG has made an initiative for standardisation of metamodels for object-oriented systems. The two major competitors for this forthcoming standard are Rational's UML (Unified Modelling Language) [RA96] and the metamodels under development by the OPEN team [OPEN96]. Both metamodel proposals are still subject to discussion. Because of the relatively recent appearance of these metamodels, we did not incorporate or adopt them for our work. A transition or mapping from our metamodels to the forthcoming standard seems, however, possible. Our metamodel is designed for metrics definition. It is therefore semantically non-restrictive. For example, our metamodel allows any object to have zero, one or even multiple types and change its types during its lifetime. This property of non-restrictiveness gives us confidence that - after a more thorough and detailed comparison of the metamodels - a usable match can be made, and the metrics, namely the System Meter, can be applied to models conforming to the new metamodels as well.

Denoting Metamodels

We use metamodels primarily to define or assess measures (cf. 3.1 for details). Metamodels of measures should reflect the minimal structure of the things being measured. In order to explain the notation we use in our work, we give an example: the metric "percentage of people in a given country living in big cities" presumes the following (meta)model[13] of populations:

[13] This actually is not a metamodel because its modelled entites are real-world concepts. However, the discussion of metamodel notation can well be conducted using this sample.

I)	There are people.
II)	There are locations people live in (each person in exactly one location).
III)	Those locations belong to exactly one country.
IV)	The locations may be categorised as "is big city" and "is not big city".

Figure 14: A Sample (Meta)model Denoted in Prose

More formally denoted (similar to the notation used by [WA90]) we would state:

$Population \equiv \langle P, L, C, bc, co, li \rangle$, where

$P = \{p1, p2, p3, \ldots\}$ a set of people,

$L = \{l1, l2, l3, \ldots\}$ a set of locations or places,

$C = \{c1, c2, c3, \ldots\}$ a set of countries,

$bc: L \rightarrow \{isBigCity, IsNotBigCity\}$ a function,

$co: C \rightarrow \wp(L)$ a function "country" with its inverse being definite

$li: L \rightarrow \wp(P)$ a function "lives in" with its inverse being definite.

Figure 15: A Sample (Meta)model Denoted Formally

It is not practical however (especially in situations more delicate than the one above) to represent the metamodel as a list of natural language statements or as a list of formal definitions. The author adopted a special form of graphical notation (ER-Model, cf. [CH76] also used to define the CDIF-Metamodel [CDIF91]) to denote metamodels. The small example looks like this:

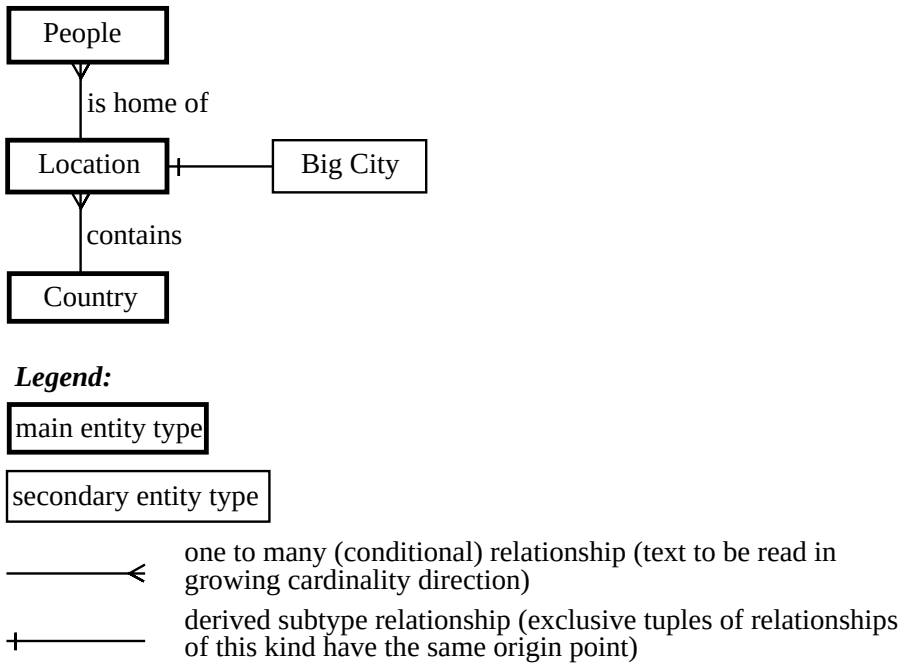

Figure 16: A Sample (Meta)model Denoted Semi-graphically

For our purposes, these representations of metamodels are equivalent in their expressiveness. We therefore use the most intuitive representation to us which seems to be the graphical notation as exemplified and explained in figure 16.

Metamodels are used to derive and assess measures in our work. For details on how metamodels serve these two purposes refer to section 3.1.

2.2 Existing Measures and Metamodels for Software

In section 2.1.6 we elaborated 9 formal requirements for software measures of complexity derived from measurement theory and mathematical properties. Additionally and more importantly, we gained the insight that we cannot assess measures with such formal means alone. We also need semantic assessments, be they formal or informal. In order to assess measures and, as a consequence, estimation techniques based on their semantics we developed a metamodel assessment technique which is described in more detail in 3.1.2. In order to understand the following assessments, which make use of our new technique, we give a brief summary here:

The first step of the metric metamodel assessment technique consists of elaborating or obtaining a reference model of the thing to be measured. In the context of software, i.e. models of systems, this reference model is a so called metamodel. It is derived independently of any metric. The second step consists of elaborating the minimal metamodel of the metric, i.e. those - and only those - constructs that are necessary for the complete metric definition. The minimal metric metamodel is then checked for 1) completeness with respect to the reference metamodel, 2) the time of availability within the software process, 3) rigidness, i.e. language and modelling independence and 4) "non-fakeability".

The reference model against which we assessed the measure's metamodels is the System Metamodel described in 3.2.1 and already introduced in 1.6.

Each metric is presented in the following scheme: 1) the metric's metamodel, 2) the metric's definition, 3) common usages, 4) metamodel assessment, 5) formal metric assessment, 6) practical assessment and 7) summary. Positive evaluation points in the assessment parts are marked with white numbers (e.g. ③), negative points are marked as black numbers (e.g. ❼), ambiguous assessments are marked with both (e.g. ❷/② for rather negative, and ②/❷ for rather positive overall assessment).

2.2.1 Lines of Code

Metamodel

Figure 17: The LOC Metamodel

The metamodel of the Lines of Code metric views an implemented system as a set of isolated *code lines*, i.e. description elements separated with carriage returns. Usually, but not always, blank and comment lines are not (!) considered to be code lines. However, headers, declarations, executable and non-executable statements as well as multiple statements and statement fragments are considered one line of code if placed on one line (cf. [CO86]).

Definition

Because the LOC metamodel is structurally very easy (i.e. consists of a single unrelated entity) we do not have many choices for the formal metric definition:

$$LOC = |\text{Line of Code}|$$

LOC is simply the cardinality of the set of lines of code.

Usages

After coding is complete one may observe - ex post - that the resulting LOC correlate with the effort spent. The ratio of LOC and effort measures (e.g. person days) is a common productivity measure of software processes. Typical values from traditional COBOL MIS-SPUs from the mid-eighties are reported as 5-10 LOC/PD [IBM84]. Before coding, LOC may be estimated (often rather obscurely) and this estimate may then be used as a predictor for effort [BOE81]. During maintenance, the code parts in question may be measured with LOC and used as a predictor for maintenance effort. The LOC metric may also be used to derive all kinds of secondary metrics like "number of comment lines"/LOC to measure the "commentedness" of a program or "number of LOC for a class"/LOC to measure the relative sizes of the classes in an object-oriented program. LOC is very widely used. This use of LOC, however, typically requires an enhancement of the metamodel with a categorisation of the Lines of Code, for example into those which contribute to a class implementation and those which do not. One may consider our metamodels as a set of such enhancements that led to a completely new and richer metamodel. The original LOC metamodel assessed here is very limited.

Metamodel Assessment

❶/① Metamodel completeness: The metamodel is incomplete. Only the aspect of messages (cf. 3.2.1) is captured. The messages, however, indirectly capture many other aspects because every other kind of description object must somewhere occur in a message (at least in order to be created). Therefore we may observe an acceptable behaviour of LOC with respect to effort correlation. When the metamodel is enhanced (often implicitly) with categorisations, we may achieve more coverage of the reference metamodel. However, aspects that can never be captured are 1) structural relationships between description objects, 2) the distinction of the external and internal view of objects, as well as 3) the distinction of components reused from a programming environment (language and library/framework components). LOC does additionally not capture the functionality of a system. This is not due to structural deficiencies of the LOC paradigm, but because only one software layer, the implementation layer, is addressed. It prevents the LOC measure from being a good measure of productivity, because mere code production not production of usable functions is measured.

❷/② Time of availability: The lines of code of the final tested product are established after the implementation and test phase only. Initial untested LOC may be obtained before the test cycles. Most development effort, however, is spent before LOC become available as a measurable entity. With respect to estimation of early development effort, LOC is thus too late. Estimation of maintenance and testing efforts, however, are supported.

❸ Rigidness: This is one of the weaker aspects of the LOC metric. First and most important: LOC values of one language can not directly be compared to those of another. For heterogeneous (concerning the languages used to implement) systems it is therefore impossible to give a single meaningful value of size when using LOC and because a single line of code is generally not equal to exactly one lexical object of the language, categorisations (e.g. into comment and non-comment lines) are often left to interpretation. It is also left to interpretation whether comments, block marks, labels, declarations, etc. are to be considered as a line of code. Furthermore, code lines with more than one lexical statement in it, may or may not be considered as more than one line of code (thus being another source for interpretational variance). As an annoying consequence of the drawbacks just stated there exists an overwhelming number of LOC dialects and counting guidelines (e.g. SLOCs and ELOCs [CO86] [JO85] [GR87] [BOE81] [PU80] [ZU91]). For

old-fashioned languages like some Assemblers, (old-style) BASIC and FORTRAN which were line-oriented (i.e. one statement equals one line), the LOC paradigm is well-suited.

❹ "Non-Fakeability": This is another weak aspect of the LOC metric. Since the LOC metamodel is simplistic and non-rigid, clever programmers may generate extra lines of code to score high LOC/PD productivity rates. On the other hand, when low LOC per functionality values are aimed, programmers may produce very nasty pieces of code, where several statements are packed onto one line.

Formal Assessment

① Counting: The LOC measure is a simple count and therefore its scale is absolute and every arithmetic and statistical operation may be applied safely.

② Single items: The LOC measure may be applied to a single line (however yielding a constant value of 1 because it ignores the line's semantical and inner complexity).

③ Sets of items: LOC is easily measurable for sets of lines of code.

④/❹ Elemental items: The line of code is fairly elemental. However, for most modern programming environments (C, C++, Smalltalk, etc.) one line of code is not equal to an elemental language construct. When measuring parts of systems or selected subsets we therefore always face language-specific problems of interpretation.

⑤ Dimensionally safe: The LOC metric is dimensionally safe because only one kind of entity is counted.

⑥ Dimensionally uniform: LOC is uniformly defined for all kinds of lines of code.

❼ Sequencing: LOC ignores the sequence within system descriptions because it views the lines of codes in isolation.

❽ Naming: LOC also ignores the naming of system description objects.

⑨ Triangle condition: If we use a line difference algorithm to yield the two sets of subtracted and added lines of code for any two pieces of code x and y and sum up the LOC values for the two sets to obtain d (x, y), then the triangle condition of $d(x, z) \leq d(x, y) + d(y, z)$ is always fulfilled, because we can always - in case the direct difference operation between x and z yields bigger sets - consecutively apply the difference sets of x to y and then y to z and add their cardinalities to achieve equality.

Practical Assessment

We may positively mention that in practice LOC is more frequently used than any other metric, because LOC is measurable very easily. Therefore, much experience and knowledge with LOC measurement and usage is available in industry both among managers and engineers. The idea of a line of code is also intuitively - not formally with respect to modern languages - easy to understand. Furthermore, estimates for testing and maintenance may be soundly based on LOC measurements.

Because the LOC metamodel is heavily dependent on the programming language and on interpretation, however, one can unfortunately not adopt coefficients and statistics from industry surveys without care. When used for early development effort estimates, it cannot be obtained as a measured value, but rather must be (intuitively or rationally) estimated itself. Thus, it does not substantially contribute to early estimates.

Summary

The LOC measure has its merits as an easy and intuitive metric. Its formal qualities are good. Therefore it can be safely used as a base measure for a variety of derived measures. The LOC has its weaknesses in the areas of 1) time of availability, 2) rigidness of definition and 3) over-simplicity in the underlying metamodel. The first weakness is severe

with respect to early effort estimation, the second to technical independence and comparability of measured values.

2.2.2 McCabe's Cyclomatic Complexity

Metamodel

The underlying metamodel of Tom McCabe's so called cyclomatic complexity metric ν [MC76] is based on the control flow of a program. Software is viewed as a directed graph wherein two kinds of nodes exist: 1) normal sequential processing nodes with several input and one output edge and 2) decision/branch nodes with several input and two alternative output edges. A start and end node with no ingoing and no outgoing edge respectively are also required. For structured programs - which are assumed in our further discussions - the number of input edges per node is restricted to exactly one. A truly minimal metric metamodel for structured programs (cf. definition below) would therefore even consist of the decision/branch nodes only and ignore the others:

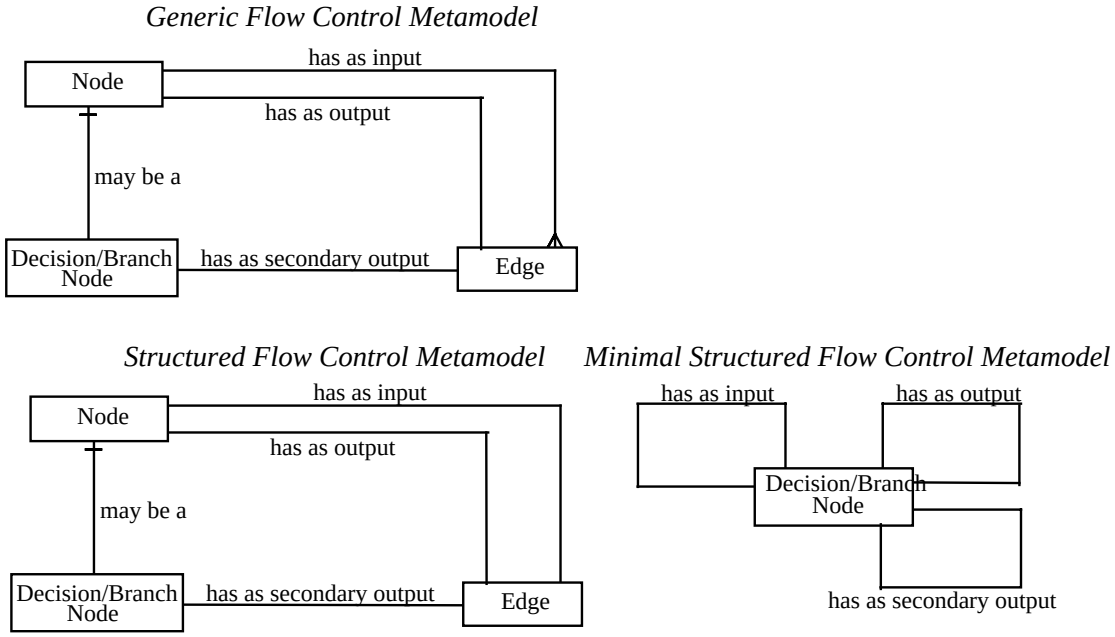

Figure 18: The Control Flow Metamodels

A variety of some 100 more control flow or program logic metrics are proposed (cf. [ZU91]) based on the same metamodel. Their application, though, is either not widespread or they differ only minimally from McCabe's ν. Therefore we discuss this metric only.

Definition

The original definition of the cyclomatic complexity for a directed graph is:

$$\nu = |Edge| - |Node| + 2$$

This essentially is the count of all possible static paths through the graph. For structured programs however, where the rule "one-node-equals-one-input-edge" holds, the formula may be reduced to:

$$\nu = |Decision/Branch\ Node| + 1$$

Usages

The cyclomatic complexity is mainly used as a descriptive measure of quality, for which thresholds may be set, e.g. for "maximum allowed ν per function/method". Thresholds are set from 5 to 10 depending on the programming language used, for object-oriented

languages thresholds even below 5 may be reasonable [LO94]. Furthermore, it is successfully used to predict the maintenance effort for selected functions or methods [OM92] [STA94].

Metamodel Assessment

❶ Metamodel completeness: The metric ν ignores virtually all entities of the reference metamodel of software systems. Just calls to decision/branch methods are counted. Even though the decisions and branches are necessary constructs in software, they are of subordinate importance in modern, object-oriented systems.

❷/② Time of availability: The cyclomatic complexity is available after first implementation only. It is therefore useful for testing and maintenance effort prediction only which can be accomplished successfully.

③/❸ Rigidness: Even though variants exist, the original definition of McCabe and the simplified version for structured programs do not leave any room for interpretation for single methods or functions. When summing up values for sets of methods or functions, variations of counting rules exist, though.

❹/④ "Non-Fakeability": Dummy conditional statements may easily be introduced into code, even though not as easily and not as camouflaged as plain lines of code. The danger of faking, however, is less prominent for ν because it is very rarely used as a basis for productivity measures.

Formal Assessment

①/❶ Counting: The metric ν is essentially a simple count for structured programs. We would prefer the elimination of the constant summand of the formula, though, because when summing up ν for several parts of system descriptions, problems may occur [HS92] [ZU91].

② Single items: It is defined for method or function bodies.

❸/③ Sets of items: For disconnected sets of items, i.e. methods or functions, different propositions of counting or non-counting the constant summand are proposed. When adhering strictly to McCabe's definition, no practical problems will occur.

❹ Elemental items: Because ν is defined for whole methods only, we sometimes have problems in measuring partly rewritten or enhanced methods. It is left to interpretation which items belong to the part under consideration and which do not. For being a generally usable metric of complexity, McCabe's cyclomatic number is too coarse.

⑤ Dimensionally safe: The dimensional analysis of ν reveals no problems.

⑥/❻ Dimensionally uniform: Because ν is basically defined for a single kind of software artefact only, we encounter no problems. For higher order system descriptions we must be careful about the summation rules applied. Comparing differently summed up numbers may yield wrong interpretations.

❼ Sequencing: The metric is not sensitive to rearranging. This was already critiqued by several authors who would like ν to yield higher values for nested branches than for sequenced branches.

❽ Naming: Naming aspects are not dealt with by ν.

⑨ Triangle condition: Assuming a hypothetical method-based difference operator on software the triangle condition is fulfilled for reasons analogous to those for the LOC metric.

Practical Assessment

McCabe's ν is easily measurable with simple lexical code analysers. The usability of ν in modern, i.e. object-oriented environments is decreasing, because other structural elements (classes, inheritance, coupling, cohesion) become more important. Logically complex pieces of software are furthermore reused more and more as opposed to being rebuilt.

Summary

McCabe's cyclomatic complexity number ν is a sound complexity measure for pieces of software that obey the flow chart paradigm. Its main drawbacks however are 1) late measurement (after implementation), 2) coarse granularity and some difficulties when summing up unconnected pieces of software and 3) the ignoring of many important aspects like inheritance, coupling, cohesion, etc.

2.2.3 Halstead's Software Science Metrics

Metamodel

Software Science was "invented" at the end of the seventies by late Maurice Halstead [HA77]. It was intended to give a sound formal basis to software engineering. Especially the inconveniences of the LOC metric and metamodel were attacked. A token-based view on software was adopted. Software is a set of tokens which can by categorised into operators and operands.

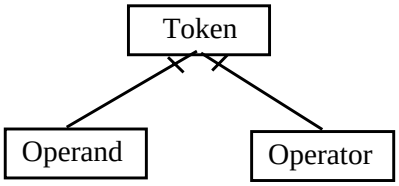

Figure 19: The Software Science Metamodel

This metamodel achieved some language independence, even though the semantics of operands and operators differ grossly between languages. Halstead also proposed extensions of this metamodel to higher layers of software, especially requirements analysis documents. Those ideas implicitly made use of the metamodel mapping technique as described in 3.1.3. No applications of the metric to higher software layers were however reported to our knowledge.

Definition

Halstead defined several metrics based on his metamodel. The first metric he called the length N of a piece of software defined as

$$N = |\text{operators}| + |\text{operands}| = |\text{tokens}|$$

The length thus is simply the number of tokens. The second metric is the vocabulary metric η which is defined as

$$\eta = |\text{unique(operators)}| + |\text{unique(operands)}|$$

Whereas the operators and operands correspond to function calls and actual parameters, the unique operators correspond to functions and the unique operands to variables. The metrics of length and vocabulary are then used in a more aggregate metric of volume V as follows:

$$V = N \cdot \log_2(\eta)$$

The rationale behind this metric is to calculate the average binary information contents of a token and then multiplying it with the number of tokens. V thus corresponds to the number

of binary decisions necessary to write some piece of code. Some more metrics were defined (including an effort metric) which are not discussed here.

Usages

The simple token count N as well as the volume V correlate best to maintainability and maintenance effort of the most widely known metrics according to a survey conducted by Oman [OM92]. The volume metric V is also used as a descriptive measure of size, for which thresholds may be set, e.g. "maximum V per method".

Metamodel Assessment

❶/① Metamodel completeness: Whereas Halstead's metamodel is more refined than the LOC metamodel and even implies notions of functional abstractions (unique operators) and object instances, i.e. variables or constants (unique operands), it is still not suited to model the structure of modern software concepts like data abstractions and inheritance. Furthermore, it does not capture the dependency links between operators and operands as well as functions and variables. Concepts of coupling and cohesion within system descriptions may thus not be measured.

❷/② Time of availability: In practice the Software Science metrics are restricted to implementation and maintenance phases. Halstead proposed an extension to analysis and design descriptions but these extensions were never put into practice.

❸/③ Rigidness: Whereas the notions of operands and operators are rather strict in theory, the different programming languages still leave room for interpretation. It is e.g. not clear how to proceed with declarative statements as well as with parentheses and other separators. The token count is very sensitive to those interpretations and therefore measurements with different guidelines must be treated with great care.

❹ "Non-Fakeability": As well as the lines of code, superfluous tokens may deliberately be introduced into code in order to pretend high productivity rates.

Formal Assessment

The formal assessment is accomplished in two parts: first for the token count N, then for the volume V:

① Counting: N is a simple count and therefore of absolute scale.

② Single items: N may be counted for a single item.

③ Sets of items: N may also be counted for sets of items without difficulties.

④ Elemental items: The most elemental items for the token count are lexical tokens in code. This a very fine granularity level and therefore supports the measurement of even the slightest differences between programs. Because however there exist groups of tokens that make up a single operand (e.g. <function name> (, ...) in C-style function calls) the token basis may be too fine and token counts may differ where no essential difference exists.

⑤ Dimensionally safe: Since only one kind of description item, the token, is counted, N is dimensionally safe.

⑥ Dimensionally uniform: Because only one kind of description item is honoured, we also have dimensional uniformity.

❼ Sequencing: Resequencing of tokens is not relevant to N.

❽ Naming: Renaming of tokens is not relevant to N.

⑨ Triangle condition: The triangle condition is fulfilled by N when using a token based difference operation between pieces of source code.

Now we assess V:

❶ Counting: V is not a simple count and its scale is therefore subject to discussion.

② Single items: V may be measured for a single item but always yields 0 because the logarithm of 1 is 0.

③/❸ Sets of items: V may also be measured for sets of items. Because of V's dependency on the whole set of tokens, i.e. the unique operation yields different results for different sets, there is no formal relation between the value of V for the total set and for subsets. This obstructs the use of V for partial measurements and percentage comparisons.

④ Elemental items: The same comments apply as for N.

❺ Dimensionally safe: The dimension of V are bits of information. However, the vocabulary used to determine the bit-basis varies from piece to piece of code. Values of V may therefore not be arithmetically used without the greatest care.

⑥ Dimensionally uniform: Since only one kind of description item is honoured, we have dimensional uniformity. But because of the dimensional unsafety, we cannot make use of this uniformity.

❼ Sequencing: Resequencing of tokens is not relevant to V.

⑧ Naming: Renaming of tokens is captured by V, because the unique operation behaves differently on renamed token sets.

❾ Triangle condition: The triangle condition is not fulfilled by V, because the merged vocabulary of merged pieces of code may increase the bit-basis leading to a greater value than the sum of the values for the single pieces.

Practical Assessment

In practice, the length N and volume V are quite successfully used for effort prediction of testing and maintaining software [OM92] in spite of V's formal incorrectness. The volume V definable as the number of binary decisions necessary to write a piece of code, does not reflect the mental processes used by human beings which is based on chunking and tracing (cf. 5.3). It therefore shows substantial weaknesses especially when applied to parts of code and the overall code. Writing a single token is even considered being of zero complexity.

Summary

Halstead founded (without general acceptance though) a new branch of science he called "Software Science". His texts are full of splendid ideas. Many other researchers have however found errors in form and contents, so it's not wise to use his work without care. We also found serious flaws in one of his measures, the volume measure V. But the ideas of operators and operands as well as unique operators and operands were used as a basis for our metamodel entities of messages (= operators), actual parameters (= operands), description objects (= unique operands) as well as methods and formal parameters (= unique operators extended to cope with multi-token forms).

2.2.4 Albrecht's Function Points

Allan J. Albrecht from IBM has presented in 1979 [AL79] a first version of Function Points, a new kind of software metric. It basically differs from all other metrics because it does not address the implementation layer but the user requirements or analysis layer. Albrecht revised his definition in 1984. There is also an International Function Point User Group [IFPUG] that defined several releases of counting guidelines. More fundamental changes were made by Symons [SY93] in his Mk II Function Point Analysis (FPA) proposition. The most valuable extension is currently Jones' Feature Points [JO85] who assigns complexity ratings to each user function instead of only assessing a system wide complexity characteristic. We will present and assess the 1984 definition of Albrecht

because of its most widespread use. The assessment statements are annotated for substantial improvements in derivatives of the FP.

Metamodel

The metamodel of the Function Point metric views a system's analysis as a triplet of 1) a data, 2) a functional and 3) an informal part:

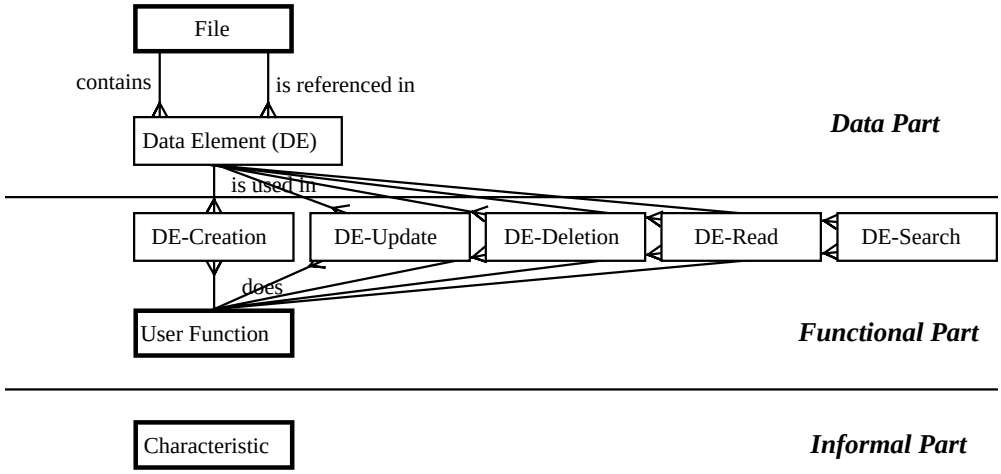

Figure 20: The Function Point Metamodel

The *file* (one might call it "persistent data store" using modern terminology) is the essential concept of the data part. A file consists of several data elements (one might also say attributes) which the system recognises as input from or output to any system user. Some of those user data elements are read only. If all the data elements of a file are read only the whole file is classified as "read only". Files are also classified as "easy", "medium" or "complex" according to the number of plain data elements and the number of referential data elements (also: foreign keys) they contain. This data metamodel represents a user view of the information a system may store and/or (re)produce.

The *user function* is the central concept of the functional part. Within such a functional element - which may correspond to a screen or report in an information system - some subset of the data part is accessed in 5 different modes (CRUD and/or as a search criterion). The files in such local access subsets are locally classified in easy, medium and complex according to the same rules as above, but the data element counts are (obviously) restricted to the local access subset.

Finally we have an isolated set of so-called system *characteristics* which make up the informal system part.

Definition

The "top" formulae that define the "unadjusted system size" (usize) and the "adjusted system size" (size) in FPs are:

$$\text{usize} = \sum_{f \in \textbf{file}} P_{\text{emc}(f),\,\text{ro}(f)} + \sum_{uf \in \text{user function}} \left(\sum_{\text{cud}(uf)} P_{\text{cud},\,\text{emc}(lf)} + \sum_{lf \in \text{r}(uf)} P_{\text{r},\,\text{emc}(lf)} + \sum_{lf \in \text{s}(uf)} P_{\text{s},\,\text{emc}(lf)} \right)$$

$$\text{size} = \textit{usize} \cdot \left(P_{\text{base}} + \sum_{c \in \textbf{characteristic}} \text{influence}(c) \right)$$

Globally viewed those formulas simply sum up the three main entity types (files, screens/reports and characteristics) we saw in the metamodel. The sum is made up using parameterised weights (denoted $P_{\text{index1, [index2]}}$). The weights in turn are defined based upon

the file classification function (denoted emc(), returning one of the values "easy", "medium" or "complex"), upon the file type function (denoted ro(), returning one of the values "normal" or "read only") and upon the access types (cud = create, update or delete (or "input" as Albrecht stated), r = read (Albrecht's "output") , s = search (Albrecht's "inquiry")). The following table represents the most widely used parameter set:

	easy	***medium***	***complex***
input (create, update, delete)	$P_{easy,\ cud} = 3$	$P_{medium,\ cud} = 5$	$P_{complex,\ cud} = 7$
output (read)	$P_{easy,\ r} = 4$	$P_{medium,\ r} = 5$	$P_{complex,\ r} = 6$
internal files (normal)	$P_{easy,\ normal} = 5$	$P_{medium,\ normal} = 10$	$P_{complex,\ normal} = 15$
external files (read only)	$P_{easy,\ r/o} = 5$	$P_{medium,\ r/o} = 7$	$P_{complex,\ r/o} = 10$
inquiry (selection)	$P_{easy,\ s} = 3$	$P_{medium,\ s} = 4$	$P_{complex,\ s} = 5$

Table 4: Historical Parameters for the Function Point Metric

The characteristics' weights are defined by two parameters: P_{base} is usually set to a value near 0.7 and the sum of the values of the (informal) influence function is usually limited to 0.6. Thus the final *size* function varies in the range 1±0.3 of the *usize* function.

Usages

FPs are measurable on (detailed) requirement analysis models that follow the database paradigm as set out by the metamodel. The FPs quite nicely predict the effort of the rest of the software process [MO91]. In the empirical database used there the following linear-quadratic approximation function for the effort in person days is reported:

$$\text{effort} = 0.6 \times FP + 0.001 \times FP^2$$

These coefficients were gained from several 4GL developments and shows a ±15% bias. For more details cf. chapter 4. The Function Points are also successfully used as a productivity goal and measure. Industry surveys accomplished by DeMarco [DM88] and more recently by Rubin [RUB96] show productivities ranging from 0.5 to 10 FP. These ranges are partly due to substantial personal and technical reasons but also due to the biases occurring when measuring FPs.

Metamodel Assessment

①/❶ Metamodel completeness: The metamodel is a good representation of what a user sees of a database system. This view might also be called a "black box", requirements or domain analysis view. The algorithmic processing part however is only taken into account by one of the system-wide characteristics. Jones' Feature Points offer a remedy for this weakness by allowing to assess the complexity per user function. As a drawback of the metamodel we may consider its restriction to the software requirements layer, it is not useful for assessment and estimation of purely "technical" tasks. Furthermore, it does not allow the modelling of different situations of reuse, i.e. the existence of frameworks or libraries for parts of the requirements formulated in the domain model (cf. 3.2.1 for more details on our metamodel of reuse).

②/❷ Time of availability: The FPs are measurable early within a software process. However, it requires a fairly detailed domain analysis model with a specification of the access types from user functions to user data elements. In the context of shortened development cycles (e.g. prototyping cycles) one needs estimates and measurements before such a detailed model is elaborated. Our empirical research shows that typically around 15% of the software process effort is spent on the elaboration of a domain analysis.

Estimates - also in the context of project proposals, offers and contracts - should however be obtainable with less effort (cf. our proposal for applying the System Meter for preliminary analysis in 3.2.2).

❸ Rigidness: A rather weak point of the FP metric is its absence of rigidness. It may not really be called an objective measure that could automatically be measured unless (semi)formal requirements statements were available. According to [KE93] this leads to high interrater and intermodel biases. The interrater bias may be observed when different human raters count the same system. The intermodel bias is introduced by the weak formal definition of the metamodel's entities like "file", "data element" and - most of all - the "user function". Different approaches to modelling, e.g. SA/SD modelling, ESA/MSA modelling or OMT modelling yield very different results for the same systems.

④ "Non-Fakeability": Because the user requirements are (usually) introduced in a reviewed and stable manner, the possibilities for cheating on Function Points are low. The rather opaque manual rating procedure however leaves some marginal possibilities open. When requirements are gathered in a (semi)formal way, e.g. using CASE tools and the FPs are measured automatically, then FPs are sound with respect to this criterion.

Formal Assessment

①/❶ Counting: Even though the FP definition contains historical factors to be summed up, we may consider the unadjusted FPs an essentially counted measure, thus being absolute scale. The final adjusted FP count, however, is not a counted value and therefore its scale is subject of discussion.

② Single items: The FP metric can be applied to single items of its metamodel except the system-wide characteristics.

③ Sets of items: Again the metric is well defined for any set of items except the system-wide characteristics.

④/❹ Elemental items: The granularity of items is sufficient except for the system-wide characteristics which must be assessed as "monolithic" blocks for the whole system. Jones' Feature Points remedied this deficiency for the complexity characteristic. The rest of the characteristics remained system wide, though.

⑤ Dimensionally safe: The unadjusted FP metric unifies files, data elements, data accesses and user functions in order to render them summable. While this is semantically questionable we may not formally state any flaw.

⑥/❻ Dimensionally uniform: Again except for the system-wide characteristics, we may apply the FPs uniformly to all entities of the metamodel.

❼ Sequencing: The ordering of the analysis artefacts is irrelevant for the FPs.

❽ Naming: The naming of the artefacts is also irrelevant.

⑨/❾ Triangle condition: This condition is fulfilled for the unadjusted Function Points, i.e. the FPs without the system-wide characteristics. When taking the characteristics into consideration we, again, face some problems: first it is difficult to define a sensible difference operator for the characteristics, second, depending on the difference operator chosen, the condition may be fulfilled or not. If we take the maximum characteristic for a combined system description, the condition is violated because the high value is multiplied with the unadjusted part of the previously lower rated value, yielding a positive residual for the combined system description. When taking the minimum value, on the other hand, the condition is always fulfilled. Choosing difference and combination operators between the maximum and minimum may yield any result, i.e. either fulfilling the criterion or not.

Practical Assessment

A very positive aspect is the good empirical correlation to overall effort for database application developments. This allows for fairly accurate and practicable estimation in this area. Old fashioned terminology and unnecessary historical factors [KI93], however, make the use of this measure difficult in modern environments. There is much room left for interpretation. Some remedial evolvments were proposed to tackle this problem: cf. Symons [SY93], Graham [GRA95] and Sneed [SN94].

Summary

The FP measure may be gained early in the development life cycle, i.e. as soon as a detailed modelling of the user functions is done. The values correlate quite well with the remaining development effort (our empirical database shows a 95%-confidence bias of approximately ±20%, cf. 4.2.2). The FP measure, furthermore, is a good measure to track the productivity of the analysis phase in a development project. It is also of use when tracking overall development productivity as long as the inner complexity in the compared projects remains approximately the same as it does for database applications.

The FP measure, however, shows quite a high "interrater" and "intermodel" bias. Thus good estimating results can only be achieved when systems are modelled with the same method and measured by the same persons. Furthermore, the FP measure is not applicable to purely technical tasks and is only suited for the database paradigm.

Two well-known minor drawbacks of the FPM seem to be amplified in modern environments: 1) the relative lateness of its availability (approximately 15% of the effort is already spent) and 2) its ignorance of reusable components.

Finally the adjusted FP measure shows some formal flaws due to the mathematically arbitrary incorporation of the system-wide characteristics. The unadjusted Function Points, nevertheless, are formally sound.

2.2.5 Chidamber and Kemerer’s Object Oriented Metric Suite

Based on work by Morris [MORR89], Chidamber and Kemerer [CHI94] present 6 metrics defined especially for object-oriented systems. Even though none of those metrics are designed as true metrics of complexity but as descriptive design quality metrics, we will briefly assess the metrics with respect to our metamodel evaluation technique. We did not, however, assess them with respect to our (semi)formal criteria for complexity measures, because they do not apply for quality measures.

Metamodel

The C/K measures are explicitly based on the WW-metamodel. The C/K measures however only make use of the following part of that model:

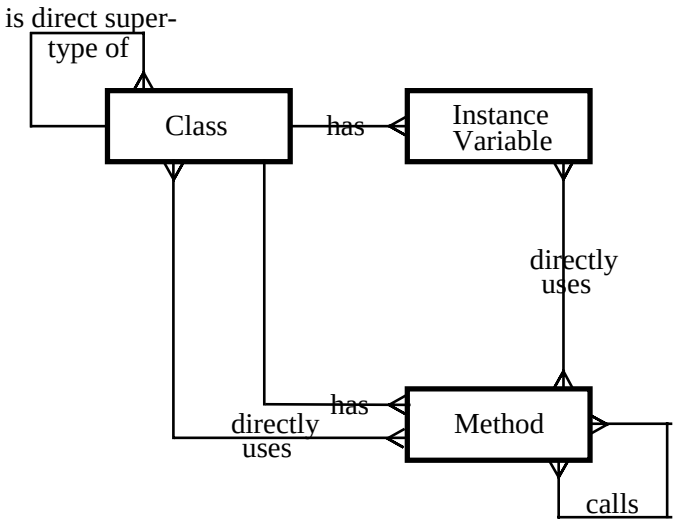

Figure 21: The Subset of the Wand/Weber Metamodel as used by Chidamber and Kemerer

We point to the fact that the C/K measures only consider the newly introduced instance variables and methods of a class, instead of considering both the implemented and inherited features. This was done so to be able to measure more easily the values on real systems.

Definition

All C/K measures are defined for a single class and must be aggregated to give numbers for a total system or subsystem.

In order to save space when formally giving definitions, we use the following abbreviation for the metamodel concepts: S(c) = direct subclasses of a class, M(c) = methods in a class, I(c) = instance variables in a class, M(m) = methods called in a method, C(m) = classes used in a method, I(m) = instance variables used in a method. The 6 metrics thus are:

1) The weighted methods per class:

$$WMC = sum_{M(c)} [c(m)]$$

In this formula c(m) denotes the weight (or complexity) for a method measured by LOC or McCabe's ν. When c(m) is set to 1, WMC equals |M(c)|, i.e. the number of methods in a class.

2) The depth of the inheritance tree relative to a class:

$$DIT = 0: \text{ if } S(c) \text{ empty}$$
$$\max (DIT (S (c))) + 1: \text{ else}$$

3) The number of children:

$$NOC = |S (c)|$$

4) The coupling between objects:

$$CBO = |\cup_{M(c)}[C(m)] \vee \cup_{I(c)}[C(i)]|$$

5) The response for a class:

$$RFC = |M(c) \vee \cup_{M(c)}[M(m)]|$$

6) Finally the so-called lack of cohesion in methods:

$$LCOM = sum_{M(c)} [lco(m)]$$

In this formula, the lack of cohesion lco for a single method is defined as:

$$lco = sum_{M(M^{-1}(m))} [dis(m,m_i)]$$

Therein the disjointness dis of two methods is defined as:

dis = 1: if $I(m_1) \wedge I(m_2) = 0$
-1:else

Usages

All measures are mainly used as descriptive quality metrics for which thresholds may be given in certain contexts (e.g. for C++ classes a CBO threshold of 20 may be reasonable for more details cf. [LO94] or [HS96]). A recent revision of the COCOMO estimation method [BOE95] has made use of the WMC and RFC metrics as a substitute for LOC. It was however formally criticised by [ZU94] and does not improve the inherent flaws of the method as explained in section 1.5.

Metamodel Assessment

❶/① Metamodel completeness:

This metamodel roughly represents what object-oriented systems are (statically). However, many facets that determine the inherent size/quality or complexity are not covered, for example:

1 The "normal" objects are not covered. There generally exist some global variables in a system which are neither classes nor class features nor methods. Those globals may help to structure a system but may also be a critical element of bad system behaviour (via "hidden" or overwhelmingly many couplings).

2 The distinction of which system elements belong to the measured system, the library used and the language used is not modelled. Especially in object oriented systems the limits of "what is what" tend to vanish. To have project control, nevertheless, we must know which system elements were "constructed" in the project and which were obtained from a library (or from the basic programming language, e.g. the control flow methods).

3 The notion of the "scope" of a system element X, i.e. the set of system elements ≠ X which may use (or even alter) X is lacking in the WW-metamodel. It is generally accepted that software elements should be "opened" for use and especially for change only to a minimal set of other software components. This minimal set may either be the whole system (for constants this is of no harm but for variables this is undesirable) or only a statement block (good for block (Smalltalk) or local (C/C++) variables).

4 Desired "side effects" (like device write/read, persistency write/read, object construction, alteration and deletion) are not modelled. Side effects make up two things in a system: a) they are a good deal of the system's functionality and b) they are a good deal of the system's complexity (e.g. write/read to a persistency device ("database") is a very wicked form of data coupling).

5 The distinction between denotational and implementational part of a system component is not modelled (the denotational part of a method is its name and the types of its formal parameters (i.e. its signature (C/C++), whereas the implementational part of a method is the set of messages which make up its implementation). This distinction is necessary to identify the effectiveness of some implementation (small implementations are preferred).

❷/② Time of availability: All the metamodel items are established in some first implementation phase only. Due to the nature of the metrics we cannot obtain them from object-oriented analysis models or design models based on patterns. This critique is true for

all metrics except the WMC, weighted methods per class. The WMC metric however ignores many aspects of object oriented analysis models, e.g. the relationships between classes and the non-method class elements.

③ Rigidness: Besides the "openness" of the WMC definition, i.e. it is left to the practitioner which kind of weight to use for method complexity adjustment, the C/K metrics suite is unambiguously defined due to the clarity of the underlying WW-metamodel.

④ "Non-Fakeability": Due to the rigidness and structural complexity of the metamodel, the C/K metrics may not easily be faked. This is a strong argument in favour of using the metrics as objective quality indicators.

Practical Assessment

Some of the metrics (NOC, DIT, WMC, CBO) have proved to be of great value in assessing object-oriented designs (cf. [SH93], [LO94]). For all of the metrics, however, it is not possible to give generally accepted thresholds that would define good and bad values. None of the metrics can be used as a sound measure of complexity except WMC. WMC however ignores many aspects that contribute to complexity and is not rigorously defined. LCOM is a very unsound measure for its intended purpose, because even very loosely cohesive classes (according to the Law of Demeter) can yield very low values and vice versa [HS96]. The use of RFC is still unclear.

Summary

Since no empirical data are available to the author, an assessment of the C/K measures can only be made based on a discussion of the underlying metamodel.

The sum of the WMC measure for all classes in a system could be a rough indicator of the effort/cost it takes to develop a system as they reflect some of the real "flesh" of a system (cf. metamodel remarks 1 and 4). The usefulness of the WMC measure however is highly dependent on how the methods are weighted. Moreover, as every C/K measure, WMC comes too late for early estimation.

The traditional system quality notion of coupling is quite nicely represented by CBO, but LCOM is not suited to measure cohesion (or its absence). Furthermore, in order to really be quality measures CBO and LCOM should not be counts but relative measures that are independent of the considered object's size. As the measures are defined now, values of small classes may not directly be compared to those of big classes. CBO and LCOM furthermore suffer from the fact that they are defined only for classes, although methods, instance variables, etc. also have their coupling and cohesion with respect to other objects. These non-class couplings and cohesions also severely influence a design's quality.

No C/K metrics are measures of quality in the sense that everyone agrees on which values are "good" or "bad". Local guidelines must evolve if the measures are to be used as quality assessment measures.

2.2.6 Other Approaches

Many other metrics than the ones just presented have been proposed. Only a few of them are truly defined for object-oriented systems though. The author is aware of several ongoing and partly finished research which may lead to new metrics [CHE92], [LA92], [SH93], [LI92], [RO89]. We do however not discuss these approaches here, because they are entirely focused on descriptive quality metrics.

Three recent proposals, however, explicitly address software sizing:

1. Harry Sneed's Object Point (or rather Data Point) metric is a simplification of the FP metric which drops the functional part [SN94]. This is useful for database-oriented applications, because it simplifies the modelling and measuring task, thus supporting the criterion of low estimation effort. It does not address other requirements such as the incorporation of a complete object-oriented metamodel (including reuse and non-tiedness to database applications).
2. The Task Point metric is a development of the Swiss Bank Corporation of London (contact: Mark Lewis, Bezant Ltd., Wallingford, UK) and is supported by a Task Point Collection Club [SBC95] initiated by I. Graham [GRA95]. It can be seen as a variant of the FP metric that has been terminologically adapted to business process descriptions, but still essentially is equivalent to FPs. We therefore did not consider the Task Points, which are a sound approach of their own, for our purposes any further.
3. Henderson-Sellers' and Pant's S/C metric (size/complexity) is an implementation tied combination of the metric LOC and McCabe's cyclomatic complexity. Even though this approach is reasonable and led to an intuitive and easily measurable metric, it still ignores some essential concepts (data abstraction, inheritance, information hiding, reuse) of modern software engineering. Recent empirical studies [HS96b] showed that S/C does not perform significantly better than LOC or McCabe's ν with respect to effort correlation. Furthermore it is measurable only after implementation which is too late for development effort estimation. We adopted the idea of evolutionarily combining known measures into a new one but tried to include more concepts (e.g. the Halstead concept of counting token-based and considering unique operands (=object instances), the FP concept of persistency operations, etc.). We also added new aspects e.g. information hiding by separating the external and internal complexity and reuse by differentiating between reused and newly built components.

