Knowledge Engineering

vs.

Software Engineering

University of Amsterdam,

Computer Sciences Department

Master's Thesis

by

R.P. Ogilvie

Date: October 1988
Reference: UvA/P8902-ENG

© 1988, '93 R.P. Ogilvie

Reproduction is permitted provided that the author's full name is included in any such reproduction.
R.P. Ogilvie, Master Thesis
Report P8902 (Dutch)

University of Amsterdam,
Faculty of Mathematics and Computer Sciences,
Programming Section

P.O. Box 41882
1009 DB Amsterdam
The Netherlands.

Author's present address:

Aart van der Neerv 47
1191 ED Ouderkerk a/d Amstel
The Netherlands
## INTRODUCTION

### 1 METHODOLOGY

1.1 What is Methodology?

1.1.1 Definitions

1.1.2 Formal versus Empirical Sciences

1.1.3 Theory and Model

1.2 What is a Methodology?

1.2.1 Elements of a Methodology

1.2.1.1 Language

1.2.1.2 The Empirical Cycle

1.2.1.3 Implementation

1.2.2 Requirements of a Methodology

1.2.2.1 Methodological Requirements

1.2.2.2 Social Requirements

1.2.2.3 Organizational Requirements

1.2.2.4 Economical Requirements

1.2.2.5 Comparing the Methodologies

1.2.3 Recipe or Checklist?

### 2 ENGINEERING

2.1 Definitions

2.2 Requirements Engineering

2.2.1 Non-functional Requirements

2.2.2 Functional Requirements: the Analysis

2.2.3 The Requirements Document

2.3 Design Engineering

2.3.1 Realization

2.3.2 Introduction, Use and Control

2.4 Life-cycle Models

### 3 KNOWLEDGE ENGINEERING

3.1 Definitions

3.2 The Necessity for Knowledge Based Systems

3.3 The Origination of a Knowledge Engineering Methodology

3.4 Knowledge in KADS

3.4.1 The Acquisition of Knowledge

3.4.1.1 Types of Knowledge

3.4.1.2 Elicitation Techniques

3.4.2 Knowledge Levels

3.4.2.1 Domain-layer: Concepts, Relations and Structures

3.4.2.2 Inference-layer: Meta Classes, Knowledge Sources and the inference Structure

3.4.2.3 Task layer: Goals, Tasks and Control

3.4.2.4 Strategic level: Plans, Meta-knowledge, Repairs

3.5 KADS Phasing

3.5.1 Internal Branch: Analysis

3.5.1.1 Knowledge Identification

3.5.1.2 Knowledge Conceptualization

3.5.1.3 Conceptual Model Construction
<table>
<thead>
<tr>
<th>FIGURE</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>The Empirical Cycle</td>
<td>5</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Cycle and Meta Cycle</td>
<td>6</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Relationship between Engineering Concepts</td>
<td>11</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Symbolic representation of Requirements Engineering</td>
<td>12</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Cycles in the Analysis Process</td>
<td>13</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Models in the development of an Expert System</td>
<td>19</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Two views on the four Knowledge Layers in KADS</td>
<td>22</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Levels in the description of Knowledge</td>
<td>22</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Inference and Task Structure for Systematic Diagnosis</td>
<td>24</td>
</tr>
<tr>
<td>Figure 10</td>
<td>KADS Phasing</td>
<td>25</td>
</tr>
<tr>
<td>Figure 11</td>
<td>The Design Process in KADS</td>
<td>27</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Some examples of Environments</td>
<td>28</td>
</tr>
<tr>
<td>Figure 13</td>
<td>KADS Life Cycle Model: Requirements Engineering</td>
<td>29</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Reports of KADS Requirements Engineering</td>
<td>30</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Interaction between World and System</td>
<td>37</td>
</tr>
<tr>
<td>Figure 16</td>
<td>Relationship between Engineering Concepts</td>
<td>48</td>
</tr>
<tr>
<td>Figure 17</td>
<td>The Software Development Process</td>
<td>53</td>
</tr>
</tbody>
</table>
INTRODUCTION

In his famous epic [Turing50] Turing answers the question "Can machines think?" using an imitation game in which the answers from the one questioned is passed on by an intermediary to the interrogator. He replaces the original question by a new one, which basically implies: is the interrogator able to decide whether the one questioned is a man or a machine, and he poses:

"May not machines carry out something which ought to be described as thinking but is very different from what a man does? This objection\textsuperscript{1} is a very strong one, but at least we can say that if, nevertheless, a machine can be constructed to play the imitation game satisfactorily, we need not be troubled with this objection."

By now (1988) this has resulted in machines that can model an essential part of human capacity. Built on the technical achievements and realised by the Software Engineering, software systems with a higher order of functionality have been developed, called Expert System Shells, in which aspects of human knowledge can be represented and which are capable of some reasoning with this knowledge. Knowledge Engineering, the young discipline that is involved in this imitation game, is generally considered to be a completely different discipline than Software Engineering. The tendency is to consider Knowledge Engineering as an application of computer sciences, like an end user who uses a statistical package, instead of considering it as a vital part of computer sciences.

However Knowledge Engineering constructs systems for the same machines as Software Engineering does, using software systems that have been realised by Software Engineering: it remains a finite state machine, equivalent to the Turing machine, and the heart of the matter is the code that calculates and that manipulates the data structures. This raises the question what the difference really is.

In this research the methodologies and the methods and techniques that are used by both disciplines to realise their systems with are examined. Methodologies direct the development process and the similarities and the differences between the methodologies and the methods and techniques determine which aspects in the real world are modelled and what the physical characteristics of the realised system are.

To put this research in a clear framework the chapters 1 (Methodology) and 2 (Engineering) present concepts from respectively methodology and engineering. Chapter 1 contains a theoretical discussion of methodology and in this way arrives at a number of theoretical and practical characteristics that are required for a good methodology, summarised in a checklist. Chapter 2 comprises an attempt to put the most important concepts from engineering in a framework for further reference. Although an attempt is made to discuss the two subjects as general as one can, diversions to the field of computer sciences could not be avoided.

Chapter 3 discusses Knowledge Engineering. Being the one "real" methodology in this field, KADS is used to describe it. First the ideas behind KADS are summarised and next the methodological aspects are discussed. Then other work is briefly discussed, practical experiences with KADS are listed and KADS is matched against the requirements from chapter 1. The restriction to KADS does not seem to detract from the value of the research as we will arrive at quite an accurate description of what is being modelled and how a system is accomplished.

\textsuperscript{1} 'objection' refers to 'think' in the title.
Chapter 4 discusses Software Engineering. This field is divided into two parts: specifying and developing systems, and developing supportive tools and standards. Next the trends in Software Engineering are discussed and the checklist from Chapter 1 is matched against a hypothetical, average Software Engineering methodology. Chapter 5 (Summary and conclusions) comprises short summaries of the preceding chapters. Next separate conclusions and recommendations for the respective disciplines are presented. At the end of the chapter an attempt is made to integrate both disciplines into one methodology with next to no difference between "traditional system" and "knowledge based system". This integration is rather idealistic and does not take into account many practical problems. However, let us hope this example calls for ideas for further developments towards an integrated discipline.

This Master's Thesis has been supervised by:
J. Treur (general supervision),
J.A. Breuker (KADS, Methodology)
G.R. Renardel de Lavalette (Software Engineering),
A.S. Spijkervet (practical implications).

Amsterdam, 10 October 1988
Paul Ogilvie

This English version has been created from the original Dutch version by Céline François, to whom many, many thanks go and without whose effort this thesis would have been unavailable to the international community. I promise never again to write a computer sciences document in Dutch.

Ouderkerk, December 1992
1 METHODOLOGY

1.1 What is Methodology?

1.1.1 Definitions

The Concise Oxford dictionary defines methodology as "science of methods" and method as "a systematic procedure for doing". According to this definition methodology is a science that is engaged in studying methods, while method is an executive instrument (of a science).

A more specific definition is given in [Kaste86] who describes a methodology as "the examination of arguments or reasons that are adduced to label certain statements in theories as scientific statements". According to this definition the scientific character of a statement is determined by the manner in which the statement has come to be and is accounted for or justified, not by the statement itself.

For that matter one can also speak of "result" instead of "statement", and likewise an operational expert system, for instance, may be considered a statement in a world of potential expert systems, about which then it may be disputed whether the system has been realised in an accountable manner. From now on "statement" thus shall be used as a synonym for every product or result of a science (and in this context software engineering, knowledge engineering and computer science are considered to be sciences or the applications of science).

1.1.2 Formal versus Empirical Sciences

The account for or justification of statements is different for formal and for empirical sciences. In formal sciences (like mathematics and logic) the statements and definitions refer only to themselves and the justification is in the correct use of formalisms. In an empirical science the statements refer to a reality that lies outside the statements themselves, and the truth value that must be attributed to the statements must be determined by some kind of observation. In the remaining we restrict ourselves to empirical sciences since software engineering and knowledge engineering have been originated for an important part in an empirical manner.

1.1.3 Theory and Model

Empirical sciences acquire knowledge from observations and apply induction, where from a restricted number of observations a theory is derived that generalises to "all cases" (including the cases that have not been examined). But what do we observe? Implicitly we assume that we want to observe a limited number of phenomena, and not the world as a whole. Apparently we restrict ourselves to those phenomena that seem to be important to us for whatever reason: we have restricted ourselves to a model of the world in which has been abstracted in favour of a particular goal.

---

2 In order to be complete also Technique will be defined here: "The whole of operations or performances needed to bring about something in a particular branch of art, industry, etc." This implies that technique points out the effect on physics, whereas method implies the mental activity (therefore needed).

3 Longman Concise English dictionary
A second way to acquire knowledge is by deduction, where from preceding generalisations the argument has been continued and new laws have been derived. In this way a new theory is created that explains and predicts phenomena in the model. Therefore an (empirical) theory is not absolutely true; it is only an ordered scheme of known facts. A good theory does indicate however which experiments should be conducted and what to pay attention to in order to verify the theory. In the long run however there can be so many observations that are not well described anymore by the theory, that the theory stands itself in the way and should be renewed (1.2.1.2).

1.2 What is A Methodology?

Methodology examines whether statements are scientific by examining their origination. Conversely, by observing the origination and applying induction and deduction, methodology can formulate norms and rules that enable the production of an accountable statement or product in the field of the science concerned. A statement in methodology addresses then whether a statement in a (another) science has been accomplished in an accountable manner; a statement of methodology describes then how to achieve statements in another science in an accountable manner. This is called respectively the descriptive and the prescriptive (or normative) character of methodology.

For practical application of a science often a normative methodology is derived from the descriptive methodology by using for instance only the construction rules and leaving out the rules of proof. In that case it is called "a" methodology, being the product of the science methodology.

By now the terminology might get a little confusing, because a normative methodology, being a prescription of a "a systematic procedure for doing", is thus a method (see 1.1.1). We obtain some more clarity by considering methodology to be a meta-science as it examines other sciences, and to consider a normative methodology on this meta level to be a method for practising a particular science. On the level of that science this is called a methodology and a method is then a manner to perform a certain task in that science. Still simpler, we could classify "a methodology" as a generic name for the product of the science methodology.

1.2.1 Elements of a Methodology

1.2.1.1 Language

To make statements about observations a descriptive language is essential in which accurate and unambiguous statements about the model can be formulated. Here the primary choice is the choice between an informal (verbal) and a formal (mathematical) manner of expression. In case the primary function of the model is to give a general approach to a composite of relationships that cannot be represented satisfactorily by relationships between measurable variables, a careful verbal framework is the most adequate.

1.2.1.2 The Empirical Cycle

The primary manner in which knowledge is acquired in an empirical science is by observing and experiencing, and a learning effect can be seen if purposeful behaviour is conducted in a more accurate, quicker or more effective manner as a result from preceding experiences [de Groot61]. The learning effect is known in cybernetics as feedback. In the process of
experience the fundamental empirical cycle described in Figure 1 can be perceived. De Groot observes that this cycle "returns over and over again, in the small and in the large, either in the more or less fundamental shape or in a complex interrelationship of cycles falling over each other and linked to other processes (...)". Furthermore he gives a more accurate classification of this cycle:

1. **observation**: collecting and classifying empirical facts; establishing hypotheses (conjectures);
2. **induction**: formulating hypotheses;
3. **deduction**: deriving special consequences from the hypotheses in the shape of testable predictions;
4. **testing** the hypotheses: by verifying the predictions against new empirical material;
5. **evaluation**: evaluating the outcome of the test in connection with the established hypotheses and in connection with potential new successive researches.

**Figure 1** The Empirical Cycle

The fifth phase, evaluation, flows back into the first phase, observation, and often the classification is limited to four stages (observe-guess-predict-check). However in a single, completed research and the report of that research, a separate evaluation phase can definitely be recognised. This takes place on two levels. On the subject level (the science that is the subject of the methodology) the effectiveness of the actions are assessed and deviations of the norms from the reality that has been observed are attributed to inaccurately following the rules of the methodology. The premises (and goals) are adapted and the cycle is executed again.

On the object level (where the science itself is the subject of study) one judges whether the theory still describes (explains) all observations adequately or whether the theory (and thus the methodology) should be adapted. For instance a generalisation might have been made on an incomplete or incorrect model or the model has changed. Reticence is needed here to some extent. On one hand there might be no need for a 'new' or 'better' model as the work can be continued as it is, on the other hand a new model must be confirmed first by empirical evidence.

A cycle that brings about (important) changes in the theory, will usually also affect the normative methodology. In that case we will speak of a **meta-cycle** of the methodology, as opposed to the cycle that only gives feedback so that eventually the desired output is achieved (operational feedback). This is depicted in Figure 2.
Concerning the empirical cycle the following conclusions can be drawn:
* The cycle offers a *decomposition* in steps that have to be taken to achieve in an accountable manner the goal aimed at.
* A methodology for an empirical science contains *cyclical elements* as the science has no absolute world-view (stepwise refinement).
* Every phase should as a *product* supply the necessary and sufficient premises for the succeeding phase (necessary: so that the next stage is well-founded, sufficient: so that no distorting factors will be introduced).
* The steps of the cycle are *not fixed* but the three steps of observation, action and evaluation are always present.
* The methodology must be *flexibility*. This is because the feedback from the evaluation may concern the object of research, but may just as well concern the theory or the methodology itself.

