# Measuring and Improving Component-Based Software Development

**Pentti Virtanen**

TUCS

Turku Centre for Computer Science

TUCS Dissertations

No 45, March 2003

# Measuring and Improving Component-Based Software Development

Pentti Virtanen

To be presented, with the permission of the Faculty of Mathematics and Natural Sciences of the University of Turku, for public criticism in the Auditorium of the Department of Information Technology on April 12th, 2003, at 12 noon.

University of Turku
Department of Computer Science
FIN-20014 Turku Finland

2003

ISSN 1239-1883

ISBN 952-12-1137-7

Painosalama Oy, Turku Finland

UNIVERSITY OF TURKU
Department of Information Technology

VIRTANEN, PENTTI: Measuring and Improving Component-Based Software Development

Ph. D. thesis, 172 pages and 27 appendix pages
Computer Science
March 2003

--------------------------------------------------------------------------------

*Component Reuse Metrics (CRM) is a new approach to estimating the effort of component-based software development. A product structure describes a software system which is made of reused components. The history data of projects is stored in a repository. The average effort of the components is used to calculate the baseline effort of the forthcoming project. The characteristics of the project and human effects are assessed and used to correct the baseline effort in the CRM-calculations. The effect of the changes in the project takes into account differences between the planned product and the final product. The process effect includes the impact of process changes between the new project and the projects in the repository. The team effect estimates the effort which the team members spend in their mutual communication. The expectation value of additional effort due to project risks is calculated in the risk effect. The human effects are the skill effect and the motivation effect. CRM equations calculate the efforts of tasks in the project to be estimated. The assessment of the estimation method and current practices was done using a survey and by case studies in companies that tested CRM.*

*CRM is closely related to the second main contribution in this thesis, namely new approaches to improving productivity in software development. The criteria for reusability are understandability, ease to find, adaptability and trustworthiness. The strategy for reusability is to combine components that are based on generic abstractions. Separate generalization-specialization-structures to both functions and traditional objects are used to improve adaptability. An analogy from natural languages, a new concept, verb classes, is introduced. An example is included to demonstrate the feasibility of this approach.*

TURUN YLIOPISTO
Informaatioteknologian laitos

VIRTANEN, PENTTI: Komponenttipohjaisen ohjelmistokehitystyön mittaamisesta ja parantamisesta

Väitöskirja, 172 sivua+27 liitesivua
Tietojenkäsittelytieteet
Maaliskuu 2003
--------------------------------------------------------------------------------
*Component Reuse Metrics (CRM) on uusi lähestymistapa komponenttipohjaisen ohjelmistokehitystyön työmäärän arviointiin. Tuoterakenne kuvaa ohjelmiston, joka on valmistettu uudelleenkäytetyistä komponenteista. Projektien historiatiedot talletetaan hakemistoon. Komponenttien keskimääräistä työmäärää käytetään tulevan projektin perustyömäärän laskemiseen. Projektikohtaisten ja inhimillisten tekijöiden erityispiirteet arvioidaan ja niitä käytetään CRM-laskennassa korjaamaan perustyömäärää. Projektin muuttumisen vaikutus huomioi suunnitellun ja lopullisen tuotteen välisen eron. Prosessi-vaikutus sisältää prosessimuutosten vaikutuksen uuden projektin ja hakemistossa olevien projektien välillä. Tiimi-vaikutus arvioi työmäärän, jonka tiimin jäsenet käyttävät keskinäiseen kommunikointiin. Riskivaikutus laskee projektin riskeistä aiheutuvan lisätyön odotusarvon. Inhimilliset vaikutukset ovat osaamisvaikutus ja motivaatiovaikutus. CRM yhtälöt laskevat arvioitavan projektin tehtävien työmäärät. Nykykäytäntöjen ja arviointimenetelmän arviointi tehtiin kyselyillä ja tutkimuksilla CRM:ää testanneissa yrityksissä.*

*CRM liittyy läheisesti toiseen tämän väitöskirjan pääkontribuutioon eli ohjelmistokehitystyön tuottavuuden parantamisen uusiin lähestymistapoihin. Uudelleenkäytön kriteerit ovat ymmärrettävyys, löytämisen helppous, mukautettavuus ja luotettavuus. Uudelleenkäytön strategia on yhdistää yleisiin abstraktioihin perustuvia komponentteja. Erillisiä erikoistamisrakenteita sekä funktioille että perinteisille objekteille käytetään parantamaan mukautettavuutta. Uusi luonnolliselle kielelle analoginen käsite verbiluokat esitellään. Esimerkki osoittaa lähestymistavan hyödyllisyyden.*

# Acknowledgements

This work was carried out at the University of Turku and Turku Centre for Computer Science as I was participating in a programme, where the participants work full time in the industry and carry out their PhD studies at the same time. The programme began in 1994 and it has been the supporter of this study ever since. The idea for the study had existed from the beginning of the 1990's when I led my first projects which utilised object-oriented programming.

I am greatly indebted to my supervisors Anne Eerola, Ph.D. from the University of Kuopio, and Jukka Teuhola, Ph.D. from the University of Turku for their courtesy, encouragement and advice in all phases of the study throughout these years.

The University of Turku and The Turku Centre for Computer Science have given me grants to travel to conferences at which I met several researchers interested in the same area. The conferences gave me encouragement and greater confidence in the ideas I had.

The Finnish Information Processing Association provided the addresses for the survey carried out in this study. The respondents gave their valuable time answering the survey. In particular, I want to thank the participants in the case studies for their valuable contributions on CRM.

Finally, I wish to express my deepest and warmest gratitude to my family, Marjatta, Heikki and Heidi, for their support and patience during the period of this research.

# Contents

# Preface

I saw the potential of this study in 1990 when I estimated the first application projects which utilised an object-oriented programming language. The proponents of the language (Object/1) argued that it was highly productive. It was obvious that the method at that time used to estimate the needed effort was not effective. No better methods were available, therefore, there were no ways to validate the claims of high productivity. The estimate of the project was accomplished by counting the windows and controls in the windows.

The original idea of this thesis work was to evaluate the benefits of object-oriented methods [Virtanen 1998b]. This in turn led to measuring the development and to studying productivity. The first version of the new estimation method was called Object Component Process Metrics [Virtanen 1998a]. The method was renamed Component Reuse Metrics because the method counts components instead of classes and objects [Virtanen 2000b]. At the same time the calculation rules and the evaluation of human aspects of the development have been improved.

The productivity improvements of component-based development are based on the reuse of components. Adaptability was seen as the enabler of reuse [Virtanen 1999b]. Verb inheritance is an architecture which creates very adaptable components [Virtanen 1998b]. The study of verb classes proposes an implementation of this idea [Virtanen 2000c].

My licentiate thesis contained the collected improved versions of the above ideas [Virtanen 2000]. The practical evaluation of Component Reuse Metrics [Virtanen 2001] depicts the results of the questionnaire with regard to its basic assumptions. This thesis is an extended, updated and uniform presentation of the above ideas.

# 1 Introduction

## 1.1 Topics

Measurement is important because a scale is needed to manage the efforts required to accomplish or improve something. The main contribution of this study is to introduce a new method, Component Reuse Metrics, CRM, for estimating the effort of component-based software development. CRM is a new way to combine human related effects and effects related to the size and structure of the application to be developed. CRM also suggests a new way to measure the size of an application. It counts the components of the future application instead of lines of code or calculated numbers such as function points. The total effort is the sum of the efforts of the components. In this method effort contains the thinking work of each individual and their co-operation during a development process. The skills and motivation of the staff are important parts of the success of development projects.

The study compares traditional estimation methods, which are based on standard units such as lines of code and function points, to CRM. The solutions to problems involved in each of them are addressed in developing Component Reuse Metrics.

The second issue in this study is the productivity of component-based development. Reuse is important in increasing the productivity. The key factors in reusability are understandability, ease to find, adaptability and trustworthiness.

The foundations of reuse are studied in order to reveal the connection between reuse and the properties of components. Several factors influencing adaptability are discussed. The most important of them is to have good abstractions. Object-oriented methods and programming languages have several mechanisms to support the creation of extensions. They are analysed in order to find improved mechanisms.

Finally, this study presents a new way of increasing reusability. In an object-oriented approach inheritance is the main technique for reuse. In linguistics, generalisation and word derivation are much more versatile. An object model is at its most comprehensible when it utilises the structures of natural languages and makes it possible to visualise these structures. This is a way of expanding the current limits of object modelling. It has connections to the analysis and design of frameworks and patterns.

## 1.2 Motivation

### 1.2.1 Why is estimation needed?

A *measure* is a standard or unit of measurement; the extent, dimensions, capacity etc. of anything, especially as determined by a standard; a result of measurement [Baumert 1992]. Measures are used to evaluate properties of something being measured, such as quality, complexity or effort, in an objective manner. In this study a *metric* is a synonym of a measure though in some metrics programs measures are directly observable data and metrics are results of combining several measures [Poulin 2001]. A *measurement* is an act or process of measuring something [Baumert 1992].

In software engineering, metrics are needed to calculate the estimates of the development cost and to decide if the proposed software should be produced or not. A process used in making such a decision is called *a feasibility study* [Kendall 1987]. The purpose of a feasibility study in the preliminary analysis is to estimate the benefits of the software and compare them with the costs of the development. The difficulty with the estimation is that most of the required data are not yet available. The requirements are not fully understood and the technical solution is obscure. As most of the costs of the software development are related to human effort, this study focuses on effort estimation. The cost calculations, e.g. costs of purchasing hardware, are not considered.

At the beginning of any project, resources and the schedule of their usage should be assigned. The estimation method should help to allocate people and tools to the tasks. When the project is running, its progress (costs and outcomes) should be measured. The estimation method should give the means to do this objectively. When new features are added to the project, their impact on the cost and schedule should be estimated.

The estimation can be used to evaluate the productivity of different project teams, methods and tools. Using the evaluation, the methods, processes and tools can be improved. The usage of metrics is an essential part of the optimising level of *t*he *Capability Maturity Model (CMM)*, where continuous process improvement is enabled by quantitative feedback from processes [Paulk 1993]. The CMM models have been developed for various purposes. Capability Maturity Model - Integrated (CMMI) harmonises them under one framework [CMMI 2002].

### 1.2.2 Goals of component-based development

The goals of component-based development are, according to Brown, to utilise the best solutions and to increase productivity [Brown 2000]. Component-based development manages complexity and improves consistency. Supporting parallel and distributed development is included in the component-based development environments. Increased productivity reduces delivery time, time-to-market, and maintenance costs. The use of components increases the controllability and visibility of the development projects.

This study emphasises the role of visible, consistent components in development projects. Productivity increase through reusable components is the second topic.

## *1.3 Methods of the study*

The research method used this study is *constructive*. Järvinen defines constructive research as research that tries to answer questions: Can we build a certain artefact, how ought we do it and how useful is a particular artefact. We can also ask, what type a certain artefact ought to be [Järvinen 1999]. Programs, methods and models are examples of software engineering artefacts.

The research question of this study is: *How to measure and improve component-based software development?* The question of measuring is handled first. A measuring method, CRM, is constructed. The method is based on the hypothesis that certain factors are important in the software development processes. The developed model of the cost structure of the software development is based on practical experience. The equations for estimating the development effort are heuristic. The study does not try to find a decisive validation of the correctness of the method. Instead, the results of empirical research are used to ensure that the parameters of the cost model are reasonable and that the calculations can be accomplished. The weights of the parameters are also obtained empirically. The method is groundwork to construct practical tools to measure component-based software development.

Theory-testing research methods [Järvinen 1999] are used while assessing whether the developed estimation method is possible and useful in practice. This part of the study includes a survey and case studies. The goal of the survey is to look into current estimation practices and to collate the factors involved in software development effort. Though the survey is linked to CRM, the views of experts that have been obtained are also generally useful. The case studies evaluate whether CRM can be used in practice.

According to the hypothesis of this study, the groups of factors are *components, process, project change, team, risks, skills and motivation*. Components are identified and counted and each group of factors is studied in a separate estimation form, which quantifies the factors by using a 5-level numerical scale. The effort due to each of these groups of factors should be measured in project tracking. The set of questions used have been developed with regard to the factors which have influence on the effort of component-based software development.

The second part of this study is to develop methods of improving component-based software development. It is also developed from constructive research. The produced measurement method is used in order to find out what kind of components are needed in improved component-based software development, where a decrease in the effort required is emphasised. Finally, an example program shows that such components can be built.

## 1.4 Definitions

### 1.4.1 Component-based software development

A *module* is a unit of software composition. It is an early example of a software component. In the object-oriented approach, classes provide the basic form of the module [Meyer 1997]. "A *component* is a high-quality type, class or other work-product, designed, documented and packaged to be reusable" [Jacobson 1997]. Another definition emphasises composability:

> A *software component* is a unit of composition by third parties and it contains only contractually specified interfaces and explicit context dependencies. It can be deployed independently [Szyperski 1998].

*Component-based software development* is an approach in which systems are built from well-defined, independently produced pieces by combining the pieces with self-made components [Brown 2000]. Some definitions emphasise that components are conceptually coherent packages of useful behaviour, while some others state that components are physical, deployable units of software which execute within a well defined environment [Brown 2000]. There are several kinds of components and the *granularity* of these components can vary [Herzum 2000]: A *distributed component* is a possibly network addressable component which has the lowest granularity. It may be implemented as an Enterprise JavaBean, as a CORBA component, or as a DCOM component. A *business component* implements a single autonomous business concept. *A business component system* is a group of business components that co-operate to deliver a cohesive set of functionality and properties required in a specific domain [Herzum 2000]. A *tier* is a group of components in the same layer. The classic three-tier architecture consists of the presentation tier (windows, reports, …), application logic tier (business rules of the application) and resource tier (persistent storage mechanism) [Larman 1998]. As this study emphasises the reuse of conceptually coherent packages of useful behaviour, the definitions of Szybersky and Jacobson are the most relevant. The independence of the programming language and its deployment as executable binaries is emphasised in other definitions.

The reuse requirement and the goals of developing component markets change software development radically. One part of the effort of programming is the effort of finding, understanding and reusing appropriate components. On the other hand, additional effort in documenting, configuring and releasing the component is required, if the component is made for sale. Traditional effort estimation methods assume that applications are developed from scratch without considerable reuse. Though reuse effects have been added to these methods later, the paradigm has not been changed. The meaning of the word *reuse* in software development is the usage of some previously developed component in constructing new software. In other words, *reuse* is the process of adapting a generalised component to various contexts of use [Basset 1997].

Every software product has some fixed features which determine what the software product is. *Reuse* considers utilising these fixed properties during the *construction time* of the software product. In *use* developed application calls the components at *run time* according to the choices made by the person utilising the application.

A *use case* defines interactions (sequences of actions) between a user or another kind of an actor and a software system [Booch 1999, page 222]. The use cases are used in designing the components of the application [Cheesman 2001]. *Adaptability* is the ability of a component to adapt to different reuse situations. Complexity measures the number of components and their interconnections. Measures of complexity are typically based on internal product attributes, such as cohesion and coupling [Golgberg 1995]. *Coupling* is a measure of the degree of interaction and interdependence between software components [Constantine 1995]. *Cohesion* measures the degree to which a component comprises a well-defined functional whole [Constantine 1995].

This study focuses on the measurements of the component-based software development process and the possibilities of improving reusability of components. *A software process* is a set of activities, methods, practices, notations and transformations that software engineers use to develop and maintain software and associated artefacts (e.g. project plans, design documents, code, test cases and user manuals) [Paulk 1993]. A *project* is an instance of a process. It assigns the resources, such as the team, tools and facilities to the project phases. A *phase* is a period during which a project team focuses primarily on a specific kind of job, such as requirements engineering, design, construction or release [McConnell 1998]. A *task* is an element of a software project [Humphrey 1995]. Tasks are scheduling and tracking units of a project. A *method* is a specific way of carrying out an activity, a step-by-step procedure that takes you from a set of inputs to an end result or an interim result [Goldberg 1995, page 48].

Component-based development often uses *object-oriented* methods, for example, Object-Oriented Development by Booch [Booch 1991], Object-Oriented Analysis by Coad and Yourdon [Coad 1990], Responsibility-Driven design by Wirfs-Brook, Wilkerson and Wiener [Wirfs-Brook 1990], Modelling the World by States by Shlaer and Mellor [Shlaer 1992] and Object Modelling Technique by Rumbaugh, Blaha, Premerlani, Eddy and Lorensen [Rumbaugh 1991]. Unified Modeling Language (UML), which is a result of the co-operation of Booch, Rumbaugh and Jacobson, includes model elements, notations and guidelines for describing object-oriented software [Booch 1997]. The Object Management Group (OMG) is an organisation that establishes and promotes standards of object-oriented software development [Herzum 2000]. There is no single definition of *object-orientation*. It depends on the frame of reference: programming language, user interface, application, database or analysis or design methods. A data type oriented definition is “Object-oriented software construction is the building of software systems as structured collections of possibly partial abstract data type implementations [Meyer 1997]." The general definition used in this study is [Goldberg 1995]:

> “Something is *object-oriented* if it can be extended by composition of existing parts or by refinement of behaviours. Changes in the original parts propagate, so that compositions and refinements that reuse these parts change appropriately.”

There are several general concepts to which object-orientation is connected. These are discussed below. A *class* is an implemented abstract data type. It serves both as a module and a type (or a type pattern if the class is generic) [Meyer 1997]. A *generic* class is a class which has formal parameters representing its types [Meyer 1997]. An *attribute* is a data element of a class [Goldberg 1995]. An attribute can be an object. An *instance variable* is a synonym for an attribute. An *operation* is an implementation of a service that can be requested from any object of a related class in order to affect the behaviour of the object [Booch 1999]. A *method* is an implementation of an operation [Booch 1999]. In C++, the synonyms *member function and member variable* are used for an attribute and a method [Andrews 1993].

*Inheritance* is a way of propagating properties and behaviour. A subclass inherits properties and behaviour from its super classes. Object-oriented programming languages have it as a built-in property. The concept *generalisation/specialisation* is semantically better than the concept inheritance, because the child class should always represent a specific version of the general abstraction of the parent class [Booch 1997]. In programming it is also common to use this propagation system as a way to copy code from one place to another without any conceptual relationship. This complicates the maintenance of the applications, because developers cannot rely on the "is-kind-of"-relation.

*Encapsulation* is a way of representing a cohesive set of properties and behaviour as a single unit. Its background is in an idea taken from "software integrated circuits" and uses an analogy from electronics. Thus the software can be constructed by connecting an appropriate set of separately manufactured software parts. *Information hiding* is the ability of the author of a class to specify that a feature is available to all clients, to no client, or to specific clients [Meyer 1997]. Usually, components are seen as packages of classes. A *contract* is a set of precise conditions that establish the relations between a component and its clients [Meyer 1997]. Components have explicit interfaces to interfere with other components, however, they hide their internal structure and implementation. An *interface* is a collection of operations that are used to define services of a class or a component [Booch 1999]. Interfaces are important parts of the documentation of the components. A *signature* is the name and parameters of an operation [Booch 1999]. The definition of an interface contains operations, semantics, preconditions, postconditions, and invariants. Interfaces and their dependencies can be expressed formally:

$$interface = \{operation, \{parameter, type, [in \mid out]\}*\}*$$
$$dependencies = \{required\_interfaces, offered\_interfaces\}$$

*Polymorphism* means that one can specify operations with the same signature at different points in an inheritance hierarchy [Booch 1999]. *Composition* is a logical combining of several objects to form a new conceptually distinct object [Goldberg 1995].

### 1.4.2 Reuse

The *boxes of reuse* is a paradigm which describes the independence of the reused components. A reusable component can be seen as a box which contains the code and the documentation. Though terms such as black box and white box are used, reuse is not only black or white; there are tones of grey too.

In *black box reuse* only the outside can be seen [Goldberg 1995, page 208]. The reuser sees the interface, not the implementation of the component. The interface contains public methods, user documentation, requirements and restrictions of the component. If a programmer were to change the code of a black box component, compiling and linking the component would propagate the change to the applications that reuse the component. As the users of the component trust its interface, changes should not affect the logical behaviour of the component. The clients will get what the contract promises only if the postcondition is true after the changes to the internal implementation.

In *glass box reuse* the inside of the box can be seen as well as the outside, but it is not possible to touch the inside [Goldberg 1995, page 208]. This solution has an advantage when compared to black box reuse, as the reuser can understand the box and its use better. The disadvantage is that it is possible that the reuser will rely on a particular way of implementation or other factors that are not in the contract. That can be hazardous when the implementation changes.

In *white box reuse* it is possible to see and change the inside of the box as well as its interface [Goldberg 1995, page 208]. A white box can share its internal structure or implementation with another box through inheritance or delegation. The new box can retain the reused box as such or it can change it. It is necessary to test anything new that is created or changed.

*Transformational reuse* is an approach where a developer provides the description of what is wanted and a black box program generates the implementation details [Goldberg 1995, page 209]. *Application generators* use this approach.

Cloning is easy to do. One simply copies a chunk of code and pastes it to another place. It is not strategic reuse because the copy can change without reflecting the change back to the original code. When the amount of code increases, the effort in maintaining it also increases.

It is possible to reuse personnel and every software artefact such as models, plans, designs, code, and components [Meyer 1997]. As software artefacts are mental models, they can be reused as such. Reuse of an idea is easy in theory but hard work is needed in the implementation.

*Application reuse* is a large-scale reuse. Using COTS-products instead of producing the software is also a form of reuse. Such software typically has a large number of parameters, which are used to adapt the software to the varying needs of the stakeholders.

Object-oriented software has two mechanisms for adapting its components: *extensibility* and *inheritance* [Carroll 1995, pp. 48]. Extensibility includes other kinds of flexibility than inheritance. The generic classes, also called *templates*, are the most important of these. Different kinds of patterns are important extensibility mechanisms in higher levels of abstractions.

*A software framework* is a set of related objects which provide a well-defined set of services for the reuser. It can be seen as a frame to which a developer can add functionality. The most important difference between a program library and a framework is that the framework calls the added code, but when using program libraries the code developed by the user calls the routines in the library.

A *design pattern* is a description of communicating components which are customised to solve a general design problem in a particular context [Gamma 1995]. Each pattern solves one design problem. The problem and its solution are described as a cognitive model, which can be adapted to solve similar problems. Automation has been developed to support the use of design patterns [Florijn 1997].

Typical *class specification reuse* includes adapting the class to another context using specialisation. The inheriting class can have more member variables and member functions than the super class. It is also possible to change or modify the member functions in the inherited class.

## 1.5 Estimation of software development

### 1.5.1 Goal

The goal of estimation is to find out the costs and timing in a software development project. Software estimation methods focus almost solely on the effort because the effort is the major cost factor in software development.

Timing depends on the effort, available resources, and required time-to-market.

### 1.5.2 Desired properties of the measures

In order to be useful, the measure must be valid, reliable, accurate, and practical. A measure is *valid* if it accurately characterises the attribute it claims to measure [Fenton 1997]. A measure is *reliable* if the application of its algorithms produces consistent results. Thus a measure can be reliable but not valid. The measure is *accurate* if the result is close enough to the true value. The measure is *practical* if it is easy to use and requires a small effort in real-life projects. A practical measure does not require excessive measuring effort or data which is not available at the time of the measurement.

The issue of *objectivity* against *subjectivity* is important. Objective measurements are *independent* of the person who measures and are *repeatable* and they should also cover all of the important facets of the software development. Measures, such as lines of code, classic complexity metrics, object-oriented metrics and component counts, can be measured quite objectively after the implementation but many important measures, such as human skills and motivation, always depend on the judgement of the estimator.

Quite often the desired properties cannot be measured directly. Instead, some other measurement is done and the target metrics are derived from this. In this case, the validity of the reasoning must be assured very carefully. For example, in geometry we can estimate the height of a tree by measuring the length from the observation point to the bottom of the tree and the angle from the observation point to the top of the tree. Now there is a function to calculate the height of the tree. In this case the validity of the function is not questionable. In measuring human behaviour the reliability of the measurement is always questionable. The observations have influence on the measurement. The observations are typically not repeatable because humans do not always behave in the same way. There is no direct function because it is difficult, if not impossible, to predict how humans act. As the real essence of software [Brooks 1995] i.e. mental models, cannot be measured, the outcomes of the development process must be measured instead. If the mapping between mental models and outcomes is incorrect, the obtained measure is invalid. If the calculation process is not unambiguous, the result will be unreliable. Estimates made without knowing all the functionality of the software to be developed are at best inaccurate, if not invalid.

We can use more and more accurate measuring units but the phenomenon itself will limit the usefulness of the accuracy. For example, the length of a rail changes if the temperature changes. There is no point in making the metrics too accurate because the random fluctuations of the phenomenon can be larger than the intended accuracy of the metrics.

In order to be practical the collection of data should be automatic. Without computer-supported data collection, data for metrics would not be collected at all in practice. The needed data should be available in the early phases of the development when important decisions are made. A method which requires an excessive number of parameters also requires an unnecessary amount of effort in order to accomplish the estimation. CRM is not too laborious because only its central parameters must be actively used.

Dependencies on programming languages and development methods should not restrict the usability of the metrics. The same applies to dependence on the project size.

Estimates for measures are based on statistics which are collected from previous projects and stored in a repository. They contain implicit assumptions about the properties of projects that are to be estimated. The problem is that new projects are not the same as old ones. New tools and methods can be used or at the very least the staff will have learned how to accomplish the work better. The similarity of the project to be estimated and that of previous projects can be performed better if in-house statistics are used instead of world-wide statistics of all available projects. However, in this case random effects have a greater influence on the estimate because the size of the sample of the in-house projects is smaller than samples from a more extensive set of projects in industry.

The solution suggested in this study is to isolate the factors relating to the effort inside the development projects and use the statistics from them in new estimates. The quality of these measurements will naturally have a large impact on the quality of the estimate.

### 1.5.3 Product metrics and process metrics

Metrics can be divided into two groups. *Product metrics* measure the artefacts of the software project. *Process metrics* measure the features of the development process [Donaldson 1997]. Both of these are needed in the development of products and processes and to make good effort estimations.

The product metrics approach focuses on the end product itself, not on how it is produced. The properties of the product are analysed from the code, from the documentation, from the user's external view of the product and/or from the profits of the business. The idea is to separate the size and complexity of the product from the productivity of the process. Though code is used to analyse the complexity, each ramification is considered to be due to a requirement of a stakeholder. Thus dependency on the used programming language can be taken into account by using a language specific productivity ratio, if needed.

A large volume of code is not an indication of good productivity. This is especially so in component-based development where reuse is a major issue. A product with less code may produce more user functionality with less cost than a product with a large amount of code. In assessing different paradigms, it is essential to look at the product from the user's point of view. The value of a system for a user is not a consequence of larger program size and complexity. The usability of the system may be better in a product which has fewer features but where the features it does have are important, than in a product which has a large number of unimportant features.

Product metrics are also quality metrics. They can be used to make an analysis of the *maintainability, portability, usability, and correctness*, to mention a few of its applications. These attributes are closely connected to each other. For example, better maintainability is correlated to better correctness because higher productivity in making corrections adds the probability that the errors are corrected. Cost estimation is connected to quality.

Process metrics measure the attributes of the development process. A process model defines a set of activities which occur over a period of time. Typical attributes of the process metrics are the number and the duration of these activities. The time needed for analysis and the number of bugs found in the system are examples of the attributes to be measured. Variables such as staffing levels or effort, both as a function of time, may be used directly or may be included in prognostic models to forecast other characteristics. Process improvements are normally related to each other and it is difficult to create laboratory environments to measure individual process factors. In most cases process metrics are also connected to product metrics.

### 1.5.4 External and internal measures

It is important to distinguish the measures that can be detected by the users of the application from those measures that measure the internal properties of the products. The former properties may be called *external* measures and the latter ones *internal* measures [Meyer 1997]. Examples of the former are the number of windows in the user interface and the number of use cases in the system. Examples of the latter are the number of classes, the average number of methods in the classes and the average number of lines of code in the methods.

Henderson-Sellers enumerates metrics for the size of the program, data structure, control flow complexity, inter-module coupling and modular cohesion as examples of internal factors [Henderson-Sellers 1996]. Complexity, understandability, modifiability, testability and maintainability are his examples of external factors. Nowadays, *interoperability* is very important as software is often composed of third-party components [Szyperski 1998]. The advantage of the internal factors is that they can be calculated objectively and compared to external factors. Internal factors can be useful as estimators or predictors of external values if the link between internal and external factors can be defined.

*Complexity* denotes the degree of mental effort required for comprehension. The s*tructural complexity* measures the structure (for example, control flow structure, hierarchical structure or modular structure) of the software [Fenton 1997]. The *inherent size and complexity* of the problem to be solved is the portion of complexity which can be determined without assigning people to the development project.

Productivity can be increased by avoiding unnecessary code and by decreasing the effort spent on the remaining code. The latter can be accomplished, for example, by using better tools. The selected productivity measure can take that into account only if it is based on an external view of the product. An internal measure, such as lines of code in an hour, suggests bad practices such as excessive use of copying and pasting of code. The amount of code increases rapidly but the functionality of the software does not.

### 1.5.5 The unit of size

The entity of *software size* is widely used but still obscure. One valid definition is to define it as the size in bytes of the installed software. In this scenario the entity is restricted to the measure space requirements of the software. In proposed metrics software size is used to estimate the work needed to produce the software. Here the typical unit is lines of code (LOC). If effort estimation is the ultimate goal then a function, called productivity, between effort and size must exist:

Equation 1.

$$Effort = Productivity\_function(Product\_size, productivity\_parameters)$$

*Productivity* is a metric of efficiency measured by comparing the amount of code produced with the time taken or the resources used to produce it. Alternatively, productivity can be measured related to other factors, for example, to profit. Additivity is a desired property of the size. If additivity holds, the measure can be used as a progress indicator during the project. In equation form the total effort can be calculated by either dividing the size of the whole product by the productivity coefficient or by summing the efforts of the tasks of the project. Formally:

Equation 2. $$Effort = \frac{\sum_{p \in parts\_of\_product} Size_p}{Productivity\_coeffiecient} = \sum_{t \in tasks\_of\_project} Effort_t ,$$

where

- $Size_p$= size of the part p of the product,

- $p$= part of the product,
- $Effort_t$ = effort of the task t of the project, and
- $t$ = task of the project.

If the previous equation were true, the productivity would be a linear function of the size and combining the parts would require no effort. Here its unit is the unit of size per time unit, for example, lines of code per hour, or function points per hour. The equation is true only if a size unit corresponds to a fixed effort. Using lines of code is based on that assumption. A function point is a computational size unit in which each counted item (for example an input or an output) is weighted by a weight proportional to the effort.

In the real world software products are composed of many different kinds of components which each have different complexities. The proportional amounts of the components and the productivity of reusing these various components vary widely. For example, the productivity of generating code for a business application is higher than the productivity required to create code for mathematical algorithms. Therefore, it is important to know how much difficult code is needed in each application.

An easy solution to the previous problem is to exclude the concept of size and concentrate on calculating the sums of efforts needed to do the tasks in the project. In this case, productivity cannot be calculated because there is no connection to the inherent size and complexity of the problem. A solution used in car maintenance and repair is to define a standard effort for each standardised task that is visible to a customer, for example, to state that an oil change will take 15 minutes, which is normally enough time for the task.

In this study the components of the applications are handled separately in order to take the variations of the complexity and productivity into account.

### 1.5.6 Task assignments

The use of the person-month concept is questioned because people are not interchangeable [Brooks 1995]. Each person has different productivity because his/her personality, knowledge, experience and motivation differs. If a developer writes a 1000 line program in 40 hours, 5 developers won't necessarily be able to do the same task in 8 hours, and by logical extension 40 developers won't necessarily complete the task in an hour [McConnell 1996]. This is because it is not possible to perfectly partition a programming task and construct the different parts simultaneously. Adding manpower adds effort due to the need for co-operation and communication. It is not intuitively clear whether this added effort belongs to the inherent complexity of the task or to productivity.

In early estimates the task assignments cannot generally be known and averages must be used. However, if the assignments are known, the use of them increases the accuracy of the estimate.

For example, consider two tasks, named T1 and T2, and two developers, named D1 and D2. D1 is familiar with task T1 and D2 with task T2. The fictitious example in Table 1 shows us that the total effort with assignment D1 to task T1 and D2 to task T2 is 30 hours and assignment D1 to task T2 and D2 to task T1 is 250 hours. The work difference is due to the time needed to learn the task up to the same level as the other developer. If the complexity and the productivity are estimated separately, what are the inherent complexities of the tasks and what is the productivity? Clearly there is the trivial solution. The estimate of the total effort of T1 and T2 is 138 hours because the complexity is 400 function points and average productivity is 2.9 function points per hour. Function points are used in this example, though this study does not advocate the use of function points. The estimate of the example is not accurate because the productivity range was from 1 FP/ h to 15 FP/ h.

Table 1. Influence of task assignments to the total effort.

| | Developer D1 | | Developer D2 | | Complexity (function points : FP) |
|---|---|---|---|---|---|
| Task T1 | 10 FP / h | 10 h | 1 FP / h | 100 h | 100 FP |
| Task T2 | 2 FP / h | 150 h | 15 FP / h | 20 h | 300 FP |
| Average | 2.5 FP / h | | 3.3 FP / h | | 2.9 FP / h |

In this study project and human effects are calculated explicitly. As the same task is easier for a developer who is familiar with it, it is not appropriate to estimate the effort without considering task assignments. Cognitive complexity metrics provide another solution, which is arrived at by including the familiarity with the tasks in the calculation rules of the complexity metric [Cant 1994].

## *1.6 Related work*

### 1.6.1 Effort estimation

The measures most suited to component-based development and measures which are used widely, are presented here briefly and compared to CRM in chapter 4 (page 65). The discussed metrics are:

- Number of lines of code (LOC) is easy to collect and has been collected historically, going back over the last 20 years [Poulin 1997, Conte 1986].

- Constructive Cost Model, COCOMO, [Boehm 1981] and COCOMO II [Boehm 2000] calculate the effort required for software development using the size of the product in the LOC.
- Function point analysis, FPA, is a widely referred estimation method [McConnell 1996, Dreger 1992]. It is a measure of program size that is often used in the early phases of a project.
- PROxy-Based-Estimating, PROBE, method uses classes as proxies of lines of code [Humphrey 1995].
- Complexity metrics study the psychological complexity of programs [Henry 1981, Halstead 1977, Henderson-Sellers 1996, McCabe 1976]. Lines of code are not considered equal, some of them are more difficult to produce, modify and understand. That difficulty is correlated to the effort required in the development.
- Suites of object-oriented metrics comprehend the applications according to the object paradigm [Chidamber 1991, Lorentz 1994, Putkonen 1994].
- Task based estimation is the choice of industry for software project estimation. The project is broken down into tasks, forming a structure called project breakdown structure or work breakdown structure [Cantor 1998, Metzger 1996, Symons 1991].

### 1.6.2 Effort estimation in construction business

As the construction of houses and buildings has many similarities to software development, their estimation methods are briefly introduced. For example, both software and buildings can be produced in projects in which humans assemble prefabricated components. The goal is to find useful analogies in order to improve the estimation methods of software development. The main difference between these businesses is that a software product is abstract and intangible and a building is tangible. Rewriting software is cheap and easy compared to knocking down a building and constructing it again because the cost of materials is much less than the actual construction. In the construction business the effort is calculated component-by-component using component specific calculation rules. Productivity is not a general coefficient that is applied to the size of a building. General size metrics, such as square-metres and cubic-metres, are used only to compare overall measures of similar buildings.

Nowadays, even the most exceptional buildings are made using factory made components. Component standardisation and usage is so widely spread that books are published which contain catalogues of these components. Further, they contain equations from which the amount of work required to assemble these components can be calculated [Ojala 1989].

The equations for work effort calculation are based on the components. As the effort is different in different project or human-related situations, it must be adapted for the forthcoming project. For example, the effort of using ready-made concrete is different in wintertime and in summertime. The unit is hours per cubic metre because ready-made concrete is sold in cubic metres and is delivered by a special lorry. The team needed for the assembly work, for example, is defined as being two professional workers and one handyman. By defining the team the cost differences can be taken into account. The method by which the job is done is also implied, it is based on using a crane. The estimates for making floors and walls use different units, teams and tools.

Next, granularity of the components varies. The wall can contain only a single wall-element, or it can consist of several wall-elements or it can be constructed from bricks. In the feasibility study phase components of coarser granularity are used. For example, we consider the wall to be a single component, not wallpaper alone [Hyttinen 1987]. The wall may consist of bricks, an outer board, a thermal insulator, a wooden skeleton, an inside board and finally the wallpaper. Now we estimate that the whole wall costs 77 €/ $m^2$ and the amount of work is 4 hours / $m^2$. Using varying granularities leads to varying flexibility in design. The different combinations of the standard components create the flexibility needed in constructing non-similar houses.

The estimation of constructing houses and buildings has no general house level unit (analogous to function points or lines of code). The progress of the project is estimated by component level units. The milestones are based on the phases of the construction; for example, groundwork ready and roof ready. We do not say that we have now constructed 58 $m^2$ of our 300-$m^2$ house or 3 floors of a 10 floor building because the effort involved in one unit varies. However, saying that we have assembled 42 500 bricks into our 100 000 brick wall is meaningful. Adding bricks and windows does not make sense. The same applies to software components. We can count (similar) windows and (similar) database tables but a combined sum is not reasonable. Doing a sum requires measurement or estimation of the effort. In construction work the effort can be assigned to a component. This does not mean that components are isolated. There are walls, floors and windows, which are tightly coupled to each other. The effort needed in assembling a component, say a door, is coupling it to the rest of the system, in the case of a door to a wall in the house. The effort itself depends on the type of process in which the work is done. For example, the work of assembling a wall-element is not the same if a crane is used. Productivity is a combination of the effects of tools, the team and the process.

### 1.6.3 Reuse taxonomy

There are two views to reuse [Tracz 1995, pages 93-94]:

- Before you can reuse something, you need to find it, know what it does, and know how to reuse it.

- Firstly, before you can reuse something, there has to be something to reuse and secondly before you can reuse something, it needs to be useful.

Poulin has made a taxonomy, which classifies several approaches to reuse metrics [Poulin 1997, page 110]. There are two main groups: empirical methods and qualitative methods. *Empirical methods* depend on objective data. An analyst can calculate them easily and cheaply, which is a very desirable property of a metric. Empirical studies compare attributes of reused components to attributes of components which are not reused. The attributes of reusable software must influence reusability. For example, there are more reused small components than large ones. Making all components small is not good practice because the benefits obtained by using large reusable components are greater. *Qualitative methods* define the attributes of reusable software and assessors subjectively assess how the software to be studied adheres to these attributes. The use of assessments makes qualitative methods more expensive than empirical methods [Poulin 1997].

Reusability is related to *portability and co-operativity* as they both assess the possibility of using one component with another component. *Maintainability* is related to reuse because both assess the ease of adaptation of a component. *Understandability* is an attribute which assesses the ease of comprehension of a component.

### 1.6.4 Reusability metrics

Prieto-Diaz and Freeman's model encourages white-box reuse and evaluates which components the programmer can modify most easily [Prieto-Diaz 1987]. Their metrics apply traditional program size and complexity metrics to particular programming languages. Documentation is evaluated subjectively. Reuse experience is the most important human factor in their metrics.

Selby looked at instances where reuse succeeded in the NASA environment and tried to determine why [Selby 1989]. He found that a reusable software module is small and its interface is simple. It has few dependencies on other modules. Good documentation is also one of its properties. The reuse of the low-level system and utility functions is more common than the reuse of human interface functions. Selby validated these results statistically.

The ESPRIT-2 project REBOOT (Reuse Based on Object-Oriented Techniques) developed the taxonomy of reusability attributes [Poulin 1997, page 116]. It has four reusability factors which are specified using a given number of criteria. Each criterion has at least one metrics. Reusability is a number from 0 to 1, which is calculated by normalising the metrics. REBOOT allows an analyst to change the weights of the metrics while calculating reusability. This is done because the importance of metrics may change from site to site. The four reusability factors of REBOOT are portability, flexibility, understandability, and the confidence of the reuser.

Portability expresses the ease of reuse in another environment. The criteria of flexibility are generality and modularity. Criteria of understandability include code complexity, self-descriptiveness, documentation quality and component complexity. *Confidence* is the probability of the error-free reuse as assessed by the reuser.

The NATO Standard for Software Reuse Procedures is more concerned with the statistics in the reuse process [Poulin 1997, page 123]. This metrics is helpful in eliminating unsuitable candidates for reuse. Modules, which have been considered many times without actual reuse, can be treated as not being useful. The same applies to modules of high complexity that have a large number of defects.

The Army Reuse Center inspects all software submitted to the Defense Software Repository System [Poulin 1997, page 123-124]. The preliminary inspection estimates the effort required to reuse the component without modification, the effort required to develop a new component instead of reuse, the yearly maintenance effort and the expected number of reuses. The most important factor here is that the estimated effort required to reuse the component, which is needed in Component Reuse Metrics, is recorded as part of the reuse library administration.

IBM's method stresses that the developer needs to have access to the development project's documentation in addition to the code [Poulin 1997, page 126].

*Understandability, ease to find, adaptability and trustworthiness* are common factors in previous metrics, though the views are different. Thus, it is considered adequate to use them as the criteria for reusability in this study.

## 1.7 Outline of the thesis

Chapter 1 explained the motivation for the research, the research problem and the research methods. It introduced the measurement of software development and the related work. The common vocabulary of this thesis was also defined. Chapter 2 explains Component Reuse Metrics, CRM [Virtanen 2000b], which is the major contribution of this thesis. Chapter 3 gives a calculation example of CRM. Chapter 4 contains the practical evaluation of Component Reuse Metrics [Virtanen 2001]. It presents the results of the questionnaire concerning its basic assumptions and case studies of practical CRM tests. Chapter 5 compares CRM to related work in the estimation of software development. Chapter 6 focuses on the productivity improvements of component-based development. The paper "Adaptability -the enabler of reuse" [Virtanen 1999b] is rewritten, and the use of CRM has been added. Chapter 7 introduces "Verb classes", which is an architecture for creating highly adaptable components [Virtanen 1998b], [Virtanen 1999], [Virtanen 2000c]. It is also a contribution of this thesis. Chapter 8 summarises the thesis and gives the conclusions.

# 2 Component Reuse Metrics

## 2.1 Outline of CRM

The presentation describes the research method used in this study. The hypothesis is explained first and its rationalisation and validation follows. The representation of CRM is spread across several chapters. Firstly, in chapter 2, the general concepts of CRM are introduced. Secondly, the CRM-equations follow. Thirdly, the details of the parameters of the equations and the estimation process are described. Chapter 3 contains an example of the CRM calculations in order to clarify the preceding theory. The empirical study in chapter 4 gives a validation of the CRM method. Finally, in chapter 5, the related work is presented to create a context for CRM.

## 2.2 Introduction

A new method for estimating software development efforts, Component Reuse Metrics, CRM, developed in this study, combines the estimation of component-based technology and the analysis of human behaviour into simple calculation rules. CRM uses the following definitions:

> An *effort* is the amount of time that one person uses in accomplishing a task. The unit of the effort is an hour.
>
> An *effect* is a characteristic of software development which affects the effort significantly.
>
> A *factor* is a characteristic of an effect. For example, a skill needed to use tools and teamwork skills are factors of the skill effect.

According to the hypothesis of this study, the groups of the factors (called effects), which have influence on the effort of component-based software development, are *components, process, project change, team, risks, skills and motivation*. This kind of hypothesis can not be validated decisively, but empirical research can be used to ensure that it is reasonable and useful. The hyphothesis can also be rationalised by the goal-question-metric paradigm, which is a framework for guiding measurement efforts [Basili 1984]. According to this paradigm, the steps needed to create valid metrics are:

- define the principal goals of interest,
- construct a comprehensive set of questions to achieve these goals, and
- define the data required to answer to these questions.

As the goal is to estimate software development effort, an appropriate question for Goal Question Metrics is "what affects the effort of software development?" The affecting characteristics can be found from practical experience. Hakkarainen et al. have studied the factors used in the estimation of the effort of software projects in Finland [Hakkarainen 1993]. Their factors were hardware, user interface, developers' experience, size of application, generality of application, complexity of processing, development environment, and project organisation. The existing estimation methods use their own factors, which resemble the factors of CRM (see chapter 4). For example, COCOMO II has 5 scale factors and 17 effort multipliers, which characterise the product, the process, the team, the risk and the personnel [Boehm 2000]. COCOMO II has separate submodels for different kinds of products. In CRM the component effect contains all of the product related factors, which adapts the method to different kinds of products. As COCOMO II estimates the new lines of code in the forthcoming product project change effect is not explicit. COCOMO II does not include motivation factors though motivation is important in managing technical people [Humphrey 1997].

Figure 1 presents the classification of effects into component, project and human effects. The component effect includes all of the factors related to the product to be developed. In addition to the complexity of the product, the factors include quality attributes such as reliability, reusability and documentation. Selecting a component implicitly defines the platform and a number of properties related to it. The human effects of CRM are the skill effect and the motivation effect. The skill effect takes into consideration the required training. The motivation effect calculates the influence of the motivation of the staff on the development effort. The project effects are risk effect, team effect, project change effect and process effect.

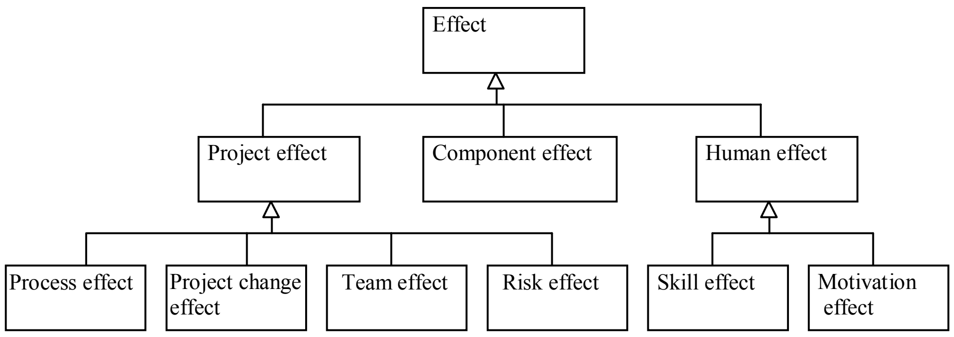


Figure 1. Effects used in CRM.