2.3 Existing Measures and Metamodels for Software Processes

In this section we will briefly explain the most fundamental measures and underlying paradigms for software processes, i.e. the (mostly) human activities that ultimately result in software.

2.3.1 Effort, Duration and Staffing

Effort

The most prominent measure for the software process - as well as for any other human process - is the effort. Effort may be defined as the expenditure of resources needed to run the process from start to completion. It requires, as its metamodel of processes, a clear view of what activities - and as a consequence what resources - belong to the process. The metamodel therefore is that of a so-called "defined" process (cf. [HU89]) with a crisp border distinguishing the inside and the outside:

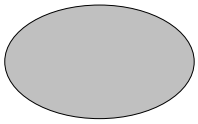

Figure 22: The Effort of a Process (shaded area)

Effort is usually measured in units like man-months, man-days or similar metrics. We use the neutral term person day (pd) as our base metric of effort. The pd definition honours

effective process-specific work time only, i.e. without holidays, leaves for sickness and other absences. Individual differences in productivity are not captured with this measure. Instead we will introduce a specific productivity measure in 2.3.4.

Relations to other metrics of effort are given as follows:

Measure	***Relation to Person Days***
person-hour (ph)	= 1/8 pd
person-week (pw)	= 5 pd
person-month (pm)	= 20 pd = 4 pw
person-year (py)	= 200 pd = 10 pm

Table 5: Relations between different Metrics of Effort

Because the measurement of effort is usually biased by local i.e. corporate-specific reporting and accounting standards as well as by managerial issues (e.g. not reporting overtime, no longer reporting effort on accounts as soon as they have reached budget, etc.) we never indicate values of effort with more than 2 digits of significance.

Effort expenditure may also be expressed as a dimensionless number, the ratio of effort spent to total effort estimated or budgeted. This ratio can be compared to the degree of completion (cf. 2.3.3) for progress tracking.

Duration

Based on the same metamodel of a defined process, which has a well defined starting point and a termination point on the time axis, we can define the duration as the elapsed time between start and end:

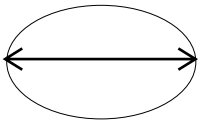

Figure 23: The Duration of a Process (start to end distance)

Duration is usually measured in units like months, days or similar metrics. We use the so called "elapsed" day (d) as our base metric of effort. The d definition includes the standard work times between the start and end dates of the process. It is not sensitive to holidays or other absences of individuals (it however takes into account corporate holidays). Relations are given as follows:

Measure	***Relation to elapsed days***
hour (h)	= 1/8 d
week (w)	= 5 d
month (m)	= 20 d = 4 w
year (y)	= 200 d = 10 m

Table 6: Relations between different Metrics of Duration

Measurement of duration is usually - as well as the measurement of effort - dependent on local standards and biased by similar managerial issues. We therefore never indicate values of duration with more than 2 digits of significance. In practice we usually took the date of the official project kickoff-meeting as the start date and the date of official user acceptance as the end date of processes spanning the full development life-cycle. For partial software processes we used the official start and end dates of the covered phases.

Elapsed time may also be expressed as a dimensionless number, the ratio of time spent and total time estimated or budgeted. This ratio can be compared to the effort expenditure ratio or the degree of completion (cf. 2.3.3) for progress tracking.

Team size

The last fundamental process metric, average team size, is derived from the two measures above as follows:

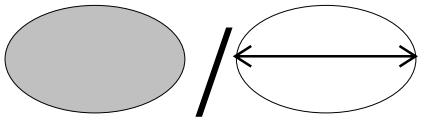

Figure 24: Average Team Size as the Quotient between Effort and Duration

Non-averaged team size is, however, a more complex parameter which dynamically varies during a software process. Refer to the following subsection 2.3.4 for more details on metrics and metamodels of project dynamics.

2.3.2 Cost and Revenue

The management of a Software Producing Unit (SPU) is usually primarily interested in monetary data (measured in "metrics" like $, £, CHF, FF, etc.). The effort as defined in the previous chapter 2.3.1 is the major component of a software process' cost. Because this work's topic is software engineering and not cost management, we will not elaborate the means of cost budgeting and controlling in detail. The following cursory comments on the links between our instruments and cost control instruments will be everything one finds in this document on this topic.

As already stated, the cost - to be precise: the direct cost - of a software process may be primarily derived from its effort metric by multiplying it with a mixed or resource-specific cost factor (i.e. including the direct costs of the resource(s) per unit of effort). Some casual fixed cost directly attributable to the software process in focus, e.g. for specific development hardware or software, should secondarily be added to the effort cost.

Definition of the revenue of a software process is often also directly related to effort depending on the kind of contract between the customer and the SPU. In case of "body-shop" contracts the customer pays directly per person day, i.e. effort expended, whereas in fixed price contracts he usually pays on result delivery or in rates. The SPU's favourite form of contract is usually the "body-shop" mode, because it transfers the risk of bad estimates to the customer.

We may optionally extend a software process' revenue metric to revenues that are not immediately linked to the process but nevertheless attributable to it, e.g. by adding some of the maintenance contract sum to the previous development process. This technique, however, tends to make things complicated, because the metric will no longer be measurable immediately after process completion but only one or even more years later.

A third derived monetary parameter that can be measured may be called a project or process cash flow, i.e. the difference between revenue and (direct) cost. Sometimes it is very important to document the planned cash flow before a process has started because it may (also in commercial environments) purposely be negative, e.g. in the case of so-called "door-opener" projects.

To conclude this brief subsection on financial issues we state that 1) financial aspects should always be treated and documented separately from the model based estimates of

effort and 2) one should always keep the link - also if there is none - transparent and documented.

2.3.3 Degree of Completion and Process Completeness

In order to understand the following two process metrics we have to extend our process metamodel with the notion of the process result. For the sake of simplicity we ignore the fact that there are possibilities of 1) multiple results and 2) intermediate results. The practitioner has to deal with those notions in a similar way to the one explained for sub- and span-processes (2.3.7). By incorporating the result into our process metamodel we have included all product metrics as process metrics as well - we will make use of them immediately.

Degree of Completion

The degree of completion of a process measures the state in which a running software process is. It is based on a percentage comparison of the result achieved so far versus the planned result. An analogy from construction would be to state "We are 50% done with that wall.", when the wall is planned to be e.g. 12 meters long and 6 meters are done. We formally define:

%C = size (result at work) / size (planned result)

In general, for an unstarted process %C is 0% unless we have some parts of the result already prebuilt. For a terminated process the value is always 100% because we assume that the planned result is explicitly or implicitly always adapted to reflect the current reality.

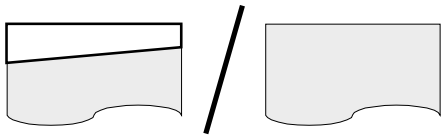

Figure 25: %C, the Ratio of Achieved versus Planned Result

%C is a typical derived metric that makes use of a metric of size (or complexity). It requires that this metric is applicable on result plans (or models) as well as on the result. The scale of %C is absolute and dimensionless. It presumes that the scale of the base metric is ratio or absolute scale and dimensionally safe.

When tracking project progress, this ratio is typically compared to the percentage of expended effort and percentage of elapsed time.

Process Completeness

Another derived ratio metric is called process completeness or %P. It is - either at the planned or actual level - the ratio of the results developed to the results foreseen in some process template. This implicitly extends our process metamodel towards a quality management metamodel (as outlined in 1.7) where every process and result follows some standard or template. In practice we cannot define a standard software process that fits every concrete case. In a special planning step called "tailoring" [HERMES95] the often maximal template is cut down to the actual needs. In order to keep differently tailored processes comparable, the effects of the tailoring are quantified as

%P = size (concrete results) / size (template results)

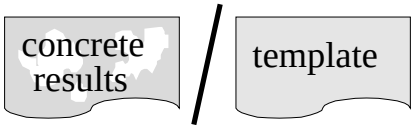

Figure 26: %P, the Ratio of Concrete and Template Results

To give an example: when the template process defines that an object model and a dynamic model should be elaborated, but we only elaborate the object model (because we have a database application only), then %P will be some value around 40%. In order to give exact values, the different template results that are cut must hypothetically be elaborated and assigned a size, according to the original formula. This unsound step can be avoided by assigning empirically determined %P values for each kind of template result. This empirical %P value is calculated as the effort ratio of the result elaboration effort to the total process effort. %P is then approximately definable as the sum of the empirical %P for all the kinds of results that are not tailored away (cf. 3.3.3 for a list of template results with empirically determined %P values, cf. 4.1.3 for details of %P measurement in our surveys). %P can take on values from 0% to over 100%. Values over 100% can occur when repetitive tasks have to be performed which are foreseen only once in the template, e.g. when a user interface has to be translated into several languages or software has to be installed at several sites.

2.3.4 Derived Measures: Productivity, Velocity, Acceleration, Inflation

For software processes (as well as other human processes) many measures may be defined derived on top of the basic measures explained in 2.3.1 and 2.3.3. We present a few here which we found to be useful in the context of our quality management system [MO94] and for which we have gathered some empirical data.

Productivity

The measure of productivity P is naturally defined as the ratio of result size to process "size", i.e. effort. It measures the amount of output relative to the input into the process.

P = size (result) / effort

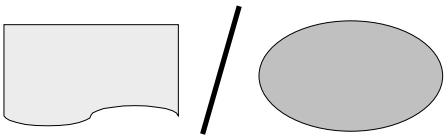

Figure 27: Productivity P, the Ratio of Product and Process Size

This notion of productivity, however, which is measurable immediately after process completion, may be too short-sighted. It can be optimised without producing really usable and useful results. We therefore also introduce the notion of quality based productivity which in turn is based on the empirical result quality Q. Q is the ratio of the size of the unchanged parts of a software product after some time of usage (default = a year) to the original size.

Q_x = size (unchanged result parts after time x) / size (result)

$Q = Q_1$

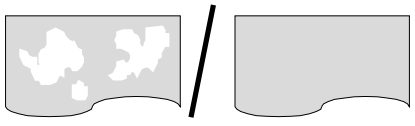

Figure 28: Quality Q, the Ratio of Unchanged versus Original Result

The obvious drawback of Q's definition is its late point of measurement. It also requires the metric of size to be applicable to subsets of results as well as supporting elements of fine granularity in order to support crisp distinctions between unchanged and changed parts of results. These requirements are part of our (semi)formal requirements list for metrics of complexity (cf. 2.1.4) and we honoured them in the definition of the System Meter (cf. 3.2.1). Quality based productivity PQ is then defined as

$$PQ = P \cdot Q$$

Our observations showed that P and PQ may differ significantly, typically PQ being 20% below P. This means that in average projects, 20% of the originally released code (and design and requirements) change within one year of production. Our empirical sample set, however, is too small for statistical evidence. We assume that an industry wide survey would yield lower values for this change rate, because our observed processes were highly customer driven.

Velocity

Similar to productivity we define a process' velocity V as the ratio of result size to process duration.

$$V = \text{size (result)} / \text{duration}$$

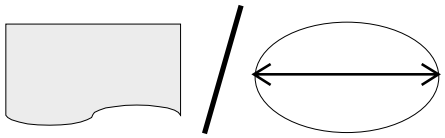

Figure 29: Velocity V, the Ratio of Product Size and Process Duration

Sometimes mere velocity is more important than productivity, because a product has to be developed for some fixed deadline (e.g. the effective date of a new taxation law). It is therefore important to distinguish between the two ratios. Often velocity is tried to be raised by adding manpower to a late software project. But, as Brooke has observed in his famous book "The Mythical Man-Month" [BR75] we may eventually observe that "adding manpower to a late software project makes it even later". This is true for large teams where the coordination effort exceeds the contributions to the desired result. As a consequence from this "law" we observe that software development cannot be accelerated to unlimited speed in spite of increasing effort spent on it. The exact empirical limitations and correlations between acceleration and effort inflation (and therefore team size inflation) become measurable using the two following measures:

Acceleration

Acceleration A of a software process is the ratio of 1) the estimated duration and 2) the duration imposed by some fixed deadline.

$$A = D_{estimated} / D_{imposed}$$

The measure only makes sense for values between 1 and 3. Values above 3 are beyond an empirically observed [NO70][PU80] limit. This limit may be substantiated by the following case: a piece of software estimated to take 10 person years simply cannot be produced by 100 developers in one month. Values below 1, i.e. when the deadline is behind the ordinarily estimated schedule will not affect the dynamic aspect of a process[14] and are therefore not of interest.

[14] It has to be noted, however, that overly delayed processes (e.g. one person working for 5 years on a 5 person years project) will also have negative effects on productivity, i.e. will inflate it, too (cf. [DM83]).

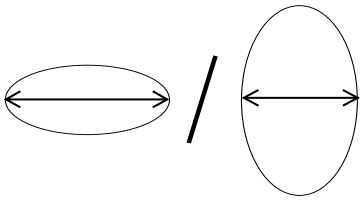

Figure 30: Process Acceleration A, the Ratio of Estimated and Imposed Durations

Accelerated processes typically require bigger teams to be established more quickly. Overall effort, however, does not remain constant, mainly because of the increased coordination demands of the bigger team. In order to empirically measure these effects we define:

Effort Inflation

The inflation I of a process is the ratio of the effort under "pressure", i.e. for the accelerated process to the estimated effort under normal conditions.

$$I = E_{accelerated} / E_{estimated}$$

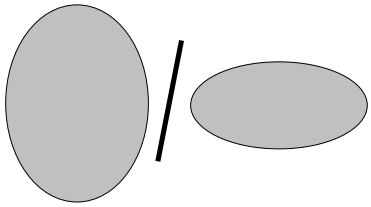

Figure 31: Inflation I, the Ratio of Accelerated and Normal Process Efforts

Theory and empirical studies conducted in the eighties [PU80] [DM82] have shown a correlation between A and I as follows:

$$I = A^2$$

We assessed this proposed correlation in four projects (cf. 4.2.4) and observed values in rough accordance to it. In practice, however, when assigned the task of estimation and scheduling, we seldom make use of those so called "dynamic cost models", because the values of A are often only slightly above 1 (typically around 1.0 and 1.2). The imposed schedules, furthermore, are within the estimation variance of the duration and we made it a rule not to apply dynamic modelling in these cases. For a few more severe cases we did the modelling with the result that the customer chose project proposals with less tight schedules at reduced cost. We conclude that dynamic cost modelling is a usable instrument for contract negotiation and it is used in step C of the ABC estimation strategy (cf. 2.1.1).

2.3.5 Waterfall Models

The task of developing software is complex. The overall software process is therefore traditionally - and according to established techniques of project management - partitioned into so called phases. The most commonly used phase model distinguishes between 4 phases that correspond to major indispensable results, i.e. ① requirements engineering / analysis, ② technical design, ③ implementation and test and ④ maintenance. The fourth phase of maintenance actually is a purely managerial phase that drives small cycles of the previous 3 phases to adapt and correct software during software operation. The result of some previous phase is the prerequisite of the follow-up phase as shown in the following diagram:

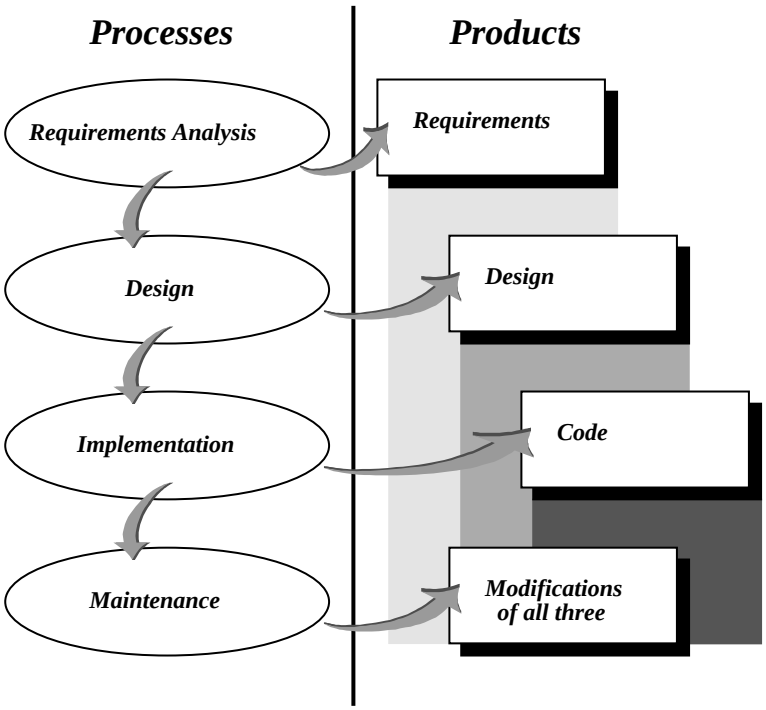

Figure 32: The Traditional Waterfall Process Model

We thus observe - for phases ①-③ - a 1:1 relationship between phase and software product layer (as introduced in 2.1.4). Those phases can be viewed and treated as sub-processes (cf. 2.3.7).

More schematically diagrammed, i.e. with the process or dynamic dimension on the Y-axis (from top to bottom) and with the product dimension on the X-axis (from left to right), we observe that the waterfall paradigm proceeds through the fields of software like wearing blinkers, especially there is no prototypical looking forward:

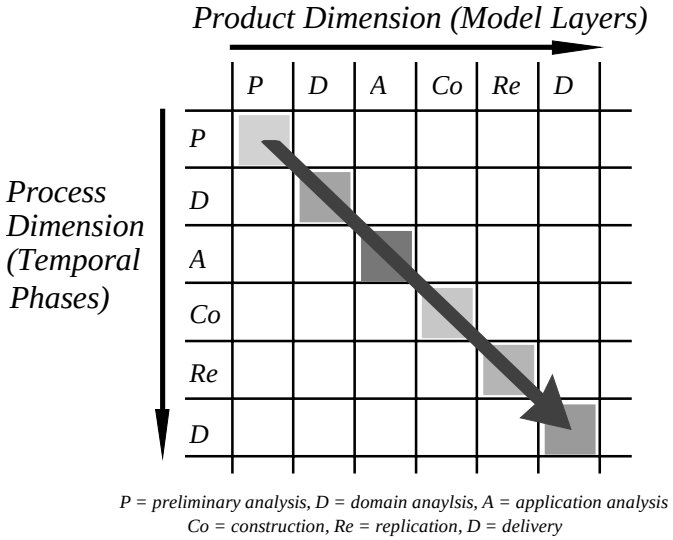

Figure 33: The Waterfall Software Process in a Process-Product Diagram

In this figure we further refined the phase model according to OMT [RU91], SSADM [SSADM90] and others. We split the requirements analysis phase into 3 stages of ① preliminary, ② domain and ③ application analysis which elaborate the three analysis layers of ① rough functional goals and system scope, ② the essentials of the "real" i.e. technology independent system to be supported and ③ the specification of the computerised system. Construction corresponds to technical design whereas the implementation and test phase is

enhanced by the additional delivery phase which comprises acceptance tests, installation, education as well as the organisational and technical preparation of the maintenance phase. Waterfall models typically are problematic because none of the stages can profit the experience of looking ahead to future layers. Especially application analysis may completely be invalidated when the specified core concepts may only poorly be supported by technical design patterns in the chosen implementation environment.

2.3.6 Spiral, Fountain and other Non-Waterfall Models

The first basic idea to improve the process structure, whilst not actually violating the waterfall model, is not only to partition the process but also the product, i.e. software system to be built. This partitioning, however, cannot be made before we have at least some coarse model of the system at hand, so we can "cut" it into suitable pieces. Preliminary analysis, therefore, may typically not profit from this technique. For each of the system parts (usually called subsystems, versions or realisation blocks) a waterfall process is instantiated. The benefits are that 1) the customer gets parts of the system earlier, 2) he can give feedback before the total system is specified and 3) the developers may reuse and improve technical patterns from earlier cycles. This approach was chosen in the SOMA life-cycle model [GRA94]. SOMA restricts the process instantiations on subsystems in a time-box manner, i.e. each cycle is limited to say 4 months of development duration. From our practical experience, we may highly recommend time-box cyclic development after a system-wide domain analysis. If we split up the system earlier, we risk that the evolving implementations prohibit the seamless incorporation of essential concepts modelled later.

In order to plan and estimate the processes for the subsystems, a useful derived product metric is the part fraction %F:

%F = size (part) / size (total system)

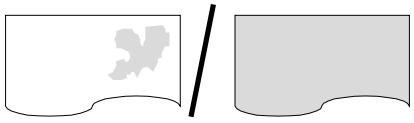

Figure 34: %F, the Fraction of a System Part

This derived product metric requires a base measure of size (or complexity) that allows a dimensionally uniform assessment of any system part, i.e. independent of its nature. This requirement is reflected in our (semi)formal requirements list in 2.1.5 and is fulfilled by the newly developed System Meter (cf. 3.2.1).

Spiral, Fountain and Pinball Models

More revolutionary than the system partitioning approach are three recent propositions by Boehm [BOE81], Henderson-Sellers and Edwards [HS94] and Ambler [AM94]. The spiral model of Boehm proposes the 4 phases of 1) setting objectives, 2) risk analysis, 3) prototype/product development and 4) evaluation and (re)planning to be iterated until the product is finished. It was observed, however, [BOE86] that this model is not suited for building software under contracts because, one never knows when to stop iterating. This problem is also known as "the customer always wants more" syndrome.

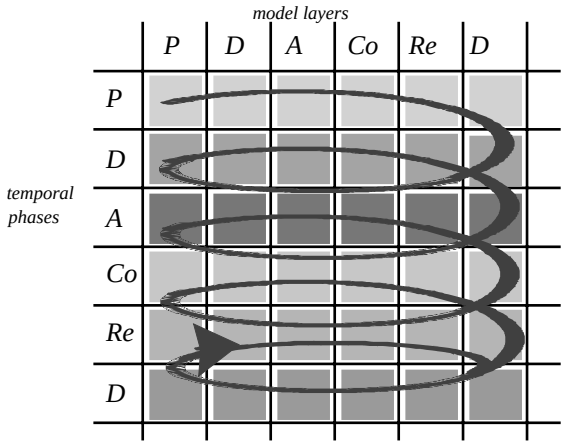

Figure 35: An Unrestrictedly Cyclic Software Process in a Process-Product Diagram

Henderson-Seller's and Edward's fountain model is more realistic in the sense that it retains some sequence, i.e. requirements study, analysis, system design, component design, coding, unit testing, system testing. It furthermore allows that each phase is overlapped with one or even two of the next phases before it is completed. Feedback from the later phases may therefore "drop down" fountain-like into the result-pools of earlier phases. We adopted a similar process model (cf. the restrictively cyclic model described in 3.3.2) which is more formalistic though, maybe due to our ISO 9000 certified quality management system [MO94].

Ambler [AM94] proposed an even less restrictive so-called pinball model where the developer can chose at any time depending on arbitrary needs, from any development activity (e.g. find classes, find attributes, program, test, define subsystems). This model comes close to reality, we have to admit. However, in order to produce systems with a predictable schedule and quality some more prescriptive standards have to be followed. During small prototyping processes we may totally adhere to Ambler's pinball model. For the overall software process, however, we would not recommend it.

2.3.7 Sub-Activities and Span Activities

Sub-Activities

Phases and the activities within the phases partition the total software process. In a generic view of this partitioning, we may observe super-activities and sub-activities. Let us consider one activity (or process) P together with its sub-processes P_1, P_2, ..., P_n, e.g. the domain analysis process with its steps of object modelling, use case modelling, dynamic modelling and object model refinement.

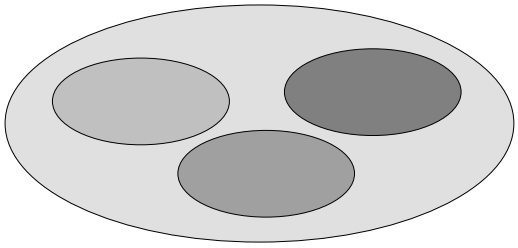

Figure 36: An Activity with its Sub-Activities

With respect to the effort metric E we observe that

$$E_P = E_{P_1} + E_{P_2} + ... + E_{P_n}$$

The duration metric is not as easily determined because sub-activities may overlap. But here we are not interested in the aspects of time, because in practice the schedule for sub-

activities is never established with mathematical or statistical means but always by planning the actual resources upon their availability (i.e. considering vacancies, other projects etc.).

Regarding effort, one observes that values for sub-activities are proportional to values of super-activities. Especially for the phase efforts E_{P_i} versus the total effort E_P we have:

$$E_{P_i} = l_1 \cdot E_P$$

We empirically regressed the linear coefficients l_i of this proportional correlation for our software process template called BIO (cf. 3.3.3). This correlation is itself a small estimation model as defined in 2.1.3, i.e. one may either estimate the total effort from the phase effort (e.g. derive the total effort from the effort of the preliminary or domain analysis) or vice versa (e.g. when the total effort is given one may estimate the budget for technical design / construction). Very often these small estimation models are used as chained models (cf. 2.1.3) after some initial estimate as part of step C of the generic estimation procedure.

In general the complete calculation for a sub-activity also takes into account the sub-activity's process completeness %P and degree of completion %C as well as the subsystem fraction %F. Thus, when the known value of the calculation is some given (or estimated) effort E_b of some activity with linear coefficient l_b over a subsystem with fraction $\%F_b$, we have for the remaining effort of the sub-activity i:

$$E_{P_i} = (E_b / (l_b \cdot \%F_b)) \cdot \%F_{P_i} \cdot \%P_{P_i} \cdot (1-\%C_{P_i}) \cdot l_{P_i}$$

This may look complicated, but it is not; everything is purely linear. This formula[15] is useful for both: 1) productivity tracking of partially completed processes (and sub-processes) as well as 2) estimating the remaining effort of partially or wholly incomplete processes.

Span Activities

A special kind of sub-activity exists, however, that does not exactly obey the rules just outlined: the span activity. The span activity is typically of supportive nature. It does not belong to the actual core production processes. Examples of span activities are project management and quality management. Those activities do not have an existence in their own right, they depend upon the core activities. Their start and end dates adapt to those of the super-activity, i.e. they "span" over the entire super-activity.

When doing calculations for span activities we can therefore not apply the same formula as for sub-activities. Although we also have an empirically regressed coefficient l_{span} for the span activity and its process completeness $\%P_{span}$, we need as input the total core activity effort E_{core} (which may be summed up from multiple efforts calculated with the formula above) as well as l_{core} the sum of the non-span coefficients (which can also be calculated as $1 - \Sigma l_{span_i}$). We then may use the formula

$$E_{span} = (E_{core} / l_{core}) \cdot l_{span} \cdot \%P_{span},$$

to calculate the effort of the span activity to support the remaining core activities.

[15] Note that we consider percentages as values between 0 and 1, e.g. 0.5 equals 50%. This simplifies many of the equations given in this thesis.

3 New Approaches

After having presented an analysis of existing solutions for estimation and measurement of software and software processes in chapter 2, we will now present in detail our constructive contributions to the topic. Our own new propositions are evolutionary i.e. based on some of the most useful ideas previously published. The main idea we adopted to metrication was that of metamodelling i.e. building (semi)formal models of the domain in focus. When deriving new metrics from the metamodels we were additionally guided by the criteria used to analyse the existing solutions. We also used metamodelling to define the software process, i.e. a reference model for the software process. In this area we were not overly restrictive in order to allow practical mappings from real software processes to the theory.

The new solutions, namely the BIO software process, have been in practical use in several companies since 1993. The new metrics were, however, only used as an "ex post" means of analysis and validation of themselves. Since June 1996 two metrics, the System Meter for preliminary analysis results (PRE) and for domain analysis results (DOME), have been in practical use as predictive metrics of complexity and as a basis for productivity tracking and analysis.

3.1 New Generic Concepts of Estimation and Measurement

In order to stay compatible with existing approaches and knowledge, we adopted many of the generic concepts presented in section 2.1, namely the ABC estimation strategy, the calibration techniques and the techniques of chaining and backfiring. In the field of metrication we found the AMI/GQM approach particularly useful but it lacks guidance and means for developing completely new metrics. AMI/GQM instead, presumes that one can chose from a list of pre-defined metrics. This gap was filled with our metamodel-based metric derivation strategy. After defining this strategy, we found it useful for evaluating the non-formal part of metrics, i.e. its meaning or semantics. In order to reuse the definition of metrics for different metamodels (comparable to the definition of the measurement of length for different real world objects) we also introduced and used a metamodel mapping technique.

3.1.1 Derivation of New Metrics with Metamodels

Let us assume that the AMI/GQM method led to the question "What facts of a piece of (object-oriented) code let us predict its testing effort?" in order to fulfil the goal "Improve test effort estimates". The answer is obvious at first sight: the code's complexity. But what is that? A metamodel of the object in focus, i.e. the 'piece of code' in our example, must be elaborated in this case (as done in sections 3.2.1 and 3.2.2). The complexity metric should then obey the constraint that every entity type and relationship type of the metamodel should be accounted for. Other kinds of metrics, e.g. quality metrics, can also be defined by using metamodels but should fulfil other kinds of constraints. We did not further investigate in this direction.

Metamodelling is not a particularly difficult thing to do. It may be conducted as a regular domain modelling process (cf. 3.3.3), e.g. using the two steps:

① Identification of the objects (classes) in the domain

② Identification of the associations between the objects

The only possible pitfall is the fact that modelling itself is modelled. The domain objects thus identified typically are classes, methods, messages, instance variables, etc. and the domain associations are e.g. "a class may contain one or more methods", "a method may

contain one or more Messages as its implementation". Ordinary models in contrast, deal with real-world entities like customers, accounts, invoices, etc. One should therefore never confuse the two terminologically overlapping areas of model and metamodel. One typical remedy for this problem is attributing a metasign (like $) to the signature of the metamodel objects. Because, however, we exclusively deal with metamodels here, we didn't use any metasigns.

The last step of metamodel-based metric derivation is to define the new metric in terms of the elements found in steps ① and ②, e.g. by defining a class' complexity as the number of methods it contains.

Derivation of a Sample Metric

In order to exemplify the metamodel based metric derivation technique we derive a metric from the sample (meta)model given in section 2.1.7 on metamodelling. Formally denoted the (meta)model is:

$Population \equiv \langle P, L, C, bc, co, li \rangle$, where
$P = \{p1, p2, p3, \ldots\}$ a set of people
$L = \{l1, l2, l3, \ldots\}$ a set of locations or places
$C = \{c1, c2, c3, \ldots\}$ a set of countries
$bc: L \to \{IsBigCity, IsNotBigCity\}$ a function
$co: C \to \wp(L)$ a function "country" with its inverse being definite
$li: L \to \wp(P)$ a function "lives in" with its inverse being definite

Figure 37: A Sample (Meta)model Denoted Formally

We would now like to define a metric named %PinBC which stands for the percentage (%) of people living in big cities (in a certain country). This may be accomplished with a definition formula like:

$$\%PinBC(\gamma) \equiv \frac{\left| \biguplus_{\lambda \in co(\gamma) \ \wedge \ bc(\lambda) = IsBigCity} li(\lambda) \right|}{\left| \biguplus_{\lambda \in co(\gamma)} li(\lambda) \right|}$$

where $\biguplus$ denotes the union operator on sets and $|\ |$ the cardinality operator on sets (i.e. returning the number of elements in a set).

The metamodel-based technique is, however, not the only one for all kinds of metrics. Especially dimensionless metrics, i.e. percentages or ratios, are as well defined using a base metric of size (or complexity) and the following two strategies:

Deriving New Metrics from a Metric of Size

① The first case in which we may make use of a base metric of size to derive a new metric is when a ratio metric for a subset or part of a global set of things is needed. The strategy is as follows:

1) Set/Subset Search:

Try to formulate the metric's idea as a set/subset sentence, e.g. the external coupling of a class may be formulated as the ratio of the subset of externally coupled elements (the class itself, its methods and features (= class and instance variables)) to the total set of class elements.

2) Abstraction Search:

Find the nearest abstraction A that encompasses all the kinds of elements used in the set/subset statement. E.g. a class, method and feature are different things but all may be viewed as software components which then may serve as the abstraction A.

3) Metric of Size Search:

Finally, try to find a metric of size S_A for A. The new metric NM is then definable as:

$$NM \equiv S_A \text{ (subset)} / S_A \text{ (set)}.$$

② The second case we may use a metric of size is when the anticipated derived metric compares two different things. E.g. if one wants to define the velocity of a software engineer, one may proceed in two steps:

1) Analogy Search:

Try to find real world analogies, e.g. in physics, and map them to the software engineering domain: e.g. time t maps to the elapsed time t_p of a software process and distance d to the size metric s_{sw} (having the property of a mathematical metric as explained in 2.1.5) of the changes the software engineer has made to some software artefact.

2) Define Metric in Real World Analogy and Map Back:

Then define the metric in the real world domain and use the mapping to derive the analogous metric in the software realms:

$$\text{velocity} = d/t \quad \Rightarrow \quad \text{velocity}_{\text{software engineer}} = s_{sw}/t_p$$

3.1.2 Assessment of Metrics with Metamodels

In section 2.1.6 we elaborated formal requirements for metrics assessments and used them in 2.2 as part of our analysis of existing metrics. Formal assessments were also - more rigidly though - done by others like Zuse [ZU91] [ZU94] and Fenton [FE91] [FE92]. The most widely used criteria set of Weyuker [WEY88] furthermore was shown to be contradictory in itself ([ZU91]). We therefore did not focus on this formal approach which seems to be of limited practical use. Instead we tried to assess the metrics against their underlying semantics. A metric's semantic or meaning, however, may be viewed as equal to its metamodel, which is defined as follows:

metamodel of a metric ≡ minimal set of meta-classes and meta-associations appearing in the metric's definition

The second component of the metric metamodel assessment technique consists in elaborating or obtaining a reference model of the thing to be measured. In the context of software, i.e. models of systems, this reference model is a so-called metamodel. It is derived independently from any metric. We may then assess a metric with respect to four semantic criteria:

① The metric metamodel may be compared for completeness with the reference metamodel. In this context completeness means that every meta-object and meta-association should be included in the metric's definition, i.e. should also appear in the metric's metamodel. The inclusion of meta-constructs in a metric metamodel needs not be direct, i.e. there may exist rules that map associations A1 of one kind to associations A2 of another kind. We then only have to show that the metric measures A2. The same indirection may be applied to the meta-objects (= meta-classes).

② The metric metamodel is also assessed as to its time of availability within the software process. This criterion is especially important for predictive metrics which should be available as early as possible. Because the time of availability of a metric

entirely depends on the time it takes to elaborate its underlying objects, i.e. metamodel, we only need to consider the latter's behaviour with respect to this criterion.

③ The soundness of the metamodel is also assessable. We are especially concerned about language and modelling independence. Language independence is important for the comparability of values as well as for controlling heterogeneous projects. Very often systems are not modelled and implemented in a single language but with specialised languages for the database part (e.g. SQL), for the GUI part (e.g. Smalltalk), for large server transactions (e.g. COBOL) and for system management (e.g. some Shell-language of the operating system). Modelling independence is the same as programming language independence but for the higher software layers of requirements analysis and software design. Estimation biases and productivity measures will be inherently flawed when the metric's metamodel is not reasonably rigid.

④ Finally, the metric metamodel is also assessed as to the criterion of "non-fakeability". Productivity programs are reported [DM82] to have failed for this reason. An overly simplistic metamodel and counting procedure, like the LOC metric, may be obstructed by "clever" software engineers. They simply write code generators to simulate a high productivity.

We have seen that the procedure of deriving metrics based on metamodels (as outlined in 3.1.1) may also be used backwards: For a given metric the minimal domain model is constructed and then compared to an independently developed domain model of the metric's intention domain. For example the simplistic LOC metamodel may be compared to a comprehensive domain model of (object oriented) software. Refer to section 2.2 for an application of this assessment technique to various existing metrics.

3.1.3 Metamodel Mapping

Looking at the various ways, i.e. metamodels, of programming and modelling software, the search for a "unified" metric of size seems like an impossible task. Several parallel attempts however can also be observed, e.g. the unification attempt of the various software constructs as components and the modelling unification attempts by Rational [RA96] and the OPEN team [OPEN96].

We developed and made use of a technique called "metamodel mapping" to be able to reuse the same metric definition for several metamodels. The base component of this mapping technique is a tolerant and complete metamodel of the domain in question, i.e. the software artefacts in our case. The properties of tolerance and completeness are defined as follows:

1) A metamodel is tolerant when it does not require all its meta-classes and meta-associations to be actually used in a certain instance. The system metamodel as presented in 3.2.2 e.g. may also be used for conventional non-object-oriented software by simply omitting the use of class and feature concepts. Methods are interpreted as plain procedures in this example, i.e. as purely functional abstractions not belonging to a class. Metamodel tolerance is achieved by not enforcing 1:1 or 1:m cardinalities on the meta-associations but instead metamodelling them as optional c:1 and c:m cardinalities.

2) The second property is the completeness of a metamodel. In the example of software or system metamodels this means that every information of other metamodels may be contained. The semantic completeness is typically demonstrated by constructing a complete mapping of any source metamodel into the target metamodel. If this mapping is possible

without losing information, i.e. if it is reversible, the target metamodel is complete with respect to the source metamodel. For example a domain class from the domain analysis metamodel layer which is modelled without its attributes, may be mapped into a class of the System Metamodel SMM (cf. 3.2.2), which, of course, has attributes. We always use the SMM as the mapping target and all the metamodels we investigated could be mapped to it. We therefore assume - as a working hypothesis - that this metamodel is complete for the software or system domain. Note that the mappings should not be confused with implementations. For example a domain class may be implemented using a full-fledged technical class. The mapping, on the other side, is done only to the role of a class. A complete metamodel therefore has to offer a sufficiently rich set of roles in order to allow mappings from all possible other metamodels.

We make use of the metamodel properties of tolerance and completeness for reusing measure definitions: Every measure defined for a tolerant and complete metamodel is also definable for metamodels that are mapped into it. In the case of system descriptions the SMM is tolerant and complete, i.e. the metamodels of the higher level system descriptions may be mapped to it. In order to get a measure XY for layer i we first have to apply the metamodel mapping from layer i into the SMM and then apply the measure XY for the latter as shown in the following diagram:

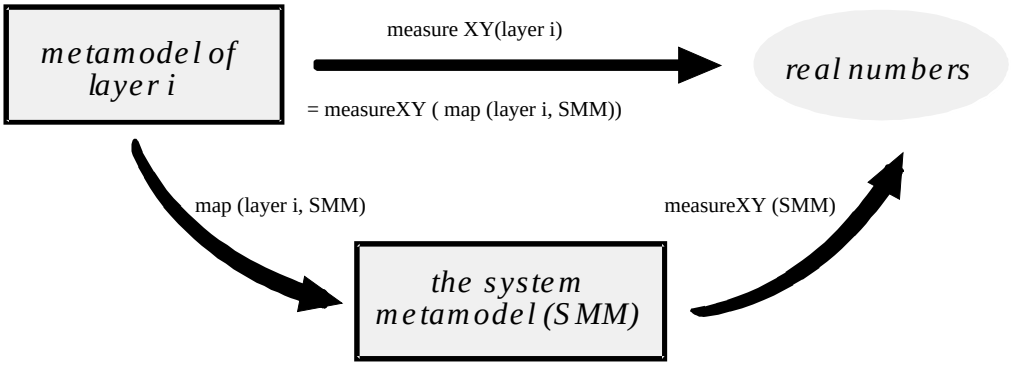

Figure 38: Measuring any SMM-defined Measure XY on System Description Layer i through Combination of the Mapping and Measurement Functions

To give an example for this technique we may look at the metamodel for preliminary system descriptions (cf. 3.2.7) and the newly defined measure System Meter (cf. 3.2.1). First, the System Meter is defined for the full-fledged SMM. The metamodel for preliminary system descriptions is then mapped into the SMM. The System Meters for a preliminary system model are thus measured by first mapping the model into SMM-objects and then applying the measurement rules. In the forthcoming section 3.2 we will first define the System Meter for the SMM (in 3.2.1 and 3.2.2) and then define it for all the other software engineering layers using the metamodel mapping technique (in 3.2.3-7). Each metamodel mapping is made explicit in a separate subsection.

3.2 New Measures and Metamodels for Software

This section documents our main constructive contribution to the domain of measurement and estimation of software and software processes. We propose a unified new measure of complexity and size, the System Meter, for a generic metamodel of system descriptions. Based on metamodels for other layers of software artefacts and on the metamodel mapping technique (cf. 3.1.3), we also defined the System Meter for more specific purposes. The

two earliest versions of the System Meter (for preliminary and domain analysis) were validated in an extensive field study (cf. chapter 4).

Layering of Software Artefacts (=Metamodel Object Types)

In order to understand the different layers of software used in our metamodelling they are summarised in the following table. Refer to 3.2.3-7 and 3.3.3 for more detailed information.

Layer#	*Name and Description*	*Metamodel Object Types*
1	***Preliminary Analysis*** Describe the rough scope of the future system and derive (rough) functional goals the future system has to fulfil. The preliminary study layer is comparable to OMG's strategic modelling layer [OMG92].	goals, data subject areas
2	***Domain Analysis*** Describe the logical system's objects as if there were no computers around (i.e. "the system's underlying physics") and describe its activities in an event driven way (use cases). Define what logical objects are used by the use cases in what way (creation, read, update, delete, and other function types) and refine some goals from the preliminary study into more precise requirements. This layer describes what OMG calls domain objects within the analysis modelling layer.	domain class, domain association, subsystem, use case, actor, non-functional requirement
3	***Application Analysis / Specification*** Define the border of the automated system within the logical system description. Describe the automated system's objects (= application classes) and the user's view of the system by specifying the models, i.e. the usage views of the object model in certain contexts (reports, windows). Try to identify and reuse types of specification triplets (model, view and controller types). This layer corresponds to the design modelling layer of OMG's reference model. The aspect of reuse is emphasised through the notion of reusable specification types.	models, model types, view types, controller types, application classes, application class elements, reused elements
4	***Design / Construction*** For each specification type define at least one design pattern (consisting of cooperating sets of technical class, feature and method specifications) that describes a suitable implementation strategy. Like the specification types the design patterns are candidates to be reused from a library. For each design pattern describe a fully functional sample implementation. Refine the application class model into a (tuned) technical model and eventually into a relational model for persistence. This layer corresponds to the implementation modelling layer of OMG's reference model.	design pattern, technical class, feature, method, formal parameter, table, column, reused elements
5	***Implementation / Replication*** For each client specification elaborate a tested and tuned implementation. As a basis for the implementation the design pattern corresponding to the specification's type is to be instantiated i.e. replicated. Tune the relational model and eventually implement server procedures. This layer corresponds to the construction and delivery layer of OMG's reference model.	object, class, feature, method, formal parameter, message, actual parameter, reused elements

Table 7: The 5 Layers of Software Artefacts

As a refinement to the traditional waterfall software process model (cf. 2.3.5) and layering of software (cf. 2.1.5) we split the requirements analysis layer into 3 distinct layers. This allows for rapid and prototyping based developments as well as sound estimates earlier. For each of the 3 analysis layers a separate prototyping cycle may be instantiated to support the stabilisation of the requirements.

General remarks on metamodelling software artefacts

In order to establish metamodels that are useful for software engineers we had to analyse what the software engineer's job is. Obviously a software engineer builds software systems. But what he ultimately does is writing text and/or drawing diagrams. So we would say that

he describes a system (as a synonym: he models a system). Therefore we state that he builds a system model rather than a system. Even when he is writing compilable code this is just an abstraction or model of the systems that are actually taking inputs and produce outputs, i.e. are executed on some machine. The results and items of software engineering are therefore system descriptions and the metamodels presented here reflect those.

The structure of the following subsections

Our newly proposed software metric, the System Meter (subsection 3.2.1) is presented in a similar manner to the presentation and analysis of the existing measures in section 2.2, i.e. 1) the metric's metamodel, 2) the metric's definition, 3) intended usages, 4) the formal metric assessment and 5) a summary. The derivative System Meter definitions (subsections 3.2.2-7) for the other layers may then be presented in an abbreviated scheme, i.e. 1) the metric's metamodel and 2) the metamodel consistency rules. Because we reuse the metric's definition, we can rely on the assessments in 3.2.1 for all the other variants of the System Meter.

3.2.1 The System Meter

Premises

As we have seen in section 2.2 the mainly used or recently proposed existing measures suffer from several drawbacks: they either are too simplistic (LOC), historically overloaded and database-tied (Function Points) or non-standardised and not size oriented (the C/K-Metrics, representative for virtually all the recently proposed object-oriented measures). In order to avoid the drawbacks observed among the discussed measures the following premises were adopted:

① The definition should be accomplished over a *sound metamodel* of software systems.

② The metamodel should be tolerant and therefore *generic* (cf. 3.1.3), i.e. encompassing both object oriented and conventional systems.

③ The metamodel should incorporate aspects of *reuse*.

④ No historically influenced parameters should occur in the measure's definition and it should be *language and modelling independent*.

⑤ The metamodel should be *complete* (cf. 3.1.3), i.e. encompass every kind of system description entities (e.g. classes. methods, objects, parameters, messages, ...) and their associations (e.g. class inherits from (super)classes, method belongs to class, method is implemented by messages, ...).

Additionally all of the 9 formal requirements elaborated in 2.1.5 were reflected in the metric's definition.