1.2.1.3 Implementation

Finally guidelines are needed to describe ‘how’, the procedures of the science concerned. This has the purpose to guarantee more or less (the scientific character of) the result. The methodology must then answer the following questions:
   a) how to perform the task or step correctly,
   b) how to control the complexity of the task,
   c) how to enforce the rules that must be observed.

ad a) To avoid incorrect implementation of the task, the methodology should supply the field of science concerned with methods and techniques with which to carry out the individual tasks. The methodology thus tells which sequence of activities or operations will lead to the result aimed at in an accountable manner. It is the responsibility of the methodology that the methods are geared to one another, that is, that the results of a method are indeed the necessary and sufficient premises for
the succeeding task.

ad b) To control complexity, to make it comprehensible for human intellectual grasp, two options are available. The first option is to construct tools that relieve human beings from trivialities and thus enable them to concentrate on the real task. A second option is to describe a more detailed decomposition for the task and to assign specific methods and techniques to each. In that case it is not unthinkable to split up the cycle again in the phases mentioned before, each of which will be passed through cyclically again.

ad c) The observation of the rules can be enforced by supplying tools that are based on the methodology and that implement the methods and techniques.

1.2.2 Requirements of a Methodology

Which requirements should be demanded from a methodology? This question lies in the field of the practical application of the science, in our case computer science. For a methodology to be practically useful it should contain at least the elements mentioned in the preceding paragraph. Furthermore social requirements are posed, like “thou shouldst not pollute”. The classification of requirements may be the following:
- methodological requirements,
- social requirements,
- business requirements.

The latter type of requirements, the business requirements, may be sub-classified as follows:
- organisational requirements,
- economical requirements.

We specify the business requirements based on observations from the actual practice of IT as is described in several publications ([Bem84], [Vliet84], Martin82], [Boehm81], [Nij88]).

1.2.2.1 Methodological Requirements

The subject-science requires from a methodology that, provided that the established norms are observed, the methodology guarantees that the result will have been achieved in an accountable manner and will thus answer the goal. So in this case requirements of the quality of the methodology’s elements are concerned. Referring to the preceding paragraph the requirements are as follows:
- the set of definitions is accurate and unambiguous: it will not cause any confusion;
- the (plural) decomposition of tasks makes the complexity controllable;
- every task or phase delivers a well defined and well geared product;
- the tasks can, either each on their own or as a whole, be performed cyclically;
- for every task the necessary methods and techniques are described systematically and the advantages and disadvantages of the methods and techniques have already been evaluated.

4 In case these options do not provide a solution anymore a third, more rigorous, option is to review the methodology by modifying the model, reject generalizations, and again consider all observations, which may result in a model in which the complexity is controllable.

5 Business requirements imply those requirements that enable a company to use the production factors capital, labour, raw material and entrepreneurship optimally [Heertje79].
determined empirically;
- if possible practical tools are described that relieve man from trivialities and that enforce
the adherence to norms and rules of the methodology.

1.2.2.2 Social Requirements

The social requirements strongly depend upon the field of the science concerned and upon
the prevailing culture. The requirements for the chemical industry have for instance been
changed considerably in recent years\(^6\). These requirements have an ethic character and do
not attribute to a better methodology, so in this paper they will not be discussed anymore.
However we want to stress the point that it is essential to meet these requirements in order
to make the methodology, and thus the science, socially acceptable.

1.2.2.3 Organisational Requirements

The organisational requirements concern two separate organisations. On one hand there is
the *subject organisation* that uses the methodology to support the production process (for
example the software house). On the other hand there is the *object organisation* that is the
object to which the methodology is applied (the client of the software house).

The requirements that have been found regarding the subject organisation, concern the
functioning of the methodology in the organisation. They are the following\(^7\):
- the methodology is *effective* and when used well it supplies the intended result;
- the methodology is *efficient*, that is, within a minimal time span and using a minimum
  amount of resources;
- the methodology delivers *transferable results*;
- one must be able to *learn* the methodology.
- the methodology *stimulates* communication;
- the methodology is *complete* and supports all phases of the development process.

The requirements that have been found regarding the object organisation (the organisation
in favour of which the system is developed) are the following:
- the methodology supports the *strategic planning* of the (computerised) systems in the
  organisation;
- the methodology supports the *organisational transformation process* as a consequence
  of (and pre-condition for) the introduction of the (computerised) system;
- the methodology supports the *continuous change of the organisation* and the necessity
  that the system must also be adapted (maintenance);
- the methodology acknowledges the *importance of user-participation* because of
  motivation and acceptence (environmental requirement).

---

\(^6\) Forced by government regulations concerning the environment, which may even cause a company to be closed
down, ecologically sound production has become one of the critical factors for a company, on which her survival
may depend.

\(^7\) As for validation (building the right system) and verification (building the system right) I remark that these can be
considered the evaluation phases of the respective sub-cycles of analysis and design.
1.2.2.4 Economical Requirements

Some of the preceding requirements may also be considered to be economical requirements. For instance efficiency obviously is a normative rate of return. A typical economical requirement is for instance instruments for a good cost/benefit analysis. However this appears to be extremely difficult for the simple reason that most benefits of computerisation cannot be measured, like human efficiency, motivation, increasing abstraction, etcetera. On the contrary, the costs of an automation project can be measured perfectly well but it is hard to estimate them beforehand. Calculative models are available though, that, once incorporated in the subject organisation, can estimate the costs.

Examples are CoCoMo [Boehm81], PRICE SP [Cuel87], and Function point analysis (FPA) [Strat87]. A comparison between four models (SLIM, CoCoMo, FPA, ESTIMACS) can be found in [Kem87]. Strikingly there are enormous deviations between the estimated costs that the models predicted and the actual costs. These sometimes were running to 600% or more, probably because the models were badly calibrated towards the environment in which they were used, which supposition makes the results of the research seem to be disputable. Nowadays FPA is generally preferred. It is the only method that is based on pre-measurables, namely on numbers and complexity of the user functions, as opposed to for instance CoCoMo that is based on the number of lines of source code to be produced. Also in the research mentioned this method proves to be good.

The most important economical requirements are concerned with the control of the project are:
- the methodology contains a planning method for the allocation of people and resources;
- the methodology contains methods and techniques with which norms for the costs can be set (pre-calculation) and can be tested (post-calculation) and thus provides for continuous calibration of the model;
- the methodology must have milestones for time, money and quality where a go/no go can be commanded for the following phase. Mostly these are the phase boundaries.

1.2.2.5 Comparing the Methodologies

Comparing is often a subjective process. For instance when a DBMS or a mainframe is purchased, the various people involved have various demands and different "feelings" about a certain product or solution. A discussion will evolve and the result will be the consensus of the people involved. A decision made in this manner is not objective and will probably have been influenced by politics and emotions. This may lead to a final decision that is not only sub-optimal, but that may even be completely wrong.

A more objective decision process starts off with the determination of the requirements and the norms. Lists of requirements by everybody involved are put together and each requirement is assigned a relative weight. For every requirement it is determined whether not meeting a minimum norm will label the product unsuitable and out of question. Discussion now focuses on the foundation of the requirement and on determining its relative weight and minimum norm. One such method, MECCA (Multi Element Comparison Component Analysis), is described by Gilb in [Gilb77].

1.2.3 Recipe or Checklist?

The different kinds of users of a methodology often use it for different reasons. For management the methodology is an instrument for control, and milestones and prescribed (periodical) reports are considered to be evidence of progress and as "a finger on the pulse"
of analysts and designers. For experienced analysts and designers this recipe-like procedure usually is a disaster: one is required to write documentation for intuitively obvious steps ("everybody can see that"), or to document models that document themselves, or to use techniques that are unfit. The recipe-like procedure and the enforcement of the rules evoke the following objections:
- unfit techniques yield unfit results;
- time is wasted on unnecessary steps and reports;
- one is not optimally creatively occupied and the level of frustration rises.

Following the recipe-like procedure too strictly thus is costly and may even lead to the non-realisation of the intended product. Products have more chance to be realised by competent technicians than by a methodology, or, like Rees says in [Rees84]: "The method does not do it" (but the designer does!).

One should realise that the methodology is based on possibly incomplete theories on models that do not necessarily represent all relevant elements of the world to be modelled, in other words, the methodology can be wrong. On the other hand, for people who are not experienced in a particular discipline, applying the methodology like a recipe is the best way to gain experience.

To avoid this situation, the methodology should be configured for every project and it should be determined which steps do or do not have to be carried out and why, which documents should be or should not be produced and why, and which methods and techniques shall be used. Thus the methodology also gains the function of a checklist and, in case not every task is performed in reality, the checklist urges to think about these aspects.

In this context the following UNIX-fortune cookie all of a sudden becomes very appropriate: "The level of management is inversely proportional to the degree of technical competence".
This chapter summarises engineering concepts used when building computerised systems, without directly taking either software engineering or knowledge engineering into consideration. Most engineering methodologies being based on the industrial life cycle model this model will be discussed successively.

2.1 Definitions

Boehm ([Boehm81]) found in the Webster's New Intercollegiate Dictionary (1979) the following definition of engineering:

"Engineering is the application of science and mathematics by which the properties of matter and energy in nature are made useful to man in structures, machines, products, systems and processes."

Following that engineering has the function of a bridge between society and technique: "...matter and energy are made useful to man...". This justifies why most engineering methodologies can be divided into two branches, namely an internal branch and an external branch (terminology derived from KADS). The external branch is focused on the object-organisation in which the system is intended to function, on the determination of the needs and constraints, and on the adaptation of the organisation to the new system (introduction, use and control) which will make the product "useful to man". The internal branch is focussed on the analysis and construction of the product (matter and energy) and is the actual application of the science concerned.

Another classification often used is that of Requirements Engineering (RE) and Design Engineering (DE). The relationship between these four terms is presented in Figure 3. The quadrants in the figure represent the task fields. The arrows indicate a certain order in which the tasks are executed. This order is based on the starting point of the first activity of each task. Once started, the activities are performed in parallel or alternating and a (cyclical) interaction between tasks may develop. For instance the first task of the field "non-functional requirements" may be a subtask like "analyse present situation" and only after this activity has been performed the first phase of the analysis can begin.

![Figure 3 Relationship between Engineering Concepts](image-url)
In the following sections the four task fields will be described in more detail. Finally we identify a preparatory task that is concerned with the objectives (goals) of the object-organisation, strategic planning, for which techniques like BSP (Business Systems Planning [IBM82]) and CSFs (Critical Success Factors) can be used. In the strategic planning process subsystems should be demarcated and priorities should be assigned. Strategic planning is not further discussed in this paper.

2.2 Requirements Engineering

Often two different meanings can be attached to Requirements Engineering (RE). On one hand RE should identify needs and express them in a specification; on the other hand RE should define requirements on the project-organisation that must turn this specification into an operational system. This double meaning assigned to RE is perhaps not too fortunate because control and specification aspects are mixed up. Definitely a separate control task can be distinguished throughout the development, however this is usually regarded to be an integral part of RE. In the following the control and organisational aspect will be discussed separately and will be called in analogy Project Engineering (PE) (see also [Jans87] and [Nij88]).

A requirement is defined in [Barth86] as "a property that the system must satisfy, independent of any actual realisation". Requirements can be classified as functional and non-functional. Functional requirements (analysis) model the relevant internal states and the behaviour of the components and of the interaction with the environment. Non-functional requirements are the constraints that limit the potential types of solutions. Hereafter both are discussed in more detail.

![Symbolic representation of Requirements Engineering](image)

**Figure 4** Symbolic representation of Requirements Engineering

2.2.1 Non-functional Requirements

Determining the non-functional requirements usually takes place in a more or less formalised discussion between the client and the principal and the result is usually in natural language. Here a rough model of the environment in which the system will be functioning is drawn up and the goals of the system are identified. These data make up the input for the analysis phase. Furthermore constraints are specified like cost, speed, reliability, flexibility, extensibility, integration, user interface, etcetera, that, together with the analysis model, are...
the input for the design stage. Typical tasks are "Determine scope of project" and "Analyze present situation", to determine for instance whether a strategic planning must be performed as yet. Furthermore the principal should point out to his client which alternatives exist and the consequences of a certain choice, for instance "island automation" versus integrated planning. Finally these requirements often help when describing the subject of the agreement.

Methods and techniques for these tasks usually are checklists of activities that should be performed and documents that must be produced. Examples are SDM [Turner87], that provides a good checklist of activities, and the KADS RE-documents (paragraph 3.5.3 and [Barth86]).

### 2.2.2 Functional Requirements: the Analysis

The analysis, determining the *internal specifications*, includes inventoring tasks, procedures and data. The result is a detailed functional (conceptual) model of the system that describes the "what" of the future system. This analysis task comprises two types of activities: acquiring information and knowledge and recording this in a model. A third task is the *validation* of the model: check whether the model still describes a system that fulfils the goals stated before. The cyclical aspect of these tasks is depicted in Figure 5.

![Figure 5 Cycles in the Analysis Process](image)

RD = Requirements Document

Unless indicated, all arrows are input

For the analysis no task model has been found, like the task model for design that has been found in KADS (11). Nevertheless I will here describe a tentative classification8 (see also17):

- **collection**: the analyst acquaints himself with the field (documentation, books, interviews) to get an overview of the field to be researched; he collects as many data as he can on the field to be modelled;
- **classification of data**: the data found are classified in structures, data collections,
diagrams etcetera. The purpose of this is to get an overview over the tangle of data, not to get a picture of the total system yet;

- **synthesis**: the structures and diagrams are merged into a conceptual model of the future system.

Methods and techniques for the acquisition have only been found in KADS (interview techniques, orientation phase). Methods and techniques for modelling are ERAE (Entity Relation Attribute Event), DFDs (Data Flow Diagrams) and generic task models (KADS interpretation models). These methods and techniques will be discussed in more detail in the chapters 3 (Knowledge Engineering) and 4 (Software Engineering).

### 2.2.3 The Requirements Document

The product of RE, the Requirements Document (RD), describes a model of the world with the system (in which the system is reality). It contains the formulation of **goals** that must be met by the system, a specification of the **tasks** that have to be performed by the system in that world in order to reach the goals, and a specification of all **interactions** between the system and the world (interfaces). This model is depicted symbolically in Figure 4. The polygon demarcates the field that the conceived system will take up in the world; the arrows represent the interactions between the world and the system. What the implementation of this system will look like is not important as yet.

### 2.2.4 Project Engineering

The tasks of Project Engineering concern the object organisation (user participation, transformation analysis, progress monitoring by the client) as well as the subject organisation (planning, resources) and the things that matter are **time**, **money**, **quality**, **people**, **organisation** and **communication**. The tasks of Project Engineering include:

- **compose user groups and a steering committee** from the object organisation and the subject organisation, lay down the frequency of their meetings, their tasks, documents to be produced by them, their communication structure;
- **project planning**: time periods for products to be delivered and evaluation, planning of people and material, monitoring progress, check-points;
- **project organisation**: internal organisation structure, authorities, task formulation, determining which capabilities (people) and equipment are needed in which stage;
- **analysis of costs and benefits**;
- **methodology configuration**: which documents are needed and why and which are not needed and why not; which activities must be implemented in which order (or in parallel) and which activities are not needed and why not. (In [Barth87] (KADS-RE) methodology configuration is called a crucial task: "instead of forcing the project into a rigid framework this Model itself is modelled to suit each project").
2.3 Design Engineering

2.3.1 Realisation

In the realisation phase a system is being designed and built that realises the proposed tasks from the Requirements Document. The constraints are included in the process and the task concentrates on delivering an efficient and maintainable system. The first step often is the functional design (functional decomposition) in which the functions are specified that are needed to implement the task (tasks) with. After this the realisation usually takes place "middle-out", implying that the design is created top-down, while bottom-up a supporting modular architecture is built. This bottom-up part constructs tools and design elements (primitives) that are used and put together into functional modules from the top-down design that realise the desired behaviour of the system. The realisation will be described in more details in paragraphs 3.5.2 and 4.2.2.

