Bob Walraet

A Discipline of Software Engineering

North-Holland

A Discipline of Software Engineering

Bob WALRAET

Ethica / Coopers & Lybrand
Brussels, Belgium

1991
NORTH-HOLLAND
AMSTERDAM · LONDON · NEW YORK · TOKYO

ELSEVIER SCIENCE PUBLISHERS B.V.
Sara Burgerhartstraat 25
P.O. Box 211, 1000 AE Amsterdam, The Netherlands

Distributors for the United States and Canada:

ELSEVIER SCIENCE PUBLISHING COMPANY INC.
655 Avenue of the Americas
New York, N.Y. 10010, U.S.A.

1991 ACM classification: D2 Software Engineering (K.6.3), H1 Models and Principles, J1 Administrative Data Processing, K6 Management of Computing and Information Systems.

Library of Congress Cataloging-in-Publication Data

Walraet, Bob.
A discipline of software engineering / Bob Walraet.
p. cm.
ISBN 0-444-89131-5
1. Software engineering. I. Title.
QA76.758.W37 1991
005.1--dc20 91-3990
CIP

ISBN: 0 444 89131 5

Printed in The Netherlands

TABLE OF CONTENTS

RELATIONSHIP BETWEEN ENTITY1 AND ENTITY 2 : ORGANIZATION

CHAPTER 5: Semantic Action Model

ENTITY 2: ACTIONS

CHAPTER 6: Function & Task Design

APPENDICES

CHAPTER 8: Normalization is With Us To Stay

CHAPTER 9: About the Bill Of Materials

CHAPTER 10: About Structured Programming

CHAPTER 11: Object-oriented Programming

TABLE OF INSERT BOXES

This page intentionally left blank

OVERTURE

- *If the System is a potato, then I don't know what I'm writing about.*
- *If the System is a load of bricks, then you should not read this book.*

This page intentionally left blank

CHAPTER 1
The State of the House

Curtains for programming

Many still claim that programming is an art. Today it is more accurate to say that programming is an engineering craft which is based upon a formal apparatus of great depth, that has every chance of being complete and correct. But theory does not do the one thing that everyone expects: explain *how* one programs. True enough, theory gives correctness proofs. But they are of overwhelming complexity in all practical cases, and cannot be used. Composing a program is an act of creation which can be compared to that of solving a problem of geometry. In some way the composer must see, almost feel, a potential solution to the problem on hand. Next, he must fill in the necessary material which establishes the truth of his solution. Some people can do this, some can't and never will be able to.

That programs have a structure is not at all surprising, but structure is not the target in itself. In fact it is the problem to be solved that contains the structure. Perception of the structure, which is a work of abstraction, is the creative act. And again, there are those whose mental powers allow them to see that structure, even when it is masked by problem specification of an intricate nature. For a long time, many people have believed that structured programming was going to help them. How wrong they were! Structured Programming is nothing more than style. And style alone does not allow the novelist to write a successful book: the story must be conceived before any point of style appears... It suffices to take a look at many programs to see what has happened: an effort to disguise bad concept (non-solutions in fact!) by flourishing texts, written in an extravagant style. Scratching only the merest bit reveals the emptiness of the program: many flag settings for later testing, merely avoiding non-structured writing, but not fundamentally solving the problem. Programs of today are just as error-prone as before. Just as difficult to maintain or expand.

An impressive number of authors have tried their best to improve things. They have invented *methods*. We must certainly give credit to the people who did so: their work was and remains extremely valuable. They, the visionaries, realized an important fact: a problem must be *analyzed* so as to make structure appear. The problem must be decomposed into more elementary structures that are somehow connected. And we should note that it is the *problem*, i.e. the real world, *not the program*, that must be so mastered. In almost all cases, the problem will appear to

have an overall structure, but it becomes confused because of the number of exceptions. These must be analyzed just as deeply since they have an important influence upon the whole body. The authors therefore proposed disciplines, steps as it were, to do the work. But the major problem remains: analyzing (decomposing) an amount of work must have a starting point somewhere. It is not clear how to begin the decomposition. In other words, in order to do it, one needs a global insight into the structure from the start. Which completes the circle. Alas, no method gives that initial spark.

Moreover, methods are largely subjective: the methods proposed by the authors are those that fit their own intellectual approach. That particular author is very good at his particular method because that is the way his brain "feels" structure. And this is a process that cannot be transferred to other people. It can definitely not be taught. In a way methods have failed.