The effort needed in sharing information is included in the team effect. The project change effect calculates the influence of additional features added to the project during the development. This can be revealed by looking at the difference between the original and the final component structure. The process effect takes the changes in methods and tools into account. The effort of the risk effect is the probable additional effort due to the risks within the project.

The survey made in this study confirms that all of the effects of CRM are significant and that these effects can be estimated (chapter 3). The number of effects is small but each of them consists of a large number of flexible factors. New factors can be added if they are needed. This ensures that all the significant characteristics in the software development are taken into account. The use of a small number of effects is adequate as additional effects would increase the estimation effort.

The variation of complexity and productivity within a project is an essential property of software development (see chapters 1.5.5 and 1.5.6). CRM is based on a component structure for the complexity variation and a project breakdown structure for the productivity variation. CRM estimates the efforts of personal tasks.

> A *personal task* is the share one person has of a task in a project. The outcome of each personal task is progress in developing the tracking set of components connected to the personal task.
>
> A *tracking set* of the product is a component, aggregate component or a set of components, which is used to observe the development of the product during the project.

Figure 2 shows a small project, which contains two tasks: T1 and T2 and three personal tasks: PT 1, PT 2 and PT 3. The personal task PT 1 produces the tracking set TS 1 from a component T, a component C and two occurrences of component B. PT 2 and PT 3 produce TS 2 similarly. As the efforts of the components T, C and B are too small to be separated, the persons assigned to the personal tasks PT 1, PT2 and PT 3 only report the actual efforts of the personal tasks. The effort estimates of the personal tasks are based on the realised historical efforts of the components T, C and B.

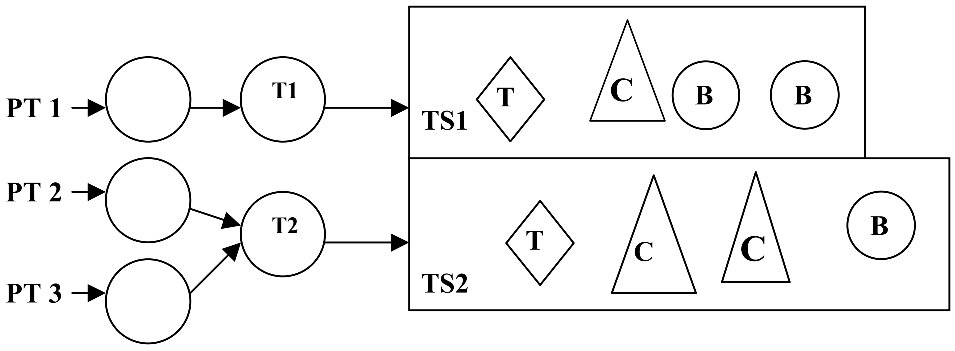


Figure 2. Personal tasks and tracking sets.

In CRM, historical data is collected into a project repository, which is described in chapter 2.3. The repository contains, for example, component structures and tasks from reusing components. The average effort of reusing a component can be used in estimating forthcoming projects because a component has by definition a fixed contract in its reuse. The estimation process of CRM resembles the estimation process of PROBE [Humphrey 1995], because it uses components as proxies of effort.

The first steps in the CRM estimation process are to plan the project and define the component structure of the product to be developed. As each personal task in the work breakdown structure of the project plan is connected to exactly one tracking set in the component structure, the baseline efforts of the personal tasks can be calculated by using the averages of the efforts of the components in various tracking sets in history. A *baseline effort* is the effort of the component effect only.

In the next stage of the estimation process the project manager uses estimation forms to estimate the importance of the project and human effects (Appendix A: Estimation forms and survey results). This can be done when a preliminary project plan is available. A *factor estimation form* of an effect contains one question about each factor of the effect. The answers to the questions scale the influence of each factor to a small number. The weighted averages of the answers to the estimation forms are used to estimate the impact of the project and the human effects in the forthcoming project.

Finally, the estimates of the efforts of the personal tasks are calculated by using the CRM equations, which will be presented later.

## 2.3 Project repository

*A project repository* is a database which stores selected data about previous, current and estimated forthcoming projects. In addition to effort information, the collected data contains assessments of the influence of project and human effects to the effort. C*omponent structures, project plans, CRM estimates and tracking data* are the main parts of the repository. The tracking data contains the realised efforts of the personal tasks classified according to the effort types. Figure 3 depicts the class diagram of the project repository. The large rectangles with pointed lines depict the grouping of the classes. A bullet-headed line denotes a one-to-many association between the classes. The bullet-head is at the many-side. The open-triangle arrowhead denotes generalisation, where the triangle is on the general side. The diamond arrowhead denotes aggregation, where the diamond is on the whole side. The attributes of the classes are described and used in the CRM equations in the following chapter. The symbol '#' designates a variable which contains a realised value. The meanings of the variables will be explained later.

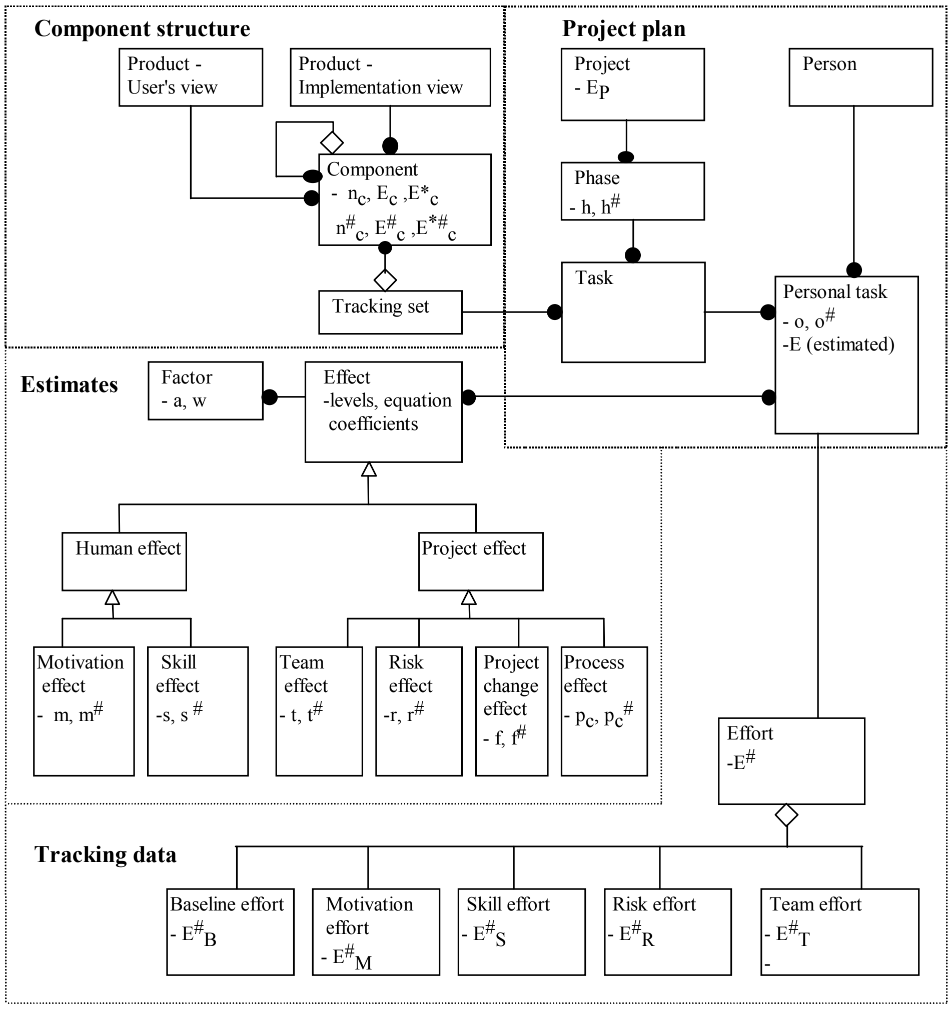

Component structure
Product - User's view
Product - Implementation view
Component - $n_c$, $E_c$, $E^*_c$ $n^\#_c$, $E^\#_c$, $E^{*\#}_c$
Tracking set
Project plan
Project - Ep
Person
Phase - h, h#
Task
Personal task - o, o# -E (estimated)
Estimates
Factor - a, w
Effect -levels, equation coefficients
Human effect
Project effect
Motivation effect - m, m#
Skill effect -s, s #
Team effect - t, t#
Risk effect -r, r#
Project change effect - f, f#
Process effect - $p_c$, $p_c^\#$
Effort -E#
Tracking data
Baseline effort - $E^\#_B$
Motivation effort - $E^\#_M$
Skill effort - $E^\#_S$
Risk effort - $E^\#_R$
Team effort - $E^\#_T$ -

Figure 3. Class diagram of CRM project repository.

There are two *component structures*, the external user's view and the internal implementation view, which the tracking set connects tightly together. The outside view of the product (customer view) is needed to make estimates based on information which is available early in the project, and to compare design solutions. If more information is available, the implementation level component model (implementation view) will enable more accurate estimates. The effort of a component ($E_c$) is the effort which is required to add and integrate the component into new software (see details in chapter 2.7, page 39). The estimator can use the construction effort of a component ($E^*_c$) to estimate the implementation effort of similar components. The attribute $n_c$ stands for the number of occurrences of the same/similar sub-component within the component structure. When a component has been constructed, CRM calculations use the effort of the constructed component.

From the *project plan* data about phases, tasks, persons and personal tasks is stored into the repository. The sets of tasks depend on the method and process model used in the development. A task can be accomplished by a group of persons. The share of an individual person in a task is called a personal task and the contribution coefficient (o) gives the person's proportion of the effort of the task. The baseline efforts of the personal tasks are calculated by using the tracking sets that are connected to the tasks. The CRM estimate of the effort of the project ($E_p$) is the sum of the efforts of the personal tasks. The phases are groups of tasks. Phase coefficients (h) show the division of the total effort into the phases.

CRM *estimates* the project and human effects by utilising *estimation forms*. Each *question* in an estimation form defines a factor of an effect. The *answers* (a) to the questions are weighted (w). Answers and weights are stored in the repository.

An *answer* is a small number describing the status of a factor in the forthcoming project.

A *weight* is a small number describing the importance of a factor to the effort. It is obtained from the repository in order to calculate the weighted average of the answers.

An *effect level* is the weighted average of the answers to the questions in the estimation form of the effect. The levels in CRM are process level ($l_p$), project change level ($l_f$), risk level ($l_r$), team level ($l_t$), skill level ($l_s$) and motivation level ($l_m$).

An *effect coefficient* (m, s, t, r, f, $p_c$) is a number that is used to calculate the effort of a personal task. It is also called a *correction coefficient* because it corrects the baseline effort according to the influence of the effects.

The *equation coefficients* ($\alpha$'s and $\beta$'s) are used to calculate the effect coefficients when the levels are known. See Equation 4.

The levels are assessed in the CRM estimation process by using the factor estimation forms. The levels are estimated for projects or groups of tasks and persons only because it is not practical to assess project and human effects at a more detailed level. If nothing or only a little has changed in the projects, the results of previous factor estimations can be used. The effect coefficients (m, s, t, r, f, $p_c$) can now be calculated because the project repository contains the equation coefficients based on the previous projects.

In *tracking*, the realised efforts are classified according to the type of effort and stored in the repository. A realised effort ($E^{\#}$) is a placeholder of a small, individually tracked part of work time. There is a correlation between the types of efforts and the types of effects. The baseline effort only contains the efforts of the components. The skill effort contains the time needed to study within the work. The team effort records the time needed for meetings and other collaborations. The risks of the project cause the risk efforts. The motivation effort is normally estimated after the project in order to correct the tracking data. The project change effort does not appear in Figure 3 because the project change effect illustrates the change in the component structure during the project. Similarly, the process effect describes the influence of the changes in the processes on the efforts of the components in the project history.

Figure 4 presents an overview of the CRM calculation. The estimation forms give the levels which are needed to calculate the coefficients. The estimates are calculated from the efforts of previous components using the coefficients. Personal time reports give the realised efforts for the recalculation which updates the repository.

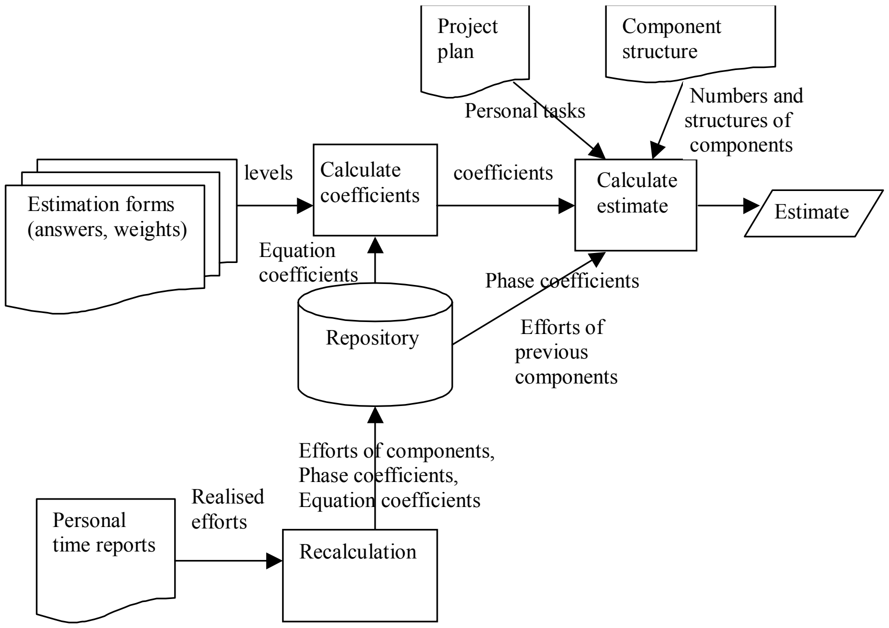


Figure 4. Overview of CRM estimation of personal tasks.

## 2.4 Estimation equations

### 2.4.1 Hypothesis

Software cost models can be based on the analysis of the factors and statistical evidence. The conclusion of the analysis is a hypothesis which can be tested empirically. The option of deriving statistical equations was not selected in this study, as not enough data is available. Finding the data can take a long time, which may mean that by the time it has been collected the changes in software development can render it obsolete. The criteria for the equations and their parameters are that

- they give sufficiently valid, accurate and reliable estimates,
- they are sufficiently understandable and intuitive as to be used with task based estimation, in bargaining the software contracts and in the management of the project and its personnel,
- the estimation effort is reasonable, and
- equations and coefficients can be updated flexibly according to experiences learned in different projects.

In this study a survey and a small number of case studies test the hypothesis. A more thorough analysis of the equations and the calibration of the parameters are left for future research. Any future research should have a large number of pilot projects and comparisons with other estimation methods.

### 2.4.2 Estimation forms

First of all, the estimator fills in the estimation forms (appendix A, page 162) in order to determine the levels of the effects. Estimation forms contain sets of questions which analyse the affecting factors of the project and human effects. The estimator's answers to the questions are adjusted to a small number (1 to 5) and weighted by a number which corresponds to the importance of the factor. Thus, the levels of the effects are weighted averages of the answers to the questions in the estimation forms. For example

Equation 3. $$l_s = \sum_{i \in skill\_effect} (a_i \cdot w_i) / ( \sum_{i \in skill\_effect} w_i ), \text{ where}$$

- $l_s$ = skill level in the forthcoming project,
- $a_i$ = answer to the question about a factor *i* (scaled from 1 to 5),
- $w_i$= weight of the factor (question) *i*, and
- summations contain the factors of the skill effect.

The estimator adapts the answers to the situation of the forthcoming project. For example, the experience of a developer in relation to a given task is assessed. The range of the values of the levels is based on the values of the answers. The initial values of the weights were calculated from the results of the survey by scaling the assessment of importance from 1 (very small) to 5 (very large) and counting the average (Appendix A: Estimation forms and survey results, page 162). The use of the questions and weights, presented in the appendix, creates an unambiguous way to get the levels, whilst retaining a degree of flexibility. The estimator can estimate the weights of the factors and change them to adapt the CRM method to the environment of a particular company. As the weights are presumed to be rather static and because it is difficult to get statistical data to confirm the correctness of the weights, it is impractical to change the weights for every project. The estimators can also add questions to the estimation forms or remove questions, if needed. For practical reasons the number of questions should be reasonably low. To decrease the effort of the estimation process, experienced estimators can also assess the levels or effect coefficients directly and skip the estimation forms. Old forms may also be used if the factors remain almost the same.

### 2.4.3 Effect coefficients

The second step is to calculate the effect coefficients using Equation 4. The levels of the effects of the forthcoming project have been estimated. The equation coefficients are taken from the project repository. The name equation coefficient refers to the Equation 4.

Equation 4. $$\begin{cases} m = \alpha_m \cdot l_m + \beta_m \\ s = \alpha_s \cdot l_s + \beta_s \\ t = \alpha_t \cdot l_t + \beta_t \\ r = \alpha_r \cdot l_r + \beta_r \\ f = \alpha_f \cdot l_f + \beta_f \\ p_c = \alpha_p \cdot l_p + \beta_p \end{cases}$$ , where

- $l_m$ = motivation level in the forthcoming project,
- $l_s$ = skill level in the forthcoming project,
- $l_t$ = team work level in the forthcoming project,
- $l_r$ = risk level in the forthcoming project,
- $l_f$= project change level in the forthcoming project,
- $l_p$ = process level in the forthcoming project,
- *m, s, t, ,r, f and* $p_C$ are effect coefficients and

- $\alpha_m$, $\beta_m$, $\alpha_s$, $\beta_s$, $\alpha_t$, $\beta_t$, $\alpha_r$, $\beta_r$, $\alpha_f$, $\beta_f$, $\alpha_p$ *and* $\beta_p$ are equation coefficients of each effect from the project repository.

Simple linear equations are the most obvious choice without extensive statistical evidence. Further, they are not as sensitive to error-prone assessments of the levels as higher-degree formulas. A more cautious method is to use the repository to look for the projects where the levels are almost similar to the levels in the project to be estimated. The process coefficients, $p_C$, can be determined for each component *c* separately, but normally only one coefficient for a project is used.

### 2.4.4 Baseline efforts

Thirdly, the estimator calculates the baseline efforts with the process correction. The baseline effort of a personal task contains the component-related part of the effort. It is the sum of efforts of the components which are handled during the personal task. In equation form:

Equation 5. $$\begin{cases} E_B(PT) = h \cdot o \cdot \sum\limits_{c \in C(PT)} n_c \cdot E_c \\ E_{BW}(PT) = h \cdot o \cdot \sum\limits_{c \in C(PT)} n_c \cdot p_c \cdot E_c \end{cases},$$ where

- *PT* = a personal task,
- $E_B(PT)$= baseline effort of the personal task *PT*,
- $E_{BW}(PT)$= process ("*W*ork-flow") corrected baseline effort of the personal task *PT*,
- *C*(*PT*) = set of components (tracking set) which are developed at least partly during the personal task *PT*,
- *h* = phase coefficient (the proportion of effort of the task of the total effort of the components of the tracking set), and
- *o* = contribution coefficient (the proportion of the person of the total effort of the task),
- *c* = a component which belongs to *C*(*PT*),
- $n_c$ = number of components *c* in *C*(*PT*),
- $p_c$ = process effect coefficient of component *c*, and
- $E_c$ = effort of the component *c* in the project repository (hours).

Due to the fact that tasks are small the baseline effort of a personal task can be calculated as the sum of the individual efforts of the components in its tracking set. As CRM uses the historical information from a development organisation, it is assumed that the average properties of the components are adequate in the estimates.

### 2.4.5 Effort estimates

Finally, the efforts of the personal tasks are obtained when the phase coefficient, contribution coefficient and effect coefficients are substituted into CRM equations. CRM primarily estimates the efforts of the personal tasks because they are needed to schedule forthcoming projects and because the effort depends strongly on persons and on components. In addition, different project members have different costs (price, salary) which must be taken into account in cost estimates. The timing of the tasks determines the schedule of the project. From the estimation point of view, estimating personal tasks gives a lot of timely feedback, which can be used in the recalculation to adapt the coefficients of the CRM. A personal task participates in the construction of exactly one tracking set. A tracking set can be any component, aggregate component or set of components that is a suitable part of the product to be developed in one task.

Several formulas of the effort of the personal task have been tried during the evolution of CRM (see chapter 4.15, page 85). The general form of the equation is:

Equation 6. $E(PT) = F(m, s, t, r, f, E_{BW}(PT))$, where

- $E(PT)$= estimated effort of a personal task $PT$,
- $m$ = personal motivation effect coefficient of the person in the personal task $PT$,
- $s$ = personal skill effect coefficient of the person in the personal task $PT$,
- $t$ = team effect coefficient of the personal task $PT$,
- $r$ = risk effect coefficient of the personal task $PT$, and
- $f$ = project change effect coefficient of the personal task $PT$,
- $E_{BW}(PT)$= process ("*W*ork-flow") corrected baseline effort of the personal task $PT$ (see Equation 5).

Statistical evidence suggests that the relationship between the effort and the size of the software is (almost) linear [Maxwell 2002]. In COCOMO II scale factors (exponent of size) account for the relative economics of scale in software projects of different sizes [Boehm 2000]. CRM equations do not have such factors because CRM equations estimate tasks, not large projects. Thus, our hypothesis is that that the effort of a personal task can be obtained from the baseline effort using coefficients.

One of the ideas within this study was to utilise time reporting to get classified effort data. When the project and human efforts are reported as separate numbers, it is natural to try to estimate them in the same way. The project managers can also estimate the relative influence of the effects. Consequently, we relate the other effects to the baseline effort. Our first hypothesis was that the estimated effort of a personal task is the product of the **sum** of effect coefficients, and the sum of the process corrected efforts of the components. The coefficients represent the additional effort to the baseline effort. Their values are normally slightly above zero.

Equation 7. $E(PT) = (1 + m' + s' + t' + r' + f') \cdot E_{BW}(PT)$, where

- $E(PT)$ = estimated effort of a personal task $PT$,
- $m'$ = personal motivation effect coefficient of the person in the personal task $PT$,
- $s'$ = personal skill effect coefficient of the person in the personal task $PT$,
- $t'$ = team effect coefficient of the personal task $PT$,
- $r'$ = risk effect coefficient of the personal task $PT$,
- $f'$ = project change effect coefficient of the personal task $PT$, and
- $E_{BW}(PT)$= process ("*W*ork-flow") corrected baseline effort of the personal task $PT$ (see Equation 5).

In the case studies (see chapter 4.15, page 85) this equation gave inadequate results when the project changed and risks were realised. The team, skill and motivation efforts also changed although their corresponding levels remained the same. Changing the equation could solve this problem. Our second hypothesis is that the estimated effort of a personal task is the **product** of effect coefficients and the sum of the process corrected efforts of the components. In equation form

Equation 8. $E(PT) = m \cdot s \cdot t \cdot r \cdot f \cdot E_{BW}(PT)$, where

- $E(PT)$ = estimated effort of a personal task $PT$,
- $m$ = personal motivation effect coefficient of the person in the task $PT$,
- $s$ = personal skill effect coefficient of the person in the task $PT$,
- $t$ = team effect coefficient of the task $PT$,
- $r$ = risk effect coefficient of the task $PT$, and
- $f$ = project change effect coefficient of the task $PT$ , and
- $E_{BW}(PT)$= process ("*W*ork-flow") corrected baseline effort of the personal task $PT$ (see Equation 5).

The values of the effect coefficients (also called correction coefficients) m, s, t, r, f and $p_c$ are normally close to 1.00 (contrary to Equation 7, where they are slightly above zero). The values are 1.00 if the baseline effort can be used. These coefficients are positive and their upper bound is not fixed because the efforts corresponding to the effects are not restricted. For example, a realised risk can double the effort. To decrease the workload needed in the estimation, the same project effect coefficients are used in all the tasks in the project and the same human effect coefficients are used in all the tasks of the person. The phase coefficient (*h*), which describes the relative effort of the phase in the project, can be obtained from the previous projects in the repository. The project planning determines the contribution coefficient (o) and the tracking set. The value range of the coefficients *h* and *o* is from 0 to 1. As a tracking set can contain a large number of similar construction blocks, the number of components ($n_C$) can be any integer, though normally it would be quite small.

The estimated effort of a project can be calculated by summing up the efforts of the personal tasks:

Equation 9. $E(P) = \sum_{PT \in P} E(PT)$ , where

- $P$ = the forthcoming project to be estimated, and
- $E(P)$ = estimated effort of the new project $P$, where the efforts of all personal tasks of the project have been summed, and
- $E(PT)$ = estimated effort of a personal task $PT$.

## 2.5 Recalculation equations

### 2.5.1 Reported data

As a large application contains thousands of components which have different complexities and a large project has hundreds of tasks, project tracking gives CRM a lot of timely feedback. This information is used to adjust the estimates of the successive phases of the project and the estimates of other projects. The purpose of the recalculation is to correct the estimates during a project and to provide data for future estimates after the project. During a project actual efforts are classified (baseline, motivation, skill, teamwork and risk) and stored in the repository. The efforts for the process and project change effects are not reported, but they are calculatory. The details of the effort types are described later in this subsection. New tasks for additional tracking sets will be created when needed. The developer, who reports his/her realised effort, distinguishes the types by the quality of the work accomplished. Normally the effort is related to one effect only and overlapping efforts can be separated using the ordering rules (see chapter 2.5.3). The project manager estimates the factors of the motivation effect because the interpretation offered by the individual may be biased. If the efforts are not classified during the project, they must be estimated after the project.

After the project, the data of the realised efforts is collected and stored in the repository. Recalculation is then used to calculate the efforts of the components ($E^{\#}_{C}$), the equation coefficients and phase coefficients ($h^{\#}$). The symbol '#" here designates a variable which contains a realised value. The realised effort of a personal task is the sum of all the types of efforts related to the personal task.

Equation 10.

$$E^{\#}(PT) = E^{\#}_{B}(PT) + E^{\#}_{M}(PT) + E^{\#}_{S}(PT) + E^{\#}_{T}(PT) + E^{\#}_{R}(PT), \text{ where}$$

- $E^{\#}(PT)$ = realised total effort of a personal task *PT* of the project (hours),
- $E^{\#}_{B}(PT)$ = realised baseline effort of a personal task of the project (hours),
- $E^{\#}_{M}(PT)$ = realised motivation effort of a personal task of the project (hours),
- $E^{\#}_{S}(PT)$ = realised skill effort of a personal task of the project (hours),
- $E^{\#}_{T}(PT)$ = realised team effort of a personal task of the project (hours), and
- $E^{\#}_{R}(PT)$ = realised risk effort of a personal task of the project (hours).

The realised baseline effort represents the effort of the realised product which has been accomplished by the current process. Thus, it contains the effort created by process and project effects.

### 2.5.2 Phase coefficient

In the first stage of the recalculation process, the phase coefficients, which give the relative efforts of the phases, are calculated. As each personal task belongs to exactly one phase of the project, the phase coefficient is the quotient of the sum of the realised baseline efforts of a phase of a project and of the sum of the realised baseline efforts of the whole project. In equation form

Equation 11. $$h^{\#}(Phase) = \frac{\sum_{PT \in Phase} E^{\#}_{B}(PT)}{\sum_{PT \in Project} E^{\#}_{B}(PT)}, \text{ where}$$

- $h^{\#}(Phase)$= realised phase coefficient of the tasks of the phase *Phase*,

- $E^{\#}_{B}$ $(PT)$ = realised baseline effort of a personal task of the project (hours),
- the numerator contains the realised baseline efforts of a phase of the project, and
- the denominator contains the realised baseline efforts of the whole project *Project*.

### 2.5.3 Efforts of the components

Secondly, the efforts of the components ($E^{\#}_{C}$) are solved from a group of linear equations in which the baseline efforts of tasks are made equal to the efforts of the components of the corresponding tracking sets. A numerical solution is required, because the number of equations and unknowns does not match and because the unknown efforts are always positive. In a good solution the difference between the effort calculated using the solved efforts of the components and the actual baseline effort is small. Multiple linear regression is a mathematical method of getting the solution. The equations are a consequence of Equation 5, when we notice that the realised baseline efforts contain the effort that belongs to the phase and the person of the personal task. The process correction is not needed if the efforts of the components are calculated for the only process used during the project. The use of task level equations eliminates the contribution coefficient. The equations are of the form

Equation 12. $$\sum_{PT \in T} E^{\#}_{B}(PT) \cong h^{\#}(Phase(T)) \cdot \sum_{c \in C^{\#}(T)} n^{\#}{}_{c} \cdot E^{\#}_{c}$$ , where

- $E^{\#}_{c}$ = realised effort of the component $c$ (unknown),
- $T$= a task,
- $E^{\#}_{B}(PT)$ = realised baseline effort of a personal task $PT$ of the project (hours),
- $C^{\#}$ $(T)$= realised set of components (tracking set) which are developed at least partly during the task $T$,
- $h^{\#}$ $(Phase(T))$= phase coefficient for the task $T$ and
- $n^{\#}_{c}$ = realised number of the components $c$ in the personal task.

The phase coefficients are calculated by Equation 11. The numerical non-negative linear least square method can be used in the calculation because the efforts of the components are always larger than zero [Lawson 1974].

Equation 13 and Equation 17 (page 59) show examples of the previous formulas. The equation group could be

Equation 13. $$\begin{cases} 3 \cdot E_1 + 2 \cdot E_2 = 7 \\ 5 \cdot E_1 + 3 \cdot E_2 = 12 \end{cases}$$ , where

- the right sides of the equations contain the realised baseline efforts of the tasks (in hours)
- for clarity, the phase coefficients are 1 in both of the equations,
- $E_1$ and $E_2$ on the left hand sides of the equations are the realised efforts of the corresponding components and
- 3, 5, 2 and 3 on the left hand sides of the equations are the realised numbers of the components $E_1$ and $E_2$ in the personal tasks.

The solution to the equation group is $E_1$= 3 hours and $E_2$= -1 hours, which indicates negative effort for the component $E_2$. A more appropriate approximate solution $E_1$= 1.81 hours and $E_2$= 0.90 hours can be obtained by the numerical method (above), which assures a solution where $E_1$ and $E_2$ are positive.

### 2.5.4 Effect coefficients

Thirdly, the correction coefficients are extracted from the totals of the classified effort data (Equation 14). The totals are used to smooth the random fluctuations of the software development. As the process coefficient measures the change of effort of the components in different projects, the projects (processes) involved must be named within the context of the coefficient. The project change coefficient is obtained by dividing the effort of the realised product by the effort of the planned product.

The calculated effect coefficients reveal what the coefficients should have been while estimating the project. If we substitute the baseline efforts and these coefficients to the product-form formula (Equation 8), the total effort is obtained (Equation 10). If the addition-form formula (Equation 7, page 30) were used, the last four extraction formulas would contain the sum of the baseline efforts in the denominator and quotients were not added to one.

Equation 14.

$$\begin{cases} p^{\#}(c, p1, p2) = \dfrac{E^{\#}{}_{c}(p1)}{E^{\#}{}_{c}(p2)} \\[2ex] f^{\#}(ES) = \dfrac{\sum\limits_{c \in actual\_components} n^{\#}{}_{c} \cdot E^{\#}{}_{c}}{\sum\limits_{c \in planned\_components} n_{c} \cdot \; E^{\#}{}_{c}} \\[2ex] r^{\#}(ES) = 1 + \dfrac{\sum\limits_{PT \in ES} E^{\#}_{R}(PT)}{\sum\limits_{PT \in ES} E^{\#}_{B}(PT)} \\[2ex] t^{\#}(ES) = 1 + \dfrac{\sum\limits_{PT \in ES} E^{\#}_{T}(PT)}{\sum\limits_{PT \in ES} E^{\#}_{B}(PT) + \sum\limits_{PT \in ES} E^{\#}_{R}(PT)} \\[2ex] s^{\#}(ES) = 1 + \dfrac{\sum\limits_{PT \in ES} E^{\#}_{S}(PT)}{\sum\limits_{PT \in ES} E^{\#}_{B}(PT) + \sum\limits_{PT \in ES} E^{\#}_{R}(PT) + \sum\limits_{PT \in ES} E^{\#}_{T}(PT)} \\[2ex] m^{\#}(ES) = 1 + \dfrac{\sum\limits_{PT \in ES} E^{\#}_{M}(PT)}{\sum\limits_{PT \in ES} E^{\#}_{B}(PT) + \sum\limits_{PT \in ES} E^{\#}_{R}(PT) + \sum\limits_{PT \in ES} E^{\#}_{T}(PT) + \sum\limits_{PT \in ES} E^{\#}_{S}(PT)} \end{cases}$$

where

- $c$= a component,
- $n_C$= the planned number of components $c$,
- $n^{\#}{}_{C}$= the realised number of components $c$,
- *actual_components*= the set of realised components,
- *planned_components*= the set of planned components,
- *p1*= the process of the current project,
- *p2*= the process of the historical project to be compared,
- $p^{\#}(c,p1,p2)$= process coefficient of component $c$ between processes p1 and p2. The process coefficient divides the efforts of the component $c$, which are calculated using the Equation 12,
- *ES*= set of personal tasks in a project or a subproject where the same human and project effect coefficients are used (="estimation set"), and

- $s^{\#}(ES)$, $m^{\#}(ES)$, $f^{\#}(ES)$, $t^{\#}(ES)$ *and* $r^{\#}(ES)$ = the realised effect coefficients of the corresponding effects.

The previous extraction of coefficients is based on the mathematical presentation of a sum as a product (Equation 15). Therefore, the extraction order itself is not significant, presuming it has been fixed in each repository. The objective of the selection is to create stable coefficients which correspond well to the levels by placing large efforts in the denominators first. The project effects are extracted before the human effects in order to avoid estimating and extracting them separately for each person. The risk effect is extracted before the team effect because realised risk efforts can be large. The skill effect is extracted before the motivation effect because it is less subjective. The selection of the extraction order is useful in adapting CRM to the corporate environment and reporting practices.

Equation 15.

$$a+b+c+d = a\cdot\frac{a+b}{a}\cdot\frac{a+b+c}{a+b}\cdot\frac{a+b+c+d}{a+b+c} = d\cdot\frac{d+b}{d}\cdot\frac{d+b+c}{d+b}\cdot\frac{d+b+c+a}{d+b+c}$$
$$= a\cdot\left(1+\frac{b}{a}\right)\cdot\left(1+\frac{c}{a+b}\right)\cdot\left(1+\frac{d}{a+b+c}\right) = d\cdot\left(1+\frac{b}{d}\right)\cdot\left(1+\frac{c}{d+b}\right)\cdot\left(1+\frac{a}{d+b+c}\right)$$

The mutual values of the coefficients are not computationally symmetric, but those coefficients which were extracted first are relatively larger. If the project managers operate on the effect levels as recommended, not on the effect coefficients, the extraction order should not influence their assessments. The unsymmetry problem can be avoided by a symmetric solution. In Equation 16, auxiliary variable x is solved numerically from the upper equation and substituted to the other equations.

Equation 16.

$$\begin{cases} \dfrac{\sum\limits_{PT\in ES} E^{\#}(PT)}{\sum\limits_{PT\in ES} E_B^{\#}(PT)} = \left(1 + x \cdot \sum\limits_{PT\in ES} E_R^{\#}(PT)\right) \cdot \left(1 + x \cdot \sum\limits_{PT\in ES} E_T^{\#}(PT)\right) \cdot \\ \qquad \left(1 + x \cdot \sum\limits_{PT\in ES} E_S^{\#}(PT)\right) \cdot \left(1 + x \cdot \sum\limits_{PT\in ES} E_M^{\#}(PT)\right) \\ r^{\#}(ES) = 1 + x \cdot \sum\limits_{PT\in ES} E_R^{\#}(PT) \\ t^{\#}(ES) = 1 + x \cdot \sum\limits_{PT\in ES} E_T^{\#}(PT) \\ s^{\#}(ES) = 1 + x \cdot \sum\limits_{PT\in ES} E_S^{\#}(PT) \\ m^{\#}(ES) = 1 + x \cdot \sum\limits_{PT\in ES} E_M^{\#}(PT) \end{cases}$$

where

- $x$= an auxiliary coefficient.

Statistical factor analysis could be used to investigate the mutual dependencies of the project and human efforts. This could also lead to improvements in CRM equations. This is, however, left for future work.

### 2.5.5 Equation coefficients

As the project and human effects may have been misjudged in the original estimate, they can be reassessed. The recalculation uses the estimation forms, which are answered again after the project so that the answers correspond to the assessments of the factors of the realised project.

Effect levels and corresponding effect coefficients are added to the repository database. If the project repository data includes several projects and their estimates, the confidence to the equation coefficients will be better and statistical procedures such as removing outliers can be applied. The judgement of the estimator determines whether the new data is useful at all. Exceptional and false data is simply discarded.

Finally, the simple least squares method is used to calculate the realised equation coefficients ($\alpha$'s and $\beta$'s) by using the correction coefficients and the levels of the corresponding effects. Simple least squares is chosen because it is widely available and because it is a simple method by which the levels and the coefficients can be correlated.

## 2.6 Effort estimation process

*Initially,* the CRM repository contains the efforts of the components, the phase coefficients and equation coefficients (see Equation 4, page 27), which have been calculated in the previous projects. In the first project an expert can estimate these parameters.

The forthcoming *application is designed* in such a way that its component structure is established. The analysis of the customer requirements is not enough because the efforts of different design solutions vary widely. The (estimated) numbers of internal components of each kind can be counted from the component structure and the effort for each of them ($E_c$ in Equation 5) is in the project repository database. If the process has been changed compared to the previous projects, each component-based effort must be multiplied by the process coefficient p (Equation 5).

The *project plan* is created in parallel with the application design. It has a large influence on the quality of the estimate because it defines the personal tasks of the project and the outcome of each of them. The tracking sets of the product are sets of components which are used to track the progress of the development of the product during the project. Each window in the user interface, which gives similar accuracy about the application as function point analysis does, is a natural tracking set. The numbers ($n_c$ in Equation 5) are counted for each of the components belonging to the tracking set. For example, a window may contain 5 entry fields, 3 buttons and 3 database queries. If very complex components are reused useful tracking sets are parts of the whole component. The baseline effort of each tracking set can be calculated by summing the process-corrected efforts of each sub-component (Equation 5).

As tracking sets are usually built during more than one task, the component-based efforts must be divided into tasks. Each task has a phase percentage which tells the part of the total effort of the tracking set which belongs to that task. The percentage shares of phases can be found from the project repository database. The baseline effort of a task can be calculated by multiplying the baseline effort of the tracking set of the task by the percentage share (h in Equation 5) of the phase of the task. The effort of the personal task can be calculated by multiplying the baseline effort of the task by the personal share of the person (o in Equation 5). This number is determined in the planning of the forthcoming project.

*The human and project effects are assessed* and added to the calculations when the tasks are assigned to the staff. For practical reasons the same correction coefficients of project change, risks and teamwork (f, r and t) are used in every task in the project. The skill and motivation of the staff is normally estimated once for each person and the same coefficients (s and m) are used for every personal task in the project. It is possible to estimate separate coefficients for each personal task, but this increases the effort needed in the estimation process.

As the projects are not similar, the estimation forms are generally used at the beginning of new projects. The answers to the estimation forms give a small number from 1.00 to 5.00, which corresponds to the change of the effort due to project change, process changes, teamwork, risks and the skill and motivation of the staff. Equation 3 is used to calculate the level of the effect, when the answers to the estimation forms are known. The coefficient of each effect can be calculated using these numbers and the Equation 4.

*Finally*, the corrected effort of the personal task can be calculated by multiplying the baseline effort by the correction coefficients (Equation 8). The effort of the whole project is the sum of these personal efforts (Equation 9).

At the beginning of CRM use the project repository is empty. The results of the survey of this study can be used as default values of the correction coefficients (see chapter 4.4, page 70) and weights of the factors (see Appendix A: Estimation forms and survey results, page 162). As soon as a personal task is finished, the recalculation can be accomplished to tune the efforts of the components and correction coefficients. This is especially useful in incremental software development.

As the division of effort in *iterative development* is based on an agreement about the contents of iterations, the phase coefficient is used in distributing the calculated total effort to the iterations of the project. It is also possible to use larger components in the first iterations and smaller components in the last iterations. The tracking sets associate the different component hierarchies in these cases. Project change coefficient can be used in estimating the growth of the number of components. In this case the original estimation form of project change effect must be adapted to consider the change factors of iterative development.

The calculation can be accomplished by *spreadsheets* which contain estimation forms for the estimation of the project and human effects, calculation tables for estimating the effort and for the recalculation. The effort of maintaining the repository can be considered reasonable, if the sizes of the tasks and the number of assessments are appropriate (see also chapter 4.12, page 80).

## 2.7 Counting the components

Though components are usually designed by object-oriented methods and they normally contain classes, CRM can be used with traditional components such as sub-systems, modules and module-libraries, too. Components can be counted objectively after the implementation, but in the early design the estimator must have a good view of the outcome of the project and of the components needed.

The effort related to a component of an application depends on the component. There are components which provide a huge amount of functionality and which have complicated interfaces and lots of dependencies. The component may be seen as a subsystem of the application. On the other hand, there are components which have a single simple interface and which provide a limited service.

CRM sees the contract between the component and its customer as the most important factor in estimating the effort of the component. If a component provides a group of services and interfaces, CRM can handle these cases (different contracts) as separate components. If a component has several types of usage, the classification of its reuses, for example, in the scale difficult, normal and easy will make the estimates more accurate. The effort related to a component can be thought of as the effort needed to create the application A′, containing the component c, from an otherwise similar application A, which does not have c. The effort of the component contains any effort which is needed to add and integrate the component to the application. The efforts for finding (from a component library), assembling, adapting and testing the components are included. The purchase and evaluation of the components are usually excluded and estimated separately because this is done only once, which distorts the reuse effort.

Often at the beginning of a project only the architecture and the user's view of the component structure is available. The user's view contains the components of the presentation tier only. Thus the effort of these components should also include the effort of the related components of the other tiers in the architecture. The implementation view of the component structure recognises the components of the separate tiers and other system interfaces of multi-tier software architectures. It should be used as soon as the component design is available.

The hierarchy and the order of the assembly of the components have an influence on the personal tasks. In Equation 5 (page 28) the personal task contains only the components which are actually handled in that particular task.

The definition of a component in applying CRM is above all practical. A component must be general enough to be reused several times, at best thousands of times even in the same application. On the other hand, a small variation of the reuse effort improves the estimates.

It is typical to assemble the same/similar component several times to the application. With same/similar components we mean components that are similar after adaptation and integration because they reuse the same library component. For example, user interface windows contain several entry field and button components. As the effort of creating the input parameters, making the call and using the output data, must be done each time, CRM multiplies the number of reuses by the average reuse effort to get the total effort involved in reusing the components (Equation 5).

The effort of a single reuse decreases when the number of reuses increases because the first finding, assessing, purchasing and studying the component takes time. CRM places this one-time effort into the skill effect. If the purchasing and study of a new component is more elaborate, the process effect calculates the process difference. The team effect and risk effects are also included as possible places for the additional effort. Gradually, a component becomes familiar and its average reuse effort becomes a good estimate for the CRM calculations.

Constructing a new component must be estimated as an assembly of lower level components, which can have reuse efforts available. If such data does not exist, the construction effort of similar components can be useful. Finally, a task based estimation can be used if no data exists in the repository.

The change of a component is estimated by dividing the effort into sub-efforts, which handle their own components. As individual changes are different, it is not useful to store their efforts in the repository. The reuse effort of the component is often a good approximation of the effort of a change.

The design of the application defines the component structure of the application. The CRM estimation of a project can be done when the structure is available. As a partial component structure is sufficient for making the estimates of the tasks, which are related to the designed part of the application, the CRM estimation can be done gradually.

Typical projects also have tasks which do not have a direct connection to a certain component, but are related to the whole product or to an aggregate component in the component structure. For example, analysis and deployment tasks are often related to the whole product. If, for example, the design task of a sub-component can be tracked separately, it is possible to use that as the component of the task. In these cases, the same CRM-formulas are applied to large tracking sets.

## 2.8 Process effect

In the easiest case processes, methods, tools and phases are the same in the forthcoming project as in the projects chosen from the project repository database. If changes are needed, the process effect in CRM is used to calculate the influence of these changes on the effort.

Projects are divided into phases, which are defined by their input and output. In the Rational Unified Process [Jacobson 1999] the phases are Inception, Elaboration, Construction and Transition. The core workflows are Requirements gathering, Analysis, Design, Implementation and Test. The project tracking is organised according to the phases and they contain one or more tasks. As the effort of a component is divided into several tasks, CRM uses the proportions of the phases (h in Equation 5) to partition the effort of the component into phases. A typical iterative project allocates 5% to the inception phase, 20% to the elaboration phase, 65% to the construction phase and 10% into the transition phase [Jacobson 1999, page 335]. This is one part of handling the process effect. CRM calculates these proportions into the repository database to help the estimator design the project. The other part is adjusting the repository data when tools and methods are changed.

The choice of a programming language is one of the most important design issues, particularly in relation to productivity. The whole programming environment usually supports the chosen programming language. The language itself with its support libraries will become part of the future product. There are studies about productivity using different programming languages. One of these depicts very large differences in effort due to the programming language [Dreger 1992]. The conclusion is that higher level languages are very expressive. One instruction in a higher level language has aggregated the functionality of many lower level language instructions [Dreger 1992]. The programming language is one of the factors in the process effect estimation form.