2.3.2 Introduction, Use and Control

The introduction of every computerised system results into changes in the tasks or the performance of tasks. The preparations for the introduction of the system include the following fields:

- *organisational changes*, caused by the new possibilities of the system, for instance is decentralisation wanted or possible;
- *training and documentation* so that the new system can be used optimally;
- *conversion and acceptance*, the (gradual) conversion from the old (computerised) system to the new one and the supervision to prevent that the new one does not disturb the organisation;
- *system control* which includes operating, system management and materials.

Finally the evolution of the organisation usually will lead to the necessity of the evolution of the system, the *maintenance*. This phase is comparable to the evaluation phase in the empirical cycle. Besides changing organisational goals, also the observation of the organisation with the system will cause a drive to new steering actions. New requirements will expand the system, adding new tasks or requiring existing tasks to be changed.

2.4 Life-cycle Models

Life-cycle methodologies and models (LCMs) are used very often in the development of industrial products and do not only occur in software development. LCMs actually model one empirical standard cycle having the shape of *observation* (pre-study), *induction* (analysis), *deduction* (design), *check* (realisation) and *evaluation* (use and maintenance). The standard approach to adapt the generic life cycle model to software is to instantiate it with a number of methods and techniques for every stage. A LCM would rather be considered to be a general model for the management than being a specific model for the development of software.

---

9 Depending on the degree in which the computer is indispensable, a company may go bankrupt if the performance of the computer system is not good enough [Berghe86].

10 If a good data planning has been performed beforehand, usually the data base will not need to be changed: data are fairly static, tasks change more dynamically.
In a LCM several stages are passed through sequentially\textsuperscript{11} and it is attempted to establish distinct boundaries between the phases to facilitate decision making (go/no go). Therefore LCMs hardly, if at all, take the inherent cyclical aspect of the analysis and design process into account. The main cause of these cycles is the impossibility to separate the 'what' from the 'how' of a task. The desirability of interaction and iteration of tasks can be understood if one realises that errors in the specifications can later carry exponential recovery costs. Also in practice passing iteratively through the specification and design process has proven to be desirable if not necessary, and the definite completion and documentation of every task one after the other has proven to be impossible.

When LCMs are observed closer, implicit cycles can be distinguished. For instance in the design phase in the software life cycle methodology the next step usually is detailing the product of the preceding phase, right up until the encoding in an executable formalism. In KADS the Requirements Engineering phases orientation, problem identification and problem analysis are considered to be three cycles of the same process in which every time deeper (other) aspects of knowledge are contemplated. The phases here prescribe the level of detail before the next check point has been reached, which makes the process implicitly cyclical. Instead of a fixed number of defined phases the management might also define the number of planned refinement cycles.

\textsuperscript{11} Particular tasks may be performed at the same time, for instance such as some preparations for the conversion.
3 KNOWLEDGE ENGINEERING

3.1 Definitions

No description of the term Knowledge Engineering could be found. It may be defined as the art or the ability to elicitate knowledge and to conceptualise and realise Knowledge Based Systems. An explanation of Knowledge Based System (KBS) is given in [Jack86]:

"...systems which solve problems by applying a symbolic representation of human expertise, instead of employing more algorithmic or statistical methods. In other words, KBS's attempt to encode the domain-specific knowledge of everyday practitioners in some field, rather than using complex and comparatively domain-free methods derived from computer science or mathematics."

The absence of a clear input-output relation is striking. In other words, there is no well-defined transformation from an input-state into an output-state. The term 'problem solving' rather indicates that a desired output-state is offered and the system is requested to reach this state by applying the knowledge stored in the system. One could say that the system is requested to generate the transformation that belongs to the input and the output concerned.

In a way this contradicts traditional software development where one wants to build a transformation machine that, given a certain input-state, supplies a well-defined output-state. A traditional system can obviously be considered to be a finite state machine (modulo the bugs) whereas a knowledge based system could, for the time being, perhaps better be considered to be a non deterministic machine. An important reason for this is that since we have the machine combine knowledge (according to rules), the number of possible combinations will easily increase explosively, which makes it almost impossible to predict which transformation the machine will generate.

With this characterisation a meta-level is reached, namely that of the dynamic transformation generation (KE), versus the statical transformation generation (SE). This also implies a shift in what should be researched to construct the system.

In Software Engineering "correctness-preserving transformations" are searched, capable of transforming a given specification (preferably mechanistic) into a working system that is the implementation of the specification. In the Knowledge Engineering "correctness-preserving elicitation" is searched, methods based on psychology to elicitate knowledge from experts in an undistorted manner. Perhaps the most important difference between KE and SE is that SE hardly bothers about how to get a (correct) specification whereas KE is centred around just that aspect. Conversely KE still gives little attention to formalisms for writing down the knowledge or to transform the thus begotten specification into a system, whereas SE concentrates on that aspect. The latter will probably also have been caused by the fact that KE still is a relatively young discipline in science.

3.2 The Necessity for Knowledge Based Systems

Complaints based on experiences with traditional software systems, that systems do not fulfil their requirements, are mainly due to conceptualisation problems of analysts and designers. Analysts and designers must put themselves in the user's place and must understand completely why and how the users perform certain activities to be able to
represent this in a formal specification later on. How thoroughly they have to understand the user's world will be made clear by reminding that often organisational or procedural inconsistencies in the object organisation are discovered during a system analysis. In some handbooks for business administration this screening is even mentioned as one of the advantages for doing a thorough system analysis.

Furthermore traditional systems are designed for a number of previously quantified (actually: quantifiable) problems with parameters that are within accurately fixed bounds (the finite state machine). All relationships between data (knowledge) have been classified in explicit procedures. The system is no more than a passive tool that is only able to follow accurate instructions of the user, a kind of super calculator (increase of abstraction for the end user by relieve). A well known expression that recognises this problem is: "If you can't turn programmers into specialists, you must turn specialists into programmers" [Martin82b]. One attempted to develop "end-user processing" using non-procedural (fourth generation) languages. Actually this is an attempt to have the user tell the computer directly his mental model.

This does not solve the problem of the vagueness that users often have about their mental model (which is one of the causes that make systems deficient). In knowledge based systems however the "best" experts in the field can be selected and their mental model can be elicited and implemented. The implementation of this model can support every user with actualising his or her mental model used during the performance of the task concerned.

With knowledge based systems one attempts to build a model of the real world, but the model usually is (unconsciously) in the mind of an expert. An attempt is made to analyse the structure of the model: what does the world of an expert look like, which goals and tasks does the expert set when solving problems (how does the expert generate the desired transformation). In principle there is no need for the knowledge-analyst to check the validity of the domain and the relations. That is the responsibility of the expert. The analyst tries to draw a map of the structural characteristics of the knowledge domain, based on the way the expert handles it. He tries to classify concepts in the manners in which they can be used and he tries to classify processes in the types of input concepts that use them and the types of output concepts that are supplied by them. Every time he can add a layer that reasons about the layer below it (by considering the layer below it to be the domain and by identifying structural characteristics in this domain).

On the lowest level of the domain the procedural aspects will still play a role, for instance how to obtain certain data from the outside world.

3.3 The Origination of a Knowledge Engineering Methodology

Only recently Knowledge Engineering has gained the status of a discipline in itself; before that, all there was were knowledge-engineers who had heuristic experiences while prototyping knowledge based systems. However, only when standardised methodologies are available, with methods and techniques that can be learned by everybody, can it be acknowledged as a true discipline.

The previous experiences with prototyping have indeed supplied sufficient data and insights to be able to develop methodologies for the development of knowledge based systems. However because of the enormous investments needed for their development (dozens of man-years) very few serious attempts have been made as yet. A striking exception is KADS, Esprit project P1098, that is about to enter the stage of maturity by now. KADS means Knowledge Acquisition and Documentation Structuring, a name that does not convey the contents of the methodology anymore.
Just as with traditional software development, one wants to build maintainable systems. The usual project problems will arise, like manpower, quality and transferability, and methodologies are used to reasonably guarantee the realisation of the product, notwithstanding these problems. In these aspects a knowledge engineering methodology will be hardly any different from the usual (project-oriented) methodologies.

The contents of a knowledge engineering methodology however differs in many elements from the traditional methodologies. This relates mainly to the products and the methods and techniques. Cause of this is the difference in the model that is to be incorporated in the computerised system: one attempts to represent a mental model of a particular world in (semi) formalisms and one is only concerned with the quality of the elicitation and transformation.

3.4 Knowledge in KADS

The KADS-methodology focuses on the quality of the transformation from mental model (M0) into conceptual model (M1), from conceptual model into design (M2), and from design, via the logical model (M3) into implementation.

![Diagram of models in the development of an Expert System](image)

**Figure 6** Models in the development of an Expert System

Figure 6 presents these models and on which levels of abstraction they are. The horizontal axis can be considered to represent the steps or transformations needed to construct a system, the vertical axis represents the extent of abstraction needed. The dashed line and the labels represent the levels of abstraction that are identified by Bachman ([Bachman79])\(^{12}\), the solid lines are the ones used by KADS. The rectangles are the sequential KADS models that are produced. As a comparison the dotted line marks the level of abstraction in prototyping. On the highest level of abstraction, the epistemological level, knowledge structuring primitives are used rather than specific knowledge primitives. On this level one concentrates on the type of structures; on the lower levels one concentrates on data classification or on the construction of machine formalisms.

---

\(^{12}\) The context of his research is one of semantic networks.
3.4.1 The Acquisition of Knowledge

During almost all KADS-phases data is needed that must be incorporated into the system. The methods from traditional automation, the interviews with users and experts, are inadequate here, mainly because these are not methods, but merely are unstructured communications. No strategy is employed to obtain specific data, or to guarantees elicitation of correct data. This is another cause why systems to fail. Because of the usually considerable complexity of the domain and because the task of the analyst is not to understand the domain, but only to map it, it is essential to have good (interview) techniques at hand that objectively coach the analyst when elicitating and constructing the model.

Problems encountered while eliciting knowledge are for instance that the expert does not always consciously know why he performs certain activities. In that case he cannot be asked why he performs the activity; if asked, the expert will probably tell what he thinks he does (which is not necessarily what he actually does).

3.4.1.1 Types of Knowledge

The most basic type of expert-knowledge is that of highly automated skills, like typing, video-games or the pinball machine. This knowledge is extremely domain-specific and if, for example, the keyboard of the typist is changed, the performance will decrease dramatically. The knowledge is ‘hard wired’ and not flexible.

A related type of expertise is that of a large knowledge base of problem situations, associated with precompiled action patterns [Breu83a], for instance the operator of complex machinery. These kinds of knowledge are called shallow knowledge and these may lead to serious mistakes in situations in which a more profound understanding is needed (Tschernobyl, airplane crashes). This kind of knowledge is heuristic, that is, associations are made while it is not exactly known why these associations may be made. A synonym for this type of problem solving is empirical associations [Wie88].

It is obvious that this should not be the only kind of knowledge that is stored in a knowledge base. We are looking for the systematic problem solving methods that are maintained by the expert, for the meta-levels that he possibly unconsciously maintains, for the conceptual model of the underlying principles of the domain of expertise and for the strategies that he uses for problem solving. We are looking for the insight, or deep knowledge.

3.4.1.2 Elicitation Techniques

A problem with all elicitation techniques is that the interviewer or observer unconsciously influences the object of observation (the expert). If the expert is asked something while performing his duty, then his actions are disturbed and his desire to answer the question may result in incorrect answers. Good elicitation techniques should either eliminate these psychological factors or should weigh them and incorporate them into the results.

In KADS several techniques are advised that can be applied in different stages of the analysis process. In all cases the interviews should be recorded on tape and transcribed for thorough analysis later on by the analyst in a kind of hypertext system [Cacm88]. These are

Note that for example strategical knowledge often contains heuristic aspects: a clear division often cannot be made
transcripts also serve as a reference later on; for instance when constructing a rule one refers to a piece of the transcript of an interview where the justification of the rule can be found.

Table 1 presents a summary of the interview techniques maintained by KADS. A precise description of this can be found in [Edin87], chapter 1: Data Collection.

<table>
<thead>
<tr>
<th>METHOD</th>
<th>SPECIFIC DATA ON</th>
<th>STAGE*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Focussed interview</td>
<td>factual knowledge</td>
<td>1, 2</td>
</tr>
<tr>
<td></td>
<td>types of problems</td>
<td></td>
</tr>
<tr>
<td></td>
<td>functions of expertise</td>
<td></td>
</tr>
<tr>
<td>Structured interview</td>
<td>structure of concepts</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>mental models</td>
<td></td>
</tr>
<tr>
<td></td>
<td>explanation</td>
<td></td>
</tr>
<tr>
<td>Introspection</td>
<td>global strategies</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>justification</td>
<td></td>
</tr>
<tr>
<td></td>
<td>evaluation of solutions</td>
<td></td>
</tr>
<tr>
<td>Thinking aloud</td>
<td>use of knowledge</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>reasoning strategies</td>
<td></td>
</tr>
<tr>
<td></td>
<td>inferences</td>
<td></td>
</tr>
<tr>
<td>User-dialogues</td>
<td>modality tasks</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>reasoning strategies</td>
<td></td>
</tr>
<tr>
<td></td>
<td>interface behaviour</td>
<td></td>
</tr>
<tr>
<td>Review of data</td>
<td>additional data</td>
<td>2, 3</td>
</tr>
<tr>
<td></td>
<td>justification</td>
<td></td>
</tr>
<tr>
<td></td>
<td>support knowledge</td>
<td></td>
</tr>
</tbody>
</table>

* 1 = orientation, 2 = problem identification, 3 = conceptual model construction

Table 1 Interview Techniques

3.4.2 Knowledge Levels

In KADS knowledge is described and analysed on four levels. This is known as the four-layer model of KADS. The layers are the domain-layer, the inference-layer, the task-layer and the strategic layer and every layer acknowledges for different structures in the domain. Currently there are two views on this four layer-model [Breu88b]. Figure 7 shows on the left hand side the layers hierarchically and it can be imagined to be a cascade of interpreters. On the right hand side the strategic level represents the meta-knowledge about the domain and the two remaining layers are interfaces between them. The strategic level reasons about the domain in terms of potential inferences and it generates goals and puts tasks on the agenda to reach the goals with. These layers are discussed hereafter in more detail. See also Figure 8.
3.4.2.1 Domain-layer: Concepts, Relations and Structures

On the domain layer, the lowest level, the knowledge consists of concepts, relations between concepts (is_a, consists_of, caused_by, etc.) and of structures built with these relations. The knowledge in this layer is impartial to tasks and is passive. One could say that “all the knowledge is there, but it still has to be made to work”.