Fortunately, the application world abounds in paradigm situations. Quite a number of problems are more than similar, and a global solution certainly can be adapted to many particular situations. The systematic approach, via a method, at least prevents DP people from re-inventing the wheel. That alone, of course, is an interesting aspect. But whenever a totally new problem occurs, the programmers are at a loss. They have nothing to rely on: the whole thing is untouched wilderness in which roads will have to be cut without any means of orientation. This also explains the failure of relying upon experience. Experience induces duplicate solutions. A new problem will be solved along the lines used for all or some of the problems previously solved by that programmer. As a result, experience only helps when one remains in the same domain. It is no help for novelties... And, education is just as bad. First: not everyone can be trained to become a programmer. It is a story of haves and have-nots. But next, whatever education has brought is either specific to a given domain, and thus confines the student, or it is very general but then it has no depth so that it remains void... Moreover, many educational systems lay the accent upon rather trivial matters, idiosyncrasies of the tutors, so quite often biasing results.

There is another point that has considerably confused the issue: the abundance of programming languages. A very common mistake is the confusion between fluency in a language and aptitude at programming. Surely, a problem that gets solved must eventually be expressed in a programming language. The argument should however be: choose the language at the end of the conception process and select the one most indicated. Unfortunately this is never really allowed. A language is imposed by company standards (and usually it is COBOL, of all things!). As a result, the idiosyncrasies of the language affect the design in the earliest stage, since it is definitely not true that languages are general purpose (with the possible exception of ADA). Educating programmers so that they become good Cobolists (or anything else) is a gratuitous exercise and is harmful rather than solving anything.

I believe that correct data specification is a key factor in success. Identification of problem data is certainly a large part of the analysis. Indeed, programs are data transformers, in other words they are transfer functions that can be described as the "ratio" between output data and input data. Therefore, data analysis is an essential aspect. Not only should one devote careful attention to the static structure of data (as enforced by data-base technology) but also the evolution of data must be taken into account. In that field also, errors are commonplace. Methods abound, but, again, they only help if the designer has the initial spark. Data structures are not there for the fun of it. They must express a reality. The same problem exists as for programming: normalization (i.e. structuring) of data is seen as the target. It should be the means instead.

In the methods jungle

Every DP professional uses the words method, methodology, technique and technology. These words have become very fashionable. But they serve to cover up one of the major crises of DP: a lack of solid tradition. After all, DP is still very young; there can not yet be traditions...

Let us be clear about the terminology: a *method* is a set of rules (a discipline as it were) which, when followed, will achieve a very precise (and predictable) result. The method may be facilitated by means of a tool that implements it[1]. *Methodology* is the study of methods. The usage of the word as a synonym of method is utterly wrong. A *technique* is the sum of all methods, theories and knowledge that concern and build a well-delimited view of reality. *Technology*, then, is the study of all the techniques covering a consistent and closed part of reality. By metonymy, the word technology is also used to stand for the set of techniques. Thus, 4th generation development is a technology. Structured programming is a technique; Nasi-Shneiderman diagramming is a method to achieve structured programming. As usual, definitions like these, which are somewhat philosophical and not without emotion, are debatable. But I think the real point is the following: **method = discipline.**

In no other domain of technology[2] do we have as many methods as in DP. There is no way to compare them. Most of them are named by acronyms: SDM, LSDM, ISS, ISAAC, NIAM,... a new vocabulary, used in a way similar to car names. Although there is almost no difference between two cars of a same category, you'll find that both have their totally convinced drivers. Emotion crept into the game. The users of one method explicitly despise those of other methods. Moreover, methods are made fashionable by containing the usage of "brilliant" techniques! See how it evolved: a method contains techniques! It should be the other way

1. Computer Aided Software Engineering (CASE) tools are in that category.

2. Notice the use of the word "technology", taken here to mean: that part of reality that can be (re)-constructed by pure human endeavour.

around... Methods have overshot their goal. Some of the best ones tend to confuse project management and development technology in one single (huge) set of commandments. If they could, they would tell you how to set up the furniture of your office! Most of the methods get clobbered by the incredible amount of documentation they produce. They become document-management systems, and get hopelessly lost in the volumes involved... I still have to meet an analyst who actually re-uses the documentation. I remember one particular DP project, of rather average size; its documentation covered 5600 pages (26 binders!). There were endless discussions about storing away this volume.

Methods have one advantage, though: everyone can use them. Just follow the rules. Don't try to understand them! Many of the methods have killed creativity and reduced the demands upon human intelligence. Rather sad, this. Admittedly, there are methods that work in another way. They tend to channel creativity rather than crush it. Such methods are much more acceptable, but unfortunately, they are somewhat ill-considered because of the degrees of freedom they contain. In fact, the management feels a method is good when it is heavily prescriptive. Descriptive methods, on the other hand, are seen as rather worthless. Still, it is my belief that descriptive methods are much better in guiding the developer to a quality result. Such methods tell you what to do, but don't impose how.