The tools and programming environments are handled in the same way as programming languages. They are factors needed to correlate the efforts of the components in different processes together. In a study by Bruckhaus [Bruckhaus 1996], the impact of tools to the effort varied from -26 % to +108 % compared to the original situation, depending on the size of the project and the changes in a process due to usage of tools.

As constructing components for reuse uses more rigorous processes than constructing them for one case only, the process effect coefficient is larger in the former case. The same applies to the process of purchasing and evaluating a component, which is not the same as the process of reusing a component routinely.

The process effect coefficient ($p_c$ in Equation 5) corrects the components' baseline efforts to a level which can be achieved by the tools and methods that are used in the forthcoming project. The process estimation form gives an assessment of the impact of the changes (Table 28, page 162). As the repository data of a component can be from different projects (and methods and tools), the process effect coefficient is connected to a component. In practice successive corrections can be multiplied and groups of components are handled together.

## 2.9 Project change effect

Software products are collections of various kinds of features. CRM handles the changes of the feature set as a property of the development, not as a risk, because their probability is high. Project change includes both adding features gradually and major changes. In a study by Jones [Jones 1994], 60 projects were considered and the probability of change was 70 %. The average amount of additional features was 35 % and the maximum amount of additional features was 200 %. The project change effect is also called the *feature creep effect* [Virtanen 1998a].

It is easy to separate the original features (components) and tasks from new ones if the history of the product model and the project model are saved. A more difficult case is creeping complexity. A problem seems to be easy at first sight but after learning more about the problem and its solutions, a more sophisticated solution is preferred. If a software configuration management (SCM) tool or a CASE-tool is used, the change history of the configurations is available at the implementation and/or design levels [Haikala 1998]. SCM can handle individual files, collections of components, transactions of changes or change requests [Kilpi 1998]. If the change requests are connected to the personal tasks, it is possible to have a more detailed view of the project change effect than the presented CRM equation (Equation 14, page 35) suggests. The original information about the project is usually kept intact in the project management tools.

Project change differs from the incremental development, which is applied in the Rational Unified Process method, because in incremental development taking into account new features in the project is carried out on purpose. A planned introduction of a new feature in projects does not result in difficult scheduling, organisational and quality problems as are found when ad-hoc changes in a project are made . In both cases the amount of work that is not known at the beginning should be forecast. As the goal of CRM is to estimate the effort, rejected features must also be counted if the effort has already been spent. The correction of errors in the original product belongs to the baseline effort, not to the project change, if the design is not changed. The change of a feature can be handled as a change of the components, which are related to the feature.

The amount of project change depends on the corporate culture. In some organisations projects are carried out nearly as originally planned and in other organisations a large amount of additional functionality is accepted. The most important factors that contribute to project change are the attitudes of the project team to the stability of the feature set and the quality of the analysis and design. The project change estimation form is used in estimating these factors (Table 29, page 163). It gives the project change level ($l_f$ in Equation 4), which determines the project change coefficient (f in Equation 8).

## *2.10 Risk effect*

A *risk* is an accident which has low probability and serious negative consequences. As the risks in software development projects are considerable, projects are prepared for this eventuality. Using principal component analysis Ropponen and Lyytinen identified six software risk components: 1) scheduling and timing risks, 2) functionality risks, 3) subcontracting risks, 4) requirement management risks, 5) resource usage and performance risks and 6) personnel management risks [Ropponen 2000]. According to McConnell, the most common risks are [McConnell 1996]:

- project change (CRM handles this as a separate effect, not as a risk),
- gold-plating (the users or the developers use too much time in polishing the application),
- decreased quality,

- overly optimistic schedules,
- inadequate design,
- silver-bullet syndrome (unfounded trust on new methods and tools),
- research-oriented development (the goal of the project is unknown or it is not sure that it can be achieved or lack of market research),
- weak personnel (the skills of the project members are not adequate) ,
- contractor failure (the outsourcing of the work fails),
- friction between the developers and the customers, and
- turnover of the key personnel.

The spiral life-cycle model is a risk-oriented life-cycle model. Risk analyses are included at the beginning of each cycle. It breaks the project up into mini projects. Each of these addresses one or more major risks until all the major risks have been addressed [Boehm 1988].

The risk effect coefficient (r in Equation 8) produces an expectation value of the additional effort due to risks. It includes the effort needed to avoid the risks and the preparation for the anticipated risks. The risk level ($l_r$ in Equation 4) is assessed by the risk estimation form (Table 31, page 165). The impact of some risks is very large; it may even indicate the failure of the entire project. In these cases the use of expectation values is not enough. If the project repository contains a large number of risk level - risk coefficient pairs, a prediction interval of the estimate can be obtained. When the risk coefficient of a forthcoming project is estimated, the upper limit of the interval is the largest risk coefficient at the risk level of the forthcoming project and the lower limit is at the lowest level of risk.

## *2.11 Team effect*

Adding manpower to a software development project increases the effort. As software development cannot be partitioned into separate independent tasks, a large amount of communication is needed [Brooks 1995]. This includes meetings, project management, documentation and continuous interpersonal communication.

Scheduled meetings are typically included in the project plan as their own tasks. There is also time allocation for project management. The consideration of travel time is also essential in distributed projects. The component to be tracked in CRM for these tasks is the whole product or a selected tracking set, if the meetings and other collaboration only concern the selected part of the product.

Teamwork requires more documentation than working alone. Time is therefore needed to read and write meeting minutes, project progress reports and so on. There are also small interruptions in work which reduce productivity [Constantine 1995]. These include short meetings and questions and answers between the team members. Also phone calls, faxes and e-mail are all examples of small-level communication. The team needs co-operation, common understanding and a common language. Waiting times must also be added in because waiting for somebody to accomplish a task, especially when the tasks and division of effort are not well planned is a very common occurrence. These communication tasks are so small that it is not reasonable to report them individually. All these kinds of communication work should be recorded to the component that is being done. The team effect coefficient is an estimate of this effort.

The team structure and size are the most important factors in estimating the effort needed for communication. The numbers of stakeholders (for example, developers, users, managers) are one part of the equation. It is more difficult to assess the communication culture of the project. It is influenced by both the customer and the supplier organisations. Some organisations require more written communication whilst others operate with more person-to-person communication. The team effect estimation form assesses these factors and is used in calculating the team effect coefficient (t in Equation 8) (see Table 30, page 164).

Adding staff adds work, because more learning and instruction is needed. Each of the developers must have the necessary skills and knowledge. Thus the effort needed for training and familiarisation with the problem and solution increases. CRM includes the previously mentioned efforts in the increased skill effect.

The staff working on the project may participate, often on an unanticipated basis, in other projects. For example, in pending projects, which can mean that the work of the project becomes fragmented. This means that more people and more calendar time is needed for the project. This must be counted in the team effect of the project.

## 2.12 Skill effect

Software development requires many types of skills [Hohmann 1997]. The knowledge of the application domain is especially important to analysts. The technical skills, such as knowing programming languages and environments, are important in the implementation. Communication, management and leadership skills become more important in project manager positions, though every team member needs team skills. It is also essential that some of the project team members have a higher proficiency level at the beginning. The skill effect has a large impact on the effort of the project. In COCOMO the coefficients for analyst capability multiply the effort exponent by 0.74 to 1.37 [Boehm 2000].

There are two kinds of learning time: the time used to participate in training and the time used within the productive work. Often lecturing and participating in training are separate tasks which contribute to the whole project. Mentoring is normally included in the team effect. Learning within work belongs to the skill effect. Its coefficient (s in Equation 8) is assessed using the skill effect estimation form (Table 32, page 166).

The scale of the estimation form can be more practical if specialised proficiency levels are used. An applicable 5-level scale of proficiency is defined as follows [Goldberg 1995]:

- Ignorant: The person is completely ignorant of the subject.
- Conceptual: The person has a preliminary grasp of the concepts but is not prepared for project assignments that require a working knowledge of the subject.
- Basic: The person has a working knowledge of the subject and can work on small assignments.
- Functional: The person has good working knowledge of the subject and possesses a set of useful skills and needs minimal guidance to complete assignments.
- Advanced: The person is an expert in the subject area. He/she is the authority to whom others turn for advice and solutions for unusual problems.

The time required to get to the conceptual level is a few weeks, and to basic level from a month to a few months. The functional level is achieved in about a year and an expert level takes from two to four years [Goldberg 1995].

Assessing the skill effect coefficient once for each person in a project is normally enough. However, the learning time is in principle task and developer specific. The skill level is relative to the requirements of the task. If a person performs tasks which require different skills, the skill levels should be calculated separately. The effort also depends on the task assignments. A developer who is familiar with the task needs less time to learn and vice versa. The learning time also decreases during the project. The curve in the skill effort should resemble the learning curve because there is more learning at the beginning of a project. This is accomplished by using different skill coefficients in different tasks in Equation 8 (page 30).

## 2.13 Motivation effect

Many references emphasise the importance of the motivation of the software developers. The most important motivation factors for programmers and analysts are achievement, possibility of growth, work itself, personal life and technical supervision opportunity [McConnell 1996], [Boehm 1981]. Management practices are also important. Bad management, such as displaying a lack of respect, excessive pressure and manipulation, is one of the most important morale killers [McConnell 1996]. The most important management method in motivating technical people is to provide freedom and to be supportive of them [Humphrey 1997]. One result of bad management is frustration, which is common if resource usage is too low or too high.

CRM has an estimation form (see Table 33, page 167) for assessing motivation factors. Motivation is task and person specific, but one assessment of each person in a project is normally adequate. The privacy of the motivation estimates must be assured. The developmental discussions between the project manager and the person are a natural place to discuss motivation factors, problems and the corrective actions and their influence on the productivity. As direct tracking of motivation is prone to dishonesty, the project manager estimates the realised motivation efforts afterwards (see Equation 10, page 32). The motivation effect coefficients and corresponding equation coefficients are calculated in the recalculation (see Equation 14, page 35).

## *2.14 Practical issues*

The underlying assumptions of CRM are the existence of an acceptable component structure, the ability to assess the project and human effects as well as the usefulness of the historical reuse data. In its most accurate form a considerable amount of work is needed in the estimation process. Summary level assessments will decrease the effort of the estimation. A Microsoft Excel-spreadsheet for CRM calculations was made in this study. It was used to ensure that the calculations are unambiguous.

The most important problem at the beginning is the dependency on the project repository data. Component-based development has not existed for very long and the data about the components needed in estimating future projects is not widely available. The lack of historical data can be compensated for by manual estimates, or by using data about similar projects. However, the collection of data is necessary in order to increase the quality of the estimates. There is also a lack of data on project and human effect assessments and their relations to the effort. The default values of the correction coefficients are calculated in the experimental part of this study (see chapter 5). Businesses can use similar studies to adapt the coefficients to the corporate environment.

## *2.15 Summary*

The CRM estimation process calculates the efforts of the personal tasks:

1. Design the application. The component structure contains the tracking sets so that the efforts of the components can be found from the repository.
2. Plan the project including its personal tasks and assignments of the personnel.
3. Use the estimation forms to assess the CRM effects.
4. Calculate the baseline efforts based on the tracking sets.
5. Calculate the effect coefficients by using the effect levels.

6. Calculate the effort of each personal task.

The CRM recalculation process calculates data for future estimates:

1. Collect the effort data during the project.
2. Calculate the phase coefficients.
3. Solve the efforts of the components from the baseline efforts and component counts.
4. Calculate the realised effect coefficients.
5. Add the data points into the repository.
6. Calculate the equation coefficients using regression to the data points.

The CRM calculations are straightforward substitutions for the formulas when the design of the forthcoming application and the plan of the project are available. The judgements of the effects are not easy but they are absolutely necessary.

# 3 Example of Component Reuse Metrics

## 3.1 Introduction

In the following example, Component Reuse Metrics is used to evaluate the effort of developing a simple course administration system which records course data. Each course has one subject and several students. Students study several subjects, which may also have sub-subjects. A student gets a mark for each subject (s)he passes. Figure 5 represents the class diagram of the system.

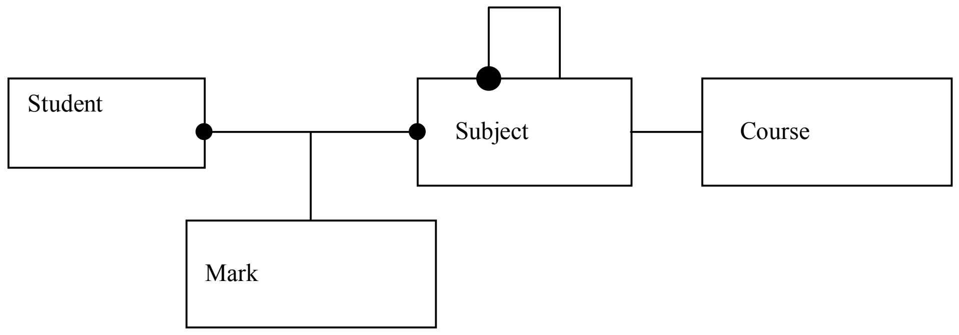


Figure 5. The class diagram of the course administration system.

The CRM-calculations are shown in several spreadsheet tables and the text gives detailed explanations. The convention in this example is to use

- white back-ground in the cells, the values of which the estimator takes from the "real world" of the example (project plan, component structure, repository, assessments),
- white back-ground in the cells, which are copied from previous tables,
- grey back-ground in calculated cells,
- bold font style to emphasise cells referenced in the text,
- italic font style in the variables of the CRM equations and
- normal font style, if the cell value is not referenced.

## 3.2 Estimation

### 3.2.1 Application design

The forthcoming product is based on a two-tier architecture and contains a database and three simple data entry windows. Table 2 contains the original component structure of the system in which each user interface window is a tracking set. Each of these connects the repository efforts of the database components and text fields together with the checking of the user entry. In this example, the customer view of the product is used instead of the implementation view. The reason for this being that the exact component structure has not yet been designed. Figure 6 represents the window showing the entry for the marks.

Table 2. Customer view of the component structure of the course administration system.

| Tracking set | Windows | Data | Text fields |
|---|---|---|---|
| Student window | 1 | 3 | 5 |
| Subject window | 1 | 5 | 3 |
| Mark window | 1 | 2 | 4 |

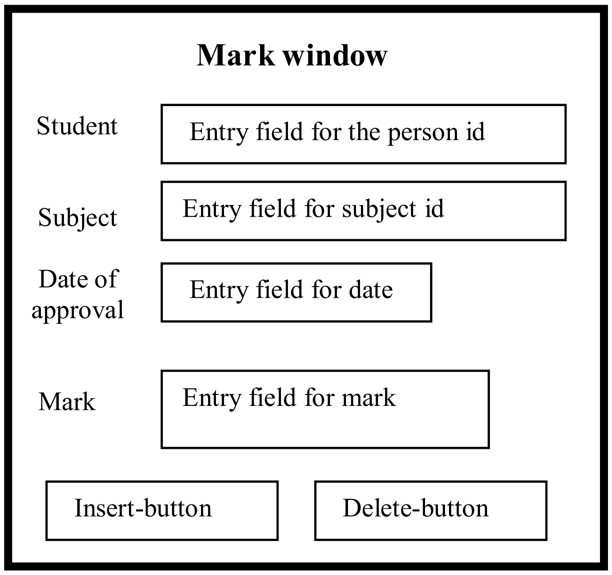


Figure 6. Mark window of the course administration system.

### 3.2.2 Project plan

The hypothesis here is that this project team has an experienced but busy project manager and a novice programmer. The manager takes part in the planning and quality assurance and the rest is assigned to the programmer. The manager has all the skills needed for his/her tasks. The low motivation estimate of the manager is based on the fact that (s)he is quite busy. This will increase the sharing of work and the need for repetition of work done previously. The low motivation estimate of the programmer is due to probable frustration arising from the lack of support from the manager. The novice programmer is left to work alone. There will be delays in helping him/her. During the project another programmer is hired and assigned to the project.

The next stage involves creating tasks for each component and assigning the work to the team members. In order to estimate the project and the human effects information about the team is needed. As the project is divided into phases, phase coefficients describing the distribution of the effort are needed. Table 3 contains the proportions of the work in each phase of the project. These numbers are based on the process model of the project and obtained from the project repository data, which has used the same process model.

Table 3. Distribution of the effort to phases.

| Phase | Phase (*h*) |
|---|---|
| Inception | 0.10 |
| Elaboration | 0.30 |
| Construction | 0.50 |
| Transition | 0.10 |

If there is more than one person in a task, the effort must be shared by the participants. The personal tasks must be planned and the percentages of each share must be evaluated. In this example the manager (20%) and the programmer (80%) share the construction of the mark window. Table 4 describes the personal tasks in the project.

Table 4. The project plan.

| Phase | Component | Who | Phase | Contribution (o) |
|---|---|---|---|---|
| Inception | Product | Manager | 0.10 | 1.00 |
| Elaboration | Product | Manager | 0.30 | 0.80 |
| | Product | Programmer | 0.30 | 0.20 |
| Construction | Student window | Programmer | 0.50 | 1.00 |
| | Subject window | Programmer | 0.50 | 1.00 |
| | Mark window | Programmer | 0.50 | **0.80** |
| | Mark window | Manager | 0.50 | **0.20** |
| Transition | Product | Manager | 0.10 | 0.60 |
| | Product | Programmer | 0.10 | 0.40 |

### 3.2.3 Assessments

The estimation forms are answered after the project planning (Appendix A: Estimation forms and survey results Page 162). As all of the estimation forms are similar, only the skill effect estimation form is shown here (Table 5). The skill level (2.30) is calculated using Equation 3, where the default weights ($w_i$) of the skill factors have been used (Table 32, page 166).

Table 5. Example of skill effect estimation.

| What is the influence of the following factors of the skill of the programmer in this project? | Answers ($a_i$) | | | | | Weight ($w_i$) |
|---|---|---|---|---|---|---|
| | Advanced/ Very large (5) | Functional/ Large (4) | Basic/ Medium (3) | Conceptual/ Small (2) | Ignorant/ Very small (1) | |
| Education | | | 3 | | | 3.15 |
| Courses | | | | 2 | | 3.06 |
| Length of experience | | | | 2 | | 3.72 |
| Quality of experience | | | | 2 | | 4.31 |
| Familiarity with the application area | | | | | 1 | 4.22 |
| Experience of team work | | | 3 | | | 3.12 |
| Familiarity with the program (to be maintained) | | | | 2 | | 4.20 |
| Knowledge of methods and tools | | | | 2 | | 4.02 |
| Personality (intelligence, emotional intelligence, sense of responsibility, diligence) | | 4 | | | | 4.13 |
| Skill level $l_s$ (Weighted average, See **Equation 3**) | **2.30** | | | | | |

Equation 4 (page 27) gives the effect coefficients corresponding to the levels. Table 6 presents the resulting skill and motivation levels and corresponding coefficients. The skill effect coefficient is 1.50. The equation parameters, which are shown in the last row of the table, are fetched from the project repository. As $\alpha_s$ and $\alpha_m$ are negative, the skill and motivation coefficients are small when the corresponding levels are high. In this example the equations are fitted to the levels and coefficients shown in the table.

Table 6. Estimates of personal productivity.

| Who | Skill level | Skill coefficient ($s$) | Motivation level | Motivation coefficient ($m$) |
|---|---|---|---|---|
| Manager | 4.10 | 1.10 | 2.5 | 1.30 |
| Programmer | **2.30** | **1.50** | 3.9 | 1.20 |
| Equation | $s = -0.222 * l_s + 2.011$ | | $m = -0.071 * l_m + 1.479$ | |

The procedure for the estimation of the project effects is similar. The estimator fills in the estimation form and gets the level of the effort, which corresponds to the effort coefficient of the particular effect. The effect coefficient can be obtained mathematically from the Equation 4. Table 7 contains the levels and the coefficients of process, project change, risk and team effects. The equation coefficients are not shown. The process effect has been included in the calculation of the baseline effort because it modifies the efforts of the components before the other project effect corrections are applied in the Equation 8.

Table 7. Estimation of project effects.

| | Level | Coefficient |
|---|---|---|
| Process | 3.2 | **1.10** |
| Project change | 4.2 | 1.50 |
| Risks | 1.4 | 1.10 |
| Team | 3.5 | 1.20 |

### 3.2.4 Calculation

The estimates for assembling these user interface components can be obtained from a project repository database. Table 8 contains the historical baseline efforts of the components of the system. These efforts have been corrected using the process coefficient (1.10), which has been estimated using the process estimation form (Table 7).

Table 8. Baseline and process effect corrected efforts.

| Baseline efforts (hours) | Efforts of components [$E_C$] | | |
|---|---|---|---|
| | Window | Data | Text fields |
| Baseline effort | 4 | 3 | 2 |
| Process corrected baseline effort | **4.4** | **3.3** | **2.2** |

Table 9 shows the summary of the baseline effort calculation. The windows for the student data and the subjects are similar to the window for marks shown in Figure 6. This is because the mark window contains two data components visualised by the buttons and 4 text fields, its baseline effort is 19.8 hours (1*4.4 + 2*3.3 + 4*2.2). See Table 9 and Equation 5.

Table 9. Process corrected baseline efforts of the tracking sets.

| | Effort | Numbers of components [$n_C$] | | |
|---|---|---|---|---|
| | | Window | Data | Text fields |
| Student window | 25.3 | 1 | 3 | 5 |
| Subject window | 27.5 | 1 | 5 | 3 |
| Mark window | **19.8** | **1** | **2** | **4** |
| SUM | 72.6 | 3 | 10 | 12 |

Table 10 contains the summary and calculation of the estimate. There is a row for each personal task, which is collected from the project plan (Table 4), the baseline effort (Table 9), estimate of the project effects (Table 7) and the estimate of the human effects (Table 6). The numbers in the source tables are shown in brackets in the column headers. For example, the estimate of the inception phase is 20.6 hours, which is the product of the baseline effort of the whole product (72.6), the percentage of the inception phase (0.10), the contribution of the manager (1.00) and the correction coefficients 1.50, 1.10, 1.20, 1.10, and 1.30 (Equation 8 and Equation 5). The baseline effort of the tasks in the inception, elaboration and transition phases is the baseline effort of the whole product (72.6 hours), as the components are not handled separately during these tasks. The estimated effort of the project is the sum of the estimates of the personal tasks, which is 236 hours (Equation 9).

Table 10. Summary of the estimate.

| Phase [5] | Component [4] | Baseline [9] | Who [4] | Phase [4] | Contribution[4] | Project change[7] | Risks [7] | Team [7] | Skill [6] | Motivation [6] | Estimate |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Inception** | **Product** | **72.6** | **Manager** | **0.10** | **1.00** | **1.50** | **1.10** | **1.20** | **1.10** | **1.30** | **20.6** |
| Elaboration | Product | 72.6 | Manager | 0.30 | 0.80 | 1.50 | 1.10 | 1.20 | 1.10 | 1.30 | 49.3 |
| | Product | 72.6 | Programmer | 0.30 | 0.20 | 1.50 | 1.10 | 1.20 | 1.50 | 1.20 | 15.5 |
| Construction | Student window | 25.3 | Programmer | 0.50 | 1.00 | 1.50 | 1.10 | 1.20 | 1.50 | 1.20 | 45.1 |
| | Subject window | 27.5 | Programmer | 0.50 | 1.00 | 1.50 | 1.10 | 1.20 | 1.50 | 1.20 | 49.0 |
| | Mark window | 19.8 | Programmer | 0.50 | 0.80 | 1.50 | 1.10 | 1.20 | 1.50 | 1.20 | 28.2 |
| | Mark window | 19.8 | Manager | 0.50 | 0.20 | 1.50 | 1.10 | 1.20 | 1.10 | 1.30 | 5.6 |
| Transition | Product | 72.6 | Manager | 0.10 | 0.60 | 1.50 | 1.10 | 1.20 | 1.10 | 1.30 | 12.3 |
| | Product | 72.6 | Programmer | 0.10 | 0.40 | 1.50 | 1.10 | 1.20 | 1.50 | 1.20 | 10.3 |
| | | | | | | | | | | | **236** |

## 3.3 Recalculation

The purpose of the recalculation is to correct the estimates during a project and after the project to provide data for future estimates. Recalculation is used to calculate the efforts of the components ($E_c$), the equation coefficients ($\alpha$ and $\beta$) and phase coefficients (h). Table 11 presents the tracking data after the project. It contains observed and classified efforts of the personal tasks. The developers also report the effort, which belongs to the motivation effort as a baseline effort. The project manager estimates the actual motivation effect and subtracts the estimate from the reported baseline effort. The total actual efforts of the personal tasks are calculated by summing the classified efforts recorded for the personal task **(**Equation 10**).** The last column shows the distribution of the baseline effort to the phases of the project.

Table 11. Summary of the tracking data.

| Phase | Component | Who | Actual [$E^{\#}_{PT}$] | Skill [$E^{\#}_{S}$] | Motivation [$E^{\#}_{M}$] | Team [$E^{\#}_{T}$] | Risks [$E^{\#}_{R}$] | Baseline [ $E\#_{B}$] | Phase [$h$] |
|---|---|---|---|---|---|---|---|---|---|
| Inception | Product | Manager | 13.0 | 1 | 2 | 1 | 0 | 9.0 | 0.077 |
| Elaboration | Product | Manager | 37.5 | 2 | 3.5 | 4 | 0 | 28.0 | 0.299 |
| | Product | Programmer | 9.0 | 0.5 | 0.5 | 1 | 0 | 7.0 | |
| Construction | Student window | Programmer | 35.0 | 10 | 5 | 2 | 3 | **15.0** | 0.389 |
| | Subject window | Programmer | 33.0 | 10 | 5 | 3 | 3 | **12.0** | |
| | Mark window | Programmer | 18.0 | 5 | 3 | 3 | 3 | **4.0** | |
| | Mark window | Manager | 2.0 | 0 | 0 | 2 | 0 | **0.0** | |
| | Mark window | Programmer 2 | 11.0 | 2 | 1 | 1 | 1 | **6.0** | |
| | Course window | Programmer | 9.5 | 2 | 1 | 1 | 1 | **4.5** | |
| | Course window | Programmer 2 | 13.5 | 5 | 2 | 1.5 | 1 | **4.0** | |
| Transition | Product | Manager | 21.5 | 2 | 2 | 2 | 0 | 15.5 | 0.235 |
| | Product | Programmer | 17.0 | 2 | 1 | 2 | 0 | 12.0 | |
| SUM | | | 220.0 | 41.5 | 26.0 | 23.5 | 12.0 | **117** | 1.000 |

### 3.3.1 Phase coefficients

Table 12 gives the distribution of the effort to the phases of the project. The phase coefficient, which is calculated from the baseline effort, is stored for future calculations (Equation 11). For example, the phase coefficient of the construction phase is 0.389, which is obtained from (15.0+12.0+4.0+6.0.45+4.0)/117. The estimate column is copied from Table 3 and the actual column is calculated from the data in Table 11, but actual efforts are used in the equation instead of baseline efforts.

Table 12. Phases of the project.

| Phase | Estimate | Baseline [*h*] | Actual |
|---|---|---|---|
| Inception | 0.10 | 0.077 | 0.06 |
| Elaboration | 0.30 | 0.299 | 0.21 |
| Construction | 0.50 | **0.389** | 0.55 |
| Transition | 0.10 | 0.235 | 0.18 |
|  | 1.00 | 1.000 | 1.00 |

### 3.3.2 Efforts of the components

Table 13 shows the summary of the baseline efforts of the tasks and the numbers of the components in the final product. During the project one window and several additional components have been added to the planned system. A component of type date control is added. As the effort of a component contains parts of the effort of all the phases of the project, the realised baseline efforts of this example must be divided by the phase coefficient of the construction phase (0.389). For example, the construction effort of the student window was 15.0 hours and the share of the student window of the whole project was 38.6 hours. The numbers from the original features are copied from Table 9.

Table 13. The numbers of the components and the baseline effort.

| Tracking set | Constr. effort | Baseline effort | The numbers of components ($n^{\#}_{c}$) | | | |
|---|---|---|---|---|---|---|
| | | | Window | Data | Text fields | Date ctrl |
| Student window | **15.0** | **38.6** | 1 | 4 | 7 | 1 |
| Subject window | 12.0 | 30.9 | 1 | 5 | 4 | 0 |
| Mark window | 10.0 | 25.7 | 1 | 3 | 3 | 1 |
| Course window | 8.5 | 21.9 | 1 | 2 | 4 | 0 |
| Sum | 45.5 | **117** | 4 | 14 | 18 | 2 |
| Original features | | **77.6** | **3** | **10** | **12** | **0** |

A row from the previous table corresponds to an equation in the equation group (Equation 12). The tasks of the inception, elaboration and transition phases are omitted because they are linearly dependent on the tasks of the implementation phase. The equation group is

Equation 17.
$$\begin{cases} E1 + 4 \cdot E2 + 7 \cdot E3 + E4 = 38.6 \\ E1 + 5 \cdot E2 + 4 \cdot E3 = 30.9 \\ E1 + 3 \cdot E2 + 3 \cdot E3 + E4 = 25.7 \\ E1 + 2 \cdot E2 + 4 \cdot E3 = 21.9 \end{cases}$$
, where

- $E1$ is the effort of the window component,
- $E2$ is the effort of the data component,
- $E3$ is the effort of text component and
- $E4$ is the effort of date component.

Table 14 shows the efforts of the components as the solution of Equation 17. They are calculated by multiple linear regression from the previous equation group. The effort of the original components (the last line of Table 13) is calculated by using the original numbers of the components and the new component based efforts: 3*6.00+10*3.00+12*2.46≅77.6 hours.

Table 14. Efforts of the components.

| $E_C$ hours | Window<br>E1 | Data<br>E2 | Text<br>E3 | Date ctrl<br>E4 |
|---|---|---|---|---|
| Component effort, now | **6.00** | **3.00** | **2.46** | **3.32** |
| Component effort, original | **4.0** | 3.0 | 2.0 | |

### 3.3.3 Effect coefficients

The recalculations of the project and human effects are based on the observed efforts (Table 11). Table 15 is a summary of the calculations of the process and project change effects (Equation 14). The estimated effect coefficients are copied from Table 7 and the realised efforts from Table 11.

The process correction is used to exclude the effect of process changes between the projects in the project repository database. The process correction coefficients are calculated separately for each component (Equation 14) by dividing the efforts of the components solved from Equation 12 by the original efforts of the components. For example, the process coefficient of the window component is 1.50 (=6.0/4.0).

The effort of the project change effect is the quotient of the final baseline effort and the original baseline effort. In this case 117 h / 77.6 h gives f=1.51 (Equation 14).

Table 15. Project effects.

| | Component | Estimated coefficient | Actual effort | Actual coefficient |
|---|---|---|---|---|
| Process [$p_C$] | Window | 1.10 | | **1.50** |
| | Data | 1.10 | | 1.00 |
| | Text | 1.10 | | 1.23 |
| Project change [f] | | 1.50 | 39.4 | **1.51** |

Table 16 shows the summary of the project and human effect calculations (Equation 14). The realised effect coefficients are copied from Table 11. Risks are the next correction in the CRM-calculation order. The coefficient (1.18) for the programmer is obtained by adding one to the quotient of the efforts of the risks (10 hours) and efforts after project change and process corrections (54,5 h) (Equation 14). The team effect coefficient (1.19) is calculated by adding one to the quotient of the efforts of the team effect (12 hours) and efforts after the project change, process and risk corrections (64,5 h =54,5 h+10.0 h) (Equation 14).

The skill coefficient of the manager (1.08) is calculated by adding one to the quotient of the effort spent in learning (5 hours) and the manager's project corrected effort (61.5 h=52,5h+0h+9h). The motivation coefficient of the manager (1.11) is calculated by adding one to the quotient of the effort assessment for the motivation (7.5 h) and the manager's effort after project and skill corrections (66.5 h = 61.5 h + 5 h). The calculations based on the symmetric equation (Equation 16) are shown for reference.

Table 16. Project and human effects.

| What/Who | Manager | Programmer | Programmer 2 | Sum |
|---|---|---|---|---|
| Baseline effort (h) | **52,5** | **54.5** | 10 | 117 |
| Risk ($E^{\#}_R$) (h) | **0** | **10** | 2 | 12 |
| Risk coefficient | 1.00 | **1.18** | 1.20 | |
| - symmetric | 1.00 | 1.13 | 1.14 | |
| Team ($E^{\#}_T$) (h) | **9** | **12** | 2.5 | 23.5 |
| Team coefficient | 1.17 | **1.19** | 1.21 | |
| - symmetric | 1.15 | 1.16 | 1.18 | |
| Skill ($E^{\#}_S$) (h) | **5.0** | 29.5 | 7.0 | 41.5 |
| Skill coefficient | **1.08** | **1.39** | **1.48** | |
| - symmetric | 1.08 | 1.40 | 1.50 | |
| Motivation $E^{\#}_M$) (h) | **7.5** | 15.5 | 3.0 | 26.0 |
| Motivation coefficient | **1.11** | 1.15 | 1.14 | |
| - symmetric | 1.13 | 1.21 | 1.21 | |

### 3.3.4 Equation coefficients

The objective of this stage is to calculate the correspondence of the effect levels and the actual correction coefficients. Data points (level-coefficient pairs) and the simple least square method are used to calculate the equation coefficients. The estimation forms could be filled in again to improve the assessments of the levels. In this example the human effects of the new programmer and risk and team levels of all of the staff are reassessed. The other assessments are kept as they were at the beginning of the project (Table 6 and Table 7). Table 17 also shows the equation coefficients, which are calculated using simple regression (Equation 4). For example, the data points for the skill effect are (4.1, 1.08), (2.3, 1.39) and (2.1, 1.48) and the calculated equation coefficients are $\alpha_S$=-0.189 and $\beta_S$=1.892 (see Figure 7).

As there is only one data point for the project change equation, $\beta_f = 0$. Also $\alpha_p = 0$ because there is only one assessment of the process level. The estimated levels are copied from Table 7.

Table 17. Equation coefficients

| | Component / Who | Estimated level | Actual coefficient | Equation coefficient [$\alpha$] | Equation coefficient [$\beta$] |
|---|---|---|---|---|---|
| Process | Window | 3.2 | 1.50 | **0.000** | 3.200 |
| | Data | | 1.00 | | |
| | Text | | 1.23 | | |
| Features | | 4.2 | 1.51 | 0.359 | **0.000** |
| Risks | Manager | 1.1 | 1.00 | 0.7 | **0.24** |
| | Programmer | 1.3 | 1.18 | | |
| | Programmer 2 | 1.4 | 1.20 | | |
| Team | Manager | 3.4 | 1.17 | 0.2 | **0.49** |
| | Programmer | 3.5 | 1.19 | | |
| | Programmer 2 | 3.6 | 1.21 | | |
| Skill | Manager | **4.1** | **1.08** | **-0.189** | **1.852** |
| | Programmer | **2.3** | **1.39** | | |
| | Programmer 2 | **2.1** | **1.48** | | |
| Motivation | Manager | 2.5 | 1.11 | -0.029 | 1.278 |
| | Programmer | 3.9 | 1.15 | | |
| | Programmer 2 | 3.8 | 1.14 | | |

Figure 7 shows the correlation of the skill coefficient and the skill level (based on the data of Table 17). Generally, the values of the equation parameters depend on the meaning of the scale of the answers to the estimation forms. They are naturally defined so that increasing the level of skill, motivation and process will decrease the effort coefficient and the effort. Decreasing the level of risk, project change and teamwork has the same effect.

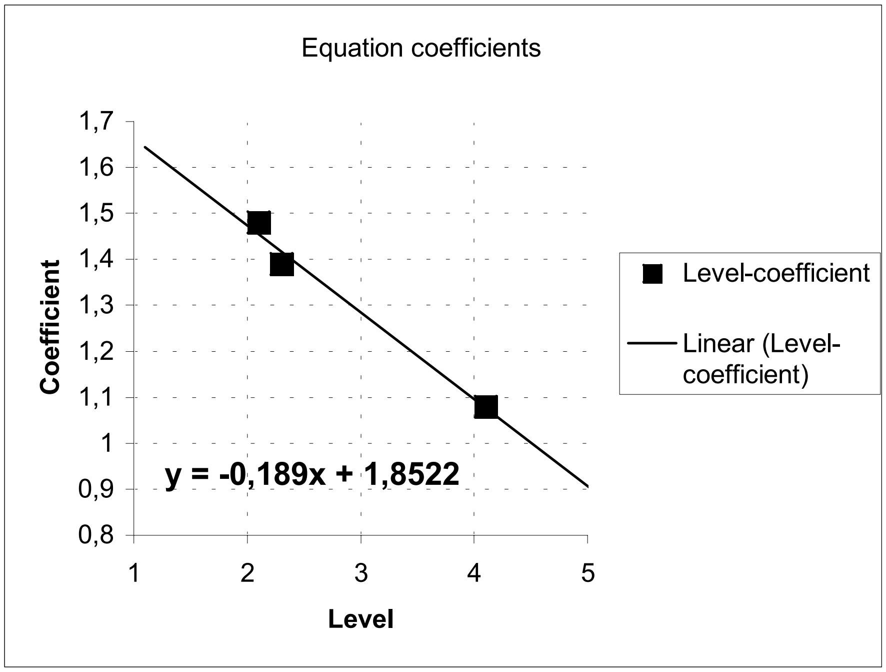


Figure 7. The equation coefficients.

The CRM calculation and recalculation can be done automatically by assigning the planned numbers to the CRM equations.

## *3.4 Conclusion of the example*

The CRM calculation and recalculation are straightforward substitutions to the given formulas. Tables visualise the calculations and spreadsheets can be used to create software for CRM-estimation. A database solution depicted in Figure 3, page 23 is more automatic than spreadsheet-based calculations, because adding rows for tasks and persons and fetching data from the repository can be facilitated. Interfaces to external software engineering, project management and time reporting software are also very useful.

# 4 Empirical Study for Evaluating CRM

## 4.1 Introduction

This chapter presents the results of the survey and the field experiments, which were conducted in order to validate CRM. The survey was sent to every project manager who was a member of the Finnish Information Processing Association. These experts assessed the importance of the project and the human factors of CRM estimation forms (see Appendix A: Estimation forms and survey results, page 162). The respondents estimated the influence of the asked factor in software development using a 5-level (5 to 1) scale corresponding to very large, large, medium, small or very small influence. The averages of the answers establish the default values of the quantitative parameters of CRM, though CRM itself was not presented to the respondents. Another section of the questionnaire covered current practices and abilities to provide the data that CRM requires. The results of the survey are useful even though there is no explicit connection to CRM.

The data from the survey is presented in the appendices A, B and C and the narrative descriptions and graphical illustrations are given in the text of this chapter and in [Virtanen 2001]. The narrative presentation is ordered by the classification and the importance of the questions in the survey. In the text, we use combined categories {always, mostly}, {very large, large}, {never, rarely}, and {small, very small} for simplicity.

## 4.2 Survey

The addresses of all 955 project managers and system managers were selected from the member database of The Finnish Information Processing Association. The project and system managers were selected because they are normally responsible for estimating software projects. About 60% of them lived in the Helsinki area. In May 2000, 516 letters were sent out and in June 434 letters. 5 foreign addresses were ignored. In total 70 replies were received, three of which were not filled in. The percentage of replies was 7.1 % (67 replies out of 950). The small proportion of respondents diminishes the representativeness of the results among Finnish project managers as a whole. The results of most of the individual questions are statistically significant because the number of replies was larger than 50. However, comparisons of the answers to different questions must be omitted and any conclusions must be drawn carefully. The questions, which could not pass the Chi-square test with at least 98% confidence have been omitted, the reason being that the answers were distributed so evenly that the zero-hypothesis of randomness could be true. Questions about ER-analysis, application frameworks, design patterns and use cases have been omitted for the same reason. These questions have been marked with "*" in both the figures and in the appendices.

A phone survey was conducted in June 2000 in order to evaluate the differences between the obtained replies and the replies of people who did not respond. 30 randomly selected people were called. One of them had changed his job and had not received the survey, 14 of them would have responded in the same way as the actual respondents, and 15 did not reply because estimation of software development projects was not in their brief at that time. The phone survey confirms that the results are representative of the population of project managers.

The 67 respondents had considerable experience in estimation of software development projects, on average 14.6 years (confidence interval 1.5 years) and only four of them (6.0%) had less than 5 years experience. Eight of the respondents were interested in taking part in an interview. The results of 6 interviews are presented later in this chapter; two of the eight were not interviewed due to work commitments.

## *4.3 Background information of the respondents*

In order to obtain some background information from the respondents a few questions were asked about their current practices in relation to software development and software project estimation.

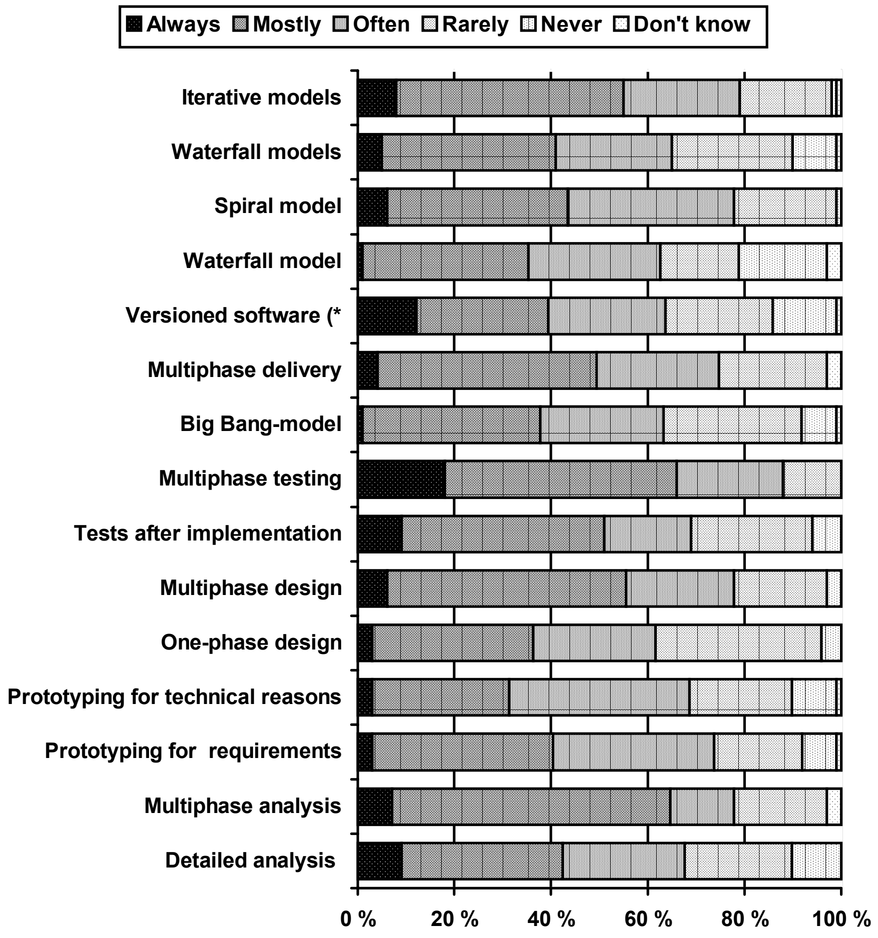


Figure 8. Current use of process models.

Figure 8 presents the current usage of process models (Table 41, page 173). Analysis, design, testing and delivery were often done in more than one phase. If the use of one phase is classified as a waterfall model and the use of multiple phases is classified as an iterative model, 55% of the respondents mostly used iterative models and correspondingly 41% of the respondents used waterfall models. Thus the estimation models must support iterative software development. Due to the versatile use of these models it is important that estimation methods are not restricted to one process model. Prototypes were mostly used by 40 % of the respondents in order to ascertain customer requirements and by 31% for technical reasons. CRM supports these current practices by providing feedback data for the estimation of successive projects and successive iterations of projects. The process correction of CRM facilitates the use of different process models (see chapter 4.6 on page 72).

Figure 9 shows that the usage of design methods is low (Table 42, page 174). The use of the wall-board in analysis and design meetings was common.

The answers relating to the use of the traditional entity-relationship model were spread so evenly that the result was not reliable (chi-square value 21%). 29% of the respondents mostly used data flow analysis and only 19% primarily used object-oriented methods. CRM can be adapted to traditional analysis and design methods, although component-based and object-oriented methods are preferred.

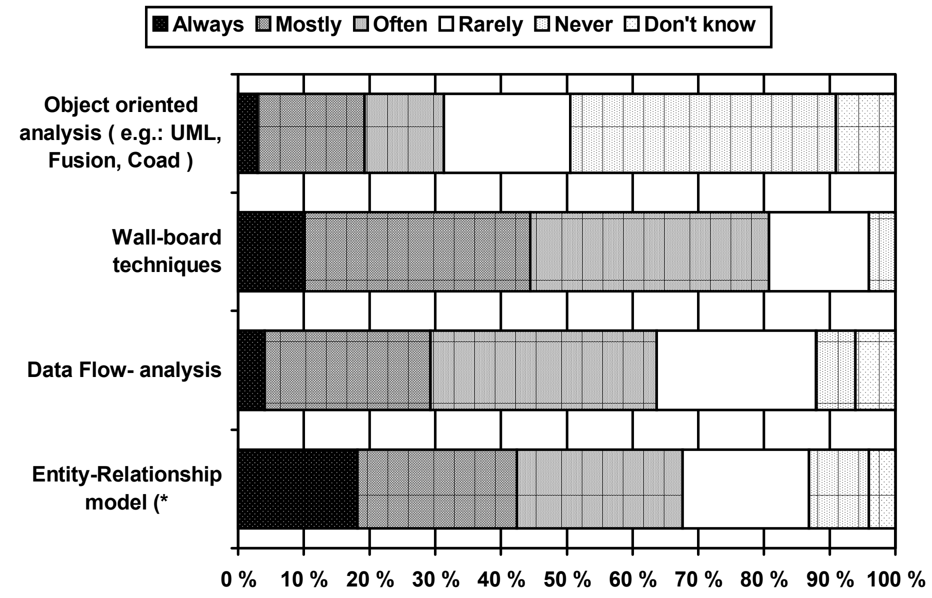


Figure 9. Current use of analysis and design methods.