3.4.2.2 Inference-layer: Meta Classes, Knowledge Sources and the inference Structure

On the inference layer is registered which conclusions can be drawn from the knowledge on the domain level. However potential conclusions are numerous and this must be limited to the ones relevant to the domain. Thereto meta-classes and knowledge sources are identified.

Meta-classes tell which role domain concepts take on in the inference process. For instance, by associating the domain concept `switch' with the meta class `component' we know it can never be a malfunction (though it may be a malfunctioning component). The meta-classes of a concept may also change during the consultation. For instance, as soon as during a diagnosis a hypothesis is confirmed, it can take the role of symptom or cause. A
meta-class describes, on a higher level, knowledge about domain concepts (knowledge about knowledge). On the implementation level it can be considered as a number of rules and objects that reason about other concepts.

A **knowledge source** is a primitive process that produces new knowledge. Knowledge sources can be decomposed again, more or less in the same way that data flow diagrams can be decomposed. The specification of a knowledge source describes the source of knowledge or how to produce it. The specification of a knowledge source consists of:

- **type**, e.g. classify, match, compute;
- **method**, e.g. classification, inheritance;
- **support knowledge**, any domain concept on which the knowledge source may operate, e.g. is_a-hierarchy;
- **input and output meta-classes**, e.g. a number of symptoms as input and a hypothesis as output.

The meta-classes and the knowledge sources are related to one another in the *inference structure*. The relations in these structures represent interdependencies between the various inference steps, however, it does not present a strategy that says which inferences when, or in which order, to make.

### 3.4.2.3 Task layer: Goals, Tasks and Control

On the *task layer* the model describes a task decomposition that directs the inference process. This task structure describes how and when elements in the inference structure are used.

Psychology has shown that for problem solving human beings maintain typical task structures, called **generic tasks**. Generic tasks cannot be divided into more basic tasks (without affecting the functionality). These generic tasks are domain-independent.

Basically there are two types of generic tasks, **analysis tasks** and **synthesis tasks**. The purpose of an analysis task is to determine as yet unknown characteristics (attributes) of the system that is the subject of the research (using e.g. diagnosis). The purpose of a synthesis task is to supply a structural description of the system under research (e.g. design or planning). An important difference between the two is that the analysts task leaves the structure of the system intact, whereas the synthesis task attempts to create a structure.

A composite of the two tasks is the **modification task**, in which the system is analysed (possibly after synthesis of the system model) and after which the system is modified, for instance the repair of malfunctioning equipment.

A task is described by a **goal-statement** (e.g. obtain(data), compute(parameter), estimate(value)), by the **control-statements** if <state> then <goal> and while <state> do <goal> and by sequencing (goals that have to be reached one after the other).

**Interpretation models** are examples of generic task structures. It is the task of the knowledge engineer to choose an interpretation model and to refine this to model the specific task. Figure 9 shows an example of the inference structure of a generic task. The interpretation model contains a more detailed description of knowledge sources (the squares) of the meta-classes of the inference structure and of the task structure.
3.4.2.4 Strategic level: Plans, Meta-knowledge, Repairs

A strategic component generates goals depending on the current state of the problem solving process. Generally the order in which the problems are solved is fixed (fixed task-structure). The requirement for a (dynamic) strategy follows for instance from the diagnosis of diseases, in which expensive laboratory tests are postponed in favour of more simple investigations which may supply sufficient information in the first place. In order to be able to formulate a strategy dynamically, the system needs a monitoring component and a planning component that generates the goals.

Some recursion in the development process appears to arise now, namely the analysis and the design of the monitor component and of the planning component. Clearly these have their own task structure and their own domain. Now we can really speak of meta-knowledge, the knowledge about the domain that enables the problem solving process to proceed as efficiently as possible.

In KADS the description of the strategic layer has not been completed yet, however, a system can evidently be equipped with a monitor and planning component for which interpretation models already exist.

3.5 KADS Phasing
The KADS-methodology is constructed around the Life-Cycle-Model, in which the phases succeed each other sequentially, even though tasks can be completely or partly performed in parallel. KADS is classified on one axis into internal branch and external branch, and on another axis as Requirements Engineering (RE) and Design Engineering (DE). This classification is presented in Figure 10.

<table>
<thead>
<tr>
<th></th>
<th>internal</th>
<th>external</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR</td>
<td>determine scope of project</td>
<td>analyse present situation</td>
</tr>
<tr>
<td>RE</td>
<td>analyse static knowledge</td>
<td>analyse objectives &amp; constraints</td>
</tr>
<tr>
<td></td>
<td>analyse expert and user tasks</td>
<td>determine functional requirements</td>
</tr>
<tr>
<td></td>
<td>construct conceptual model</td>
<td></td>
</tr>
<tr>
<td>PE</td>
<td>estimate feasibility</td>
<td></td>
</tr>
<tr>
<td>DE</td>
<td>functional design</td>
<td>selection of methods (introduction, use and control)</td>
</tr>
<tr>
<td></td>
<td>physical design</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 10** KADS Phasing

3.5.1 Internal Branch: Analysis

The analysis phase, the internal branch of the RE of the KADS-methodology, is *model-driven*. In this approach the analyst selects a generic model (interpretation model) and modifies and refines this into a model of reality. The KADS analysis phase consists of the following main activities [Breu87]:
- knowledge identification and orientation
- knowledge conceptualisation
- conceptual model construction

3.5.1.1 Knowledge Identification

In this first step the analyst gains knowledge about the domain of the expert. He composes a lexicon and a glossary with words from the domain of the expert and reads textbooks about the subject. Purpose is that the analyst will be able to communicate with the expert and obtains a substantial notion of the domain. Also the suitability for representation of the domain in a knowledge based system is investigated.

**Techniques:** various interview techniques.

**Tools:** information management tools, like outline processors, graphic editors, browsers and for instance Hypertext.

**Output:** the lexicon and the glossary.
3.5.1.2 Knowledge Conceptualisation

The domain terminology is arranged in structures, like "is_a", "consists_of" and "caused_by", and tasks are described using action terms, like obtain(data), compute(sum), diagnose(symptoms).

Techniques: various interview techniques.

Tools: concept editors, together with the information management tools mentioned above.

Output: the structures.

3.5.1.3 Conceptual Model Construction

When constructing the conceptual model the structural characteristics of the knowledge are modelled and formalised into an epistemological framework. The conceptual model is a description of the reasoning on all levels of knowledge. The conceptual model consists of:
- the specification of the domain concepts with the necessary attributes,
- an inference structure,
- a task structure and
- the description of a potential strategic component.

Refer to paragraph Error! Reference source not found. for a description of these structures. The construction of this model, the output of the analysis phase, is the main focus of the KADS-methodology.

The construction of the conceptual model of the system requires an epistemological analysis. Here the question of "the right level of abstraction" plays a role: because a model is already a (necessarily) limited projection of reality, where on behalf of certain goal there has been abstracted from this reality, there is a chance, when abstracting too much, of loosing details about this reality. On the other hand, of one chooses the abstraction level too low, then one runs the risk of getting stuck in details (which is often the case in prototyping) and the model will insufficiently represent characteristics of the knowledge.

Marr [Marr82] already noted that "how information is represented can greatly affect how easy it is to do different things with it". In [Byl87] this is called the interaction problem, namely that on one hand the representation and inference method determines which problems can be solved, while on the other hand the problem determines which representation and inference method are needed. They classify generic tasks in type of problem, knowledge representation and inference strategy needed and propose to develop for each such task proper knowledge acquisition methods. Whether the level chosen for KADS is the right level, experience must prove.

A tool for the development of this conceptual model is the library of interpretation models. These models are domain independent conceptual models. The knowledge engineer uses these as blueprints for the domain that is to be modelled. This library is also incorporated in the KADS-tools, integrated with a graphic editor.

3.5.2 Internal branch: Realisation

In KADS the realisation phase, the internal branch of the DE, consists of the following main activities:
- functional design
- selection of methods
3.5.2.1 Functional Design

In the functional design the conceptual model is decomposed into a set of functional blocks. The external requirements, the analysis of the user task and the interfaces with other systems are included here into the KADS design process.

Every functional block is responsible for a particular (internal or external) behaviour of the system. Relations between blocks are input/output, control and decomposition. Examples of functional blocks are: explanation, data I/O, problem solving, control, etc.

![Diagram of the Design Process in KADS](image)

**Figure 11** The Design Process in KADS

Tools are amongst others: DFD's (Data Flow Diagrams) or SADT (Structured Analysis and Design Technique).

3.5.2.2 Selection of Methods

When selecting (AI-)methods the main point is which technical methods are appropriate to implement the functional behaviour of the system with. Methods often are quite abstract specifications how to refine data into knowledge or information, like hierarchical
classification (also known as diagnosis) or synthesis (forward chaining). These can also be traditional methods, like databases and parsers.

Methods usually must be composed of design elements, primitive elements from the disciplines of AI and computer science. Now the choice of the environment becomes important: which design elements are offered by the environment, is the environment open or closed, strong or weak?

For this task no tools are available yet. There is a demand for a library of methods and their potential decompositions into design elements. Requirements on the output documents of the have been formulated.

3.5.2.3 Physical Design

In the physical design, the design elements and the description of the functional blocks are combined, or further decomposed, into implementation modules. Their coherence forms the architecture of the system. Examples of the physical modules are: forward chaining inference engine, backward chaining inference engine, agenda, rule network, object network, etc. Classical AI-systems consisting of a knowledge base, inference engine and working memory, are an example of skeleton architecture.

Principles of the composition mostly have a background in software engineering, like minimal coupling with maximal cohesion, efficiency and non-redundancy.

For the support of this stage no specific tools have been developed (Knowledge Engineers' workbenches). Structure diagram editors can be useful.

3.5.2.4 Implementation

The implementation phase is not directly supported by KADS. This however is exactly where good shells and KADS touch upon each other: the shell offers the design elements, editors, debuggers and other tools to implement the physical design with.
### Environment vs. Methods

<table>
<thead>
<tr>
<th>Environment</th>
<th>Methods</th>
<th>Design Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMYCIN</td>
<td>Backward chaining of AND/OR tree</td>
<td>Rules</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Parameters</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Contexts</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rule interpreter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Solution trace</td>
</tr>
<tr>
<td>KL-ONE</td>
<td>Hierarchical classification</td>
<td>Frames</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Single world database</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Subsumption relations</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Classifier</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Database procedures</td>
</tr>
<tr>
<td>KEE</td>
<td>Hierarchical classification</td>
<td>Frames</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Production system</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Single world database</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Subsumption relations</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Classifier</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rule interpreter</td>
</tr>
<tr>
<td>KES</td>
<td>Backward search</td>
<td>Parameters</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Single world database</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Interpreter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Database procedures</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Interface procedures</td>
</tr>
<tr>
<td>Prolog</td>
<td>Depth-first backward chaining</td>
<td>Rule interpreter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Single world database</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Horn clause logic</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Database procedures</td>
</tr>
</tbody>
</table>

**Figure 12** Some examples of Environments

### 3.5.3 External branch: Non-functional Requirements

Specifying the non-functional requirements, the external branch of the KADS Requirements Engineering, is described in [Barth86]. The report does not contain a description of how to perform the tasks, but rather gives an exhaustive classification of the elements of the methodology from a management point of view (the mixture pointed out in paragraph *Error! Reference source not found.*). Still this report gives the most accurate information found on this aspect of the tasks. The role of (this part of) the RE is described as "to facilitate mutual understanding between a customer and an analyst" and is the part "...where the expectations of the users are investigated and recorded".

This specification of the non-functional requirements is not discussed here in detail, as there are a lot of similarities with traditional Software Engineering, from which it has been derived. Figure 13 (from: [Breu88c]) shows the relationships between the tasks of Requirements Engineering. Tasks 3, 5 and 7 form the internal branch. Tasks 2, 4 and 6 form the external branch, and tasks 1 and 8 form the project management activities. Figure 14 lists the names of the reports that are produced by the tasks and that are combined into the phase documents.

---

14 We have declined here from describing each report in detail.
Figure 13 KADS Life Cycle Model: Requirements Engineering
1. **Determine scope of project:**
P1: Background and prerequisites  
P2: Project terms and directive  
P3.1: Project Life-cycle model (LCM)  
P3.2: Project plans  
P3.3: Project organisation  

2. **Analyse present situation:**
R1: Model of present situation  
R2: Functioning objectives of the user organisation  
R3: Functioning problems of the user organisation  
R4: Task orientation  
F1: Feasibility estimate  

3. **Analyse static knowledge:**
M1: Lexicon  
M2: Static structure  
F1: Feasibility estimate  

4. **Analyse objectives and constraints:**
R5: Objectives of prospective system  
R6: Compatibility requirements  
R7: Man-machine interface  
R8: Development and operational environment  
R9: Control and security constraints  
R10: Organizational model  
F1: Feasibility estimate  

5. **Analyse expert and user tasks:**
M3: Selected Interpretation Model  
M4: Inference structure  
M5: Task structure  
M6: Strategies  
M7: User model  
F1: Feasibility estimate  

6. **Determine functional requirements:**
R11: Functional requirements  
R12: System structure  
R13: Information requirements  
R14: Expected future enhancements  
R15: Consequences  
F1: Feasibility estimate  

7. **Construct Conceptual Model:**
M5: Strategies  
M7: User model  
R16: Knowledge base requirements  
F1: Feasibility estimate  

8. **Estimate Feasibility:**
R17: Development requirements  
R18: Validation procedures  
F1: Feasibility estimate  

**Project Management:**
P4: Project evaluation  
P5: Project reports  
P6: Minutes  
P7: Information and training plans  
P8: Correspondence  

**Figure 14** Reports of KADS Requirements Engineering
3.5.4 External branch: Introduction, Usage and Maintenance

This aspect of the KE-methodology has received little attention in KADS as yet, with the exception of document P7 (Information and training plans). This will also include the specification of procedures, system manuals, organisational preparations, etcetera. Other aspects that are of importance are maintenance, for instance agreement on the availability of the knowledge engineer and analyst, maintenance and extension of expertise (human and system expertise).