There is another point as well: whatever method is chosen (and everyone chooses a method), the issue is further obscured by standards. All entities used in a DP development trajectory, to start with fields, have names. As an example, a very "obvious" way to give a field a name is: ENM-0415C3-BW035. In more human terms, the field means *employee-name*. Don't look for any cryptographic explanation, however. It is easier than that: ENM means Employee NaMe; 0415 is the identification of the record of which this is a field (this record would be called R0415, of course!); C3 indicates a computational-3 usage of the field (Cobol stuff); BW are the initials of the developer and 035 is the project for which this field was conceived. Obvious, no? Someone has clearly overlooked the fact that we have data dictionaries and repositories in which fields can be registered with a convivial name and attributes that further connect the field into its environment. A good dictionary will deliver any cross-reference required, just at the stroke of a key. Other standards: how is the address of a person recorded? 20 positions or 30 positions? House number before or after street name?

The common pitfall of standards is that they are too drastic, too involved. There is no incentive to apply them other than reprimand. One wonders: what is the true need for a standard? From the answer follows what the standard should then be, provided all other ways to achieve the same goal have been explored, and found unsatisfactory. It is my belief that the best effect standards can achieve is to avoid quality degeneration by imposing one choice at places where the development systems offer a multiple choice possibility. Something like:

- all relational tables should have a primary key (even if the system does not require this);

- there should be no sorting in on-line programs;
- for reasons of performance a 4th generation application module should not have more than approximately 400 code statements (notice the word *approximately*: a standard must be *flexible*)...

To cut a long and painful story short, I could cite a classic aphorism: *It doesn't matter which method you use, provided you use one*. There is a lot of truth in this sentence. Indeed, known methods intend to achieve a stated goal. In essence, the method is guaranteed to reach that goal. So, if the goal is not debatable, the method is OK as far as I am concerned. However, use sound judgement along the way. The interpretation of the discipline should be indicative rather than imperative. Deviate where it is indicated. Allow for creativity. Having no method, on the other hand, means that one is going to re-invent the wheel, most obviously by trial and error. A time-consuming and demotivating process. Adopt a method, but don't spend too much time in selecting one. I repeat: they are all acceptable (provided you throw away the heavy body of standards).

The worst one could do is see the method as a goal in itself. Unfortunately, this happens more often than not.

Not programming but software engineering

Creating a good program appears to be a rather delicate job. But making programs is not the only thing a DP professional undertakes. There is more, much more that has to be done at an earlier stage. Programming is the last part of the job.

In fact, what people create today are *systems*. In their technological implementation, systems are huge conglomerates of many programs that together deliver a solid and well-defined closed functionality. The making of such a system creates immense problems. The major one is in mastering the sheer size of the whole thing. Just imagine that an application of rather conventional nature takes something between 10,000 and 100,000 lines of (3rd generation) code! And that is small compared to some medical systems or NASA systems (tens of millions of lines of code!). Such volumes cannot be mastered by just one person any more. An accepted statistic states that one programmer produces 2000 lines of code in one year (from pre-study to accepted result). In order to solve this kind of problem, we need many people working together. And this is not a linear rule either: people have communication problems which increase quadratically with the number of people. A team of 10 people is considered a maximum. When a project is bigger than 10 people can manage to produce in a reasonable time interval, the project must be cut up in separate pieces and given to various teams. And this creates new communication problems. But even for a smaller subject, there is the problem of how one divides the project over the team members. This in itself is far from trivial, as we will see in the next section.

The situation further complicates because the business problem we try to automate is itself moderately to very complex, due to the great amount of detail one must master and the incredible number of exceptions that must be taken into account (many of them with related second-order effects). The result of such endeavours constitutes a package which does not have a linear behaviour: small changes in the code may induce disproportionate consequences in the outcome, so that one can never assume that safety margins are really obeyed. And if this weren't enough, systems once finished start to evolve, either because one must correct errors they still contain (corrective maintenance), but mostly because the reality evolves so that the systems must be continuously adapted lest they become obsolete much too soon (perfective maintenance). Also modifications for the sake of performance must be brought into the system (tuning). As a result, there is a continuous drop in quality because of the unsound haste with which such corrections and adaptations have to be realized. Implemented systems obsolesce very fast...

The discipline that groups all techniques and methods used to create a computerized system is called *software engineering*. Unfortunately, software engineering has no scientific grounds whatsoever. Apart from planning methods that allow a project leader to keep (some) track of what is currently happening, there is no method guaranteeing a quality result. At present, therefore, the only way is to achieve quality control at every moment of the implementation. This is done by continuous prototyping (evolutive prototyping) and keeping the knowledgeable end-user tightly involved. Nevertheless, managing a software development project is one of the most nerve-wrecking jobs ever.