Figure 10 illustrates the usage of components (Table 43, page 174). As can be seen from the survey, self-made module libraries, application frameworks and design patterns and self-made class libraries are the most important reusable components. Seventeen percent of the respondents mostly used acquired class libraries and module libraries. Modern component technologies were rarely used. Java Beans were mostly used by 12 % of the respondents. The corresponding numbers for ActiveX, DCOM and CORBA were 7%, 3% and 4% respectively.  Software components were rarely produced for sale. The market for them has not yet been created (see [Szyperski 1998]).

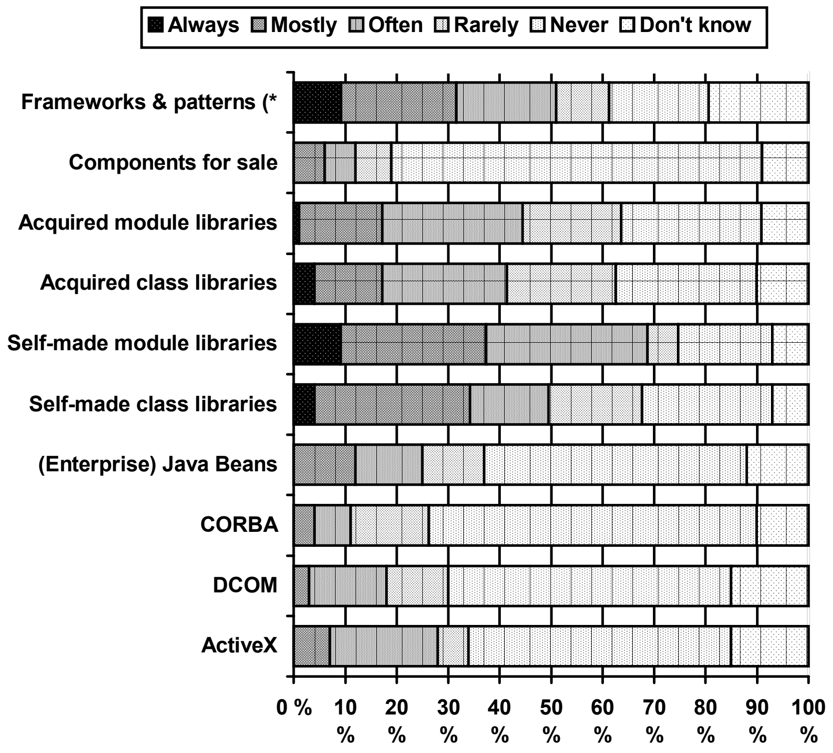


Figure 10. Current use of component technologies.

Figure 11 illustrates the current estimation methods and metrics (Table 44, page 175). Task based estimation is the most common estimation method. The forthcoming project is compared with similar projects in 62% of the cases. The answers that relate to the adaptation of the project to a budget were more even. The respondents utilised the project tracking history in a variety of different ways. In theory, it is better to calculate the budget according to the objectives of the project but in practice the project is planned by the budget in 40% of the cases recorded here. The Function-point method has some active users, but 51% rarely used it. Only two percent of the respondents mostly start a project without an estimate. Forty-six percent of the respondents answered "don't know" to the question about COCOMO and another forty-six percent stated that they never used it.

Numbers of windows, reports and database tables are the most common metrics. Seventy percent of the respondents generally counted them. Numbers of use cases, subsystems and classes were also counted but not as often. Lines of code (LOC) were rarely counted. This was because the production of components for sale was rare (see Figure 10,)and the number of customers for these components was low.

The previous results suggest that the introduction of CRM is easy because CRM can be seen as an extension of current common practices.

CRM estimates tasks by using components which resemble the most important current metrics, such as numbers of windows etc.

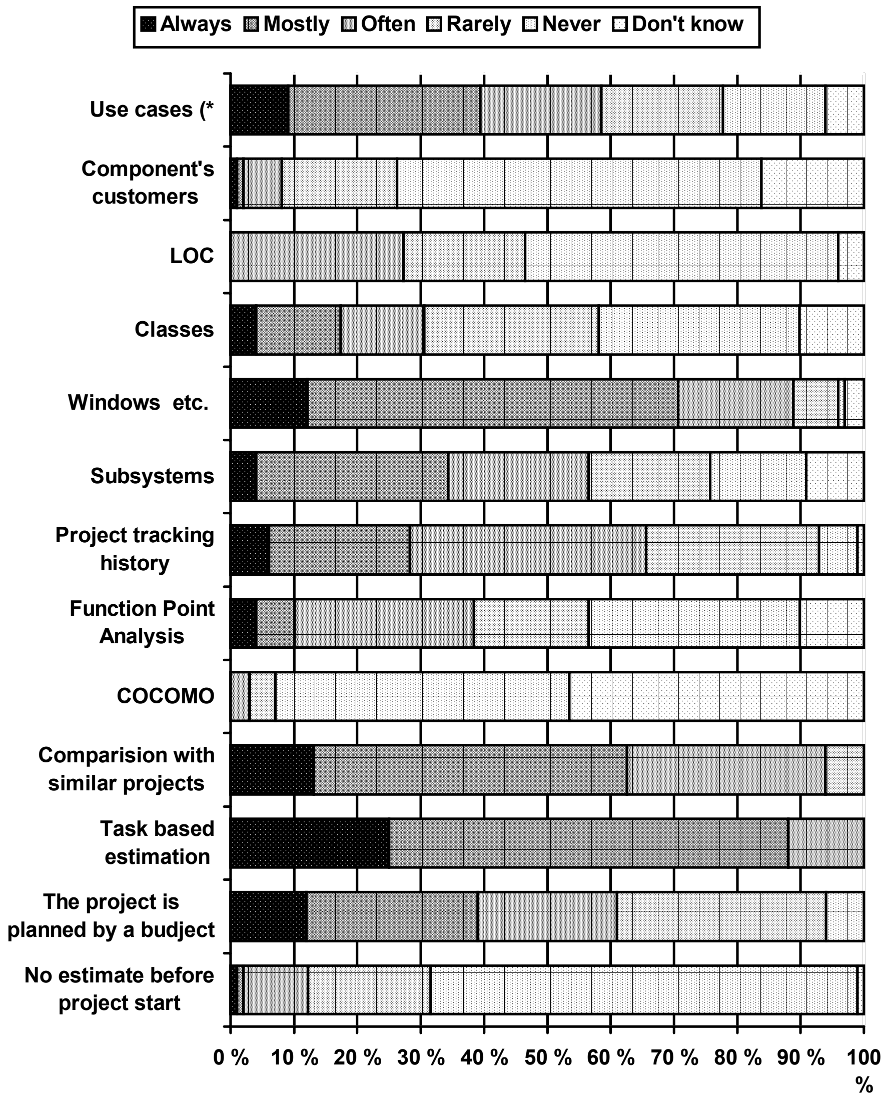


Figure 11. Current use of estimation methods.

## 4.4 Distribution of the effort

The respondents to the survey saw the distribution of the effort of a typical software development project as being: planning and implementation of the components 35 %, project change 21 %, teamwork 13 %, skill 11 %, risks 11% and motivation 10 % (Figure 12, Table 35, page 169). The process effect was not included in this question because it measures the differences between two different processes. This result confirms that the effects which CRM measures are import parts of the effort. This finding is based on the opinions expressed by the respondents and not on observations from actual projects. The default values of the correction coefficients of the CRM method (see Equation 8, page 30) can be calculated by substituting the previously obtained efforts of components $E^{\#}_{B}$=35 h, project change $E^{\#}_{P}$=21 h, teamwork $E^{\#}_{T}$=13 h, skill $E^{\#}_{S}$=11 h, risks $E^{\#}_{R}$=11 h and motivation $E^{\#}_{M}$=10 h into Equation 14 (page 35):

- project change f = 1+21/35 = 1.60,
- risk r = 1+11/(21+35) = 1.20,
- team t = 1+13/(21+35+11) = 1.19,
- skill s = 1+11/(21+35+11+13) = 1.14 and
- motivation m = 1+ 10/(21+35+11+13+11) = 1.11.

The correction coefficients compare the effort of the effect in question to the efforts of the preceding effects in the calculation order. Therefore, the skill and risk coefficients are different though their overall proportions are the same. The problem of distribution based coefficients are that a single change of a coefficient will change all the coefficients.

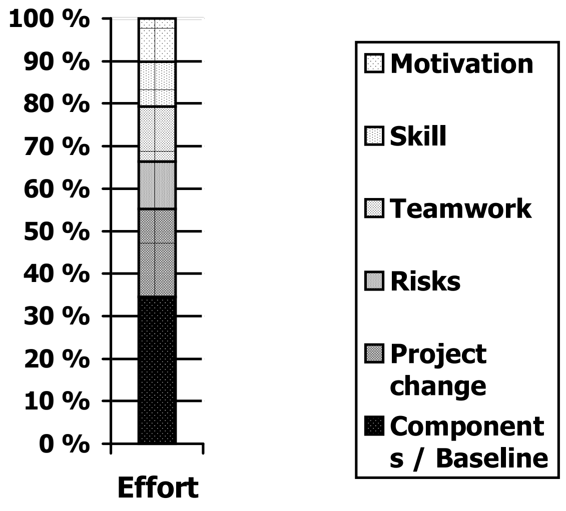


Figure 12. Distribution of the effort by correction effects.

## 4.5 Counting Components

One question in the survey was used to assess whether the average effort of reusing a component is a useful estimate for successive reuses. In the example, the respondents estimated the effort in different reuse cases of a business graphics component (like Microsoft Chart Control), with the average effort being 20 hours. The deviation from the time frame of 20 hours was less than 2 hours in 17% of cases, 2 to 5 hours in 26% of the cases, 5 to 10 hours in 30% of the cases, 10 to 20 hours in 17% of the cases and more than 20 hours in 9% of the cases (Figure 13, Table 36, page 169). The answers reveal that CRM estimators, as is the case with all measurers, cannot apply the effort averages to different cases without consideration.

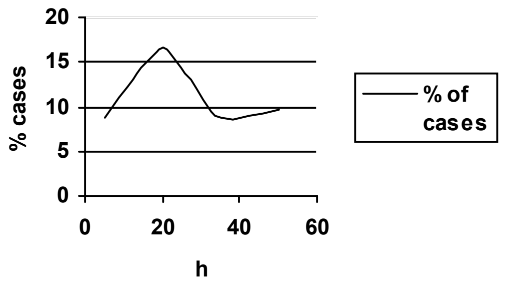


Figure 13. Distribution of effort by reuse cases.

## 4.6 Process effect

The process effect in CRM is used to calculate the influence of the process changes to the effort. Figure 14 shows the factors in the process effect (Table 28, page 162). Software development requires many co-ordinated methods and tools and none of them alone can dramatically improve the productivity of a project. The effect of the development environment was large in 41% of the answers. The impact of the tools was slightly lower. Testing tools were the most important. The effect of the programming language was large in 38% of the answers. Configuration management tools, database tools, analysis and design tools and documentation tools were less important. The effect of project management tools was considered small. The most important process factors were testing methods, special quality requirements (such as fault tolerance in medical applications), assuring of generality and phasing the project. Quality assurance methods, project management methods, finding and assessing components and documentation standards were considered to be less important.

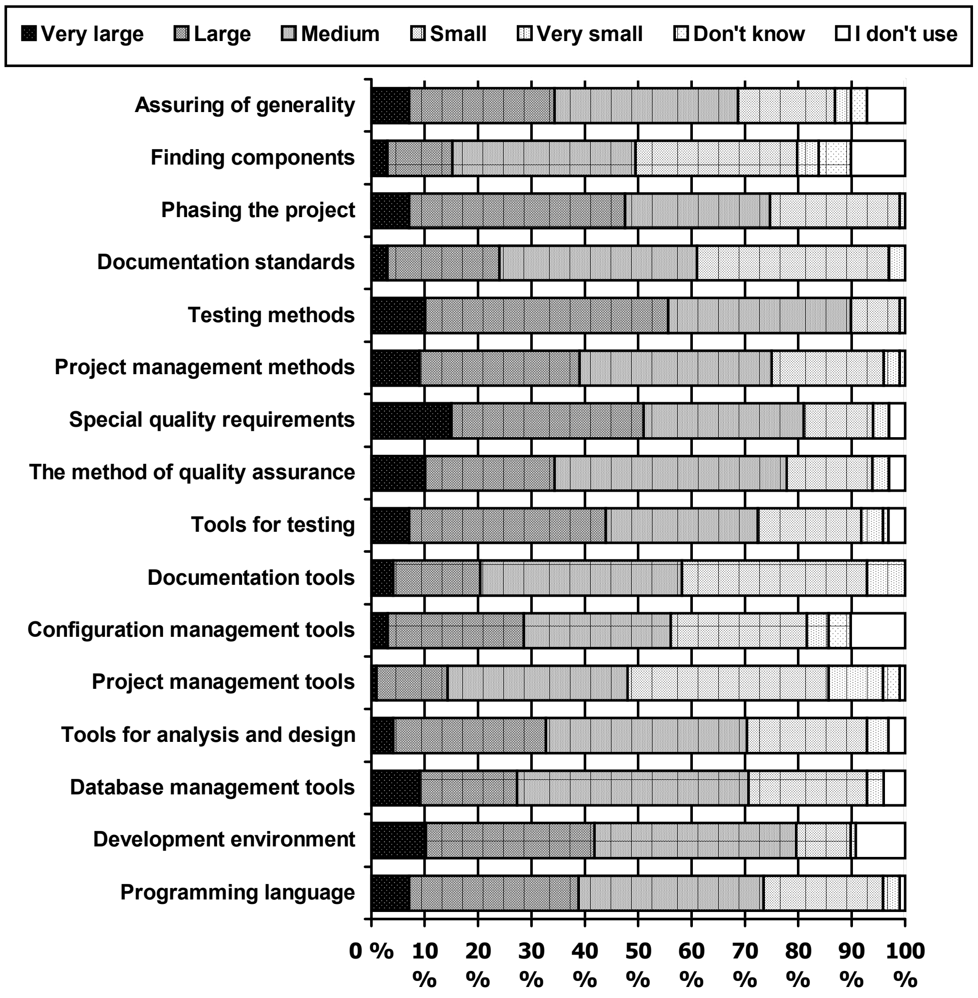


Figure 14. Factors of process effect.

## *4.7 Project change effect*

The section of the survey related to the amounts of added and removed features showed that the adding of new features is common. These numbers are illustrated in Figure 15 (Table 38, page 170) and in Figure 16 (Table 37, page 170). The amount of added effort is 10-20% in 43% of the cases and 20-50% in 22% of the cases. The amount of cancelled effort is less than 10% in 46% of the cases and 10-20% in 15% of the cases. In Jones' study the average amount of additional features was +35% [Jones 1994]. The answer to the question reported in Figure 12 (the share of the project change of the final total effort is 21%) differs from the answers in this chapter because the additional and removed features are compared to the planned effort and not the final effort.

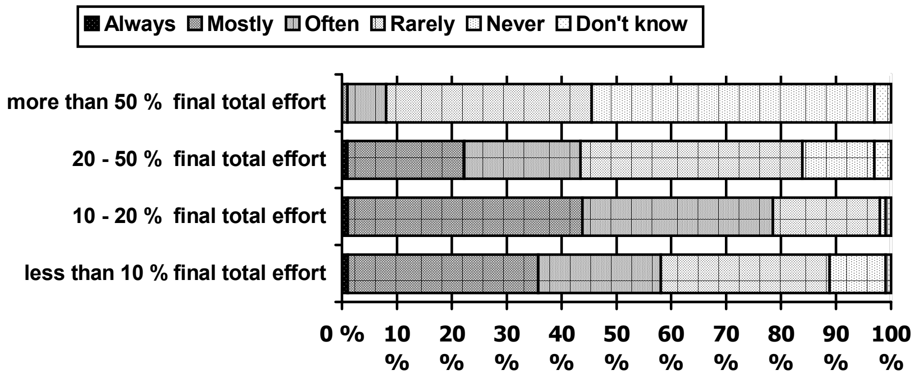


Figure 15. Added features.

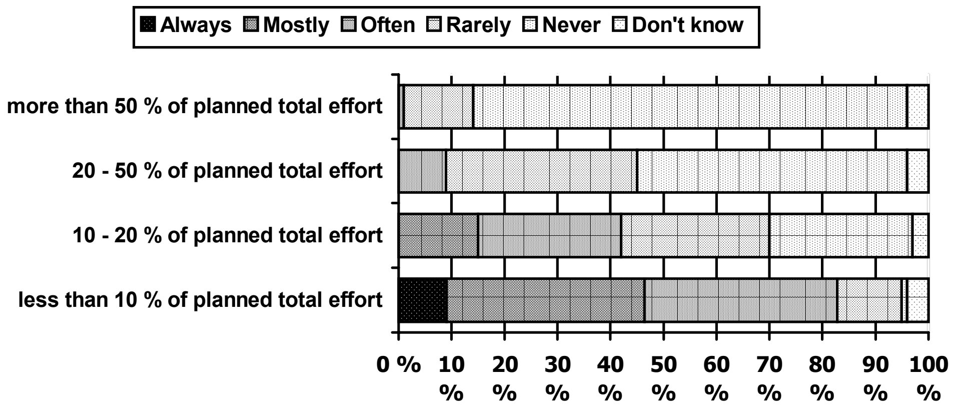


Figure 16. Removed features.

Figure 17 shows the factors of project change effect (Table 29, page 163). The most significant factors are end-users' views, inaccurate analysis, requirement errors, good ideas developed during the project, and inaccurate design. Technological surprises and the views of the project manager are also common factors. The estimators in the case studies had some conception of possible areas of new features (see chapter 4.15, page85). The end user organisation normally pays for these features and this can often be done without renegotiating the contract.

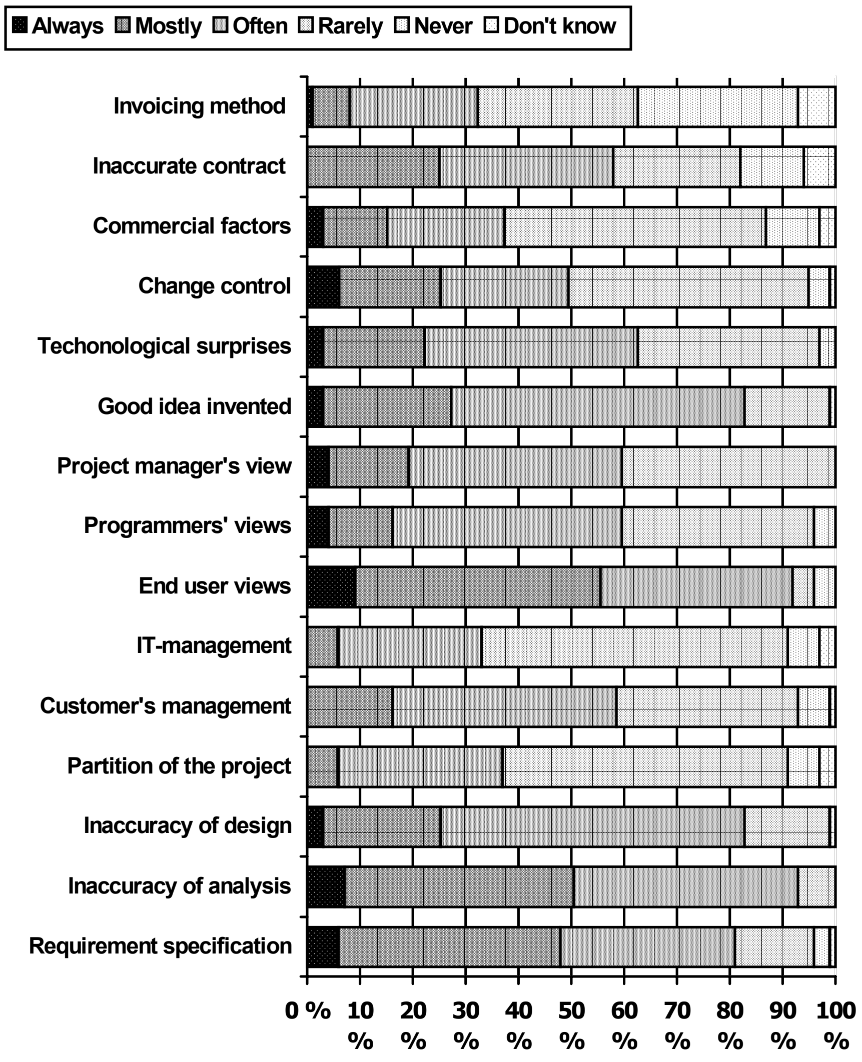


Figure 17. Factors of project change.

## 4.8 Team effect

Figure 18 shows the factors needed for estimating the team effect coefficient (Table 30, page 164). They can be classified as meetings, travel, documentation, discussions and conflicts. The effort of discussions was large (discussions with users, with customer's management and personal supervision). The handling of conflicts requires a large effort (disturbances in the information flow, disputes about objectives, disputes about working methods and interruptions of work). The influence of meetings was also large (ad hoc meetings, planned meetings and meeting practices). Writing and reading email was more important than writing and reading meeting minutes and other documents. One of the case studies revealed that travel time is significant in international projects, though it is a minor factor for the average respondent in this case as they work in the Helsinki area. The organisation and the numbers of developers and users add another view to the factors of the team effect. Interviewee number 3 (see chapter 4.13.3, page 83) noticed the effect of synergy, which tends to decrease the effort because it increases the productivity in teamwork.

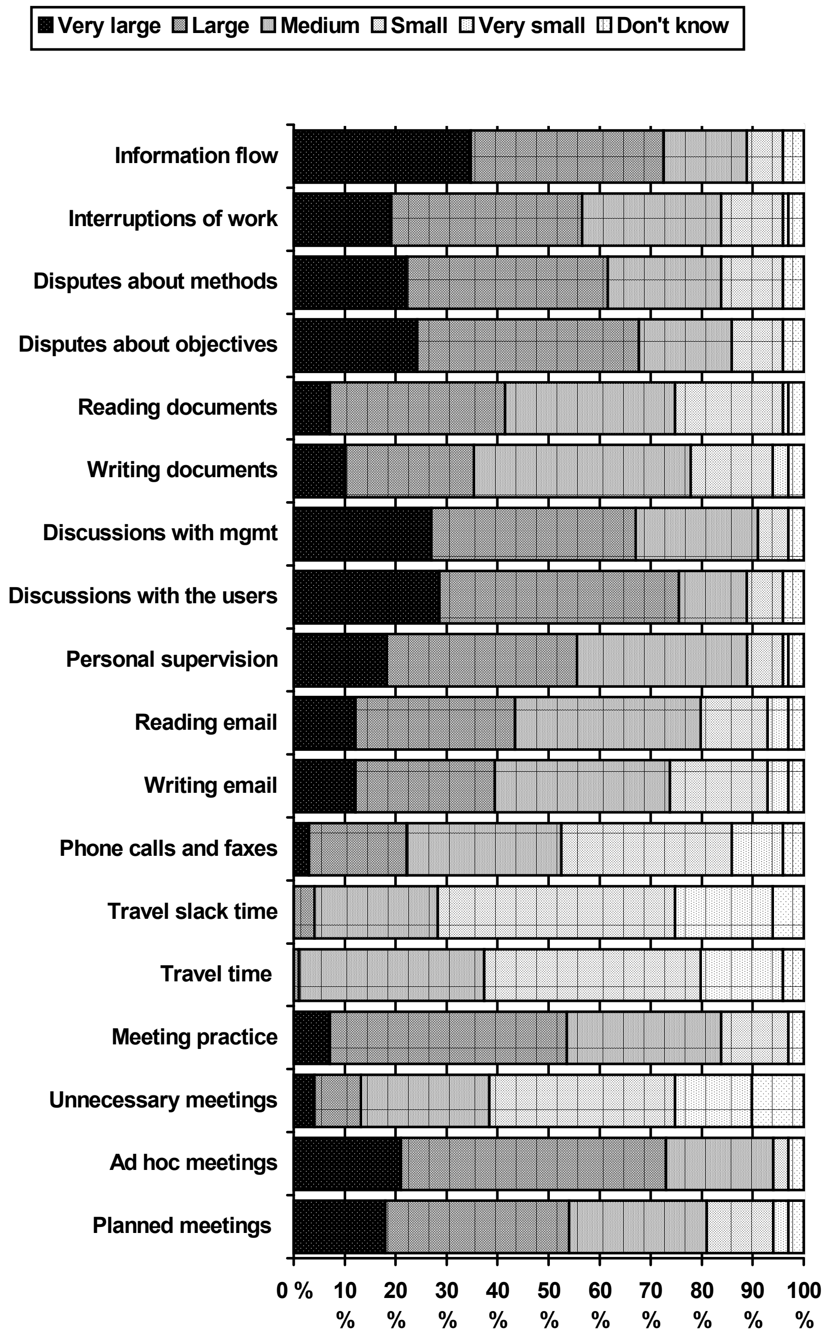


Figure 18. Factors of team effect.

## 4.9 Risk effect

Figure 19 shows the factors needed for estimating the risk effect coefficient (Table 31, page 165). The questions assess the expected influence of the risk if it becomes true. The respondents thought that personnel risks are the most significant risks in their projects. These include changes in personnel and sickness. The problems related to the organisation are also important (unpunctuality of a contractor, subcontractor or customer and failures in subcontracting and purchasing and organisational changes). The failures in technology and unexpectedly difficult software bugs were more important than technical disturbances and sabotage. Disputes and carelessness were seen as being less important. Estimation errors, economic risks and juridical risks make up a group of management related risk factors. In summary, the risks of software development are considerable (see chapter 2.10, page 43).

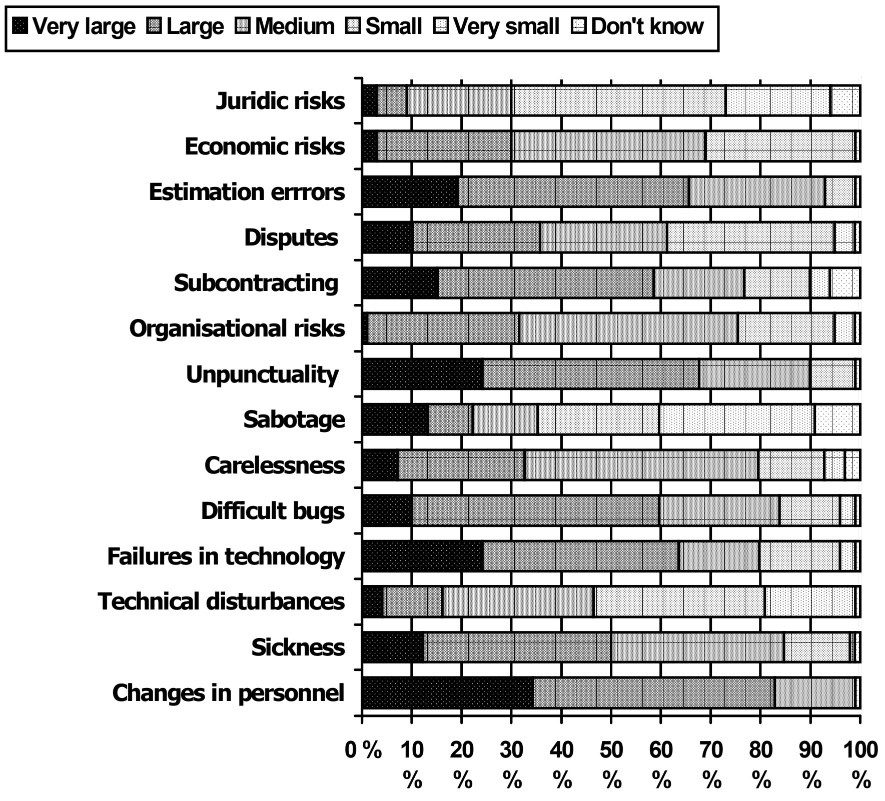


Figure 19. Factors of risk effect.

## 4.10 Skill effect

Figure 20 shows the survey results related to skill factors (Table 32, page 166) The quality and length of the experience, the familiarity with the application area and with the program to be maintained, personality and knowledge of methods and tools were seen as being important. Education, experience of teamwork and courses attended were considered to be less important.

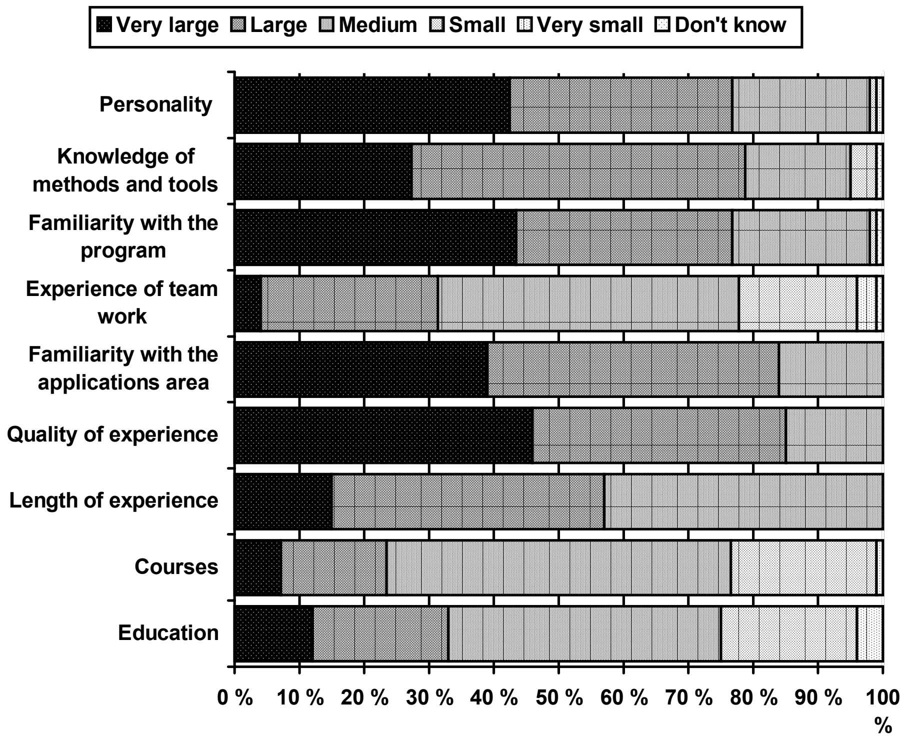


Figure 20. Factors of skill effect.

In order to assess the effect of the skill to the effort, the personnel were classified into four categories (expert, professional, normal, novice) for the assessment of skill. Each person was assumed to be motivated and self-directed in learning and accomplishing given tasks. In the survey the respondents estimated the additional effort (original 20 h) needed for learning (including supervision) during the reuse of the example component, which was the business graphics component Microsoft Chart. The expert does not need additional effort. They estimated that the additional effort of a professional is less than 25% in 81% of cases. The same effort for a normal developer is less than 25% in 16% of cases and 25%-50% in 34% of cases. The additional effort for a novice is much larger: more than 50% in 91% of cases.

## 4.11 Motivation effect

Figure 21 shows the results of the question relating to managers' motivation factors (Table 33, page 167). Here, the respondents assessed themselves. Challenges, the work itself, self-development, independence, the possibility of achieving results and relationships within the team were all seen as important motivators. Working conditions, salary and benefits and career were seen as being less important. The motivation factors of the team members are listed in the appendix A (Table 34, page 168). The Pearson correlation between the replies of team members' motivation factors and project managers' motivation factors was from 0.79 to 1.0, excluding the possibility for initiative and independence (0.61) and responsibility (0.67). As the motivation factors of the team members and leaders are approximately the same, it is possible to use only one set of motivation factor weights in the CRM-calculations. Table 34 (page 168) shows the weights for the team members.

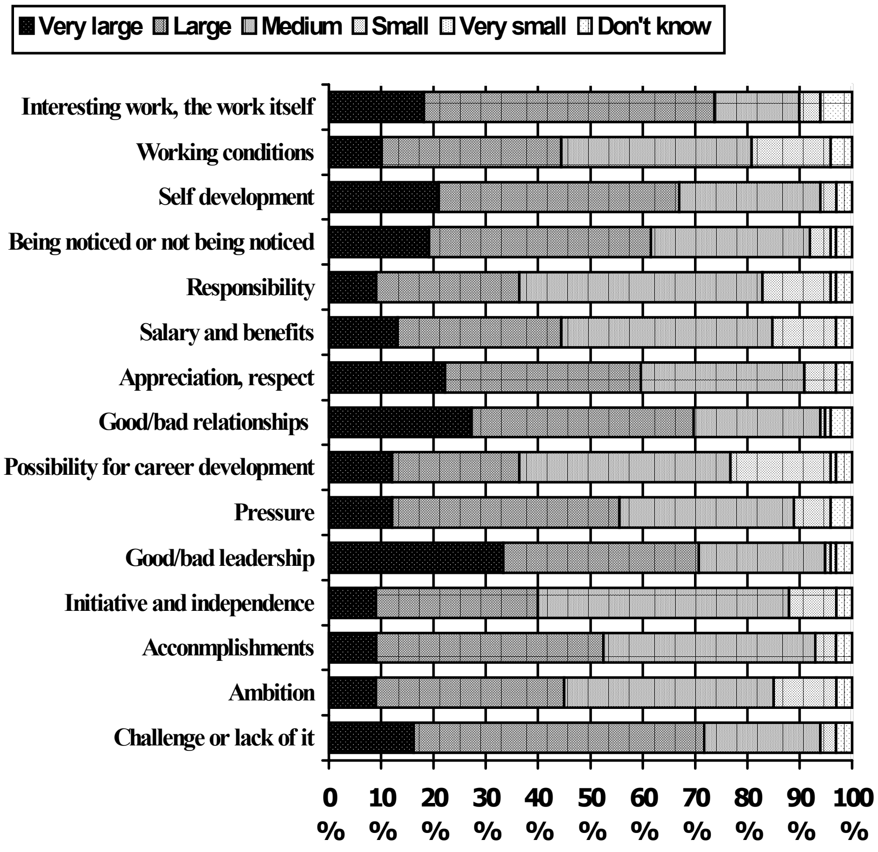


Figure 21. Factors of motivation effect.

### 4.12 Practical issues

Figure 22 shows that the survey respondents trusted in the idea of estimating the effects needed in CRM calculations, especially with regard to any required recalculations after the project (Table 45, page 176). The ability to estimate the process effect was 28%, when the new methods and tools have not been tested, 38%, when they have been tested in a trial and 56% when they have been used in actual projects. The ability to estimate the project change effect after the project was good. Naturally, the estimation was seen as being easier after the project than beforehand.

In its most accurate form the CRM estimation process requires a considerable number of calculations. The survey respondents considered the amount of estimation work to be reasonable if these estimates are made for each project or for each subproject and for each person. This was seen as being particularly so when only estimation work that is considered necessary is carried out. Each personal task can have its own estimation forms, if required (see Figure 3, page 23). The case studies confirmed this assessment.

It is possible to create useful project histories because estimating and tracking the effects of CRM separately after the project is feasible. Tracking of the co-ordination work has also been found to be successful in another study [Toffolon 2000]. In the case studies no historical data was available and the project managers estimated the correction coefficients directly. Additionally, the estimation forms supported the estimation process.

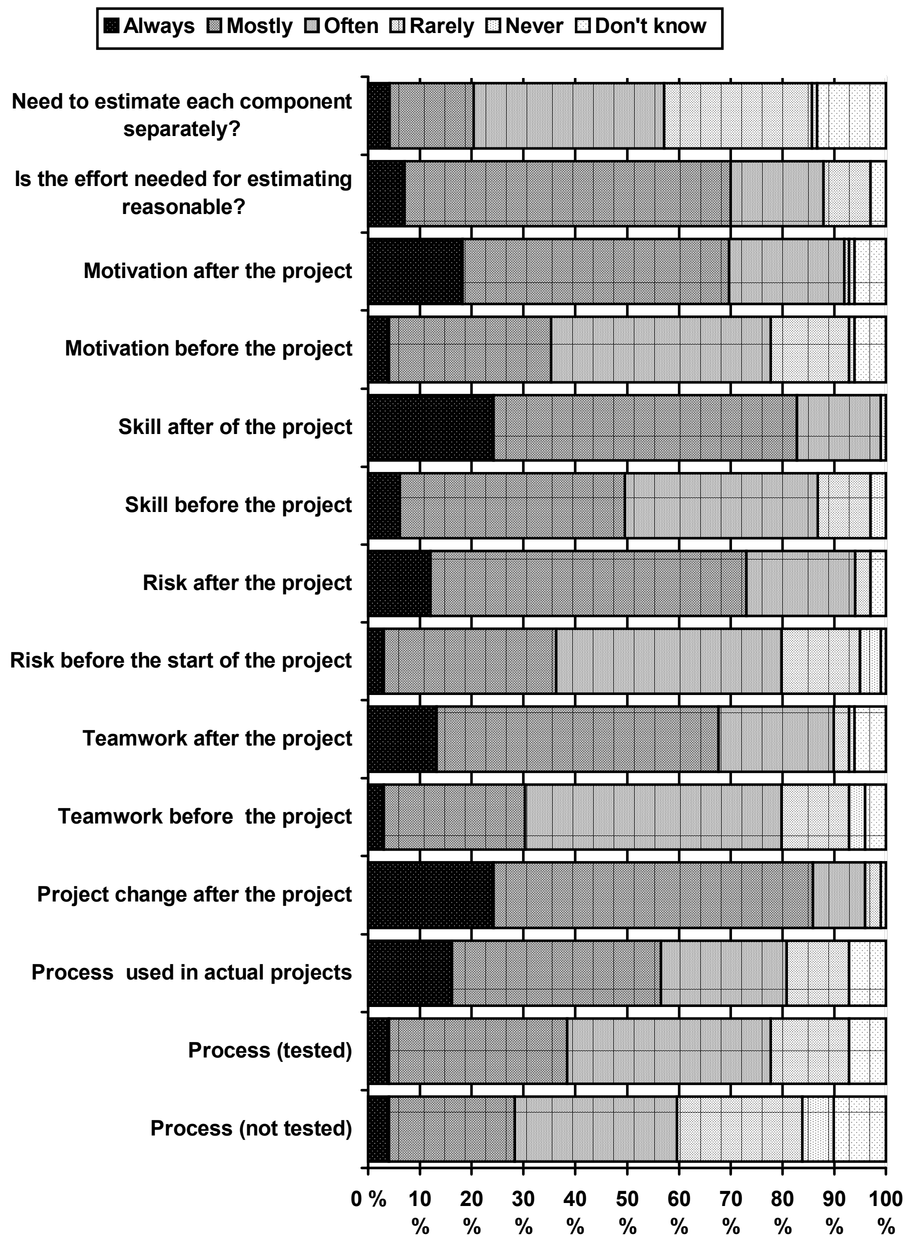


Figure 22. Ability to estimate.

## 4.13 Interviews

### 4.13.1 Overview

This chapter reports the interviews conducted. They were made up of a half-day interview and a trial of CRM in selected companies and their projects. Two of the original eight voluntary companies did not participate in the interviews because of work commitments. In interviews 1, 2 and 3, a light estimation of actual projects was conducted (see chapters and 4.14, 4.13.2 and 4.13.3 correspondingly). In the other three interviews there were no on-going projects to assess within the session.

In all of the six interviews, the CRM method was presented to a project manager in the company. The general opinion was that CRM is useful and it could be applied to the applications used by the company. The problems of component definition were addressed and the project managers thought that the problems could be solved.

### 4.13.2 A large traditional application

In interview 1 the effort of a large transition project of a rather traditional application was recalculated successfully within the session. The parts of the application were viewed as business components as they were similar in succeeding projects. They were good CRM components because the project tracking history showed that the efforts of the components were equal to within a reasonable margin. In this case, CRM was applied successfully to a project in which a commercial-off-the-shelf product was introduced to a branch office.

### 4.13.3 A HTML-application

In interview 2 the effort of an HTML application was seen as being difficult to estimate using CRM because the structure of the software was unclear. There were large differences in the effort of using the same user interface component in various places because a large amount of hand coding was needed in each case. Copy-paste-reuse increased the differences.

## *4.14 A small case study*

In the small case study in interview 3 the effort of a small mobile application was estimated on 29.6.2000 in two hours. Components were easy to find because they were common user interface components included in the development toolkit. The effort of a component was easy to assess because the project manager had experience of similar components in a PC environment. The baseline effort was 108 hours (= 4*8 + 4*4 + 5*4 + 3*8 + 1*16 hours; Equation 5). The original estimate given by CRM was 184 hours (= 1.7 * 108 hours). The project manager estimated the effect coefficients because no history data was available. The product of these coefficients was 1.7. Table 18 shows the baseline and total efforts that were initially and finally recorded on 10.11.2000.

Table 18. Baseline efforts of the small case study.

| Component | Effort $E_C$ | Estimated amount $n_C$ | Final amount $n^{\#}_C$ |
|---|---|---|---|
| Window | **8** | **4** | 9 |
| Button | **4** | **4** | 9 |
| Text field | **4** | **5** | 3 |
| Combo box | **8** | **3** | 3 |
| List | **16** | **1** | 1 |
| Time selection | 8 | | 2 |
| Baseline effort | | **108** hours | 176 hours |
| Total effort | | **184** hours | 204 hours |

The assessment of project and human effects initially looked straightforward, but their actual influence was somewhat surprising. The first implementation of the project was discarded after 381 hours. The programmer was seen as not being up to the task and the project was started again from the beginning. The risks of the second implementation were smaller because technical problems had been solved by neighbouring projects. The original estimate was 184 hours and the actual effort of the second, successful, implementation was 204 hours (Table 19). The effect levels in the second column are based on the assessments of the project manager. The actual efforts in third and fifth column has been obtained from the time reports. The effect coefficients are calculated by substituting the efforts into Equation 14 (page 35).

Table 19. Recalculation of case 1.

| | | First implementation | | Second implementation | |
|---|---|---|---|---|---|
| Effect | Effect level | Effort (hours) | Effect coefficient | Effort (hours) | Effect coefficient |
| Baseline | | (108 h) | | 108 h | |
| Process | | 0 h | 1.00 | 0 h | 1.00 |
| Project change | 2.35 | 0 h | 1.00 | 68 h | 1.63 |
| Risks | - | 100 h | 1.93 | | 1.00 |
| Team work | 2.03 | 19 h | 1.09 | 10 h | 1.06 |
| Person 1 skill | 1.28 | 205 h | 1.90 | | |
| Person 1 motivation | 1.36 | 57 h | 1.13 | | |
| Person 2 skill | 3.38 | | | 9 h | 1.05 |
| Person 2 motivation | 3.95 | | | 9 h | 1.05 |
| Sum (total 585 h) | | 381 h | | 204 h | |

This case was a good example of the need for good assessments. The project manager misjudged person 1, the amount of the additional features and the risks of the project. The product of the correction coefficients was originally 1.7 and finally 5.42 (see Equation 8. Page 30), because risks and the skill of person 1 were assessed wrongly.

## 4.15 A CRM evaluation project

A-system (name changed) was a medium-size software house that produced custom software for media and healthcare archiving and medical information technology applications in Finland. It employed about 100 persons. The examined projects were Web UI and Quick UI and they were looked at in one of A-system's branch offices. The data for the estimation was collected from an already finished Web UI project and CRM estimation was tried in the successor to that project, Quick UI.

The estimation process differs from the CRM process described earlier because this was the first CRM estimation. Specifically:

- Assessments of the history data of the Web UI project were needed to calculate the efforts of the components.
- The component structures were obscure,
- The equation coefficients and effect coefficients were both estimated because the amount of data from the Web UI project was so small,
- The actual efforts of different effort types were estimated because the time reporting did not separate them.

In this case study older versions of the CRM equations were used (Equation 7, page 12). The other CRM equations differed correspondingly. For example, the version of the Equation 14 divided the efforts of the effects by the baseline effort. The values of the coefficients for the older equations are often less than one. The problem with the addition based Equation 7, was its weakness in adapting to the changes in a project. In this case study the effort estimates changed greatly because the project changed and unanticipated risks arose. This also influenced the team and human effects and changed their realised coefficients, though the actual levels of these effects did not change.

The original estimate was adjusted once and a recalculation was done after the project. The numbers are summarised Table 20 (page 90) Table 21(page 91) and Table 22 (page 92).

### 4.15.1 A-System Quick-UI-project v. 1.0 - estimation

The first evaluation of the Quick UI project was done on 4.12.2000 by using the recalculation data from the earlier Web UI project. Its project manager supplemented the available written data. The calculation took approximately half a day. The requirement phase of the Quick UI project had already started and it was planned to end on 11.12.2000. This project did not produce any kind of user interface or a solution, which was a prerequisite of sending an offer to the customer concerning the implementation of the Quick UI. The parameters of CRM could be found from the actual data of a very similar Web UI project, but the outcome of the Quick UI project was not defined.

The component-based estimate was based on the product structure and efforts of the components of the earlier Web UI project. During the recalculation it was found that the accuracy of the history data was not adequate and that the share of single components of "search results" and the "difficult button" was very large. The effect of single components in the history data was excessive because it was known that it was not necessary to duplicate the work of the earlier project. The baseline effort of the implementation was calculated to be 620 hours if the data from the earlier project were used as such.

The project manager estimated the levels of the CRM effects using estimation forms and also the correction coefficients because the amount of history data was very small. The most influential project change factors were commercial factors, inaccurate agreement between A-System and the customer and the views of the project members and the users. The project manager estimated that the project change level was 3.11 and the correction coefficient 0.30. The process change level was 2.06 and the estimate was done without process correction. The change from Microsoft technology to Java-technology was added to the technology risk factor. The team level was 3.19 and the team effect coefficient 0.25. Disagreement about the goal and the methods was seen as the most important factor increasing the teamwork. The risks related to Business Object Logic (1 to 3 person months), tools (few person days) and the customers' inexperience in computing were estimated separately. As adding Business Object Logic was a considerably large effort, the total risk effect was considered to be 2 person months (300 hours).