3.6 Other Work and Evaluation

3.6.1 Other Work

**Buchanin** [Buch83] describes an evolutionary method for the development of expert systems. In principle this should be considered as (rapid) prototyping, which is typical for the development of first generation expert systems. Relating to previous remarks it will be obvious that this prototyping was necessary to collect the empirical material needed to develop a true methodology.

The article does describe a number of phases in the development (identification, conceptualisation, formalisation, implementation and testing) but an early construction of the first prototype is strongly advised upon ("A common error is waiting until the knowledge base is close to complete before programming."). The article furthermore means with the word ‘tools’ only Expert System Shells and not elicitation or structuring tools. According to the KADS philosophy the ES-shell is an implementation formalism that in principle does not affect elicitation and conceptualisation. This prototyping method causes the problem that the knowledge engineer soon starts thinking in implementation structures which makes the problem and the solution inseparable. Because techniques are missing, much importance is given to previous experiences of the knowledge engineer.

In [Breu88] the following causes for the poor expertise in the first generation expert systems are mentioned:
- shallow knowledge;
- knowledge is implicit;
- no differentiation in types of knowledge;
- no explicit control;
- one method of reasoning;
- no meta-knowledge;
- no user model.

KADS rejects prototyping because of the costs, the time it takes, the lack of structure in the implemented knowledge and because of the danger that a (wrong) method, once chosen, will be maintained. Finally KADS has the opinion that a prototype can be used as an interpretation model with which verbal data can be interpreted\(^\text{15}\), but that it is not the only model [Breu83b, p.59].

---

\(^{15}\) This remark also goes for prototyping in the conventional software development: the prototype serves as a subject of discussion with which the client validates the analysis and with which the development of the model can be continued.
Freiling [Frei85] describes a step-by-step approach for the prototyping of expert systems. This method, that has already been used for some time at Tektronix, consists of two phases, each of which consists of several steps. These are the knowledge definition phase (familiarisation, knowledge organisation, knowledge representation) and the prototype implementation phase (acquiring knowledge, inference strategy design, interface design).

In principle this work should be classified with the prototyping mentioned before, however, the phasing is considered to be a guidance for inexperienced knowledge engineers, a basis for communication and as a measuring method for management. Based on experiences, a general language has been developed for the specific domain (GLIB, General Language for Instrument Behaviour) and tools have been developed to build these instrument trouble-shooter systems with (INKA: INglish Knowledge Acquisition, PIKA: Pictorial Knowledge Acquisition, CHEKA: Checker for Knowledge Acquisition).

Evaluation: This is a next step on the way to a knowledge engineering methodology. The value of this approach lies in the prescriptive nature and in the supporting tools and techniques that are not directed only towards the implementation.

ROGET [Benn85] is a knowledge based system for the acquisition of knowledge for diagnostic expert systems. The most important task of Roget is the construction of the conceptual structure of the system to be developed. For this Roget has dialogues with the expert and guides this dialogue using a set of advice- and proof categories that have been stored in its knowledge base. The construction of this conceptual structure is mainly achieved by investigating differences and similarities with existing expert systems that Roget knows about. Roget starts with the selection of a conceptual structure. During the dialogue, categories are added or removed, depending on the similarities or differences observed. The contents of these categories are recursively elaborated in dialogues with the user.

Evaluation: Roget is based on one generic task-model only, namely that of classification, which is because Roget is limited to EMYCIN which was expected to become a powerful and generally used tool. The possibilities of Roget are thus restricted to the models that are present in the knowledge base. By extending Roget and first letting it select the most suitable task model (interpretation model) together with the user, Roget may become a good complement of KADS. Having determined the interpretation model, Roget can turn this into the conceptual model in dialogues with the user.

Bylander and Chandrasekaran [Byl87] identify an interaction problem, namely that on one hand representation and inference method determine which problems can be solved, while on the other hand the problem determines which representation and inference method are needed (see also Error! Reference source not found.). A generic task is defined as "basic combinations of knowledge structures and inference strategies that are powerful for solving certain kinds of problems". Generic tasks are introduced at the "right" level of abstraction as a combination of the type of problem, representation method of the relevant knowledge and by determining a most proper inference strategy. The argument is that different knowledge acquisition methods are needed for different kinds of generic tasks.

Evaluation: The generic tasks in the article are the same as those in KADS, although KADS does not advise acquisition methods for task models and neither does KADS limit an interpretation model to some kind of representation. The article positions generic tasks on a somewhat lower level of abstraction than KADS does. It seems possible to integrate this approach into KADS and this should then be positioned after the Knowledge Identification phase since by then a general view on the domain and the problem area have been acquired. Also the KADS analysis cycle could be executed once, in a restricted way, to get
a prototype on paper that can be used to select the "generic task" (this cycle is caused by the fact that before the generic task selection according to [Byl87], the task, representation, inference strategy and relevant data must partly be known, whereas with KADS the interpretation model is selected only after most of the data is known).

3.6.2 KADS in Practice

This paragraph lists the conclusions and results on experiences with KADS that have been found in a number of papers

3.6.2.1 StatCons

StatCons [Greef88] is a knowledge system that advises in the field of statistics. The domain is called statistical consultation and it concerns advice on the planning of data collection and statistical analysis. It is classified as a difficult domain: it concerns a design task with complex objects and in the situations of advice a complex interaction between the system and the user may be necessary, depending on how far the user has progressed in planning and actions. More specifically it concerns a system that gives statistical advice to students or employees who want to do a psychological research.

Evaluation: The following remarks are made on KADS:
- The one-sided attention for what is called the "problem-solver" is considered to be a problem. What is missing is a clear distinction between the analysis of the problem solving task (the work that has to be done, no matter who does it) on one hand and the teamwork on the other hand (who does what and which objects need to be exchanged). This resulted into complex and vague task decompositions.
- The question is whether the "analysis-before-implementation"-principle of KADS (without prototyping) results into better systems or into systems with shorter development time.
- The modelled approach of KADS was experienced as being very valuable. It enabled a refining structure in which decisions on details could be postponed.
- The KADS-documentation with the analysis model for the problem solving task has enabled a separate team to design and implement the StatCons-1 system.

3.6.2.2 Mixer

In [Billau88] the knowledge acquisition and the construction of the conceptual model for a configuration task using KADS is reported. The domain concerns the design of standardised mix apparatus in the field of chemical engineering.

Evaluation: According to the article the utility of KADS is beyond dispute. Further more the article contains the following remarks:
- In the beginning the assumption was that it concerned a synthesis task, what later on appeared to be a configuration task. This confusion existed because to an expert the names of his activities are not so important, whereas in the KADS vocabulary more preciseness is required.
- One states that interpretation models actually only contain a theory on domain-specific reasoning processes, that have to be validated by constructing the model of this realistic reasoning process (description versus prescription).
- One suggests the use of (the equivalent of) Data Flow Diagrams (DFDs) for the detailed development of the conceptual model. This concerns the refinement of the inference
structure diagram from the interpretation model into the diagram of the conceptual model, as well as the possibilities of the data dictionary regarding knowledge sources and meta classes. From the context level on, where the diagram represents the complete reasoning process, one decomposes onto a level of primitives, where knowledge sources represent the primitive inferences. This approach seems to be in closer agreement with the manner in which the inference structure is analysed in reality.

3.6.2.3 Autopes

Autopes [Wages88] is a research for a knowledge system that advises operators of an industrial process installation about the control of their processes. Because of the technical character of the equipment and thus also of the operational electronics (real-time systems) also a logical system model has been developed with the help of Structured Analysis.

The goals include gaining experience with the development of a knowledge system. The article tells KADS is used for the first time and at present the phase "Construct conceptual model" is being passed through.

Evaluation:
- To avoid confusion about the meanings of terms in the field of software engineering simplified, terminology is used.
- Knowledge sources and meta-classes are described in domain terms (and not only in generic terms from KADS). In [Breu88b] the comment is that "it appears the vocabulary for meta-classes is soon exhausted and replaced by domain terms...which had the advantage that links between the domain layer and inference structure become explicit from early on".
- The question is whether a detailed specification on the strategic level is really necessary (usually this implies that a static task structure suffices and that little control knowledge is needed).
- A knowledge model and a logic system model can be used complementary, especially when modelling conventional aspects of a knowledge system.
- A first, cautious conclusion is that KADS may be useful, but adjustments are necessary (within this project).

3.6.3 Checklist KADS

A: Methodological Requirements

A1: The terminology is precise and unambiguous: confusion is out of question. To a large extent this goes for KADS. Terms may be mixed up with terminology from the outside world but within the methodology the terminology constitutes a linguistic whole.

A2: The task decomposition makes the complexity controllable. This also goes for KADS, although people with practical experiences vote for smaller steps.

A3: Every phase delivers a well-defined and well-tuned product. Well-tuned: yes, in particular as the consequence of the linguistic frame; whether it is well-defined or not should be investigated yet.

A4: The tasks can be performed cyclically, separately or as a whole. They can separately. Not as a whole. The latter aspect can possibly cause a maintenance problem (when the final system is maintained instead of the specifications).

A5: Methods and techniques: much attention is given to the methods and techniques used for the construction of the conceptual model and they seem to be good and complete. For design and implementation the methods and techniques are
described sparingly and within KADS this is a subject for further research.

A6: Practical tools: The KADS Power Tools (KPT) are still in development (the new version is called Shelley) and are not (yet) generally available. However, commercial development in my opinion should be considered as an industrial activity.

B: Requirements regarding the Subject Organisation

B1: Effective: Experiences prove that the methodology is effective and yields the result aimed at.

B2: Efficient: Not producing prototypes and still achieving the goal make the methodology very efficient compared to other practices in the knowledge engineering.

B3: Transferable results: Yes, refer to 3.6.2.1.

B4: The methodology can be learned: Obviously they can, speaking from experience.

B5: Stimulating communication: The unambiguous documents make communication and transfer of the results easy.

B6: The methodology should be complete: No. Design and implementation should be improved as yet. Introduction, use and control are still missing almost completely. The latter should not be a drawback and for this traditional methods may be used. Design can also be implemented as a "black box decomposition", with which has been sufficiently experienced in the software engineering.

C: Requirements regarding the Object Organisation

C1: Supporting strategic planning: No. Also in the traditional computerisation this is not a direct part of any methodology so perhaps it should not be required of KADS. Of course KADS can be complemented with BSP and CSF-analysis. To what extent the planning of knowledge systems differs from the planning of the traditional systems is not clear.

C2: Supporting organisational transition behaviour: No, because of the absence of support of the phase introduction, use and control. Here also knowledge from the traditional computerisation can be useful.

C3: Maintenance: No. No directions for maintenance are given. The LCM approach can even cause a maintenance problem, the way it is known in the traditional IT (see 4.4.1).

C4: User participation: Partly. The methodology desires a steering group during the RE. However the depth (which sections of the organisation have been absorbed) is determined by the object organisation and not by KADS. For knowledge systems in particular, where users too soon fear that the system will make human beings superfluous, this may cause serious problems when it is neglected.

D: planning and control

D1: Planning method: No. Standard methods seem to be adequate here, although there is not enough experience to assess the weight and the time of phases (well enough).

D2: Pre-calculation and post-calculation: No. Again standard techniques can be used for this, however, only after some time of experience they can be incorporated in the right way.

D3: Measuring points: Yes, the phase boundaries where every time a feasibility statement is stationed. However, criteria for the assessment are not worked out.

3.6.4 Evaluation

The results achieved with KADS prove KADS to be a very useful methodology for the
development of knowledge systems. The checklist demonstrates that KADS has been well
developed in respect of methodological aspects and subject-organisational aspects (the
software house). The advantages of KADS mainly lie in knowledge acquisition and in the
construction of the conceptual model. The model driven approach seems to work really
good in the complex process of retrieving the thinking model of the experts. Regarding the
object organisation and the planning and control KADS hardly has been developed. This
however leaves room to incorporate KADS in organisations that have been maintaining
these methods and techniques for some time already. KADS provides the specific tools to
build the knowledge systems with; the existing organisation provides the tools for control,
introduction and maintenance. The classification according to the Life Cycle-model speaks
for itself but chapter 2 has demonstrated already that phase by phase finishing and
documenting does not really work in software engineering anyhow. There are indications
that also in KADS it does not work satisfactorily.

Still the value of KADS lies in the offer of a first methodology for the development of
knowledge systems at a time that these are wildly successful but nobody really knows how
to do it. In this sense KADS is the recipe that can be used to gain the first experiences. In
later projects however methodology configuration will have to be adopted to adjust it to the
project concerned. Here is emphasised again that "the method does not do it" but the
quality of the designers and the analysts does. However, only by doing, designers and
analysts will get experienced.
4 SOFTWARE ENGINEERING

4.1 Introduction

Software engineering may be compared to building bridges: the bridge will have to meet certain specifications, like length, weight, capacity, etcetera, but the designer is free to some extent to choose the shape of the bridge: numerous shapes will meet the required specifications. The designer is guided by esthetical norms and values and continually weighs "which means are available" and "which combinations are the best/prettiest" against each other.\footnote{That is why software has also been protected by copyright [Ogilvie88].}

Software engineering therefore can also be classified into the construction of systems and the construction of tools, respectively the top down aspect and the bottom up aspect\footnote{Do not confuse the terms with the use of these terms in design and implementation. Here a classification in the discipline is meant.}. Boehm unites both aspects in his definition:

Software engineering is the application of science and mathematics by which the capabilities of computer equipment are made useful to man via computer programs, procedures and associated documentation.

4.2 Software Engineering: Top Down

4.2.1 Analysis: the World Model

Starting point of the building process of every system is a specification of the system. The specification is a model of the world in which the system occupies a position and in which it has to perform tasks (also see 2.2.3).

Considering what the techniques model, it seems to be justified to assume that for the specification of all computerised systems it is sufficient to make a world model consisting of data representing the world and of tasks or processes, that operate on these data. This model is complemented with external requirements like performance, capacity, etcetera. This idea is confirmed by analysis methods for business information systems, where data analysis and function analysis are applied ([Bem84] p.210, [Bulcke84] p.309, [Martin82a] p.201). Also it appears that the structure of the data collections does not change whereas the tasks do change or are added [Martin82a]. That makes it possible to map and organise all of the data collection. It is as if a "concrete foundation" were laid under the building complex of the information system and tasks can be added or modified without needing to adjust the data collection. Figure 15 presents this in a simplified manner.

The most important differences in the specifications of the various kinds of automated systems now lie in the depth of the data analysis and task analysis. In technical systems for instance the relations between phenomena are often formally defined in books or can easily be obtained from an expert. In (business) information systems these relations often have to be obtained by an intensive analysis process.
Just like in KADS this modelling takes place in several steps, although nowhere in the literature has been discovered (by me) a complete or coherent description of these steps, equipped with a coherent collection of methods and techniques. For this reason here is chosen to describe some techniques that in my opinion can be used together to implement this modelling with. Again we assume that the modelling takes place in three steps: data collection, data classification and synthesis of the conceptual model.