3.2.1.1 The System Metamodel (Part I)

A system description is viewed as a set of description objects. The notion of a description object is an abstraction for the more concrete items introduced in part II of the System Metamodel. The kinds[16] of those items (classes, methods, features, formal parameters, messages, object instances, actual parameters, as well as several subcategories) are not necessary to be known for understanding the System Meter definition. Refer to section 3.2.2 for a detailed description of the complete System Metamodel, i.e. the typical meta-

[16] Actually those various kinds (Greek: polyios morphe) are nothing but a polymorphic structure below the abstract concept of the description object. We therefore may claim that we applied object technology to software engineering itself.

types occurring in full-fledged object-oriented systems. The first, abstract part of this metamodel, however, has less facets and may be diagrammed as follows:

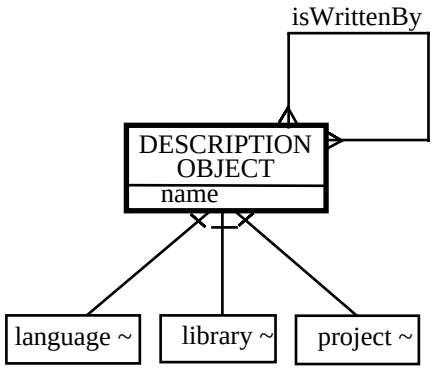

Figure 39: The System Metamodel (Part I)

At first sight, our metamodel may seem as simplistic as the LOC metamodel. Actually we intentionally chose a simple metamodel to achieve similar intuitiveness and formal behaviour as the LOC metric. In addition, however, to the LOC metamodel we introduced three enhancements:

① The description objects are linked. For each description object the set of other description objects that it depends on is identified. This association may also be viewed as a "description object writes description object" association with the "description object is written by description object" reverse association (diagrammed as the "isWrittenBy" edge). The cardinality of this association is mc:mc (cf. [CH76] for this notation), i.e. one description object may write several others and it may be written by several others. The most typical example is that of a variable X which is assigned possibly several times the values of literal constants, other variables or expressions:

Example of a description object `X` (C++):

```
{  int X = 0;          // creation i.e. the first write access, of X
   X = SOME_GLOBAL;    // update i.e. a subsequent write access of X
   }                   // deletion i.e. the last write access of X)
```

`X` viewed as a description object depends on the other description objects that appear in the writing messages, i.e. by the literal `0` and the global variable `SOME_GLOBAL`. We say `X` "is written by" the other objects. This notion is a generalisation of the idea of slices developed by Weiser [WE81] and recently used to define a quality metric of cohesion by [BI94].

② Furthermore, every description object has an external part, its name, and an internal part, its implementation. The name actually is optional, i.e. there also are anonymous objects (e.g. the elements of an array) that contribute to a system's description. The name, if present, serves as a means to reference a description object after its creation. A class, e.g., once created is referenced through its name when used for object instantiation or inheritance (or whatever else one can do to or with a class). The implementation part is equal to the isWrittenBy association described under point ① above.

③ Finally, every description object is categorised into being reused either 1) from the *language* system, i.e. from a predefined set of highly standardised description objects or 2) from some *library* system, i.e. from a set of less standardised components. The third category 3) are the newly established descriptions that are specific for the

project. Refer to the following subsection 3.2.1.2 for a more detailed discussion of our modelling of reuse.

Less formally diagrammed than in figure 39, the situation when focusing on a single description object may be depicted as:

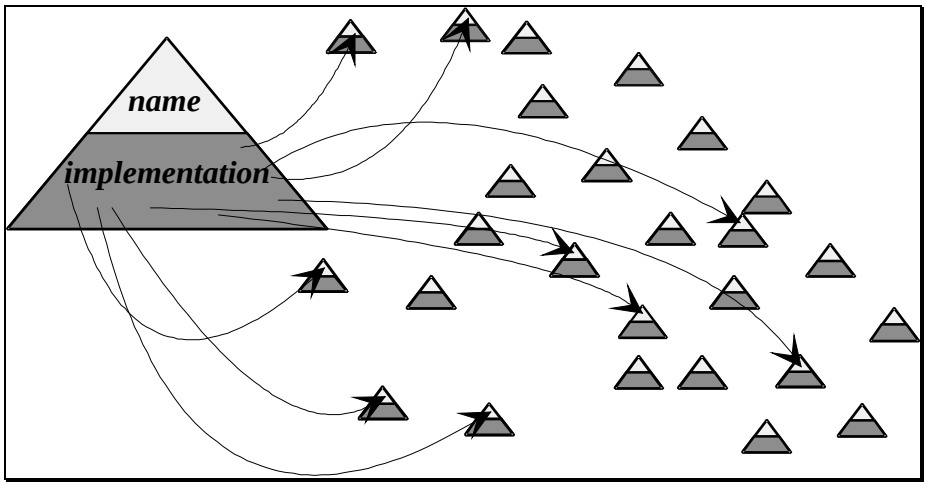

Figure 40: A Description Object together with other Description Objects

In this diagram the isWrittenBy associations of the object in focus to the subset of other objects which determine its implementation are denoted by arrows. We immediately see that this association is equivalent to a component's coupling (cf. [ME88], [YO79]). This implies that our new metric incorporates some parts of the quality aspect of coupling. Because, however, our focus lies on estimation mere sizing, we did not further elaborate towards assessing quality with the System Meter. Refer to section 5.2, the outlook to future work, for a more detailed discussion of this topic.

3.2.1.2 A Metamodel for Reuse

As may be seen in figure 39 every description object is categorised into "language", "library" and "project". This categorisation forms our metamodel of reuse. Again we honoured the KISS[17] principle in order to keep our new metric intuitively understandable and usable. The three categories are:

① Language description objects are distinguished from the library objects by the fact that they are highly standardised As an example we may list all the C++ language constructs, e.g.:

1) `class`, which is a (non-explicit) meta-class whose instances are the "normal" C++ classes

2) `int`, which is a (again non-explicit) class whose instances are objects

3) `while (<expr>) do <stmt> ;`, which is a decision/branch method

② The library objects are all components which are built on top of the language objects, but which are not (!) elaborated - in the sense of the isWrittenBy association, i.e. neither created, updated nor deleted - in the software process, i.e. project in consideration. Usually frameworks are considered an enhanced form of libraries that also allow the reuse of component cooperation (e.g. classes are statically and dynamically dependent on each other in specific ways). In practice the distinction of what is a library and what is a framework is fuzzy. Even seemingly flat libraries presume certain cooperation and interconnections between the data structures they

[17] Keep it Simple, Stupid

operate on, and - on the other hand - even the most complex framework encapsulates its functionality in methods, i.e. method signatures. Because our metamodel also includes the links and method signatures as description objects (cf. 3.2.2 for details) the gradually varying cooperation complexity of libraries and frameworks is captured.

③ Every other description object is viewed as being of category "project". Project object may depend in more or less complex ways on reused components as well as other project-specific components. Because the links - in the form of messages - from the newly built to the reused description objects are part of the isWrittenBy association of our metamodel we also encompass this kind of complexity.

The process and strategies of reuse

Reuse is considered to be one of the most promising aspects of object technology (OT) with respect to productivity and quality gains [SE89]. It is however not automatically achieved when applying OT. First, it should be a managed process as defined e.g. by the BIO software process (cf. 3.3.3). Second, it should be introduced not only on the technical layers of design and implementation but earlier at the requirements stages [HA95]. On the domain analysis layer we may reuse components in the form of so called Business Objects [FI96]. The latest point when one must incorporate reuse into a system is in the application analysis or specification layer. After a system's behaviour to the user, for instance the GUI behaviour, is specified against all previously defined - and therefore technically supported - standards, we may no longer profit from reuse in the later layers.

Categories of reuse

Reuse - applied on whatever layer - may furthermore be categorised into (according to [KA92]):

① verbatim reuse, i.e. reuse without change to the original component

② generic reuse, i.e. reuse by parametrised instantiations of classes (e.g. C++ templates)

③ leveraged reuse, i.e. reuse with modifications (e.g. through inheritance)

Orthogonal to those three categories we may further categorise reuse into

① direct, and

② indirect, i.e. without or with intermediate "wrapper" objects.

With respect to software processes all these kinds of reuse may, however, be reduced to being either verbatim or being no reuse at all. First, generic reuse will not change the instantiated template, i.e. is verbatim, whereas the instantiation is not reused but must be defined in a project-specific way. Second, reuse with modification through inheritance also does not change the superclass. The inheritance statement as well as the new subclass, on the other hand, are specifically described for the project, i.e. not reused. Furthermore, when modifying the code of a framework, the modified parts shift from the library to the project category. This shift can be accomplished with a very high-resolution granularity using the elements of our code layer metamodel (cf. 3.2.2), e.g. on a single message or even single actual parameter basis. The distinction of direct and indirect reuse is also only of technical relevance but becomes transparent in our metamodel. If the intermediate objects are developed within the software process in consideration they are not reused, i.e. of category "project", if not, they are reused verbatim, i.e. being of category "library".

One additional mode of reuse which is observable daily in practice, however, was not mentioned by Karunanithi and Bieman: the technique of copying and adapting code skeletons or entire code sequences. This simple yet powerful way of reuse is notoriously difficult to track because the source, i.e. the code parts copied from, is not identifiable ex-

post. Due to the high granularity of our metamodel - and therefore our new metric - we would conceptually be able to measure the modifications with respect to such templates. Practical means are however difficult to implement and were not elaborated in our work. They could be established on a method or class level by interpreting certain naming conventions, e.g. all classes beginning with "controller_" would be treated as modifications to a generic "controller" class template.

Summary of our reuse metamodelling

In summary we may claim that our at-first-view simplistic metamodel of reuse can capture virtually all grades of (verbatim) reuse due to the high granularity of the metamodel entities. Reuse on the requirements layers is also supported through the metamodel mapping technique. The category of "library" components for requirements items must however be interpreted in the more broad sense that they 1) may be actually reused on the requirements layer or 2) are entirely supported by existing technical design and implementation constructs. We evaluated our metamodel and metrics of reuse for a sample set of framework and subsequent reuse software processes in our field study (cf. 4.2.2) and observed a promising practical behaviour of our ideas.

3.2.1.3 The System Meter Definition, Intended Usages and Assessment

The definition for singular description objects

The counting of System Meters for a single object is made up of two parts: the external part and the internal part:

The external part, i.e. the externally visible complexity of a description object is made up of the name of the object, provided the object is not anonymous:

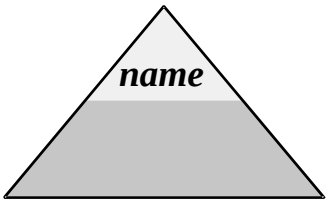

Figure 41: The External Complexity of a Description Object

The complexity of a name in turn is - intuitively speaking - correlated to the difficulty of remembering it. This in turn has to do with the number of "new elements" within the name compared to a set of "known elements". We may thus formally define this idea of external size as:

externalSize (o) ≡ isNotAnonymous (o) ? numberOfTokens (name (o)) : 1

where the operator <bool-expr> ? <expr1> : <expr2> returns <expr1> if <bool-expr> is true and <expr2> otherwise, and the function numberOfTokens returns the number of new tokens in the name (but at least 1). This accomplished based on a lexical name analysis and comparison to the tokens already encountered in the system description sequence as follows:

① The "starter token set" is established as all language reserved words. As a consequence of this rule every language object gets an externalSize of 1.

② Before counting the tokens in a name, the "new token set" is defined as the strings in the name that are separated by the following rules: 1) by case (e.g. theSetOfStrings is tokenised into 'the' 'set' 'of' 'strings'), 2) by separator characters (_, -, @, !, :, etc.), those characters are eliminated but if more than one kind of separator is used within one name, the extra separator kinds are counted as tokens, 3) by groups of n numbers

(e.g. groups of a maximum 3 numbers are viewed as one token) and 4) by changing from numbers to alphabetic characters (e.g. Eta01 → 'eta' '01').

③ Now the "new token set" is compared to the "starter token set". The number of new tokens + 1 is returned as the functions value.

④ Finally, after counting the tokens, the "new token set" is joined into the "starter token set".

More straightforward counting rules - like for example counting the number of characters in a name - were not considered because they are too easily "faked" (cf. criterion 4 in 3.1.3) and they do not reflect the psychological complexity of names [LE94].

The internal part, on the other hand, is made up of the recursive isWrittenBy association and may be measured using the external size:

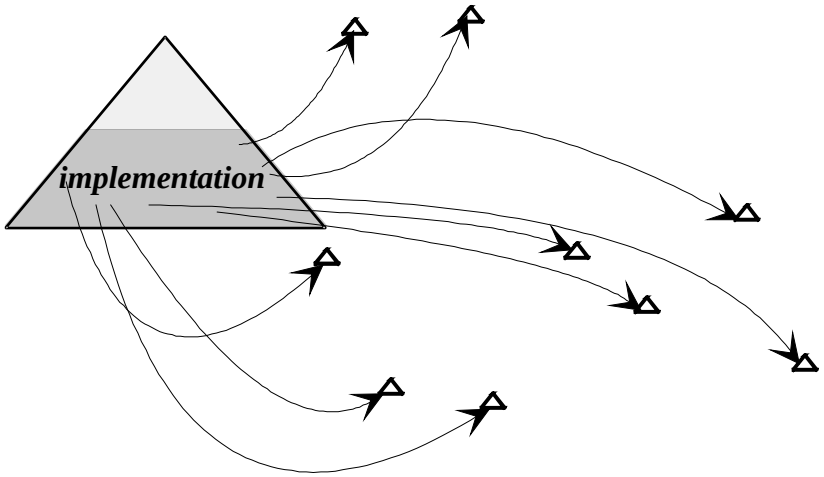

Figure 42: The Internal Complexity of a Description Object

It is intuitively "bigger" the more objects participate in a description object's definition. A variable, for example, may be considered more complex the more often it is assigned a value. This, besides, is one reason for the good engineering practice of declaring description objects, such as variables, as constant. The formula for this second System Meter part thus is:

$$\textbf{internalSize}\ (o) \equiv \Sigma_{x \in o.\text{isWrittenBy}()}\ \text{externalSize}\ (x)$$

The total size of a singular object is the sum of those two sizes:

$$\textbf{Size}\ (o) \equiv \text{externalSize}\ (o) + \text{internalSize}\ (o)$$

Informally we may state that the more complex a description object's signature is and the more the object depends on other description objects the "bigger" it is.

The definition for sets of description objects

When measuring sets of objects, i.e. whole systems, the aspect of reuse becomes relevant because whole systems usually consist of reused and newly developed parts.

The categorisation into "language", "library" and "project" objects is used in the definition of the size of a system as follows:

$$\textbf{Size}\ (\text{System}) \equiv \Sigma_{o \text{ is library object}}\ \text{externalSize}\ (o) + \Sigma_{o \text{ is project object}}\ \text{Size}\ (o)$$

As we can see, the language objects are not counted at all. The library objects are counted with respect to external complexity only, and just the project specific objects are counted in full. This corresponds to the fact that the (programming) language is a usually stable and well understood "commodity", the library - just the interface is considered - is less stable,

more specific and therefore more complex, and finally the newly developed objects are least stable, their interface as well as implementation undergo all development micro-cycles and therefore generate the most effort. The System Meter thus - also making use of the fine granularity of the description object notion- can measure virtually every gradation of reuse.

Intended Usages

The generic concepts underlying the System Meter allow this measure of size to capture the idea of description complexity. Due to the metamodel mapping technique (cf. 3.1.3) it is measurable for virtually any language and modelling technique and any layer. The main usage which was also validated in a field study (cf. chapter 4) is *development effort estimation* for software projects. In order to support this usage, which requires a very early metric application, we had to use the System Meter for the layers of preliminary analysis and domain analysis. Immediately tied to estimation is the second usage for *productivity tracking and analysis* which was also validated. The positive results of this practical validation and the good formal attributes (cf. the assessment below) of the new metric encouraged us also to use it for *deriving quality metrics*. This usage, however, is only theoretically done and the practical validation, which will probably have some impact on the theoretically derived formulae, is part of our ongoing research in this area (cf. 5.2).

Formal Assessment

① Counting: The System Meter is a simple count and therefore its scale is absolute and every arithmetic and statistical operation may be applied safely.

② Single items: The System Meter may be measured for single description objects.

③ Sets of items: The System Meter is also measurable for sets of description objects.

④ Elemental items: The System Meter is defined for the most elemental items in software engineering, i.e. for object instances, messages and actual parameters (cf. 3.2.2), as well as for more high level concepts like classes, instance and class variables, methods and formal parameters.

⑤ Dimensionally safe: The System Meter is dimensionally safe because in its definition only one kind of items, i.e. name tokens, are counted and summed up.

⑥ Dimensionally uniform: The System Meter is dimensionally uniform because all the various kinds of software engineering items are mapped to or subtypes of the notion of description object for which the metric is defined.

⑦ Sequencing: The System Meter is sensitive to the sequence within system descriptions because the token counts are based on the set of already known tokens. This set may be different if an item is introduced at another place within a system description thus yielding a different rating.

⑧ Naming: The System Meter also takes into account for the naming of system description objects by yielding the number of (new) tokens as the external size part.

⑨ Triangle's condition: If we use a token based difference algorithm to yield the two sets of subtracted and added tokens for any two pieces of code x and y and sum up the System Meter values for the two sets to obtain d (x, y), then the triangle's condition of $d(x, z) \leq d(x, y) + d(y, z)$ is always fulfilled. In case the direct difference operation between x and z should yield bigger sets, we can consecutively apply the difference sets of x to y and then y to z and add their sizes to achieve equality. It is however important to note that the System Meter values for the singular values should be determined on the whole descriptions, i.e. before the difference operator is applied. If we do not proceed like that, the omission of a single token X could enhance the rating of the token count of the following tokens. Thus,

the overall count for the set without X could be bigger than the value of the same description including X. While this metric behaviour makes sense for semantic omissions (it is, e.g., more complex to understand a C++ class implementation file without the header) this is not true in the context of token based difference operators.

3.2.1.4 Summary of the Concepts of the new System Meter metric

The newly proposed System Meter is based on the notion of a so called description object, which forms the abstract base class for richer part II of the System Metamodel (cf. 3.2.2.10). It captures the idea of coupling in the internal size and the idea of signature complexity in the external size. It furthermore differentiates between reused and newly developed system elements. The System Meter, formally assessed, has - among other formal attributes - an absolute scale measure and is dimensionally sound and uniform. Due to the genericity of the metamodel virtually every possible language may be mapped onto this notion. Even natural language constructs follow a similar pattern: words have a certain signature in order to be recognisable and - once introduced - are defined using other words. The natural language definition process, however, is in practice informal and subjective, i.e. the same words do not have the same meaning to everybody and definitions may change over time. This behaviour of natural language elements, which basically is analogous to our metamodel of system descriptions, gives us some additional confidence in the soundness and applicability of our approach.

3.2.2 The System Meter for Object-Oriented Implemented Systems (Code)

3.2.2.1 Overview of the Metamodel for Object-Oriented Code

We introduce this metamodel in seven informal steps, each corresponding to one metamodel entity. The comments furthermore try to give reason for those meta entities by pointing to some of the historical metrics and ideas of software:

① A system description at code level is, at first glance, a set of *messages* (= sentences or statements). This view of programs immediately led to the first known metric of size used in software: the lines of code (= number of sentences). Actually this metric made quite a bit of sense when one line of code equalled one "sentence" (as in languages like Assembler, early BASIC/FORTRAN/COBOL, etc.). In virtually any "modern" context though it is inadequate.

② Diving deeper into the structure of system descriptions, we find that every message consists of an explicit or implicit sender, an explicit or implicit receiver, a selector/signature, maybe some parameters and maybe a return value. We therefore not only have messages, but also *actual parameters* referenced through their name, i.e. a token, in different roles. With this extension of our metamodel, we also reach a new metric of size, Halstead's number of tokens [HA77].

③ The message's method selector or signature, then, references a polymorphic (generic) or fixed *method* of the receiver. Because a method has to be ready to accept actual parameters, *formal parameters* are also introduced. This enhancement of the metamodel was reflected in another Halstead metric, the number of unique operands, i.e. method signatures. The following special situation has to be mentioned here: as formal parameters may also be of type "message", because our metamodel treats messages like objects. This is typically the case in calls to flow control methods like "`if` <expr> `then` <statement1> `else` <statement2> `endif`" or the infamous "`GOTO` <statement-reference>" statement, where the <statement>-placeholders stand for such message parameters. Because calls to this kind of methods are of enhanced expressive power (and also of enhanced complexity), there was an extra metric

invented for them: the cyclomatic complexity [MC76]. This metric, though, ignored the metamodel parts described in steps 1 and 2 (as well as all the subsequent ones), which all contribute to a system description's complexity or size.

④ If we look into `C++` code e.g., we encounter "sentences" like `int i;`. This is a typical message (which is itself a special form of description object (cf. part ① above)) that creates a new instance or object, in our example named `i`. Each *instantiated object* which is created once and from then on has its identity, is therefore also a *description object*, actually its most basic form. This fact was reflected in still another Halstead metric, the unique operator metric. Halstead also defined derived metrics out of the token count and unique count metrics, the (in)famous volume and effort metrics (infamous because they violate some formal metrics criteria). Halstead's so called "Software Science Metrics" have however shown to be best in predicting maintainability (cf. [OM92]), maybe because - among the well known metrics - they cover the most of the system metamodel thus described (i.e. the parts described in ①, ②, ③ and ④).

⑤ We also observe the formal parameters in 4 variants: they either *create*, *read*, *update* or *delete* the actual parameters. These ideas - but coming from the area of database applications - were the backbone of Albrecht's [AL79] Function Point metric. However, the metamodel parts described in 4 and 7 were ignored, part 3 limited to a fixed predefined set of methods (create, read, update, delete) and part 6 (classes) restricted to flat "attribute containers" (file types).

⑥ Every description object is of a certain *class*, that defines both: 1) the set of publicly understandable selectors (= the object's type) and/or locally available selectors and 2) the selector's implementation as *features* (direct attributes) or methods. Moreover, classes are structured into two hierarchies: 1) the type hierarchy, where subtypes publicly understand at least all the selectors of their supertypes, and 2) the inheritance hierarchy, where subclasses inherit all the implementations from their superclasses. The most evolved of the new metrics dealing with those object oriented concepts were proposed by Chidamber/Kemerer [CHI94]. They are, though, not designed to serve as metrics of size but as metrics of quality. Therefore they ignore many of the metamodel concepts described in the other steps.

⑦ In the seventh and last step we introduce the fact that the metamodel may also be considered recursively. The metamodel entities just introduced are themselves nothing but objects, that may appear in messages as actual parameters, that are created, read/used, modified, etc. Thus description objects may also be metaclasses, metaclass-methods and meta-metaclasses, etc.; the field is open. This is a situation of freedom we almost uniquely observe in natural languages. Most conventional and modern programming languages only allow very restricted access to meta-objects (for example C++'s recent real-time type information (RTTI) extension and C++-templates which are parametrisable with classes).

As a summary, the set of the metamodel entities may be shown in the following type hierarchy diagram:

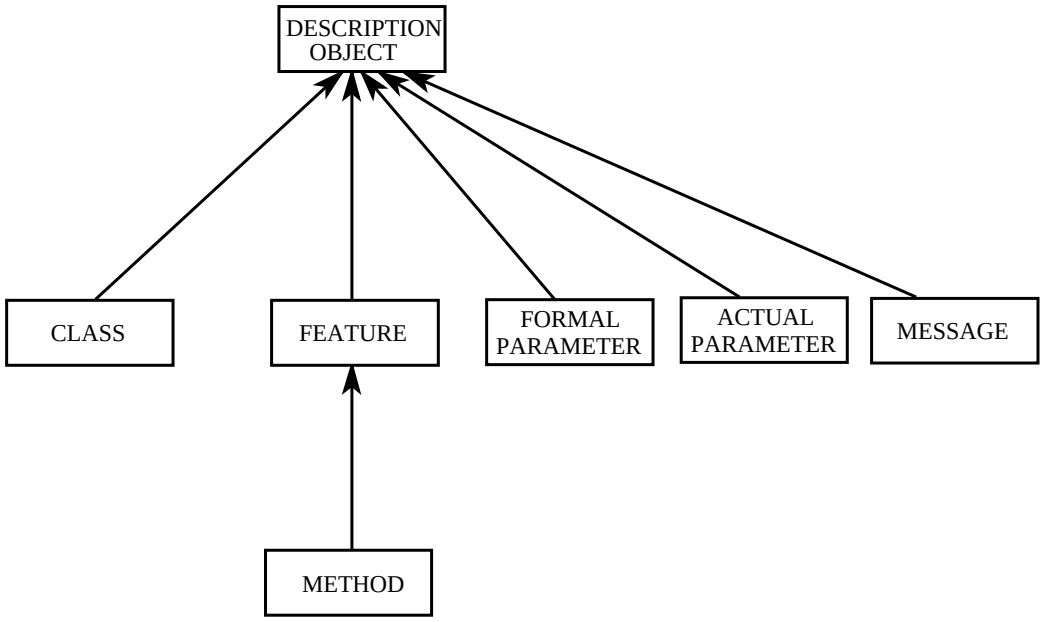

Figure 43: The SMM Type Hierarchy

Detailed Presentation of the Metamodel for Object-Oriented Code

We will subsequently present in detail several metamodels of object oriented systems that are rooted on three known metamodels for object oriented systems: 1) The OPRR-Model [WEL89] [SM91], 2) the WW-Metamodel [WA90] and the CDIF Standard [CDIF91]. The most generic and detailed metamodel thus proposed is the one that represents the structure of an implemented object oriented system presented in this subsection. This metamodel is called the System Metamodel (SMM). Even though it is based on the three metamodels just cited, it is quite extensively enhanced. The summarised enhancements in the SMM are:

- tracking of the usage and scope of description objects, i.e. their coupling
- instances, parameters and nested messages, i.e. enhanced granularity
- the distinction of project, reused library and reused language components
- modularity aspects (the ability to package system components into meaningful configurations/subsystems)
- template relationships to cope with a "copy and modify" development style

The entities of the SMM are presented in the following subsections one by one in partial metamodels. Beyond the diagrammed models, we also comment the meta-classes and associations and formulate consistency rules that are sequentially numbered and especially marked with the superscript text "consistency rule <number>". Those rules appear in the order of the explanation flow.

The complete graphically diagrammed SMM, which is the union of all the partial metamodels, is given in the last subsection 3.2.2.10 together with a list of all the consistency rules. There are, however, no further comments there.

3.2.2.2 Instantiated Objects / Description Objects

The system metamodel with the description object in focus and the surrounding meta entities diagrammed in dotted boxes looks like this (cf. 2.1.7 for the notational details):

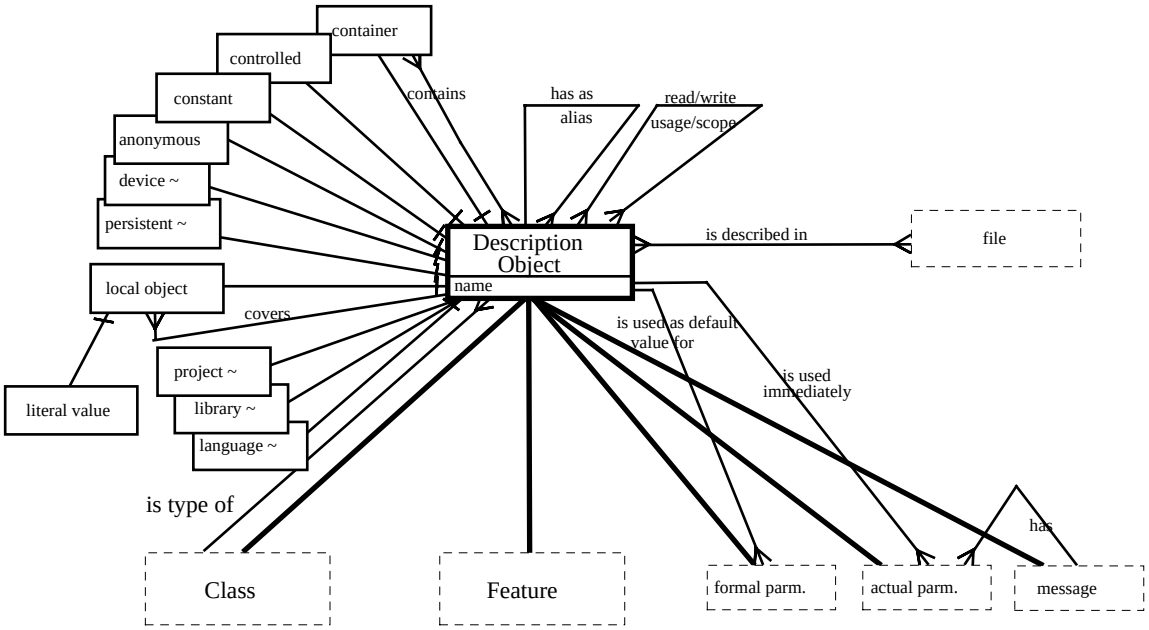

Figure 44: Partial SMM - Description Object

First of all, the class "description object" is the base class of the SMM type hierarchy. It is, however, not an abstract base class, but represent the concept of an object instance. Description objects, instantiations of "description object", thus may take on the following subforms:

① classes (or meta-classes, etc.)

② features (i.e. instance and class variables) and - as we will see later - an important subtype of features: methods

③ formal parameters

④ messages

⑤ actual parameters

We will encounter each of these subtypes in the following subsections, but will now first discuss the properties and peculiarities of the description object.

Instantiated objects consist of a name part, i.e. some identifying expression by which the object can be referenced, and an implementation part, i.e. an associated set of other description objects through which the object is defined.

Note that not every object needs a name part, i.e. there exist anonymous objects. Also the implementation part needs only - for language and library objects - to be optionally known. One of the two parts however must obviously be known for each object, or else it would not exist. The name part is also called the external part of an object, whereas the implementation part is referred to as the internal part.

A system description as a whole is nothing else than a sequence of usages of objects within messages. We might also say that a system description - at first sight - is a sequence of messages. As a side effect of those messages (which themselves are description objects) new other description objects (e.g. a class or a method) are created, read, modified and deleted.

Object Scope

The metamodel presented here encompasses the concept of scope in the following way: Basically all objects are considered to be globally accessible (i.e. their names may be referenced in all messages following the creation message). In cases where this is not true,

i.e. some objects are in a local name space, a special relationship is assumed that is called the "Object covers local Objects" relationship. A method for example is considered to cover 1) all messages that implement it and 2) all its local variables. To give other examples: a C++ block statement (which is considered to be a message) covers all its sub-statements (i.e. also messages), a class covers all its non-public features, etc. This hierarchical covering relationship allows us to define a partial order among objects, i.e. for two objects appearing in the same name space it is clear which is more local than the other (if they do not have the same covering object). We will make use of this partial order below.

Writing and using description objects

The previously introduced association isWrittenBy between description objects, which is central to the System Meter definition, may now be more closely explained: This recursive relationship is constructed as follows: The *immediate implementation* of an object X is defined as the set of messages wherein X appears as an actual parameter and is written in some form i.e. created, modified or deleted. Now, two other kinds of sets may be defined: ① In the messages of the immediate object description of X other objects may also be used (usually but not necessarily in the read form). Those objects are added to the immediate implementation to form the *direct object implementation*. ② The union of the direct object implementation sets of all the objects of equal or more localised scope in the direct implementation set of X then finally yields the *complete direct object implementation* set, which is the association we are interested in. It is also called the description of description objects. This set is related to the notion of program slice introduced by Weiser [WE81] and data slice introduced recently by Bieman and Ott [BI94]. However it is the sum of both kinds of slice i.e. statements (= messages) and data (= all other description objects) are incorporated. Furthermore it differs from the slice notion in that it is truncated by the rule that only the equally or more localised objects are recursively considered whereas the slices contain all the direct and indirect dependencies.

We may make further comments on the 4 major forms of object usage:

① *Creation Usage*

Creation or create usage is the very first usage of any object in a system. At the object's creation point the name - if it has one - of the object and optionally some first implementation of the object is defined.

This usage form is also referred to as "initialisation" or "definition" of an object. Usually some object's creation takes place with a single message[18] to some "language class", e.g. to create a C++ int named `i` with the initial value of `1` you write: `int i = 1;`. For more complex objects like classes, methods or functions the object creation may span several messages and submessages. An exemplary function with the name `func` that prints an integer value passed as a parameter on some standard output device and if it's bigger than 0 and also prints the square before would be defined in C++ with the sequence `void func (const int i) { if (i > 0) {int j = i*i; cout << j << "\n";} cout << i; }`.

Sometimes the creation of the name part (referred to as the "first declaration" or "introduction" of an object within the system description) and the creation of the

[18] We use the term "message" for things that others might actually name "message" but still others also name "function call", "procedure call", "statement", etc.. The aspect of messages that they need not be understandable by the receiver and that they must be dynamically bound to the receiver is not considered to be definitional. Thus the term message is to be understood in a more generic form in this text.

implementation part (referred to as the object's "initialisation") are split into two description parts that need not be in immediate sequence.

To give an example: the function `func` may be introduced in a separate file called a "header file" with the sequence `void func (const int);` and may be initialised in another file called an "implementation file" with the same description text as in a full creation (cf. end of last paragraph). In some special cases even the initialisation may be split into different parts[19] (but this does not affect the concept). This splitting has to be allowed, because system objects sometimes must be used before they can be implemented ("forward references"). It is also useful for disallowing access to the implementation of certain objects (for management purposes, copyright reasons or whatever).

② *Read Usage*

Read usage is the most common form of object usage and defined as any occurrence of some object's name in place of an actual parameter. Read usages (i.e. references to some previously created objects) may be observed throughout any system description. To give some examples: our C++ `int i` may be read in the message `j = i + 1;` and the C++ function `void func (const int)` may be read by calling it as done in the message `func (i);`.

Objects (in addition to their introduction declaration) may also be "redeclared" several times, i.e. their name part is just stated again. This is - for some object types - a legal but effectless form of a read usage that sometimes makes sense for code management purposes.

③ *Update Usage*

Update usage is the most dangerous but also the most powerful form of usage. Update uses are descriptions that change (or describe a change of) the implementation part of an object. The most common example is the assignment to a variable like our C++ int i, e.g. in i = 2;. In Smalltalk (not in C++) it also possible to change the implementation of a method by simply rewriting it (or by changing it in more subtle ways by calling text interpretation methods of some kernel classes).

④ *Deletion Usage*

Deletion usage is the very last usage in the life of objects. The deletion usage terminates the existence of an object. For many objects the deletion usage is an implicit usage, i.e. the corresponding message has not to be described. This is true for example for all C++ local automatic objects. Deletion that is actually described is only common for so called "controlled" objects, i.e. objects that may be created and deleted on request. Again Smalltalkers might be slightly confused because all Smalltalk objects are "controlled" in the sense that you can at any time choose to delete anything (also classes). However the deletion of description objects is not necessary for all objects to occur. For example some global objects, like most of the classes, are never deleted.

These 4 forms of description object usage may be further typed into 1) write usages (creation, update and deletion) and 2) read usages (read). In analogy to the isWrittenBy

[19] To give an example: The default values for formal parameters in C++-functions need not all be initialised (i.e. defined) in one statement. Their definition may be done one by one in separate messages (however the sequence of definitions must be "from the last parameter to first" and no defined default value can be changed in some message later in the system description, thus C++-functions are all constant objects (cf. the "comments on constant objects" paragraph below).

association which is defined through the write usages, we may thus also define the isReadBy association. Also note that the form of usage is recognised by the category of the formal parameter (cf. the subsection on formal parameters below) to which the actual parameter corresponds.

In summary we may state that a system description e.g. code is nothing else than a sequence of usages of description objects. Thus an object's description is primarily the set of its write usages.

Aliased description objects

Every description object may have several aliases (i.e. other names for the same implementation). The aliases "inherit" everything from the object they stand for. Their isWrittenBy associations are therefore almost the same as the ones of the object they stand for[consistency rule 1]. The only isWrittenBy associations exclusively held by the alias are derived through its creation messages[consistency rule 2].

Aliases may stand for the object they alias but not vice versa. The classical examples for aliases are the C/C++ typedefs. A typedef X, though, not only is an alias for the described type T but T also is an alias for X. So X can stand everywhere T stands and T can stand where X stands. This mutuality does not hold for other alias types like C++'s "enums" (special forms of ints) which are aliases for ints but ints are not aliases for enumerator types. To summarise: "normal" aliases (in the sense of C++-typedefs) are in our metamodel reflected by an object (of course) and two aliasing associations; one for each direction in which the aliasing works.

Container objects

Objects are - as we already know - composed of two parts: their name and their implementation set i.e. the isWrittenBy association. Whereas the name (if there is one) is always a string, the implementation set for container objects may be viewed as being composed of two subsets: 1) the contained objects[consistency rule 3] and 2) the other objects. We briefly discuss the subobjects contained within an object:

If the container is elementary (e.g. a pointer or array) and no device object (e.g. an input or output stream) is in its implementation or read set, then its contained objects are considered to be anonymous objects[consistency rule 4]. If however a plain description object has a device object in its implementation or read set then it is first made a container object[consistency rule 5] and each byte needed to store the object's value (determined by its class) is considered an anonymous contained object[consistency rule 6]. This increases an object's size significantly. This intended increase is rooted in viewing device reads to be equivalent to coding at random, i.e. each character (or byte) being a variable entity of its own.

Furthermore the following object types always contain the following other description objects: a method contains its list of implementing messages plus the objects it's the scope object of[consistency rule 7], any message contains its submessages (= messages occurring as actual parameters) plus the objects of which it is the scope object[consistency rule 8], a class contains its set of features[consistency rule 9], and any object X contains locally named instances of all the non method and non class feature classes of the class of X[consistency rule 10]. E.g. a Date object contains 3 anonymous Number objects if the Date class has 3 Number features.

The description object's type association:

Every object has a class as its type. Our metamodel even supports the notion that an object may have more than one class as its types. Conflicting multiple types are treated optimistically i.e. conflicts either are assumed not to occur or being prohibited by some checking mechanism before a system is ultimately described. We treat multiple classes as

the union of all their features. For simplicity we will still speak of an object's class in the following text. Depending on an object's type, i.e. its associated class, the following restriction may be stated: The messages in which some object X may immediately occur as an actual parameter is restricted by the corresponding formal parameter's type, i.e. X's type must be equal to or be a subtype of that formal type[consistency rule 11]. As a consequence the occurrence of an object in the role of a receiver is restricted to messages to the object's type's methods.

Objects as default values

As we have already seen, an object is primarily used as an actual parameter in certain messages. Another and slightly exotic usage form of an object is its use as a default value for a formal parameter of a method. For default value usage, only constant objects - usually literals - are allowed. Because the assignment of an object o to a formal parameter fparm in the role of a default value can only be accomplished in a message that changes fparm, it is assured that fparm's isWrittenBy association contains o. Conversely, o's isReadBy association contains fparm.

Controlled and automatic objects

The life of a description object is usually automatic in the sense that at some point it is created then it is maybe updated several times until it reaches its deletion point, usually when it gets out of syntactic scope. Some objects never get out of syntactic scope (the global objects) but may also considered as being under automatic life control. However, there are still other description objects that are under so called "controlled extent". The life or extent of an object is the set of messages between (inclusively) the creation and deletion messages, be they implicit (as for automatic objects) or explicit. Whereas an object's scope usually is equal to its life, the scope of a controlled object X starts immediately after the creation message of the topmost supertype of X's class and has no end point[consistency rule 12]. This rule has its reasons in the fact that a controlled object may be transferred (through pointers) up to this point in a system description. Automatic objects may be "converted" into controlled objects by applying an address or referencing method to them (cf. subsection on methods).

Furthermore, the controlled objects are always anonymous and need a special kind of associated container object that references them. That special kind of object is called a pointer, a container or a reference. The pointer itself is usually an automatic object, but in some occasions it may also be controlled which means that the corresponding reference object will be a pointer to a pointer - a construct which may even be recursed thus yielding pointers to pointers to pointers, etc. One aspect of any controlled object X is, that as soon as X's pointer object is passed as an actual parameter the corresponding formal parameter also becomes a pointer object to X[consistency rule 13]. We may even state that any pointer object X written by some other pointer object Y automatically also becomes the pointer object to all the objects Y points to[consistency rule 14]. Thus a single controlled object is generally referenced by numerous pointers. The good news is that these mechanisms are very efficient in handling objects without having to deal with their complexity. The bad news is that the read and write scopes of the controlled objects are the unions of the scopes of all their corresponding pointer objects[consistency rule 15]. Moreover, when dereferencing (cf. subsection on methods for dereferencing methods) a pointer, this is modelled as a simultaneous usage of all the controlled objects that the pointer object points to[consistency rule 16].

Thus almost everywhere within a system description (also in system parts logically described before the creation point of some controlled object) from the creation point of the

controlled object's highest supertype class any object may use and modify controlled objects which is not true for automatic objects. The category of persistent objects, finally, is nothing but an abbreviation for controlled and global objects, i.e. even if an object should be local or automatic, as soon as it is categorised as persistent this implies it is global and controlled (with all further implications).

Constant and variable objects

Any description object may either be constant or variable. The "normal" classification of an object is being variable, i.e. the read scope equals the write scope. An example for a variable in C++ is:

```
char c = '0'; c = '1'; c++;
```

There are, however, situations where the system description is more stable when an object is asserted not to change, i.e. to be constant. In terms of scope this simply means for a constant object C that the update scope of C (i.e. the set of other objects that might change C) is empty[consistency rule 17]. An example in C++ is

```
const char cc = '0'; cc = '1'; // this is not permitted
```

but also:

```
class C {...};
```
[20].

This second example demonstrates the genericity of the SMM. Not only "normal" instances are treated as description objects but also classes. It is therefore a natural idea that classes can be treated as variables (as Smalltalk does). The implicit C++ convention of constant classes is just viewed as special case, which is admittedly reasonable in many situations but not in all.

Note that a constant's create and delete scope are in general not empty, which means that also the write scope is not empty. It is however heavily reduced. For constant non composite library and language description objects only the name part is of interest.

Named and anonymous objects

As already stated objects consist of two parts, 1) the name part and 2) the implementation part. By denoting the name part somewhere in the system description after the object's creation point that object is referenced as an actual parameter. This is also the most common form of usage of some object after its point of creation within the system description. In general we may state that an (automatic) object must have a name to have a life beyond its creation point at all.

In a system description there exists a mass of objects that do not have names, e.g. literal values never have a name or the messages in general have no names, unless they are labelled to support direct jumps, the infamous GOTOs. Thus, anonymous automatic objects are created and deleted at one single point of system description[consistency rule 18]. The remarks just made on anonymous objects are not true for controlled anonymous objects. Those objects have one or more special other objects that point to them and those pointers may be used as a name for the anonymous controlled objects. For controlled objects to be used immediately, the pointer object must be passed to a so-called dereferencing method[consistency

[20] For C++-ers it is so clear that classes are constant it is not even denoted. Smalltalkers however know that classes are not a priori constant i.e. method features may be added or removed throughout the whole system description; even the whole class may be destroyed on request. The difference is that Smalltalk classes are "normal" Smalltalk objects, i.e. they are variable and controlled, whereas C++ classes are "special" C++ objects (implicitly automatic, constant and de-facto only existent at run-time). They may not be treated like normal C++ objects.

rule 19. In some cases the dereferencing message (i.e. the mechanism to access the controlled object instead of the pointer object) is implicit in some others it is not. To give an example: In Smalltalk the assignment message to some variable (Smalltalk variables are always pointers) does no implicit dereferencing, i.e. the assignment changes the value of the pointer (variable), not of the object the variable points to. For example the sequence

```
i := View new:. i := 1.
```

does the following: the (local) variable `i` first points to a new `View` object, then the `View` object is not assigned the value of `1`, but instead the variable `i` points to a constant integer object of `1`. The `View` object still exists unchanged. However, the sequence

```
i := View new:. i destroy:.
```

will also first attach `i` to a new `View` object but then not `i` is destroyed but the `View` object. The variable `i` still exists with a value of `Nil`. Thus in the non-assignment message `i destroy:. i` was implicitly dereferenced.

Local and global objects

The difference between local and global objects may be described in terms of what is called the description life or syntactical scope which was introduced above. The syntactical scope of the global objects are all the messages following the object's creation, if the global object is automaticconsistency rule 20. For the controlled objects, which are always global, rule 12 applies to find their scope. The life of some local object X, on the other hand, is restricted to the messages that are directly or indirectly contained in the enclosing scope object of Xconsistency rule 21, which usually is a method or a class. Finally there is a highly populated set of local objects of a special kind: the literal values. The enclosing scope object of literal objects is the message they are defined inconsistency rule 22, which means that the life of a literal object is reduced to a single object, its enclosing message. Literal objects can therefore always be treated as constantsconsistency rule 23. The latter rule is derived from the fact that any literal's creation and deletion point is equal, i.e. that it does not have an update scope.

Language, library and project objects and file containment

As already introduced in the System Meter's metamodel (cf. 3.2.1) every description object can be categorised as belonging to the language, the library or project. Usually this categorisation is taken over from a file based categorisation, i.e. there are library files, language files and project specific files. The description objects contained in some file will adopt the file's category. In more delicate situations - usually during maintenance - where only parts of files are modified, we need however the categorisation at object granularity.

3.2.2.3 Classes

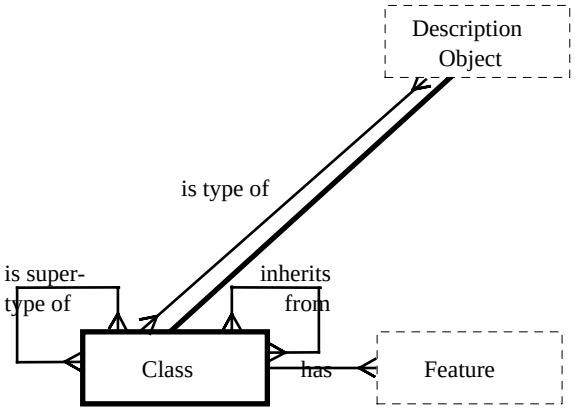

Figure 45: Partial SMM - Class

The class is a very central concept in object oriented system descriptions. A class object, e.g. a class Stack, describes a set of objects, e.g. the Stacks S1, S2, ... that are said to have the class object as their type. The implementation of a class is accomplished via the description of a set of so called features. Features are described in the next section below. The objects for which the class object is the type are called instances of that class. The class itself is also an instance of some class of some higher level of abstraction. Such a class's class is called a meta-class. If no explicit meta-class is given for a concrete class, at least the SMM-entity "class" is the meta-class. However, this meta-class, and also all the other SMM-entities, belong to the description objects of language category and are therefore never counted.

Every single object in a system description is created by sending a construction message to its class' constructor method. This implies that an object's class is always in its isWrittenBy set[consistency rule 24].