The project manager estimated the productivity differences and the skill and motivation levels of the team members (MK, PH, JJ, JK). The shares of skill and motivation were estimated to be the same, so both coefficients were (MK, PH, JJ, JK) 0.25, 0.44, 0.44 and 1.38. In this case, the correspondence between the levels in the estimation forms and the estimated productivity was weak because the intuitive productivity assessments of the project manager were volatile. As the tasks had not been allocated, the calculation used the average coefficient of 0.625 (average of 0.25, 0.44, 0.44 and 1.38) for both the skill and motivation.

The sum of the correction coefficients was 3.28, which is the sum of one and the correction coefficients of 0.30, 0.25, 300 hours (0.48), 0.625 and 0.625 (Equation 7). The estimated effort was 2644 hours, which included 23% for design and transition effort in addition to the implementation's 2036 hours. The phase coefficient of the implementation phase (h) was 0.77. The estimate is based on the risks and lower productivity than in the earlier project.

It was decided that the project should be partitioned into tasks and staff should be assigned to the tasks. Each task should produce a tangible result, which had countable components and the effort of a task should be less than 10 person days. The personal productivity differences could only be estimated when the tasks had been assigned. The second estimate was done on 15.1.2001 and the implementation effort was estimated to be 352.5 hours. The estimate for the whole project was 458 hours (=352.5 h /0.77). The main reasons for the decrease in the estimate were a better component structure and better estimates of the efforts of the components.

### 4.15.2 A-System Quick UI recalculation

The recalculation of Quick UI was done on 13.2.2001 and the missing numbers were supplemented in the following week via e-mail. The meeting took about 3 hours. The project manager gave a view of the actual hour data of the project. The CRM estimation forms for estimation of the corrective factors were filled in again and the original answers were available for discussion. The actual recalculation was done after the meeting when all of the data were available.

The same component structure was used in the recalculation as was used in the making of the original estimate. "Button"-components were classified into three types, because the effort of the server components behind the buttons varied largely. The "search results" component only included the reuse of the server component, not its construction as in the previous project. Therefore the data from 5 windows and 47 components were available to use in calculating the component-based efforts (see Table 21, page 91). The calculation of the efforts of the components was done manually by adjusting the solution in the calculation-sheet to minimise the difference of efforts in the actual and recalculated tasks. In particular, the difference of the actual effort of the project and the recalculated effort of the project was minimised.

The pure component-based effort was 275 h (=5*4+ 26*1+ 5*30+ 2*15+ 4*3,5+ 1*3+ 4*8). The addition due to skill effect was estimated to be 50% of the baseline effort (137.5 h) and the motivation effect was estimated as 30% of the baseline effort (82.5 h). The actualised risks (technical problems) were estimated to be 151 h (55% of the baseline effort). The effort of the project manager was included in the team effect and this came to 107.5 h (39% of the baseline effort). The effort of the design, integration testing and transition was 143 h (15.8% of the total effort). Installation work on the demo computer, 70 h, was seen as a part of the transition phase. The familiarisation of JK was removed from this project. This concludes the sharing of the total effort of 904 hours.

The process level in the recalculation was 3.0 and the project manager did not see the need for process correction either. The largest change was the change from Microsoft technology to Java technology. Its impact has been included in risk and skill effects. The differences between the component based efforts of the Web UI and the Quick UI projects were considered to be caused by random effects, inadequate consideration of the server components and missing checks in the text fields.

The project change level in the recalculation was 2.39. The most important factors were the views of the project members, technological surprises and change control. The number of components changed during the project. The baseline effort that was calculated using the original component structure was 267.5 h, which was 7.5 h (2.7%) less than the actual baseline effort.

The team level in the recalculation was 2.71. The most important factors to add to communications were estimated to be the meetings and the reading of minutes and other documents. Teamwork includes the meetings and the work of the project manager, which comes to a total of 107.5 h (38.5% of the baseline effort). One reason for the larger teamwork effort than estimated was the change of the project manager, which was not specified as a risk factor.

The changes in the staffing and unexpectedly difficult programming errors were seen as the most important risk factors. The impact of the atmosphere was included in the motivation effect. In addition, a part of the effort of the team members was recorded to other projects. The problem of the server component "download" was itemised as its own risk task, which took 151 hours. The risk effect was 55% of the baseline effort calculated as explained.

The project manager, MK, handed his notice in during the project but he was active in the project until its final stages. A large part of JK's effort was recorded to a separate familiarisation project. On almost every occasion a single person implemented each task. As the new project manager could not assess the personal productivity differences, he estimated that the skill effect addition was 50% and the motivation effect addition was 30% for all the team members.

### 4.15.3 Comparison of the actual and the estimates

The Quick UI project was estimated twice. The first estimate was done at the beginning, another in the middle of the project and finally it was recalculated after the project. The first estimate, which was made based on the Web UI on 4.12.2000, was considered inaccurate as soon as it was done because the decomposition of the components was clearly missing. A new estimation meeting was planned for when the implementation design had progressed a little more. It was held on 15.1.2001. The estimate of the total effort of the project was 2644 hours on 4.12.2000 and 458 hours on 15.1.2001. Based on the latter, originally task based estimate, the component-based estimates were also calculated.

Table 20 summarises the calculation of the estimates. It includes the estimates of the project manager on17.11.2000, 4.12.2000, 15.1.2001 and the recalculation on 13.2.2001. The project manager had estimated the factors using the CRM estimation form on 17.11.2000, 4.12.2000 and 13.2.2001 and the level of the effect is calculated based on that judgement. The levels have been compared to the project manager's estimate of the actual effort. The estimate on 17.11.2000 concerns the Web UI project and the estimate on 15.2.2001 did not include estimation forms/levels. It should be noticed that

- The results are not statistically significant because the amount of data is small.
- The estimation form was not used in risk estimation
- The same skill and motivation factors were used for each of the persons
- A large part of JK's effort has been removed to another project. This decreases the realised effort considerably.
- Other work, such as general design and integration testing, has not been included in the bottom line
- The older calculation rule was used. The sum of the corrections was used instead of the product. The last column contains the values calculated by

the newer equation, (Equation 8, page 30), which was developed as a result of this case study.

Table 20. The change of the estimates during the project.

| | Earlier project | | First estimate | | 2nd | Recalculation | | |
|---|---|---|---|---|---|---|---|---|
| Correction factors | 17.11 level | 17.11 coeff. | 4.12 level | 4.12 coeff. | 15.1 coeff. | 13.2 level | 13.2 coeff. Eq 14 | 13.2 coeff. Eq 4 |
| Baseline (h) | | | | 620 | 152 | | 275 | 275 |
| Process | 2.21 | 0.00 | 2.06 | 0.00 | 0.00 | 3.00 | 0.00 | 1.00 |
| Project change | 3.61 | 0.50 | 3.11 | 0.30 | 0.05 | 2.39 | 0.03 | 1.03 |
| Risks | - | 0.12 | - | 0.48 | 0.15 | - | 0.55 | 1.53 |
| Team | 2.49 | 0.14 | 3.19 | 0.25 | 0.10 | 2.71 | 0.39 | 1.25 |
| Skill avg. | - | 0.30 | - | 0.625 | 0.50 | - | 0.50 | 1.25 |
| Motivation avg. | - | 0.20 | - | 0.625 | 0.30 | - | 0.30 | 1.12 |
| Corrections | | 2.26 | - | 3.28 | 2.10 | - | 2.77 | 2.77 |
| Phase | - | 0.77 | - | 0.77 | 0.77 | - | 0.84 | 0.84 |
| Estimate (hours) | | | | **2644** | **458** | | **904** | **904** |

Table 21 describes the progress of the component structure. It includes the estimates of the project manager on 4.12.2000, 15.1.2001 and 13.2.2001. The baseline effort is calculated using a new estimate of the numbers of the components and a new estimate of the effort of each component. As it was known on 4.12.2000 that the server part of the search results component would not be constructed again, certainly not several times, the 620 h does not genuinely represent the view of the estimators. The large efforts of some of the components should have been itemised, but it was not possible to go into detailed design in the estimation meeting. Due to the project being small, a random error in a component-based effort is significant.

Table 21. The change of the estimates during the project.

| Component counts and estimated efforts(h) | Combo box | Text Field | Difficult button | Normal button | Easy button | Link | Search results | Baseline effort | Effort estimate |
|---|---|---|---|---|---|---|---|---|---|
| Counts 4.12.2000 | 12 | 8 | 3 | 6 | 5 | 2 | 1 | 37 | |
| Counts 15.1.2001 | 4 | 29 | 6 | 1 | 3 | 3 | 8 | 54 | |
| Counts 13.2.2001 | 5 | 26 | 5 | 2 | 4 | 1 | 4 | 47 | |
| Effort 4.12.2001 | 4 | 4 | 53 | 30 | 4 | 4 | 173 | **620** | 2644 |
| Effort 15.1.2001 | 3 | 0.8 | 13 | 8 | 2 | 3 | 2 | 152 | 458 |
| Effort 13.2.2001 | 4 | 1 | 30 | 15 | 3,5 | 3 | 8 | 275 | 904 |

Table 22 describes the changes of the estimates of the human effects. Notice the influence of JK in the estimates.

Table 22. The change of the estimates of human effects.

| | Earlier project | | First estimate | | 2nd | Recalculation | |
|---|---|---|---|---|---|---|---|
| Correction factors | 17.11 level | 17.11 coeff. | 4.12 level | 4.12 coeff. | 15.1 coeff. | 13.2 level | 13.2 coeff. |
| Skill avg. | - | 0.30 | - | 0.625 | 0.50 | - | 0.50 |
| Motivation avg. | - | 0.20 | - | 0.625 | 0.30 | - | 0.30 |
| Skill MK | 4.69 | 0.30 | 3.41 | 0.25 | - | 2.97 | 0.50 |
| Motivation MK | 3.61 | 0.20 | 3.95 | 0.25 | - | 3.69 | 0.30 |
| Skill PH | 4.06 | 0.40 | 3.59 | 0.44 | - | 3.16 | 0.50 |
| Motivation PH | 3.64 | 0.20 | 4.02 | 0.44 | - | 4.02 | 0.30 |
| Skill JK | | | 3.34 | **1.38** | - | 3.34 | 0.50 |
| Motivation JK | | | 3.30 | **1.38** | - | 3.30 | 0.30 |
| Skill JJ | 3.81 | 0.30 | 3.34 | 0.44 | - | 3.34 | 0.50 |
| Motivation JJ | 3.68 | 0.10 | 3.80 | 0.44 | - | 3.80 | 0.30 |
| Sum of human corrections | | 0.50 | - | 1.25 | 0.8 | - | 0.8 |
| Estimate (hours) | | | | 2644 | 458 | | 904 |

In conclusion, on 4.12.2000 both the baseline effort and the correction coefficients were overestimated whereas the estimates were too small on 15.1.2001.

### 4.15.4 Discussion about the CRM estimation in the test project

As at the beginning, the history data, which is the base of the CRM estimation, was not adequate, the CRM parameters were estimated manually. To get results in a reasonable time, small projects (Web UI and Quick UI) were selected with a small amount of data. Unfortunately, random effects were significant.

The estimates could be done with reasonable effort despite the lack of history data and stiffness at the beginning. The accuracy of the estimate was comparable with the task based estimates, but the use of CRM reveals the risk factors of the estimate better than direct judgement of the tasks. It was possible to make the CRM calculations using an Excel-sheet, but a better tool would have made the estimation during a meeting feasible.

The project tracking was based on the phase model of analysis, design, implementation and testing. As the tasks in the project plan only partially matched the component structure, the numbers and efforts of the components of the Web UI-product were estimated after the project. The user interface components were found easily, but it was more difficult to take the components of the lower levels of the architecture into account. It was obvious that the largest component ("search results") should be decomposed. A-System uses Rational's Unified Process as its quality system. In practice the model was more like cascading waterfalls. As the analysis was done "purely", without a connection to the component structure, no clear component structure was available after the end of the first phase of the Quick UI project on 4.12.2000. The design and the implementation were done in the same project, and on 15.1.2001 a component structure without server components and their efforts was available. The objective of itemising the product into small components in the project plan and time reporting was not fully achieved because the practices of the company needed to be changed.

As the used traditional phase model left the design of the implementation late in the project, the component structure was not available early enough to make a bid to the customer to carry out the project. The user interface components were found easily, but the components, which were at the lower levels of the architecture, were taken into account by classifying the user interface components. In addition, the splitting of the larger components would have been very useful.

The component structure of the finished product was in the documentation and the code but the efforts of the components were estimated afterwards because the original records did not exist. Traditional project plans divide the projects into phases and additionally the implementation phase is divided into tasks that implement tracking sets. In A-System the decomposition was worse than this and the connections between the tasks and the components were obscure in the project tracking.

The environment of the projects was turbulent. Both the technology and the staff were in a change process. The corporation, A-Systems, was sold on 18.12.2000. The technology risk actualised in both of the projects resulted in a large effort being spent on a single difficult problem. The estimated skill and motivation correction coefficients were considerably high, but they were based on the estimate of the situation. As Quick UI did not contain similar personal tasks, the productivity differences of the staff could not be verified. Even the original view of the project manager was that JK's participation in the project was not productive. As the CRM method was new to all of the project members, reluctance in making the estimate and tracking the project was natural. The speed at which the method was learnt was fast. The change of the calculation rules of CRM (the use of the product of correction coefficients instead of the sum) were not used because it was important to have a unique base for the calculation during the period of the whole study.

As the total effort is the sum of the component-based efforts multiplied by corresponding correction coefficients, using a larger correction coefficient can compensate for a component-based effort that is too small. The relationships between the correction coefficients and the baseline effort could not be analysed because the amount of data was very small.

The mathematical calculation of the efforts of the components (see Equation 12, page 33) was unsuccessful because the "least squares" method, which calculates the best fit of the data, gave negative values even though the component-based efforts are always non-negative. So, in the recalculation, component-based efforts were manually solved from the actual data by using a Microsoft Excel spreadsheet, which contained a table like Table 13 (page 59).

The correlation between the level of the effects in the estimation forms and the separately estimated correction coefficients was weak. The questions still produced ambiguity and they need to be developed. The weighted average of the levels was between 2-4, though the views of the project manager would have required more extreme numbers. The decomposition of skill and motivation was thought to be difficult. The estimation of the risks proved to be better if is done analogously with the other estimates.

In the case study, the older equation was used (Equation 7). As the correction factors are overlapping, it was difficult to view each of them separately. For example, the relative team work effort of the project manager would remain the same, though the project would last longer due to additional work of, e.g., risk effect, if the baseline effort were not changed. The same also applies to the other CRM effects. Each estimation form estimates only one effect and it would be difficult to take the other effects into account in each of them. The multiplication of the correction coefficients solves the problem.

# 5 CRM Compared to Other Software Metrics

## 5.1 Introduction

In this chapter measures which are used widely and are suited to component-based development are presented briefly and compared to CRM.

The description of each method includes a brief evaluation, which emphasises its advantages and disadvantages. The final subsection summarises these observations and makes more explicit comparisons with Component Reuse Metrics.

## 5.2 Lines of Code (LOC)

The number of lines of code is easy to collect and they have been collected in history over the last 20 years at least. One example would be in the US Defence industry [Poulin 1997, page 55]. The defence connection also connects it to research and development organisations in the USA, such as The Software Engineering Institute (SEI).

LOC is an internal measure of a software component. As its calculation uses produced code it is not available during the analysis phase, the time when it would be needed the most.

The meaning of LOC is intuitively clear but it needs a clear and concise definition. A standard line of code, *SLOC* is defined [Conte 1986] so that empty lines, comments and style differences are excluded. In refined LOC measurements of different categories of lines are counted separately. These categories should include as a minimum reuse [Humphrey 1995] and the application type [McConnell 1996]. The complexity of the control flow, programming style, cohesion and coupling are still excluded. The collection of the LOC-counts and effort estimation is not as easy today as it was in 1980 because of the emergence of integrated programming environments, visual programming and program generators. Assessing productivity by a simple "produced LOC per hour"-metric encourages a verbose programming style instead of a concise and reuse oriented style. The calculation of the estimate of the effort also requires an estimate of the productivity. As simple LOC per hour is rarely an adequate productivity metric, more sophisticated methods, such as COCOMO, have been developed.

## 5.3 COCOMO

The Constructive Cost Model, COCOMO, [Boehm 1981] calculates the effort required for software development using the size of the product in LOC. There are three different versions of COCOMO: basic, intermediate and detailed. In the basic version the equation is

Equation 18. $LM = \alpha \cdot \left( \frac{LOC}{1000} \right)^{\beta}$ , where

- $LM$ = effort in labour-months,
- $\alpha$ = complexity coefficient,
- $\beta$ = complexity exponent and
- $LOC$= initial estimate of delivered source instructions (= lines of code, LOC).

The complexity coefficient and exponent depend on the difficulty of the project. There are easy (organic), normal (semi-detached) and difficult (embedded) systems and their equations are $2.4*(LOC/1000)^{1.05}$, $3.0*(LOC/1000)^{1.12}$ and $3.6*(LOC/1000)^{1.20}$ , respectively [Boehm 1981]. In the intermediate version, there are coefficients for [Boehm 1981]:

- reliability requirements,
- size of the database,
- complexity of the product,
- response time requirements,
- restrictions in memory use,
- maturity level of the development environment,
- turnover-time of the development environment,
- skills of the developers,
- familiarity with the business area,
- skills of the programmers,
- familiarity with the development environment,
- familiarity with the programming language,
- use of modern development methods,
- use of development tools and
- tightness of the schedule.

Each of the coefficients is judged according to a 6-level scale (very low, low, normal, high, very high and extra high) giving values from 0.7 to 1.66. The numbers are then multiplied by each other and with the result of the basic COCOMO-equation (see Equation 18). In COCOMO II, cost drivers are used to capture the previously mentioned characteristics of the software development that affect the effort required to complete the project [Boehm 2000].

Equation 19. $LM = A \cdot \left( \frac{LOC}{1000} \right)^{E} \cdot \prod_{i=1}^{n} EM_i$ , where

- LM = effort in labour-months,
- $A$= 2.94 (for COCOMO II.2000),
- $EM_i$ = effort multiplier,
- $E$ = scale exponent and
- $LOC$= initial estimate of delivered source instructions (= lines of code, LOC).

The equation resembles the CRM equation (see Equation 8, page 30). The effort multipliers are basically the same as CRM effects but in COCOMO the efforts corresponding to the effects are not measured individually. An exponential equation suits curve fitting well, but the impacts of the changes of the parameters are difficult to illustrate. For example, the effort change due to use of a more skilled developer is straightforward in multiplication-based CRM compared to COCOMO. The scale factors are needed because COCOMO estimates projects, not tasks as CRM does.

The COCOMO-model has been extended to take reuse into account [Balda 1990]. In this equation the LOC of unique code developed, code developed for reuse, reused code and code for modified components are calculated separately. The results of an experiment state that it takes 20 times more effort to build software for reuse than it does to reuse it. Using this relationship the equation can be further simplified.

COCOMO was extended later [Boehm 2000] to adapt to situations that exist in modern software development. COCOMO II contains variations for application composition, phase scheduling, rapid application development, commercial-off-the-shelf product integration, quality estimation, productivity estimation and risk assessment. The application composition model is interesting from CRM's point of view, because it estimates projects which are developed using the composition of components. The lines of code have been replaced by application points, which count and classify screens and reports from the forthcoming application. Reuse is calculated estimating the reuse percentage.

The disadvantage of COCOMO is its connection to lines of code. Getting the LOC count into the equation early enough is a problem which COCOMO II solves by supporting different equations and sizing methods (such as function points).

The reuse variation in the effort can be taken into account by extending the equation. Hence, there are a lot of considerable variations which have a separate equation in COCOMO II. This makes the method quite complex.

The model includes a number of fixed parameters that have been derived statistically from available finished projects. The similarity of these projects and the projects which are to be estimated is then assumed.

COCOMO illustrates the relationship between lines of code and the effort: it is exponential rather than linear (Equation 18, page 96), though the exponents are small. This conforms to the intuitive and experimental view of this area: if LOC counts are equal, it is more difficult to produce one large program than several small ones. That is true even when there is one developer in the project. When more developers are added the effort grows exponentially [Brooks 1995].

## 5.4 Function points

Function point analysis, FPA, is a widely referred estimation method. It is a synthetic measure of a program size that is often used in the early phases of a project. The unadjusted function point count, UAFPA, is

Equation 20. $UAFPA = \sum_i \sum_j w_{ij} \cdot F_{ij}$ , where

- $UAFPA$ is the number of unadjusted function points,
- $w_{ij}$ is a complexity weight and
- $F_{ij}$ is the number of functions *ij* in the system.

Functions are classified into types such as inputs, outputs, inquiries, logical files and external files. Each function type has its own complexity weight for low, medium, and high complexity [McConnell 1996, Dreger 1992].

After the basic calculation is done, the result is multiplied by an adjustment multiplier, which takes into account, for example, complex processing, reusability, performance objectives and operational easiness. A value from 0 to 5 is given according to how much influence that factor has on the product. The sum of these is used in adjusting the original function point count. The adjusted function point total is

Equation 21. $AFPA = UAFPA \cdot \left( 0.65 + \frac{\sum f}{100} \right)$, where

- AFPA is the adjusted function point total and
- f is an influence factor.

The form of FPA equations resembles the CRM equation (see Equation 8, page 30). Though FPA uses its own functions and CRM is based on component and work breakdown structures, the functions in FPA can have corresponding components in CRM. The FPA weights of the functions are fixed and in CRM the efforts of the components are measured. CRM is not limited to a fixed set of functions because CRM can measure all the different kinds of components. The FPA influence factors adjust the size of the software while CRM includes these in the estimates of the components. CRM focuses on the productivity, which FPA ignores.

Making function point analysis is based on strict rules which guide decisions, for example, about what is an inquiry with low complexity, output with high complexity or a strong factor of reusability. The success of implementing function point counting has been considered poor without the aid of a trained function points counter. This is because counting is not easily repeatable or independent of the estimator who interprets the rules. To make consistent counts you should have one person who does a lot of function points counting as opposed to trying to make each member of the team a part-time function points counter. Computer support for function point calculations exists [Rask 1992].

Humphrey [Humphrey 1995] does not consider function points fully satisfactory because they cannot be measured directly from the code or specifications and because they are not sensitive to implementation decisions. Development costs are typically sensitive to the implementation language, the design style and the application domain. The estimation method should take this into account.

Function points work well with a limited class of problems. Software that "has no user interface or database, but does some complex computations or generates graphic output" does not lend itself well to function point counting [Keuffel 1994]. FPA was originally developed for large mainframe MIS applications, and it needs some adaptation before use in object-oriented software development [Goldberg 1995, Graham 1994]. *Use case points* are a way of making the adaptation of function points to object-oriented paradigm. From preliminary applications in web-based projects, it has been conjectured that this could in fact be more reliable than FPA. The problem here is that the waterfall process model must be used and use cases must be available right at the requirements gathering phase. [Nageswaran 2001]. COSMIC-FFP is a new FPA variation, which is targeted at early estimation [Meli 2000]. Functional User Requirements are represented by a set of functional processes, each of which is a unique and ordered set of data movement sub-processes. The unit of measurement is 1 data movement, referred to as 1 COSMIC Functional Size Unit, e.g. 1 $C_{FSU}$. The users' functional requirements are mapped into the generic COSMIC-FFP software model, which simplifies the functionality to entries, exits, reads and writes, which are then counted. This model has been developed for various levels of functional abstraction, such as software layers, functional processes and data movement sub-processes.

The greatest value of function point metrics is in its basic view. It is based on the external view of the product - it counts what a user gets and sees. It can therefore be used in assessing productivity, utilising the external view contrary to lines of code or other internal measures. While implementation independence is an advantage in cross-language or cross-system comparisons, it is a disadvantage in development cost estimates [Humphrey 1995].

## 5.5 PROBE

As estimates are required before the development can begin and at a time when little is known for sure about the product itself, the estimation methods must use data from previously developed similar products. This suggests a generalised estimation process [Humphrey 1995]. The first step is to make the conceptual sketch of the new product and divide the product into parts. Then a resembling part is looked for from a database which contains historical data about the size of the historical parts. After that the estimates for the sizes of the parts picked from the history database are summed up. Finally, the estimates of the total size and the total effort are calculated.

It is difficult to directly judge how many LOC it will take to meet the requirements. The need is for a proxy which relates a known product size to the functions that the estimator can visualise and describe. Examples of proxies are objects, screens, files, scripts or function points. The development effort required for the proxy must have a demonstrably close relationship to the effort required for developing the target of the estimation. As historical data are needed for making the estimate, it is desirable to have large amounts of data. This means that the data needs to have the facility to be automatically collected. This is possible if the proxy is a physical, precisely defined entity. The proxy is used to denote the size of the new product and it should be easy to visualise at the requirement capture phase. The proxy should be customisable to the needs of the using organisation so that differences in resource usage and the products can be taken into account. This also means that the proxy is sensitive to the implementation variations.

Many potential types of proxies exist. In large scale, it is possible to use other projects as proxies. By classifying them by size a rough estimate can be obtained. This is called the fuzzy-logic method [Putnam 1992]. In object-oriented analysis application entities, business objects, and the system's main functions and use cases are early visible. This would suggest their usefulness in being used as proxies. For example, in automobile registration, the entities might include automobiles, owners, registrations, titles or insurance policies.

The PROxy-Based-Estimating (PROBE) method uses classes as proxies of lines of code. Here a C++ class is an example of a proxy. Proxies are named and categorised in the conceptual design. They are categorised by reuse category and type. The reuse categories are base product, base addition, new object and reused object. The base product is the base program which is to be enhanced. Its size is given in LOC. Added, modified and deleted LOCs in the base program will be estimated. New proxies (objects) and reused proxies are their own categories. LOCs from reused proxies and the base program can be taken directly from the database. These LOCs are used in a separate procedure which evaluates the reuse effort. In estimating new proxies, the type of proxy, judgement of its relative size and the number of methods are needed. The type classification is application area and implementation method dependent. The relative size categories are very small, small, medium, large and very large. Their sizes are calculated using normal distribution such that 6.68 % of the methods are very small, 24.17 % of the methods are small, 38.4 % of the methods are medium, 24.17 % of the methods are large and 6.68 % of the methods are very large.

Table 23 shows as an example a history database which contains the method sizes in LOC according to these categories.

Table 23. C++ proxy categories in LOC per method [Humphrey 1995].

| Category | Very small | Small | Medium | Large | Very large |
|---|---|---|---|---|---|
| Calculation | 2.34 | 5.13 | 11.25 | 24.66 | 54.04 |
| Data | 2.60 | 4.79 | 8.84 | 16.31 | 30.09 |
| I/O | 9.01 | 12.06 | 16.15 | 21.62 | 28.93 |
| Logic | 7.55 | 10.98 | 15.98 | 23.25 | 33.83 |
| Set-up | 3.88 | 5.04 | 6.56 | 8.53 | 11.09 |
| Text | 3.75 | 8.00 | 17.07 | 36.41 | 77.66 |

Notice that a method consisting of 12 lines of code is considered very large if it is a set-up method and small if it is an I/O-method. Thus proxy category classification gives a better match to an intuitive definition of relative size. Now, using a table lookup, an estimate for a new medium size data class in C++, which consists of 8 methods, is 8 * 8.84 LOC = 114.9 LOC.

The PROBE method uses linear regression in order to take into account the fact that programs are larger when finished than when originally estimated. Using linear regression is appropriate because the estimated LOC and actual total program LOC are correlated [Humphrey 1995]. The estimated new and changed LOC (N) can now be calculated [Humphrey 1995] as

Equation 22. $\mathrm{N} = \beta 0 + \beta 1 \cdot (\mathrm{BPA} + \mathrm{NO} + \mathrm{MO})$, where

- N = estimated new and changed LOC,

- β0 and β1 = regression coefficients (calculated using historical data),
- BPA= base product additions LOC,
- NO= new LOC and
- MO= modified LOC.

The use of regression also gives a measure of the accuracy of the estimate. The regression calculation can also produce a prediction interval of the estimate.

The development time estimate can be obtained from the historical data by

1. regression calculation using estimated LOC and total actual development hours in previous projects,
2. regression calculation using actual LOC and total development hours or
3. using historical productivity in LOC per hour.

The first choice is the best. If enough data is not available the second or third choice must be used.

Previously only new and modified objects were shown in calculations. In practice it is not easy to collect the time usage of new objects, modified objects and reused objects separately. The effort estimate can be calculated using multiple regression coefficients for equation [Humphrey 1995]:

Equation 23. $\mathrm{E} = \beta 0 + \beta 1 \cdot NO + \beta 2 \cdot RO + \beta 3 \cdot MO$, where

- E = estimated effort (hours),
- β0, β1, β2, β3= regression coefficients,
- RO= reused LOC
- NO= new LOC and
- MO= modified LOC.

The disadvantage of PROBE is that it is based on lines of code. Not all lines of code require the same amount of work. That is partially taken into account in the multiple regression formula. The advantages of using hours directly are also admitted. The idea of object proxy is used to estimate the number of lines of code. Objects are not seen as parts of what the user gets from the system. The PROBE method is thus more an internal measure than an external measure that could be used in estimating productivity. Calculations needed, especially in multiple regression, benefit from using a proper tool because they are so long and tedious to perform.

The advantages of PROBE are to be found in its ideas of proxies, proxy categories and statistical calculations. Studying the prerequisites for statistical analysis prevents using the equations blindly in situations where previously developed programs do not resemble the one under estimation. PROBE does not have intrinsic weights in the method itself as function point analysis does. Classifying tables are calculated using the data from previous product development projects. This prevents comparing the results world-wide but it may lead to more reliable results in the organisations using it.

## 5.6 Classic complexity metrics

The term complexity is typically used in studying psychological or computational complexity of programs. Lines of code are not considered equal, some of them are more difficult to produce, modify and understand. That difficulty is correlated to the effort required in the development. The goal is to develop a complexity metric that can be defined unambiguously using properties of the developed code that has a good correlation to the effort.

The complexity can be divided into a large number of factors. The main division is separation of inter-module and intra-module complexity. The intra-module metrics consist of procedural complexity and semantic complexity. *Procedural complexity* consists of style metrics, size metrics, data structure metrics, control flow metrics, and internal cohesion metrics [Henderson-Sellers 1996]. Semantic complexity is measured by discerning semantic cohesion. In object-oriented programming s*emantic cohesion* evaluates whether an individual class is really an abstract data type in the sense of being complete [Henderson-Sellers 1996, pages 56 and 119].

A simple measure of inter-module coupling is the fan-in/fan-out metric [Henry 1981]. Informational *fan-in* refers to the number of locations from which control is passed into the module (for example, calls to the module being studied) and the number of global data [Henderson-Sellers 1996]. *Fan-out* measures the number of other modules required plus the number of data structures that are updated by the module being studied [Henderson-Sellers 1996].

These metrics have been widely utilised and were thoroughly tested by their developers. These metrics show directly that encapsulated modules which have a small fan-in and fan-out are less complex than modules which are not encapsulated.

According to Halstead's Software Science metrics [Halstead 1977], counts of operators and operands and their sums can be used to estimate the size of programs or algorithms. A *token* is an operator or an operand in a programming language statement. The idea behind counting tokens instead of lines of code is that a line which contains more tokens is more complex than a line with fewer tokens.

The use of the graph theory in module metrics for procedural complexity has also been very popular. McCabe's [McCabe 1976] *cyclomatic complexity* metric studies a program's control flow graphs as a measure of its complexity.

It is “a metric based on graph theory that reflects the total number of paths through a software module” [Poulin 1997, McCabe 1976].

Empirically LOC (see page 95) has been shown to be at least as good as Halstead’s metrics and McCabe’s cyclomatic complexity metric [Henderson-Sellers 1996]. The usability of McCabe’s metrics is not good in object-oriented programming because the size and cyclomatic complexity of the methods can be rather low, even though the complexity of the class or a product is high [Kolewe 1993]. In object-oriented programming, for example in C++, much of the explicit branching statements, that is if, while and case statements, have been replaced by implicit branching due to inheritance and event-driven programming. This means that cyclomatic complexity alone cannot explain the complexity of an object-oriented program.

The major disadvantage of these approaches is that they rely on the code, which does not exist at the beginning of the project.

## 5.7 Object-oriented software metrics

The emergence of object-oriented programming led to attempts to find new ways to comprehend the produced code. Putkonen [Putkonen 1994] presented a suite for object-oriented software metrics as an extension to the suite developed by Chidamber et al. [Chidamber 1991]:

4. *Weighted Methods Per Class* is defined as a sum of the static complexities of each method in the class. It measures the size and the difficulty in order to understand the methods of the class.
5. *Depth of Inheritance Tree* is defined as a total number of ancestors of a given class. Every ancestor class must be understood in order to use a method in a particular class since a method may be defined (or redefined) in many classes in an inheritance tree.
6. *Number of children* is defined as the number of immediate subclasses of the class.
7. *Class Coupling* is defined as the number of use and association relationships with the class and other classes. It measures the number of classes we need to understand in order to use a particular class.
8. *Response for a Class* is defined as a sum of the number of all the methods of the class and the number of all the methods of other classes called by the methods of the particular class.
9. *Lack of Method Cohesion* is defined as the number of disjoint sets formed by the intersection of the sets of instance variables used by each method of the class. A high value in this metric indicates possible under-abstraction where a class should be split into a number of more cohesive classes.
10. *Weighted Attributes Per Class* is defined as the number of attributes of the class, weighted by the number of parts of aggregate attributes.

These metrics are targeted at evaluating the complexity of an object-oriented program. They can be calculated objectively if the program has already been produced. Contrary to lines of code and other classic measures, the calculation can also begin after the design phase when the classes and their relationships are designed. Chidamber and Kemerer admit that their work is based on theory, not on experience, and invite the reader to validate and expand upon their work. For the most part, the measurements they suggest can be collected automatically. Unfortunately, they present no cause-and-effect relationship data to tie these metrics into attributes of greater practical interest, such as maintainability or reusability [Keuffel 1995].

Lorenz and Kidd propose another suite of object-oriented software metrics [Lorentz 1994]. Table 24 presents the project metrics that contain measures of application and staffing size and scheduling.

Table 24. Project metrics [Lorentz 1994].

| Metric | Usage | | |
|---|---|---|---|
| | Estimating | Scheduling | Staffing |
| Number of scenario scripts | Applies | | |
| Number of key classes | Recommended | | |
| Number of support classes | Applies | Applies | Applies |
| Number of subsystems | Applies | Applies | |
| Person-days-per-class | Applies | | Recommended |
| Classes per developer | Applies | | Recommended |
| Number of major iterations | | Applies | |
| Number of contracts completed | Recommended | | |

It is useful to look in detail at a few of their features: Firstly, project metrics, which are used for estimating, scheduling and staffing, are process metrics. Secondly, the data are available at the time of using the metric. Thirdly, it takes an iterative development model into account. Keuffel writes about the disadvantages of these metrics: “Lorenz and Kidd offer so many measurements (they provide eight project metrics and 30 design metrics) that you hardly know where to begin. In some regards, this represents the shotgun approach to measurement, in which you load up as many metrics as will fit in the gun and blast away, hoping that at least one of the measurements will hit something interesting” [Keuffel 1995].

## 5.8 Task based estimation

Project design and management tools such as Microsoft Project support a very simple but efficient, practical and reasonable way to make estimates. The project is broken down into tasks, forming a structure called project breakdown structure or work breakdown structure [Cantor 1998]. All the work in the project is included in its tasks. In the project breakdown structure each task can have efforts and tasks can again be broken down into lower level tasks. An effort can be a part of only one upper level task so that a project breakdown structure is a hierarchy. The work breakdown structure is not a product structure. If a component is used several times in the product, it is included once in the creation task of the component. In the assembly tasks it is included each time it is used. The resources will be assigned to each bottom level task. Resources, which are typically people doing the job and also equipment or any item which is necessary to accomplish the task, can be taken into account. Resources require time, money and hardware to accomplish the task. Several resources can be assigned to each task. For example, several programmers can be assigned to the same programming task. Each programmer does a certain amount of work. The sum of these is the total effort needed for the project. Figure 23 depicts the class diagram of simple project management.

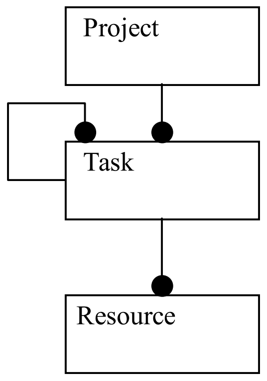


Figure 23. Class diagram of simple project management.

Table 25. Task based estimation.

| Level | Task name | Effort ( h ) |
|---|---|---|
| 1 | Analysis | |
| 1.1 | Definition of the use cases | |
| 1.1.1 | Definition of the use case: customer identification | 7 |
| 1.1.2 | Definition of the use cases for customer accounts | 12 |
| 1.1.3 | Definition of the use cases for customer addresses | 8 |
| 1.1.4 | Definition of the use cases for administrative information | 15 |
| 1.2 | Definition of the class diagram | |
| 1.2.1 | Definition of the class diagram of customer identification | 12 |
| 1.2.2 | Definition of the class diagram of customer accounts | 8 |
| 1.2.3 | Definition of the class diagram of customer addresses | 13 |
| 1.2.4 | Definition of the class diagram of administrative info | 15 |
| 1.3 | Quality assurance | 25 |
| **SUM** | | **115** |

Table 25 shows a simple example of a task based estimation table. It is used to calculate the work required in making a part of an analysis phase of a simple customer information system. The system consists of four use cases, customer identification, customer accounts, customer addresses and administrative information such as user privileges. The resource level is summed up. Project management tools can automatically sum up thousands of tasks and evaluate several schedule and staffing alternatives.

This estimation method is subjective because it relies on the expert estimator. There are several factors that should be taken into account [Metzger 1996]: these include the experience of the developers and the customers in the project area and technical difficulties such as distributed development and distributed installations. This method is the most accurate project estimate available from about mid-design time to the rest of the project [Symons 1991]. There are several variations of this method. Sometimes minimum and maximum and probable values are estimated. There can be one or several estimators. If the developers who accomplish the task also make the estimate, a commitment to a promised estimate and schedule can be obtained. Discussions about the reasoning of the effort of each task are important because they can reveal the flaws in individual estimates. Finally, initial estimates are put together.

Theoretically, mistakes in this method are due to wrong estimates of known tasks or to missing tasks. The errors can sometimes be avoided if the estimator has adequate design and knows the goal of the project very well. If the effort of a task is smaller than 40 hours, it is easier to visualise and thus to estimate. Missing tasks may be the result of errors during the estimation or tasks that became desirable after some amount of work has already been done. Controlling the emergence of new features is one of the most important tasks of project management. It is a common practice to add dummy tasks into the project plan for new unanticipated features, which will be added to the outcome of the project (see Table 37, page 170).

An advantage of using this estimation method is that additional work is not needed. Most of the work it needs must be done anyway in making the project plan. The project plan enumerates the tasks and assigns resources and timetables to them. Using another estimation method would require additional, unnecessary work, for example, counting the function points.

Task based estimation is prone to an overly optimistic result. As the schedule is shown to the customer there is pressure to bargain over it. This easily leads to the removal of politically undesirable tasks. It is not easy to introduce developer training, or to take risks such as sicknesses and staff turnover into account. Similarly, budgeting work to unknown tasks is not usually something that is carried out. There is pressure to minimise the work in order to make the competitive position of a software supplier better or to get the development money from upper management. This has negative effects on the estimate because the estimation method relies on human estimators. The quest for objective ways to make estimates is thus well founded. A solution to the pressure to bargain is to have separate budgets, internal ones for the project team and external ones for the customer.

As there is no explicit connection from the task to the product to be developed, this method cannot be used to estimate productivity. In the implementation phase it is normally implicitly done. For example, the name of the task includes the name of the window to be programmed. In future estimates the expert intuitively uses that window and the effort it required as a proxy to a future similar window. The lack of product information makes the assessments of the project difficult because reorganisations of work are usual in software projects and tasks can change, even if the product remains the same.

As the tasks need to be known before they can be estimated, projects are typically divided into two sub-projects. The definition project defines the product and the implementation project implements it. The estimate of the implementation project is then based on an adequate design of the product. Task based estimation can easily be used in analysis projects. Task based estimation is also well suited to iterative and incremental development because it is very flexible.

Task based estimation is a process metric, not a product metric. Most of the scientific work in the software metrics area is about product metrics. Despite its weaknesses, task based estimation is very popular in practice, (Table 40, page 172).

## 5.9 New metrics research

The trends in software metrics are related to the trends in software engineering because metrics is needed to manage the engineering. This chapter considers the influence of component-based development, commercial off-the-shelf-products (COTS) and extreme programming on software metrics.

The business, engineering, design, infrastructure, management and technological aspects of component-based software engineering have been developed [Heineman 2001]. Poulin's metrics for Software Components are schedule, lines of code per component, labour hours, component classification, costs, change requests and defects [Poulin 2001]. His primary reuse metric is reuse percentage calculated as reused lines of code/ total lines of code. Smith uses intensity, concurrency and fragmentation of the tasks, component project experience, programmer project experience and team size as parameters in effort estimation for component-based software development [Smith 1997]. Later Smith used task assignment patterns to improve the effort estimates [Smith 2001]. Briand and Wust showed that simple size measures, which can be obtained from class diagrams or code, explain most of the effort variance. More sophisticated coupling measures do not bring substantial gains in effort estimation accuracy [Briand 2001].

Measurements are needed to choose between COTS-products for a certain application. The requirements of the application and the study of the candidate COTS-solutions are developed in parallel [Solberg 2001]. The effort of constructing an application based on a COTS-product is the sum of integration efforts of its components required in the final application. This effort is one criterion in selecting the COTS-product. Ochs et. al. propose a Goal-Question-Metrics (GQM) based COTS-assessment and selection method [Ochs 2001]. Quality and risk concerns currently limit the application of COTS-based system design to noncritical applications. Sedigh-Ali proposes the use of software metrics to guide quality and risk management [Sedigh-Ali 2002].

Agile methods such as extreme programming (XP) are emerging software engineering processes which target getting the results earlier and investing later [Beck 2000]. It has short iterations and release cycles. XP always keeps the system in prime condition by continuous integration, writing tests early and running them automatically. Refactoring is a process for improving the design of existing code [Fowler 2000], which is needed to flatten the exponential rise in the cost of changing software over time. The management of extreme programming requires continuous measurements that control cost, time, quality and scope.

Industrial experiences show that cost-benefit analysis and feedback sessions are important in software process improvement programs [Solingen 2001]. There are several tools for automated support for the GQM measurement process [Lavazza 2000], [Komi-Sirviö 2001]. Measurement based continuous assessment of software engineering processes is also feasible and useful [Järvinen 2000]. The same applies to using measurement to optimise the software process at the individual level [Coleman 2000].

General studies on organisational behaviour can be applied to software development processes. However, specific research is more relevant [Robbins 1998]. There are techniques and models for understanding human factors which are not as unpredictable as we would like to think [Oosting 2000]. Reo has used balanced IT scorecards to measure the satisfaction and competence of an organisation's personnel [Reo 2000]. Meli has defined formulas to take the changes in the requirements into account in effort estimates by using functional measurements of Change Requests [Meli 2001]. The identification of risks, analysing them, using statistics to estimate them and seeking improvement potential belong to the risk driven effort estimation method of Schmietendorf et. al. [Schmietendorf 2001]. Toffolon proposes a framework where each development activity is composed of two separately tracked parts: the production task and co-ordination task [Toffolon 2000]. The previous results relating to CRM effects are specific, but do not conflict with the results of this study either.

Fenton's foundations in software measurements are often used [Fenton 1997], but systematic capture of historical data for effort estimation is still a common problem. Quite a lot of accurate, consistent and complete data is needed [Shepperd 2001]. The value of the collected data may also diminish over time due to advances in development technology or organisational changes. It is possible to pool or reuse data across different measurement environments, but sophisticated statistical procedures are still required for data mining. Maxwell presents a data analysis methodology for extracting formulas of development effort [Maxwell 2002].

## *5.10 Summary*

Effort estimation is needed in a feasibility study, calculation of offers, scheduling and process improvement. Lines of code, classic complexity metrics, COCOMO and function points require information that is not available at the time the estimation is accomplished. Their extended versions (for example, the application point model of COCOMO II) are better. Task based estimation, PROBE, some object-oriented metrics and CRM are able to make use of gradual designs of the forthcoming product. This is required in modern iterative process models. Task based estimation and CRM are suited to iterative and incremental development. There are also variations of the other metrics which are targeted to those methods.

Table 26 presents a qualitative comparison of CRM to the basic variants of other software metrics. COCOMO, CRM and Task based estimation are targeted to effort estimation and the others measure the size of software. Only CRM and object oriented measurement suites use multidimensional size. FTP and application points of COCOMO II transform the size measures into a one-dimensional number. The user of the estimation method (estimator) defines the units of size (components) in CRM, PROBE and task based estimation.