Figure 15 Interaction between World and System

4.2.1.1 DFDs and SSA

Data Flow Diagrams (DFDs) and Structured Systems Analysis (SSA\textsuperscript{18}, [Gane79]), are similar techniques that analyse the world in a top down fashion by decomposition. The starting point is the world of the user or the expert and this is mapped as a hierarchy of diagrams of processes and the data that is exchanged between them.

The methods start out with interviews with the user or the expert. The information elicited is expressed in diagrams and in a data dictionary the sort, type and use of data is registered. The diagrams rather present a description of tasks or processes than a description of objects and their relations. In the model tasks are described and the entities that play a role in it. The task process, that is how the input is transformed into output, should be described separately. Usually this is done in a kind of pseudo-code.

\textsuperscript{18} SA, SSA and DFDs are more or less similar techniques
An advantage is that the diagrams are also easy to grasp for the user, which enables the analyst to validate and refine his picture fairly easily. This makes DFDs an excellent method for the initial analysis (current situation). They map, starting out with the user, possibly overlapping parts of the user's world. The model supplies a (not necessarily complete) picture of the data used (data-dictionary). For the construction of the conceptual model the method is not so suitable because it does not produce a coherent data model. There is, for instance, no way of representing entities that are not involved in any process. Cameron in [Cam87] finds that every decomposition implies a commitment for a particular system structure so that decomposition is a method for description and not for system development.

4.2.1.2 Entity-Relationship

The Entity-Relationship-model (ER) has been introduced for the first time by Peter Chen in [Chen76]. Since then it has become widely known as it offered a method not depending on DBMS to design the conceptual data model with. ER-models can be represented in the more implementation driven, hierarchic, network and relational models (refer to [Car87] for more references on these representations). Also DBMSs based on the model exist so that one could call this executable specifications.

Entities are described by Lindgreen in [Lind87] as "Things about which we wish to inform" and they are contained in systems that he describes as a collection interrelated phenomena that form one coherent whole, that is related in a sensible way to the environment, and he adds: "Systems are not a priori. They are, if someone perceives them to be." As kinds of phenomena in the systems theories he mentions among others:
- Activities, that change the state of the system and/ or the environment;
- Events, that activate agents;
- Data, that represent the information that is exchanged between the agents;
- Time, in regard to the existence of entities for instance and the duration of activities.

Just like with systems entities are not a priori, but they are "if someone perceives them to be". In the ER model phenomena are represented by entities and relations between entities. Relations are associations between data that usually can be considered to be new data. For instance the relation between employee and department produces the information at which department the employee works. That allows a phenomenon to be an entity in one model and a relation in the other model. Only in the same model a phenomenon can never be an entity as well as a relation.

Ever more people do accept though, that the ER model does not have enough manners of expression to be able to represent all phenomena in the right manner. Dubois in [Dubois] mentions as one of the shortcomings the insufficient support of dynamic characteristics (of systems) and he introduces the EREA model (Entity-Relation-Attribute-Event).

In [Cars87] roughly a method is described to go from SA to ER. He considers DFDs as possibly overlapping "views" of the system and he uses view integration techniques to synthesise them into one (non-redundant) ER model. For the process description of this conceptual model he sticks to SA.

---

An attribute may be considered to be a field of a record that may contain a value with which something for us interesting of/about the entity (the phenomenon) is registered.
4.2.1.3 JSD

Jackson System Development (JSD) [Camp87] models a system as the sequential events that an entity may pass through and the activities that are involved in this. A JSD specification mainly consists of a network of communicating sequential processes that communicate message passing and that can read each other's variables (CSP, [Hoare78]). This model is an instantly executable specification and lends itself for prototyping and as input for program transformation systems (see 4.4.2).

Even though it is clear such a specification is close to implementation, it is not clear how to obtain such a specification. For instance to me it seems hard (refer to example in [Cam87]) to retrieve from a book in the library which events it passes through when being lent (the "circulation" of the book in the library). The book is not able to tell me and, being an analyst, in principle I do not know the specific processes in the organisation (in this case the library). It is possible for instance to retrieve this circulation by cross-referencing, from DFDs that models the tasks (of people). Furthermore in [Mees87] the absence of data modelling possibilities is called a shortcoming of JSD and a method is roughly described to use ER in JSD.

Thus as an analysis method JSD seems to be unfit. As a task- or process modelling method it does appear to be fit and then it is close to an implementation. A manner to go from DFDs to JSD could not be discovered.

4.2.1.4 Conclusion

A combination of DFDs, ER and JSD may be a good instantiation of the analysis process to obtain a conceptual system model. DFDs then are used for the initial analysis phase and task description. Afterwards ER is used to obtain the conceptual data model from the DFDs and at the same time JSD is used for process description.

None of the three techniques seem to be fit in themselves for the construction of a conceptual model and together the three techniques do not (yet) form a linguistic whole.

It is also striking that, despite the enormous amount of models that have been designed for the benefit of systems, no model-based method exists as yet. I suspect that in this case the "right level of abstraction" (see 3.5.1.3) is concerned.

4.2.2 Realisation

When realising, designing and implementing the systems, the external constraints are involved in the engineering process. This concerns speed, reliability, capacity, etcetera. The designer makes implementation decisions based on these constraints, like choice of data structures, dynamic or static memory allocation, optimisations etcetera. If a choice is hard to make, usually the specification is incomplete or ambiguous. Designing thus brings to light shortcomings in the specifications and next initiates an interaction with the requirements engineering.

Clear methods to go from specification to design could not be discovered. Mainly rules of thumb and common sense are used.

Usually the specification is decomposed top down into functional blocks that each must realise a particular system behaviour or a part of that (black box decomposition). Next is
determined which system functions are needed for that and implementation modules are
designed that realise these system functions. Finally the functional blocks are composed
from this (compare to figure 3.4, the KADS design process).

A rule of thumb for the functional decomposition includes that a module does fulfil one well-
defined function. Rules of thumb for the modularization include:
- a maximum of cohesion with a minimum of de-coupling, the boundaries being at the
  points of minimal interaction and dependency;
- non-redundancy, where operations that more often appear as a procedure are
  implemented;
- abstraction hiding, where all operations on a particular data-type or object are united in
  a module, where the only access is via procedures.

As tools in the design process usually structure diagrams are used (structure charts
[Gane79], control flow charts, Nassi-Schneiderman), but the general critic is that the
diagrams are huge and obscure and only show a limited number of aspects of the software.
Usually the diagrams are complemented with (ambiguous) natural language or pseudo-
code. In this traditional method it should be verified time and again whether the specification
and the design still agree with each other and the final product must be scrutinised for
mistakes. Software that has been developed in this manner can hardly be said to be correct:
not finding any mistakes does not imply that there are no mistakes. Searching for methods
that supply definitely correct software, therefore continues undiminished, that is mainly
searched in formalising the product software.

4.3 Software Engineering: Bottom Up

This aspect of software engineering implies the supportive tools and techniques with which
the systems are designed and built. Two kinds can be distinguished here:
- implementation formalisms in which implementation decisions can be expressed, that
  next can be translated into object code automatically and
- tools to relieve people from their tasks.

4.3.1 Implementation Formalisms

The implementation formalism abstracts from the machine to get an executable formalism.
Examples are assemblers and programming languages, but database management
systems too may be classified as such a formalism. An important characteristic that is
typical of nearly all the formalisms is, that an attempt is made to limit the number of potential
interactions to a convenient number, with which it would be easier to control the complexity.
Examples are scope-rules like global and local variables, modularisation (ASF [Berg87],
Modula-2), type-checking and the elimination of the goto. However, a lot of attention is still
directed towards implementation decisions, like the use of machine resources (memory,
cycles, I/O) and the whole concentrates on tools to supply an efficient program with a
correct syntax and a modular structure and which is readable, maintainable and reusable.

20 The traditional example is the one of the imaginary numbers that as real and imaginary part, or as modulus and
corner can be implemented. With abstraction hiding the implementation can change now without any impact on the
modules used.
4.3.2 Tools

Tools vary from editors, debuggers, etcetera (low level), to complete, integrated programming environments (high level). Especially in case of the latter the distinction between tool and method diminishes: the tool implements a method (top-down) or the tool makes up a method (bottom up).

Programming environments usually consist of a number of tools that are more or less integrated. The effectiveness of a programming environment is achieved in several ways. As a kind of spectator the system can help the user to find information quickly, to present it effectively, and to modify it in a simple manner. The tools have an advanced user interface (windows, menus, mouse), maintenance and storage of information (data-dictionary, database), debug-facilities, incremental compilation, etcetera. Besides this, more advanced environments have incorporated the most important abstract operations and objects from a particular application area [Sheil83]. These enable the user to concentrate thinking in these abstractions and gains advantages from the implementation and design knowledge that are represented by these abstractions\textsuperscript{21}. Finally there are (AI-)systems that more or less make use of knowledge for the development of the programs. Examples of such systems are mentioned in paragraph 4.4.2, program transformation systems.

4.4 Trends in Software Development

This paragraph wants to give an overview in the trends in the methods of software development. The most important observation is that nowadays there is a stronger tendency to consider software as a formal object, rather than as an industrial product as is implied in the software life-cycle [Feij86]. This consideration has the consequence that one wants to specify software in a high level, problem oriented language that delivers executable specifications. This solves an important part of the maintenance problem\textsuperscript{22}: modifications in the functionality requirements of the software are processed in the specification and because these are executable, a new version of the software is generated.

Follows the discussion of three approaches:
- the conventional approach,
- the transformational approach [Partsch83],
- the operational approach [Zave84].

4.4.1 The Conventional Approach

The most existing software is developed under direction of the life-cycle model (also see 2.4). The course of development is divided in a number of phases and each phase brings the developer closer to the goal. The primary goal of the life cycle methodology is the division of the development process into phases and interfaces between the phases for the benefit of controlling the complexity. The starting point for the realisation in this approach is the top down decomposition of black boxes while as little interaction possible with previous

\textsuperscript{21} Embedded abstractions however are only useful within the domain for which they have been designed (for instance fourth generation languages). If the domain is expanded, then the strength of the abstractions may diminish [Sheil83].

\textsuperscript{22} Sometimes more than half of the of the capacity of the automatization departments is devoted to maintenance. The result is that there often is no time left for new developments to replace outdated systems, which makes the claim on the capacity by maintenance even bigger.
phases is allowed to obtain a process that is controllable for the management.

Advantages
- division into phases for the benefit of the control of the complexity (work-breakdown), which makes the development controllable (in the sense of time and money);
- the possibility that each business can instantiate this generic methodology with (own) methods and techniques for every phase.

Disadvantages
- methods and techniques usually do not form a closed linguistic whole (gearing of products to one another);
- usually the final product is maintained instead of the specification. After a while it gets extremely hard to understand the product at all, as the specification cannot be used as the documentation of the product anymore;
- the necessity of interaction between requirements-engineering and design-engineering is not acknowledged. The inherently cyclical aspect of the analysis and design process is neglected;
- reuse is made difficult because this nearly always concerns the code and that is designed with a specific application in mind.

4.4.2 Program Transformation Systems

Since recently the so called correctness retaining transformation has received ever more attention as a method for the development of programs. The purpose of this approach is to have the software-production cycle passed through as controllable and automated possible, starting out with a formal specification. The main idea is that software is a product that can be defined formally and the sooner these definitions can be caught in a closed definition, the sooner an automated process can generate an implementation. Another purpose is the maintainability of software. The automated process namely does allow to modify the specifications (as they should be) and next, to generate in a cheap and simple manner a new version. Furthermore the system documents all implementation decisions during the transformation process, so that modifications in these decisions may optimise the program structurally (for instance hashing versus linked lists).

The advantage of this approach is the guarantee that the product fulfils the specifications as well as that the product is correct and maintainable. In other words the specification of the system is the definition of the system.

Transformational programming is a methodology (method) that constructs programs applying transformation rules successively. Usually this process starts out with an initial specification and ends up with an executable program.

Transformational systems are computerised systems that support transformational programming. Some transformational systems have as input an incomplete specification and develop this into a complete one; other systems manipulate a complete (formal) specification until an implementation is obtained.

Generally transformational systems contain one or more knowledge components that, together with the user, select transformation rules. Below the examples PSI (1979), CHI (1982) and (CIP) [in: Partsch83] are described.

4.4.2.1 PSI
PSI (Stanford) is a big LISP system designed for the synthesis of efficient programs. The input of PSI is a specification that has been obtained in a dialogue with the user. This specification contains among other things natural language and examples of calculations.

PSI has been designed as a collection of experts for various tasks. The acquisition group is responsible for obtaining specifications from the user. Various experts in this group elicitate information from the dialogue and turn these into program fragments that form the input for another expert, the program model builder (PMB). That person delivers a program model that is the interface with the synthesis group that consists of a coding expert and an efficiency expert.

The Program Model Builder gets as input program fragments that may be incomplete, ambiguous, inconsistent and arbitrarily arranged. Building a complete program model is an incremental process consisting of gathering information and next adjusting a partial model. If the PMB is not able to solve a particular problem, the acquisition expert is called on for help or in the last resort the user himself.

The coding expert, called PECOS, refines the program model of the PMB by applying rules. If several rules are applicable, the refinement is split up so that a tree is developed. The roots of this tree is the original specification, the leafs are the programs in the goal-language and every path in the tree is a series of sequential refinements. To limit the size of this tree some possible refinements are rejected and, making use of the user's advice, some heuristics and the efficiency expert, usually only one path is followed.

The efficiency expert, LIBRA, advises the coding expert and thus helps with making the design decisions. Besides planning rules the efficiency expert uses flow analysis and the upper- and lower limits of the costs assessed (memory, storage, machine cycles, etcetera) are incrementally calculated. This costs analysis identifies parts that form bottlenecks in the refinement process, so that these can receive extra attention during the refinement. Also see [Kant81].

4.4.2.2 CHI

CHI (Stanford) builds on the PSI experience. Instead of a number of autonomous experts CHI uses a homogeneous set of tools that uses one huge knowledge/database. The major system components are:
- an object oriented database of programming knowledge;
- a knowledge base manager who saves contexts and versions;
- a structure based editor who modifies programs and synthesis rules;
- an agenda mechanism to guide the user and to help with heuristic searching actions.

The most important difference between the previously mentioned PSI and CHI is the use of a broad-spectrum language, V, that is more fit for people and with which specifications as well as program can be expressed.

As the CHI system itself has been described in V, CHI can be used to adjust CHI. Indeed CHI has been used to develop the rule-compiler of the system again, which resulted into a program that is only one tenth of the original.