There is a sub-/supertype relationship that relates classes with common features; one might also say common behaviour interface. A class is considered a direct subtype of some other class when it offers at least the same features to its (possible) clients. We then may introduce the term direct supertype in the following way: a class Base is a direct supertype of another class Derived when Derived is a direct subtype of Base. When subtyping is defined like that, a single class may have many subtypes and (nota bene) may also have many supertypes[21]. Also note that the sub-/supertyping relationship is not necessarily an inheritance relationship and vice versa - in practice however subtyping usually goes hand in hand with inheritance[22]. Non private, i.e. "normal", inheritance is such an elegant way to ensure the subtype condition that many people have serious doubts that it makes sense to separate the two concepts of subtyping and inheritance at all. However there is a nice example where subtyping is desirable but not inheritance: The real numbers are subtypes of rational numbers and these of integers, but the implementations may only awkwardly be accomplished through inheritance. So we may conclude, that it is poorly chosen language design when C++ offers inheritance without subtyping but not subtyping without inheritance, and also when Smalltalk offers no separation at all.

A class A together with all its (not necessarily direct) subtype classes S1, S2, ... is called a polymorphic structure. Whereas in C++ all polymorphic features of such a structure have to be explicitly present, i.e. at least declared and marked as pure virtual in the base class A, this is not true[23] for Smalltalk. Therefore sometimes the C++-polymorphism is called "narrow" or "controlled" whereas the Smalltalk way of polymorphism is called "wild" or "real" (depending on who is giving the adjectives). The author thinks it is good style to explicitly introduce polymorphic features at some top level class, but he also thinks that in order to oblige programmers to introduce those features (as C++ does) multiple subtyping is required (as C++ offers). So Smalltalk with single inheritance is right not to oblige its

[21] This many to many relationship is known as "multiple inheritance" to some people (even though the author would prefer the term "multiple subtyping"). In general it is unlikely for a class of things to be the subtype of only one supertype (i.e. more abstract concept). To give an example: The set (or class) of BMW 850ci cars is a subtype of BMW cars but also of sports cars and also of expensive cars and 12cyl. cars and ...

[22] In Smalltalk this is even syntactically a must whereas in C++ you can choose to "inherit privately" which inherits but does not subtype.

[23] In Smalltalk there exists kind of an "attractor" method for every possible message sent to an object (this method typically rises some debug window with a message that reads like "object of class XY doesn't understand message "sonofabitch"). So every interface feature may also be considered to be "introduced" in Smalltalk.

programmers to introduce all features at the "correct" level. If it would, the top classes (like Object) would be even more crowded with features. C++ offers the possibility of creating more than one (partially) disjoint polymorphic structure whereas Smalltalk essentially consists of exactly one huge polymorphic structure under the class Object. This philosophy allows polymorphism everywhere.

We have separated the two concepts of inheritance and subtypes in our system metamodel to be able to model a generic system view. Thus the following rules apply: If a class Sup is a supertype of class Sub, Sub has at least the same global features as Sup[consistency rule 25]. If a class Derived inherits from another class Base then all inheritable (cf. next section) features of Base become inherited features of Derived[consistency rule 26]. Besides those two direct associations between classes there are also various indirectly associated classes:

1) those associated via the feature's association and then - because features are also objects - the object's type association,

2) those associated via the feature's association, the feature to method subtype association, the method to formal parameter association and then - again - the object's type association

and

3) those associated, first again via methods, to messages and actual parameters to objects and then to types.

Meta-classes, finally, are classes like any other except that their instances reflect classes (or metaclasses) themselves. Special forms of meta-classes are also "templates", i.e. types of classes like `const` or `*` in C++ (constant classes and pointer classes). The instances of those "templates" are created "on the fly" i.e. when an instance of that template instance is needed. In some cases the instantiation of the "templates" is done through special statements (C++: `typedefs` of complicated derived types especially for function prototypes). In that case the `typedef` is not considered a mere aliasing mechanism (as explained in 3.2.2.2) but also an instantiation message to one or many template meta-classes.

3.2.2.4 Features

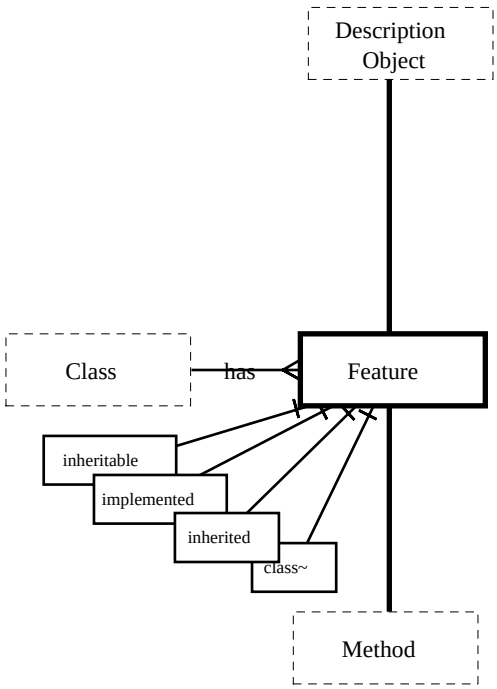

Figure 46: Partial SMM - Feature

Features are the components of classes regardless of their dynamic or static nature, i.e. regardless of being methods or variables, i.e. just data holders. The methods however (described in the following sections) form one main subtype of features. Methods are considered like calculated data values. From a class client's point of view it is not relevant if a feature of a class is stored (i.e. implemented through a real data feature) or calculated (i.e. implemented through a method feature). The methods though give a new dimension to features as they may take parameters thus behaving differently in different contexts and may produce side effects by sending messages to other objects.

In addition to that subtyping of features there are the following four non-exclusive categories:

① the *class features* (in contrast to the instance features)

② the *inherited features* (in contrast to newly implemented features)

③ the *implemented features* (in contrast to those without implementing message body)

④ the *inheritable features* (in contrast to the non inheritable ones)

The *class features* are those features the class as a whole owns. There is one kind of feature (methods to be precise) that is always a class feature: constructor methods. Those are methods that create a new object of their class. This may be compared to a factory producing a product, where the factory corresponds to the class, the assembly line to the constructor method and the product to the instantiated object. To hold on to that analogy: the class features correspond to the things a factory can do whereas the instance features correspond to the product's features. For the process of instantiation the following rule applies: only plain normal data features become contained sub-objects of the object newly instantiated. This rule was already stated before (consistency rule 10).

An *inherited feature* is a feature any, i.e. direct or indirect, superclass of the considered class has introduced and is "allowed" to be inherited i.e. inheritable, e.g. in C++ an inheritable feature must be non private, a distinction irrelevant to Smalltalk programming). An inherited feature basically has the same signature, i.e. name and formal parameters, and attributes as the feature it inherits from and - most important - defers its implementation to the base class' feature implementation (which in turn may also be deferred). There are, however, exceptions to that rule: 1) a local feature can be made global and vice versa, 2) an implemented method can stay unimplemented (thus inheriting becomes effective) and vice versa, and 3) an inheritable feature can be made non inheritable. Maybe there is some confusion about inherited features in the sense that one might believe inherited features are the same objects as their base features. This is not true in our metamodel. An inherited feature even if it stays unimplemented and inherits all from "above" is another feature than its base feature.

An *implemented feature* is a feature that has a genuine implementation in the feature's class, i.e. not only in super- and/or subclasses. Every non-inherited data feature must have an implementation and cannot be re-implemented in subclasses. So for data features the "implemented" category directly depends on the "inherited" category[24] The "implemented" category therefore is only of interest for method features. Every combination together with the "inherited" category may occur when considering methods: 1) there are inherited and implemented methods, these are called the reimplemented or redefined methods, 2) there

[24] The dependency for data features may be formulated as two implications:

if	inherited	then	not implemented
if	not inherited	then	implemented

are inherited but non implemented methods, these are the plainly inherited methods, 3) there are non inherited and implemented methods, these are newly introduced methods with an implementation and finally 4) there are the non inherited and non implemented methods, these are methods just introduced to define abstract polymorphic structures[25].

Finally we have the category of *inheritable features*. To define this category negatively: the non inheritable features are restricted in their scope exactly to the class they belong to instead of being accessible also by the subclasses, i.e. subclass features. In C++ therefore the category of non inheritable features is indicated by a restriction of scope, i.e. with the `private` keyword. In Smalltalk it is not possible to have non inheritables.

3.2.2.5 Methods

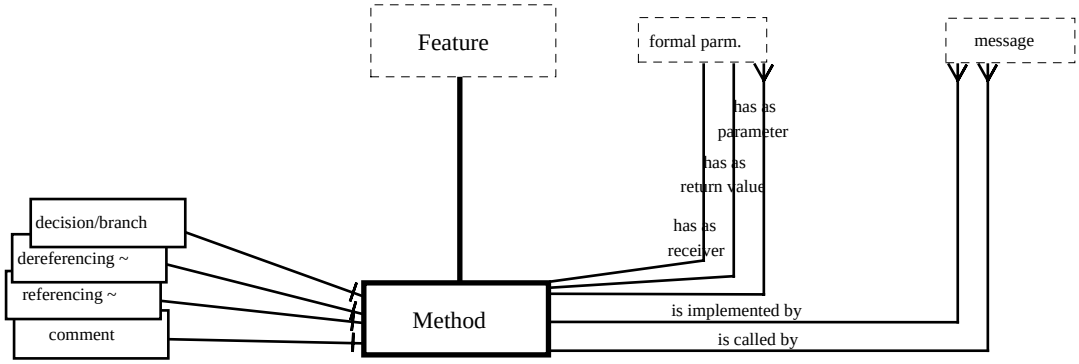

Figure 47: Partial SMM - Method

Every method has its corresponding ordered set of formal parameters. In particular two formal parameters are always implicitly or explicitly present: the formal receiver and the formal return value. A method, if implemented at all, is implemented by a sequence of messages. On the other hand a method may be called several times in implementations of some other or the same method. If a method is used in its own implementation we call that a recursion. Methods are one very important subtype of features. A method's name not only consists of its name as it appears when calling the method but also of the class names of the corresponding formal parameters[consistency rule 27]. Finally there are two subforms of methods that require special treatment:

① The dereferencing methods operate on container or pointer objects. These methods, when called in a message, return the contents of a container (or pointer) object[consistency rule 28]. Thus, they are important to track the scopes and usages of controlled objects.

② The decision/branch methods are methods that take messages as arguments and perform some conditional execution on those messages. These methods imply the addition of every actual parameter received together with a message block to the immediate read set of the message block's direct and indirect submessages[consistency rule 29]. Branch messages with backward branches (`goto` <label before goto-message>, <end of loop blocks>), i.e. loops, increase the isWrittenBy associations even more in order to minimise loops in practice. In addition to rule 28 the following rule also applies: Each direct and indirect submessage of the skipped message block is added to the backward branch's message isWrittenBy association[consistency rule 30]. This yields

[25]In C++ these methods are called pure virtual. In Smalltalk it is not possible to have such methods, but it is common to have "dummy" methods in any class X representing abstract concepts. Usually these methods call some dialogue that pops up and says something like "This feature has to be defined in subclasses of X" thus enforcing the redefinition of the method.

an explosion in size, depending exponentially on the number of messages skipped, because as its net consequence each message skipped backwards is added to the parameter list of each message skipped. With the same intention in mind as is behind McCabe's cyclomatic complexity [MC76], we therefore also especially account for branches in our metamodel and - as a consequence - metric.

3.2.2.6 Formal parameters

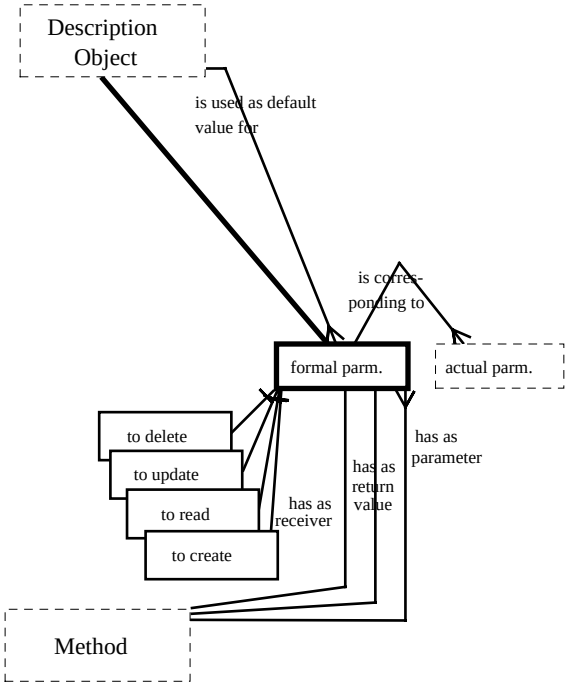

Figure 48: Partial SMM - Formal Parameter

Formal parameters are parts of a method's external part, i.e. the methods "name" (C++-ers call this the method's signature). To be precise, it is the formal parameter classes, i.e. the classes linked through the type association, that make up a methods external part, i.e. the name or interface. The formal parameters themselves are of minor interest. They may be named in order to be accessed within a method's implementation. In a message, i.e. a call to a certain method X, there are actual parameters supplied which must correspond to the formal parameters defined for X. These parameters can be stated explicitly or implicitly. The implicit way of providing actual parameters in a message only works for formal parameters with so called default values. If a default value for a formal parameter is present and no actual parameter is given then the default value is used instead[consistency rule 31]. Default arguments are (of course) objects (usually literals or constant globals). There are special parameters that stand for a list of parameters that are not limited in their number. Those parameters are of a container class of some type. Usually the last of the parameters is of such a kind. The most general form is a container of objects parameter, which allows actual parameters of any type to occur. In C++ this most general form is supported with the "ellipsis" construct (i.e. the three dots … that may appear at the end of a formal parameter list). This special form had to be chosen in C++ because there is no general type like Smalltalk's Object[26]. In Smalltalk it is no problem to give a method a dynamic array of

[26] An alternative to the ellipsis in C++ would be to pass an array of void* (i.e. a void**). This however enforces that the actual parameters must be put into an array before being passed to a method instead of being directly passed in the calling message. This void** alternative is a analogy to Smalltalk's way of dealing with varying numbers of parameters. Our metamodel covers both possibilities.

objects as a parameter. A syntactical form to provide a varying number of parameters is however not provided (because it is not needed). In Smalltalk - as a minor drawback - an object array must be built up before calling the method.

Formal parameters appear in four subforms: 1) they may create an object, 2) read an object, 3) update an object or 4) delete an object passed to them. Update parameters may change only the non constant objects and the delete parameters may destroy only the controlled or local objects (note that also constant objects may be deleted!). The effects of those parameter categories are directly modelled in the isWrittenBy association and indirectly in many of the consistency rules. The idea to make use of these 4 basic operations was adopted from Albrecht's Function Point metric and metamodel [AL79].

3.2.2.7 Messages

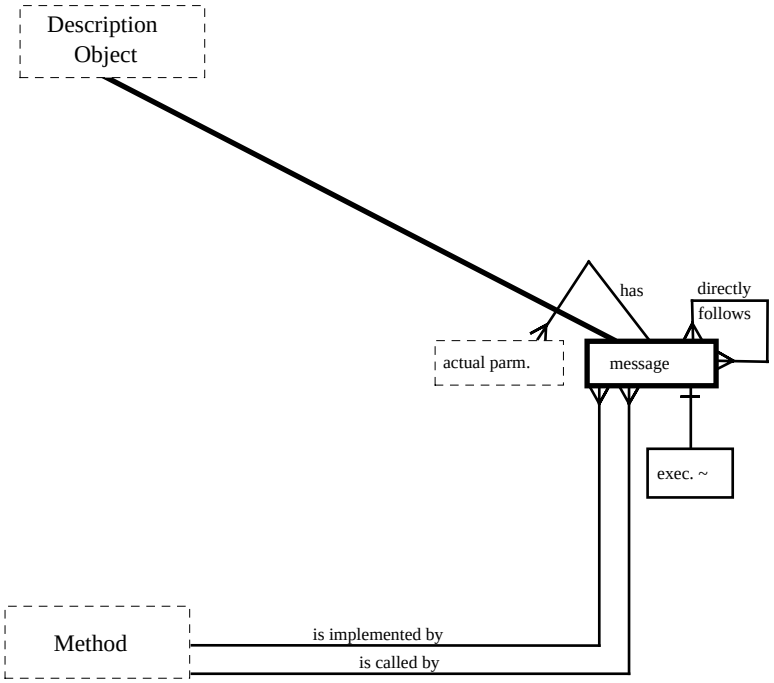

Figure 49: Partial SMM - Message

Messages are the second to last object subtype we consider in our metamodel of object oriented system descriptions at implementation level. As already stated the primary reason for messages is to provide an implementation for methods. Those messages are said to be "compiled" into the method's or function's or procedure's body. There are, however, a lot of other messages that are directly executed to build up the system objects, e.g. the C++-statement `class C { ... };` is not a message compiled into a method's body but a message to the implicit and compiler-built-in constructor of the C++-language element `class` (a language meta-class to speak in our metamodel's terms). This message is said to be compile-time executed or, simply, executed in contrast to the normal, i.e. compiled messages. Besides the already stated rule 7 which enforces all compiled messages to be contained in their method, we do treat the two kinds of messages differently. The distinction may however be reflected in some future metrics of quality. The view of a system description as a set of executed and compiled messages is very similar to one that FORTH-programmers know very well [FORTH78]. A FORTH machine - when reading FORTH source, which is analogous to "normal" compilation - is either in interpreting or compiling mode. The compiling mode is active only within word definitions, i.e. function or method definitions. Our metamodel sees systems described in any language like that. Also the view of a system as a growing set of objects - starting from language objects over

library objects to project specific objects - is adopted from FORTH's philosophy of growing vocabularies, i.e. sets of words, to form systems.

It is obvious that messages, i.e. statements of a system description appear in a certain order. The external part of the System Meter is sensitive to that fact. The order of compiled messages is very relevant to system correctness, quality and robustness as well as system description stableness and quality with respect to the executed messages. The basic ordering is given by the object or file dependencies, e.g. all the messages in <iostream.hxx> are stated before those in "main.cxx" to give a C++ example. Within each file the messages are already ordered because it is simply not possible to write more than one character at a file's elementary position. The metamodel's relationship that models this ordering is a many to many relationship thus allowing parallel system descriptions, an aspect maybe of interest for future research.

3.2.2.8 Actual Parameters

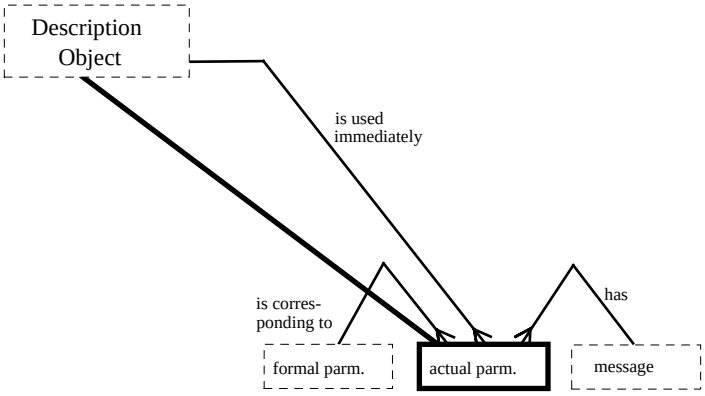

Figure 50: Partial SMM - Actual Parameter

The actual parameter is the last metamodel entity we comment. It is also the least important entity. Its main purpose is to serve as a reference holder to the calling message, the formal parameter and the referenced object. Actual parameters are always anonymous, automatic and local[consistency rule 32]. Their life is therefore restricted to a single message. Due to the incorporation of the actual parameter we can model even the slightest differences in system descriptions, thus assuring the usefulness of our newly proposed metric for maintenance and modification purposes.

3.2.2.9 Systems and Configuration Containers

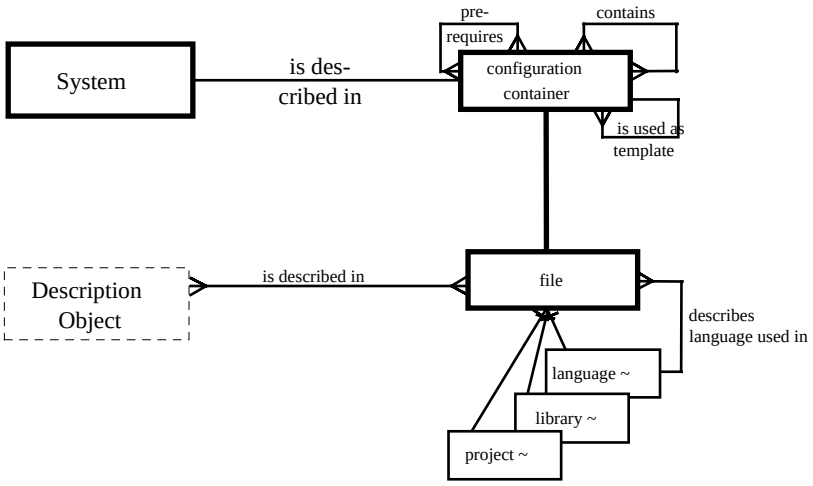

Figure 51: Partial SMM - System and Subsystems

A system or configuration is essentially a set of objects. If we were interested in a metamodel of systems we could leave that as it is. But, we are metamodelling system descriptions where it matters in which sequence and in what packages or modules the system objects are described. In our metamodel we therefore do not directly "link" the system to its objects but state that a system is described by a set of so called "configuration containers" (= packages). Actually one configuration container might describe objects of more than one system, but for simplification we will consider situations when only one system is in focus. This is of no harm because if two or more systems have to be considered simultaneously they can easily be joined into one logical system by joining their configuration containers into one "super" configuration container.

A configuration container (CC) is a notion for a thing real world people might call a "makefile", "file", "application definition file", "configuration", "module", "version", "subsystem" or whatever. However, it is a logical abstraction of those sorts of things. A CC may contain several other CCs, and in turn a CC may be contained in more than one "top" CC. For example, a network of configuration containers is possible to model several decompositions of systems. To give some examples for different decompositions: you can have one for modelling the layering in "model", "view", "control", one for application subsystems like "customer information", "order entry", "billing system", still another for language subsystems "C++", "SQL", "csh-Scripts" and still another in processor subsystems like "Client-System" and "Server-System". As already stated a system description depends on the sequence of the description elements. Between CCs one of those dependencies is modelled: the fact that the contents of one CC in order to be understandable must be preceded by several other CCs. To give an example: In order to be able to understand the C++ description

```
main ()
     { cout << "Hello world"; }
```

contained in a CC called "main.cxx" the descriptions in CC "iostream.hxx" are needed. This is something Smalltalkers might not understand, because they always have the complete system at hand. But even in a Smalltalker's system a subclass can't be created before the superclass exists - and that's what dependencies are all about. The importance of prerequisites seems not to be as big for Smalltalkers as for compiler-dependent developers but the fact of prerequisites per se exists anyway, so they are included in our metamodel.

Another association between configuration containers is also reflected here, it is the template association, which might also be defined on single objects, especially the more complex types of methods and classes. This association identifies for configuration containers the base CC from which it was originally copied from. The identification of this association is not trivial, because copying is normally not protocolled or notified on CCs. A normal systems developer would simply copy a CC which appears to be near the anticipated solution and modify what needs to be modified. The developer is not forced to indicate from where the code template was copied from. The system description, however, is essentially only the difference between the template source and the modified version. Due to the fine granularity of the object metatypes our metamodel and metric is sensitive to even the slightest modifications provided the template relationship may be reconstructed. We would count the modifications to the project objects, and the template to the library or - in rare cases - the language objects. A template identification based on naming schemes may be most practical even though we did not further investigate in this direction. The topic will become more important when we focus on empirical validation at the implementation layer.

CCs are further typed into logical ones, the base meta-class CC, and physical files, a derived meta-class. This typing reflects the fact that certain CCs do not directly contain object descriptions while others, the files, do. All comments on CCs are valid for files, too. Additionally the following comments apply: Files may be parts of CCs but may themselves never contain other CCs. This is the definitional difference between CCs and files. Furthermore files have exactly one language attached to them in which the objects they contain are written in. CCs may thus define heterogenously described system parts (i.e. system parts described in more than one language) whereas files are homogenous. Files are also - like the description objects - categorised into the three kinds of origin: language, library and project. Finally there is the most important aspect of files: the fact that they contain descriptions of several objects. A single object may be described in several files which must then be of the same language. The object descriptions must be identical in signature, i.e. name, or else we would consider the two descriptions or description parts as belonging to two different objects. Even though there may be many objects in language and library files only the objects actually used by project objects are considered[consistency rule 33]. The file categorisation is furthermore used as a default for the object categorisation of the contained objects.

3.2.2.10 The complete System Metamodel (SMM) - Part II

We may now merge all the diagram parts into one complete diagram which reflects the essentials of the System Metamodel (SMM):

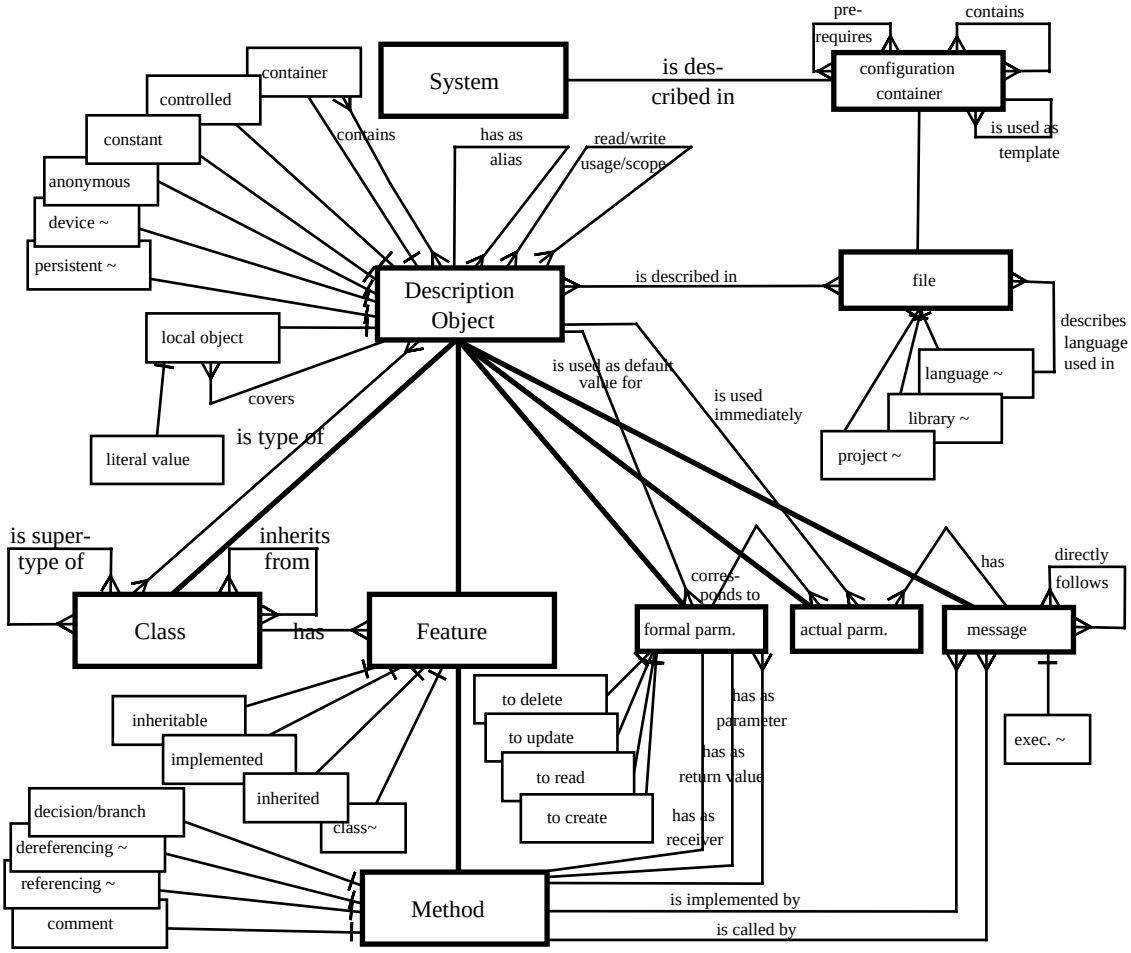

Figure 52: The Complete System Metamodel (SMM) - Part II

This diagram seems rather complex but we used it as a handy reference when discussing the metamodel. We also used it frequently when working on our measurement

instrumentation. The metamodel entities found more or less direct representations in our C++ implementation of the system description analyser (cf. the detailed comments in 4.2 and appendix D).

The consistency rules are listed as follows:

#	*SMM Consistency Rule*
1.	The isWrittenBy association of aliases is directly forwarded to the object they stand for.
2.	The only isWrittenBy associations of an alias are derived through its creation message.
3.	The implementation set of container objects is composed of at least the contained objects.
4.	If no device object is in the implementation or read set of a container, then its contained objects are anonymous objects.
5.	If a description object has a device object in its implementation or read set then it is assured to be a container object.
6.	Within a container having a device object in its implementation or read set, each byte needed to store a contained object's value is itself considered an anonymous contained object.
7.	A method contains its list of implementing messages plus the objects of which it is the scope object.
8.	A message contains its submessages (= messages occurring as actual parameters) plus the objects of which it is the scope object.
9.	A class contains its set of features.
10.	An object X contains locally named instances of all the non method and non class feature classes of the class of X.
11.	The messages in which some object X may immediately occur as an actual parameter is restricted by the corresponding formal parameter's type, i.e. X's type must be equal to or be a subtype of that formal type.
12.	The scope of a controlled object X starts immediately after the creation message of the topmost supertype of X's class and has no end point.
13.	One aspect of any controlled object X is, that as soon as X's pointer object is passed as an actual parameter the corresponding formal parameter also becomes a pointer object to X consistency rule 13
14.	A pointer object X written by some other pointer object Y automatically also becomes the pointer object to all the objects Y points to.
15.	The read and write scopes of the controlled objects are the unions of the scopes of all their corresponding pointer objects.
16.	When dereferencing a pointer, this is modelled as a simultaneous usage of all the controlled objects that the pointer object points to.
17.	For a constant object the update scope of is empty.
18.	Anonymous automatic objects are created and deleted at one single point of system description.
19.	Controlled objects may only be used immediately by passing their pointer objects to a dereferencing method.
20.	The syntactic scope of global automatic objects are all the messages following the object's creation.
21.	The life of a local object X is restricted to the messages that are directly or indirectly contained in the enclosing scope object of X.
22.	The enclosing scope object of a literal object is the message it is defined in.
23.	Literal objects are always constants.
24.	An object's class is always in its isWrittenBy set.
25.	If a class Super is a supertype of class Sub, Sub has at least the same global features as Super.
26.	If a class Derived inherits from another class Base then all inheritable features of Base become inherited features of Derived.
27.	A method's name not only consists of its name as it appears when calling the method but also of the

	ordered sequence of the class names of the corresponding formal parameters.
28.	Dereferencing methods, when called, return the contents of a container object.
29.	Branch/decision methods add any actual parameter received together with a message block to the immediate read set of the message block's direct and indirect submessages.
30.	Each direct and indirect submessage of the skipped message block of a backward branch (GOTO) is added to the backward branch's isWrittenBy association.
31.	If a default value for a formal parameter is defined and no actual parameter is given then the default value is used instead.
32.	Actual parameters are always anonymous, automatic and local.
33.	Only the language and library objects actually used by project objects are considered to belong to the system description.

Table 8: The SMM Consistency Rules

3.2.3 The System Meter for Technical Design / Construction Results (CON-SM)

Metamodel of technical design artefacts

A constructional description of a system in general consists of the same object types as the final implementation (e.g. classes, methods, ...). It is the level of detail, completeness and perfection that is reduced within construction to "implementation patterns" for each specification type (cf. 3.2.5). An implementation pattern may be viewed as a prototypical but well-documented and well-tested implementation for a kind of specified problem. Implementation patterns are candidates to be reused. They usually encompass small subsets of classes or methods, sometimes even only sequences of variable instantiations, usages and certain message sequences. The problem they solve may vary from generic problems (for example, how to implement rational numbers in general or in a specific environment like Smalltalk) to very system specific ones (for example, how to generate the registration code for registered documents). Some very generic implementation patterns which are successfully reused several times may evolve into design patterns (as described by [GA94]). The technical design activity may profit from this pattern based strategy in several ways:

① effort is focused on a few kinds of problems rather than on many problems that differ only slightly, thus, the design is reduced,

② eventually patterns from earlier projects may be reused,

and

③ the newly developed patterns are more likely to be reused within the same or subsequent projects.

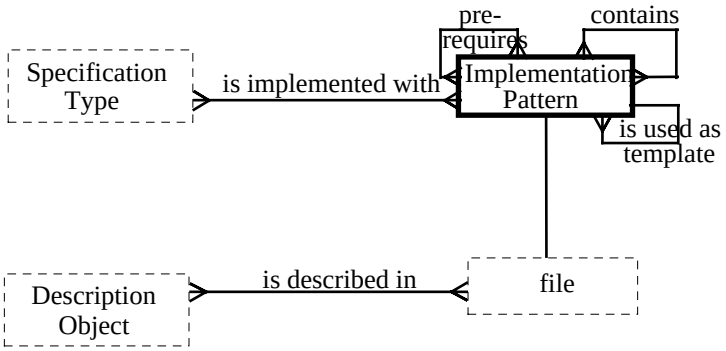

Figure 53: Metamodel of Technical Design Artefacts

The technical design thus is mainly a set of implementation patterns, each of them either reused or newly developed. The implementation patterns serve as templates in the

implementation. An implementation pattern itself is a set of cooperating description objects. An implementation pattern thus can be a small setting of two or three classes that communicate through the protocols of a couple of methods or it can be a complete or partial framework consisting of dozens and maybe hundreds of classes and methods. Implementation patterns furthermore are not restricted to abstractions like classes or methods but may also encompass certain object instances, messages and even specifically placed actual parameters.

Metamodel mapping into the SMM

The notion of an implementation pattern is mapped into the notion of a configuration container (CC) in the SMM. The other meta entities are entities of the SMM itself and therefore need not be mapped.

3.2.4 The System Meter for Relational Database Schemes (RDB-SM)

Metamodel of a relational database scheme

For modelling persistent classes, i.e. classes that may store and retrieve permanent instances in some way, very often relational descriptions (cf. [CH76], [DA83]) are used. This metamodel is viewed as a submodel of the technical design metamodel. It is not necessary when object oriented databases are involved. Then the simple categorisation of objects into persistent and non-persistent, i.e. transient objects, suffices.

Furthermore, very often relational schemes are the only stable and documented artefacts for existing systems. They therefore have a special importance in assessing existing systems towards re-engineering or porting them to new platforms. The ultimately sound basis for quantitative measurements will always be 1) the code and 2) the relational scheme, provided it exists, i.e. for database applications using relational databases. Because the relational scheme is usually smaller, less scattered and its description language - usually SQL - syntactically easy, it is more convenient for measurements than code. In order to support measurements on this non-object-oriented artefact, we additionally included this metamodel here.

The metamodel diagram is as follows:

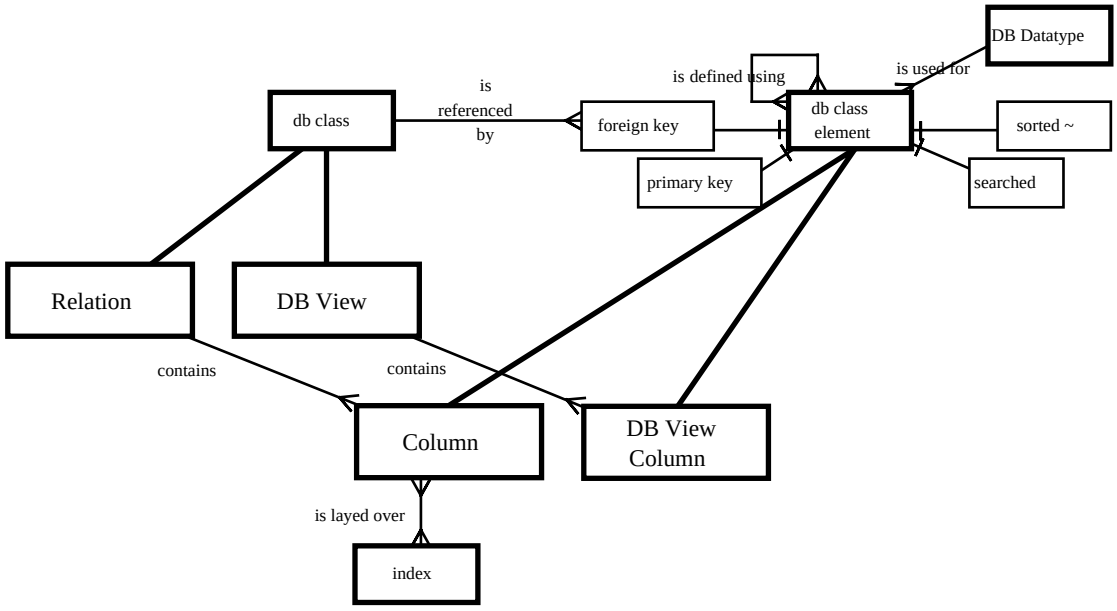

Figure 54: Metamodel of a Relational Database Scheme

This metamodel represents the structure of a relational model. We comment it with respect to the application analysis model, i.e. from the view of a forward development process. Application classes are primarily transformed into relations, i.e. tables, or views. Application methods, e.g. `create` in the class `customer` may however also be transformed into stored procedures, e.g. `customer_create`. This metamodel though does not explicitly support procedures or triggers, which may be viewed as automatically called procedures. These procedural constructs must be modelled as a normal implementation language using the metamodel entities given in 3.2.2. Data class elements from the application analysis layer may be transformed into relation columns, i.e. persistent data elements, or into view columns. The two application class element categories of identifying and referential elements are matched into the relational constructs of primary and foreign keys. The information of sorting and searching of class elements is taken into the relational metamodel, too, as well as the identification upon which table columns indexes are laid. Finally, predefined or user definable database data types are identified and the columns are typed accordingly.

Metamodel mapping into the SMM

This mapping is not as trivial as the technical design mapping in the previous subsection. We explain it in several steps:

1. A database class, i.e. a relation or a view, is mapped into a class. The relation classes get the four methods "create", "read", "update" and "delete" whereas the view classes only get "read". Hence, we assume that each view is a read only view.
2. Database datatypes are also mapped into classes. They, however, do not get any features or methods.
3. Columns are features of the class mapped from the corresponding relation. The types of the features are either the datatype classes or, if the column represents a foreign key, the associated relation class.
4. View columns are mapped into methods that are implemented using the recursive database column association "is defined using" as follows: 1) the database columns that are view columns, i.e. mapped into methods, are called in implementing messages, and 2) the database columns that are plain columns are put into a local container object that is passed as a parameter in a message to a predefined language method "is defined with".
5. Columns marked as primary keys are asserted to be categorised as 1) sorted and 2) searched.
6. Each searched for database column is mapped as a formal parameter of the corresponding "read" method. The formal parameter gets the appropriate type association.
7. For each sorted database column an implementing message for the "read" method is created that calls a language predefined method "sort" with the mapped feature as its actual parameter.
8. Finally, the indices are mapped into methods that are called in the "create", "update" and "delete" methods of the corresponding relation class. The index methods in turn are implemented by messages - one per involved column - to the predefined "sort" method using the column feature as the actual parameter.

The entities mappable into library objects usually are relations and views together with their columns and indices and - most commonly - DB data types.

3.2.5 The System Meter for Application Analysis / Specification Results (SPEC-SM)

Similarly to the essential system description also the system specification is split (for reasons of comprehensibility) into two subject areas. The first metamodel covers the system's interface specification, the second the application domain class specifications.

The interfaces specification metamodel is based on the two ideas of

1) specifying types (instead of repeatedly specifying almost similar functions) and of

2) specifying an interface in three parts: 1. specifying what is to be done (model), 2. specifying how it is presented to the system environment (view) and specifying in what order it is done and how it is steered (control).

These two ideas are nothing but applying object oriented concepts (as elaborated e.g. by Dahl (SIMULA) [DAH66], Goldberg (Smalltalk) [GO81] and Meyer [ME88]) and ideas about separating contents, presentation and control (e.g. Goldberg [GO85]) to the specification phase.

The graphical notation of the specification results metamodel is the following:

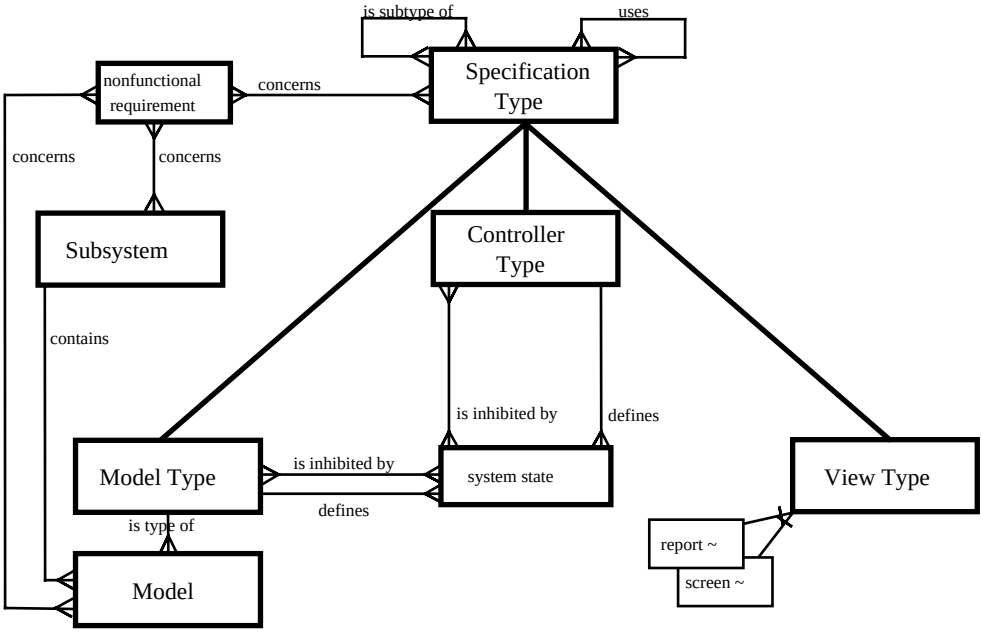

Figure 55: Application Analysis -Interface Specification

In the specification of interfaces we already adopt the paradigm of MVC, model-view-controller, that is widely used in object oriented implementations of graphical user interfaces, e.g. in Smalltalk-80, C++/Views and Enfin. It is, however, a new specification technique not to flatly specify views, entities of the user interface, but to already try at specification level to identify types of views. View types usually are based on model types i.e. repeating structures of objects to be presented in a view. Those model types are typically: 1) to show a top entity and its linked descendants, 2) to show one entity in detail, 3) to show many entities in a row, 4) to show a network of linked entities, etc. Finally we have the controller types and controllers that link user events on the views to model components or other controller actions and define which view components are affected in which way.

This typing of specifications not only allows the consistent and uniform specification of one system but also delivers candidate elements, the types, of reuse for follow-up projects. It is recommended [WALL96] to build up a library of specification (and corresponding implementation) types that may be reused. This approach is implicitly chosen by project

teams adopting GUI-standards like CUA or OpenView. These standards though leave lots of things, e.g. how to support standard operations like "create new objects", unstandardised. The typed specification approach tries to fill out that gap. A specification type thus may be the standard look and behaviour of a function to create a new object, or the standard look and behaviour to enter search arguments and get a list of the entities found, or the standard look and behaviour of the help system, etc.

A system may also get into some special system states globally or locally in only one view. Those states may either be defined as some model state or some executed controller actions. The effect of a system being in a certain state is usually that some subset of the controller actions may no longer be executed by the user. This is usually shown to the user by dimming the view elements which could trigger the inhibited controller actions. Some state may also inhibit models or model types to be accessed thus inhibiting all controller actions leading to those model components and indirectly also dimming the corresponding view elements.

The models may be aggregated into what is called a subsystem in this context. A subsystem may be formed either by organisational by functional or by technical aspects. Functional subsystems at specification level, e.g. an address administration subsystem with all layers (model, view and controller) and the corresponding class model elements, are of invaluable use as they can be reused together with their eventual implementation when the specification fits a requirement formulated in the earlier models, i.e. the domain and preliminary analysis models.

Finally, the concept of the model, here, is a grouping of elements - whole classes or just class elements - of the class model that are used in a situation together, e.g. the customer's number, name and city is used to identify the customer in a layout that otherwise displays accounting information. The model is a refinement and adaptation of what has been known as local data models in conventional structured software engineering.

The definition of the models leads us to the second part of the application analysis metamodel, the application class model. Every model must make use of a certain vocabulary, i.e. only well defined terms must be used. All those terms, i.e. names for classes, properties, methods, etc., are introduced and defined in the application class model. Such class or object models are parts of every modern design method ([RU91], [BO91], [RA96]). The metamodel diagram is as follows:

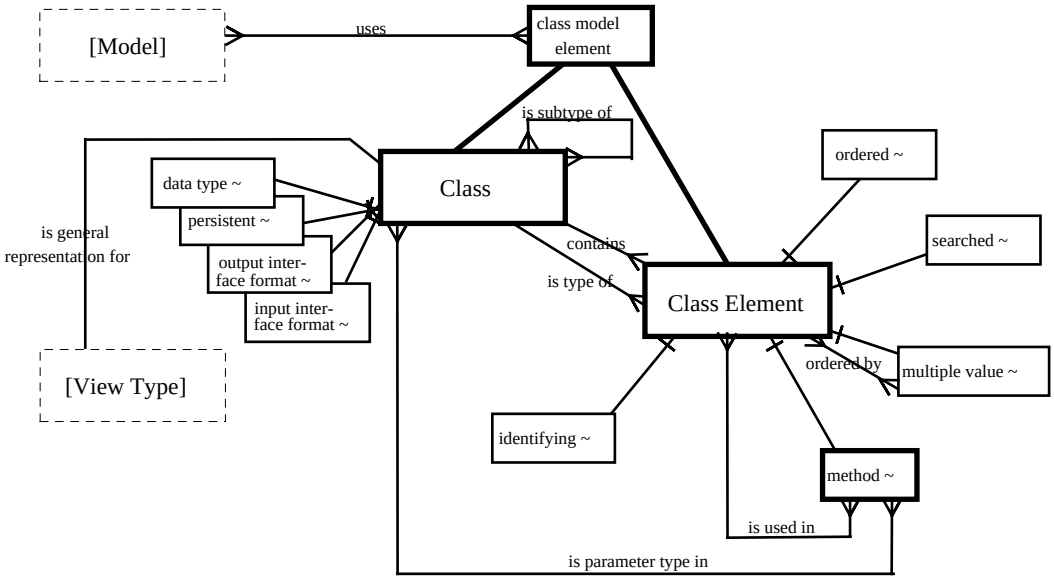

Figure 56: Application Analysis - Class Model

The notions of domain class, data and function types from the domain analysis are all subsumed in the concept of the application domain class. Also, new classes may be introduced at this level in order to represent data interface formats (to exchange date with other systems), concepts that may be useful for specifying the user interface, technology specifics or concretisations of non-essential requirements, e.g. help text classes, language classes to support multi-language applications, etc.