Table 26. CRM compared to other software metrics.

| | CRM | LOC | PROBE | FPA | TASK | OBJECT | COCOMO |
|---|---|---|---|---|---|---|---|
| Size dimensions | N | 1 | 1 | 1 | 0/N | N | 1 |
| Size unit definition | User | Rules | User | Rules | User | Rules | Rules |
| Size availability | Design | Impl. | Design | Design | Design | Design | Impl. |
| Project change | Yes | No | Yes | No | No | No | No |
| Team | Yes | No | No | No | Yes | No | No |
| Risk | Yes | No | No | No | Yes | No | No |
| Process | Yes | No | Yes | No | Yes | No | Yes |
| Skill | Yes | No | Yes | Yes | Yes | No | Yes |
| Motivation | Yes | No | No | No | Yes | No | No |
| Incremental & iterative | Yes | No | No | No | Yes | Yes | No |
| Parameters from | History | Method | History | Method | User | Meth. | Method |
| Task assignments | Yes | No | No | No | Yes | No | No |

As task based estimation relies on a subjective expert estimator it is often ignored in computer science. On the other hand, measurement systems that use well-defined size metrics use a rough estimation of the project and human factors. Only task based estimation and CRM explicitly handle the effect of staffing the project. COCOMO has cost drivers for the staff, but handles them at a high level. It is important to estimate individual tasks because productivity and labour cost often vary from person to person. The other methods rely on statistical averages of these factors, often using them as constants of the method. As CRM and PROBE count physical, precisely defined proxies of the product, they avoid a time consuming and subjective mapping phase, which is needed in function point analysis.

The measurement methods depend on the application. Lines of code are not available in commercial-off-the-shelf products and reusable components and the calculation rules of function points are targeted to applications which contain databases and user interfaces. The extensions of the old methods take reuse and other modern techniques (for example, component-based development) into account, but the paradigm has not been changed. For example, COCOMO II has extensions to commercial-off-the-shelf products, rapid development, component-based development, phase scheduling and risk assessment. CRM is planned for component-based development, but it can be adapted to measure traditional development. As project tracking gives CRM a lot of timely feedback, which is used to adjust the estimates of the successive phases of the project, its parameters can measure the circumstances of the project better than the parameters of other methods, which are at worst constants of the method.

The lists of factors of the effort in the estimation methods presented in this study resemble each other. CRM emphasises the project change and motivation more than the others do and it places the product quality factors into the estimates of the component, instead of in the general parameters. CRM has flexible weights of the factors and uses extensive feedback from actualised projects to adapt them. An expert estimator can use any relevant factor, but statistical data is often missing.

CRM and task based estimation estimate the effort directly without a product size metric. It is easier to measure productivity if it is calculated by dividing the effort by product size. However, productivity is not uniformly distributed. The effort needed to handle different types of components varies because they require, for example, different skills and tools. Measuring the productivity at a component level can guide process improvements more accurately.

# 6 Improving component-based development

## 6.1 Introduction

The research question of the forthcoming chapters is how to produce more reusable software. At the beginning the approach is conceptual. The criteria for reusability are found from reuse metrics. The use of metrics is necessary because it is the way in which the reusability of the components can be scaled. The next step is to develop a strategy for improved reusability. Adaptability is especially important because it leads to the mechanisms by which it becomes possible to produce highly reusable software. After the concepts are analysed, the research continues by using the constructive approach. The result is a set of means for the construction of reusable components. Finally, the problems in current programming languages are presented with the implications for their solutions.

The most important enablers of reuse are the adaptability of the components and the organisational support of reuse. As there are a number of studies focusing on process improvement [Jacobson 1997], the following chapters will focus on the study of adaptability. In this chapter CRM will be used in finding ideas that can be used to improve component-based development. One of the ideas, adaptability, will be extended in the successive chapters.

## 6.2 Criteria of reusability

### 6.2.1 Measuring the reuse process

In this chapter the idea of the reuse process at the developer level is used to get a better understanding of reusability. The reuse process consists of four major steps [Goldberg 1995, pp. 223-246]:

11. Define reuse.
12. Set up a process of populating a library of reusable assets (purchase, construction).
13. Set up a process of sharing reusable assets.
14. Set up a process of maintaining reusable assets.

The process of sharing reusable assets is interesting because it is used in developing applications from components. It involves three steps [Goldberg 1995, pp. 241-242]:

15. Communicate the availability of reusable assets.
16. Locate and retrieve reusable assets.
17. Understand and use reusable assets.

Though reusability includes the suitability of the components to each of these phases, finding the components, knowing what they do, and knowing how to reuse them are the most frequent reuse activities [Tracz 1995, page 93]. If we choose money as the unit of reusability, the *total cost benefit of reuse* is identical to reusability. In equation form

Equation 24. $R = n \cdot (cr - cf - cu - ca) - cp$, where

- *R*= reusability,
- *n*= number of reuses,
- *cf*= cost to find,
- *cu*= cost to understand,
- *ca*= cost to apply,
- *cr*=cost to reproduce and
- *cp*=.cost to produce or purchase.

Poulin represented a more detailed cost benefit analysis [Poulin 1997, pages 77-83], which also takes indirect consequences of reuse into the equation. The effort can be used instead of prices without producing a large error. In the case of application frameworks, business objects and parameterised applications the licence costs are large, but they present the largest part of the functionality of the final application.

The cost of reuse is also a central factor in the CRM model. It can be found from the efforts of the components, which include the efforts of finding, understanding and applying the component (chapter 2.7). The baseline effort is calculated by using the number of reuses and the effort of each component (see Equation 5, page 28). The baseline effort is adjusted by the skill effect (cost to understand) and process effect (cost to find). The risk effect is used to assess the trustworthiness of the component.

In CRM, the productivity of reusing a component can be assessed because the effort related to the component is calculated. Productivity in producing an average component is useful if the composition of the products remains the same. Using an analogy from function-points, the average component could be called the component-point. The following study minimises the effort, which satisfies a set of requirements. It does not maximise the productivity of binding standard units of software together.

The CRM-equations (Equation 9 - Equation 5; pages 31 - 28) show that there are several options for minimising the effort of a project. As a mathematical optimisation would require actual data, this study is only a descriptive one.

The functional requirements of an application are expressed as use cases and the solution is a component structure. There is a group of alternative personal tasks which lead to the required solution. The optimal choice has the minimum effort. The effort of a personal task in accomplishing the solution varies because there are several construction alternatives. The minimum effort is obtained when a small number of components, which have a small reuse effort, satisfy the requirements. Mathematically

Equation 25. $$E = \min_j \left( \sum_{c \in C(j)} n_c \cdot m_j \cdot s_j \cdot t_j \cdot r_j \cdot f_j \cdot p_c \cdot E_c \right),$$ where

- $E$= minimum effort to satisfy the requirements,
- $j$ = a personal task choice to create a component structure, which fulfils the requirements (here the task can be quite large),
- $C(j)$ = set of components (=tracking set), which are developed at least partly during the personal task $j$,
- $n_c$= number of (sub)components $c$ in $C(j)$,
- $m_j$= personal motivation effect coefficient of the person in the task $j$,
- $s_j$= personal skill effect coefficient of the person in the task $j$,
- $t_j$= team effect coefficient of the task $j$,
- $r_j$= risk effect coefficient of the task $j$,
- $f_j$= project change effect coefficient of the task $j$,
- $c$= a component, which belongs to $C(j)$,
- $p_c$= process effect coefficient of component $c$ and
- $E_c$= the effort of the component $c$ from the project repository (hours).

The minimum can be found by calculating the efforts of the alternatives and selecting the optimal one. Table 27 shows the use of Equation 25 in finding the best choice. In this example, the requirements can be fulfilled by reusing a group of components A and B or by constructing them again from a group of components, C and D. As the effort of the first choice is 19.5 hours and the latter 32.4 hours, choice 1 gives the minimum effort for the task.

Table 27. Component selection.

| | Choice 1 | | Choice 2 | |
|---|---|---|---|---|
| Component | A | B | C | D |
| Number | 2 | 3 | 5 | 8 |
| Effort | 2 h | 3 h | 2 h | 1 h |
| Product of correction coefficients | 1.5 | 1.5 | 1.8 | 1.8 |
| Sum | 19.5 h | | 32.4 h | |

One requirement for reuse is that the reuse effort of a component is smaller than the effort of constructing it. CRM stores the construction and the reuse effort into the project repository. If the reuse effort is much smaller than the construction effort, the minimum effort will be obtained when the number of reuses is maximised. In conclusion, the components should be easily adaptable to different situations and they should be a solution to a large number of customer requirements.

The process effect coefficients are used to correct the efforts in the project repository in order for them to be applicable in the forthcoming project. Tools and methods can be used to decrease the reuse effort of a component. If the set of components is small, specialised tools (for example program and documentation generators) can be developed to support their reuse. The process of finding and purchasing a reusable component is significantly more laborious than the process of finding one from a local object gallery.

The familiarity with the set of components of the forthcoming application is the key to minimising the effort. The project change effect is mainly concerned with the change of requirements, but familiar components can help in inventing a good, stable solution that meets the requirements. Common and familiar components decrease the risks. Components which have been reused many times have a smaller technological risk than newly constructed components. Thorough testing and reviews decrease risks and thus increase trustworthiness. A smaller effort is needed for teamwork if components of the solution and their interfaces have already been stabilised. The skill in reusing a common set of components will increase during a project, but the motivation is at its highest at the beginning of the project. This is because the development of the application is at its most challenging when the application is constructed from scratch.

As in CRM the total effort is the product of the baseline effort and correction coefficients, which are normally larger than one, decreasing the baseline effort will decrease the total effort significantly.

Due to the fact that large components contain a larger number of features and their interfaces than small components, the efforts of reusing and reproducing them are greater but the benefits are also greater.

### 6.2.2 Conclusion

The criteria for reusability are understandability, ease to find, adaptability and trustworthiness. In CRM terms, understandability increases the skill level and adaptability. Ease to find decreases the baseline effort and decreases the process effort. Trustworthiness decreases the risk level.

## *6.3 Strategy for developing reusable software*

### 6.3.1 Ease of finding

The property "easy to find", is not primarily a property of the component itself, but mostly the property of the reuse library and the reuse organisation. One property of the component which has an impact on reusability is its *proper classification.* A properly classified component can be found from the place where its existence is expected. In order to assist the potential user in searches, keywords can be attached to components. Classification was one of the qualification criteria in the REBOOT project [Poulin 1997, pages 129-130]. In Smalltalk type browsers, the components can be found through inheritance references and from cross-reference lists. Integrated development environments and configuration management tools contain facilities that assist the component search. They all require a component to contain proper keywords or to be located in a logical place. Search engines can perform a search of commercial components from the Internet. A search engine finds the seller's web site using the keywords which the potential buyer gives to the engine.

### 6.3.2 Trustworthiness

The *confidence* factor describes the developer's confidence in successful reuse. If the source code is available and the developer understands it readily, this can bring enough confidence for reuse. Without the source code the developer will assess the documentation and its *understandability*. The user can use the data about previous cases of reusing the component to get a view of the component. Poor *error tolerance* and errors shown in the first tests will decrease confidence. Before large-scale reuse, the components should be tested in pilot projects. This will increase *observed reliability*. Commercial component manufacturers will tend to use their reputation as a means of increasing confidence in their components.

### 6.3.3 Understandability

*Understandability* measures the ability of a developer to understand the syntax and semantics of the component. It is a property of a component but it depends largely on the individual who is assumed to be able to understand it. When we consider the understandability of a component, increasing familiarity is the best way to increase understandability. A component is *familiar* if the developer has used it one or more times. The Cognitive Complexity model estimates that the effort of reusing the component will be two-thirds of the effort of its previous reuse [Cant 1994], which means that using the same components repeatedly will be effective. Familiarity is a factor of the skill effect in CRM.

The effort of understanding is basically done by studying the semantics and the documentation. The understanding of a chunk is based on the mental models which the reuser, studying the component, already has. If the abstraction of the component is already known, it can be understood readily. If this is not the case, all unknown concepts must be studied. This includes tracing them to their origins. Here the reuser is studying and tracing back and forth until every necessary part of the component is understood. The properties of the component which decrease the effort are thorough documentation, including self-documenting code and in-line comments. A component with a smaller size, simple interfaces, fewer parameters, high cohesion and low coupling is likely to be easier to understand.

One part of understandability is the pure volume of diagrams, methods and processes which are required in the documentation of the component. Thus, it is important to avoid overlapping information and too many details. *Analysis paralysis* is a situation in which a development project cannot end the analysis phase because there are always inconsistencies to correct and new ideas to include [Basset 1997, page 207].

### 6.3.4 Adaptability

*Adaptability* is the ability of the component to adapt to different reuse situations. Basically the question of code reuse and adaptability centres on how to use the functionality of a chunk of code without copying and changing it slightly. The copy and change strategy will lead to large programs that mostly contain the same code. It is very difficult to maintain the integrity of the copy and the original code.

Several aspects of adaptability must be considered. Firstly, a component must be *portable* enough to be used in its new environment. Reuse of a component is more likely if it is independent of the environment and it is written using the same programming language as the reusing application. Modularity is also a very important property. The component which has a large number of connections cannot be reused because it cannot be disconnected from its current use. *Generality* is the property of object-orientation which has almost made it a synonym of reuse. Increased generality means that a component can be used (without change) in more varying situations than before. For example, consider reusing a sorting algorithm which can only handle arrays of integers. A more general sorting algorithm, which can handle any kinds of collections, is preferable.

Modularity is a very important property. Fan-out and fan-in-metrics can be used to measure it by counting the dependencies between the modules. The called modules and their callers and common variables create dependencies between the modules. A *package* contains several components which are deployed together. Packaging is essential for successful reuse because proper packages will generate confidence in the reuse of the component. The package includes good documentation and it is easy to locate reusable assets in it. The packages of multiple components can be organised as component libraries or frameworks.

In the next example, one typical chunk of code is presented (Figure 24) and its reusability is evaluated with reference to both black box reuse and white box reuse. The programming language is Java, though the syntax is not important here. The method used in the example could have been taken directly from an application where such functionality is needed. The *semantic mismatch* of abstractions is in special focus.

```
public class AddressBook {
// member variable m_addressBook is a collection of bookItems
// each bookItem contains a Full Name and an address

BookItemCollection m_addressBook;

public Address  getAddress ( FullName parameterName)  {
//get the address of a person whose fullname is given as a
parameter.
//However only surname is used in the search
BookItem auxItem;
do {
          auxItem = this.getNextItem ();
          if ( auxItem.compare(parameterName.surnameOf()))
                     {   break;    }
 }   ( while ! this.endOfItems() );
return auxItem.addressOf();
}
```

Figure 24. Question of adaptability.

Here the reusability of the method getAddress in Figure 24 is considered. Suppose that the customers' addresses are available using the business object class Customer and that the customer's names can be taken from a user interface class. The number of customers is assumed to be small to keep the reuse of the inefficient search algorithm appropriate. In white-box reuse all the code is available and changing the code is permitted. What prevents us from making a copy of it and pasting the chunk into another place? Clearly, there is not enough information to make precise conclusions. The purpose of this method is to perform a search from an address book using a surname and return the address. In black box reuse only the interface is used. The call is of the form address=getAddress (parameterName). There are several problems in the reuse:

- The method getAddress is a method of an address book object. *That object must be within the scope of a possible user.* The same also applies to the class of the parameter, FullName, and to the class of the return value, Address. As an object-oriented programming language is used, their descendants can also be used. It is possible to construct the parameter object parameterName and set a value to it because the class FullName is available. The values in the address book object must be stored somewhere before the making a search becomes reasonable.
- The knowledge of member variable m_addressBook and its item class BookItem can be obtained by tracing them to their class definitions. The member functions of these classes can also be read from their class definitions. *This knowledge is not necessarily public* if it does not belong to the interface, which is available in a black box reuse.
- Without looking at the source code or documentation it is not clear that the method in question uses getNextItem and endOfItems methods.
- If the reuse is put into practice using a descendant class, it is essential that these methods are used for the same purpose.
- The member functions from BookItem class, *compare and addressOf* that are used here, *should also be known* if this class has been overridden. Fortunately, compile time errors are prevented if it is not possible to remove member functions in an inheritance.
- The member function surnameOf in parameter class FullName is also interesting because then its name, surnameOf, reveals that *only surname* is used as a parameter to function Compare.
- The loop in the method *assumes* that the collection is not empty.
- Nothing in the code reveals whether any *side effects* exist. It is not good practice to change the value of the object referenced by parameterName during such a call. Keyword final, const in C++, should be used to make sure that it is the case. Confidence is a substantial promoter of reuse as was deduced earlier.
- If *effective code* is needed and the address book has a large number of items, the search algorithm is an essential property of the method. If the only search function is getNextItem, it is clear that black box reuse is not reasonable if the address book is large. If a hash algorithm is also available it is not clear that it is also used in the member function getAddress.
- In a black box reuse, it is not possible to find out what the developer of this code had in his/her mind. The cues from variable names and the substantial amount of tracing and studying are used yet the cues are still not necessarily valid.

- It is not clear that *the cost of reuse* is smaller than that of rewrite. Reuse based on scavenging old code is typically possible only for the original developer [Carroll 1995, page 2].
- The addressBook class has *no conceptual connection* to the customer class though they both contain names and addresses. A customer is not a special case of an address Book. As the class AddressBook is a singleton class [Gamma 1995, pp. 127-134], it is not a good idea to attach it to the customer class as a member variable. The name of the member function SurnameOf should also be changed to the new use. Depending on the solution, the BookItem class must be rewritten. It could be connected to class Customer.
- The customer object *already contains* a customer name and address.
- The *architecture* creates another complication in the problem. The previous Java-application could be distributed to a browser client, a www-server and a database server. Then the architectural mismatch [Garlan 1995] would make the reuse even more difficult.

The lesson to be learnt from the reuse example is that it is difficult to get anything useful from the old code if the context changes considerably, even though the general purpose of the code remains the same. It is important to notice the objections against possible reuse. Firstly, taxonomy problems, which are implemented by inheritance, prevent reuse. Secondly, the misuse of concepts is a considerable reuse inhibitor. Even if multiple inheritance were used, a reasonable class diagram would not result. If the rewrite were not as easy as in this example, various kinds of implementation inheritance solutions would be introduced. A common factor in these solutions is that the pureness of object taxonomy is sacrificed. This will lead to problems later because there is no longer any justification for placing trust in conformance between business concepts and concepts used in the application.

It is difficult to achieve a fully generic solution to the problem. One component-based solution might be to create a component which separates peculiarities of the address book handling from the other parts of the application (see Figure 25). The component would contain the classes StorageCollection, AddressBook, BookItem, BookItemCollection, FullName and Address. The public interfaces of the package would provide higher-level support for address book handling and nothing more. The search algorithm belongs to utility packages such as java.util, which already have support for handling collections of objects and their enumeration [Borland 1999]. In a pure component-based solution the contract signature would provide more information about the preconditions, post-conditions, invariants and parameters. However, they are restrictive, which decreases adaptability and increases trustworthiness. Notice also that Java supports assertions only for this purpose.

```
public class StorageCollection {
// member variable m_collection is a collection of generic items

Vector m_collection;
Public Address searchItem (String parameter)  {
// get the first object from the collection in which
// the compare method of the item object returns true when
// the parameters match

Address auxItem;  // temporary storage for an item
For (int i=0; i < m_collection.size();i++)
{          auxItem = this.m_collection.elementAt(i);
           if ( auxItem.compare(parameter))
                      { break; }
 };
return auxItem;}

public class AddressBook extends StorageCollection {
// note! The pecularities of AddressBook handling are placed
here
public Address getAddress(FullName parameterName)
{
Address auxAddress=super.searchItem(parameterName.surnameOf();
return auxAddress.addressOf();
}
```

Figure 25. A more adaptable code.

The conclusion is not that reuse is impossible. On the contrary, these problems can be avoided by changing semantically mismatched abstractions. A generic algorithm for a search from a collection was needed. It was placed in a separate base class, which had semantically fewer connections to the possible reuse clients. A large re-engineering of the class structure was done. The lesson remains the same: a considerable additional effort is needed to produce reusable components.

### 6.3.5 Combinations

The ability to establish combinations is a very important property of a reusable component. Let us consider why a list box component is so useful. A list box component is a chunk of code that draws a list box in a window. This kind of component is available in all popular GUI programming environments.

The application program adds text items to a list box. The list box is shown in a window and a user selects one or more items from the list box. The application program acts based on the selection event of the user. Code can be attached to list box events, such as event "got-focus", thereby making the code available in the other parts of the application program. The outlook of the list box can also be modified at run time.

A list box is highly reusable because it is generic. It can be used in any application. It produces a well-defined contribution to the application yet it is dependent on few other components.

A generic component can be coupled with many other components.

Figure 26 shows an example of a component: a list box. The lines describe the data flow. The projects are fetched using a query of projects, and their names are added to the list box in a project maintenance window. In a window for handling tasks a list box is filled with task names using a query of tasks of a selected project. In the person window, the names of the staff are fetched using a query of persons and inserted to the list box of persons. It is important to notice the ability of a list box to connect to a very large number of other components.

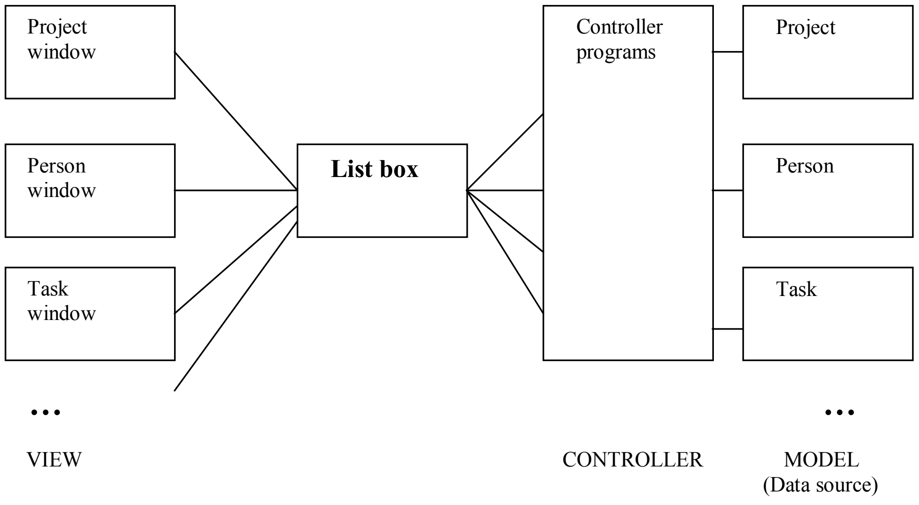


Figure 26. List box in a project management application.

Part of the reusability of a list box comes from generic interfaces. The message parameters are strings and integers which do not restrict the usability. Any component can send a message of a type myListBox.addItem ( textStringToAdd ). If the interface were in the form myListBox.addItem ( queryToGiveTheItem), the list box could only be used with specific queries. In the same way it is essential that a window is a container which can hold any types of objects. It is not restricted to just list boxes, combo boxes and grids can also be used. The only restriction is that the objects in a window must be able to support the responsibilities of a window control. In practice they are inherited from the control class.

Generic interfaces make the combining of components easier. The list box can be used with any data source and in any window. Though the component itself is a standard component, there are millions of ways of using it. In some environments it is possible to make specialised list boxes: a project list box, a task list box and a person name list box. This could be useful if each of them can be used on many occasions and there is a large amount of new code in each specialisation. It should also be noticed that the specialisation should make a special kind of list box. It is not appropriate to place the code to fetch task data in the list box class. Applying inheritance when no new code is introduced is known as taxomania [Meyer 1997, pp. 820-821].

In a three-tiered architecture the business objects are used to couple the user interfaces, business logic and the data storage. They fix the combinations used in a specific business case. It should be possible to change the user interface classes and data storage classes without major maintenance work in the business classes. It is possible if the details of the user interface and storage management are properly hidden from the business objects.

In some cases using static combinations does not produce the required degree of flexibility. The requirement of recompilation each time means that any combination changes will result in too much maintenance and installation work. In distributed object environments it is possible to make the matching of components together at run time. The performance overhead can sometimes be tolerated.

The encapsulation of the code only within abstract data types (and their classes) can lead to an explosion in the number of classes and methods. Suppose, for example, that in an electronic data interchange (EDI) application a supplier's customers had different product ordering formats though only the supplier's generic order processes should have been adapted to each format. There were, for example, 100 data formats and 10 processes. The traditional object-oriented solution would be 100 classes which contain a method for each process. The total number of methods is 1000. Any change in the process must be made in all of these 100 classes. 10 classes for each process or one class which contains methods for all 10 processes would be a better solution. Each method must be capable of handling every data format. If there is a method for adapting the customer's ordering format to the supplier's generic format, only 100 format-specific methods and 10 methods to support the processes are required, therefore only 110 methods are required. A solution, which is based on a similar idea, has been successfully implemented in Noma Industries [Bassett 1997, pp. 188-195].

### 6.3.6 Conclusions

In summary, the strategy for increasing reuse is to use clear and generic abstractions to develop cohesive components which easily form a large number of useful combinations when software products are assembled.

## 6.4 Means for reuse

### 6.4.1 Architectures and patterns

The architecture of an application describes the most important components within the application. Interfaces are used to reduce dependencies between the components. The overall view of the architecture can be seen, for example, as layered [Jacobson 1997, pp. 170-212]. Three-tiered and four-tiered architectures are common, though other kinds of architectures exist. A well-defined object-oriented architecture consists of [Booch 1996, page 43] a set of classes, typically organised into multiple hierarchies and a set of collaborations between those classes. Compliance with the common architectures increases reusability because the possibilities of architectural mismatch decreases.

Patterns are solutions to common design problems [Gamma 1995]. As the use of some of them increases reusability of the created components, a few patterns are described in this chapter.

The use of architectures and patterns increases the understandability of the components because the developers are familiar with the common solutions that they promote.

### 6.4.2 Interfaces

Interfaces are used to reduce complexity. The goal is to reduce coupling and increase cohesion. In reuse, it is essential that a component can be removed from its original context and used in another.

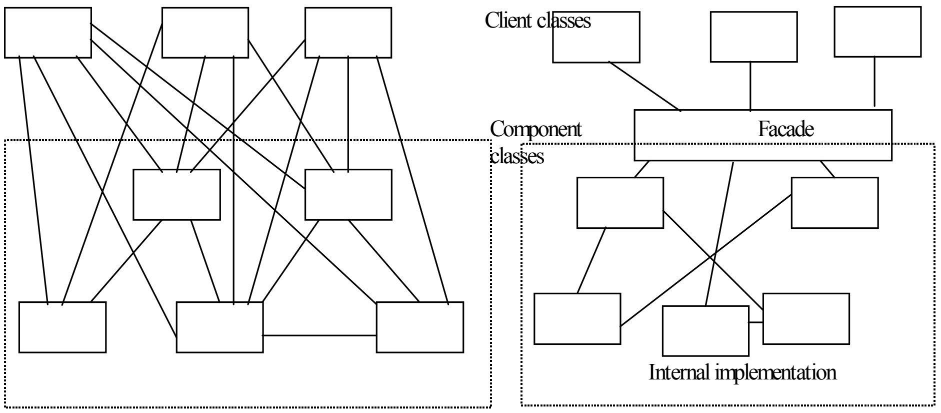


Figure 27. A facade design pattern [Gamma 1995, pp. 185-193].

A facade design pattern is a simple interface. Figure 27 is an example of the use of the facade where the coupling between the client classes and the component classes has decreased. The application programmer needs less tracing to find the services of the subsystem. He/she does not need to understand the implementation issues. This is another advantage because the implementation can be changed when the interface remains the same. UML has new diagrams for supporting the use of interfaces, packages and components. Jacobson proposes using a facade and a package in reusable software components [Jacobson 1997, page 93]. A distributed component separation layer can be partly based on the façade pattern [Herzum 2000]

An interface introduces one additional level of indirection between the components. Adding levels of indirection gives flexibility but it has a performance penalty. Orfali has presented two quotations of generalisations related to this area [Orfali 1996, page 505]:

> Maurice Wilkes: There is no problem in computer programming that cannot be solved by an added level of indirection.
>
> Jim Gray: There is no performance problem that cannot be solved by eliminating a level of indirection.

The interfaces of the components remain more separated from the implementation than object-oriented interfaces. A special modelling language defines the interface and the execution environment provides a number of services, such as security and distribution to the applications [Orfali 1996]. The interface defines contracts, interface definitions (called, offered, hardware, database), preconditions, postconditions and invariants.

### 6.4.3 Inheritance

Inheritance, aggregation and message sending are the only original object-oriented ways of adapting code to a new context [Rumbaugh 1991]. In inheritance the new context should conceptually specialise the original context. For example, an address book is a special case of a collection because it contains only names and addresses. Specialisation is an additive process where new data members and member functions can be added. It is also possible to change the member functions of the original by overriding them.

It is possible to use member functions which have the same names in different classes. The decision relating to which of them is to be used can be deferred to run-time. It is also possible to create a class which is a specialisation of two or more classes.

The advantages of inheritance are quite well known. Code and data from the parent class can be reused in the descendant class. Only the changes need to be handled. Inheritance hierarchies using business concepts will ease understanding of application concepts. The use of abstract data types will increase the level of abstraction. The ease of understanding and reuse will increase the productivity.

The problems of inheritance are:

- *Dynamic binding* is “the guarantee that every execution of an operation will select the correct version of the operation, based on the type of the operation’s target” [Meyer 1997,page 1195]. *Dynamic typing* has a serious drawback in that it is not known before a particular run if every method invocation will be solved at all. If there is not a method that corresponds to the invocation, an error message “message not understood” is given. That is unacceptable in mission critical applications.
- The second problem comes from the generalisation and specialisation structures. Reality cannot be modelled using a single inheritance taxonomy. A bird is a flying animal. An ostrich is a bird, but ostriches do not fly. For example, if a customer is an external agent and an employee is an internal agent, an object, which is both the customer and the employee, cannot be defined in a single inheritance taxonomy [Bassett 1997, page 144]. One solution to the latter would be to duplicate the properties of the classes or to fragment them into clusters of smaller classes. The multiple inheritance solution requires that conflicts between the duplicate properties are solved. Both the customer and the employee might have a name and address. Having two copies of these, as object attributes, would not be a good solution because it reduces cohesion and manageability.
- A problem previously mentioned was the inability to remove member variables and member functions from the inherited classes. That inability guarantees that the invocations of member functions will succeed. In other words, it prevents the removal of the ostrich's ability to fly. Aggregating all member variables will also result in unnecessary variables. Their existence will lead to more error prone code and added difficulty in understanding and maintaining the code.
- The fragmentation of member functions is also a property of object-oriented code. Small method size is a desired property [Lorentz 1994, page 43] because reusing such methods by inheritance is easier than reusing larger methods. A part of the code of the method cannot be overridden without first splitting the method. In object-oriented programming a new class must be created. The resulting web of calls between layers of code is sometimes called lasagne code [Bassett 1997, page 153].
- Another problem is that the adaptation of a method for reuse often requires a definition of a new class where the method can be overridden. However, every reuse is not due to a new specialised concept. For example, where a new context of invocation is found and the adaptation is better suited to the class in question. In C++ and Java member function overloading can be used for that purpose. Overloaded member functions have the same name but their parameter lists are different. However, there may be a new context which would like to use the earlier parameter list but would also want to adapt the code. Scattered fragments of code add to the tracing time needed to understand and change the code and thus reduces productivity.

- The implementation of a polymorphic call in different classes is also a problem. Their interface is the same but the implementation can evolve in different directions, which can lead to inconsistent behaviour later. Some of these problems can be avoided by status checking at run time. For example, we have methods A and A' to get the time. At first both of them use the system time. If we change A' to use time of a different computer, the successive calls to A and A' can show inconsistent time values.
- The problem of the safe modification of base classes in the presence of independent extensions is called Fragile Base Class Problem [Mikhajlov 1997].
- Inheritance creates dependencies between classes.

### 6.4.4 Extensibility by templates

Templates are a way of increasing flexibility in object-oriented class hierarchies. Templates are used to define parameterised (generic) classes. For example, it would not be reasonable to define an individual class for each type of item in a collection. Figure 28 describes the flexibility of horizontal and vertical generalisation.

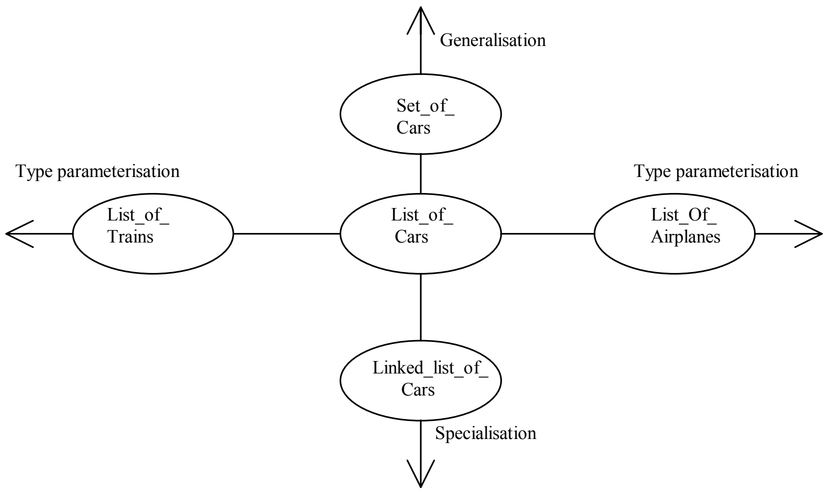


Figure 28. Horizontal and vertical generalisation [adapted from Meyer 1997, pp. 317-318].

Parameterised classes are also added to Unified Modelling Language [Booch 1997, pp. 26-27] to be used in static structure diagrams. UML also includes another mechanism for the classification of types and use cases: stereotypes. A stereotype is a classifier marked by <<stereotype name >> near the name it is classifying. For example, the type parameters in the previous example can be classified by a stereotype <<vehicle>>.

The previously described problem in the getAddress-function can be solved using C++ templates (see Figure 24, page 119). A generic find algorithm is needed [Koenig 1996]. In a template the class to be used is parameterised. The requirement that the used classes implement a compare-operation still remains. The support of operation ++ corresponds to the support of the function GetNextItem.

Templates are useful in producing collections which contain many types of items. These items are iterated and polymorphic calls are done during the iterations.

The benefits derived from templates are their versatile use in classifying abstractions and in producing generic reusable components. Templates add a second way of producing reusable components other than traditional inheritance.

### 6.4.5 Bassett Frames

Frames are a specific way of implementing reuse [Bassett 1997, pp. 70-195]. They are based on a frame processor, which modifies source code using specific directives. These directives direct normal copy-paste and find-replace modifications, which the programmers usually make in white-box reuse.

Bassett's frames are a variation of a macro language. Many current programming languages include a macro language but this remains in the background. Copying code is familiar from COBOL copy-statements and C 's #include directives. Variables can be assigned by #define directives and they can be tested by #ifdef and #ifndef directives. Macros are mechanisms for reuse which have already been used for some time.

The first lesson to be learnt from Basset comes from the better utilisation of the typical features of a macro. Adding a programming language level support to direct the generation of the source code adds flexibility to it. In macros there were multiple level copies but the intelligent steering was usually missing. Macros are also viewed as tools used by experts which are difficult to understand because the outcome of realising the macro was typically left inside the compilation process. The code construction process only contains recompilation. Very simple directives typically direct code generators. Typically the outcome of a generation is hard-coded inside the generator.

The second lesson comes from seeing reuse as a construction time process ruled by the same-as-except principle. It is stressed that in construction the properties of reuse such as generality and adaptability are very important. In use it is important that the component's functionality, efficiency and ease of use are appropriate.

The traditional way of programming, taking a chunk of code and pasting it into a new place and doing a little editing, is extremely effective in a settled environment. Maintenance is the problem because keeping copies consistent remains the programmer's responsibility.

The third lesson comes from the principle of maximum diversity with the minimum number of components. Frames capture processes in a way that makes it possible to reuse the process implementation with the objects needed.

### 6.4.6 Viewpoints on typing in current programming languages

The basic construct of type checking is the execution of sending a message x.f ( arg ), which executes the operation f on the object attached to x using argument arg [Meyer 1997, page 611]. It is also possible to have no arguments or more than one argument. A type violation occurs if there is no function f in the class of the object attached to x or in its ancestor classes or arg is not an acceptable type of argument of function f.

In dynamic typing the checking of type violations is done at run time during the execution of the function call. Static typing uses rules that determine that type violations will not occur. These are checked at compile time. Statically typed languages require that each variable is declared to be of a certain type. They check every assignment and function call and assure that no type violations occur.

In an inheritance tree, there can be more than one function which realises the typing rules. The difference between typing and binding is that typing considers whether there is at least one applicable function. Binding considers which one of these should be used. So it is possible, even desirable, to use static typing and dynamic binding.

The benefits of static typing come from better reliability, readability and efficiency. Static typing is used to detect type violation errors at a compile time. Dynamic typing could only detect violation errors at run time and in certain runs. If high reliability is required, for example, in patient monitoring programs, dynamic typing is not appropriate at all. Type definitions make programs more readable because the programmer can find out the types of variables from the definitions. Variables, which change type during the program run, are especially difficult to trace. Better efficiency is due to better binding algorithms. It is quite normal for the name of a function to be ambiguous. In the context of static typing the algorithm only searches for the polymorphic functions rather than all the functions in the application [Meyer 1997, pages 615-616].

The drawback of static typing comes from its restrictions in reuse. It is restrictive in that the code of function f can be reused only when the types of x and arg are within their own inheritance trees. Programmers who are used to programming using Smalltalk state that this is their main reason for choosing this particular programming language. In some applications it is even desirable to be able to add new functions and attributes to a class at run time.

Static typing also has problems if it is needed to override a function. These are related to properties called covariance and descendant hiding [Meyer 1997, pages 621-628]. Covariance allows for changing the argument types of a method when the class is redefined, if the types conform to the original argument types. In descendant hiding, the method does not exist in the descendant class. Both of these features are very desirable [Bassett 1997, page 141]. Fortunately, it is possible to check both of them in compile time [Meyer 1997 pages 621-628]. The solution is in checking all the assignments and function calls and rejecting those which could result in a type error. The fact that it also prevents some possible solutions, which would not result in a type error in any practical program run, is not a serious problem.

The possibilities offered by added reuse can be improved if the static check is based on checking valid calls. The purpose of type checking is to prevent invalid function calls. It is sufficient that the arg is attached to an object that supports all the function calls of arg within f. It is worth looking back at the previous example of getAddress function (see page 118). Its parameter parameterName is of the type FullName. The parameter is used in making a function call parameterName.SurnameOf(). It is not necessary for parameterName to be of the type FullName or some of its descendants. Syntactically, it would suffice that its type contains the function SurnameOf. That would increase reusability at the cost of decreased readability. The semantics of SurnameOf must also be assured.

# 7 Verb Classes - Design for Reuse

## 7.1 Introduction

The aim of this chapter is to introduce an architectural design that can be used as a means of applying the reuse strategy presented in chapter **6.3**. The question here is how to organize the software components to achieve the clearest and the most cohesive abstractions, whilst making the combination straightforward.

The combination of functional and object-oriented programming languages is useful because most nontrivial designs need the properties of both in order to meet needs of business [Coplien 1999]. This approach, 'verb classes', is one in which the combining of components is easily done because the expressions occurring in natural languages can be adapted to a programming language [Virtanen 1999]. This approach has both object-oriented and functional features.

## 7.2 Related work

In object-oriented programming, the location of methods in class hierarchies is a problem because of the large number of objects and methods. Another problem is that it may not always be easy to determine which of two or three classes should contain a given method. For example, in a library information system a method which implements the loan of a book can be placed in class Book, LibraryUser or Library [Wilde 1992]. In the class Book the method emphasises the movements of the book and in class LibraryUser it focuses on the library user. The class Library contains the view of the services of the library. To gain the reuse benefits it must be possible to locate the code to be reused fairly efficiently and unambiguously.

The problem studied by Krishnamurthi et al. is that many problems require recursively-specified types and a collection of tools that operate on those data [Krishnamurthi 1998]. In anticipation of future extensions and reuse, the data and the tools should be implemented in such a way that it is easy to add a new variant of data and adjust the existing tools accordingly and extend the collection of tools. As the source code may not be available and because changing the code is cumbersome and error-prone, these extensions should not require any code changes.

As a solution to these problems, Krishnamurthi proposed a synthesis of object-oriented and functional design to promote reuse [Krishnamurthi 1998]. The recursive data types can be implemented as classes and the tools are methods in those classes. The first solution was implemented as an extensible visitor pattern (Figure 29.) and the final solution will be to create a new programming language, called Zodiac, which contains these features as a language extension. The example application specifies a set of data (Shape) partitioned into three subsets: squares, circles and translated shapes and a tool that determines whether the point is inside the shape (ContainsPt). The set of shapes is then extended to the union of the square and the circle and a new tool, which can shrink a given shape, is added. The form of the system after extensions is shown in Figure 29. The thick rectangles represent concrete classes, the rhomb (Shape) an abstract class, and the thin rectangle (ShapeProcessor and UnionShapeProcessor) interfaces. Solid lines with an arrowhead show inheritance, while those without an arrowhead indicate that a class implements an interface. Dashed lines connect classes and interfaces. The label on a dashed line names a method in the class that accepts an argument whose type is the interface. The boxed portion is the extended data type and its corresponding processor. For a processor and data type extension all code outside the box can be reused without any change.

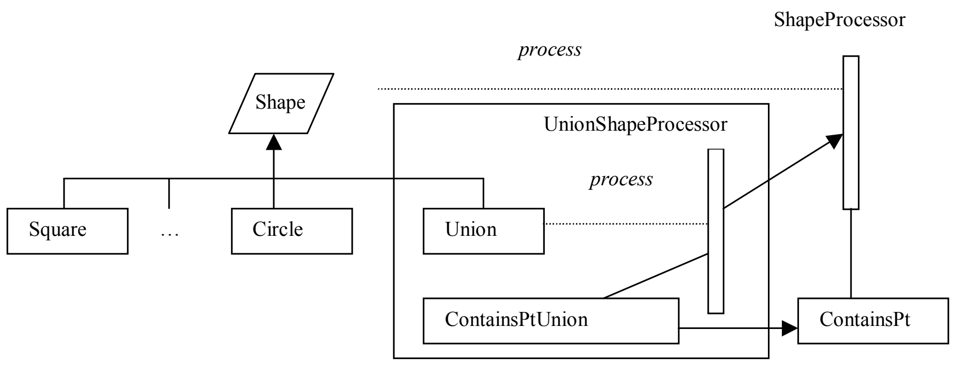


Figure 29. Extensible visitor [Krisnamurthi 1998].

Holzmüller proposed another way of synthesis which uses polymorphic sets of types and subprograms to create an experimental language called 'HOOPLA'. [Holzmüller 1998]. In HOOPLA, the dispatching is decided partly statically, partly at run time based on the types of the arguments and the result type.

The Semantic Object Modelling Approach (SOMA) has some interesting issues that should be examined when considering a new approach to object modelling [Graham 1994]. Like many other methods, it uses text analysis as a means of making the analysis. The text that is analysed is task scripts, but other documentation is used as well. At the beginning a list of possible objects and operations is constructed using the principle that nouns are candidate objects and verbs are candidate operations. In discussions with the users the analysts aim to eliminate duplicates and objects which are not significant to the area of the system. There may be objects, which should be attributes and vice versa as well as nouns, which are used as stand-ins for verbs. It is also difficult to decide whether operations should be placed in a general class or whether they should be distributed among specialised classes.

As the previously described development steps include properties that are common in natural languages, it is useful to have a brief look at them.

## *7.3 Natural languages*

There are differences between natural languages and programming languages [Naur 1975]. Natural languages are mostly used in their spoken form and programming languages are written using a very exact and formal syntax and semantics. Natural languages are intentionally fuzzy, which is vital to the ability of the language to develop and extend itself to cover new ground.

In his study of natural languages, artificial auxiliary languages such as Esperanto and programming languages, Naur defines the quality of a language by stating that the language which is able to express the greatest amount of meaning with the simplest mechanism ranks the highest [Naur 1975]. In their evolution over many centuries irregularities, such as the number of conjugations, in the natural languages have decreased and the forms of the languages, such as the word order, have been standardised. The increase in the abstraction level of natural languages facilitates a larger variety of combinations than that which was possible earlier. The study by Naur, who is best known as co-originator of the Backus-Naur notation, supports the idea that programming languages should preferably be built from a few, very general, very abstract concepts that can be applied in many combinations, thereby yielding the desired flexibility of expression. The desired freedom of combinations implies that every combination of operands and operators should have a meaning. Type restrictions in some programming languages should, according to that principle, be replaced by the rapid check of correctness in the combinations.

The basic elements of natural languages are words and sentences, which are structured collections of words. There are abstraction hierarchies of words, which facilitate the ways in which a language can be extended. For example, a car is a special case of a vehicle, which means that it inherits the general properties of the vehicles and has some specific properties of its own.

Figure 30 is an example of a hierarchy based on verbs, these being parts of speech that are common in natural languages. The example is based on MOO, a special language for virtual reality games, in which these kinds of structures are common [Curtis 1997].

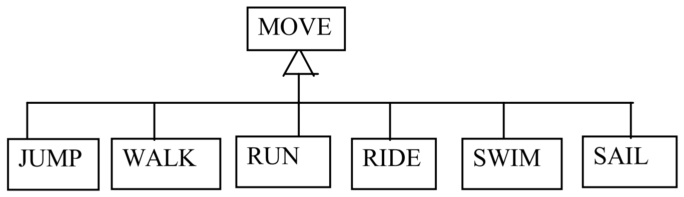


Figure 30. The specialisation hierarchy of the verb "move".

Classification of data, sorting data in ascending order and sorting data in descending order are examples of specialised arrangements of data structures (for example collection, array, matrix, queue, cube, database table, user interface control) . The algorithms are easier to reuse if they are not tightly coupled to the data types.

In addition to this, new verbs can be derived from existing ones and it is also common to construct a noun from a verb [Fromkin 1993]. The use of the form 'moving' or 'movement' instead of the verb does not change the argument that natural languages have separate hierarchies for these actions.

## *7.4 New ideas for extended reuse*

The new design architecture for extended reusability is based on the analogy with natural languages. The concept verb class is an original contribution of this study.

We suggest that the class hierarchies of user-defined nouns and verbs imply the structure of the software. The nouns correspond to objects in the object-oriented approach and the verbs correspond to functions in the functional approach. Both nouns and verbs are part of their own class hierarchies, which contain generalisation-specialisation, aggregation and association relations in the same way as in object-oriented programming. This separates the verb classes from the functional paradigm.