4.4.2.3 CIP
CIP (München) is the most formal approach of the transformational programming. The project consists of two parts: the design of a programming language and the development of a program-transformation system. The project concentrates on correctness-preserving, source-to-source program transformations, varying from non-algorithmic specifications, applicative (functional) formulations, to imperative and machine oriented styles.

The algorithmic language CIP-L is a broad-spectrum language that puts all these programming styles mentioned into one syntactic and semantic frame. CIP-L is a schema-language. A schema is a representation of a class of related objects. The schema is created from an object by parameterising it, and conversely an object is created from a schema by instantiating the parameters.

A transformation in its most general form is a relation between two schemas P and P’. The transformation is correct (valid) if a particular desired semantic relation exists between P and P’. The most important relation here is (functional) equivalence and in order to allow the composition of the transformations, the relations should be transitive (R(x,y) & R(y,z) => R(x,z)), reflexive (R(x,x)) and monotonous. Transformation rules are partial representations from one schema to another. As the representation is allowed to be partial, enabling conditions are used to limit the domain of the rules, so that the domain-element and his picture together form a valid transformation.

A small core of CIP-L has been defined by a mathematical semantic. All the other language constructions are considered to be extensions of this by applying definitional transformations. This enables a modular description of the language and makes it easy to expand the language.

The CIP system (1983) does not contain a component for the automatic selection and application of transformation rules and all of this is the user's task.

Advantages
- a simple software development process consisting of a series of refinements which all of them are executed within the same linguistic frame;
- guarantee that the system answers the specifications because of the correctness preserving transformations;
- formal specifications enable a strongly automated development process;
- software has been verified implicitly and is correct;
- the specifications can be reused.

Disadvantages
- it is not clear how to obtain the first formal specification;
- the selection and application of rules for a great part remains human work;
- just a little experience with mainly small systems with mostly a mathematical-algorithmic character.

4.4.3 The Operational Approach

This approach can be considered to be a variant of the (phased) conventional approach, the variations of which have been inspired by the formal-transformational approach. The core of

---

23 For instance the relation 'implements': P' implements P is true if P' is an implementation of P, no matter how P' has been created from P.

24 That program domains are allowed to increase, but not to diminish and certainly not to have gaps.
this approach is formed by a distinction between problem oriented and implementation oriented things. The operational approach can be divided into three main phases: specification, transformation and realisation.

In the first phase the specifications of the external behaviour is expressed in an implementation-independent, executable formalism. By means of the prototype thus begotten (=conceptual model) the specifications can be validated against the wishes of the client. The purpose is to obtain a problem oriented specification.

In the second phase the specification is subjected to transformations that retain the external (functional) behaviour, but that modify or complement the internal structure that realises this external behaviour, in order to obtain an implementation oriented specification. Here interactions take place with the specification phase and the coupling between what has to be done and how should it be done is acknowledged. A simple example is that nearly every system restricts the size of input fields for processing transactions and that the external behaviour implies how the system reacts when these internal limits are reached.

In a third phase, realisation, the implementation oriented specification is transformed into, and represented in, the implementation language and an efficient program is obtained.

Advantages
- formal specifications and the use of correctness preserving transformations verify the system implicitly;
- validation of specifications takes place by the execution of specifications (prototype);
- formalisms enable a strongly automated development process;
- the specifications can be reused.

Disadvantages
- it is not clear how to obtain the first formal specification;
- the selection and application of transformation rules for a great part remains human work;
- there is a the danger of rash design decisions.

4.5 Checklist Software Engineering

In this paragraph the checklist from chapter 1 is matched against the general picture of a methodology for Software Engineering. The objective is to give more or less an overview of the present status of the discipline.

A: Methodological Requirements
A1: The definitions are precise and unambiguous; no confusion is possible. This is not the case. The terminology and the meaning of the symbols of the various methods and techniques sometimes differs considerably (for instance the use of diagrams in SADT and in DFDs: they seem to be similar but in SADT an arrow may represent data as well as an activity, whereas this is not the case in DFDs [Vlie84]).
A2: The task decomposition makes the complexity controllable. This is right. The complexity is not controllable after all, but that is caused by other things.
A3: Every phase delivers a well defined and well adjusted product. This is not the case. The instantiation with different methods and techniques for every phase delivers products that use a different methodology or that model an conflicting part of the world (for instance data-oriented analysis and process oriented design).
A4: The tasks can be performed in a cyclical manner, individually or as a whole. They can individually. Not as a whole. The strict maintenance of the life-cycle phases
(officially) does not allow feedback. Applying program transformation systems do allow a reasonable feedback because of the lower costs of the cycles.

A5: Methods and techniques: Numerous methods and techniques are available that do not fit in well with each other. Techniques for the elicitation are not available.

A6: Practical tools: Excellently developed. There are workbenches, programming environments, database systems etcetera. If there is a method, tools can be built for it. Because of the high degree of automation the method then seems to be sufficient.

B: Requirements regarding the Subject Organisation

B1: Effective: The methodological problems sometimes cause the methodologies to produce systems of the like that had not been intended.

B2: Efficient: After a laborious and generally inefficient analysis process the implementation can be generated in a reasonably efficient manner. This will be even more efficient with program transformation systems.

B3: Transferable results: As analysis methods are not clear in what they model, such a model cannot really be called transferable. The complementation of such a model with natural language usually is ambiguous too. Furthermore the similarities between analysis and design often are not clear, just like implementation decisions. The code is (especially after some time of maintenance) obscure. Attempts are made to improve the transferability of all products by complementing natural language.

B4: The methodology can be learned: Technically speaking that is true. With respect to content the question arises time and again where and in which phase to locate the methods, and how to use them.

B5: Encouraging communication: It is in the sense that many documents with many obscurities arise many questions, which enforces communication.

B6: The methodology must be complete: The methodology has methods and techniques (which are not adjusted to one another) for all aspects, from organisational aspects to maintenance, use and control. There are gaps on the level of the construction of the actual model.

C: Requirements regarding the Object Organisation

C1: Support of strategic planning: Yes, for instance Business Systems Planning.

C2: Support of the organisational transformation process: Yes.

C3: Maintenance: The maintenance problem is very big as usually finally the code is the subject of maintenance instead of the conceptual model.

C4: User participation: Yes, meant to put the users at ease and to involve him actively in the analysis process (to prevent misconceptions with analyst and to have the user think actively about his tasks).

D: Planning and Control


D2: Pre-calculation and post-calculation: They are present, but some have deviations up to 600%. Function point analysis is considered to be the most adequate method.

D3: Measuring points: Yes, the boundaries of the phases, where always a feasibility estimate has been stationed. Criteria for the judgement are not clear.

Résumé

Software Engineering has been insufficiently developed in respect to the methodological aspects. The consequence is that numerous problems arise on other fields, like the requirements concerning the subject organisation (the software house).
5 SUMMARY AND CONCLUSIONS

5.1 Summary

5.1.1 Chapter 1: Methodology

This chapter attempts to build a framework of a generic methodology for an empirical science. Viewing methodology for empirical sciences then identifies cycles on several levels, the most important of which are the meta-cycle of a methodology (that a methodology itself is subject to change) and the inherent cyclical aspect of all the partial tasks. From this the conclusion is drawn that the interaction between tasks is essential and cannot be suppressed, like many life-cycle methodologies try to do. The structure of this empirical standard cycle is furthermore identified as a generic model for empirical tasks.

The most important elements of a (normative) methodology for an empirical science are furthermore a language, or framework of concepts and definitions, to be able to communicate about the model, the empirical cycle as a guideline in the task decomposition, and methods and techniques to execute the tasks with. In a checklist a number of requirements are mentioned that help to integrate the methodology into the practical and commercial pursue of the science.

As a methodology in the end is just an empirical theory about the pursue of this science, that has to be validated, recipe-like working practices are discouraged in principle and the advice is to configure the methodology again for each project.

5.1.2 Chapter 2: Engineering

Chapter 2 attempts to put terminology and definitions used in engineering into a framework.

Engineering fulfils the function of a bridge between science and society, respectively expressed in the internal branch (application of the science) and the external branch (social function). A second classification is the one in Requirements Engineering and Design Engineering and these are concerned with respectively the modelling task and the construction task. The coherence between these tasks is represented in figure 3 that is repeated here (16).

The implementation of the analysis contains three tasks: collection, classification and synthesis. For the design tasks refer to 11 (KADS design process) which presents three tasks: functional decomposition, transformation and composition. The structure clearly shows the "middle-out" approach: the top-down functional decomposition is transformed into a system that is composed of the bottom-up constructed design elements and methods.

Controlling the project is considered to be a separate task throughout all phases, called Project Engineering. Finally the life-cycle model is identified as a "managerial backbone" for the project-wise execution of the work that models exactly one empirical standard cycle. The boundaries of the phases serve as measuring points for making decision (go/ no go), a reason why hardly any interaction between the phases can be allowed. A more detailed consideration however points out that implicit cycles can be distinguished, where the succeeding phase a detailing is of the previous phase. Instead of a fixed number of phases the management could also speak of the number of planned refinement cycles.
5.1.3 Chapter 3: Knowledge Engineering

Knowledge Engineering can be defined as the art or science to build knowledge based systems. A knowledge based system is seen as a system that one can request to generate and execute a transformation, that starting from the present situation (input) reaches the desired situation (output). This as opposed to a traditional system, where all input-output relations that a system can process, have been coded in advance and are limited to the relations that are quantifiable25. With this, the level of dynamic transformation-generation is reached. The task then that remains for the knowledge engineer is to elicitate the model that is in the mind of the expert in an undistorted manner, and present it in a model. For this elicitation several interview techniques are recommended, depending on the type of information that one wants to elicitate and in which phase one is working.

Reasons for the need of knowledge based systems include the conception problem of analysts (that they have to become experts before they can construct a model of the quantifiable part of the domain) and the vagueness that end users have about their own task model (that an advanced calculator does not make any sense if you do not know how, when and why to use it). Both problems are circumvented in knowledge based systems because the expert is responsible for the correctness of the model, and (the implementation of) his model can help the users to gain more insight in their tasks.

Using KADS as an example, the world of knowledge engineering is researched in more detail. KADS is model driven and the abstraction levels of the various models to be produced are indicated explicitly. What the "right" level of abstraction is for the conceptual model discussion is still going on.

Knowledge in KADS is divided into four layers (domain layer, inference layer, task layer and strategic layer) and every layer identifies different structures in the domain. The strategic layer represents meta-knowledge about the domain and the remaining two layers form interfaces between these. The strategic layer reasons about the domain in terms of possible inferences and generates goals and puts tasks on the agenda to reach these goals with.

The end of the analysis process is the conceptual model on a high level of abstraction. The model describes the knowledge on the four layers and consists of specification of domain

25 Somebody once asked me: "What is the difference anyway - we use a pile of if-statements for that". In any case in knowledge systems there is no need to know how many if-statements are concerned.
concepts, an inference model, a task model and a set of strategies.

Together with the external requirements this model is the input for the design and construction of the system. This construction phase is fairly traditional and works "middle-out": top down a functional decomposition takes place and bottom up the selection and construction of methods and design elements takes place, linked by a transformation.

5.1.4 Chapter 4: Software Engineering

This chapter makes a distinction between the top-down application of the discipline (the construction of systems) and the bottom-up application (the development of implementation-formalisms and tools), not to be confused with the top-down or bottom-up construction of systems.

In the top-down application the world is modelled to reach a specification (RE) and the system is implemented (DE).

The conceptual model for traditional systems consists of a data model and a process- or task model so that tasks can be implemented dynamically on a stable representation of the world. For the acquisition of "a" model various methods and techniques are available, of which DFDs, ER (ERAE) and JSD have been examined closer. These three methods each are on a different level of abstraction and each models different aspects of the world. Together it seems they may be used to make a complete conceptual model, however, the methods are not (yet) an integrated whole and no method has been found to get from DFDs (acquisition) to JSD (process model), for example. There is a method to get from DFDs to ER (data model) and to use ER as a data model in JSD. ER as well as JSD have the advantage that they are executable specifications so that the conceptual model can be validated fairly easily by the prototype. Another question concerning the analysis is, why there is not yet a model driven method.

During the realisation non-functional requirements (constraints) are included in the process and the designer takes implementation decisions based on specifications and constraints, in order to obtain an efficient and maintainable program.

During the bottom-up part of software engineering it concerns on one hand implementation formalisms where the machine being abstracted from to arrive at executable formalisms. An important observation here, is the wish that the number of interactions allowed by the formalisms is limited to the ones that are advisable or manageable (e.g. use of goto). On the other hand it concerns tools to support the design and implementation process, where the advanced tools usually are knowledge based.

The consideration of trends in the software engineering brings up as the major new branch transformation systems, where formal specifications are more or less automatically transformed into correct implementations. Thus an important part of the maintenance problem is solved; one just needs to modify the specifications and "replay" the automated development process in order to get a new version of the software.

The most important problem is the acquisition of a correct specification. Besides the question of what a formal specification looks like, in particular the question remains how to get a correct specification which contains all relevant aspects and how to verify that this is indeed the case. The often unstructured procedure of analysts does not guarantee this.
5.2 Conclusion

5.2.1 Conclusions regarding Knowledge Engineering

In Knowledge Engineering, at least in KADS, at present the emphasis is on obtaining knowledge in the correct manner and modelling this in a specification. The precise selection of the level of abstraction for the conceptual model intends to elicit the sufficient and required details to be able to deliver a specification, without losing the overview by getting bogged down in details. Whether this is the right level is still a subject of research. The level of generic tasks proposed in [Byl87], each provided with proper acquisition methods, certainly seems to be worth to integrate into KADS, though a lot of work then remains to be done. Perhaps in KADS the detailing of generic tasks and a refinement of interpretation models was still an outstanding task.

The analysis process of KADS has been developed excellently from a methodology point of view and in my opinion no further comment is needed.

The analysis model is still under development and practical experiences point out shortcomings and obscurities. The representation of the four layer model, where inference and task structures form the interface between domain knowledge and meta knowledge, reminds strongly of the data model and task model that has been identified in software engineering as comprising the conceptual model. Research in this direction may lead to an extensive representation of the domain, on which independent tasks can be placed and modified again. In this sense the product of KADS analysis can be compared to Data Flow Diagrams, that only models the objects that are directly involved in the tasks. I propose to call this vertical modelling, where only that what is needed and what suffices for the implementation of a particular task is modelled, as opposed to what I propose to call horizontal modelling, where the relations between as many elements from the domain as possible are modelled. The model driven approach seems an excellent approach for the inexperienced knowledge engineer (and who isn't), but it does not acknowledge the possibility to develop models of your own, based on ones own meta-knowledge ([Billau88], [SPIN88]). Furthermore an integration with the traditional computer science is absent: the logical system model, like used in Autopes, should get a clear place in the KADS analysis to avoid confusion. In my opinion the logical system model should model the conventional aspects of knowledge sources that produce data in an algorithmic manner (sensors, real-time systems, data base access, etcetera).