The attributes of a class are called class elements, because we did not want to imply the idea of data elements but rather subsume both data and functional elements under that meta-entity. The class elements are categorised into ① identifying elements, i.e. primary keys, and non identifying elements, ② data and method elements, of which the latter have an associated class for each parameter, ③ single and multiple value elements, of which the latter subsume lists, when some order is specified, or unordered sets of values, ④ into searched and unsearched elements and finally ⑤ into ordered and unordered elements (ordered here means that sets of instances may appear ordered by the values of the class element). Note that those 5 categories are not exclusive, i.e. one categorisation does not influence an other.

Whereas in the domain class model we do not yet model data elements or functionality, in this model we allow more detailed information by defining the class elements and even identifying the number and types, i.e. classes, of the method parameters. The domain analysis associations, which were modelled as a meta entity of its own there, is now transformed into the element to class association "is type of".

A final, rather marginal, association is also identified between class and view type. The association defines the typical layout representation for the class which may be especially useful for the data type classes. This supports the consistency of the specified interface by guaranteeing that identical objects are represented with identical graphical elements, e.g. the social security number always appears in the same format, address information always appears uniformly, etc.

Metamodel mapping to the SMM

This mapping is again not trivial. The class model part's mapping is only a slight variation of the one for the relational model (cf. 3.2.4). We will first explain the class model mapping and then the interface specification mapping:

1. An application class is mapped to a class. The persistent classes get the four methods "create", "read", "update" and "delete", the datatype classes do not get any method whereas the output interface format classes only get "create" and the input interface format classes get a "read" method only. The subtype association is directly mapped to the subtype association of the SMM.
2. Class elements are mapped to features of their class. The types of the features are their associated type classes.
3. Methods are mapped to methods that are implemented using the association "is used in" as follows: 1) if the used element is a method it is called in an implementing message providing local objects of the correct types as actual parameters, and 2) if the class element is not a method, it is put into a local container object that is passed as a parameter in a message to a predefined language method "is defined with". The method finally is added formal parameters with the associated classes as their types. The type class of a method element is mapped to the method's return value parameter class.
4. Columns marked as identifying are asserted to be categorised as 1) ordered and 2) searched.

5. Each searched for element is mapped as a formal parameter of the corresponding "read" method. The formal parameter gets the appropriate type association.
6. For each sorted database column an implementing message for the "read" method is created that calls a language predefined method "sort" with the mapped feature as its actual parameter.
7. Multiple valued class elements get an additional type in the SMM, which is allowed in this most generic metamodel. The associated type is the predefined language class "container object". In case an ordering sequence is specified, the element feature together with the referenced other elements are put into a local container object which is passed as an actual parameter in a message to the predefined method "sort in every method of the element's class.

The interface specification is mapped by the following sequence of rules:

1. Each specification type is mapped to a class. The subtype association is directly mapped to the SMM subtype association. The "uses" association is mapped by creating features for each of the used classes in the using class. Those features will get the types of the used classes. They are all global and inheritable.
2. The top view type inherits from a language predefined class "view type" with a global inheritable method "initialise". The top screen view types additionally inherits from a predefined class "screen view type" with the global inheritable methods "refresh" and "destroy".
3. The top model type inherits from a language predefined class "model type" with global inheritable methods "execute" and "flush" as well as a global inheritable feature "elements" of the predefined type "container object".
4. Each controller type gets a method "execute" which is implemented by messages to all methods of the features, i.e. used other specification types, it contains.
5. Each system state is mapped to a class with the methods "set" and "test". The defining controller type or model type add a message to "set" in their "execute" method's implementation. The inhibited types add a message to "test" in their "execute" method's implementation.
6. Each model is mapped as an instance of its type. The class model elements linked to the model are put into the appropriate "elements" container objects that are also instantiated according to the SMM consistency rule 10.

The entities mappable to library objects usually are the three kinds of specification types. Whole subsystems together with their models, used specification types and classes may also be mapped to library objects. Seldom, singular classes are also subject to reuse.

3.2.6 The System Meter for Domain Analysis Results (DOM-SM)

Many modern development strategies have adopted an analysis phase that focuses on the system's essentials, i.e. the concepts and mechanisms underlying a system that are "simply a fact" and independent of whether a computer based software system is built or not (cf. McMenamin/Palmer's Essential Systems Analysis [MP84], Yourdon's Modern Structured Analysis [YO89] or OMT Domain Analysis of Rumbaugh et al. [RU91]). Hence, a well elaborated essential system model is a sound foundation for building a good software system. The metamodel of an essentially described system consists of two main subject areas: 1) the class model and 2) the dynamic model consisting of use cases. Both models are linked via class views that are used by the elements, i.e. signals, of the dynamic model. Another - even though optionally elaborated - domain analysis result are state transition models. We start with the class model's metamodel:

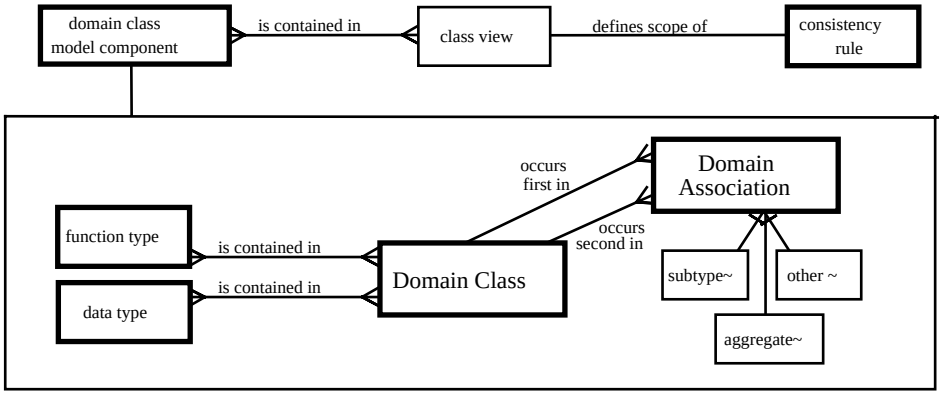

Figure 57: Domain Analysis - Class Model

Because the reader is assumed to be familiar with the basic entities occurring in the diagram above, we do not comment them in detail here; however, there are some remarks about some special semantics to be made:

First, domain associations are categorised into 1) subtype associations, i.e. class links that indicate a specialisation/generalisation or 'is kind of' relationships, 2) aggregation associations and 3) all the other associations.

Second, instead of modelling attribute types at this level of system description we directly assign one or more data types to the classes. For each of those assignments we also give a rough indication of how many attributes of that type are contained in the class. The detailed class model including all attributes, i.e. class elements, is deferred to the layer of application analysis (presented in 3.2.5).

Third, instead of modelling detailed methods at this level of system description we assign one or more function types to the classes. Each of those assignments implies that exactly one method of the function type's kind is contained in the class. Again, the detailed class model, including all method class elements, is deferred to the next level of modelling.

Fourth, the class model, i.e. the information entities that are supposed to be consistent within the system, is further described by a set of consistency rules that should cover aspects not possible to model in terms of class, association and data/function types. A common example for such a rule is the fact that a person's civil state cannot switch back from married to unmarried but can only change from married to divorced or widowed. The entities of the class model component and class view are included for the simplification of the metamodel.

The class model may also be interpreted as an extended ER model [CH76] because the focus does not lie - at this layer - on the object oriented association of subtyping and the incorporation of methods into entity types by attaching function types to classes.

In the domain use case model, the next model we present, all is centered around the concept of the use case [JA92] which is triggered by some event in the systems environment. The use case has its roots in the concepts of essential activity first presented by McMenamin/Palmer [MP84]:

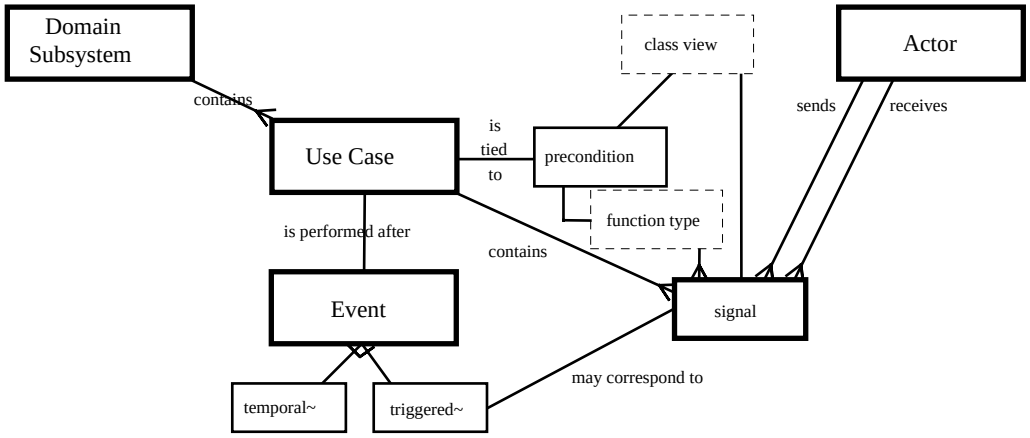

Figure 58: Domain Analysis - Use Case Model

The use case model consists mainly of the 1:1 linked pair of event, i.e. something that happened in the system's environment, and use case, i.e. the system's reaction scheme to an event. The system communicates with its environment, which is modelled by a set of actors, through signals. Signals may also be used for mere system internal communication, i.e. the send and receive associations to actors are optional.. Signals use certain subsets of the domain class model via class views, i.e. subsets of classes, associations, function and data types.

We finally have some refinements: events may be categorised into 1) temporal events, i.e. when some time has elapsed or a system state is reached that forces a system action, and 2) triggered events, i.e. events that are actually triggered via a signal from the environment. Use cases usually assume some conditions in order to be executed correctly. These presumed facts are stated in preconditions that are - as the signals - linked to a class view.

The state transition model, finally, enhances the domain class model by introducing additional subtypes to some domain classes. Those subtypes are the states. The point is that a certain object may change its state dynamically but in a controlled manner, i.e. only an identified set of functional components, signals or function types, may push that object into some new state.

The graphical notation of the domain state transition metamodel is the following:

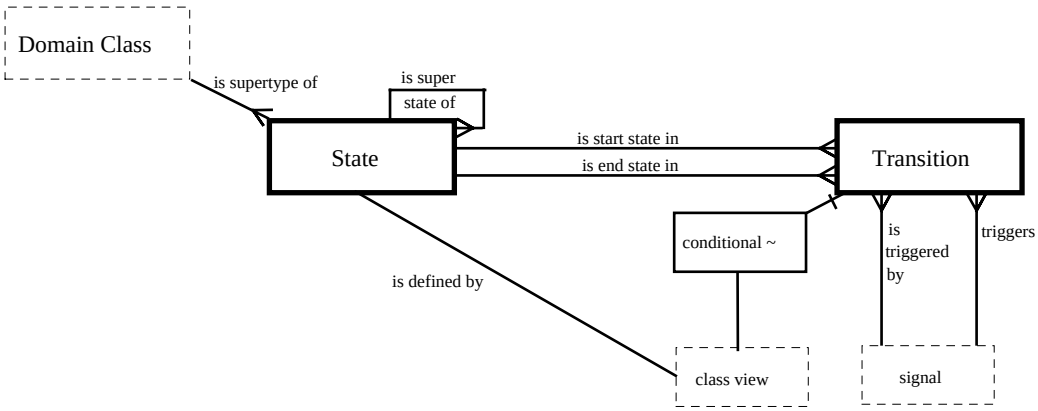

Figure 59: Domain Analysis - State Transition Model

A state is some specialisation of a domain class, e.g. an account in overdrawn state. States may also be specialisation's of some other states, e.g. an account may be slightly or severely overdrawn; two substates of the overdrawn state. The latter enhancement of the traditional state transition paradigm (cf. Hopcroft/Ullman [HO79]) was first introduced by David Harel [HAR87]. A state is defined by a set of some related values, e.g. "overdrawn"

is defined by the condition that the account's balance < 0, "severely overdrawn" when the account's balance < -$10'000. Any state may be left through one or several transitions and entered through one or several transitions which are triggered by signals and may - as a side effect - trigger some other signals. Transitions may furthermore be bound to some condition. We speak of conditional transitions then. To give an example: the state "normal" of an account may be left by a transition "drawback" under the condition that the amount > the account's balance. We would model the signal "draw back amount" sent by an actor "bank customer" to trigger that transition. As the transition's reaction the state "account overdrawn" is entered and maybe the signal "mail overdrawn state" received by the "customer" is additionally triggered.

As the careful reader may have noticed this modelling interferes to some degree with the use case modelling: e.g. there are temporal events based on system internal states that may also trigger essential activities. The author suggests state-transition-modelling for truly state based behaviour and temporal event modelling for actually timed events, e.g. "1 ms later", "two days after", "each week", etc.

Metamodel mapping to the SMM

The class model part's mapping is only a slight variation of the one for the relational model (cf. 3.2.4). We will first explain the class model mapping and then the interface specification mapping:

1. Domain classes and data types are mapped to classes.
2. The subtype associations are mapped to subtype associations.
3. Aggregate associations are mapped to features of the first involved class. The feature types are the second classes. The features are named as the second class.
4. The other associations are mapped to features of each class returning a formal parameter typed with the other class. The feature of class1 is named as the association and the one in class 2 is prefixed the keyword "back". In case the partner side of the association is of m or mc cardinality, the feature is a method with a formal parameter of the appropriate type. This formal parameter is furthermore a container object and of category update, i.e. objects passed to it are modelled as being updated.
5. Data type contained in a class are treated like aggregates.
6. Function Types are mapped to classes named like the function type with the postfix "-able", e.g. "create-able" for a "create" function type. This class is added an inheritable method named exactly like the function type. Each domain class that contains a function type is a subtype of the function type class. The method is added formal parameters per kind of basic function encompassed in the function type.
7. Consistency rules are mapped to methods of a predefined language class "Domain Class Model".
8. Each use case is mapped to a method.
9. Each actor is mapped to a class with the two methods "send" and "receive".
10. Each event is mapped to a method of the language class "Events". This method is implemented with a message to the linked use case method.
11. Each domain subsystem is mapped to an object containing the use case methods.
12. Each state is mapped to a class which is a subtype of the domain class or the super state class, depending on what applies.
13. Each transition is mapped to a method of its start state returning a formal create parameter typed with the end state. This method is called in the implementation of

each use case that is linked via the "is triggered by" association with the end state class as actual parameter.

14. Each class view, finally, is mapped to message calls as follows:
    - If a domain class and an appropriate function type are in the view, then the inherited function type method of the domain class is called with all the class features as actual parameters.
    - If a domain class, an appropriate function type and an appropriate data type are in the view, then the inherited function type method of the domain class is called with just the data type class features as actual parameters.
    - If an association (no aggregate and subtype associations are usable in views because they are anonymous) is in the view, the corresponding method or the language method "uses" is called with the second class as actual parameter.

    The method implemented with those message calls depends on the view links:
    - If tied to a precondition it implements the use case method.
    - If tied to a signal it implements the use case method. If the signal is linked to an actor via the send association it implements the actor's send method and if via the receive association the receive method. If the signal is linked to a transition via the "triggers" association it also implements the transition method.
    - If the view is linked to a conditional transition it implements the transition method.
    - If the view is linked to a state it implements the "isDefinedBy" method of language class "States".

    Signals and preconditions, thus, are only used as a linking means. They are not mapped directly into SMM entities.

3.2.7 The System Meter for Preliminary Analysis Results (PRE-SM)

The software process we defined (cf. 3.3.3 and [MO95b] [MO96b]) starts with a very coarse model that consists of a set of functional system goals, a set of subject areas, i.e. anticipated groupings of domain classes and links between those two sets. The links are defining the goals for each subject area. Typical functional goals[27] in information systems are the ubiquitous "creation", "read", "update" and "deletion" functionalities. In device control systems these might be "monitoring", "steering" and "protocolling". The subject areas - on the other hand - might be "stock information", "customer information" and "order information" for IS and "heating device" or "cooling device" for control systems, i.e. rough indications of what the system is about. In order to distinguish those very coarse concepts with respect to their complexity, every subject area is given an approximate number of contained classes and every goal is given an estimated complexity number[28]. Additionally, the goals may be hierarchically ordered.

We thus have the following diagrammed preliminary analysis metamodel:

[27] In more recent works - as well as in the industry application of the System Meter method - we are using the term „functionality“ instead of „functional system goal“. We do so because the notion of a goal may lead to some confusion with project and quality objectives.

[28] The complexity number is an integer. The values are pseudo-defined by the rule that each basic database functionality (CRUD) has complexity 1.

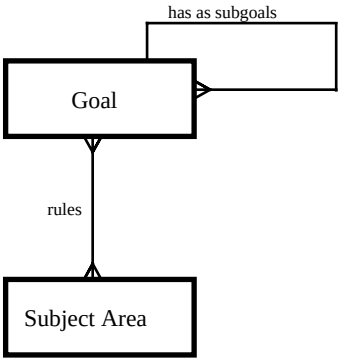

Figure 60: A Metamodel of Preliminary Analysis Results

A preliminary description of a system is more a case of limiting the set of possible implemented systems than actually describing a system in its traditional sense[29]. The description gives an idea about the systems functionality and of its scope.

Metamodel mapping to the SMM

1. Any subject area of complexity c is mapped to a description object that contains c classes. The classes are just "dummies" and each contains a single method called "fulfilGoals".
2. Any elementary goal (with complexity 1) is mapped to a method.
3. Any higher order or non elementary goal is mapped to a description object containing the subordinate goal objects (either description objects or methods).
4. If a subject area is linked to a goal, this is mapped to an isImplementedBy association between the "fulfilGoals" method and a message calling the goal method.

A sample application of the mapped System Meter just introduced is given here using the rules given in 4.2.1 to denote goals and subject areas. Let us look at a sample preliminary system description (of a truly tiny system):

```
Goal create.
Goal delete.
Goal administrate = create, delete.
Goal "save historical states" 2 .

Subject Area orders 2 .
Subject Area orders isRuledBy create.

Subject Area customers 2.
Subject Area customers isRuledBy administrate.

Subject Area stock 3.
Subject Area stock isRuledBy administrate,
          "save historical states".
```

Figure 61: A Sample Preliminary System Description

The first message

[29] A system is traditionally viewed as an entity which accepts some kinds of inputs and produces some outputs.

```
Goal create.
```

is a message to the language method 'Goal <name> [= <complexity>] .' with the following actual parameters: it creates a method named 'create' out of a literal object passed as the input/read parameter. For the parameter <complexity> a language provided literal value of 1 is used. So in summary we have the following list of description objects:

Meta-Class	*Name*	*Meta Associations and Categories*	*isWrittenBy Association*
Message	01	calls 'Goal <name> [= <complexity>] .', has actual parameters 02, 04, 05 anonymous	'Goal <name> [= <complexity>] .'
Actual Parameter	02	equals 03 correspondsTo <name> isUsedIn 01 anonymous	
Object	03	value is 'create' isReferredIn 02 constant literal anonymous	
Actual Parameter	04	equals language literal 1 correspondsTo <complexity> isUsedIn 01 anonymous	
Actual Parameter	05	equals create correspondsTo <new goal method> isUsedIn 01 anonymous	
Method	create	isReferredIn 05	01, 03, language literal 1

Table 9: The Description Objects Implied by the 1st Message

In the next message we create the goal method "delete" in a similar manner. Next we encounter is a call to 'Goal <name> = <subgoals> ... .'. This now introduces an object 'administrate' and adds the two previously created methods into its list of contained sub-objects. The object list is enhanced as follows:

Meta-Class	*Name*	*Meta Associations and Categories*	*isWrittenBy Association*
Message	15	calls 'Goal <name> = <subgoals> ... .', has actual parameters 16, 18, 19 anonymous	calls 'Goal <name> = <subgoals> ... .'
Actual Parameter	16	equals 17 correspondsTo <name> isUsedIn 15 anonymous	
Object	17	value is 'administrate' isReferredIn 16 constant literal anonymous	

Actual Parameter	18	equals create, delete correspondsTo <subgoals> isUsedIn 15 anonymous	
Actual Parameter	19	equals administrate correspondsTo <new goal object> isUsedIn 15 anonymous	
Object	administrate	isReferredIn 19 contains create, delete	15, 17, create, delete, create, delete (create and delete are contained twice because they are contained and they occur in administrate's creating message)

Table 10: Additional Description Objects Implied by the 3rd Message

In this manner all the describing messages are analysed. Then the external sizes (cf. 3.2.1) are assigned to the objects. For the objects we have listed above this means:

Meta-Class	***Name***	***External Size***	***Internal Size***	***Size***
Message	01	1	1	2
Actual Parameter	02	1	0	1
Object	03	1	0	1
Actual Parameter	04	1	0	1
Actual Parameter	05	1	0	1
Method	create	1	3	4
Message	06	1	1	2
Actual Parameter	07	1	0	1
Object	08	1	0	1
Actual Parameter	09	1	0	1
Actual Parameter	10	1	0	1
Method	create	1	3	4
Message	11	1	1	2
Actual Parameter	12	1	0	1
Object	13	1	0	1
Actual Parameter	14	1	0	1
Actual Parameter	15	1	0	1
Object	administrate	1	6	7

Table 11: The System Meter Values for the Sample Description Objects

In order to get the system's total size we finally sum up the sizes of all the objects. This procedure results in a size of 33 for the objects in the list above. For the total system described here a size of 186 was measured.

3.3 New Measures and Metamodels for Software Processes

3.3.1 The Estimation Quality Factor

In section 1.5 we introduced two most important criteria to assess estimation methods: 1) the effort of estimation and 2) the estimation quality. In section 2.2 we concluded that estimation quality may be measured using the approximation bias dA, if - and this is the point here - the estimation methods are both metrics based, i.e. follow the strategy given in 2.1. In practice, however, one should also have an instrument to measure estimation quality of arbitrary methods. We may then use this instrument, which itself has to be a process metric, to assess the benefits of rational estimation methods and growing empirical databases over "chaotical" estimates (as done in 4.3.4). As shown in a survey (cf. [DM83]) the estimation quality correlates best with project success. It outperforms factors such as technology used, staff experience and others.

A Metamodel of the Estimation Process

According to DeMarco [DM83] estimation is assumed to take place at certain points of time within project duration. Project start is denoted as T_0. At least at the end of the project T_{end} - which by definition of a project must exist - the estimate will converge to the actually measured value M. A non existing estimate is modelled as a value of 0. The estimation process or history of a project may therefore be viewed as a set of pairs of time and estimated values $\{(T_0, E_0), (T_1, E_1), (T_2, E_2), ..., (T_{end}, M)\}$.

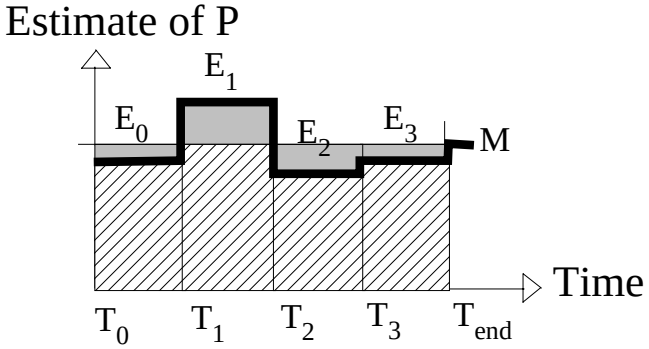

Figure 62: The Estimation History of a Parameter P

The Estimation Quality Factor (EQF)

DeMarco defined an EQF metric based on the metamodel above that yields values from 0 to infinity. This scale is however difficult to interpret, e.g. it is not easy to answer whether 10 is a good or bad value. We therefore propose and use a new metric that measures the quality in a dimensionless scale from 0% to 100%. It follows the viewpoints that

① if no estimate existed for the whole project duration EQF yields 0%

② when from project start on the right value was estimated throughout the project EQF is 100%

③ Under- and overestimates are treated equally.

Any values in-between 0% and 100% can be interpreted as having estimates of an average bias of 100%-EQF throughout the project. We furthermore extended DeMarco's EQF

definition, which was restricted to effort estimates, to any quantitative parameter P of software or software processes. We therefore denote the EQF for a parameter P as EQF_P and define it using the formula:

$$EQF_P \equiv \sum_{0}^{end-1} [\min (E_i, M) \times (T_{i+1} - T_i)] / \sum_{0}^{end-1} [\max (E_i, M) \times (T_{i+1} - T_i)]$$

More intuitively we can also define the EQF metric using the differently textured areas from figure 62 above:

$$EQF_P = (100\% - \frac{\square}{\square + \boxslash})$$

Figure 63: The Intuitive Definition of the EQF

For the software process as a whole, which is - according to section 1.1 - mainly described with the three parameters of 1) effort, 2) time and 3) the product, we introduce the aggregate EQF defined as

$$EQF \equiv (EQF_{effort} + EQF_{duration} + EQF_{product\ size}) / 3$$

This measure was evaluated in a subset of some 10 projects in our field study (cf. 4.3.4). Within a quality management system it can also successfully be used as a goal measure. One has to be careful however to assign the goals of a high EQF and a high productivity to the same person. If one produces above the standard productivity rate, estimates will be bad, and if the estimates are optimised, productivity will never rise (cf. DM82]).

3.3.2 The Restrictively Cyclic Model

As we saw in 2.3.4 and 2.3.5 there exist various models for the software process, e.g. the waterfall model, spiral, fountain and pinball models. We found that the waterfall model is too restrictive and the non-waterfall models are too open to be of practical use. We therefore introduce our own process metamodel that combines the waterfall idea with the pinball model.

The Key Ideas

The first basic idea is to define the process steps or phases through the results. For example, we define the preliminary analysis phase as all activities leading to a first released version of the preliminary analysis results, i.e. goals and subject areas. The structure of the activities within the phase is left open, only a few recommendations are made. During a phase you can therefore - adhering to the pinball model - 1) work directly on the phase's results in several iterations, 2) work on experimental or 3) evolutionary prototypes of results from coming phases.

The second idea is that after the first release of a system description layer it is no longer allowed to change anything on that layer unless it is 1) an error correction or 2) a formally tracked and approved change that has its effects on cost, time and the product delivered. This requires that each result layer, i.e. a phase, describes the system as a whole and without incorporating redundant parts of the other layers. The total system description, i.e. the dynamic system documentation, consists of the currently valid descriptions on each of the layers. System documentation thus starts already after the completion of the first phase and is controlled - already during development time - by a formal problem, change and configuration management process. We may also view this as pushing the released system descriptions into their usage and maintenance phase before this phase actually begins for the coded system.

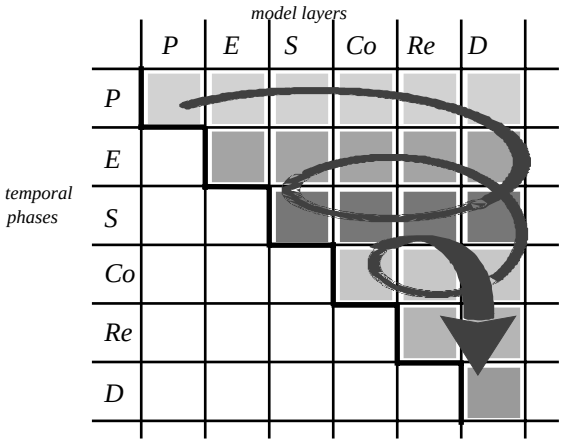

Figure 64: The Restrictively Cyclic "Funnel" Software Process Model

In summary we may describe the restrictively cyclic software process as being based on the results, allowing the prototypical elaboration of any result at any time but restricting the change of results that are once released. This strategy yields a "funnel" shaped macro process that guarantees that the development cycles will become smaller and smaller with the completion of phases, i.e. release of result layers. This is a substantial difference to the traditional waterfall (cf. 2.3.5) or cyclic software process models (cf. 2.3.6).

Prototyping Modes

According to modern practices we can distinguish three types of prototypical system descriptions that essentially differ only with respect to their method of verification, i.e. amount of quality assurance:

① Experimental prototypes

These prototypes are unproved or proofs are left to the authors.

② Evolutionary prototypes

Evolutionary prototypes are proven in a single formal verification step, i.e. a review or a test.

③ RAD (Rapid Application Development) prototypes

These prototypes actually have a misleading name because they are fully verified[30] system descriptions. Nevertheless they are considered to be prototypes because they encompass only a reduced subset of the total system.

The points of usage of those three modes of prototypes - together with the maintenance, i.e. problem and change management process - may be positioned in the diagram of model layers and temporal phases as follows:

A prototyping process may - according to the loose pinball process metamodel - instantiated at any time it is needed. However, it always includes:

❶ The definition of the prototype's intent and feedback mechanism into the main software process. Typical intents are a) feasibility study, b) user interface verification, c) specification/functional verification, d) performance analysis and e) (pilot) usage.

[30] The expression "fully verified" is theoretically and practically unsound because it is impossible to verify a system by testing or reviewing it. Fully verified here means that all suitable quality assurance techniques are applied in full to the result.

❷ The definition of the prototype scope, i.e. a subset[31] of the current system description. For example, if we detach a prototyping process at the preliminary analysis level, we define the prototype scope as a subset of subject areas and functional goals (unless we have prototypical results of other phases already at hand).

❸ The definition of the result layers, i.e. phases, that should prototypically be elaborated. For example, if we want to verify that some functional goal is technically achievable we would undergo the technical design and implementation phases.

❹ The definition of the prototype mode, i.e. experimental, evolutionary or RAD, i.e. fully verified. The mode is usually dependent on the intent and the phase we are already in. If the prototype phases skip some phases with respect to the current phase we usually chose experimental mode. Prototypes of phase results just following the current phase will usually be evolutionary because we want to reuse the results immediately afterwards. If the prototype is going to be installed and used productively we will choose the fully verified mode.

The points of usage of those three modes of prototypes - together with the maintenance, i.e. problem and change management process - may be positioned in the diagram of model layers and temporal phases as follows:

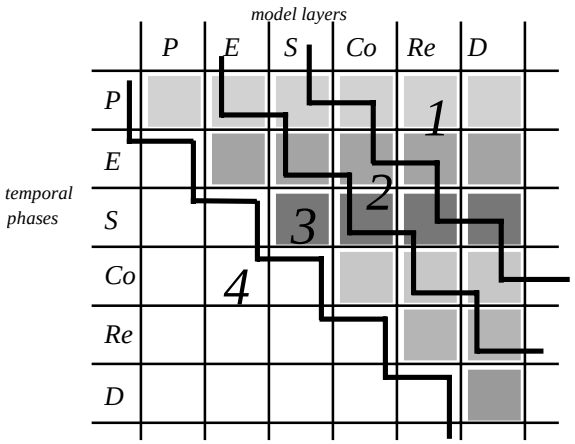

Figure 65: Software Process Modes

A prototyping process is planned, estimated and managed as an individual detached software process, although it may also be viewed as a sub-process of the top-process. It is even possible within a prototyping process, to start yet another prototyping subprocess. The efforts of prototyping processes that do not lead to fully verified results are added to the current phase, whereas the efforts of fully verified prototypes - which then are not really prototypes anymore - are added to the respective phases.

For any software process - be it a prototyping or primary process - the definitional element is the system scope over which the process runs. As soon as the scope is changed - usually

[31] For RAD prototypes this corresponds to a versioning of the system into pieces of delivery. Those pieces are typically defined using the "fit to schedule" planning technique, i.e. each version is of such complexity that it can be developed and set to production within a given time. This development strategy is also known as "Time-Box" development. Typically each time-box spans over 3 to 6 months.

it is partitioned, i.e. only some subset is further developed - either a subprocess is instantiated or the primary process is stopped while only partially complete. Eventually a new follow-up process, which may reuse the already developed descriptions, is then started on the changed scope.

3.3.3 The BIO Software Process

The BIO[32] software process is an instantiation of the "funnel" software process model explained in the previous subsection. It was developed in several released versions ([MO93] [MO95b] and currently [MO96b]) and was used as a template model for all the processes investigated in the field study (cf. chapter 4). Even though not all of the process used BIO it was possible to map the artefacts to the BIO template artefacts. The template consists of 6 model layers and temporal phases and the two accompanying span activities (cf. 2.3.6) of project and quality management:

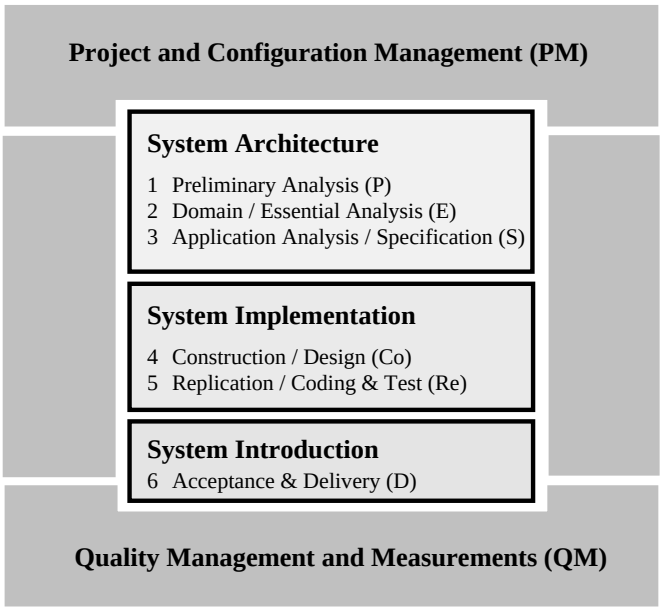

Figure 66: The BIO Software Process

Besides the fact that BIO follows the restrictively cyclic process model the following main new idea was introduced:

Requirements which traditionally are considered to form one layer are split into three distinct layers. Each of the layers describes the complete system or subsystem although the description granularity or scale is different. This idea may be compared to drawing maps of decreasing scale of some country. The benefits of this splitting are twofold: First, during development we deliver stable results earlier which may then be used to estimate and plan the future development process. Second, during maintenance changes of requirements may be classified using the layers: minuscule changes of layouts and attributes that should appear on reports may be assigned to the specification layer, more severe changes of the systems real world base, i.e. entire new classes or use cases to be supported, may be assigned to the domain analysis layer and entire new subject areas or functional goals are assignable to the preliminary analysis layer. Hence, we may keep entire requirement layers unchanged when only details change. Also the rates of change or change velocities are different for the different layers:

[32] This acronym is derived from **B**edag **I**nformatik **O**bject Oriented Process. It is a publicly available document at reproduction cost. The copyright is however corporate owned.

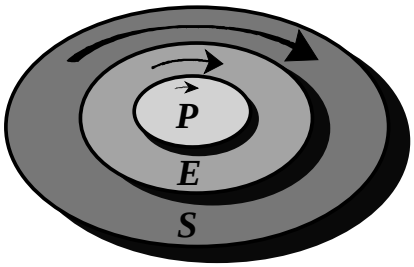

Figure 67: Change Velocities of the Analysis/Requirements Layers

By adhering to a separation of the requirements layer we therefore achieve a minimum of changes, thus introducing the desired stability into the development and maintenance processes.

The Detailed BIO Results

Instead of focusing on activities, steps and procedures BIO focuses on the results, the deliverables of the software process. BIO almost entirely relies on the different metamodels presented in section 3.2. There are several additional results included in the list below. Those results however need no further explanation in the context of our work. More information is included in the current BIO documents [MO96b] as well as recommendations for strategies, tools and techniques to produce the results.

The following list of deliverables covers all the layers and span activity results. For the three modes of the process, i.e. experimental, evolutionary and fully verified different columns are given. The columns contain empirically derived percentages. Those percentages are nothing but the correlation coefficients of a mass of purely linear estimation models. They correlate the effort of producing the result in the given mode to the effort of producing all the standard results in fully verified mode. The latter effort is called the standard effort. E.g. it requires 30% of the overall development effort to produce the fully verified code, i.e. tested by independent personnel for functionality and robustness, 25% to produce code that is tested for functionality (as required for evolutionary code) and 10% to produce self-tested code (as required for experimental code).

BIO Layer	***Result***	***Exp. %***	***Evol. %***	***Full %***
Preliminary Analysis 5%	Subject Areas	1/2 %	1 %	**2 %**
	Goals	1/2 %	1 %	**3 %**
Domain Analysis 14%	Use Case Model	1 %	3 %	**5 %**
	Domain Class Model	1 %	3 %	**5 %**
	State-Transition Models	1 %	2 %	4 %
	Non-essential Requirements	1 %	2 %	**4 %**
Application Analysis 18%	Specification Types *	1 %	3 %	**5 %**
	Models	2 %	3 %	**5 %**
	System States	2 %	3 %	4 %
	Application Class Model	2 %	4 %	**6 %**
	Non-functional Requirements	1/2 %	1 %	**2 %**

Construction 19%	Implementation Patterns *	2 %	4 %	**5 %**
	Relational Model	1 %	3 %	**4 %**
	Technical Class Model	1 %	2 %	**2 %**
	Test Data	1 %	3 %	**4 %**
	Test Cases	2 %	3 %	**4 %**
Replication 38%	Tuned Pattern / Rel. / Class Model *	2 %	4 %	5 %
	Code	10 %	25 %	**30 %**
	Administrative & Installation Code	1 %	4 %	**5 %**
	Platform Port *	2 %	8 %	10%
	Layout Translation (Multilanguage) *	4 %	5 %	5 %
	System Administrator Manual *	1 %	2 %	**3 %**
	User Manual / Online Help *	1 %	3 %	4 %
	Forms (for manual processing)	1/2 %	1/2 %	1 %
Delivery 6%	Acceptance	1/2 %	2 %	**3 %**
	Installations *	1/4 %	1 %	**1 %**
	User Instruction *	1%	1 1/2%	**2 %**
	Organisational Changes	1/2 %	1 1/2 %	2 %
	Data Migration	2 %	3 %	4 %
Project Management 10%	Plans	1 %	1 1/2 %	**2 %**
	Estimates	1/2 %	1 %	**1 %**
	Configuration Mgmt / Infrastructure	2 %	2 %	**3 %**
	Problem and Change Mgmt	1 %	2 %	**3 %**
	Controlling and Reporting	1 %	1 %	**1 %**
	Evaluations *	1/2 %	2 %	3 %
	Preparation of Organisational Changes	1/2 %	2 %	3 %
	Preparation of User Instructions	1 %	2 %	3 %
	Preparation of Data Migration	2 %	2 %	3 %
Quality Management 8%	Risk Analysis / Quality Plans	1 %	1 1/2 %	**2 %**
	Measurements	1 %	2 %	**3 %**
	Defining Development Standards	1/2 %	1 1/2 %	**3 %**
	Developer Instruction	1/4 %	1/2 %	1/2 %
	Project Reviews	1/4 %	1/2 %	1/2 %

Table 12: The BIO Results and their Process Completeness Percentages

In the table above the underlined items make up the standard results, i.e. the standard effort. Note that the sum of the main layers equals 100%. The percentages of the span activities are to be considered as fractions within the main effort. The items marked with a * may be repeatedly instantiated. Their percentages are given for one instance.

The purely linear estimation models, i.e. their percentages, are used for two purposes:

① In step C of the generic estimation procedure (cf. 2.1.1) we may tailor the initial standard effort estimate, which equals 100%, to the deliverables actually needed in the planned process (e.g. we may only want to produce some experimental design and code). We sum up the percentages of the needed results (e.g. 2% for experimental solution patterns, 1% for an experimental technical class model and 10% for experimental code = 13%) and reduce the initial estimate proportionally.

② When calibrating estimation models (cf. 2.1.2), which was extensively done in our field study (cf. appendix F, the detailed measurement forms), we use the percentages to assess the process completeness as defined in 2.3.4. We sum up the percentages of the actually developed results according to their observed verification state. Step C^{-1} of the calibration then consists in expanding (or reducing) the observed effort proportionally to get the standard effort.

4 Results of a Field Study

4.1 Goals and Settings of the Field Study

4.1.1 Goals and Evaluation Strategies

4.1.1.1 Assessing the Estimation Power of the System Meter for Preliminary Analysis (PRE-SM)

The System Meter was primarily designed to support the task of estimation. Because reasonable estimates should be obtainable as early as possible, we defined the System Meter for preliminary analysis results (cf. 3.2.7) and evaluated its behaviour with respect to the process metric of effort (cf. 2.3.3).

The estimation model thus evaluated was, according to the formal notation introduced in 2.1.1, a 5-tuple of:

① Predictor Metamodel = Preliminary Analysis Metamodel

② Predictor Metric = System Meter (PRE-Variant)

③ Empirical Database = 36 surveyed projects (cf. 4.1.2 for details)

④ Result Metric = Person Days

⑤ Result Metamodel = BIO Processes of Preliminary Analysis through Installation (including Project Management, Configuration Management, Estimation and Measurement and Quality Management)

Before one may understand the evaluation details (cf. 4.3.1), the evaluation criteria, introduced in section 1.4, are briefly resumed and commented. The two criteria are:

1 yield estimates as precisely as possible

2 yield estimates as cheaply and quickly as possible.

In order to assess with respect to the first criterion, one is obliged to estimate some parameter X with the method in question and, at the end of the process, measure the effective outcome of X. The relative bias, defined as:

$$\text{estimation bias} = [X_{estim} - X_{eff}]^2 / X_{eff}$$

should then be minimal compared to estimates gained with other methods. This, however, is a time consuming evaluation strategy because one must have the estimation method ready at the point of estimation and then wait until completion of the process.

On the other hand, when the method works according to the metric based strategy explained in section 2.1.1 one can also calculate the approximation bias dA of the predictor metric P with respect to the result metric R. Actually the formula for the relative correlation bias of one pair of data is very similar to the one for the simple estimation bias, namely:

$$dA = [A(P) - R]^2 / R$$

where A is the approximation or estimation function.

The assessment technique using dA rather than the estimation bias has the advantage that it can be applied entirely after the completion of the evaluated projects, i.e. the estimation method can be defined after a project's completion. We used this technique in our survey.

To understand the second criterion, the estimation cost and speed, one must be aware that estimation incorporates both 1) the efforts to establish a reviewed and stable version of the predictor model and 2) the efforts of the estimation procedure. Whereas part two of the effort, the estimation procedure effort, usually is minor - especially when automated tools

are used - part one, the model building effort, is considerable and usually also involves interactions with customers. We evaluated the estimation models with respect to effort spent (= cost) only, assuming a strong and constant correlation with duration.

4.1.1.2 Assessing the Estimation Power of the System Meter for Domain Analysis (DOME-SM)

The System Meter defined for domain analysis results (cf. 3.2.6), which is obtainable considerably later than the PRE-SM but promises better estimates, was also evaluated with respect to the process metric of effort. Additionally, the conventional technique of Function Point counting (cf. 2.2.4), which is also measurable on domain analysis results, was evaluated in the same empirical database.

The estimation models evaluated were:

A
- ① Predictor Metamodel = Domain Analysis Metamodel
- ② Predictor Metric = System Meter (DOME-Variant)
- ③ Empirical Database = 36 surveyed projects (cf. 4.1.2 for details)
- ④ Result Metric = Person Days
- ⑤ Result Metamodel = BIO Processes of Preliminary Analysis through Installation (including Project Management, Configuration Management, Estimation and Measurement and Quality Management)

and

B ①, ③, ④ and ⑤ identical to A, except
- ② Predictor Metric = Function Points

The pairs of relative biases from the SM and FP approximation function to the observed effort values were furthermore assessed against the zero-hypothesis H0 that the average biases do not differ and the one-sided hypothesis H1 that the SM average bias is lower than the FP average bias. The hypothesis analysis was conducted using the Wilcoxon signed rank sum test (cf. 4.2.4). This test was chosen because the samples 1) were paired, 2) are continuous data and 3) the sample set was not large enough for the z-test (cf. [RI78]).

4.1.1.3 Measuring the Effects of Reuse on Productivity

The survey of over thirty projects also included four independent sets (A1, A2), (B1, B2), (C1, C2, C3, C4, C5, C6, C7, C8) and (D) of projects that explicitly had to develop (first project) and then apply frameworks. Although this is too small a number of projects to allow any statistically sound conclusions to be drawn, we were interested in seeing whether quantitative metrics could tell us anything about the impact of reusable components on the productivity of projects that both develop and reuse such components. Productivity is to be understood as defined in 2.3.3, i.e. as the ratio of result size and effort, e.g. SM/PD. We used the System Meter at the preliminary analysis level and - for comparison purposes - the Function Point metric at the domain analysis level.

From the estimation point of view, we are interested in observing productivity that is as uniform as possible. From a managerial point of view, we want to observe increases of productivity. Both goals obviously cannot be achieved by the same metric. While the SM allows the measurement of reusable components it does not enforce it. The SM-variant without consideration of reuse is called the flat SM or - shortly - SM'. We used SM' in only one project A2. We expected it to report a productivity increase whereas SM should level out the reuse based productivity variances. For the FP metric, which does not take reuse into account, no variant can be defined and we therefore expected it to be sensitive to reuse. We analysed the productivity bias between the framework construction projects and each of the follow-up projects, i.e. on 9 project pairs.