The program consists of statements that combine nouns and verbs. The use of combinations of separately derived words leads to extremely versatile program code. The grammar of the statements is very regular. The basic form of a statement is one verb, which defines the action, and a noun which corresponds to the object of the action in that order. A statement typically contains more than one noun. Imperative statements can be used and the implicitly defined subject can be omitted. Explicit actors can be implemented by their own objects. Of course, the grammar must have a statement which creates the objects. Statements which combine a large number of objects to a verb are more expressive than those of the current programming languages. They are also easier to understand because the narrative descriptions of the use cases created in the system analysis can be utilised straightforwardly (see an example in page 140). Technically, the methods of the verb classes are overridden to allow a variety of numbers and types of the parameters.

The guidelines for placing the code to the classes are quite straightforward, though developing software still requires the judgement of a developer as would be expected:

- Verb classes contain methods that implement the functionality.
- Child classes in verb hierarchies reuse the code of their ancestor classes by "super"-calls.
- The parent classes in verb hierarchies have a functionality that specialises itself by invoking the code of the child classes (a "this" call).
- Noun classes are abstract data types that contain the attributes that implement the data and simple methods (such as get and set-methods) used to handle these attributes.
- Generic and abstract noun classes are used as types of the parameters of the methods of the verb classes.
- The methods of the verbs support a variety of noun classes (typically by overloading and overriding the methods).

## *7.5 The implementation*

### 7.5.1 Multi-paradigm programming

Multi-paradigm languages are a way of developing a practical implementation of the suggested 'verb-class'-concept. A programming paradigm is a way of conceptualising what it means to perform computation, and also of structuring and organising how tasks are to be carried out on a computer [Budd 1995]. The goal of multi-paradigm computing is to provide the programmer with a rich set of tools for the selection of the solution technique which matches the best the characteristics of the problem. In imperative programming the computation is viewed as a task in which the processor manipulates a memory. This is very close to the hardware view of the actual computing. In object-oriented programming there are several computing units, objects, which have their own processing units and memories which communicate by sending messages to each other. The capability of extending the system incrementally by inheriting the properties of the existing system to new computing units is a characteristic property of object-orientation. Functional programming sees the computation as a series of mathematical functions, which are applied to the original values to get the results. In logic programming the programmer writes a set of logical facts and a question and lets the computer work out the solution by using logical transformations.

Leda is a multi-paradigm programming language which contains the properties of object-oriented, functional and logic programming [Budd 1995]. Technically this has been implemented by adding types for functions and relations to an object-oriented skeleton. Variables can then hold and transfer functions and relations as values. The BETA language unifies the abstraction mechanisms of class, procedure, function, co-routine, process and exception to an ultimate mechanism called pattern [Knudsen 1993].

Templates are a way of introducing a parameterized group of classes [Koenig 1996, Booch 1997]. As they are especially useful in the context of algorithms, it is natural to use them with verb classes. As there is a need to prevent invalid function calls, the compiler or a checker program should prevent this by making a static analysis of the possible calls. This solution sees some impossible program flows become valid, but it is, nevertheless, a better solution than run time checking. Meyer has used a similar approach to the descendant hiding problem of inheritance trees [Meyer 1997].

The popularity of a language depends on the support around the language, not on a language itself [Naur 1975]. This means that the languages in general use are more important than the academic languages. C++ is an important and generally used language which supports techniques for multiple paradigms: classes, overloaded functions, templates, modules and procedural programming [Coplien 1999].

### 7.5.2 Precompiler solution

Multi-paradigm and object-oriented programming can be used to solve the problem of implementation in the presented design for extended reuse. However, a solution, which uses formal pseudo-code and a pre-compiler creates a better connection between the real world abstractions and the realm of the programmer.

The implementation of nouns is recognizable from object-oriented programming. The nouns are objects, which are instances of their corresponding classes. The derivation tree of verbs can also be implemented using an object-oriented language. Instances of verbs can be used to record the run-time data of the event of the verb. At least one instance of each verb is needed. Verb classes are usually singleton classes [Gamma 1995]. In hybrid languages, such as C++, there is no need to express functions as member functions of classes. If the functions are used directly, the derivation trees of functions must be coded explicitly. The parameters of the functions are nouns, which are implemented as objects. Verb parameters are not allowed in order to keep the syntax simple. The member functions within the data types should be specialized to manipulate that data type. A function in a verb class typically contains statements, which call other functions in other verb classes and noun classes.

Combining nouns and verbs produces statements. These can be extremely versatile because the number of combinations is very large. It is true that not all of the combinations are allowed or useful. There are also differences in the methods which implement the same functionality for different objects. This does not imply that it would be necessary to include all the functionality within data types. The components can be of any logical size; there should be higher and lower level verbs and nouns. A more detailed presentation of the pre-compiler solution is presented in [Virtanen 2000c]. Finally, the pseudo-code written in natural-like language is pre-compiled to the code of an object-oriented language.

## *7.6 Example*

The example of the use of verb classes below illustrates how they are used to control and monitor a chemical factory which produces alcohol from sugar, malt, yeast and water. The factory handles processes, containers and chemicals. The support software is organized in such a way as to mirror the processes in the factory [Fayed 1997]. Fayed argued for modeling the process instead of objects because generally the duration of the processes is longer. The target was to develop systems which could live longer than the current systems.

Figure 31 depicts the specialization of the process. A recipe contains the specialized processes in the application.

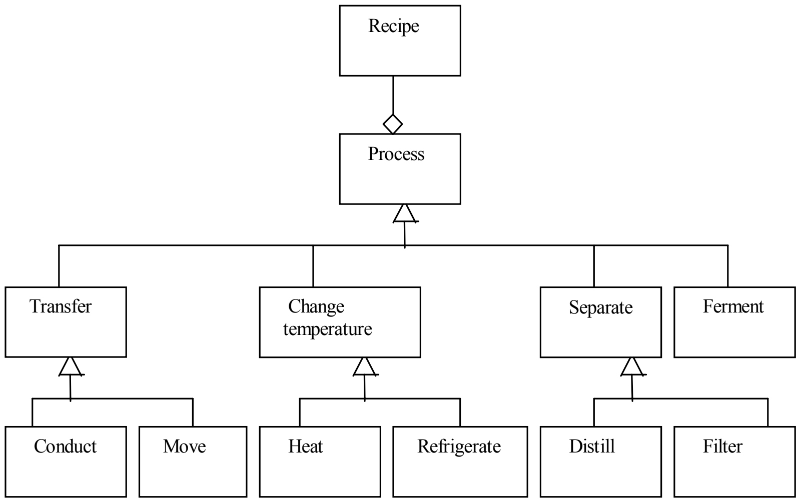


Figure 31. The specialisation of a process.

The benefit of this classification is that the most important and permanent recurring processes in the factory system are modelled. The traditional choice would be to model the equipment or the chemicals that are needed in the process. The immediate problems presented by this approach are the need to spread the code for processes to more than one derivation tree and the need to apply a process to more than one object. Normally, the methods of a class should operate on the attributes of the class. When a collaboration of several objects is necessary, it is difficult to decide to which of the classes the method should be placed. The suggested class hierarchy is useful because the child classes use inheritance to reuse the code of the parent classes. Figure 32 depicts the class hierarchy of the chemicals and Figure 33 that of the containers.

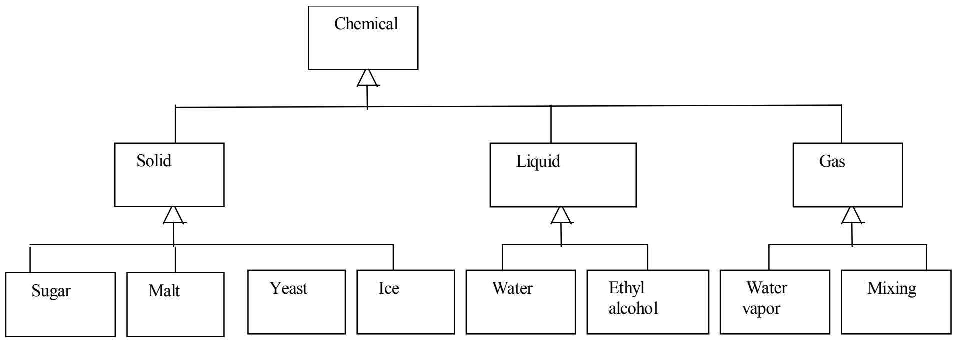


Figure 32. The class hierarchy of chemicals.

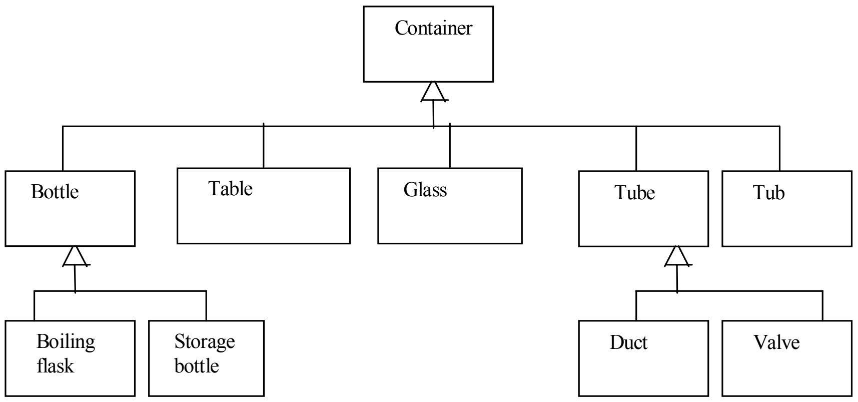


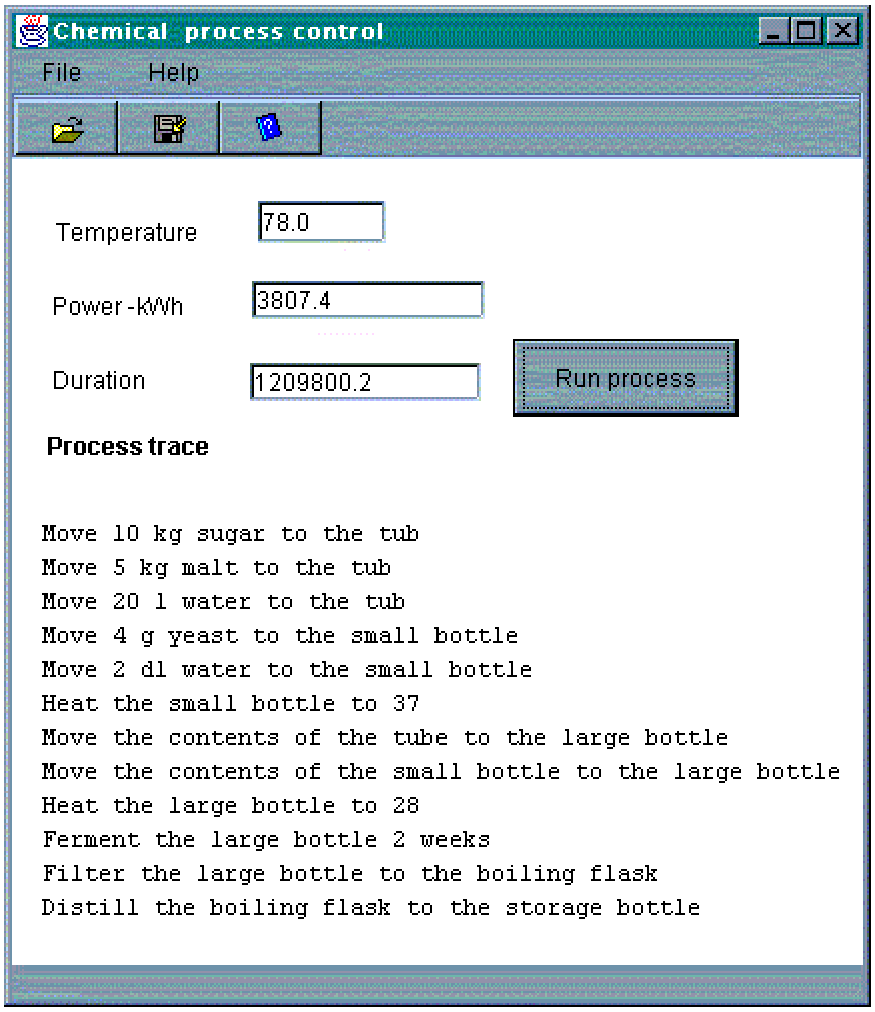
Figure 33.The class hierarchy of containers.

The use case of alcohol production can be written as pseudo-code in which statements begin with a verb:

```
18.Move 10 kg sugar and 5 kg malt and 20 litres water to the tub.
19.Move 20 g yeast and 2 dl water to the small bottle.
20.Heat the small bottle to 37 °C.
21.Move the tub chemicals and the small bottle chemicals to the large bottle.
22.Heat the large bottle to 28 °C for 2 weeks.
23.Ferment the large bottle for 2 weeks.
24.Filter the large bottle chemicals into the boiling flask.
```

25.Distil the boiling flask chemicals the into the storage bottle.

The purpose of the use case is to control the duration, temperature and power consumption of the chemical process. The code of the example is in Appendix E: Verb class - example, page 178. Figure 34 shows its user interface.



Temperature 78.0

Power -kWh 3807.4

Duration 1209800.2

```
Move 10 kg sugar to the tub
Move 5 kg malt to the tub
Move 20 l water to the tub
Move 4 g yeast to the small bottle
Move 2 dl water to the small bottle
Heat the small bottle to 37
Move the contents of the tube to the large bottle
Move the contents of the small bottle to the large bottle
Heat the large bottle to 28
Ferment the large bottle 2 weeks
Filter the large bottle to the boiling flask
Distill the boiling flask to the storage bottle
```

Figure 34. User interface of the example.

The classes Conduct, Refrigerate, Tube, Duct, Valve, Gas, Water vapor, Mixing and subclasses of Solid, Liquid and Gas have been left out to make the example simpler.

The pseudo-code is manually pre-compiled into Java-code, which creates the process when the user presses the file-open button. One process step is run with the button "Run process" and the result of the process is shown in the window.

The verb classes have constructors, which connect the verb with the nouns and a run-method, which is performed by the "Run process"-button. For example the sentence "Move 10 kg sugar to the tub" is translated to

```
Tub theTub=new Tub();
Move step1a=new Move("Move 10 kg sugar to the tub",new
Solid("sugar",10),theTub);
```

The three class hierarchies of the application are cohesive and have a limited number of connections to each other, though type checking has caused a few restricting type casts to the code. Adding a chemical, container or a process can be done without changes to the existing code. For example, adding classes Glass, Ice and Refrigerate can extend the functionality of the example application. The classes are semantically pure: the process classes extend the functional properties of the application, the containers are versatile containers with their support operations and the knowledge of chemicals is placed to the chemical classes.

As the processes, containers and chemicals can be combined flexibly, the number of possible recipes that can be controlled by this application is very large.

## *7.7 Benefits*

It is important to re-iterate that the criteria for reusability of software components are understandability, ease to find, adaptability and trustworthiness. The strategy for the creation of such components is to have clear and generic abstractions which are easy to combine.

Natural languages are the way humans think and create abstraction [Fromkin 1993]. As the presented architecture uses the same way of organizing the abstractions and expressing the statements that natural languages uses, it is easier to understand. The programmer does not need to do as large a mental conversion from the application domain to the programming realm as was the case previously. The abstractions are clearer and more cohesive if the verb classes are used in certain types of applications. This is because they are closer to the abstractions used in human thinking. Humans specialize and generalize algorithms in their thoughts and languages. Placing the variations in the actions into different classes to the variations in the data types adds cohesion in both of these classes.

The reusing work involves tracing the code, grouping the code in order to understand it and finally making the modifications. The new approach in this study reduces the need to trace large areas of the code because the code is normally placed within the derivation hierarchies of the verbs and the data types are handled separately. Dictionaries of words can be generated to assist the search of the classes. As the polymorphic functions are found within the verb classes in the normal cases, the components can be found more easily.

Words are context sensitive and phrases connect several words to new meanings. These are also included in dictionaries used in natural languages.

The most important advantage of the proposed solution is its adaptability. The versatility of the expressions, which can be measured as the number of useful combinations of nouns and verbs, is larger than has been the case traditionally. The possibility of easily adding new nouns and new verbs solves the problems posed by Krishnamurthi and Holzmüller. The best feature of verb classes is their versatility in expressions. The combinations of verbs and nouns create a rich and extensible set of useful expressions. Adding new nouns and verbs is easy because data types and functions are not too tightly coupled together.

In this study the functions are assumed to be so specialized that the trivial solution of placing all the verbs as member functions of the base class is out of the question. If the inheritance tree of the process in the previous example cannot be flattened into a one-level solution, it is impossible to add the verbs to the tree of the containers without sacrifices. Another problem to be solved would be to decide whether to include the abstractions of the actions in the tree of containers or in the tree of chemicals. Neither of them is conceptually in the right place because the tree of the process is concerned with the action, which has connections to both the containers and the chemicals. If the actions are placed near the root classes of the data type hierarchy, these classes become large, containing a very large number of methods which are loosely coupled to the data type itself [Microsoft 1993].

The drawback to the proposed solution is its weaker trustworthiness. This is because relief in type checking is implied. The functions in the verb classes will accept parameters which do not have strict type restrictions. In Java the use of the class Object as the type of parameter guarantees that any object can be used in that place. Compile time checking of the precompiler can be used to assist the programmer in avoiding the illegal function calls.

## *7.8 Applicability*

The applicability of the presented new ideas depends on the situation. The application must contain a versatile functionality. Verb derivation trees are clearly useful when there is a need to organize functionality which has common parts and which makes up a taxonomy of abstractions. The application area is not restricted. General business applications can also contain versatile functionality. For example, a payroll system contains several ways of calculating the wages. The implementations of data communication protocols vary mainly on the procedure side and use high-level abstractions for the data. Typically, a packet or part of the packet does not change in the program implementing the protocol. This implementation uses only a minor part of the packet, called the header, to direct the operation.

Many basic algorithms are not data type specific. For example, sorting algorithms can be organized according to the class hierarchy of the algorithms (see Figure 35). Each method should be as data type independent as possible. The method must contain a collection specific method invocation for the enumeration of the members of the collection. The class of the item type of the collection must also contain a 'greater than' method.

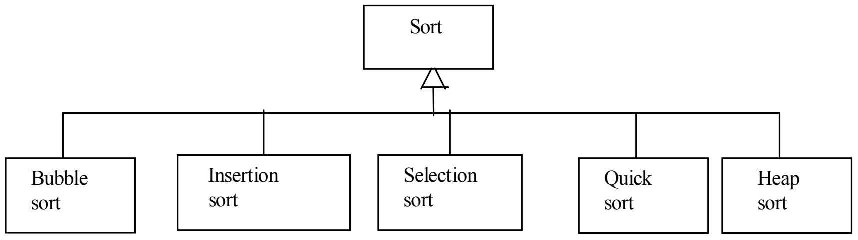


Figure 35. Sort algorithms.

The principles of 'verb classes' can be used as a design pattern, but only the use of an extended programming language or the use of a programming language extension provides tools that can implement it properly.

# 8 Conclusions and future work

The purpose of this study was to improve component-based software development by employing a better method of estimating the development effort and by eliciting the means to produce more reusable software components.

The most important steps where effort estimation is needed are in feasibility studies, scheduling and process improvement. A new general-purpose method, Component Reuse Metrics, CRM, developed in this study, combines component technology and human behaviour into simple calculation rules. CRM uses the component structures of the products, which can be obtained from a CASE-tool or from a configuration management system. CRM describes the software size by reference to the numbers of different kinds of components. There are two component structures, the external user's view and the internal implementation view, which are closely connected. Each component is assembled to a product using one or more human tasks. As this project work breakdown structure is tightly connected to the component structure, the effort of a component can be calculated. The development process defines the tasks required to produce a certain component. However, the effort of a task depends heavily on the project and human factors. CRM classifies them as skill effect, motivation effect, team effect and risk effect and estimates the efforts for each of them. The skill effect puts the required training into the calculations and the motivation effect assesses the effect of the motivation of the staff on the development effort. The effort needed in sharing information is included in the team effect. The most important risk, the additional features added to the product during the development, is taken out of the risk effect, which is otherwise calculated as an effort factor of a task. The project change effect can be revealed from the differences between the original and final component structure.

The CRM calculations require a computer and as a minimum requirement a spreadsheet in order to support the method. Accurate calculations require accurate historical data. That data should contain information about the product, the project, the process and the people in the project. Information can be collected by using CASE tools and project management tools. A project can be calculated any time before, during and after the project. The recalculations can be used to validate the method and its parameters, and in project management to adjust the effort estimates of the project and the product. At the beginning of the project, the requirements of the components are less well defined and their accuracy will increase during the project. It is useful to record this change history. The counting of components can be automated in CRM. However, humans always make the necessary judgements related to the project and human factors.

Task based estimation is the most commonly used effort estimation method in industry. The most important metric is the number of windows and database tables. CRM can be seen as an extension of that method. CRM is planned for component-based development, but it can be adapted to measure traditional development, too. As project tracking gives CRM a lot of timely feedback, which is used to adjust the estimates of the successive phases of the project, CRM is suitable for modern iterative software processes.

A survey performed in this study confirmed that tenets of CRM conform to the experts' view of the tenets of estimation of software development. In practical evaluations CRM created useful estimates without an excessive estimation effort. The case studies supported the idea that component-based development creates an acceptable component structure for CRM calculations. This can also be true in traditional applications. The weights of factors in the project and human effects were estimated in the survey and there was confidence in the possibility of assessing these effects. This information can also be used without CRM. The case studies, where actual projects were estimated and accomplished, showed that the accuracy of CRM relies on the accuracy of the assessments. The assessment of project and human effects proved to be difficult, at least without experience and any historical data. More case studies are needed to verify the results of this study in different, larger projects.

Due to the importance of the productivity part of the effort estimates, there is a need to focus on a more thorough analysis of the factors relating to productivity. CRM creates a framework for this. In its simplest form CRM attaches the historical effort to each component. The total baseline effort is the sum of the efforts of the components. CRM estimates the effort for training, motivation, risks, process, teamwork and product changes in order to make a better estimate. The new estimates are calculated using the human factors in the new project. Estimates of project and human effects are subjective and inaccurate. There are also ethical and legal restrictions in storing detailed personal information. However, project estimates which do not take these effects into account cannot be accurate.

CRM was compared to existing estimation methods. CRM emphasises the change of the project and the motivation of the staff more than traditional methods do and it applies productivity coefficients at a task level, which means that the distribution of the project and human factors is better accounted for.

CRM has several advantages. It is suitable for component-based development and iterative processes and it is usable for feasibility studies, scheduling and process improvement. CRM utilises a large amount of timely feedback and. measures the important parts of the development because it focuses on both project and human factors. The effort estimation is a by-product of the development process and the estimates are available as and when they are needed. The survey respondents considered the effort that was needed for the estimation to be worthwhile. It is also easy to change from current practices in industry to CRM.

In this study a survey and a small number of case studies test the CRM hypothesis. A more thorough analysis of the equations and the calibration of the parameters are left for future research, which should have a large number of pilot projects and make comparisons with other estimation methods. In any future research statistical factor analysis could be used to analyse the dependencies of the project and human efforts. This could also lead to improvements in the CRM equations.

The second issuein this study was ways of improving the possibilities of enhancing the reusability of component-based programming. The study started from reuse metrics, which revealed that the basic criteria of reusability are understandability, ease to find, adaptability and trustworthiness. Adaptability was chosen for deeper analysis and this analysis revealed that the best strategy for producing reusable software is to use clear and generic abstractions to develop cohesive components which easily form a large number of useful combinations when software products are assembled. Several extension mechanisms were studied and finally the restrictions of the programming languages were examined.

Reusable software components are not a by-product of normal development because typical software consists of specific parts which have a large number of specific dependencies. The means to accomplish the strategy are to use generic code, which can handle a large number of situations, and to add interfaces to decrease the dependencies between the components because in addition to increasing cohesion, decreasing coupling is also necessary. A component must have proper documentation to assist the understanding of its intended use.

It is possible to extend the traditional methods of programming to increase reuse. There is a trade-off between adaptability and type safety. More research is needed in producing programming languages which are more expressive than current languages, but which are still safe to use.

Verb inheritance is a new concept which combines features of object - oriented and functional programming. If both verbs and nouns have separate class hierarchies, the components will be more cohesive and the abstractions will be clearer. This makes it possible to more easily add new operations and objects to a program. The best feature of verb classes is their versatility in expressions. The combinations of verbs and nouns create a rich and extensible set of useful expressions.

Human thinking in natural languages also has separate class hierarchies for nouns and verbs. This resemblance makes the suggested solution easier to understand. Verbs can have the same kind of generalization and specialization hierarchies as nouns. For example, walking is a special kind of a movement. New classes of objects can be defined, each of which can be movable without a need to place the classes in the class hierarchy of "movable objects". When the nouns and verbs have separate derivation trees, the definition of a data type or a function can be found more easily because polymorphic functions are not so spread out along the classes. The polymorphic functions are also easier to clarify because they are typically defined in one place and because the taxonomy of the data types does not include taxonomies of the verbs.

It is possible to implement the presented solution by using special multi-paradigm languages or by using commercial object-oriented languages in a particular way.  This study suggests a pre-compiler, which facilitates the conversion of the pseudo-code and resembles natural language into a normal object-oriented programming language.

Finally, the new approach is easier to adapt for reuse. As the nouns and verbs can be mixed in millions of ways, even in a relatively small vocabulary, the approach is extremely adaptable. The current study of natural languages has focused on speech recognition and computer-aided translation. In addition to verb inheritance the natural language paradigm can inspire other solutions in application design.

# 9 Bibliography

| | |
|---|---|
| Andrews 1993 | Andrews M.: *Visual C++, Object Oriented Programming*, Sams Publishing, 1993. |
| Balda 1990 | Balda M., Gustafson D. A.: Cost Estimation Models for the Reuse and Prototype Software Development Lifecycles, *ACM SIGSOFT Software Engineering*, Vol 15, No. 3, July 1990, pp. 42-50. |
| Basili 1984 | Basili V. and Weiss D. M.: *A* methodology for Collecting Valid Software Engineering Data, *IEEE Transactions on Software Engineering,* Vol SE-10 no 6 (November 1984): 728-738. |
| Bassett 1997 | Bassett P. G.: *Framing Software Reuse*, Prentice-Hall, 1997. |
| Baumert 1992 | Baumert, J. H., McWhinney, M. S. *Software Measures and the Capability Maturity Model*, Software Engineering Institute at Carnegie Mellon University, CMU/SEI-92-TR-25, 1992. |
| Beck 2000 | Beck, K.: *Extreme Programming Explained,* Addison-Wesley, 2000. |
| Boehm 1981 | Boehm B. W.: *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs NJ, 1981. |
| Boehm 1988 | Boehm B. W.: A Spiral Model of Software Development and Enhancement, *Computer,* May 61-72, 1988. |
| Boehm 2000 | Boehm B. W., Abts, C., Brown, A. W., Chulani, S., Clark, B. K., Horowitz, E., Madachy, R., Reifer, D., Steece, B.: *Software Cost Estimation with COCOMO II*, Prentice-Hall,, 2000. |
| Booch 1991 | Booch G.: *Object Oriented Design with Applications,* Benjamin/Cummings Publishing Company, 1991. |
| Booch 1996 | Booch G.: *Object Solutions, Managing the Object Oriented Project,* Addison-Wesley Publishing, 1996. |

| | |
|---|---|
| Booch 1997 | Booch G, Jacobson I., Rumbaugh J.: *UML 1.0 reference*, Rational Corporation, 1997. |
| Booch 1999 | Booch G, Rumbaugh J., Jacobson I.: The *Unified Modelling Language User Guide,* Addison-Wesley, 1999. |
| Borland 1999 | *Borland JBuilder for Windows 95, Windows 98 & Windows NT Quick Start,* Inprice Corporation, 100 Enterprice Way, Scotts Valley, CA, 1999. |
| Briand 2001 | Briand, L.C., Wust: Modelling Development Effort in Object-Oriented Systems Using Design Properties. *IEEE Transactions on Software Engineering,* Vol. 27, No. 11, November 2001, pp. 963-986. |
| Brooks 1995 | Brooks F. P. Jr.: *The Mythical Man-Month,* Anniversary Edition, Addison-Wesley Publishing Company, 1995. |
| Brown 2000 | Brown A. W.: *Large-scale Component-Based Development*, Prentice-Hall, 2000. |
| Bruckhaus 1996 | Bruckhaus T., Madhavji N.H., Janssen I., Henshaw J.: The Impact of Tools on Software Productivity*, IEEE Computer*, September 1996. |
| Budd 1995 | Budd, T.A.: *Multiparadigm Programming in Leda*, Addison-Wesley Publishing Co., 1995. |
| Cant 1994 | Cant S. N., Henderson-Sellers, B,  Jeffery D. R., Application of cognitive complexity metrics to object-oriented programs, *Journal Of Object Oriented Programming*, 7(4),52-63, 1994. |
| Cantor 1998 | Cantor,M. R.: *Object-Oriented Project Management with UML*, Wiley Computer Publishing, 1998. |
| Carroll 1995 | Carroll M. D., Ellis M. A.: *Designing and coding reusable C++,* Addison-Wesley Publishing Company, 1995. |
| Cheesman 2001 | Cheesman, J.: *UML Components,* Addison-Wesley Publishing Company,  2001. |

| | |
|---|---|
| Chidamber 1991 | Chidamber S. R., Kemerer C.F. : Towards a Metrics Suite for Object-Oriented Design, Proceedings of OOPSLA '91, *ACM SIGPLAN Notices* 26(11), pp. 197-211, 1991. |
| CMMI 2002 | CMMI 2002. CMMI-SE/SW/IPPD/SS, CMMI for Systems Engineering/Software Engineering/Integrated Product and Process Development/Supplier Sourcing, Version 1.1, Continuous Representation, Software Engineering Institute, CMU/SEI-2002-TR-011, Pittsburg PA, 2002. |
| Coad 1990 | Coad P., Yourdon E.: *Object-Oriented Analysis,* Yourdon Press, 1990. |
| Coleman 2000 | Coleman, G., O'Connor, R. : Power to the programmer using measurement to optimise the software process at the individual level, , In Katarina Maxwell, Rob Kusters, Erik van Veenendaal, Adrian Cowderoy (eds), Proceedings of *the combined 11th European Software Control and Metrics Conference and the 3rd SCOPE Conference on Software Product Quality*, page(s) 181-190, Shaker Publishing, April 2000. |
| Constantine 1995 | Constantine L. L.: *Constantine on Peopleware*, Prentice-Hall, 1995. |
| Conte 1986 | Conte S. D., Dunsmore H. E., Shen V. Y.: *Software Engineering Metrics and Models,* Benjamin/Cummings, Menlo Park CA. |
| Coplien 1999 | Coplien, J. O.: *Multi-Paradigm Design for C++,* Addison-Wesley Publishing Co.,1999. |
| Curtis 1997 | Curtis, P.: *LambdaMOO Programmer's Manual,* http://mirrors.ccs.neu.edu/MOO/html/ProgrammersManual_toc.html, 1997. |
| Donaldson 1997 | Donaldson, S. E., Sielgel, S. G.: *Cultivating Successful Software Development,* Prentice Hall, 1997. |
| Dreger 1992 | Dreger B.: *Function Point Analysis*, Prentice-Hall, 1992. |

| | |
|---|---|
| Fayed 1997 | Fayed M. E.: A Lecture in *European Conference of Object-Oriented Programming,* Jyväskylä Finland, June 1997. |
| Fenton 1997 | Fenton, N.E., Pfeeger, S.L.: *Software Metrics - A rigorous Approach*, 2nd ed., International Thomson Computer Press, London, 1997. |
| Florijn 1997 | Florijn G., Meyers M., van Winsen P.: Tool Support for Object-Oriented Patterns, Lecture Notes in Computer Science Vol 1241: *ECOOP'97*, Springer-Verlag 1997. |
| Fowler 2000 | Fowler, M.: *Refactoring - Improving the Design of Existing Code*, Addison-Wesley, 2000. |
| Fromkin 1993 | Fromkin V., Rodman R.: *An Introduction to Language*, Harcort Brace College Publishers, 1993. |
| Gamma 1995 | Gamma E, Helm R., Johnsson R., Vlissides J.: *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing, 1995. |
| Garlan 1995 | Garlan D., Allen R., Ockerbloom J.: Architectural Mismatch: Why Reuse is so Hard*, IEEE Software*, vol 12, no 6, pp. 17-26, Nov. 1995. |
| Goldberg 1995 | Goldberg A., Rubin K. S.: *Succeeding with Objects, Decision Frameworks for Project Management*, Addison-Wesley Publishing, 1995. |
| Graham 1994 | Graham I.: *Migrating to Object Technology*, Addison-Wesley Publishing, 1994. |
| Haikala 1998 | Haikala I., Märijärvi J.: *Ohjelmistotuotanto*, Suomen Atk-kustannus, 1998, pp. 216-223. |
| Hakkarainen 1993 | Hakkarainen J., Laamanen P., Penttonen M., Rask R.: Ohjelmistoprojektien työmäärän arvionnissa käytettävät päätöstekijät, *Joensuun Yliopisto*, Joensuu, 1993. |
| Halstead 1977 | Halstead M. H.: *Elements of Software Science*, Elsevier/North-Holland, New York, 1977. |

| | |
|---|---|
| Heineman 2001 | Heineman, G.T., Council, W.T (eds.).: *Component-Based Software Engineering,* Addison-Wesley, 2001. |
| Henderson-Sellers 1996 | Henderson-Sellers, B.: *Object-oriented metrics*, Prentice-Hall, 1996. |
| Henry 1981 | Henry S., Kafura D.: Software structure metrics based on information flow*, IEEE Trans. Software Eng*., 7(5), 510-518, 1981. |
| Herzum 2000 | Herzum,P., Sims,O.: *Business Component Factory*, Wiley Computer Publishing, 2000. |
| Hohmann 1997 | Hohmann L.: *Journey of the Software Professional*, Prentice-Hall, 1997. |
| Holzmüller 1998 | Holzmüller B.: On Polymorphic Type Systems for Imperative Programming Languages: An Approach using Sets of Types and Subprograms, Workshop Reader of the *12th European Conference of Object-Oriented Programming*, p. 31, Springer 1998. |
| Humphrey 1995 | Humphrey W. S., *A discipline for Software Engineering*, Addison-Wesley Publishing, 1995. |
| Humphrey 1997 | Humphrey W. S., *Managing Technical People*, Addison-Wesley Publishing, 1997. |
| Hyttinen 1987 | Hyttinen R., Tuttujew J.: *Omakotirakentajan kustannustieto*, Rakentajain Kustannus Oy, 1987. |
| Jacobson 1997 | Jacobson I., Griss M., Jonsson P.: *Software Reuse,* ACM Press, 1997. |
| Jacobson 1999 | Jacobson I., Booch G., Rumbaugh J.: *The Unified Software Development Process,* Addison-Wesley, 1999. |
| Jones 1994 | Jones C.: *Assessment and Control of Software Risks*, Prentice-Hall, 1994. |
| Järvinen 1999 | Järvinen P.: *On research methods,* Opinpaja Oy, Tampere, Finland, 1999. |

| | |
|---|---|
| Järvinen 2000 | Järvinen, J.: *Measurement based continuous assessment of software engineering process*, VTT Publications, 426, Finland, 2000. |
| Kendall 1987 | Kendall A.: *Introduction to Systems Analysis and Design, A Structured Approach,* Allyn and Bacon, 1987. |
| Keuffel 1994 | Keuffel W.: Function Points: Pro and Con, *Software Development*, November 1994. |
| Keuffel 1995 | Keuffel W.: Don't Fench Me In, *Software Development,* February 1995. |
| Kilpi 1998 | Kilpi T.: *Applying product management approach to software developmeng: An SME perspective,* University of Oulu, March 1998. |
| Knudsen 1993 | Knudsen, J. L., Löfgren, O. L., Magnusson, B.: *Object-Oriented Environments - The Mjölner Approach*, Prentice Hall, 1993. |
| Koenig 1996 | Koenig A, Moo B.: *Ruminations on C++,* Addison-Wesley Longman inc., 1996. |
| Kolewe 1993 | Kolewe R.: *Metrics in Object-Oriented Design and Programming,* Software Development, October 1993. |
| Komi-Sirviö 2001 | Komi-Sirviö, S., Parviainen, P., Ronkainen, J.: Measurement Automation: Methodological Background and Practical Solutions - A Multiple Case Study, Proceedings *Seventh International Software Metrics Symposium METRICS 2001*, IEEE, 2001, pp. 306-316. |
| Krishnamurthi 1998 | Krishnamurthi S., Felleisen M. and Friedman D. P.: Synthesising Object-Oriented and Functional Design to Promote Re-use, Proceedings of the *12th European Conference of Object-Oriented Programming*, pp. 91-113, Springer 1998. |
| Larman 1998 | Larman C. : *Applying UML and Patterns,* Prentice-Hall PTR, 1998. |

| | |
|---|---|
| Lavazza 2000 | Lavazz, L. : Providing Automated Support for the GQM Measurement Process. *IEEE Software*, Vol. 17, No. 3, May/June 2000, pp. 56-62. |
| Lawson 1974 | Lawson C. L. and Hanson R. J.: *Solving Least Squares Problems,* Prentice-Hall, 1974. |
| Lorentz 1994 | Lorenz M., Kidd J.: *Object-oriented software metrics*, Prentice-Hall, 1994. |
| Maxwell 2002 | Maxwell, K.D.: *Applied Statistics for Software Managers,* Prentice Hall PTR, 2002. |
| McCabe 1976 | McCabe T. J.: A complexity measure*, IEEE Trans. Software Eng*.,2(4), 308-320, 1977. |
| McConnell 1996 | McConnell S.: *Rapid Development*, Microsoft Press 1996. |
| McConnell 1998 | McConnell S.: *Software Project Survival Guide*, Microsoft Press 1998. |
| Meli 2000 | Meli R., Abran A., Ho, V. T., Oligny, S.: On the applicability of COSMIC-FFP for measuring sofware throughout its life cycle*,* In Maxwell,K., Kusters, R., van Veenendaal, E., Cowderoy, A. (eds), Proceedings of *the combined 11th European Software Control and Metrics Conference and the 3rd SCOPE Conference on Software Product Quality*, page(s) 289-297, Shaker Publishing, April 2000. |
| Meli 2001 | Meli R.: Measuring Change Requests to support effective project management practices, In Maxwell, K., Oligny, S., Kusters, R. and van Veenendaal, E. (eds), Proceedings of the *12th European Software Control and Metrics Conference,* page(s) 25-33, Shaker Publishing, April 2001. |
| Metzger 1996 | Metzger P., Probe J.: *Managing a Programming Project*, Processes and People, Prentice.Hall, 1996. |
| Meyer 1997 | Meyer B.: *Object-oriented Software Construction 2nd Ed.* , Prentice-Hall, 1997. |

| | |
|---|---|
| Microsoft 1993 | *Microsoft Visual C++ Class Library Reference, Volume I*, Microsoft Corporation:, 1993. |
| Mikhajlov, 1997 | Mikhajlov, L., Sekerinski, E.:The Fragile Base Class Problem and Its Impact on Component Systems, In Bosh, J., Mitchell, S. (eds), *Object-Oriented Technology ECOOP'97* Workshop Reader, Springer-Verlag, 1997. |
| Nageswaran 2001 | Nageswaran, S.: Test Effort Estimation Using Use Case Points*, Quality Week,* June, 2001. |
| Naur 1975 | Naur, P.: Programming Languages, Natural Languages and Mathematics*, Communications of the ACM,* Vol 18, Nbr. 12, pp. 676-683, 1975. |
| Ochs 2001 | Ochs, M., Pfahl, D., Chrobok-Diening, G., Nothhelfer-Kolb, B.:A Method for Efficient Measurement-based COTS Assessment and Selection - Method Description and Evaluation Results, Proceedings *Seventh International Software Metrics Symposium METRICS 2001*, IEEE, 2001, pp. 285-296. |
| Ojala 1989 | Ojala I., Laurikainen M., Seppänen T., Lyytikäinen K., Tyrväinen H.: *Talonrakennennustöiden menekit,* Rakennuskirja, 1989. |
| Oosting 2000 | Oosting, K.: The human factor: predictable or unpredictable, In Katarina Maxwell, Rob Kusters, Erik van Veenendaal, Adrian Cowderoy (eds), Proceedings of *the combined 11th European Software Control and Metrics Conference and the 3rd SCOPE Conference on Software Product Quality*, page(s) 181-190, Shaker Publishing, April 2000. |
| Orfali 1996 | Orfali R., Harkey D., Edwards J. : *The essential Distributed Objects Survival Guide,* John Wiley & Sons inc, 1996. |
| Paulk 1993 | Paulk M. C., Curtis B., Chrissis M. B., Weber C. V.: *Capability Maturity Model for Software, Version 1.1,* Software Engineering Institute and Carnegie Mellon University Technical Report CMU/SEI-93-TR-24, February 1993. |

| | |
|---|---|
| Poulin 1997 | Poulin J. S.: *Measuring Software Reuse,* Principles, Practices and Economic Models, Addison-Wesley Publishing, 1997. |
| Poulin 2001 | Poulin, J.S., Measurement and Metrics for Software Components in Heineman, G.T., Council, W.T.(eds.): *Component-Based Software Engineering*, Addison-Wesley, 2000, pp. 435-466. |
| Prieto-Diaz 1987 | Prieto-Diaz R. Freeman P.: Classifying Software for reusability*, IEEE Software*, Vol 4 No 1, January 1987, pp. 6-16. |
| Putkonen 1994 | Putkonen A.: *A Methodology for Supporting Analysis, Design and Maintenance of Object-oriented Systems,* Kuopio University Publications C, Natural and Environmental Sciences 19, 1994. |
| Putnam 1992 | Putnam L. H.: Ware Myers, *Measures for Excellence*, Prentice-Hall, 1992. |
| Rask 1992 | Rask R.: *Automated estimation of software size during requirements specification phase,* University of Joensuu, Publications in sciences 28, 1992. |
| Reo 2000 | Reo, D. A.: Focusing on the raw material of software development- It's a people issue! *,* In Katarina Maxwell, Rob Kusters, Erik van Veenendaal, Adrian Cowderoy (eds), Proceedings of *the combined 11th European Software Control and Metrics Conference and the 3rd SCOPE Conference on Software Product Quality*, page(s) 181-190, Shaker Publishing, April 2000. |
| Robbins 1998 | Robbins, S.P.: *Organisational Behavior*, Prentice Hall, 1998. |
| Ropponen 2000 | Ropponen, J., Lyytinen, K.: Components of Software Development Risks: How to Address Them? A Project Manager Survey*, IEEE Transactions on Software Engineering*, Vol 26, No.2, February 2000, pp. 98-112. |
| Rumbaugh 1991 | Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W.: *Object-Oriented Modelling and Design,* Prentice-Hall, Englewood Cliffs, New Jersey, 1991. |

| | |
|---|---|
| Schmietendorf 2001 | Schmietendorf A., Dumke, R., Foltin E.: : Risk-driven effort estimation of tasks within the software performance engineering, In Katarina Maxwell, Rob Kusters, Erik van Veenendaal, Adrian Cowderoy (eds), Proceedings of *the 12th European Software Control and Metrics Conference*, page(s) 87-95, Shaker Publishing, April 2001. |
| Sedigh-Ali 2002 | Sedigh-Ali, S., Ghafoor, A., Paul, R.A. : Metrics-Guided Quality Management for Component-Based Software Systems, Proceedings of the *25th Annual International Computer Software And Applications Conference ( COMPSAC'01),* IEEE, 2002. |
| Selby 1989 | Selby, R.W.: Quantitative Studies of Software Reuse, *Software Reusability, Vol. 2*, eds. Biggerstaff, T.J. and Perlis, A.J., Addison-Wesley, 1989. |
| Shepperd 2001 | Shepperd, M., Cartwright, M.: Predicting with Sparse Data, Proceedings *Seventh International Software Metrics Symposium METRICS 2001*, IEEE, 2001, pp. 28-39. |
| Shlaer 1992 | Shlaer S., Mellor S.J.: *Object Lifecycles Modelling the World in States,* Yourdon Press, Prentice-Hall, 1992. |
| Smith 1997 | Smith, R. Parrish, A. and Hale, J.: Component Based Software Development: Parameters Influencing Cost Estimation, Proceedings of the *Twenty-Second Annual Software Engineering Workshop*, Flight Dynamics Division, Software Engineering Laboratory, NASA/Goddard Space Flight Center, December 1997. |
| Smith 2001 | Smith, R., Hale, J. and Parrish, A.: An Empirical Study Using Task Assignment Patterns to Improve the Accuracy of Software Effort Estimation, *IEEE Transactions on Software Engineering,* Vol. 27, No. 3, March 2001. |
| Solberg 2001 | Solberg, H., Dahl, K.M.: *COTS Software Evaluation and Integration issues*, http:// www.idi.ntnw.no/ grupper/su/sif8094-reports/p14.pdf, November 2001. |
| Solingen 2001 | van Solingen, R., Berghout, E.: Integrating Goal-Oriented Measurement in Industrial Software |

| | |
|---|---|
| | Engineering: Industrial Experiences with Additions to the Goal/Question/Metric Method(GQM), Proceedings *Seventh International Software Metrics Symposium METRICS 2001*, IEEE, 2001, pp. 246-258. |
| Symons 1991 | Symons C.: *Software Sizing and Estimating: Mk II FPA ( Function Point Analysis),* John Wiley and Sons, 1991. |
| Szyperski 1998 | Szyperski, C., *Component Software - Beyond Object-Oriented Programming*, ACM Press, New York, 1998. |
| Toffolon 2000 | Toffolon, C., Dakhli, S.: The Cost of Co-ordination in Software Engineering, In Katarina Maxwell, Rob Kusters, Erik van Veenendaal, Adrian Cowderoy (eds), Proceedings of *the combined 11th European Software Control and Metrics Conference and the 3rd SCOPE Conference on Software Product Quality*, page(s) 171-179, Shaker Publishing, April 2000. |
| Tracz 1995 | Tracz W.: *Confessions of a Used Program Salesman,* Addison-Wesley Publishing, 1995. |
| Virtanen 1998a | Virtanen P.: Object Component Process Metrics – A new approach to software project estimation*,* Proceedings of the *21th Information Systems Research seminar in Scandinavia,* pp. 919-937, Department of Computer Science, Aalborg University, 1998. |
| Virtanen 1998b | Virtanen P.: An Evaluation of the Benefits of Object-oriented Methods in Software Development Processes, Workshop Reader of the *12th European Conference of Object-Oriented Programming*, pp. 35-36, Springer 1998. |
| Virtanen 1999 | Virtanen P.: Extended Reuse with Verb Inheritance - A New Approach to Software Construction and Reuse, In Gerhardt F. and Benedicenti L. and Ernst E. (eds), Position Papers from the *8th Workshop for PhD Students in Object-Oriented Systems*, page(s) 147-154, Department of Computer Science, University of Århus, Denmark, 1999. |
| Virtanen 1999b | Virtanen P.: Adaptability – The enabler of reuse, Proceedings of the *22th Information Systems Research* |

*Seminar in Scandinavia,* pp. 353-370, Department of Computer Science and Information Systems, University of Jyväskylä, 1999.