The KADS design and implementation process is performed in a traditional middle-out manner, as is usual in Software Engineering, but is hardly supported yet by standard formalisms (standard shells and high level languages). This will have a negative impact on the implementation process. Two extremes will arise: one knows only one shell and every system is forced into this shell, or for every system a new formalism has to be learned. Furthermore one depends on the supplier of the shell for high level tools or one has to construct these oneself if the shell has hooks for this. Research of AI implementation formalisms fortunately is in full progress.

Also problems are identified with the choice of a knowledge representation, which forms the interaction between RE and DE (caused by the impossibility to separate what from how, in particular where constraints with respect to speed must be met) and problems with making implementation decisions [Schrei87]. No experiences have been reported on the maintenance of systems that have been developed using KADS, but I suspect that a problem will arise here because, just as with first generation expert systems and traditional systems, maintenance will be applied to the code instead of to the conceptual model. This
will cause the conceptual model to be of less and less value, in turn making it extremely hard or even impossible later on to understand the way the system works. Note that a knowledge based system will be many times more complex than a traditional system.

Just as with software engineering, both problems (interaction and maintenance) can only be solved if the cost of the cycles can be reduced, either by using an automated implementation process (transformation systems), or by using executable specifications.

Highly knowledge based, automated tools are necessary for both acquisition and realisation. In order to make such tools generally useable, and not restricted to one shell or language, an intermediate language could be defined. The tools then supply a specification in this intermediate language and separate translation programs (back-ends) translate this intermediate language into the actual shell or language.

5.2.2 Conclusions regarding Software Engineering

The construction of larger systems often fails due to conception problems of analysts and the vagueness of the users about their tasks. Also the necessity to completely quantify models places high demands on the analyst and the analysis process. And the problem of how to obtain an initial specification has not been solved yet. Each of the analysis methods and techniques is on a different level of abstraction, models a different part of the world, and thus none are complete or deliver exchangeable results.

The methodological support of the analysis process thus is insufficient. A common understanding has never been reached on what a conceptual model looks like, what the necessary and sufficient elements should be, and next what successive techniques to use for it. There is work in progress to integrate the various techniques, but this is not guided by a methodological research that defines a framework or sets goals for such research. Because the analyst lacks an overview of what exactly should be constructed (what the conceptual model should look like) there is a fair chance that he will use methods in the wrong manner. Finally good techniques are lacking that would help the analyst to elicitate the right information from experts and users.

The cause of this situation may be the fact that the discipline of software engineering has evolved slowly since 1950 and that the analysis, initially very simple because of the limited capabilities of computers, did not catch up with the increasing technical possibilities (the bottom-up part of the work) of computers. This has caused the analysis framework to resemble the Tower of Babel in some respects. The framework lacks a methodological study, as has been done for the construction of KADS, that provides a framework within which research has to be done. KADS did have the advantage that one team had to construct a methodology from scratch, in an environment that technically speaking offers quite a high level of abstraction (shells and such).

The methodological support of the realisation process also seems to be insufficiently developed as no general methods have been discovered (supported by tools) that take the conceptual model as input. Only heuristics have been found.

What the input should be of the design process is not clear either, mainly because the analysis model is not clear. Assuming that it is possible to divide the conceptual model into

\[ n + m \]

Think of the number of compilers to be constructed for \( n \) languages and \( m \) machines that is reduced from \( nm \) to \( n + m \) by the use of an intermediate language.
a data model and a functional (process) model, the realisation phase is divided into two independent parts, namely in the implementation of the data structure and in the independent implementation of several tasks that operate on the data.

From a bottom-up point of view the realisation phase has been excellently developed, perhaps even so well, that top-down methods for the realisation at present are not even much needed. Examples are executable specifications (high level interpreters) and program transformation systems that contain the heuristics. Research of strategies and transformation rules is well underway.

One could say that in software engineering the limit of our comprehension has been reached, and that it is necessary, to be able to continue, to go up the next sport on the abstraction ladder. Searching for this has provided us already with new supportive tools such as transformation systems that relieve us of "trivialities" but this has not helped us to obtain the correct input for these systems.

Adopting some of the ideas from the KADS analysis phase will result in a significant increase of abstraction for the analyst. Furthermore tasks of the KADS analysis phase primarily want to give him a better insight in the problem domain of his client and the use of several interview techniques can help him to elicitate the relevant information more easily. The use of generic models will help the analyst to interpret the information and enable him to model it on a high level of abstraction.

5.3 Integration

This paragraph is an attempt to integrate both disciplines into one common methodology, where the difference between "traditional system" and "knowledge based system" disappears almost completely.

The idea that this is possible, is inspired by the conclusion that both disciplines actually want to model the same thing and then implement it in a system. Software Engineering specialised in controlling of, and abstracting from, the machine; Knowledge Engineering concentrating on searching for the best manner to construct the world model and to represent it heuristics. A happy marriage?

This integration is quite idealistic and does not take all kinds of practical problems into account. Let's hope however that this example does call for ideas for further development of an integrated approach.

5.3.1 Process Model

When studying KADS and software engineering, the picture in 17 comes to my mind, which represents a classification of the internal branch. The vertical axis here represents the level of abstraction (the horizontal axis has no real meaning).

The figure does not make a distinction between knowledge based systems and traditional systems: a knowledge based system only implements a task, which has been chosen not to

---

27 Only the internal branch is discussed.
28 The term 'grouping' indicates the logical grouping of domain concepts and relations in subject data bases (Martin8a) or knowledge data bases, data sets and frames, etcetera.
be implemented algorithmically. Reasons for that may be:
- the task is not quantifiable and only empirical associations can be discovered;
- the task is hard to quantify and it is easier (and thus cheaper) to generate a heuristic model of the task, instead of quantifying the task completely.

Implementation as a knowledge based component or as a traditional component thus becomes a decision on the depth with which the analysis process must be performed. In the same way one could, in my opinion, continue and continue analysing a conceptual (task) model produced by KADS until the algorithm is understood.

The eventual system is depicted on a fairly high level of abstraction (the figure gives an arbitrary level of abstraction), motivated by the observation that from this level on everything is "automatic" (interpreters, compilers, assemblers, linkers, chips, micro-code).

![Figure 17: The Software Development Process](image)

5.3.2 Analysis

In this model the analysis should yield a (horizontal) conceptual data model of the environment, and a separate collection of task models. Which differences exist between a knowledge model and a traditional model is not clear. A few similarities can be discovered however:
- A task like "make invoice" implies (sequentially) reaching simple goals for which a fixed task structure suffices. For instance: obtain(customer name), obtain(balance), calculate(VAT), format(invoice), output(invoice). Also it seems unnecessary in case of simple tasks to start out with a generic model.
- The inference structure that models desirable conclusions from domain concepts (= entities), has a resemblance with ER(AE) that models desirable relations between entities (=domain concepts). A relation obviously is a conclusion (=inference) from the combination of attributes of two or more entities. An ER diagram thus presents a kind of inference structure.
- A scheduler is an example of the strategic level in a traditional system.
- An interpretation model (generic task model) is the top level of a hierarchy of DFDs [Billau88]. Conversely, if a traditional DFD is abstracted from, then the structure of a generic task may have been reached. The above mentioned invoice-example then abstracts to a (stripped) synthesis task and for instance dealing with faults abstracts to an analysis task (in such a case traditionally a set of if-statements is used).

It is not clear in which case one is concerned with the analysis of knowledge and in which case with a traditional analysis. Besides detailing (going down into the DFDs), in knowledge analysis an abstraction takes place ("going up" into the DFDs and the structuring of domain concepts) that cannot be found in the traditional analysis. Adding this abstraction to the traditional analysis seems to me highly advisable, and makes the difference between knowledge analysis and traditional analysis diminish. Perhaps the difference between them is contained in whether the analyst decides that the underlying domain is so complex that he would rather go up and in that case is occupied with knowledge analysis, than that he would go down and thus would be occupied with a traditional analysis. It is likely that both will take place, namely a detailing of the domain down to a level of primitives, and an abstraction of primitives found, up to domain concepts or structures. On which level these primitives exactly are is not clear - I suspect abstracting as well as detailing takes place cyclically.

Attaching methods and techniques to these analysis tasks may for instance be as follows:

**Elicitation:**
- **explicit orientation:** before starting, the analyst gains knowledge about the (part of the) domain and investigates which data can be found in existing documentation and literature.
- **Interview techniques:** after the analyst has decided which data he wants to acquire, he selects a technique with which to acquire them.
- **Transcripts:** all interviews are recorded on tape and the most relevant parts are typed out (entered). While listening and reading, the analyst discovers things that escaped his attention during the conversation.
- **Lexicon and glossary:** the analyst makes up a list with terms from the domain; next he explains terms that need explanation.

**Classification:**
- **DFDs and data dictionary:** these make an inventory of processes and their interaction. The workbench also has capabilities to combine generic models (among others interpretation models) and identified processes (the process then is linked with a meta-layer). The model then helps the analyst with finding the fundamental procedure of the process (the task) and mutual refinement is possible. Besides registering and characterising the data that have been identified, the data dictionary has the capability to classify data into meta-classes. (Better perhaps is to say that one desires to observe and analyse a process from a meta-level and that the workbench has the capability to connect a meta-model to a process model.)
- **Protocol editor:** (see KPT [Billau88] and Hypertext [CACM88]) between interview fragments associations are made and references are registered. These serve to gain more insight and to register the justification of this insight.
- **(graphic) Concept-editors and library:** (see KPT) domain concepts from the lexicon and glossary are classified into relations and structures. Besides data classification (a data flow is a 'consists_of'-structure) other structures can be indicated as well (caused_by, is_a-hierarchies, etc.). The editor has zoom- and composition capacities. A library contains structure schemas that indicate for each structure which aspects should be registered and how to present the structure on the screen. The analyst selects a
structure from the library and ideally should also be able to compose and add new types of structures.

**Synthesis:**

Data model:
- **Entity-Relations (or similar):** out of the primitive data from lexicon, glossary and data dictionary, ER(AE) diagram are made up to obtain a complete conceptual data model. This model is a representation of the relevant part of the world and serves as a solid foundation underneath the tasks.
- **Concept-editor:** here one concentrates on the meta-levels that are applied to the DFDs and integrates the structures that are discovered in domain concepts. For several tasks for instance identical structures have been discovered. Data in meta classes can be simple (component) or complex (system model) and how to acquire the data does not have to be specified anymore. A conceptual model of the knowledge domains is made (several domains, because at various points in DFDs links may have been made with generic models.)

Function model:
- **Task/process structures from DFDs and knowledge sources:** from data flow diagrams task- and inference structures are created. This too is a continuation of the classification. Here, however, one decides what will be a primitive task or process, or whether a further (algorithmic) decomposition and/or a generalisation towards a generic model is needed. This process is again supported by a library of models on different levels of abstraction (from interpretation-models, to standard task-models like “debtor administration”).
- **JSD as specification formalism:** note that for instance the task structure of ”systematic diagnosis” (9) can be written down directly in JSD. If a task can be specified algorithmically (decomposition into JSD-primitive operations), then that is obviously the way to go and thus an executable task- specification is acquired. For a task like `decompose(system model)` this is usually not possible. In that case the task can be called a "knowledge source" and the associated inference methods and required domain knowledge are specified.

5.3.3 Realisation

As an effect of the two-part classification into data and tasks it is possible to generate simultaneously and independently implementations of the data model and of every single task. The transformations into implementation modules take place using among others the rules of thumb that have been mentioned in paragraph 4.2.2, classified in a knowledge based system that helps to select the transformations, executes them, documents them and analyses their efficiency, in the manner of the program transformation systems from paragraph 4.2.2.

Figure 17 also illustrates the middle-out approach in the realisation. Bottom-up one uses implementation formalisms (among others DBMSs), algorithms and program schemas (design elements) to construct functional methods; top-down one groups functional methods into control-structures as implementations of tasks.

Can we finally distinguish between a knowledge component and a traditional component? Considering the available formalisms we should definitely do that. Still to me it seems possible to extend a database to an object-base (knowledge base) adding AI elements like classes (is_a), objects (consists_of) and their inheritance relations, etcetera. Also the database can be equipped with rule-retrieval mechanisms so that with large knowledge bases each time there are only parts (modules or frames) in the working memory.
5.4 Postscript

Ever more it seems that the choice "knowledge component or traditional component" can be postponed and eventually turns into an implementation decision. An expert system shell then is an implementation formalism that has been abstracted a long way from the machine, but equally well could we represent the algorithms of inference engines and associated representation methods as program schemes, store these in a program transformation system, and in functional specifications only indicate which representation we want for a particular kind of knowledge. Next a system is built containing a "tailor-made" shell: a knowledge based system now is a very complex program with many dynamic characteristics, but has remained a finite state machine.

The welcome input of Knowledge Engineering is the long searched for manner to acquire the right specifications.
REFERENCES

[Anje87] The KADS System (Anjewierden A.) and The KADS methodology (Breuker & Wielinga), Papers for the Knowledge Acquisition Workshop, Reading, 1987


[Billau88] Billault J.P., Knowledge acquisition for a configuration task, UvA Sociaal Wetenschappelijke Informatica, maart 1988


REFERENCES


[Bulcke84] Vandenbulcke J., Systeemmethodeieken, in: [NGI84]


[Cars87] Carswell Jr., J.L., Navathe S.B., SA-ER: A methodology that links Structured Analysis and Entity Relationship modelling for database design, in [Spacca87]

[Chen76] Chen P.P.S., The Entity-Relationship Model - Toward a unified view of data, ACM Transactions on Database systems, Vol. 1, No. 1, March 1976


REFERENCES


[Heertje79] Heertje, A, *De kern van de economie* (the kernel of economics), Stenfert Kroese 1979


[Lind87] Lindgreen P., *Entities from a systems point of view*, in: [Spacca87]


REFERENCES

1988

[Ogilvie88] Ogilvie, R.P., *Software is iha. een oorspronkelijk werk* (software is in general an original work (to which copyright applies)), Computerrecht nr. 2, 1988


[Rees84] Rees, J.R. van, *De methode doet het niet* (the method doesn't work), in: [NGI84]


[Strat87] Straten, R. van, *Funktiepunt Analyse: theorie, praktijk en resultaten* (function points analysis: theory, practice and results), Informatie extra editie 'Begroten', 1987


[Wage88] Wagenaar G., Schrijnen L.M., *AUTOPES, de ontwikkeling van een kennisssysteem ten behoeve van procesbesturing* (the development of a knowledge based system for process control), in: [NAIC88]