4.1.1.4 Measuring the Effects of Team Size and Acceleration on Productivity

Since the famous book "The mythical man-month" written by Brooks [BR75] appeared in 1975 we all know that adding manpower to a late software project makes it even later. This law - if one may call that a law - was formalised in our field study. Large teams are typically built when the planned and imposed duration is smaller than the duration one would rationally estimate. The acceleration (cf. 2.3.5) measure captures the amount of such enhanced project dynamics. The law may now be formulated in the hypothesis that the team size does not grow linearly with acceleration. Instead we assume, that the team size - and with it the effort inflation (also cf. 2.3.5) - increases according to the square of the acceleration. This hypothesis was tested in a few samples. The number of samples was only 4. Due to this low number no statistical evidence is presented here. The samples however showed - as singular examples - that the hypothesis is a good model for reality, i.e. none of the samples showed a completely contradictory behaviour.

4.1.1.5 Measuring Estimation Quality for Unstructured and Metric Based Estimation Techniques

Since 1993 we collected estimates and effective outcomes of some 8 projects we were especially closely involved with, i.e. we had access to the history of estimates. The estimation techniques used in these samples were unstructured for the first 2, the Function Point Method (cf. 2.2.4) for the next 6 projects. The Function Point Method was using an increasingly better empirical database. In order to compare the estimation quality of those different projects we used the EQF as explained in 3.3.1. We were interested in observing the behaviour of the EQF over time: does it increase? And if yes: at what rate and does it do so continuously? The sample base was again too low to do a statistical analysis. We simply discuss a bar chart of the EQF values on a time axis.

4.1.2 Characteristics of the Projects

The set of investigated projects (all measurements were either accomplished or supervised by the author) varied in many aspects:

The variation of the *organisational and personal aspect* may probably be best captured by analysing the contractor companies involved. The set mainly consists of projects undertaken at 3 medium sized information services companies (17, 9 and 4 projects). Additionally, there are 3 university projects and 3 projects from a large chemical industry site.

The *total project efforts* observed ranged from 1.5 person-months to more than 100 person-months. Whereas the *average team sizes* ranged from as few as 0.8 persons to slightly over 5 persons and *maximum team sizes* from 1.2 to 10.5. The month of project completion ranges from December 1987 until November 1995.

From the *technological point of view* the projects may be categorised into 25 projects using Smalltalk, 7 using 4GLs and 4 using C++.

From the *methodological point of view* there were 19 projects using the BIO method, a synthesis of Essential Systems Analysis (ESA) [MP84] [YO89] with Rumbaugh's Object Modelling Technique (OMT) [RU91] as described in section 3.3.3, 4 projects using "classical" ESA and the rest of the projects didn't really follow a method but were heuristically casted into the BIO reference process.

The *application domains* varied from work flow administration, land registry, statistics, taxation, registration of chemical formulae to decision support and management information systems. Virtually every application (30 of 36) was built using a client/server architecture with a GUI-client and a database server. A typical GUI-window from Adesso one of the larger systems in the survey is given below:

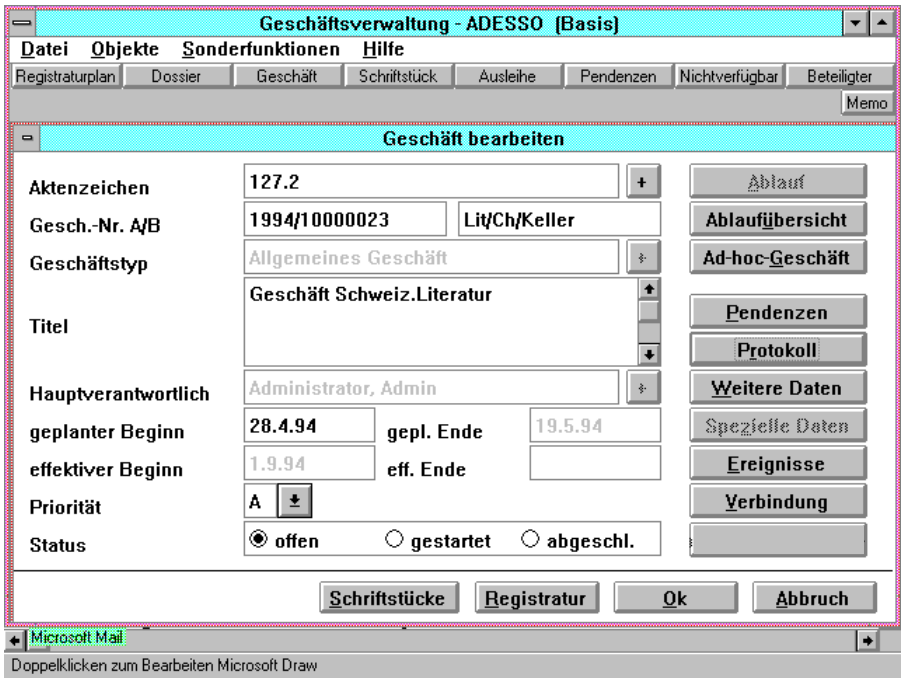

Figure 68: A Sample GUI-Window from One of the Surveyed Projects (Adesso)

4 projects explicitly had to deliver reusable frameworks. They are described in more detail below in order to make the analysis of productivity rates (cf. 4.3.3) more transparent.

The first project A1 had as its primary goal the development of a first usable version of a system for document registration and management of standard administrative procedures (such as legal initiatives, trials, applications, elections, etc.). Since this project was the first of a new generation of client/server applications, it also had a secondary goal to develop a framework dealing mainly with (1) the access of data stored in relational databases from an object-oriented client system, (2) the handling of complex interaction sequences involving several windows and (3) the construction of a programmable model for administrative procedures. Project A1 was immediately followed by a successor project A2 that produced a second release of the system using the framework and including some major enhancements, like incremental and keyword-based searches in the registry and ongoing procedures.

The second project B1 had the primary objective to produce a management information system (read only data access) for presenting information from several existing production systems (financial accounting, organisations and addresses, task management, documentation registry, etc.). At the same time it also developed a framework dealing with data access (but using stored procedures), data imports and exports, and presentation of data in flat and hierarchical browsers. It was followed by a project B2, using the framework, that dealt with acquiring and aggregating data for statistics of scholars (150’000 records).

Project C1 — in contrast — was explicitly and uniquely defined to produce a framework. This framework, too, had to accomplish (1) the task of accessing stored relational data from an object-oriented client, (2) an extensive tailoring of the MVC-framework for the search, display and modification of persistent data, and (3) the implementation of a state-transition interpreter that supports the state-based specification of the context-sensitive behaviour (dimming of impossible options and buttons) of a GUI application. It was both paralleled and followed by several projects C2, C3, C4, C5, C6, C7 and C8 using this framework to develop a taxation system.

Finally, project D — which at the time of writing is still ongoing — has the task of producing a framework for the administration of personal and organisational information together with its addresses (postal, electronic, etc.). This is a kind of system component that is commonly required in administrative systems, and it is intended that this component can be easily "plugged" into existing and future systems. Project data from the "plug in" processes, however, are not yet available.

4.2 Measurement Instrumentation

4.2.1 Instruments for Measuring the System Meter

We developed a scanning tool in C++ to measure the new SM metric. This tool operates with a simple command line interface. A tool was needed because the SM is very fine grained and thus hard - if not impossible - to measure manually. The tool was called **ma**, standing for **m**easure **a**nything. At the time being it is clearly a misleading - if not ridiculous - name. We, however, designed the tool to be very flexible. This flexibility is achieved by two means: 1) an extensive run-time parametrisation and 2) an easily extensible code architecture for incorporating new measures and other languages.

The **ma** utility scans electronically available so called *system descriptions* (sd), maps the description objects to the SMM and then computes any the desired metric(s) that are defined in the terms of the SMM. This is achieved in the following steps:

The first thing **ma** does, is reading in a so called *system description file* (sdf), whose name defines the name of the system. This file contains the names of all the other files (e.g. source code files) where the actual system descriptions may be found. Besides the file references, an sdf also may include statements to group the files into subsystems, versions, etc. as well as to categorise the files into language, library and project files.

Second, **ma** scans the given files. The first scanned files must be *language definition files* (ldf). Ldfs contain descriptions of languages in a **ma**-specific language called *language definition language* (ldl). An ldf also defines the mapping between a concrete language and the metamodel. After the ldf of some language is scanned, some optional library files may be scanned before finally the actual project-specific description files are scanned. In each of the files, except the ldfs which are always of category language, more refined groupings and categorisations may be contained.

Third, **ma** calculates the desired measures and prints them on standard output. The kinds of measures as well as groupings and some basic statistics (sum, average, median, etc.) are defined by passing command line parameters to the utility.

Among the various metrics currently measurable with **ma** the most popular are: LOC, η, V, ν, FP, NOC, DIT, WMC, RFC, CBO, LCOM, SM, COUP, COH. As its currently most severe limitation - due to our primal interest in early estimation - the currently supported languages are only 1) PRE, a language to denote preliminary analysis results, and 2) DOME, a language to denote domain analysis results. These two languages are immediately derived from the corresponding metamodels of subsections 3.2.6 and 3.2.7. Their syntax is given below. The restriction to these two languages allowed us to defer the implementation of name scopes to a next extension. We plan, however, (cf. chapter 5, outlook) to attack this next implementation phase soon in order to support quality assessment measurements in C++ and maybe other languages like C, Smalltalk or COBOL.

The syntaxes of the two languages PRE and DOME supported by **ma** are modelled as language method names, i.e. those languages are made up of description objects

themselves. In measurement practice this "finesse" is not noticed. PRE and DOME are used as any other (semi)formal language. Their syntaxes are given in the two tables below:

#	*PRE-Syntax*	*Explanation*
1	**;** 'text'	Single line comment
2	**Goal** 'name' ['complexity'] **.**	Introduction of an elemental goal. Complexity is 1 unless specified otherwise.
3	**Goal** 'name' **=** 'subgoal' **,**... **.**	Introduction of a higher level goal with the specified subgoals.
4	**Subject Area** 'name' 'number of entity types' **.**	Introduction of a subject area with an anticipated number of entity types in it.
5	**Subject Area** 'name' **isRuledBy** 'goal' **,**... **.**	This statement defines the goals to be valid for the subject area.

Table 13: PRE - A Language to Denote Preliminary Analysis Results

#	*DOME-Syntax*	*Explanation*
1	**;** 'text'	Single line comment
2	**Domain Subsystem** 'name' **=** 'use-case' **,** ... **.**	This statement introduces a grouping of use cases.
3	**Use Case** 'name' **isTriggeredBy** 'event' [**=** 'obj' **,** ...] **.**	This statement introduces a use case together with its triggering event and an optional precondition.. The precondition must reference the classes, associations, data and function types involved.
3	**Signal** ['name'] **of** 'use-case' | 'function-type' [((**from** **to**)) 'actor'] **=** 'obj' **,** ...) ... **.**	This statement introduces a signal of a use case or function type that optionally interacts with an external actor. It must reference the classes, associations, data and function types involved.
4	((**Domain Class** **Data Type**)) 'name' [**isSubTypeOf** 'et' **,**...] (**contains** ['n' **attr**] 'dt' | 'ft') **,**... **.**	Introduces a domain class which may be a subtype, i.e. having all the behaviour of zero one or more other classes. The attributes are only given as a summary: as repeatedly occurring data and function types. The number of occurrences is optionally given in n.
5	**Domain Association** ((**one** **many**)) 'et1' 'assoc-name' ((**one** **many**)) 'et2' **.**	Introduces a binary directed association between two classes with cardinalities indicated through the used keywords.
6	**Function Type** 'name' [**ofKind** 'kind' **,** ...] **.**	Introduces a function type which may be viewed as a repeated signal sequence occurring in several use cases. Use cases - instead of repeating the sequence - may then reference the function type in a single step. The basic function kinds may optionally be indicated, e.g. create, read, update, delete for a function type administrate.
7	**Consistency Rule** 'name' **=** 'obj' **,**... **.**	Introduces a consistency rule involving classes or other objects.
8	**State** 'name' **isSubstateOf** 'name' [**=** 'obj' **,**...] **.**	Introduces a state which is either a subtype of a domain class or another state. It may be defined using other objects (e.g. its associations).

9	**Transition** 'name' **startsAt** 'state1' **endsAt** 'state2' [= 'obj' **,**...] [**triggers** 'signal' **,**...] [**isTriggeredBy** 'signal' **,**...] **.**	Introduces a transition which links one state to another. It may be conditional using other objects which are involved in the condition. Also, it may trigger or be triggered by signals from the dynamic model.

Table 14: DOME - A Language to Denote Domain Analysis Results

In order to achieve more refined categorisations within system descriptions, the 'text' variable of comments is interpreted like an sdf categorisation entry when it starts with the keyword "ma_entry:". This allows the individual setting of the appropriate category, i.e. language, library and project, on a message to message basis. Conceptually an even more refined categorisation on the basis of single tokens may be achieved using an automated `diff` operation. We did not however, implement this advanced feature in **ma** yet.

The currently available version of **ma** is prototypical regarding several aspects: 1) it is reduced in functionality, 2) it was tested only by applying it on the sample base, and 3) it is implemented in a rather brute force and non optimised way. It may, however, be obtained by interested people (please, send email to the author).

4.2.2 Instruments for Measuring Function Points

As an integral part of the BIO software process (cf. 3.3.3) measurement tools for Function Point Analysis are provided. An EXCEL-spreadsheet - using the German language - is used for this task. It is made up of three tables according to the 3 FP-metamodel parts (cf. 2.2.4):

① the data part (German: "Datenteil"), measuring data elements and read-only data elements

② the process part (German: "Prozessteil"), measuring inputs, outputs and inquiries

③ the influence factor part (German: "Einflussfaktoren"), measuring the system-wide influence factor and the total system size in Function Points.

The data and process part tables furthermore allow to calculate system part percentages based on ratios of unadjusted Function Points. For some of the projects which were parts of bigger or more long-term development processes, the system size was calculated as the overall size multiplied with the corresponding percentage. The detailed calculations can be obtained from the author on request.

4.2.3 Instruments for Measuring the Effort and other Process Metrics

The effort numbers - a measure as critical as the system size measure for our purposes of effort estimation - were gathered out of company-internal reporting systems. As explained in 2.3.3, this effort only incidentally corresponded to the same development process instances from project to project. Using the process completeness measure, which is objectively measurable on the project's deliverables (cf. table 12 in 3.3.3), we therefore had to normalise the measured effort to a standard effort for a hypothetical project producing all the mandatory results of the BIO template process.

Effort, duration, the dynamic measures (cf. 2.3.4) as well as the EQF measure (cf. 3.3.1) were all documented on a project measurement form (German: "Projektmessblatt"). All these forms are included in appendix F of this thesis. The actual project names and companies, however, are not included. Measurement details can be obtained from the author less those not under corporate disclosure (usually the reporting details).

4.3 Results

4.3.1 The System Meter for Estimates after Preliminary Analysis

The survey's results are shown as a scatter chart:

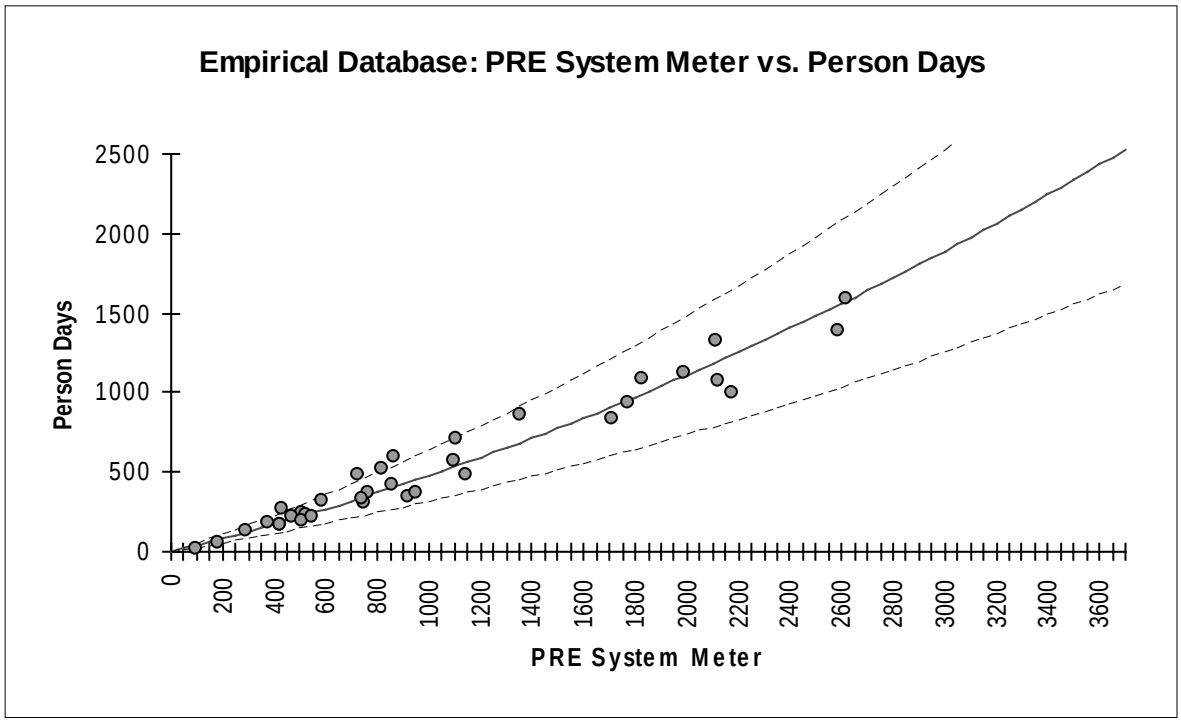

Figure 69: PRE-SM Survey Results: $A = 0.404 \cdot s + 0.0000753 \cdot s^2$, $dA = \pm 33\%$

As explained in 4.1.1.1, the quality of estimation methods should primarily be assessed by comparing their approximation biases. For the new System Meter method at preliminary analysis level dA is 33%. It cannot be directly compared to any other method's bias because we do not know of a metrics based estimation techniques for this early phase[33]. We may, however, compare the PRE-SM estimation model to the conventional Function Point method applicable at domain analysis (for details confer to 4.3.2). The bias for the conventional Function Point method is 20%. Thus, the new method delivers estimates of a lesser quality. But considering the second criterion, cost and time of estimation, there is a major difference: The effort spent until the end of a preliminary analysis typically is 5% of the total software process effort (cf. 3.3.3). In contrast, the Function Point method is applicable when typically 18% of the total effort is already spent. In summary, the new SMM estimation method can be characterised as yielding estimates of lesser quality by a factor of 1.65 with a cost reduction by a factor of 3 compared to the FPM.

One might be tempted to call for statistical analyses on subsets of interest, e.g. on all projects of company A versus those of company B, on all Smalltalk projects versus the C++ projects, etc. It would then be nice to say: "Aha, C++ projects are 2 times more productive than Smalltalk projects" or the reverse. But unfortunately, all the subsets that would make sense to analyse are too small with respect to the laws of statistics. As already stated, any set of samples should at least contain 30 independent data points.

[33] We do know of heuristical methods like the Aron method [IBM76], IBM's Phase-0-1 Guide [IBM75] and the author's KSM [MO91] which he developed in 1990, i.e. at an earlier stage of his estimation related developments.

We, however, could not resist to analyse the following question with respect to the System Meter method:

"How does the new method perform in non-database applications and/or special reuse situations (e.g. building a framework)?"

We separated the 7 projects being either a framework construction task and/or a non-database application to form an empirical database of its own. Complementarily, the rest of the projects (29) were also separately analysed. The approximation biases showed the following promising results:

The FPM correlates with a ±14% bias to the 29 database applications. This is substantially better than the correlation bias of ±20% for the total setting and is no surprise because the FPM was designed for this application type. The SMM still correlates with a ±33% bias to the 29 database applications. This may be viewed as a confirmation of the genericity of the new metric.

When looking at the other side, the 7 non-database projects, the FPM correlates with a larger bias of ±26%. This outcome is another hint for the fact that the FPM was designed and performs well for database applications and that for other application types FPM's performance sinks. The SMM, in contrast, correlates with a reduced ±22% bias. This actually means that the three times more cost efficient SMM outperformed the FPM in these 7 projects. This is, as already stated, too few to be statistically sound. However, the author considers this, together with other properties of the new method, as a promise for the future of the SMM.

Detailed Values

The detailed values of the 36 projects are given in the following list. The 7 non-database projects are marked with an *.

#	*Project-Id, Date (yy.mm)*	*PRE-SM*	*Person Days*
1.	*X1, 87.12	506	256
2.	X2, 90.07	816	528
3.	X3, 90.11	2120	1084
4.	X4, 92.04	425	274
5.	X5, 92.07	762	371
6.	Y1, 93.04	1098	716
7.	Y2, 93.06	720	491
8.	Y3, 93.08	1095	579
9.	B2, 93.11	746	318
10.	A1, 93.12	2172	1009
11.	Y4, 93.12	2616	1600
12.	Y5, 94.02	92	28
13.	Y6, 94.04	2582	1389
14.	A2, 94.06	421	176
15.	*B1, 94.07	521	239
16.	Y7, 94.10	1986	1126
17.	*C1, 91.11	1143	487
18.	*Z1, 94.12	584	327
19.	Y8, 95.01	2109	1326
20.	Y9, 95.02	859	605
21.	T1, 95.02	919	347

22.	Y10, 95.04	1346	867
23.	C2, 95.05	290	133
24.	C3, 95.05	857	426
25.	C4, 95.05	943	377
26.	T2, 95.05	1820	1095
27.	C5, 95.07	739	339
28.	T3, 95.08	1769	942
29.	C6, 95.09	1703	842
30.	*Y11, 95.10	178	65
31.	*Y12, 95.10	373	187
32.	*Z2, 95.11	468	229
33.	C7, 95.11	415	180
34.	C8, 95.11	506	200
35.	T4, 95.11	3621	2647
36.	D, 95.11	540	232

Table 15: PRE-SM and PD Values

Chi-Square-Test for Normal Distribution

We tested the distribution against the hypothesis that it is normally or Gaussian distributed. The deviations of the projects are classified into 6 ranges. The number of deviations falling into each range is summed up and compared - using the Chi-Square test measure - to a model distribution. For 6 ranges the sum of the test measures must not exceed the value of 7.6 in order to validate the hypothesis:

Standard Deviation s = 9.9%			***Ranges***					
#	***Project, Date (yy.mm)***	***deviation***	***-s***	***-s/2***	***0***	***s/2***	***s***	***∞***
1.	*X1, 87.12	0.99%	0	0	0	1	0	0
2.	X2, 90.07	5.86%	0	0	0	0	1	0
3.	X3, 90.11	-5.90%	0	1	0	0	0	0
4.	X4, 92.04	12.59%	0	0	0	0	0	1
5.	X5, 92.07	-18.48%	1	0	0	0	0	0
6.	Y1, 93.04	7.60%	0	0	0	0	1	0
7.	Y2, 93.06	4.41%	0	0	0	1	0	0
8.	Y3, 93.08	2.66%	0	0	0	1	0	0
9.	B2, 93.11	-2.19%	0	0	1	0	0	0
10.	A1, 93.12	4.68%	0	0	0	1	0	0
11.	Y4, 93.12	-6.13%	0	1	0	0	0	0
12.	Y5, 94.02	-6.03%	0	1	0	0	0	0
13.	Y6, 94.04	-2.37%	0	0	1	0	0	0
14.	A2, 94.06	1.06%	0	0	0	1	0	0
15.	*B1, 94.07	0.70%	0	0	0	1	0	0
16.	Y7, 94.10	0.94%	0	0	0	1	0	0
17.	*C1, 91.11	23.71%	0	0	0	0	0	1
18.	*Z1, 94.12	29.49%	0	0	0	0	0	1
19.	Y8, 95.01	9.11%	0	0	0	0	1	0
20.	Y9, 95.02	-0.57%	0	0	1	0	0	0
21.	T1, 95.02	-9.73%	0	1	0	0	0	0
22.	Y10, 95.04	-1.08%	0	0	1	0	0	0
23.	C2, 95.05	-2.00%	0	0	1	0	0	0

24.	C3, 95.05	-10.50%	1	0	0	0	0	0
25.	C4, 95.05	14.16%	0	0	0	0	0	1
26.	T2, 95.05	-10.87%	1	0	0	0	0	0
27.	C5, 95.07	-10.34%	1	0	0	0	0	0
28.	T3, 95.08	0.27%	0	0	0	1	0	0
29.	C6, 95.09	0.06%	0	0	0	1	0	0
30.	*Y11, 95.10	-12.27%	1	0	0	0	0	0
31.	*Y12, 95.10	16.79%	0	0	0	0	0	1
32.	*Z2, 95.11	6.40%	0	0	0	0	1	0
33.	C7, 95.11	-5.81%	0	1	0	0	0	0
34.	C8, 95.11	-5.65%	0	1	0	0	0	0
35.	T4, 95.11	7.21%	0	0	0	0	1	0
36.	D, 95.11	-3.61%	0	0	1	0	0	0
	Total		5	6	6	10	5	5
	Model Distribution		6	6	7	7	6	6
	Chi-Square Values		.17	0	.14	1.3	.17	.17

Table 16: Chi-Square-Test of Effort Distribution

The sum of the values is 1.94 which is below 7.6. We may therefore soundly assume a normal distribution for the efforts. This test was only conducted at the preliminary analysis level assuming that the hypothesis is also valid for the other levels, that do not differ with respect to efforts.

4.2.2 The System Meter for Estimates after Domain Analysis

The survey's results are shown as a scatter chart:

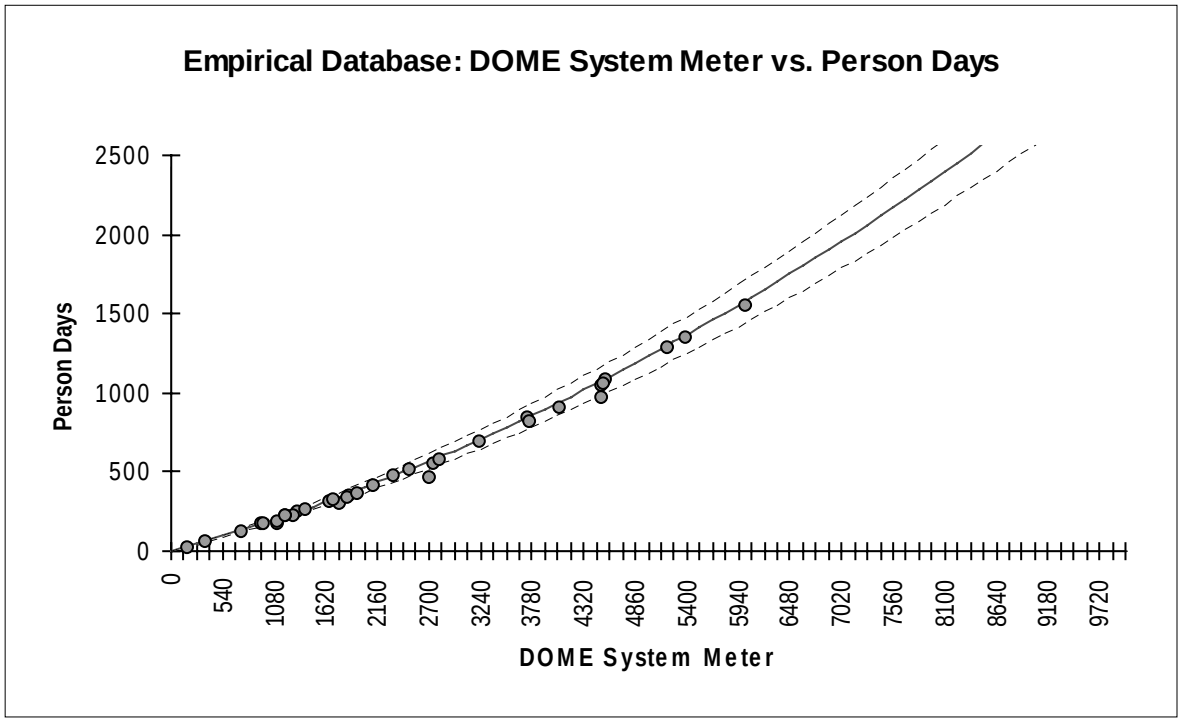

Figure 70: DOME-SM Survey Results: $A = 0.1675 \cdot s + 0.0000158 \cdot s^2$, $dA = \pm 9\%$

The approximation bias here is considerably lower than the one for the System Meter after preliminary analysis. It is also lower than the bias of the Function Point metric which can be applied at the same level of modelling. Refer to the end of this subsection for a statistical comparison - using the Wilcoxon-signed-ranksum-test - of the two methods.

Detailed Values

The detailed values of the 36 projects are given in the following list.

#	Project-Id, Date (yy.mm)	DOME-SM	Person Days
1.	*X1, 87.12	1323	256
2.	X2, 90.07	2493	528
3.	X3, 90.11	4507	1084
4.	X4, 92.04	1398	274
5.	X5, 92.07	1866	371
6.	Y1, 93.04	3214	716
7.	Y2, 93.06	2331	491
8.	Y3, 93.08	2750	579
9.	B2, 93.11	1753	318
10.	A1, 93.12	4507	1009
11.	Y4, 93.12	5998	1600
12.	Y5, 94.02	163	28
13.	Y6, 94.04	5374	1389
14.	A2, 94.06	948	176
15.	*B1, 94.07	1286	239
16.	Y7, 94.10	4539	1126
17.	*C1, 91.11	2693	487
18.	*Z1, 94.12	1656	327
19.	Y8, 95.01	5189	1326
20.	Y9, 95.02	2810	605
21.	T1, 95.02	1835	347
22.	Y10, 95.04	3714	867
23.	C2, 95.05	731	133
24.	C3, 95.05	2112	426
25.	C4, 95.05	1947	377
26.	T2, 95.05	4523	1095
27.	C5, 95.07	1685	339
28.	T3, 95.08	4063	942
29.	C6, 95.09	3742	842
30.	*Y11, 95.10	352	65
31.	*Y12, 95.10	1109	187
32.	*Z2, 95.11	1198	229
33.	C7, 95.11	958	180
34.	C8, 95.11	1102	200
35.	T4, 95.11	8591	2647
36.	D, 95.11	1186	232

Table 17: DOME System Meter Measurements

Function Point Method Results

Again the results are shown in form of a scatter chart first:

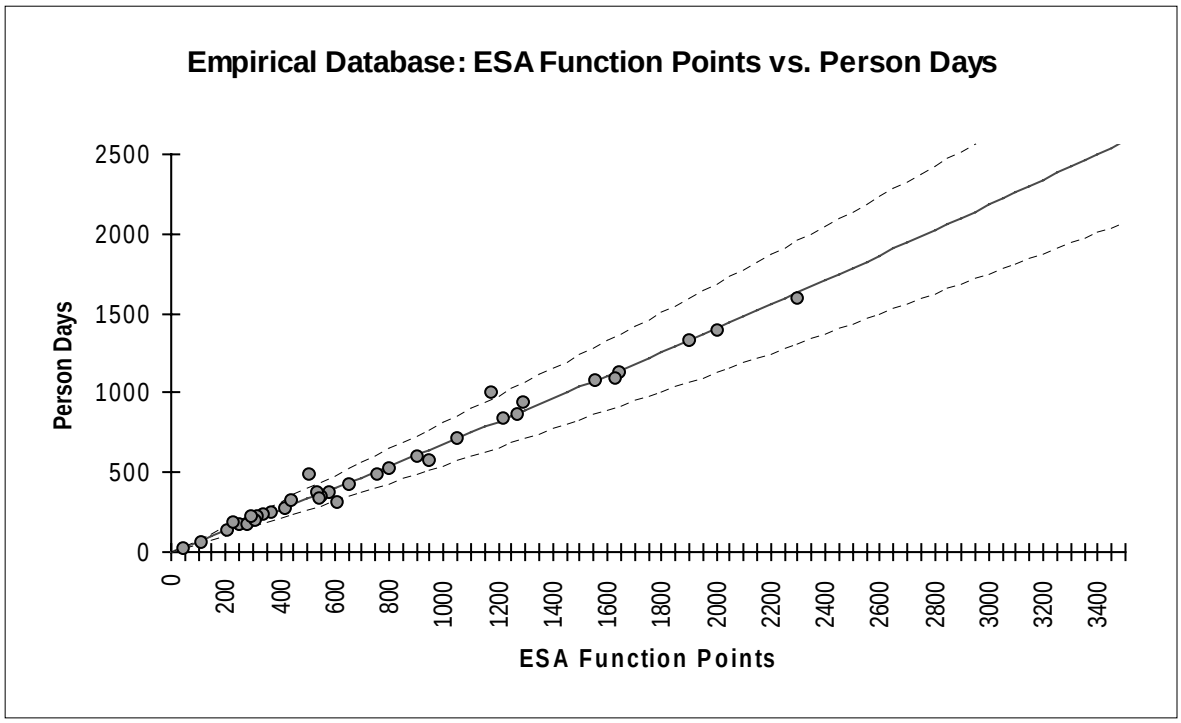

Figure 71: FPM survey results: $A = 0.656 \cdot s + 0.000235 \cdot s^2$, $dA = \pm 20\%$

The detailed values are:

#	*Project-Id, Date (yy.mm)*	*FP*	*Person Days*
1.	*X1, 87.12	370	256
2.	X2, 90.07	800	528
3.	X3, 90.11	1556	1084
4.	X4, 92.04	420	274
5.	X5, 92.07	579	371
6.	Y1, 93.04	1050	716
7.	Y2, 93.06	753	491
8.	Y3, 93.08	946	579
9.	B2, 93.11	610	318
10.	A1, 93.12	1173	1009
11.	Y4, 93.12	2300	1600
12.	Y5, 94.02	45	28
13.	Y6, 94.04	2000	1389
14.	A2, 94.06	251	176
15.	*B1, 94.07	341	239
16.	Y7, 94.10	1640	1126
17.	*C1, 91.11	505	487
18.	*Z1, 94.12	440	327
19.	Y8, 95.01	1897	1326
20.	Y9, 95.02	900	605
21.	T1, 95.02	550	347
22.	Y10, 95.04	1269	867

23.	C2, 95.05	205	133
24.	C3, 95.05	651	426
25.	C4, 95.05	537	377
26.	T2, 95.05	1630	1095
27.	C5, 95.07	540	339
28.	T3, 95.08	1290	942
29.	C6, 95.09	1215	842
30.	*Y11, 95.10	109	65
31.	*Y12, 95.10	230	187
32.	*Z2, 95.11	315	229
33.	C7, 95.11	280	180
34.	C8, 95.11	308	200
35.	T4, 95.11	3462	2647
36.	D, 95.11	295	232

Table 18: FP and PD Values

Testing the Correlation Differences

According to the lower mean bias of the DOME-SM for effort prediction we want to test the hypothesis that the deviation is significantly lower than that of the FP metric. The hypothesis is that the mean percentage deviation is lower. We validate this hypothesis using the Wilcoxon-signed-ranksum test at a 95%-confidence level. The pairwise deviations, sign of the difference and signed ranks are:

Project #	***FP*** deviation	***DOME-SM*** deviation	***absolute bias difference***	***sign***	***signed rank***
1	2.69%	2.59%	0.09%	+	2
2	1.51%	2.27%	0.77%	-	-12
3	0.39%	0.69%	0.31%	-	-7
4	1.38%	3.23%	1.85%	-	-19
5	2.91%	0.89%	2.03%	+	21
6	0.10%	1.97%	1.87%	-	-20
7	2.19%	2.96%	0.76%	-	-11
8	6.64%	0.24%	6.41%	+	32
9	14.93%	7.65%	7.28%	+	33
10	17.64%	6.69%	10.95%	+	35
11	1.46%	1.61%	0.15%	-	-5
12	3.59%	0.71%	2.88%	+	27
13	0.87%	2.28%	1.41%	-	-18
14	3.89%	1.66%	2.23%	+	23
15	3.65%	1.10%	2.55%	+	24
16	0.82%	3.51%	2.70%	-	-25
17	29.62%	16.20%	13.42%	+	36
18	7.65%	1.89%	5.77%	+	31
19	0.18%	2.31%	2.13%	-	-22
20	0.52%	1.54%	1.02%	-	-14
21	3.83%	3.95%	0.12%	-	-4
22	0.28%	3.06%	2.78%	-	-26

23	1.25%	1.53%	0.28%	-	-6
24	1.72%	0.37%	1.35%	+	17
25	3.31%	2.43%	0.88%	+	13
26	2.27%	1.24%	1.04%	+	15
27	4.12%	3.47%	0.65%	+	9
28	4.37%	0.01%	4.36%	+	29
29	0.82%	0.77%	0.05%	+	1
30	6.28%	6.16%	0.12%	+	3
31	15.13%	9.78%	5.35%	+	30
32	6.32%	2.43%	3.89%	+	28
33	2.01%	2.75%	0.74%	-	-10
34	1.42%	1.93%	0.51%	-	-8
35	2.71%	1.50%	1.21%	+	16
36	12.32%	4.75%	7.56%	+	34
			TOTAL		252

Table 19: Percentage Biases, Bias Differences, Signs and Signed Ranks of FP and DOME-SM

The 95%-quantile for of the ranksum of 36 samples is 209. Our observed ranksum is 252 which is above this quantile. We may therefore conclude that the average bias of FP based estimates is significantly higher than the average bias of DOME-SM based estimates.

4.2.3 The System Meter for Productivity Assessments in Reuse Situations

For project A1 we determined a productivity of 1.2 FP/Person Day (PD). In the follow-up project A2 the productivity rose to 1.5 FP/PD mainly due to the framework reuse, because all other influencing factors (project team, tools, methodology, domain area, infrastructure) remained constant. Expressed in percentages, this is an increase of 25%. When looking at the System Meter numbers we observe a productivity of 2.26 SM/PD for A1 and 2.52 SM/PD for A2. This means only a 12% rise. As the System Meter values decrease (cf. 3.2.1.2) when reusable parts are incorporated into a system also the productivity of a project that reuses components will decrease.

This effect of the System Meter method may seem undesirable, because we want to encourage reuse. On the other hand this effect can also be viewed at as a positive stabilising property of the new method which improves the quality of the estimates. When productivity rates are more constant, derived estimates will be more accurate (as argued for in [DM82]). In fact, this is a more consistent and reproducible way of understanding the effects of frameworks on productivity: project teams do not magically become more productive through the use of frameworks (as the FP method might have us conclude), but rather the size of the system to be designed and implemented is dramatically reduced. If we can accurately measure the size of the reduced system, then we can rely on our previous productivity rates to help us better estimate the actual cost of the project.

In case an organisation would like to encourage reuse by setting productivity goals, the System Meter could alternatively be applied without its reuse modelling component, thus rating all system description parts as "project” parts. Values obtained with this System Meter variant are denoted as "flat” System Meters. Those flat SMs were prototypically measured in project A2. (In project A1 the flat SMs equalled the SMs because there were no reusable framework components available at the start of the project). A2’s flat productivity rate was calculated as 3.12 SM/PD which means a 38% increase. These

"magical" improvements in productivity, however, can only be observed after a project has been completed, and so are uninteresting from the perspective of predictive estimation.

Another independent case of framework construction and reuse was observed in the project pair B1 and B2. Function Point analysis yielded productivities of 1.5 and 2.0 respectively (a 33% increase) whereas the System Meter method yielded 2.30 and 2.48 rates (an 8% increase). Actually, more components could be reused without modification in project B2, which explains both the higher FP increase and the more constant behaviour of the SM values. Another difference between B2 and A2 was the tighter schedule of A2. This required the formation of a bigger team, which undoubtedly contributed to the lower increase in productivity of A2 with respect to B2 (in accordance with observations described by Putnam [PU76]). This "team size" effect, also known as the "Mythical Man Month" syndrome, of course, has nothing to do with frameworks or reuse, but is a general rule of management of complex development processes.

In the setting of projects C1 — C8, we observed the following series of productivity rates:

Project	***FP productivity***	***increase vs. C1***	***SM productivity***	***increase vs. C1***
C1	1.1	n.a.	2.47	n.a.
C2	1.6	45%	2.30	-7%
C3	1.6	45%	2.12	-14%
C4	1.5	36%	2.64	7%
C5	1.7	55%	2.30	-7%
C6	1.5	36%	2.13	-14%
C7	1.6	45%	2.42	-2%
C8	1.6	45%	2.69	9%

Table 20: Productivity Rates and Percentage Deltas for Projects C1-C (C1 = Framework Project)

The observations we made concerning the A and B projects are even more pronounced here. When measured using the System Meter method, the projects that reused the framework (C2-C8) exhibited even lower productivity than the framework construction project C1.

In contrast to projects A1 and B1, where the framework construction was only a secondary goal after the development of a first version of an application software system, project C1 was entirely dedicated to establishing a framework. This explains why C1 rated notably low in the FP analysis, and consequently, why the follow-up projects C2-C8 exhibited unusually high percentage FP productivity increases. The System Meter method, on the contrary, yielded rather uniformly distributed productivity rates. Due to the heavy framework reuse, C2-C8 rated rather low. Notable exceptions are projects C4 and C8 which developed substantial non framework supported subsystems (C4 dealt with the development of a spreadsheet-like taxation calculation model; C8 encompassed a few non standard reports).

The quantitative analysis of Project D, finally, could not cover productivity increase rates, because no follow-up project exists yet for D's framework. We therefore concentrated our analysis on the estimating power of the two methods: D yielded 295 FPs and 540 SMs. When applying the currently regressed coefficients out of the 36 project empirical database, we would have estimated 186 PDs (bias: ±20%) using the Function Points and we actually

did estimate 229 PDs (bias: ±33%) using the System Meters. The effective effort spent is 220 PDs, which is substantially closer to the SM estimate than to the FP estimate. This, of course, may not be interpreted as empirical evidence of the superiority of the new method over the established FP method. On the other hand, the new method is not rejected by the empirical results, but rather, we are encouraged to continue our investigations.

4.2.4 Measuring the Effects of Team Size and Acceleration on Productivity

In 4 projects we observed considerably higher efforts than expected from system size - be it Function Points or System Meters - alone. Productivity rates were accordingly below average. We suspected this was due to the tight schedules and therefore larger team sizes observable for those projects. Thus, we applied the dynamic cost modelling techniques [DM82] [PU80] (cf. 2.3.4) in our analysis.

First we calculated the acceleration factor A as defined in 2.3.4. For the estimated duration we used the estimate minus half the 95%-confidence-level bias in order to cope with the considerable uncertainty. For example in project Y3, we calculated the estimated duration as (220 - 220 * 25%/2) = 192.5 elapsed work days. Then we calculated the effort inflation factor and put the resulting 4 value pairs into one of our regression spreadsheets. Theory and literature suggests 0 for the constant and linear regressed coefficients and 1 for the quadratic, thus the dependency would be purely quadratic. In our very small sample set we observed the following correlation:

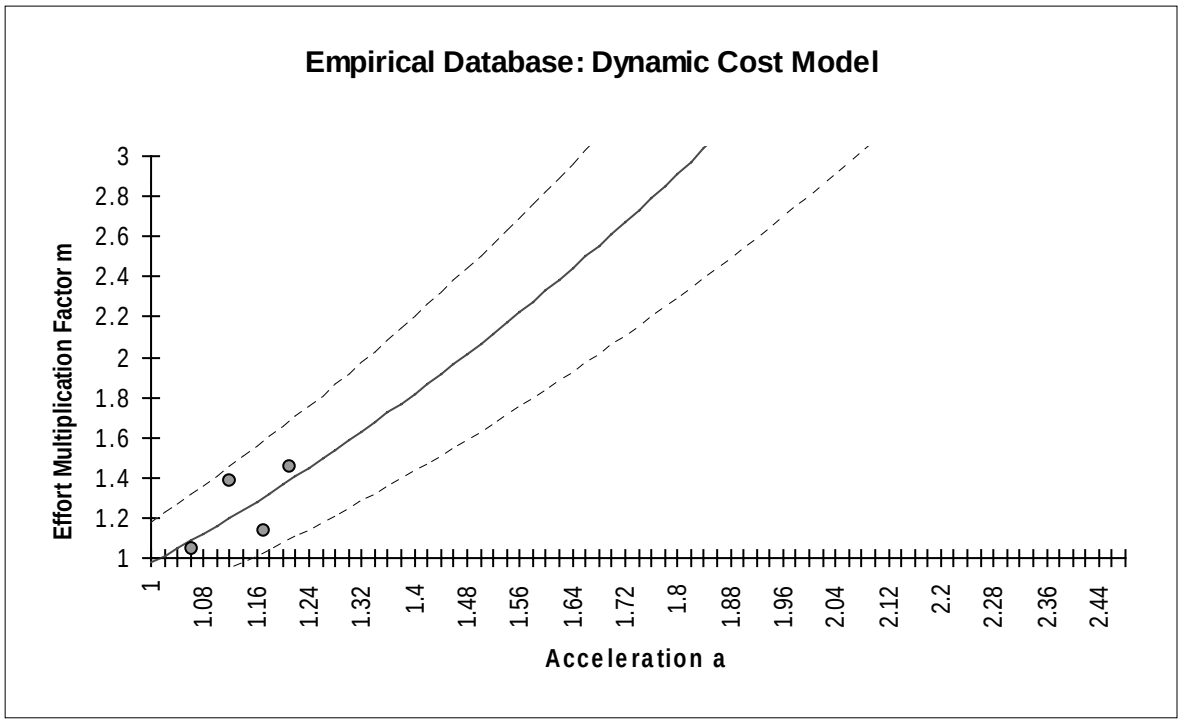

Figure 72: Correlation of Acceleration Factor and Effort Inflation

The first observation we made was that there actually seems to be a reasonable correlation. The correlation bias dA was ±21%. The regressed coefficients, however, did not exactly show the suggested behaviour. We observed a correlation of:

$$I = 0.202 \cdot A + 0.792 \cdot A^2$$

This may seem totally different from the purely quadratic formula. On the other hand, when considering the very small sample size and the relatively high correlation bias, the observations do not severely contradict the suggested behaviour. With respect to the small

number of samples we did not even try to statistically prove the significance of the differences.