Virtanen 2000 Virtanen, P. *Issues of Improving Object-Oriented Software Development,* University of Turku, January 2000.

Virtanen 2000b Virtanen P.: Component Reuse Metrics - Assessing Human Effects, In Maxwell,K., Kusters, R., van Veenendaal, E., Cowderoy, A. (eds), Proceedings of the combined *11th European Software Control and Metrics Conference and the 3rd SCOPE Conference on Software Product Quality*, page(s) 171-179, Shaker Publishing, April 2000.

Virtanen 2000c Virtanen, P.: Verb Classes - Design for Reuse, In Helmut Thoma, Heinrich C. Mayer, Alptekin Erkollar (eds), *6th International Conference on Re-Technologies for Information Systems* - Preparing to E-business, page(s) 155-166, Österreichishe Computer Gesellschaft, March 2000.

Virtanen 2001 Virtanen P., Empirical Study Evaluating Component Reuse Metrics , In Maxwell, K., Oligny, S., Kusters, R. and van Veenendaal, E. (eds), Proceedings of the *12th European Software Control and Metrics Conference*, page(s) 125-135, Shaker Publishing, April 2001.

Wilde 1992 Wilde N., Huitt R.: Maintenance Support for Object-Oriented Programs, *IEEE Transactions on Software Engineering,* Vol. 18, No. 12, Dec 1992 pp. 1038-1044.

Wirfs-Brock 1990 Wirfs-Brock R., Wilkerson B., Wiener L.: *Designing Object-Oriented Software,* Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

# Appendices

## Appendix A: Estimation forms and survey results

The tables in the appendices A, B and C gather the results of the survey of the empirical study of CRM (see chapter 3, page 49). The survey was sent to all of the project managers who were members of the Finnish Information Processing Association. Appendix A contains the survey results about the questionnaires of CRM. The questions of CRM-estimation forms are also the same (Table 28 - Table 33) but in estimation forms only the forthcoming project is considered. In the survey the respondents estimated the usual importance of the factor. The answers are classified by a small number (Very large =5, large = 4, medium=3, small = 2, very small = 1). The weight is the average of the answers. The results of the survey can be used as default weights in Equation 3, page 26.

Table 28. Process effect questionnaire; factors of the process effect.

| | Estimate the effect of the following methods and tools to the development effort (%) | Very large | Large | Medium | Small | Very small | Don't know | I don't use | Weight ($w_j$) |
|---|---|---|---|---|---|---|---|---|---|
| Factors | Programming language | 7 | 31 | 34 | 22 | 3 | 0 | 1 | 4.13 |
| | Development environment | 10 | 31 | 37 | 10 | 1 | 0 | 9 | 4.45 |
| | Database management tools | 9 | 18 | 43 | 22 | 3 | 0 | 4 | 4.06 |
| | Tools for analysis and design | 4 | 28 | 37 | 22 | 4 | 0 | 3 | 4.03 |
| | Project management tools | 1 | 13 | 33 | 37 | 10 | 3 | 1 | 3.55 |
| | Configuration management tools | 3 | 25 | 27 | 25 | 4 | 4 | 10 | 4.07 |
| | Documentation tools | 4 | 16 | 37 | 34 | 7 | 0 | 0 | 3.71 |
| | Tools for testing | 7 | 36 | 28 | 19 | 4 | 1 | 3 | 4.22 |
| | Method of quality assurance | 10 | 24 | 43 | 16 | 3 | 0 | 3 | 4.20 |
| | Special quality requirements | 15 | 36 | 30 | 13 | 3 | 0 | 3 | 4.44 |
| | Project management methods | 9 | 30 | 36 | 21 | 3 | 1 | 0 | 4.16 |
| | Testing methods | 10 | 45 | 34 | 9 | 1 | 0 | 0 | 4.47 |
| | Documentation standards | 3 | 21 | 37 | 36 | 3 | 0 | 0 | 3.79 |
| | Partitioning of the project | 7 | 40 | 27 | 24 | 1 | 0 | 0 | 4.22 |
| | Finding and assessing components | 3 | 12 | 34 | 30 | 4 | 6 | 10 | 3.88 |
| | Assuring of generality | 7 | 27 | 34 | 18 | 3 | 3 | 7 | 4.25 |

Table 29. Factors of project change effect.

| Factors influencing new unexpected features during a project (%) | Always | Mostly | Often | Rarely | Never | Don't know | Weight |
|---|---|---|---|---|---|---|---|
| Error in requirement specification | 6 | 42 | 33 | 15 | 3 | 1 | 3.33 |
| Inaccuracy of analysis | 7 | 43 | 42 | 7 | 0 | 0 | 3.51 |
| Inaccuracy of design | 3 | 22 | 57 | 16 | 0 | 1 | 3.12 |
| Partition of the project | 0 | 6 | 31 | 54 | 6 | 3 | 2.38 |
| Views of the customer's management | 0 | 16 | 42 | 34 | 6 | 1 | 2.70 |
| Views of the IT-management | 0 | 6 | 27 | 58 | 6 | 3 | 2.34 |
| End user views | 9 | 46 | 36 | 4 | 4 | 0 | 3.51 |
| Programmers' views | 4 | 12 | 43 | 36 | 4 | 0 | 2.76 |
| Views of the project manager | 4 | 15 | 40 | 40 | 0 | 0 | 2.84 |
| Good idea invented during the project | 3 | 24 | 55 | 16 | 0 | 1 | 3.14 |
| Technological surprises | 3 | 19 | 40 | 34 | 3 | 0 | 2.85 |
| Change control | 6 | 19 | 24 | 45 | 4 | 1 | 2.77 |
| Commercial factors | 3 | 12 | 22 | 49 | 10 | 3 | 2.46 |
| Inaccurate contract | 0 | 25 | 33 | 24 | 12 | 6 | 2.76 |
| Invoicing method (by hours, contract price) | 1 | 7 | 24 | 30 | 30 | 7 | 2.15 |

Table 30. Factors of the team effect.

| What is the influence of the following factors of team work in your projects (%)? | Very large | Large | Medium | Small | Very small | Don't know | Weight |
|---|---|---|---|---|---|---|---|
| Team structure and size * | | | | | | | |
| Synergy * | | | | | | | |
| Meetings included in the project plan | 18 | 36 | 27 | 13 | 3 | 3 | 3.54 |
| Ad hoc meetings | 21 | 52 | 21 | 3 | 0 | 3 | 3.94 |
| Unnecessary meetings | 4 | 9 | 25 | 36 | 15 | 10 | 2.47 |
| Meeting practice ( being punctual, preparation, … ) | 7 | 46 | 30 | 13 | 0 | 3 | 3.49 |
| Travel time | 0 | 1 | 36 | 42 | 16 | 4 | 2.23 |
| Slack time due to travelling | 0 | 4 | 24 | 46 | 19 | 6 | 2.14 |
| Phone calls and faxes | 3 | 19 | 30 | 33 | 10 | 4 | 2.70 |
| Writing email | 12 | 27 | 34 | 19 | 4 | 3 | 3.23 |
| Reading email | 12 | 31 | 36 | 13 | 4 | 3 | 3.34 |
| Personal supervision of work | 18 | 37 | 33 | 7 | 1 | 3 | 3.65 |
| Discussions with the users | 28 | 46 | 13 | 7 | 0 | 4 | 4.00 |
| Discussions with customer's management | 27 | 40 | 24 | 6 | 0 | 3 | 3.91 |
| Writing minutes and other documents | 10 | 25 | 42 | 16 | 3 | 3 | 3.25 |
| Reading minutes and other documents | 7 | 34 | 33 | 21 | 1 | 3 | 3.26 |
| Disputes about the objectives | 24 | 43 | 18 | 10 | 0 | 4 | 3.84 |
| Disputes about the working methods | 22 | 39 | 22 | 12 | 0 | 4 | 3.75 |
| Interruptions of work | 19 | 37 | 27 | 12 | 1 | 3 | 3.63 |
| Disturbances in the information flow | 34 | 37 | 16 | 7 | 0 | 4 | 4.03 |

*** Not included in the survey results and analysis but should be used in CRM.**

Table 31. Factors of the risk effect.

| A risk is an incident, which has a probability of influencing a project. If a following risk comes true, how large is its expected influence (%)? | Very large | Large | Medium | Small | Very small | Don't know | Weight |
|---|---|---|---|---|---|---|---|
| Changes in personnel | 34 | 48 | 16 | 0 | 0 | 1 | 4.18 |
| Sickness | 12 | 37 | 34 | 13 | 1 | 1 | 3.45 |
| Technical disturbances (black outs, equipment failures …) | 4 | 12 | 30 | 34 | 18 | 1 | 2.50 |
| Failures in technology | 24 | 39 | 16 | 16 | 3 | 1 | 3.65 |
| Unexpectedly difficult software bugs | 10 | 49 | 24 | 12 | 3 | 1 | 3.53 |
| Errors due to carelessness | 7 | 25 | 46 | 13 | 4 | 3 | 3.18 |
| Sabotage (viruses, hacking …) | 13 | 9 | 13 | 24 | 31 | 9 | 2.44 |
| Unpunctuality of the contractor, subcontractor or customer | 24 | 43 | 22 | 9 | 0 | 1 | 3.83 |
| Organisational changes | 1 | 30 | 43 | 19 | 4 | 1 | 3.05 |
| Failures in subcontracting and purchasing | 15 | 43 | 18 | 13 | 4 | 6 | 3.54 |
| Disputes | 10 | 25 | 25 | 33 | 4 | 1 | 3.05 |
| Estimation errors | 19 | 46 | 27 | 6 | 0 | 1 | 3.80 |
| Economic risks | 3 | 27 | 39 | 30 | 0 | 1 | 3.03 |
| Juridical risks | 3 | 6 | 21 | 43 | 21 | 6 | 2.22 |
| Preparation for the risks * | | | | | | | |

* **Not included in the survey results and analysis but should be used in CRM**.

Table 32. Factors of the skill effect.

| What is the influence of the following factors of the skill of software development (%)? | Very large | Large | Medium | Small | Very small | Don't know | Weight |
|---|---|---|---|---|---|---|---|
| Education | 12 | 21 | 42 | 21 | 4 | 0 | 3.15 |
| Courses | 7 | 16 | 52 | 22 | 1 | 0 | 3.06 |
| Length of experience | 15 | 42 | 43 | 0 | 0 | 0 | 3.72 |
| Quality of experience | 46 | 39 | 15 | 0 | 0 | 0 | 4.31 |
| Familiarity with the application area | 39 | 45 | 16 | 0 | 0 | 0 | 4.22 |
| Experience of team work | 4 | 27 | 46 | 18 | 3 | 1 | 3.12 |
| Familiarity with the program (to be maintained ) | 43 | 33 | 21 | 1 | 0 | 1 | 4.20 |
| Knowledge of methods and tools | 27 | 51 | 16 | 4 | 0 | 1 | 4.02 |
| Personality (intelligence, emotional intelligence, sense of responsibility, diligence) | 42 | 34 | 21 | 1 | 1 | 0 | 4.13 |

Table 33. Factors of the motivation effect.

| Estimate the influence of the following motivation factors on increasing or decreasing the productivity of software development. Assess your own motivation factors. | Very large | Large | Medium | Small | Very small | Don't know | Weight for Project managers |
|---|---|---|---|---|---|---|---|
| Challenge or lack of it | 24 | 67 | 6 | 1 | 0 | 1 | 4.15 |
| Ambition | 9 | 39 | 36 | 15 | 1 | 0 | 3.39 |
| Possibility to accomplish something, achieve results | 22 | 49 | 25 | 3 | 0 | 0 | 3.91 |
| Possibility for initiative and independence | 18 | 58 | 22 | 1 | 0 | 0 | 3.93 |
| Good/bad leadership | 25 | 25 | 37 | 9 | 1 | 1 | 3.65 |
| Too large, appropriate, too small pressure | 3 | 40 | 40 | 15 | 0 | 1 | 3.32 |
| Possibility for career | 3 | 22 | 48 | 24 | 3 | 0 | 2.99 |
| Good/bad relationships especially within the team | 27 | 43 | 24 | 4 | 1 | 0 | 3.90 |
| Appreciation, respect | 16 | 46 | 28 | 9 | 0 | 0 | 3.70 |
| Salary and benefits | 9 | 24 | 49 | 15 | 3 | 0 | 3.21 |
| Responsibility | 19 | 46 | 28 | 6 | 0 | 0 | 3.79 |
| Being noticed or lack of it | 13 | 37 | 37 | 9 | 1 | 1 | 3.53 |
| Possibility to develop oneself , to learn new things | 25 | 48 | 25 | 1 | 0 | 0 | 3.97 |
| Working conditions | 7 | 31 | 43 | 16 | 0 | 1 | 3.30 |
| Interesting work, the work itself | 28 | 58 | 13 | 0 | 0 | 0 | 4.15 |

Table 34. Factors of motivation effect of the team.

| Estimate the influence of the following motivation factors on increasing or decreasing the productivity of software development. Assess the motivation factors of your team(%) | Very large | Large | Medium | Small | Very small | Don't know | Weight for team members |
|---|---|---|---|---|---|---|---|
| Challenge or lack of it | 16 | 55 | 22 | 3 | 0 | 3 | 3.88 |
| Ambition | 9 | 36 | 40 | 12 | 0 | 3 | 3.43 |
| Possibility to accomplish something, achieve results | 9 | 43 | 40 | 4 | 0 | 3 | 3.58 |
| Possibility for initiative and independence | 9 | 31 | 48 | 9 | 0 | 3 | 3.42 |
| Good/bad leadership | 33 | 37 | 24 | 1 | 1 | 3 | 4.02 |
| Too large, appropriate, too small pressure | 12 | 43 | 33 | 7 | 0 | 4 | 3.63 |
| Possibility for career | 12 | 24 | 40 | 19 | 1 | 3 | 3.26 |
| Good/bad relationships especially within the team | 27 | 42 | 24 | 1 | 1 | 4 | 3.95 |
| Appreciation, respect | 22 | 37 | 31 | 6 | 0 | 3 | 3.78 |
| Salary and benefits | 13 | 31 | 40 | 12 | 0 | 3 | 3.48 |
| Responsibility | 9 | 27 | 46 | 13 | 1 | 3 | 3.29 |
| Being noticed or lack of it | 19 | 42 | 30 | 4 | 1 | 3 | 3.75 |
| Possibility to develop oneself, to learn new things | 21 | 46 | 27 | 3 | 0 | 3 | 3.88 |
| Working conditions | 10 | 34 | 36 | 15 | 0 | 4 | 3.42 |
| Interesting work, the work itself | 18 | 55 | 16 | 4 | 0 | 6 | 3.92 |

Table 35. Distribution of effort.

| What is the distribution of software development effort in a typical project? | Avg | Conf. |
|---|---|---|
| Design and assembly of the components | 34.54 | 3.54 |
| Additional effort due to added features | 20.62 | 2.00 |
| Training and study during the work | 10.53 | 1.07 |
| Slack due to lack of motivation | 10.13 | 1.53 |
| Team work | 13.01 | 1.85 |
| Risks | 11.17 | 1.32 |
|  | 100.00 |  |

Table 36. Distribution of effort because of component itself.

| How large is the distribution of effort due to the component according to your experience? | Average | Confid. |
|---|---|---|
| less than 2 h (effort is 18 - 22 h ) | 16.55 | 3.74 |
| 2 - 5 h ( effort is  15 - 25 h ) | 26.22 | 3.58 |
| 5 - 10 h ( effort is  10 - 30 h ) | 30.34 | 3.84 |
| 10 - 20 h (effort is  0 - 40 h ) | 17.33 | 2.56 |
| more than 20 h ( effort is more than 40 h ) | 9.56 | 2.76 |
| Total | 100.00 |  |

Table 37. Amount of additional features.

| The amount of new features added during the project is (%) | Always | Mostly | Often | Rarely | Never | Don't know |
|---|---|---|---|---|---|---|
| less than 10 % final total effort | 1 | 34 | 22 | 30 | 10 | 1 |
| 10 - 20 % final total effort | 1 | 42 | 34 | 19 | 1 | 1 |
| 20 - 50 % final total effort | 1 | 21 | 21 | 40 | 13 | 3 |
| more than 50 % final total effort | 0 | 1 | 7 | 37 | 51 | 3 |

Table 38. Amount of cancelled features.

| The amount of cancelled features during a project (%) | Always | Mostly | Often | Rarely | Never | Don't know |
|---|---|---|---|---|---|---|
| less than 10 % of planned total effort | 9 | 37 | 36 | 12 | 1 | 4 |
| 10 - 20 % of planned total effort | 0 | 15 | 27 | 28 | 27 | 3 |
| 20 - 50 % of planned total effort | 0 | 0 | 9 | 36 | 51 | 4 |
| more than 50 % of planned total effort | 0 | 0 | 1 | 13 | 81 | 4 |

Table 39. Distribution of effort due to skill differences.

| 1. The person has experience in comparable tasks, but needs a small amount of advice in accomplishing the task e.g. An experienced analyst who has just been moved to the project. | Always | Mostly | Often | Rarely | Never | Don't know |
|---|---|---|---|---|---|---|
| less than 2 h | 9 | 33 | 19 | 24 | 7 | 7 |
| 2 - 5 h | 3 | 36 | 39 | 13 | 4 | 4 |
| 5 - 10 h | 0 | 13 | 30 | 31 | 19 | 6 |
| more than 10 h | 0 | 4 | 3 | 37 | 46 | 9 |
| 2. The person has the necessary education and is able to accomplish simple tasks e.g. An analyst, who has few years of experience and has just started in the corporation. | Always | Mostly | Often | Rarely | Never | Don't know |
| less than 5 h | 3 | 13 | 25 | 30 | 21 | 7 |
| 5 - 10 h | 6 | 28 | 45 | 10 | 6 | 4 |
| 10 - 20 h | 3 | 36 | 33 | 18 | 4 | 6 |
| more than 20 h | 0 | 9 | 25 | 31 | 25 | 9 |
| 3. The person knows the basics of the area and has been in a short course before starting e.g. A Computer Science-student in his/her first job. | Always | Mostly | Often | Rarely | Never | Don't know |
| less than 10 h | 3 | 6 | 31 | 24 | 28 | 7 |
| 10 - 20 h | 3 | 30 | 31 | 24 | 7 | 4 |
| 20 - 40 h | 4 | 39 | 31 | 13 | 7 | 4 |
| more than 40 h | 3 | 13 | 13 | 36 | 25 | 9 |

# Appendix B. Background information

The tables in this appendix contain the results of background questions of the survey of the empirical study of CRM (see chapter 3, page 49).

Table 40. Estimation methods and metrics.

| I use the following methods and metrics estimating software development effort(%) | Always | Mostly | Often | Rarely | Never | Don't know |
|---|---|---|---|---|---|---|
| No estimate before the project start | 1 | 1 | 10 | 19 | 66 | 1 |
| Project is adapted to the budget | 12 | 27 | 22 | 33 | 6 | 0 |
| Task based estimation | 25 | 63 | 12 | 0 | 0 | 0 |
| Comparison with similar projects | 13 | 49 | 31 | 6 | 0 | 0 |
| COCOMO | 0 | 0 | 3 | 4 | 46 | 46 |
| Function-point Analysis | 4 | 6 | 28 | 18 | 33 | 10 |
| Comparison with project tracking history | 6 | 22 | 37 | 27 | 6 | 1 |
| Number of subsystems | 4 | 30 | 22 | 19 | 15 | 9 |
| Number of windows, reports and database tables etc. | 12 | 58 | 18 | 7 | 1 | 3 |
| Number of classes | 4 | 13 | 13 | 27 | 31 | 10 |
| Lines of code | 0 | 0 | 27 | 19 | 49 | 4 |
| Number of customers of the component | 1 | 1 | 6 | 18 | 57 | 16 |

Table 41. Process models.

| Phase model used ( % ) | Always | Mostly | Often | Rarely | Never | Don't know | Average |
|---|---|---|---|---|---|---|---|
| Detailed analysis before implementation | 9 | 33 | 25 | 22 | 10 | 0 | 3.07 |
| Multiphase analysis | 7 | 57 | 13 | 19 | 0 | 3 | 3.54 |
| Prototyping for customer requirements | 3 | 37 | 33 | 18 | 7 | 1 | 3.11 |
| Prototyping for technical reasons | 3 | 28 | 37 | 21 | 9 | 1 | 2.95 |
| One-phase design | 3 | 33 | 25 | 34 | 4 | 0 | 2.96 |
| Multiphase design | 6 | 49 | 22 | 19 | 3 | 0 | 3.36 |
| Testing mainly after implementation | 9 | 42 | 18 | 25 | 6 | 0 | 3.22 |
| Multiphase testing | 18 | 48 | 22 | 12 | 0 | 0 | 3.72 |
| One-phase delivery ( Big Bang-model) | 1 | 36 | 25 | 28 | 7 | 1 | 2.95 |
| Multiphase delivery | 4 | 45 | 25 | 22 | 0 | 3 | 3.32 |
| Waterfall model | 1 | 34 | 27 | 16 | 18 | 3 | 2.85 |
| Spiral model | 6 | 37 | 34 | 21 | 1 | 0 | 3.25 |
| Production of versioned software (* omitted question) | 12 | 27 | 24 | 22 | 13 | 1 | 3.02 |
| Total of waterfall models | 5 | 36 | 24 | 25 | 9 | 1 | 2.99 |
| Total of iterative models | 8 | 47 | 24 | 19 | 1 | 1 | 3.40 |

Table 42. Design methods.

| Design methods used (%) | Always | Mostly | Often | Rarely | Never | Don't know | Average |
|---|---|---|---|---|---|---|---|
| Data Flow- analysis | 4 | 25 | 34 | 24 | 6 | 6 | 2.98 |
| Wall board techniques | 10 | 34 | 36 | 15 | 4 | 0 | 3.31 |
| Object-oriented analysis (e. g.: UML, Fusion, Coad ) | 3 | 16 | 12 | 19 | 40 | 9 | 2.15 |
| Entity-Relationship model (* omitted question) | 18 | 24 | 25 | 19 | 9 | 4 | 3.23 |

Table 43. Component technologies.

| Component technologies used (%) | Always | Mostly | Often | Rarely | Never | Don't know | Average |
|---|---|---|---|---|---|---|---|
| ActiveX | 0 | 7 | 21 | 6 | 51 | 15 | 1.82 |
| DCOM | 0 | 3 | 15 | 12 | 55 | 15 | 1.60 |
| CORBA | 0 | 4 | 7 | 15 | 63 | 10 | 1.48 |
| (Enterprise) Java Beans | 0 | 12 | 13 | 12 | 51 | 12 | 1.85 |
| Self-made class libraries | 4 | 30 | 15 | 18 | 25 | 7 | 2.68 |
| Self-made module libraries | 9 | 28 | 31 | 6 | 18 | 7 | 3.05 |
| Acquired class libraries | 4 | 13 | 24 | 21 | 27 | 10 | 2.42 |
| Acquired module libraries | 1 | 16 | 27 | 19 | 27 | 9 | 2.41 |
| I produce software components for sale | 0 | 6 | 6 | 7 | 72 | 9 | 1.41 |
| Application frameworks and design patterns (* omitted question) | 9 | 22 | 19 | 10 | 19 | 19 | 2.89 |

Table 44. Estimation methods and metrics.

| I use the following methods and metrics estimating software development effort(%) | Always | Mostly | Often | Rarely | Never | Don't know | Average |
|---|---|---|---|---|---|---|---|
| No estimate before project start | 1 | 1 | 10 | 19 | 66 | 1 | 1.52 |
| Project is adapted to the budget | 12 | 27 | 22 | 33 | 6 | 0 | 3.06 |
| Task based estimation | 25 | 63 | 12 | 0 | 0 | 0 | 4.13 |
| Comparison with similar projects | 13 | 49 | 31 | 6 | 0 | 0 | 3.70 |
| COCOMO | 0 | 0 | 3 | 4 | 46 | 46 | 1.19 |
| Function Point Analysis | 4 | 6 | 28 | 18 | 33 | 10 | 2.23 |
| Comparison with project tracking history | 6 | 22 | 37 | 27 | 6 | 1 | 2.95 |
| Number of subsystems | 4 | 30 | 22 | 19 | 15 | 9 | 2.89 |
| Number of windows, reports and database tables etc. | 12 | 58 | 18 | 7 | 1 | 3 | 3.74 |
| Number of classes | 4 | 13 | 13 | 27 | 31 | 10 | 2.25 |
| LOC | 0 | 0 | 27 | 19 | 49 | 4 | 1.77 |
| Number of customers of a component on sale | 1 | 1 | 6 | 18 | 57 | 16 | 1.48 |
| Number of use cases (* omitted) | 9 | 30 | 19 | 19 | 16 | 6 | 2.95 |

# Appendix C. Practical issues of CRM

The tables in this appendix contain the results of questions about practical issues in the survey of the empirical study of CRM (see chapter 3, page 49).

Table 45. Capability of the project manager to estimate the effects.

| The project manager can estimate the influence of the effect in a software development project (%)? | Always | Mostly | Often | Rarely | Never | Don't know |
|---|---|---|---|---|---|---|
| Process when new methods and tools have not been tested? | 4 | 24 | 31 | 24 | 6 | 10 |
| Process when new methods and tools have been tested | 4 | 34 | 39 | 15 | 0 | 7 |
| Process when also new methods and tools have been used in actual projects | 16 | 40 | 24 | 12 | 0 | 7 |
| Project change after the project | 24 | 61 | 10 | 3 | 0 | 1 |
| Teamwork before the start of the project | 3 | 27 | 49 | 13 | 3 | 4 |
| Teamwork after the end of the project | 13 | 54 | 22 | 3 | 1 | 6 |
| Risk before the start of the project | 3 | 33 | 43 | 15 | 4 | 1 |
| Risk after the end of the project | 12 | 61 | 21 | 3 | 0 | 3 |
| Skill before the start of the project | 6 | 43 | 37 | 10 | 3 | 0 |
| Skill after the end of the project | 24 | 58 | 16 | 0 | 1 | 0 |
| Motivation before the start of the project | 4 | 31 | 42 | 15 | 1 | 6 |
| Motivation after the end of the project | 18 | 51 | 22 | 1 | 1 | 6 |
| Is the effort needed for estimating reasonable? | 7 | 63 | 18 | 9 | 0 | 3 |
| When must human effects be estimated for each component separately? | 4 | 16 | 36 | 28 | 1 | 13 |

Table 46. Estimation of project change effect.

| Is it possible to classify added features from the original ones after the project (%)? | Always | Mostly | Often | Rarely | Never | Don't know |
|---|---|---|---|---|---|---|
| How often is it possible | 24 | 61 | 10 | 3 | 0 | 1 |
| How often is the customer/ end user organisation invoiced for additional features | 25 | 48 | 13 | 4 | 4 | 4 |
| How often is the contract changed due to additional features? | 3 | 18 | 30 | 39 | 6 | 4 |

Table 47. Effort of CRM-calculations.

| Suppose that these estimates are made for each project or subproject and for each person. | Always | Mostly | Often | Rarely | Never | Don't know |
|---|---|---|---|---|---|---|
| The effort needed for estimating is reasonable (%)? | 7 | 63 | 18 | 9 | 0 | 3 |
| When must human effects be estimated for each component separately (%)? | 4 | 16 | 36 | 28 | 1 | 13 |

## Appendix D. Abbreviations

| Abbreviation | Description |
|---|---|
| BETA | A programming language |
| CRM | Component Reuse Metrics |
| UI, GUI | User Interfaces, Graphical User Interface |
| LOC, SLOC | Lines Of Code, Standard Lines Of Code |
| COCOMO | Constructive Cost Model |
| COTS | Commercial off the shelf |
| PROBE | PROxy-Based-Estimating |
| FPA, FP, FTP | Function Point Analysis |
| Cosmic-FFP | A new variant of Function Point Analysis |
| CBD | Component-Based Development |
| CORBA | Common Object Request Broker Architecture |
| DCOM | Distributed Common Object Model |
| UML | Unified Modelling Language |
| SQL | Structured Query Language |
| COBOL | Common Business Oriented Language |
| NASA | National Aeronautics and Space Administration |
| NATO | North Atlantic Treaty Organisation |
| US, USA | United States, United States of America |
| ESPRIT-2 | Envoi Sélectif<br>en Psychologie de Références et d'Informations Thématiques |
| REBOOT | Reuse Based on Object-Oriented Techniques |
| IBM | International Business Machines corp. |
| MS | Mi |
| SCM | Software Configuration Management |
| CASE | Computer Aided Software Engineering |
| ER | Entity Relationship |
| PC | Personal Computer |
| HTML | Hypertext Mark-up Language |

| | |
|---|---|
| SEI | Software Engineering Institute |
| MIS | Management Information System |
| HOOPLA | A special programming language |
| SOMA | Semantic Object Modelling Approach |
| MOO | A special programming language |
| IT | Information Technology |

## Appendix E: Verb class - example

```
//  An excerpt from class Frame1, which implements the user
interface  //  of this demo
//  send information from the process to UI using static
variables

  public static Vector recipe = new Vector (20,5);
  public static double power;
  public static double temperature;
  public static double duration;
  public static int stepNbr=0;


void jButton1_actionPerformed(ActionEvent e) {
  // Entry of the Process by File Open-button from the user
interface

  //define containers;
  Tub theTub=new Tub();
  Bottle theSmallBottle=new Bottle("Small",0.5);
  Bottle theLargeBottle=new Bottle("Large",40);
  BoilingFlask theBoilingFlask=new BoilingFlask("Boiling
flask",30,200);
  Bottle theStorageBottle=new StorageBottle("Storage",40,12306);

// Define the process here (see Figure 34, page 141)

  Move step1a=new Move("Move 10 kg sugar to the tub",new
Solid("sugar",10),theTub);
  Move step1b=new Move("Move 5 kg malt to the tub",new
Solid("malt",5),theTub);
  Move step1c=new Move("Move 20 l water to the tub",new
Liquid("water",20),theTub);

  Move step2a=new Move("Move 4 g yeast to the small bottle",new
Solid("yeast",0.004),theSmallBottle);
  Move step2b=new Move("Move 2 dl water to the small bottle",new
Liquid("water",0.2),theSmallBottle);

  Heat step3=new Heat("Heat the small bottle to
37",theSmallBottle,37);

  Move step4a=new Move("Move the contents of the tube to the
large bottle",theTub,theLargeBottle);
  Move step4b=new Move("Move the contents of the small bottle to
the large bottle",theSmallBottle,theLargeBottle);

  Heat step5a=new Heat("Heat the large bottle to
28",theLargeBottle,28);
  Ferment step5b=new Ferment("Ferment the large bottle 2
weeks",theLargeBottle,14);

  Filter step6=new Filter("Filter the large bottle to the
boiling flask",theLargeBottle,theBoilingFlask);
  Distill step7=new Distill("Distill the boiling flask to the
storage bottle",theBoilingFlask,theStorageBottle,78);
 }
```

```
void jButton4_actionPerformed(ActionEvent e) {
  // run the process by Run process button in the user interface
      Process step_n;
      String power_str =  "";
      String temperature_str="";
      String duration_str="";

if (recipe.size() > stepNbr)
 {
     step_n=(Process) recipe.elementAt(stepNbr);
        jTextArea1.append("\n"+step_n.processDescription);

       // System.out.println(power);
        step_n.run();

        temperature_str= temperature_str.valueOf(temperature);
        power_str= power_str.valueOf(power);
        duration_str= duration_str.valueOf(duration);

        jTextField1.setText(temperature_str);
        jTextField2.setText(power_str);
        jTextField3.setText(duration_str);


       stepNbr=stepNbr+1;
   }
   else stepNbr=0;

 }


public class Process {
// implements the base class of all processes
// a verb class example - communication to the UI is common to
all
// processes. The child classes reuse this by inheritance
String processDescription;
 double powerUse;
 double durationOfStep;

 public Process(String desc) {
 this.processDescription= desc;
 Frame1.recipe.addElement(this); }

 public Process() {}

 public void run()
 {
 powerUse=needOfPower();
 powerUse=Math.round(powerUse);

 // send information to the UI by its static class variables
 Frame1.power=Frame1.power+powerUse;
 Frame1.duration=Frame1.duration+durationOfStep;
  }
  public double needOfPower(){
  return 0.0;}
}
```

```
public class Transfer extends Process {
// an example of a specialisation of the process
// a verb class example -
// reuse of the code of class process and extending it by
duration

  Object theMovableObject;
  Object whereToMove;
  public Transfer(String desc) {
  super(desc);
  durationOfStep=10;
  }

  public Transfer(){}
  public void run(){
  super.run();
  }
}

import java.util.*;
public class Move extends Transfer {
// an example of a specialisation of the transfer
// a verb class example
// notice two Move methods for different kind of nouns
  boolean moveWholeContents;
  public Move() {
  }
  public Move(String desc,Chemical chem, Container cont){
    super(desc);
    whereToMove=cont;
    theMovableObject=chem;
    moveWholeContents=false;
   }

  public Move(String desc,Container p_contents, Container cont){
    super(desc);
    whereToMove=cont;
    theMovableObject=p_contents;
    moveWholeContents=true;
  }

  public void run(){
   super.run();
   Container cont=(Container) whereToMove;
  // cont.showContents();
   if (moveWholeContents) {
      Container  contents=(Container) theMovableObject;
      cont.addContents(contents);
   }
   else  {
     Chemical  chem=(Chemical) theMovableObject;
     cont.addChemical(chem);
    }
    cont.showContents();
   }

   public double needOfPower(){
    if (moveWholeContents)
      return 525;  /* arbitrary number for demo purposes */
    else {
       Chemical  chem=(Chemical) theMovableObject;
       return 1.2*chem.getWeight()+10.8; }
  }
}

 public class ChangeTemperature extends Process {
// an example of a specialisation of the process
// verb class example
// adding temperature handling to the process

 Container theObjectToHandle;
  double temperatureTarget;
  double tempBefore;
  public ChangeTemperature() {
  }
  public ChangeTemperature(String desc,Container
p_theObjectToHandle,double p_temperatureTarget) {
  super(desc);
  tempBefore=Frame1.temperature;
  temperatureTarget=p_temperatureTarget;
  theObjectToHandle=p_theObjectToHandle;
  }
  public void run()
 {
  super.run();
  Frame1.temperature=temperatureTarget;
  }
}

public class Heat extends ChangeTemperature {
// an example of a specialisation of the change temperature
// verb class example

  public Heat() {
  }
  public Heat(String p_desc,Container p_theObjectToHandle,double
p_temperatureTarget) {
     super(p_desc,p_theObjectToHandle,p_temperatureTarget);
  }
  public double needOfPower() {
    return 15.8*(temperatureTarget-tempBefore);}
}

import java.util.*;
public class Separate extends Process {
// an example of a specialisation of the process
// verb class example
// notice the connection to the containers
  Container fromContainer;
  Container toContainer;
  public Separate() {
  }
  public Separate(String desc) {
  super(desc);
 durationOfStep=15.00;  }

  public Separate(String p_desc,Container p_fromContainer,
Container p_toContainer){
  super(p_desc);
  fromContainer=p_fromContainer;
  toContainer=p_toContainer;
  durationOfStep=15.00;
  }
   public void run(){
   super.run();
   System.out.println("Separating");

   Container  contents= fromContainer;

   int itemNbr=0;
   for (itemNbr=0;contents.size()>itemNbr;itemNbr++)
   {
     Chemical chem=(Chemical) contents.elementAt(itemNbr);
    if (!this.isSeparating(chem)){
        toContainer.addChemical(chem);
     }
    }
     toContainer.showContents();
   }
  public boolean isSeparating(Chemical p_chem)
  {
    return false;
  }
}
```

```
public class Filter extends Separate {
// an example of a specialisation of the separate
// a verb class example

  public Filter() {
  }
   public Filter(String p_desc,Container p_fromContainer,
Container p_toContainer){
  super(p_desc,p_fromContainer,p_toContainer);
  }
  public boolean isSeparating(Chemical p_chem)
  {
  if ( p_chem.isFiltering())
    return true;
    else return false;
  }
  public double needOfPower(){
  Bottle bottle=(Bottle) fromContainer;
  return 2*bottle.getMaxVolume();
  }
}
```

```
public class Distill extends Separate {
// an example of a specialisation of the separate
// a verb class example

  double temperatureTarget;
  public Distill() {
  }

  public Distill(String p_desc,Container p_fromContainer,
Container p_toContainer,double p_temperatureTarget){
  super(p_desc,p_fromContainer,p_toContainer);
  temperatureTarget=p_temperatureTarget;
  }
  public boolean isSeparating(Chemical p_chem)
  {
    if ( p_chem.isDistilling())
    return false;
    else return true;
  }

  public void run()
  {
  super.run();
  Frame1.temperature=temperatureTarget;
  }

  public double needOfPower(){
  Bottle bottle=(Bottle) fromContainer;
  return 50*bottle.getMaxVolume();
    }
}


public class Liquid extends Chemical {
// an example of a specialisation of the chemical
// a noun class example
// notice focus on attributes

  double volume;
  public Liquid() {
  }
  public Liquid(String p_name,double p_volume){
  volume= p_volume;
  double calc_weight;
  calc_weight=p_volume;
  setWeight(calc_weight);
  setName(p_name);
  }
}

import java.util.*;
public class Container {
// a noun class example
// a base class


Vector contents= new Vector (5,2);
  public Container() {
 }
  public Container(Chemical content) {
  contents.add(content);}

  public void addChemical(Chemical p_chemical) {
  contents.add(p_chemical);}

  public void addContents(Container p_contents)
  {
  int itemNbr=0;
  for (itemNbr=0;p_contents.size()>itemNbr;itemNbr++)
  {
    contents.add(p_contents.elementAt(itemNbr));}
  }
  public void emptyContents(){
  contents.removeAllElements();
  }
  public void showContents(){
   System.out.println("Contents:" );
  int itemNbr=0;
  for (itemNbr=0;contents.size()>itemNbr;itemNbr++)
  {
    Chemical chem= (Chemical) contents.elementAt(itemNbr);
    System.out.println(chem.getName());
  }
  }
  public int size(){
  return contents.size();
  }
  public Chemical elementAt(int itemNbr){
  if (contents.size()>itemNbr & itemNbr > -1)
  {
    Chemical chem= (Chemical) contents.elementAt(itemNbr);
    return chem;
  }
  else return null;
  }
}
```

```
public class Bottle extends Container {
// an example of a specialisation of the container
// a noun class example
// notice focus on attributes


  String title;
  double maxVolume;
  public Bottle() {
  }
  public Bottle(String p_title, double p_maxVolume){
  title=p_title;
  maxVolume=p_maxVolume;}

  public String getTitle() {
  return title;
  }
  public void showContents(){
    System.out.println("Contents:"+this.getTitle() );
    super.showContents();
    }
  public double getMaxVolume(){
  return maxVolume;
  }
}
```

```
public class BoilingFlask extends Bottle {
// an example of a specialisation of the bottle
// a noun class example
// notice focus on attributes

  double maxTemperature;
  public BoilingFlask() {
  }
  public BoilingFlask(String p_title, double p_maxVolume, double
p_maxTemperature){
  super(p_title,p_maxVolume);
  maxTemperature=p_maxTemperature;
}
}
package ChemicalProcess;

public class Tub extends Container {
  double maxCapacity;
  public Tub() {
  }
  public Tub( double p_maxCapacity){
  super();
  maxCapacity=p_maxCapacity;
  }
}
```

```
public class Chemical {
// an example of a noun base class

  double weight;
  String name;
  public Chemical() {
  }
  public Chemical(String namex, double weight1) {
  name=namex;
  weight=weight1;
  }
  public void setWeight(double g){
  weight = g;
  }
  public double getWeight(){
  return weight;
  }
   public void setName(String p_name){
   name= p_name;
  }
  public String getName(){
  return name;
  }
  public void ferments()
  {
    if (this.getName() == "sugar" ) {
        this.setName("alcohol");
        this.setWeight(this.getWeight()*0.52);
     }
  }
  public boolean isFiltering()
  {
    if ( this.getName()=="yeast" || this.getName()=="malt") {
    return true; }
    else { return false; }
  }
  public boolean isDistilling()
  {
    if ( this.getName()=="alcohol") {
    return true; }
    else { return false; }
  }
}

public class Solid extends Chemical {
 // an example of a specialisation of the chemical
// a noun class example
// notice focus on attributes

  public Solid() {
  }
  public Solid(String namex,double g){
  super(namex,g);
  }
}

public class Liquid extends Chemical {
// an example of a specialisation of the chemical
// a noun class example
// notice focus on attributes


  double volume;
  public Liquid() {
  }
  public Liquid(String p_name,double p_volume){
  volume= p_volume;
  double calc_weight;
  calc_weight=p_volume;
  setWeight(calc_weight);
  setName(p_name);
  }
}
```

Turku Centre for Computer Science

TUCS Dissertations

10. **Marina Waldén**, Formal Reasoning About Distributed Algorithms
11. **Tero Laihonen**, Estimates on the Covering Radius When the Dual Distance is Known
12. **Lucian Ilie**, Decision Problems on Orders of Words
13. **Jukkapekka Hekanaho**, An Evolutionary Approach to Concept Learning
14. **Jouni Järvinen**, Knowledge Representation and Rough Sets
15. **Tomi Pasanen**, In-Place Algorithms for Sorting Problems
16. **Mika Johnsson**, Operational and Tactical Level Optimization in Printed Circuit Board Assembly
17. **Mats Aspnäs**, Multiprocessor Architecture and Programming: The Hathi-2 System
18. **Anna Mikhajlova**, Ensuring Correctness of Object and Component Systems
19. **Vesa Torvinen**, Construction and Evaluation of the Labour Game Method
20. **Jorma Boberg**, Cluster Analysis. A Mathematical Approach with Applications to Protein Structures
21. **Leonid Mikhajlov**, Software Reuse Mechanisms and Techniques: Safety Versus Flexibility
22. **Timo Kaukoranta**, Iterative and Hierarchical Methods for Codebook Generation in Vector Quantization
23. **Gábor Magyar**, On Solution Approaches for Some Industrially Motivated Combinatorial Optimization Problems
24. **Linas Laibinis**, Mechanised Formal Reasoning About Modular Programs
25. **Shuhua Liu**, Improving Executive Support in Strategic Scanning with Software Agent Systems
26. **Jaakko Järvi**, New Techniques in Generic Programming - C++ is more Intentional than Intended
27. **Jan-Christian Lehtinen**, Reproducing Kernel Splines in the Analysis of Medical Data
28. **Martin Büchi**, Safe Language Mechanisms for Modularization and Concurrency
29. **Elena A. Troubitsyna**, Stepwise Development of Dependable Systems
30. **Janne Näppi**, Computer-Assisted Diagnosis of Breast Calcifications
31. **Jianming Liang**, Dynamic Chest Image Analysis: New Model-based Methods for Dynamic Pulmonary Imaging and Other Applications
32. **Tiberiu Seceleanu**, Systematic Design of Synchronous Digital Circuits
33. **Tero Aittokallio**, Characterization and Modelling of the Cardiorespiratory System in Sleep-disordered Breathing
34. **Ivan Porres**, Modeling and Analyzing Software Behavior in UML
35. **Mauno Rönkkö**, Stepwise Development of Hybrid Systems
36. **Jouni Smed**, Production Planning in Printed Circuit Board Assembly
37. **Vesa Halava**, The Post Correspondence Problem for Marked Morphisms
38. **Ion Petre**, Commutation Problems on Sets of Words and Formal Power Series
39. **Vladimir Kvassov**, Information Technology and the Productivity of Managerial Work
40. **Franck Tétard**, Managers, Fragmentation of Working Time, and Information Systems
41. **Ján Maňuch**, Defect Theorems and Infinite Words
42. **Kalle Ranto**, $\mathbf{Z}_4$-Goethals Codes, Decoding and Designs
43. **Arto Lepistö**, On Relations Between Local and Global Periodicity
44. **Mika Hirvensalo**, Studies on Boolean Functions Related to Quantum Computing
45. **Pentti Virtanen**, Measuring and Improving Component-Based Software Development

Turku Centre for Computer Science
Lemminkäisenkatu 14
FIN-20520 Turku
Finland

http://www.tucs.fi

University of Turku
- Department of Information Technology
- Department of Mathematics

Åbo Akademi University
- Department of Computer Science
- Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration
- Institute of Information Systems Science