4.3.5 Measuring Estimation Quality for Different Estimation Techniques

The analysis accomplished in sections 4.3.1-4 makes use of the fact that the estimation techniques are metrics based. The statistical calculations may therefore be done entirely after project completion, even if the measurement technique did not exist when the project was executed. We, however, also wanted to compare non-metric based estimation techniques. For this we calculated the EQF measure (cf. 3.3.1) for the ten projects Y1, B2, Y4, Y5, Y6, B1, Z1, Y8, Y11 and Z2. In all projects we were closely involved and thus were able to obtain the estimation history. Y1 and B2, whose values are diagrammed as white squares in Figure 73, did not use an explicit or documented estimation technique, their EQF values were - even though not disastrous - considerably lower than those using metrics based techniques. Typical disaster project values are reported by DeMarco [DM83] as being below 50%. Projects Y4, Y5, Y6, B1, Z1, Y8 and Y11, diagrammed as grey squares, used the Function Point method. They achieved very good to excellent results. One may also observe an improvement over time (even though not monotonic). This effect is most probably due to the growing and more specific empirical database that was used for obtaining the estimates. Finally, in project Z2, diagrammed black, we can report the EQF from the first application of the new System Meter method (PRE SM) to a real project. Even though not quite reaching the quality of FP estimates, its performance was very good. Be aware that the PRE SM can be applied with considerably rougher input (cf. 3.2.8 and 3.2.9) than the FP method.

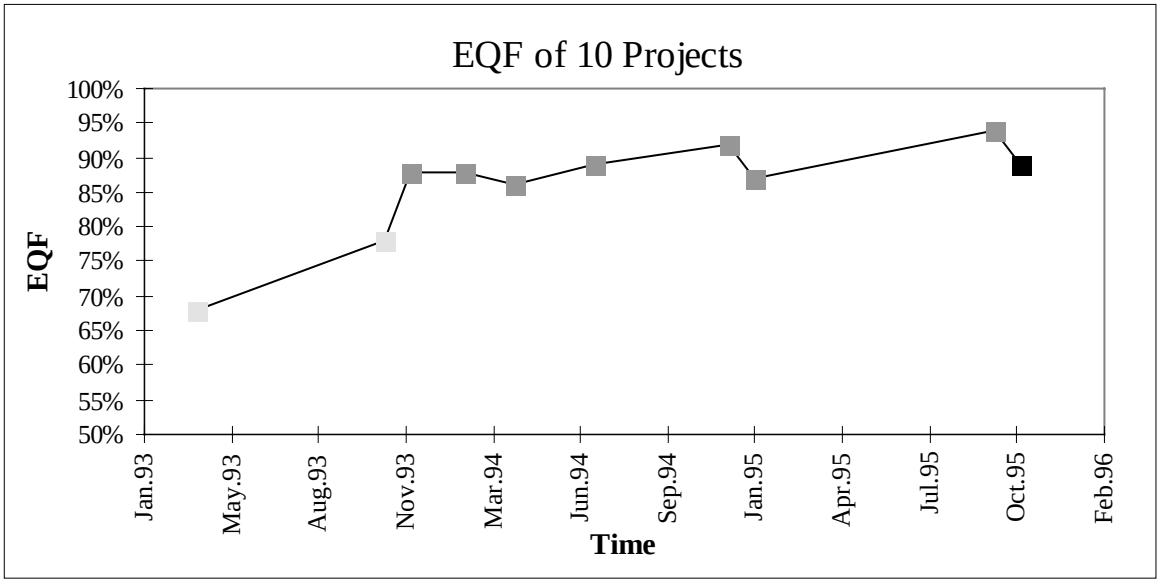

Figure 73: EQF Values versus Time

Regarding the small number of samples for the EQF measure, we do not attempt to perform any statistical analysis. Metrics based estimation and - to a lesser degree - the System Meter method are, however, intuitively validated by our observations.

At the time of writing, the PRE and DOME System Meter methods are steadily replacing the FP method for estimation at one major industrial site. More EQF values are thus expected to be measurable soon.

5 Conclusions, Outlook

5.1 The System Meter as a Basis for Estimation and Productivity Analysis

The new System Meter estimation method proposed in our work proved to be useful for estimates after preliminary system modelling and also after domain analysis. The results it yielded at the preliminary analysis stage were not quite as good as those of the traditional Function Point method (FPM), which - as its drawback - may only be applied after the more costly modelling phase of domain analysis. Thus, when comparing both the quality and cost of estimation, the new method delivers very useful results. At the domain analysis stage, the System Meter correlates significantly better to overall development effort than Function Points. These conclusions were drawn from comparing the two methods in a sample set of 36 industrial and university projects. The technology used was mainly client/server based object-oriented development using Smalltalk or C++, but also 4th-generation-language projects were included. System Meters can be applied for measuring conventional and object-oriented systems, as can Function Points.

Observations were made allowing the presumption that the System Meter method (SMM) is less bound to the area of database applications than the FPM. Also, special reuse aspects (e.g. the task of constructing a framework) seem to be better supported by the SMM. Those assumptions are supported by analysing, a set of 7 projects, either accomplishing a framework construction task and/or being a non-database application, where the SMM performed better than the FPM. The 7 samples, though, are not enough for statistical evidence.

The SMM is not a revolutionary estimating method, only the underlying metric, the System Meter, is new. It is therefore compatible to proven techniques of estimation (cf. [BOE81] or [TDM82]). Because the underlying metric captures the ideas of software size and complexity, it can not only be used for estimation activities but also for productivity analysis and control. It may for example appear as the numerator in a productivity measure like System Meter / Person Day. The application of the SMM is, furthermore, not restricted to object technology. Since the underlying metric is generic, it can be applied in virtually any context.

The System Meter is a new approach to quantitative software metrics based on a rich metamodel for system description objects rather than on the better-known, but more simplistic Function Point approach. The System Meter has been developed to address the effects of Object Technology on the way that software projects are structured and composed. The principal idea behind the System Meter is to distinguish between the "internal” and "external” sizes of objects, and thus to only measure complexity that has a real impact on the size of the system to be implemented. The System Meter essentially is a token count of the names used for modelling or describing a system. This straightforward definition yields sound mathematical properties such as the allowance of any statistical operation and of calculating ratios. Furthermore, the System Meter can homogeneously be applied to the various artefacts of software engineering such as classes, methods, variables, etc.

At the time of writing, the new method is beginning to be used in industrial development projects. Previously, the practicability of the method was tested in a research project, the development of the measurement tools used in our work. The quality of the estimates thus delivered was above the average of those of the FPM. We expect our empirical databases to grow from the 36 samples taken during the last 3 years. Estimation quality will most

probably be raised as soon as large enough databases, specific to distinct development environments (technology, application domain, infrastructure, involved people) will be available.

5.2 Outlook: The System Meter as a Basis for Quality Metrics

In our future research we intend to use the System Meter as a base measure for more refined purposes, comparable to the use of the "normal" meter for measuring distances. The System Meter has the required formal properties and has proved to be of use for the purpose of estimation. Just like construction engineers use the "normal" meter for estimation (using the m^3) and also for quality metrics like the solidity of ceilings (using Newton / m^2), we intend to use the System Meter for metrics of software quality (cf. 1.7).

The quality of software can be assessed with a static analysis of system descriptions alone or with a compilation of empirical outcomes, e.g. measuring error rates. The first category of quality metrics are called inherent quality metrics. Inherent quality is mainly determined by the two aspects of coupling and cohesion [YO79]. As shown in theoretical and empirical studies by Brian Henderson-Sellers (cf. chapter 7 "Cognitive Complexity Model" in [HS96]), two mental processes, tracing and chunking, determine the psychological (or cognitive) complexity of descriptions for the reader and writer. Chunking encompasses the formation of mental maps of the description entities (classes, methods, statement blocks, statements, etc.) and tracing is the activity of searching for dependencies among the chunks. Those two processes of understanding a certain piece of software may be diagrammed in an XY-chart as follows:

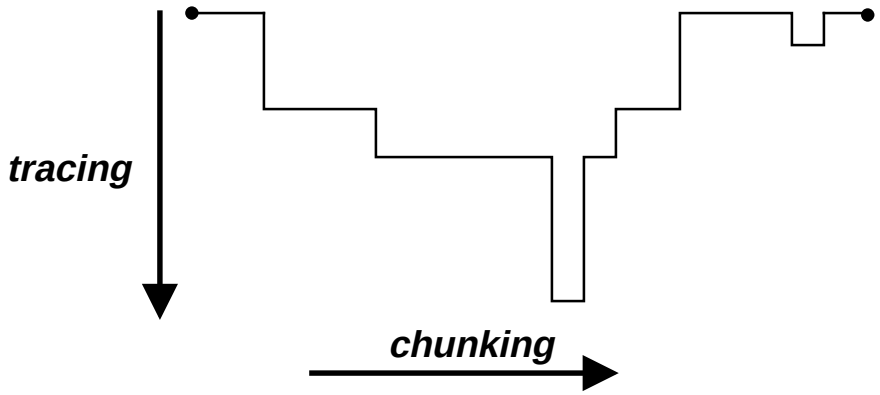

Figure 74: The Comprehension Landscape of a Piece of Code (adapted from [HS96])

Our idea is it to correlate chunking to the internal quality aspect of cohesion and tracing to coupling. To accomplish this, we need to measure those two internal quality aspects. For this we may exploit the System Meter as follows:

Coupling/Isolation

The degree of coupling for an object X is directly dependent on the number of other objects its implementation depends (or may depend) on. For low coupling this means: A) the implementation set should be small, B) the objects that could influence the implementation (write scope) should be few, C) the objects that read X should be few, because they might want to change X, and D) the read scope should be as small as possible. The coupling measure could then be defined as the average of the ratios of the sizes of the four sets just described and the total system size. This idea is shown in the following figure:

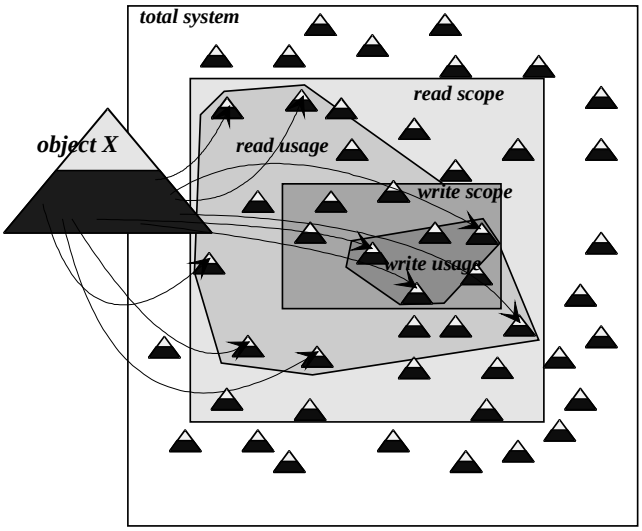

Figure 75: Coupling of an Object X with Other Objects in the System

Cohesion

The cohesion, in turn, may be expressed by two concepts:

A) The external coherence: definable as the number of different classes you have to know in order to be able to apply some object X relative to the maximum number of such classes. The "classes you have to know" are also called neighbour classes of an object X. They are e.g. the type of X, the types of the formal parameters if X is a method, the supertype if X is an inherited class, etc.

B) The internal coherence: definable by the degree to which "the law of Demeter" is obeyed that may be summed up as "define some object with its own components" [LIE96]. This law is planned to be extended by the author to "define some object with its own or its neighbour components" because the original law seems to be too rigid for generic software components such as "methods" that do not belong to a class.

The concepts of external and internal cohesion are visualised in the following figure:

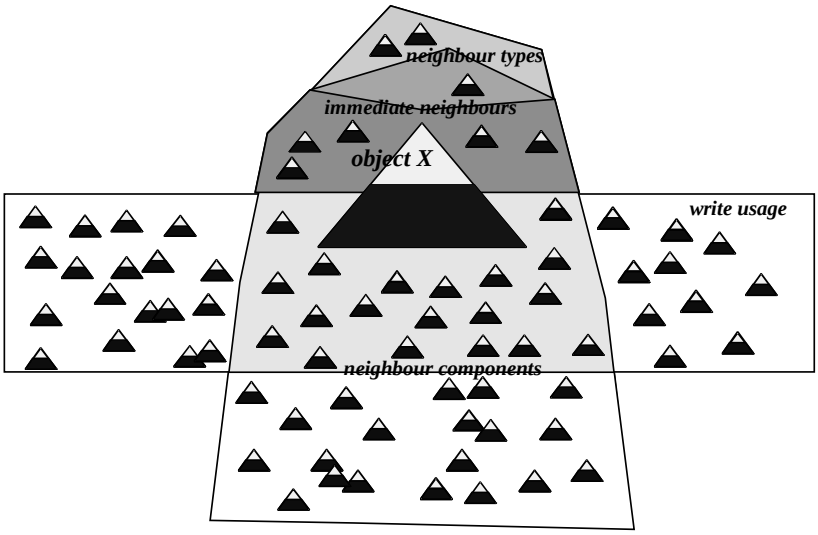

Figure 76: Internal and External Cohesion of an Object

Formal definitions for these measurement ideas will have to be elaborated.

Empirical measures might profit from the System Meter, too. For example the empirical quality (as defined in 1.7) may soundly be measured using the System Meter, because it is defined for whole systems as well as for system fractions (cf. 3.2.1.3).

Summary

The starting point of the present Ph.D. thesis was our interest in the non-trivial but important task of effort estimation for software development projects. Before starting the Ph.D., we had had several years of industrial practice with various estimation methods.

The research started in 1993 and began with a survey and analysis of existing solutions. We concluded that the most effective and sound technique available was the Function Point Method (A. J. Albrecht). Its key strategy is to measure system size on a system model - just as architects measure building cubes on their plans - and then derive the effort from that size by exploiting empirical correlations between the two measures. Using the Function Point measure in the context of modern object-oriented development environments, however, has several disadvantages: ① it does not take reusable components into account and ② it is measurable only relatively late in the software development process, because it requires a detailed model of user requirements. Other disadvantages are that Function Points are not well-defined for non-database applications, that they make use of historical parameters which do not contribute to estimation quality, and that the use of system-wide influence factors makes the measure formally unsound to use for certain statistical operations.

Based on an analysis of what the items of object-oriented software development exactly are, the metamodels of object-oriented systems, we tried to formulate an improved measure. The basic estimation strategy of measuring and exploiting empirical correlations was retained. The new measure, called System Meter, consists of two parts: ① the external size of software objects and ② the internal size of software objects. The external size of a software object, for example a class, is determined by the complexity of the object name or signature. We proposed counting the tokens within the name to capture this aspect. The internal size makes use of the dependencies that exist between software objects. It sums up the external sizes of all the other objects an object is dependent on. In order to take into account the reusable parts, reused objects are only counted for the external size when summing up the overall size of a system, i.e. a set of objects. Project specific objects are counted for both the external and internal sizes.

In a field study of over 30 projects in industry and academia, we compared the new metric to Function Points with respect to effort correlation. The System Meter is applicable on two levels of system model granularity, the more coarse preliminary analysis model and the finer domain analysis model. Function Points, in contrast, are only measurable at the level of domain analysis. We found that the first variant of the System Meter, the PRE System Meter, yields estimates of lesser quality than Function Points, but is applicable with far less effort and therefore supports faster development cycles. The second System Meter variant, the DOME System Meter, was directly compared to the Function Point measure. A statistical test on our sample base supported the hypothesis that effort estimates using the DOME System Meter show significantly less bias than using Function Points. The sample base included projects using various object-oriented and 4th-generation-language (4GL) environments (Smalltalk, C++ and others).

While the new estimation methods are beginning to be used in industrial practice at the time of writing, we look forward to investigating our proposed measure, the System Meter, in more applications. Just as the "normal" meter cannot only be used for measuring geographical distances but also to define quality measures, like the density of a material in kg/m^3, we intend to define and empirically validate quality metrics for software using the System Meter.

Appendices

A Index

F

G

H

I

L

S

T

U

V

W

Z

B Literature

[AL79] Albrecht AJ, MEASURING APPLICATION DEVELOPMENT PRODUCTIVITY, in Proc. IBM Applications Develop. Symp. Monterey, CA, Oct. 14-17, 1979, GUIDE Int. and SHARE, Inc., IBM Corp., p. 83ff.

[AM94] Ambler SW, IN SEARCH OF A GENERIC SDLC FOR OBJECT SYSTEMS, Object Magazine 4(6), pp 76-78, 1994

[AMI92] The ami consortium c/o South Bank Polytechnic, London U.K., AMI: APPLICATION OF METRICS IN INDUSTRY, 1992

[BA88] Basili VR, Rombach D, THE TAME PROJECT: TOWARDS IMPROVEMENT ORIENTED SOFTWARE ENVIRONMENTS, IEEE TA on SE Vol. 14 NO. 6, pp. 758-773, June 1994

[BI94] Bieman JM, Ott LM, MEASURING FUNCTIONAL COHESION, IEEE TA on SE Vol. 20, No. 8, pp 644-657, 1994

[BO91] Booch G, OBJECT-ORIENTED DESIGN, Benjamin Cummings, Redwood City, CA, 1991

[BOE81] Boehm BW, SOFTWARE ENGINEERING ECONOMICS, Prentice-Hall, Englewood Cliffs, N.J., 1981

[BOE86] Boehm BW, A SPIRAL MODEL FOR SOFTWARE DEVELOPMENT AND ENHANCEMENT, ACM Software Engineering Notes, 11(4), pp. 14-24, 1986

[BOE95] Boehm BW, et al., AN OVERVIEW OF THE COCOMO 2.0 SOFTWARE COST MODEL, Technical Report, University of Southern California, 1995

[BR75] Brooks FP, THE MYTHICAL MAN-MONTH, Addison-Wesley, Reading MA, 1975

[BU77] Bunge M, TREATISE ON BASIC PHILOSOPHY: VOL. 3: ONTOLOGY I THE FURNITURE OF THE WORLD, Reidel, Boston, MA, 1977

[CA94] Carmichael A ed., OBJECT DEVELOPMENT METHODS, SIGS Books, N.Y., 1994

[CDIF91] CDIF - STANDARDIZED CASE INTERCHANGE META-MODEL, EIA/IS-83, Electronic Industries Association, Engineering Department, Washington D.C. , July 1991

[CH76] Chen PPS, THE ENTITY RELATIONSHIP MODEL - TOWARD A UNIFIED VIEW OF DATA, ACM TA on DB Systems, Vol. 1, No. 1, March 1976

[CHE92] Chen JY, Lu JF, A NEW METRIC FOR OBJECT-ORIENTED DESIGN, Information and Software Technology, 35(1993)4, pp 232-240

[CHI94] Chidamber SR, Kemerer CF, A METRICS SUITE FOR OBJECT-ORIENTED DESIGN, IEEE TA on SE Vol. 20, No. 6, pp 476-493

[CO86] Conte SD, SOFTWARE ENGINEERING METRICS AND MODELS, Benjamin-Cummings, Menlo-Park, Ca., 1986

[DA83] Date CJ, AN INTRODUCTION TO DATABASE SYSTEMS, VOLUME II, Addison-Wesley, Reading, MA, 1983

[DAH66] Dahl OJ, Nygaard K, SIMULA - AN ALGOL-BASED SIMULATION LANGUAGE, Communications of the ACM, Vol. 9, No. 9, pp. 671-678, September 1966

[DC96] DeChampeaux D, <TITLE OF NEW BOOK>, <publisher>, 1996

[DM79] DeMarco T, STRUCTURED ANALYSIS AND SYSTEM SPECIFICATION, Prentice-Hall, Englewood Cliffs, N.J., 1979

[DM82] DeMarco T, CONTROLLING SOFTWARE PROJECTS, Prentice-Hall, Englewood Cliffs, N.J., 1982

[DM83] DeMarco T, A METRIC OF ESTIMATION QUALITY, Proc of the national computer conf., Washington DC. AFIPS Press, 1983

[DM88] DeMarco T, Lister T, PEOPLEWARE, Yourdon Press, 1988

[DU93] Dumke R, SOFTWARE-METRIKEN IN DER OBJEKTORIENTIERTEN SOFTWARE-ENTWICKLUNG, WP Universität Magdeburg, Sept. 1993

[EJ91] Ejiogu LO, SOFTWARE ENGINEERING WITH FORMAL METRICS, QED Technical Publishing Group, Boston, 1991

[FE91] Fenton N, SOFTWARE METRICS: A RIGOROUS APPROACH, Chapman & Hall, London U. K., 1991

[FE92] Fenton N, WHEN A MEASURE IS NOT A MEASURE, Software Engineering Journal, Sept 1992, pp. 357-362

[FI96] Fingar P, THE BLUEPRINT FOR BUSINESS OBJECTS, SIGS Books, N.Y., 1996

[FORTH78] Forth Interest Group, FORTH-78, San Carlos, CA, 1978

[GA94] Gamma E, et al., DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE, Addison-Wesley, Reading MA, 1994

[GO81] Goldberg A, et al., SPECIAL ISSUE ON SMALLTALK-80, Byte Magazine, August 1981

[GO85] Goldberg A, SMALLTALK-80 - THE INTERACTIVE PROGRAMMING ENVIRONMENT, Addison-Wesley, Reading, MA, 1985

[GR87] Grady RB, Caswell DL, SOFTWARE METRICS: ESTABLISHING A COMPANY WIDE PROGRAM, Englewood Cliffs, Prentice Hall, 1987

[GRA94] Graham IM, OBJECT-ORIENTED METHODS, 2nd ed., Addison-Wesley, Wokingham, UK, 1994

[GRA95] Graham IM, MIGRATING TO OBJECT TECHNOLOGY, Addison-Wesley, Wokingham, UK, 1995

[HA75] Halstead MH, SOFTWARE PHYSICS: BASIC PRINCIPLES, IBM Research RJ 1582, Yorktown Heights, N. Y., May 1975

[HA77] Halstead MH, ELEMENTS OF SOFTWARE SCIENCE, Elsevier North-Holland, New York, 1977

[HA95] Harmon P, USE-CASE AND OO-ANALYSIS, in Object-Oriented Strategies, Vol. V, No. 7, Cutter Information Corp., Arlington MA, 1995

[HAR87] Harel D, STATECHARTS - A VISUAL FORMALISM FOR COMPLEX SYSTEMS, in Science of Computer Programming 8, pp. 231-274, 1987

[HERMES95] HERMES 95, AN OPEN STANDARD FOR THE MANAGEMENT OF SOFTWARE PROJECTS, EDMZ P-No. 609.200d (german), Berne, Switzerland

[HO79] Hopcroft JE, Ullman JD, INTRODUCTION TO AUTOMATA THEORY, LANGUAGES AND COMPUTATION, Addison-Wesley, Reading, MA, 1979

[HS92] Henderson-Sellers B, MODULARISATION AND MCCABE'S CYCLOMATIC COMPLEXITY, CACM, December 1992, pp 17-19

[HS93] Henderson-Sellers B, Moser S, Seehusen S, Weinelt B, A PROPOSED MULTI-DIMENSIONAL FRAMEWORK FOR OBJECT ORIENTED METRICS, Proceeding of 1st Australian Software Metrics Conference, November 1993

[HS94] Henderson-Sellers B, Edwards J, BOOKTWO OF OBJECT-ORIENTED KNOWLEDGE, THE WORKING OBJECT, Prentice-Hall, Englewood Cliffs, N.J., 1994

[HS96] Henderson-Sellers B, OBJECT ORIENTED METRICS, Prentice-Hall, Upper Saddle River, N.J., 1996

[HS96b] Pant Y, Henderson-Sellers B, Verner JM, GENERALIZATION OF OBJECT-ORIENTED COMPONENTS FOR REUSE: MEASUREMENTS OF EFFORT AND SIZE CHANGE, Journal of Object-Oriented Programming, Vol. 9 No. 2, May 1996

[HU93] Hutt A, REVIEW OF OBJECT ANALYSIS AND DESIGN METHODS, Proceeding of Object Technology '93 Conf., B.C.S. London, November 1993

[HUM89] Humphrey W, MANAGING THE SOFTWARE PROCESS, Addison-Wesley, 1989

[IBM75] IBM Inc., PHASE-0-1 GUIDE, IBM USA, WTSC, 1975

[IBM76] IBM Inc., THE ARON ESTIMATION METHOD, IBM USA, Federal Systems Division, 1976

[IBM84] IBM Germany Inc., DIE FUNCTION POINT METHODE, IBM Germany, Munich, 1984

[IFPUG] International Function Point User Group (IFPUG), FUNCTION POINT COUNTING PRACTICES MANUAL [CURRENT VERSION], available at IFPUG, Westerville OH, USA

[JA92] Jacobson I, et al., OBJECT ORIENTED SOFTWARE ENGINEERING, Addison-Wesley, 1992

[JO85] Jones TC, SPQR/20 USER GUIDE, Software Productivity Research Inc., Burlington MA, 1985

[KA92] Karunanithi S, Bieman JM, CANDIDATE REUSE METRICS FOR OO AND ADA SOFTWARE, Proc. of the 1st intl. SW Metrics Symp. May 21-22, 1992, pp 120-128

[KE93] Kemerer CF, RELIABILITY OF FUNCTION POINTS MEASUREMENT, CACM Vol. 36, No. 2, pp 85-97

[KI93] Kitchenham B, Kænsælæ K, INTER ITEM CORRELATIONS AMONG FUNCTION POINTS, IEEE publication Nr. 0-8186-3740 — 4/93, pp. 11-14.

[KN91] Knöll HD, Busse J, AUFWANDSCHÄTZUNG VON SOFTWARE-PROJEKTEN IN DER PRAXIS, BI Wissenschaftsverlag, Mannheim, Wien, Zürich, 1991

[LA92] Lake A, Cook C, A SW COMPLEXITY METRIC FOR C++, Proc. of the 4th WS on SW Metrics, Mar 22-24, 1992

[LE94] Lehner F, MESSUNG DER TEXTVERSTÄNDLICHKEIT ALS BEITRAG ZUR PRÜFUNG DER DOKUMENTATIONSQUALITÄT, in DV Organisation Vol. 17 No. 1, K. G. Saur Verlag, München, 1994, pp 32-40

[LI92] Li W, Henry S, MAINTENANCE METRICS FOR THE OO PARADIGM, Proc. of the 1st intl. SW Metrics Symp. May 21-22, 1992, pp 52-60

[LIE96] Lieberherr K, ADAPTIVE OBJECT-ORIENTED SOFTWARE: THE DEMETER METHOD WITH PROPAGATION PATTERNS, PWS Publishing Company, Boston, 1996

[LO94] Lorenz M, Kidd J, OBJECT ORIENTED SOFTWARE METRICS, Prentice-Hall, Englewood-Cliffs, N.J., 1994

[MC76] McCabe TJ, A COMPLEXITY MEASURE, IEEE TA on SE Vol. 2, 1976, pp 308-320

[ME75] Mesarovic MD, Takahara Y, GENERAL SYSTEMS THEORY, Academic Press, N.Y., 1975

[ME88] Meyer B, OBJECT-ORIENTED SOFTWARE CONSTRUCTION, Prentice Hall Int. (UK), Hemel Hempstead, 1988

[MO91] Moser S, METRICS MANUAL, GfAI AG, Berne, Switzerland, 1991

[MO93] Moser S, et al., BI-CASE/OBJECT (BIO) V1.1 - A MODERN SYSTEMS DEVELOPMENT METHODOLOGY, Available at Bedag Informatik, Berne, Switzerland, 1993

[MO94] Moser S, EIN QS-SYSTEM FÜR OBJEKTORIENTIERTE SOFTWAREENTWICKLUNG, INFORMATIK, December 1994, SVI/FSI Schweiz. Verband der Informatikorganisationen, Zürich, Switzerland

[MO95] Moser S, METAMODELS FOR OBJECT-ORIENTED SYSTEMS, in Software-Concepts and Tools, Springer Intl., 1995, pp. 63-80

[MO95b] Moser S, Siegenthaler R, BI-CASE/OBJECT (BIO) V2.1, Bedag Informatik, Berne, Switzerland, 1995

[MO96] Moser S, Nierstrasz O, THE EFFECT OF OBJECT-ORIENTED FRAMEWORKS ON DEVELOPER PRODUCTIVITY, IEEE Computer, September 1996, pp. 45-51

[MO96b] Moser S, Cherix R, Flückiger J, BIO V3, Bedag Informatik, Berne, Switzerland, 1996

[MOR89] Moreau DR, Dominick WD, OBJECT ORINETED GRAPHICAL INFORMATION SYSTEMS: RESEARCH PLAN AND EVALUATION METRICS, The Journal of Systems and Software 10, pp. 23-28

[MORR89] Morris KL, METRICS FOR OBJECT ORIENTED SOFTWARE DEVELOPMENT ENVIRONMENTS, Thesis at M.I.T, Certified by Chris F. Kemerer

[MP84] McMenamin SM, Palmer JF, ESSENTIAL SYSTEMS ANALYSIS, Yourdon Press, N.Y., 1984

[NI95] Nierstrasz O, Tsichritzis D (eds.), OBJECT-ORIENTED SOFTWARE COMPOSITION, Prentice-Hall International, 1995.

[NO70] Norden PV, USEFUL TOOLS FOR PROJECT MANAGEMENT, in Management in Production, Starr MK (ed.), Penguin Books Inc., Baltimore, 1970, pp. 77-101

[OM92] Oman P, et al., METRICS FOR ASSESSING SW MAINTAINABILITY, IEEE Conf. on SW Maintenance, Nov. 1992, pp. 337-344

[OMG92] OMG Object Analysis and Design SIG, OBJECT ANALYSIS AND DESIGN, DRAFT 7.0, Object Management Group, 1992

[OPEN96] The OPEN Team, OPEN/MENTOR, A THIRD GENERATION SOFTWARE PROCESS, available at COTAR, Swinburne University of Technology, Australia (http://www.swin.edu.au/cotar/OPEN/OPEN.html), 1996

[ÖS92] Österle H, Gutzwiller T, KONZEPTE ANGEWANDTER ANALYSE- UND DESIGN-METHODEN, AIT, Hallbergmoos, 1992

[PA92] Paulson D, Wand Y, AN AUTOMATED APPROACH TO INFORMATION SYSTEMS DECOMPOSITION, IEEE TA on SW Engineering, Vol. 18, No. 3, March 1992

[PU76] Putnam LH, A MACRO-ESTIMATING METHODOLOGY FOR SW DEVELOPMENT, Proc. of 15th US Army OR Symp., 1976, pp 323-327

[PU80] Putnam LH, SOFTWARE COST ESTIMATING AND LIFE CYCLE CONTROL, Computer Society Press, IEEE, Los Alamitos CA, 1980

[RA96] Rational Inc., THE UNIFIED MODELING LANGUAGE, DRAFTS 0.8 AND 0.9, available at Rational Inc., Stanta Clara CA (http://www.rational.com), 1996

[RI78] Riedwyl H, ANGEWANDTE MATHEMATISCHE STATISTIK IN WISSENSCHAFT, ADMINISTRATION UND TECHNIK, Paul Haupt, Bern, 1978

[RO89] Robillard PN, Boloix G, THE INTERCONNECTIVITY METRICS: A NEW METRIC SHOWING HOW A PROGRAM IS ORGANIZED, The Journal of Systems and Software 10, pp. 29-39

[RU91] Rumbaugh J, Blaha M, Premerlani W, Eddy F, Lorensen W, OBJECT-ORIENTED MODELLING AND DESIGN, Prentice-Hall, Englewood Cliffs, N.J., 1991

[RUB96] Rubin H, Yourdon EN, <<CANADA INDUSTRY: INTERNATIONAL SOFTWARE ENGINEERING SURVEY, available at Rubin Inc., Albany, N.Y., 1996>>

[SBC95] OBJECT-ORIENTED TASK POINT COLLECTION AND METRICS CLUB, c/o Mark Lewis, Bezant Ldt., Wallingford, U.K., 1995

[SE89] Selby R, QUANTITATIVE STUDIES OF SOFTWARE REUSE, in Software Reusability Vol. II Applications and Experiences, Biggerstaff TJ, Perlis AJ (eds), Addison-Wesley, 1989, pp 213-233

[SH93] Sharble RC, Cohen SS, THE OO BREWERY, A COMPARISON OF 2 OO DEVELOPMENT METHODS, Software Engineering Notes 18(1993), pp 60-73

[SM91] Smolander K, METAMODELS IN CASE ENVIRONMENTS, Computer Science Report WP-20, University of Jyväskulä, Finland, 1991

[SN94] Sneed H, CALCULATING SOFTWARE COSTS USING DATA POINTS, SES, Ottobrunn/Munich, Germany, 1994

[SSADM90] SSADM VERSION 4 REFERENCE MANUAL, NCC Blackwell Ltd., UK, 1990, ISBN 1-85554-004-5

[ST86] Stroustrup B, THE C++ PROGRAMMING LANGUAGE, Addison-Wesley, Menlo Park, CA, 1986

[STA94] Stark G, Durst RC, USING METRICS IN MANAGEMENT DECISION MAKING, IEEE Computer, September 1994, pp 42-48

[SY88] Symons CR, FPA, DIFFICULTIES AND IMPROVEMENTS, IEEE TA on SE, Vol. 14, No. 1, 1988

[SY93] Symons CR, SOFTWARE SIZING AND ESTIMATING MK II FPA, John Wiley & Sons, N.Y., 1993

[TA91] Takagaki K, Wand Y, AN OBJECT ORIENTED INFORMATION SYSTEMS MODEL BASED ON ONTOLOGY, in Object Oriented Approach in Information Systems, F. Van Assche, B. Moulin, C. Rolland (eds.), Elsevier Science Publishers (North-Holland), 1991

[TE94] Texel PP, THE TEXEL METHOD, chpt. 13 of [CA94], 1994

[WA90] Wand Y, Weber R, AN ONTOLOGICAL MODEL OF INFORMATION SYSTEMS, IEEE TA on SW Engineering, Vol. 16, No. 11, November 1990

[WA92] Wand Y, Weber R, ON THE DEEP STRUCTURE OF INFORMATION SYSTEMS, IEEE Working Paper, University of British Columbia, Vancouver, December 1992

[WA93] Wand Y, Storey VC, Weber R, ANALYZING THE MEANING OF A RELATIONSHIP, Working Paper 92-MIS-011, University of British Columbia, February 1993

[WAL77] Walston CE, Felix CP, A METHOD OF PROGRAMMING MEASUREMENT AND ESTIMATION, IBM Systems Journal, Vol. 10, No. 1, 1977

[WALL96] Wallin Å, Moser S, Graber A, WIEDERVERWENDUNG MIT SMALLTALK IN CLIENT/SERVER-ANWENDUNGEN, INFORMATIK, February 1996, SVI/FSI Schweiz. Verband der Informatikorganisationen, Zürich, Switzerland

[WE81] Weiser MD, PROGRAM SLICING , in Proc. of 8th Int. Conf. on SW Eng., pp.439-449, 1981

[WEB92] Weber R, Zhang Y, AN ONTOLOGICAL EVALUATION OF NIAM'S GRAMMAR FOR CONCEPTUAL SCHEMA DIAGRAMS, Research Paper, University of Queensland, Australia, January 1992

[WEL89] Welke RJ, META SYSTEMS ON META MODELS, CASE Outlook 4/89, pp.35-45, December 1989

[WEY88] Weyuker EJ, EVALUATING SOFTWARE COMPLEXITY MEASURES, IEEE TA on SE Vol. 14 No. 9, 1988, pp. 12

[YO79] Yourdon EN, Constantine L, STRUCTURED DESIGN: FUNDAMENTALS OF A DISCIPLINE OF COMPUTER PROGRAM AND SYSTEM DESIGN, Yourdon Press, Englewood Cliffs, N.J., 1979

[YO89] Yourdon EN, MODERN STRUCTURED ANALYSIS, Yourdon Press, Englewood Cliffs, N.J., 1989

[ZU91] Zuse H, SOFTWARE COMPLEXITY, W. de Gruyter, New York, 1991

[ZU94] Zuse H, FOUNDATIONS OF THE VALIDATION OF OBJECT-ORIENTED SOFTWARE MEASURES, in Theorie und Praxis der Softwaremessung, Zuse H, Reiner D (eds.), Deutscher Universitätsverlag, Berlin, 1994

C Glossary of Terms and Abbreviations

Term / Abbreviation	*Explanation*
%C	Degree of completion of a result under construction
%F	Fraction of a result part
%P	Process completeness
4GL	= 4th Generation Language
4th Generation Language	Programming language with powerful commands supporting certain types of applications and certain kinds of user interfaces
A	In context of empirical databases: approximation function
	In context of dynamic cost models: process acceleration factor
ABC-Scheme	The structuring of the estimation process into 3 steps: A) measuring the subject's complexity on a model, B) using empirical correlations of the complexity with the desired estimate, C) adapting the estimate to specific circumstances, e.g. process completeness or process dynamics
ABC-Steps	= ABC-Scheme
ABC-Strategy	= ABC-Scheme
AMI	Application of metrics in industry [AMI92]
Application Analysis	The software process phase and corresponding artefacts which specify in detail what the software system is required to do
Backfiring	Applying the ABC-Scheme backwards
BIO	Bedag Informatik Object, an adaptable template software process incorporating modern object-oriented methods [MO96b]
C/K measures	A suite of object-oriented design measures proposed by Chidamber and Kemerer [CHI94]
CASE	Computer Aided Software Engineering
CBO	Coupling Between Objects, one of the C/K measures
CC	Configuration Container, an abstraction for containers of system descriptions; at implementation layer CCs usually are called „source files"
CDIF	CASE Data Interchange Format [CDIF91]
Chaining	The sequential application of two (or more) estimates according to the ABC-scheme, thus using the estimate coming from the first model as the predictor for the second
Co	= Construction
COCOMO	Constructive Cost Model [BOE81] [BOE95]
COH	A planned measure for capturing the idea of software cohesion
Component	A reusable piece of software with well-defined behaviour
Construction	The software process phase and corresponding artefacts of technical design, thus defining in detail how the specified system behaviour can be provided given a certain implementation environment
COOMM	Common Object Oriented Meta-Model [CA94]
COUP	A planned measure for capturing the idea of software coupling
CRUD	The 4 basic operations of information or database systems, i.e. creation, read, update and deletion of objects (corresponding to rows in a table in the relational model)
CS	Computer Science
csh	C-shell, a UNIX command line and scripting language

CUA	Common User Access, an IBM-standard for the design of graphical user interfaces
D	Duration
D	Day = elapsed work day, unit for measuring duration
dA	The relative bias of an approximation function in an empirical database, we use a 95%-confidence level, which means that the effective outcomes lie in the range of the estimate +/- dA/2 with a probability of 95%.
Data Abstraction	One of the key concepts of object-orientation, the bundling of data structures and corresponding functions into classes
DB	Database
Design	= Construction
DFD	Data Flow Diagrams [DM79]
diff	A UNIX utility to analyse the difference, added and deleted lines, between two streams of characters
DIT	Depth of Inheritance Tree, one of the C/K measures
Domain Analysis	The software process phase and corresponding artefacts of modelling the system's underlying real world or business reality as if no - or absolutely perfect, thus ignorable - technology were present, thus roughly defining how the system is required to behave
DOME	A formal language to denote the domain analysis artefacts
DOME-SM	The variant of the System Meter measure measured on system descriptions written in the DOME language
E	in the context of process parameters: manpower effort in the context of software metrics: one of the Software Science metrics [HA77]
EIA	Electronic Industries Association, a standards association [CDIF91]
ELOC	Effective lines of code, i.e. all lines in source files but without comments, declarations and blank lines, a LOC variant
Empirical Database	A set of data pairs, predictor and result values, that were empirically measured; the set of data pairs is used to derive regressed approximation functions for use in the ABC-scheme of estimation
EQF	Estimation Quality Factor
ER	Entity Relationship [CH76]
ERA	Entity Relationship Attributes [CH76]
ESA	Essential Systems Analysis [MP84]
Estimation Model	A 5-tuple of predictor metamodel, predictor metric, empirical database, result metric, result metamodel, that forms an instrument for estimation of values of the result metric
η	The number of unique tokens found in a system description, a Software Science measure [HA77]
FP	Function Point [AL79] [IFPUG]
FPA	Function Point Analysis, the human process of measuring FPs
FPA Mk II	Function Point Analysis Mark two [SYM93]
FPM	Function Point Method = FPA
Framework	A reusable set of software components that provide specific behaviour to the end-user based on certain internal collaboration; a framework can be extended and - at certain points - specific new behaviour can be introduced
GQM	Goal Question Metric, an approach for a stepwise introduction of metrics into a SPU

GUI	Graphical User Interface
I	Effort inflation factor, the ratio of effort under time pressure and normally estimated effort
IFPUG	International Function Point User Group [IFPUG]
Implementation	The software process phase and corresponding artefacts (code) of implementing and testing the system's behaviour as specified in application analysis and according to the technical patterns identified in construction
Information Hiding	One of the key concepts of object-orientation, the concept of separating implementation from externally visible behaviour of an element of system description
Inheritance	One of the key concepts of object-orientation, the fact that a class automatically contains all data and function elements of another class if it inherits from it; a distinction between inheritance of all the elements or just the externally visible ones (= subtyping) may be made
Intermodel Bias	The bias introduced into measurements of system models by the fact that the same „real" model may be modelled differently due to different human beings and different modelling techniques (= metamodels)
Interrater Bias	The bias introduced into measurements by ambiguous measurement rules which are differently interpreted by different human raters (= measurers)
IS	Information Systems
ISO 9000ff	International standard for quality management systems in manufacturing and service
KSM	from German „Konzeptions- Spezifikations-Modell", an estimation model [MO91]
Language	A category of system description elements; language elements are characterised by their highly standardisation and stability
Layering	The grouping of software artefacts into layers of abstraction, e.g. into preliminary, domain and application analysis, design and implementation; the corresponding partitioning of the software process is called phasing
LCOM	Lack of cohesion in methods, one of the C/K measures
ldf	language definition file, a required element of the ma utility
ldl	language definition language, used in ldf's
Library	A category of system description elements; library elements are characterised by not being newly developed within a software process, but - in contrast to the language objects - not being highly standardised and stabile
Life-Cycle Phases	The life cycle of a software product typically is partitioned into the phases of analysis, design, implementation and test, acceptance and delivery, maintenance
LOC	Lines of code
ma	A DOS-hosted utility for measuring system descriptions; this utility was developed for the field study we conducted in industry and was used to measure the System Meter (PRE and DOME variants)
Metamodel	A metamodel is a model of a model; in the context of software engineering metamodels are used to discuss elements used to build software, e.g. classes, methods, etc.; metamodels are also needed to support the process of software construction with tools and other aids
Metamodel Mapping	A function that defines for each element of one metamodel how it is represented in another. The representation may be direct, a „one-to-one" map, or indirect
Metamodel of a Metric	The smallest set of entities and associations needed to define the metric

Metric	in the context of software engineering: = measure
	in mathematical context: a function yielding the distance between two elements of a vector room
Metrication	The process of introducing metrics in an SPU or the process of deriving some new metric
MIS	Management information systems
MOP	Meta-object protocol, a programming language feature allowing to program the software construction elements themselves
MSA	Modern structured analysis [YO89]
N	Number of tokens, one of the Software Science measures [HA77]
NOC	Number of child classes, one of the C/K measures
υ	Cyclomatic complexity [MC76]
Object Technology	The ensemble of object oriented analysis, design and programming techniques as well as modern software processes like RAD, prototyping, etc.
OMG	Object Management Group
OMT	Object Modelling Technique [RU91]
OPRR	Object-Property-Role-Relationship metamodel [WEL89] [SM91]
OT	= Object Technology
P	in the context of process measures: = Productivity
	in the context of estimation models: = Predictor
PD	Person day, a unit to measure manpower effort
Polymorphism	One of the key concepts of object-orientation, the fact that an inheriting class may redefine methods of its superclass, thus - at run-time - behave differently, even though the static function call may look identical
PQ	Quality based productivity
PRE	A language to formally denote preliminary analysis results
PRE-SM	The variant of the System Meter measurable on systems described in the PRE language
Predictor	The value used to predict (= estimate) some other value
Predictor Metric	The metric used as predictor values in some estimation model
Predictor Model	The set of software or process artefacts measured to get a predictor; the predictor model must obey a metamodel, the predictor metamodel
Preliminary Analysis	The software process phase and corresponding artefacts of modelling a system as set of subject areas, functional goals (= basic functionalities) and specifications of which subject area should encompass which functionality
Process Parameters	The three main software process parameters are 1) cost/effort, 2) elapsed time, 3) product/quality
Product Quality	The internally measurable or empirical quality of a software product; empirical quality = Q
Productivity	The ratio of product size and process effort
Project	in the context of software artefacts: the category of newly developed software artefacts
	in the context of software management: a contract encompassing one, many or a part of software processes
Prototype	Prototype
Prototyping	The (software) process of developing a prototype
PSP	Project structure plan

PU	Product unit, e.g. the System Meter, Function Point, LOC or some other software metric
Q	Quality, the ratio of software artefacts not thrown away after some time and the original set of software artefacts
RAD	Rapid Application Development
Re	= Replication
Replication	= Implementation
Result Metric	The metric used as result values in some estimation model
Result Model	The set of software or process artefacts for which a; the predictor model must obey a metamodel, the predictor metamodel
RFC	Response set for a class, one of the C/K measures
S/C	Size/Complexity, a newly proposed software measure [HS96]
SA/SD	Structured Analysis, Structured Design
Scale	A set of values used as the target domain of a measure; scales can be nominal, ordinal, ratio, interval and absolute
SD	= System Description
sd	= SD
sdf	system definition file, a required element of the ma measurement utility
SE	Software engineering, the ensemble of techniques used in the software process
SEI	Software Engineering Institute, affiliated at the Carnegie Mellon University (CMU), Pittsburgh, PA
SLOC	Source lines of code, = ELOC
SM	= System Meter
SM’	= „flat“ SM, i.e. SM without the categorisations of language, library and project elements (all elements are of category project)
SMM	in the context of metamodelling: = System Metamodel in the context of estimation: System Meter Method
Software	All software artefacts that belong to the implementation layer
Software Artefact	All products developed by the software process
Software Life-Cycle	= Life-Cycle
Software Measure	A measure or rule that allows to compute or assign a value of some scale to one or a set of software artefacts
Software Metric	= Software Measure
Software Metrics	Research and practice of software measurement and estimation
Software Process	The human - and partly machine-supported - process of developing software; produces software artefacts
Software Product	= Software Artefact
Software Quality	Can be interpreted as product quality or software process quality
Software Process Quality	The quality with respect to all process parameters, i.e. product, cost and time
SOMA	Semantic object modelling approach [GRA95]
Specification	= Application Analysis
SPU	Software producing unit
SQL	Structured query language
SSADM	Structured Systems Analysis and Design Method [SSADM90]

System	An entity distinguished from its environment which can be characterised by the interactions going from the environment to the system and back; this definition is very generic and encompasses real world systems as well as software systems; in the realms of software this term may be used for the static, coded, system as well as for the dynamic, executed, computer process
System Description	The notion for all artefacts in the domain of software engineering; alternatively system descriptions may also be called system models or - simply - software
System Metamodel	A metamodel of system descriptions; the System Metamodel is the most detailed metamodel observable on models of the layer of implemented systems, i.e. on code; the System Metamodel encompasses artefacts like classes, methods, etc. and serves as the basis for the definition of the System Meter
System Meter	A newly proposed software measure
Technical Design	= Construction
UML	Unified Modelling Language [RA96]
V	in the context of software process measures: velocity, the ratio of achieved result size versus elapsed days in the context of complexity metrics: volume, one of the Software Science metrics [HA77]
WMC	Weighted Methods per Class, one of the C/K measures
WWM	Wand-Weber-Metamodel [WA90]

D A C++ Framework for Measuring System Descriptions

D.1 Framework Introduction and Class Diagram

The framework presented on the following few pages was used to build the measurement tool **ma**. This **ma** (measure anything) utility may be used to measure software metrics out of source code. Refer to the manual pages in MA.DOC (Version: 13. January 1996, obtainable on email request from the author) to get aquainted with the complete functionality of this UNIX-style command line utility running under the DOS operating system.

The utility makes use of a framework that reflects the metamodel entities of the System Metamodel thus allowing the easy programming of measurements based on such a metamodel in C++. This framework is briefly presented here. The reader is supposed to be familiar with basic and more advanced concepts of software metrication and metamodelling

The class hierarchy is similar to that of the System Metamodel except for two additional base classes: 1) `Named` which deals with registering and finding named items and 2) `Sizeable` which is an abstract base class for all measurable items. It is diagrammed as follows:

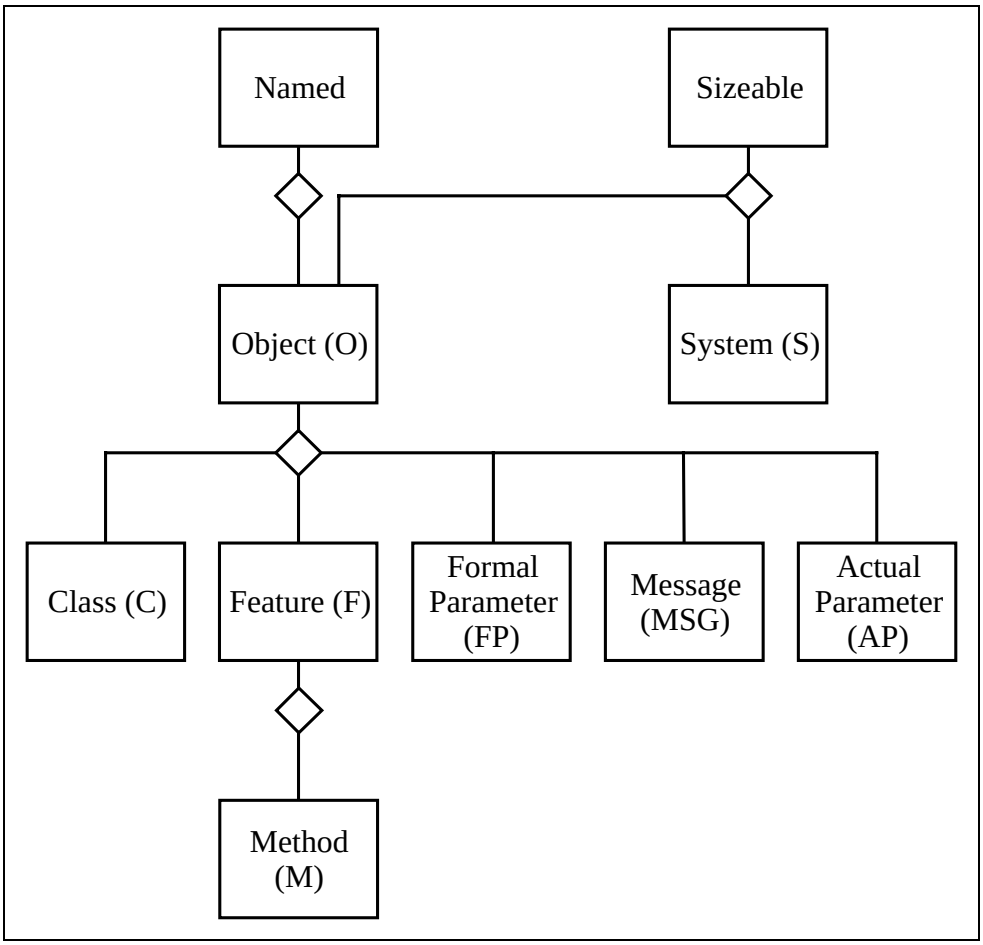

Figure D.1: Metrics Framework Class Hierarchy

The exact specific purpose of each of these classes may best be understood by browsing through the C++ header file listings as given in subsection D.2 of this appendix.

This object oriented framework is accompanied by a set of utility functions that support tokenising and file scanning. The most important are:

```
co Str& ma_tokenize         (co Str& source_string,
                             co Str& ws = white_spaces,
                             co Str& sl_comment = empty_str,
                             co Str& mt_comment=empty_str, co Str&=empty_str, Boo=FALSE,
                             co UList& tok_seps = empty_ul,
                             co UList& lits=empty_ul,co UList&=empty_ul,co UList&=empty_ul,
                             co Str& multi_tokens = s_sq, co Str& = s_bs, co Str& = s_sq
                             );
       // returns a tokenized version of the first string arg, tokens are separated by
       // \n's. 2nd arg: white space characters as a string
```

```
        // 3rd string arg: single line comment token (Nil if none)
        // 4rd-6th arg: multi line comment start and end token + nested flag
        // 7th arg: list of tokens that are recognised as separate even if
        //          not separated by white spaces (list order: len desc).
        // 8th-10th arg: triplet of lists of lit start/escape/end enclosers
        // 11th-13th arg: triplet of multi token start/escape/end enclosers

Boo     ma_scanfile (CStr& file, CStr* lan, co UList& sets, CStr* ver, CStr* cat,
                     co UList& tags, CStr* delta, CStr* sub, System* system);
        // builds up the meta-object-structure found in a file. The language,
        // settings, category and system must be set, the rest may be empty/NULL.
        // While scanning, any setting may be altered, but the scan restores
        // the settings on exit. The returned value is False on failure.

UList*  ma_scanldfile (CStr& file);
        // return Nil or a caller owned UList of 1) the main language name, 2) the
        // language's settings (in a UList) and 3) several language aliases
        // in another UList.
```

*Listing D.1: Supporting Functions for the **ma** Utility*

Note that several important abbreviations (`#define`s) are used throughout the code: `co = const, Str = String, CStr = CollectableString, UList = UnsortedList` (an implementation of a linked list which may contain collectables). Also note that each of the framework classes are collectables.

This framework is primarily designed to be easily extensible with respect to two requirements:

① Defining new metrics

② Supporting new metamodel mapping rules

This appendix will provide the solution pattern, i.e. the necessary construction and programming steps, for fulfilling requirement ① which is described in subsection D.3. Getting acquainted with the class headers in D.2 before will support the reading of this subsection.

Requirement ② is supported by the implementation of all meta-model entities as classes. It is therefore possible to create at any desired time new system description objects such as classes, methods, features, etc. and provide them with a name and associations as prescribed by the mapping rule. Note that the mapping method - together with its formal parameters - has to be introduced in the language definition file of every language that makes use of that method. The mapping method itself is then implemented as a new branch within the `execute` method of class `Method`.

Change strings

Any changes to the meta-entities are performed through the `change` method. This method takes a string as its main argument. This string contains the change information as a sequence of tokens separated by newlines, e.g. the string „has\nZIP-Code“ sent to a class object sets a „has“ association between that class object and a feature object named „ZIP-Code“ (which is assumed to be created before). All meta-associations, attributes and categories may be changed - also deleted - at run-time in this manner. Change strings may contain more than one change information, e.g. „has\nZIP-Code\nhas\nCity“ is also valid. A change string may also be passed to the constructor. Note that language definition files are nothing but untokenised sequences of change strings with the additional possibilities of comments, setting of language-specific parameters (such as the comment syntax) and creation of language description objects by stating their meta-name (e.g. `Method „my new method“` to create a new method named ‘my new method’).

D.2 Framework Class Details (Header Files)

The following excerpts from the actual C++ header files show the essentials of the classes purposes. Implementation issues (some inheritance links, #includes, protected and private members) are omitted. Also note that often the abbreviations for the metamodel classes are used, e.g. instead of `Method*` you'll see `M*`. For return values that are created on the heap (free store) and whose deletion is up to the method caller, the term "cowned" is often used, which is short for "caller owned". The class headers appear as the classes are diagrammed in the hierarchy diagram in subsection D.1 (left to right, top to bottom).

Note that the read access methods to the meta-associations and attributes are grouped in a correspondingly commented section of the header. They are followed by the list of settable categories. Change strings should be built using exactly the spelling of these methods and categories. To support the correct construction of change strings, for each meta-association, attribute and category a string named `s_`<meta- name> is defined. Also the string `s_cr` is defined which can be inserted as the token separator (instead of many literal „\n“s).

D.2.1 Class Named

```
Purpose Defines a C++ representation of a named object that may additionally take on
        a set of string values to model tags/categories. Searching methods are defined
        for name patterns and tag patterns.

        Names are sequences of tokens denoted as follows:
        token separator                     = blank
        literal token                       = sequence of chars (with \ escapes)
        paired literals                     = pairedToken1. token .pairedToken2
        named single token variable         = 'name'
        named multiple token variable       = 'name...'
        anonymous variable                  = *
        optional token                      = [ token ]
        repeated token                      = token ...
        repeated token with separator token = token separatorToken...
        ("inclusive") token group           = ( token1 token2 ... )
        one out of ("exclusive") group      = (( token1 token2 ... ))

class   Named
    {   public:
        Named   (co Str& n = empty_str, co Str& v = empty_str, co Str& t = empty_str);
            //  If n == "" then the named is assumed to be anonymous.
          ~Named   ();
    sta Named*  findName    (co Str& name = empty_str);
            //  Returns the named with the name or NULL if no match
    sta UList&  find        (co UList&=empty_ul, co UList&=empty_ul, co char* n_pat=NULL);
            //  Returns a ulist of nameds that
            //  1) are tagged with the tags in the first UList (may be empty),
            //  2) are not (!) tagged with the tags in the 2nd UList (may be empty),
            //  3) match the name pattern (regexp) or id (if pat like <#>).
        UList*  match       (co char* text);
            //  Returns (caller owned)
            //  1) the "remaining" portion of (tokenized) text,
            //  2) the matching portion (if a match occurred),
            //  3)... the variables (index i) and settings (index i+1).
            //  The variable names contain ... if repeated.
            //  The settings are either a string or a ulist. A string is used
            //  for vars with a single actual setting. The string is ended
            //  with a \n. Multitoken vars are set with tokenized strings
            //  separated with \n's. A ulist is used for repetitions. The
            //  values in the ulist are strings (formed as described above).
    sta UList*  findMatch   (co char* text,UList&=empty_ul,UList&=empty_ul,co char*=NULL);
            //  Returns a caller owned list with the following contents
            //  (see 'find' for parms 2-4):
            //  1) the portion of the text that did not match
            //  2) the portion that matched (optional)
            //  3) the named that matched (optional, non cowned)
            //  4) the rest of the text (optional)
            //  5)... the variables and settings (optional)
            //  The text must be tokenized (<cr> separated).
        int     name        (co Str&, ATy = repl); // change name (retval == 0: failure)
        co Str& name        () const;              // retrieve name
        int     value       (co Str&, ATy = repl); // changing the value (anons only)
        co Str& value       () const;              // retrieving the value (anons only)
        int     category    (co Str&, ATy = add);  // changing tags/categories
        UList&  category    () const;              // retrieving tags/categories
     vi Boo     is          (co Str& t) const;     // TRUE if this is tagged with t
        UList*  variables   () const;              // returns the variables (cowned)
        UList*  literals    () const;              // returns the literals (cowned)
        };
```

D.2.2 Class Sizeable

```
class   Sizeable
    {   public:
        Sizeable    ();
/*      abstract measures (absolute scale) */
    vi  float       externalSize ();            // es
    vi  float       internalSize ();            // is
    vi  float       size ();                    // s
/*      abstract measures (scale from 0%-100%) */
    vi  float       quality ();                 // q
/*      real measures (System Meter Suite) */
    vi  float       systemMeters ();            // sm
    vi  float       internalSystemMeters ();    // ism
    vi  float       externalSystemMeters ();    // esm
/*      real measures (Function Point Suite) */
    vi  float       preFunctionPoints ();       // prefp
    vi  float       domeFunctionPoints ();      // domefp
/*      real measures (Lines Of Code Suite) */
    vi  float       linesOfCode ();             // loc
    vi  float       weightedLinesOfCode ();     // wloc
/*      real measures (McCabe Suite) */
    vi  float       cyclomaticComplexity ();    // cc
/*      real measures (Halstead Suite) */
    vi  float       volume ();                  // vol
    vi  float       effort ();                  // eff
    vi  float       difficultyHalstead ();      // dif
    vi  float       vocabularyHalstead ();      // voc
    vi  float       lengthHalstead ();          // len

/*      real measures (Chidamber-Kemerer Suite) */
    vi  float       numberOfChildren ();        // noc
    vi  float       depthOfInheritanceTree ();  // dit
    vi  float       couplingBetweenObjects ();  // cbo
    vi  float       weightedMethodsPerClass (); // wmc
    vi  float       responseForClass ();        // rfc
    vi  float       lackOfCohesionInMethods (); // lcom
        };
```

D.2.3 Class Object

```
class   Object
    :   public
        Sizeable, public Named
    {   public:
        Object  (Str& str = empty_str, co UList& tags = empty_ul);
            //  Creates an object out of ldl-text in str.
            //  The unscannable rest of str is left in str.
            //  Scanning stops when encountering a non conforming token.
            //  Sets the given tags in addition to the std tags.
       ~Object ();

    sta O*      findObject (co Str& str_1);

    sta O*      replaceObject (O* old_1, O* new_2);
            //  replaces the old object with the new one, i.e. every
            //  occurrence of old (= pointer to old) in the associated objects
            //  is replaced by an occurrence of new (= pointer to new).
            //  Old is deleted and new is assigned old's name.

     vi int     change(Str& str,co UList& tags=empty_ul,ATy=repl,Boo=TRUE);
            //  Changes the object's state according to ldl-text in str.
            //  The eventually newly created subobjects are tagged with tags.
            //  As in the constructor, str is changed. Also the object's
            //  name may be changed. The Boolean controls whether changes
            //  are propagated (=default) or the change is just flat.
            //  If assoc is given, no object names are assumed to be
            //  in str. Instead the change always references the object
            //  pointed to by assoc. Retval == 0: failure.
     vi void    remove (co Str& str, co O* op);
            //  Removes the object pointed to by op from the relationship list
            //  named str (e.g. 'remove ("isAliasFor", <obj>);').

  vi co Str&    metaType ();

        // metamodel association and attribute access members
     co Str&    hasName () co { return name (); }
     co Str&    hasValue () co { return value (); }
     vi O*      isAliasFor ();
     vi UList&  hasAsAlias ();
     vi UList&  contains ();
     vi UList&  isContainedIn ();
     vi UList&  scopes ();
     vi UList&  isInScopeOf ();
     vi UList&  isOfType ();
     vi UList&  isDefaultValueFor ();
     vi UList&  isReferredIn ();
     vi UList&  isUsedBy (co Str& = empty_str);
     vi UList&  mayBeUsedBy (co Str& = empty_str);
     vi UList&  uses (co Str& = empty_str);
     vi UList&  mayUse (co Str& = empty_str);

//      meta-categories settable as tags/categories and in the change method
//      pers = persistent object
//      dev = device object
//      ctrld = controlled object
//      cont = container object
//      const = constant object
//      anon = anonymous object
//      loc = local object
//      lit = literal object
```

```
        // measures
        float   systemMeters ();
        float   structuredDesignQuality ();
        float   internalSystemMeters ();
        float   externalSystemMeters ();
        };
```

D.2.4 Class System

```
enum    sdfEntryType // used while scanning an sdf
    {   undefined_set, lan_set, ver_set, tag_set, sub_set
        };
typedef sdfEntryType sdfET;

class   System : public Sizeable
    {   public:
        System  (co char* sdf); // creates a system out of an SDF-File
       ~System  ();
        int     name    (co char*, ATy = replacing_aty);
     co char*   name    () co;
        int     alias   (co char*, ATy = additive_aty);
        int     isNamed (co char* pattern) co;
        // settings of the current language
    sta UList*  ulp_currentLanguageSettings;

        // inherited and redefined from sizeable
        float   externalSize ();
        float   internalSize ();
        float   size ();
        float   qualityBasedExternalSize ();
        float   qualityBasedInternalSize ();
        float   qualityBasedSize ();
        float   quality ();
        float   systemMeters ();
        float   structuredDesignQuality ();
        float   internalSystemMeters ();
        float   externalSystemMeters ();
        float   coupling ();
        float   cohesion ();
        float   preFunctionPoints ();
        float   essoaFunctionPoints ();
        float   functionPoints ();
        float   linesOfCode ();
        float   weightedLinesOfCode ();
        float   cyclomaticComplexity ();
        float   volume ();
        float   effort ();
        float   difficultyHalstead ();
        float   vocabularyHalstead ();
        float   lengthHalstead ();
        float   numberOfChildren ();
        float   depthOfInheritanceTree ();
        float   couplingBetweenObjects ();
        float   weightedMethodsPerClass ();
        float   responseForClass ();
        float   lackOfCohesionInMethods ();
        };
```

D.2.5 Class Class

```
class   Class : public Object
    {   public:

//      administrative public methods

        Class   (Str& str = empty_str, co UList& tags = empty_ul);
            //  Creates a class out of ldl-text in str.
            //  The unscannable rest of str is left in str.
            //  Scanning stops when encountering a non conforming token.
            //  Sets the given tags in addition to the std tags.
       ~Class   ();
            //  Deletes a class.

    sta C*      findClass (co Str& str_1);

     vi int     change (Str& str, co UList& tags=empty_ul, ATy=repl, Boo=TRUE, O*=NULL);
            //  Changes the class's state according to ldl-text in str.
            //  As in the constructor, str is changed. Also the class's
            //  name may be changed. The Boolean controls whether changes
            //  are propagated (=default) or the change is just flat.
            //  Retval == 0: failure.

     vi void    remove (co Str& str, co O* op);
            //  Removes the object op from the relationships in str.
     vi void    infect (co Str& str);
            //  Marks the object elements listed in str as outdated.

  vi co Str&    asString ();
  vi co Str&    metaType ();

        // metamodel association and attribute access members
        UList&  isSubTypeOf ();
        UList&  isSuperTypeOf ();
        UList&  inheritsFrom ();
        UList&  inheritsTo ();
     co Str&    creates ();
        C*      hasClassTemplate ();
        UList&  isClassTemplateFor ();
        UList&  has ();
        UList&  isTypeOf ();

//      special members
        O*      createInstance (co Str& name = empty_str, UList& tags = empty_ul);

//      sizeable methods
/*      real measures (Function Point Suite) */
    vi  float   preFunctionPoints ();        // prefp
    vi  float   essoaFunctionPoints ();      // essoafp
    vi  float   functionPoints ();           // fp

/*      real measures (Chidamber-Kemerer Suite) */
    vi  float   numberOfChildren ();         // noc
    vi  float   depthOfInheritanceTree ();   // dit
    vi  float   couplingBetweenObjects ();   // cbo
    vi  float   weightedMethodsPerClass ();  // wmc
    vi  float   responseForClass ();         // rfc
    vi  float   lackOfCohesionInMethods ();  // lcom
        };
```

D.2.6 Class Feature

```
class   Feature : public Object
    {   public:

//      administrative public methods

        Feature (Str& str = empty_str, co UList& tags = empty_ul);
            //  Creates a feature out of ldl-text in str.
            //  The unscannable rest of str is left in str.
            //  Scanning stops when encountering a non conforming token.
            //  Sets the given tags in addition to the std tags.
       ~Feature   ();
            //  Deletes a feature.

    sta F*      findFeature (co Str& str_1);

     vi int     change (Str& str, co UList& tags=empty_ul, ATy=repl, Boo=TRUE, O*=NULL);
            //  Changes the feature's state according to ldl-text in str.
            //  As in the constructor, str is changed. Also the feature's
            //  name may be changed. The Boolean controls whether changes
            //  are propagated (=default) or the change is just flat.
            //  Retval == 0: failure.

     vi void    remove (co Str& str, co O* op);
            //  Removes the object op from the relationships listed in str.
     vi void    infect (co Str& str);
            //  Marks the calculated object elements listed in str as outdated.

  vi co Str&    asString ();
  vi co Str&    metaType ();

        // metamodel association and attribute access members
        C*      isPartOf ();

//      meta-categories settable as tags/categories and in the change method
//      cla = class-wide feature
//      inhable = inheritable feature
//      inhted = inherited feature
        };
```

D.2.7 Class Formal Parameter

```
class   FP : public Object
    {   public:

//      administrative public methods

        FP      (Str& str = empty_str, co UList& tags = empty_ul);
            //  Creates a FP out of ldl-text in str.
            //  The unscannable rest of str is left in str.
            //  Scanning stops when encountering a non conforming token.
            //  Sets the given tags in addition to the std tags.
       ~FP      ();
            //  Deletes an fp.

    sta FP*     findFP (co Str& str_1);

     vi int     change (Str& str, co UList& tags=empty_ul, ATy=repl, Boo=TRUE, O*=NULL);
            //  Changes the FP's state according to ldl-text in str.
            //  As in the constructor, str is changed. Also the class's
            //  name may be changed. The Boolean controls whether changes
            //  are propagated (=default) or the change is just flat.
            //  Retval == 0: failure.

     vi void    remove (co Str& str, co O*);
            //  removes O* from the list stated in str.
     vi void    infect (co Str& str);
            //  Marks the object elements listed in str as outdated.

  vi co Str&    asString ();
  vi co Str&    metaType ();

        // metamodel association and attribute access members
        O*      hasDefaultValue ();
        M*      belongsTo ();
        UList&  isFilledBy ();

//      meta-categories settable as tags/categories and in the change method
//      crea = formal parameter whose actual parameters are newly created objects
//      read = formal parameter whose actual parameters are read
//      upd = formal parameter whose actual parameters are changed
//      del = formal parameter whose actual parameters are deleted
        };
```

D.2.8 Class Message

```
class   Message : public Object
    {   public:

//      administrative public methods

        Message (Str& str = empty_str, co UList& tags = empty_ul);
            //  Creates a message out of ldl-text in str.
            //  The unscannable rest of str is left in str.
            //  Scanning stops when encountering a non conforming token.
            //  Sets the given tags in addition to the std tags.
       ~Message ();
            //  Deletes a message.

    sta MSG*    findMSG (co Str& str_1);

     vi int     change (Str& str, co UList& tags=empty_ul, ATy=repl, Boo=TRUE, O*=NULL);
            //  Changes the message's state according to ldl-text in str.
            //  As in the constructor, str is changed. Also the msg's
            //  name may be changed. The Boolean controls whether changes
            //  are propagated (=default) or the change is just flat.
            //  Retval == 0: failure.

     vi void    remove (co Str& str, co O*);
            //  removes O* from the list stated in str.
     vi void    infect (co Str& str);
            //  Marks the object elements listed in str as outdated.

  vi co Str&    asString ();
     co Str&    metaType ();

        // metamodel association and attribute access members
        UList&  isDirectlyUsedBy (co Str&); // redefinition from Object
        M*      calls ();
        M*      implements ();
        UList&  with ();

//      meta-categories settable as tags/categories and in the change method
//      exec = executed message

//      special functions
        void    execute (UList& tags = empty_ul, UList& parms = empty_ul);
            //  executes the message (by calling the called method and
            //  passing the actual parameters). The parameters in the list are
            //  formal parameters each followed by the object passed as the
            //  actual parameter. Whenever a formal parameter in this list
            //  is passed as an actual parameter to the called method, it is
            //  substituted with the object immediately following in the
            //  list.
        };
```

D.2.9 Class Actual Parameter

```
class   AP : public Object
    {   public:

//      administrative public methods

        AP      (Str& str = empty_str, co UList& tags = empty_ul);
            //  Creates an AP out of ldl-text in str.
            //  The unscannable rest of str is left in str.
            //  Scanning stops when encountering a non conforming token.
            //  Sets the given tags in addition to the std tags.
       ~AP      ();
            //  Deletes an ap.

    sta AP*     findAP (co Str& str_1);

     vi int     change (Str& str, co UList& tags=empty_ul, ATy=repl, Boo=TRUE, O*=NULL);
            //  Changes the ap's state according to ldl-text in str.
            //  As in the constructor, str is changed. Also the ap's
            //  name may be changed. The Boolean controls whether changes
            //  are propagated (=default) or the change is just flat.
            //  Retval == 0: failure.

     vi void    remove (co Str& str, co O*);
            //  removes O* from the list stated in str.
     vi void    infect (co Str& str);
            //  Marks the object elements listed in str as outdated.

  vi co Str&    asString ();
  vi co Str&    metaType ();

        // metamodel association and attribute access members
        UList&  eq (); // more than one actual value (i.e. a list of values)
        FP*     correspondsTo ();
        MSG*    isUsedIn ();
        };
```

D.2.10 Class Method

```
class   Method : public Feature
    { public:

//      administrative public methods

        Method  (Str& str = empty_str, co UList& tags = empty_ul);
            //  Creates a method out of ldl-text in str.
            //  The unscannable rest of str is left in str.
            //  Scanning stops when encountering a non conforming token.
            //  Sets the given tags in addition to the std tags.

       ~Method  ();
            //  Deletes a method.

    sta M*      findMethod (co Str& str_1);

     vi int     change (Str& str, co UList& tags=empty_ul, ATy=repl, Boo=TRUE, O*=NULL);
            //  Changes the method's state according to ldl-text in str.
            //  As in the constructor, str is changed. Also the method's
            //  name may be changed. The Boolean controls whether changes
            //  are propagated (=default) or the change is just flat.
            //  Retval == 0: failure.

     vi void    remove (co Str& str, co O* op);
            //  Removes the object op from the relationships listed in str.

     vi void    infect (co Str& str);
            //  Marks the object elements listed in str as outdated.

  vi co Str&    asString ();
  vi co Str&    metaType ();


//      access members for the meta-associations
        UList&  parameter ();
        UList&  isImplementedBy ();
        UList&  isCalledBy ();
        M*      hasMethodTemplate ();
        UList&  isMethodTemplateFor ();

//      meta-categories settable as tags/categories and in the change method
//      deref = dereferencing method
//      debra = decision/branch method
//      impl = implemented method
//      cmt = comment method

//      sizeable methods
/*      real measures (Function Point Suite) */
    vi  float   preFunctionPoints ();        // prefp
    vi  float   essoaFunctionPoints ();      // essoafp
    vi  float   functionPoints ();           // fp

//      special functions
        void    execute (UList& tags, UList& parms);
                    //      executes the method (i.e. establishes the meta-associations
                    //      defined for the method). The parameters in the list are
                    //      formal parameters each followed by the object passed as the
                    //  actual parameter. Newly created objects are tagged with tags.
        };
```

D.3 Using the Framework: Defining New Metrics

As an example, we may want to define some of the object-oriented metrics of Chidamber and Kemerer [CK94]: NOC, DIT and WMC. We do this in the following steps:

① We introduce the method signatures - together with some dummy implementation - in the Sizeable base class:

Note that every metric is of type float even if the anticipated values are integers. This convention is chosen because in calculations (especially multiplications) values exceeding integer (or long) maximums may occur.

② We chose the framework class for which the metrics may be (re)defined. Because all Chidamber/Kemerer metrics are defined at class-level we take class Class. Other metrics might be defined in other classes, e.g. McCabe's Cyclomatic Complexity in class Method - or even in several classes, e.g. McCabe's Cyclomatic Complexity may also be defined in Class, there being the sum of the complexities of the class's methods.

③ We implement the metrics using the metamodel associations and classes:

```
float   Class::numberOfChildren ()
    {   return (float)inheritsTo ().entries ();
        }
```

Listing D.2: Implementation of NOC

For NOC we may simply count the entries of the list returned by the inheritsTo association method. A little bit more tricky is the definition of DIT:

```
float   Class::depthOfInheritanceTree ()
    {   ULister next (inheritsFrom ()); // iterator for assoc.
        Class* cp = NULL; // pointer to class inherited from
        float depth = -1.0f; // current maximum depth
        float newDepth = 0.0f; // depth of investigated class
        while (cp = (Class*)next ()) // loop through assoc.
            if ((newDepth=cp->depthOfInheritanceTree ()) > depth)
               depth = newDepth ; // set new maximum
        return depth + 1.0f; // this depth = maximum + 1
        }
```

Listing D.3: Implementation of DIT

Note that this implementation of DIT uses a recursive call to `depthOfInheritanceTree` of all classes inheriting from in order to determine the maximum depth of inheritance of any of those classes. It then simply adds 1 to that depth to count for the inheritance link of the class in focus (C++: this). The maximum depth is initially set to -1 because when not inheriting a class's DIT value should be 0, i.e. -1 + 1.

```
float   Class::weightedMethodsPerClass ()
    {   ULister next (has ()); // iterator for "has" association
        Feature* fp = NULL; // current feature of class
        float methods = 0.0f; // initial weighted methods value
        while (fp = (Feature*)next ()) // loop through assoc.
            if (fp->is (s_M) && fp->is (s_impl))
                methods += ((Method*)fp)->weight ();
        return methods;
        }
```

Listing D.4: Implementation of WMC

In the implementation of WMC we switch - via the "has" meta-association - from the class in focus to its features. The features than may be tested with the "is" method against their meta-properties. Especially the property "s_M" tests whether a description object is of meta-class Method, in contrast to plain features which are of meta-class Feature. The property "s_impl" tests whether a method is implemented or just inherited. As soon as we know the feature is a method we may convert it into one and call its weight method.

E A Practical Cookbook for Software Project Estimation

This appendix is for the practitioner. The new estimation method is described step by step. Explanatory comments are reduced to a minimum. First the method for estimation after preliminary analysis PRE-SM and then the one after domain analysis DOME-SM is explained. Estimates for the preliminary analysis process must be derived heuristically. We recommend a bottom-up approach of 1) setting up a project structure plan by eventually copying from a template like BIO, 2) assigning personnel to the tasks identified, and, 3) summing up the resulting manpower. We further recommend to engage only visionaries, stakeholders and executive decision-makers in preliminary analysis. The process should be completed within at most 3 to 4 months, or else the underlying assumptions about technology and „reality“ will be outdated, due to the rapid change rates in our times.

E.1 Estimation after Preliminary Analysis

1. Describe the system you are about to build in terms of subject areas and goals.

The subject areas - on one side - are to be viewed as anticipated groups of classes (or entity types). A project management system, e.g., could be split into the subject areas of "planning and control", "resource management" and "reporting and documentation". For each subject area the number of classes or "complexity" (with the definition that a subject area with one class has complexity 1) should also be given (approximately of course, and as positive integers, no fractions allowed). Document the subject areas in an unformatted text file using the syntax:

```
Subject Area <name> <number-of-classes> .
```

The goals - on the other side - are to be viewed as rough functional requirements, describable independently from any specific class or subject area. The typical goals for any database application, e.g., are "object creation", "object retrieval", "object update" and "object deletion". Each of these goals just cited equals a complexity of 1 per definition. The goals are to be interpreted as sets of more detailed functional requirements, e.g. the goal "object creation" may - later in the software process - be supported by a standard window layout and user interaction sequence, then by consistency checks before and after the bare object creation task, etc. Also, be sure not to mix those goals for the software system with goals for the software process (e.g. "programs should be written in C++" is not considered a goal in our context). Document the goals either with the syntax:

```
Goal <name> [ <complexity> ] .          (default <complexity> is 1)
```

or

```
Goal <name> = <subgoal> { , <subgoal> } .
```

The latter form introduces a (potentially unlimited) hierarchy of goals. Yet, such a hierarchy is not required. Unless specified otherwise (i.e. a <subgoal> may appear in the <name> role in another statement before or after its subordination), the subgoals are considered to be of complexity 1.

Finally the subject areas and goals must be linked, i.e. it must be stated for each subject area by which goal(s) it is ruled. This linkage can be one reason for building a goal hierarchy, because, maybe, typical groups of goals are attached to more than one subject area. This can elegantly be modelled by attaching the higher level goal, which represents the typical group, to the subject area, instead of attaching each lower level goal to it. The subject area versus goal linkage should be denoted with the syntax:

```
Subject Area <name> isRuledBy <goal> { , <goal> } .
```

The system description thus elaborated is assumed to reside in a file named SD.PRE for the subsequent steps. Any other name can however be chosen. The language used to denote such high level system descriptions is called PRE and was first described in [MOS95]. PRE also honours the single line comment syntax:

```
;<comment>
```

Tokens in PRE should be separated by arbitrary groups of whitespaces (blanks, newlines, tabs, etc.). If names should consist of more than one token use ' as the enclosing character (with " standing for ' within such names) and separate the token with arbitrary groups of whitespaces. If names should contain an exact whitespace pattern (e.g. two blanks) use " as the enclosing character (with \" or "" standing for " within such names)

2. Model the existence of frameworks, libraries, design pattern or previous implementations

For each part of the system description which is (almost) 100% supported either with a framework, "traditional libraries", design pattern or a previous reusable implementation of the system part, enclose the description part in SD.PRE with the statements:

```
;ma-entry: category library
[... the supported system part ...]
;ma-entry: category project
```

Because for each PRE statement the framework support should be clear (either nearly 100% or non existing) this has an influence on the modelling in step 1. On the other hand, the granularity of the PRE language is so low, that there are situations where a partly support of a system model element cannot be avoided (e.g. for the goal "object creation" there may be a DBMS at hand, but a DBMS typically supports the mere persistence operation. There's no integrated and seamless support for the user interface and/or the consistency checks that are also subsumed in this goal). It is recommended not to mark such partially supported elements as "library" elements.

3. Measure the System Meter metric.

Actually this may be the most tedious and boring step or the easiest. In case you try to measure the metric manually, expect hard times. On the other hand it is relatively straightforward to build a small tool to accomplish that laborious task. The author implemented such a program, which he used to accomplish the field study. The measurement step thus may reduced to entering, e.g.:

```
ma
```

or something similar on the command line and waiting for the resulting number to plop out. The following steps assume this number to be called `s` (for size).

The `ma` utility can be obtained on request from the author (moser@acm.org).

4. Estimate the effort and duration for the complete software process (preliminary analysis to delivery).

The effort estimation procedure consists in applying the approximation function of the current corresponding empirical database (see the introduction for comments about empirical databases). If you wish to rely on the empirical database the author has established in his field study, use the following formula to estimate the effort e of the complete software process:

$$e = 0.404 \cdot s + 0.0000753 \cdot s^2$$

Be aware that this number is an estimate, i.e. it is equally possible that the effective effort will be below or above e. The bias is also taken from the same empirical database. The author's samples showed a relative standard deviation of 16.5% (or a ±33% bias at a 95%-confidence level).

5. Check and accommodate for the tailored project model.

The full details of a software process template (= methodology, phase model) would be a prerequisite to discuss this accommodation in depth. For our purposes let us only consider the accommodation on the level of the phases and let us assume that the BIO [MO95b] phasing 1) preliminary analysis, 2) domain analysis, 3) detailed requirements specification, 4) construction, 5) implementation and test, 6) delivery/installation is used. BIO includes all project management, quality management and configuration management activities, therefore the estimates also cover those parts.

Empirical studies (following the ABC estimation strategy outlined in the introduction) have shown linear correlations between the efforts of the different phases. Those correlations can be expressed by the percentage ratios p_1, p_2, p_3, p_4, p_5, p_6 of phase efforts versus total effort. Thus, if the tailored project plan omits a whole phase (e.g. customer installation), the solution is straightforward (i.e. apply percentage calculations). If it omits parts of phases we are into the details mentioned above. The author's survey showed the following phase percentages:

5%, 14%, 18%, 19%, 35%, 9%

The following recommendations (also based on empirical results and standard tailorings for prototyping processes, i.e. special instances of the BIO template process) may also be useful:

- An experimental prototyping process (or subprocess within phases 1 or 2) should be calculated as a third of phase 5 for the system to be prototyped, which is usually a part of the full system. This system part may also be modelled as described in steps 1 and 2 in this chapter.
- An evolutionary prototyping process (or subprocess within phases 2 or 3) should be calculated as a third of phase 3 plus a full phase 4 for the system (usually a part of the full system) to be prototyped. Estimates for the rest of the enclosing software process should then be revised to take the frameworks, design pattern and/or implementations already at hand into account.
- The construction phase (phase 4) should always be reduced by the ratio between the system size s (cf. steps 1 and 2 in this chapter) and a hypothetical size s' (= flat System Meters), which is measured on a version of the system description where no elements are categorised as "library" (actually the system description before entering step 2 in this chapter). This should be done because the construction phase assumes that for all components a design (i.e. a solution pattern) must be "invented" instead of just identified and reused.

6. Check and accommodate for the process dynamics.

The estimation of the „normal“ duration (and the „normal“ average project team size, the ratio of) is then accomplished by consulting an empirical database that correlates the accommodated effort e’ with the duration. The authors coefficients for the duration d are:

$$d = \sqrt{\#\#\#}\ [426'000 + 879 \cdot e] - 653$$

This estimate again has a bias, dA-duration, of its own which was ±37% at a 95%-confidence level in the authors survey. Note that the total dA-duration equals 74%, i.e. spans the positive and negative bias.

Because, usually the customers want a system to be built in no time at all, it is an art of its own to convince customers of a duration d which is substantially above 0. Eventually the duration negotiation process will end up with some

compromise, say d' = d/2. That's what's called "process dynamics". The dynamics can be quantified in form of the acceleration coefficient a:

$$a = (\#\#\#d - d \cdot dA\text{-}duration/4) / d'$$

Investigations conducted by DeMarco, Putnam showed that this acceleration of the process not only means an affection of the duration parameter but also of the effort parameter. Thus corresponding to a d' there's also a e'. Literature proposes:

$$e' = e \cdot a^2$$

We observed in a very small sample set of accelerated projects a correlation of:

$$e' = e \cdot (0.20 \cdot a + 0.792 \cdot a^2)$$

This confirms the behaviour proposed in literature, although in a slightly „softened" quadratic correlation. Both formula for e', however, have an explosive effect to the team size (t = e' / d'). Because we all know that teams are not built in a day, accelerations above 2 (i.e. scheduling half the "normal" time to delivery) are typically impossible.

E.2 Estimation after Domain Analysis

Estimation after Domain Analysis is completely analogous to the steps explained in E.1. The only changes are found in step 1, the modelling, which is more refined, and - of course - in the correlation and precision of the estimated values.

1. Describe the system you are about to build as a class model, use case model and state-event model.

The class model consists of five elements: domain classes, domain associations between classes, data types, function types and consistency rules. Actually data types are a synonym for domain classes although the perception and role for the end-user is different: domain classes are entities of primary interest and life whereas data types are simple (or not so simple) value ranges. Domain analysis is intentionally reduced to the non-technical aspects of the „real world" underlying the system in focus. It is also intentionally reduced in the level of modelling detail. Especially, the detailed attributes and methods of the classes are not modelled in order to avoid ripple effects with the subsequent modelling layer of application analysis. For domain classes simply the approximate number of attributes per data type (or aggregate class) is required. Method modelling is reduced to so called function types which represent basic essential functionalities. Function types may be composed of more basic functionalities, the so-called kinds. If a domain class is modelled as containing an attribute of a function type, this may be viewed as this class having such a generic functionality, i.e. method. Finally, consistency rules are modelled with the involved classes, associations and/or function types.

The formal language to denote the class model is:

```
Data Type <name>
       [ isSubTypeOf <superclass-name> , ... ]
       { contains [ <n> attr ] <data/function type> } .

Function Type <name>
       { ofKind <function-part> , ... } .

Domain Class <name>
       [ isSubTypeOf <superclass-name> , ... ]
       { contains [ <n> attr ] <data/function type> } .

Domain Association one|many <class1> <assoc-name> one|many <class2> .

Consistency Rule <name> = <class/assoc/function-type> , ... .
```

The use case model consists of three elements: use cases, their signals and domain subsystems which are groupings of use cases. The use cases are defined through a unique name and a triggering event (also a unique name). Optionally preconditions for the use case may be modelled by indicating the involved classes, associations and/or function types. The signals are optionally named (for reference in the state-event model) optionally linked to an actor and - most important - tied to the classes, associations and function types involved.

The formal language to denote the use case model is:

```
Use Case <name> isTriggeredBy <event-name>
       [ = <class/assoc/function-type> , ... ] .

Signal [ <name> ] of <use case/function type>
       [ from|to <actor> ] = <class/assoc/function-type> , ... .

Domain Subsystem <name> = <use case> , ... .
```

The state-event model, finally, consists of the two elements of states and transitions. States are defined as uniquely named sub-concepts of classes or superstates. Optionally the other classes, associations or function types involved in the definition of the state may be modelled. Transitions link two states together. Optionally the can be conditional with classes, associations or function types involved in the condition. Also optionally they can trigger or be triggered by signals of the use case model.

The formal language to denote the state-event model is:

```
State <name> isSubStateOf <class/state>
       [ = <class/assoc/function-type> , ... ] .

Transition <name> startsAt <state1> endsAt <state2>
       [ = <class/assoc/function-type> , ... ]
       [ triggers <signal> , ... ]
       [ isTriggeredBy <signal> , ... ] .
```

The system description elaborated is assumed to reside in a file named SD.DOM for the subsequent steps. Any other name can however be chosen. The language used to denote such high level system descriptions is called DOME. DOME also honours the single line comment syntax:

;<comment>

Tokens in DOME should be separated by arbitrary groups of whitespaces (blanks, newlines, tabs, etc.). If names should consist of more than one token use ’ as the enclosing character (with ” standing for ’ within such names) and separate the token with arbitrary groups of whitespaces. If names should contain an exact whitespace pattern (e.g. two blanks) use " as the enclosing character (with \" or "" standing for " within such names)

2. Model the existence of frameworks, libraries, design pattern or previous implementations

[same as for PRE-SM]

3. Measure the System Meter metric.

[same as for PRE-SM]

4. Estimate the effort and duration for the complete software process (preliminary analysis to delivery).

The effort estimation procedure consists in applying the approximation function of the current corresponding empirical database (see the introduction for comments about empirical databases). If you wish to rely on the empirical database the author has established in his field study, use the following formula to estimate the effort e of the complete software process:

$$e = 0.168 \cdot s + 0.0000158 \cdot s^2$$

Be aware that this number is an estimate, i.e. it is equally possible that the effective effort will be below or above e. The bias is also taken from the same empirical database. The author's samples showed a relative standard deviation of 4.5% (or a ±9% bias at a 95%-confidence level).

5. Check and accommodate for the tailored project model.

[same as for PRE-SM]

6. Check and accommodate for the process dynamics.

[same as for PRE-SM]

F Details from the Field Study

The following 36 sheets summarise the measurements obtained from the surveyed projects in Swiss industry. The sheets are laid out in German. We therefore translate the important terms (in the order they appear on the sheet):

German	*English*
Projekt	project
Projektleiter	project manager
Firma	company
Entwicklungsumgebung	development environment
Ertrag	gross income
Messung	measurement
Aufwand	effort
Voranalyse	preliminary analysis
Konzept	domain analysis
Spezifikation	application analysis
Konstruktion	construction / technical design
Fertigung	implementation & test
Rahmenorganisation	organisational activities
Einführung	delivery / installation
Hauptprozess	software process without management activities
Bandaktivitäten	supporting management activities
Dauer	duration
Systemgrösse	system size
Prozessabdeckungsgrad	process completeness
ohne Prüfung	without validation
einfache Prüfung	simple validation
vollständige Prüfung	complete validation
Datenbereiche	subject areas
Funktionalitäten	functional goals
essentielles Klassenmodell	domain class model / essential class model
nicht-essentielle Anforderungen	non-essential requirements
Spezifikationstypen	user requirement types
Datengehalte	models
Systemzustände	system states
betriebliches Klassenmodell	application class model
nicht-funktionale Anforderungen	non-functional requirements
Lösungsmuster	solution patterns
technisches Klassenmodell	technical class model
Testdaten	test (input) data
Testfälle	test cases
Testverfahren	test procedures

[continued]	
Installationsverfahren	installation procedure
Installationsanleitung	installation documentation
Portierung	technical platform port
Übersetzung	translation
Betriebshandbuch	system administrator manual
Benutzerhandbuch	user manual
Evaluationen	product evaluations
Organisationsvorbereitung	preparation of organisational changes
Schulungsvorbereitung	preparation of training
Migrationsvorbereitung	preparation of application migration
Formulare	(paper) forms
Abnahme	acceptance
Schulung	application training of users
Organisationsanpassungen	organisational changes
Planungen	plans
Schätzungen	estimates
Risikoanalysen	risk analysis
Ausbildung Projektmitarbeiter	training of developers
normierter Aufwand	normalised effort
dynamische Kostenmodellierung	dynamic cost modeling
nur falls	only if
aus	from
geschätzter Aufwand	estimated effort
Aufwandmultiplikator	effort inflation factor
entzerrt	normalised with respect to process dynamics
Kennzahlen	key figure
finanzielle Produktivität	financial productivity
fachliche Produktivität	technical productivity
Teamgrösse	team size
Schätzqualitätsfaktor SQF	estimation quality factor EQF
Termintreue	deadline hit factor
geschätzt	estimated

getunt tuned