The fact that most budget makers have unbelievably unrealistic requirements as to deadlines, does not help at all.

The life-cycle

The creation of a software system (even a system comprising only one program, however simple) is a work which proceeds in phases, not unlike the phases we find in, for instance, automobile industry when a new car type must be developed. In the automobile analogy we can certainly recognize three major phases:

- a phase of study, where the needs for the new type, its constraints, the wish list and some state of the art considerations are put together; this is the *analysis* phase;
- a second phase during which the new model is conceived, plans are made, prototypes and models are built and tried; this is the *design* phase;
- the last phase, when the first production car of the new type is assembled; this is the *implementation* phase.

Although the phases of software development are not exactly identical in detail, we find the three same general steps: analysis, design and implementation.

1) The analysis phase

A problem to be solved needs a clear problem statement. Usually however, this is missing or is formulated in the vaguest of ways. Therefore, it is required to push the investigation: what is really needed? How is the problem handled today (i.e. before automation)? Is there a problem solution (even if only in principle) without the need for a computer? Is this solution adequate on a computer also? Can it be improved by computer usage? If there is no "exact" manual solution, is there an approximate (or heuristic or good-sense) solution? Can this solution be computer-based?

In order to answer these questions, one has to go and capture the reality, in the world of every day[3]. In a business application this can be a straightforward activity, although it often involves a vast amount of work. Most of such applications are a link in an information flow network. Information coming from the outside world is input and results in the creation of transformed and usable information, which in turn serves as input for yet another process. Thus, investigation of the functional information flow in the company's departments is the key to this part of the work. This will require interviews, observation and quite a lot of communication psychology. The result is a set of specifications, usually (but not necessarily) expressed as a set of input values (possibly with an involved structure), corresponding to a set of output values. In this light, an automated package is a transfer function, creating a mapping between the input and the output set.

If the analysis work is to be complete, the model of information flow should be analyzed so as to find unproductive situations: processes that are never executed, data that serves no one, loops that do not really produce anything. Formal methods such as Petri networks (and its derivatives such as event diagrams) may be used here.

The mental process needed during analysis is one of *abstraction* or, put more pragmatically, generalization of the detail information gathered by investigation. Here it is of paramount importance to perceive the fundamental structure in the mass of detail (and exception) cases. The information needed by the processes (and external to them) must also be investigated, so as to detect its static structure. It is important at this stage to perceive the data structure as independent from the programs: it pre-exists by essence and should therefore be represented in a natural, application-independent way.

The major part of the analysis work relies upon a "carefully conducted" dialogue with the concerned (and hopefully strongly involved) end user. Such a dialogue is difficult, even under ideal conditions, because the end user is not very apt at expressing abstractions. A way to structure the dialogue is by breaking the functions down into manipulations of *user views*. This is certainly a good thing to do. But

3. This world, in which end users manipulate objects and understand objects is often referred to as the *world of discourse*.

being able to build a *prototype*, even at this early stage, would help even more. A prototype is a quick and easy (hopefully not too dirty) mixture of means that emulates the problem solution (preferably on-line) and shows the user what functionality can be expected. The prototype is a truly operational object, and will serve to refine both the user's and the analyst's perception of the problem. It is precisely by using fourth generation techniques in association with relational data-bases and a number of "tools" (such as query languages (e.g. SQL, report writers, screen mappers, spreadsheets, graphics,... not to forget data dictionaries) that prototypes can be realized, the cost of them becoming justifiable.

The analysis stage ends with the production of a set of specifications and requirements. It also produces some models: a process model representing the activities with their information-carrying user views such as take place in the application domain, an information model which represents the major entities carrying the information, their fields and their inter-relationships and finally an organization model representing the relationships between people and the company's activities. It should also deliver a choice of solution outlines.

In some connotations, the analysis phase is called *study phase*, *conceptual level*[4], *enterprise level*. A question of what's in a name.

2) The solution design and data structure design phase

Having established the problem specifications and requirements, one now determines a solution. In essence, the work proceeds from the models obtained during analysis, by chosing where choice exists, by filling in detail where detail is missing, by correcting erroneous situations where they exist. More specifically, the design phase will represent the solution in some normalized model, that is a model which obeys easy and strictly enforceable rules. This model will emphasize the functions which will be offered to the users of the application, the user interface (screens, reports,...) as it will manifest itself to the user. It will also indicate the new information flow. Where appropriate, standards will be formulated: for instance in order to achieve a consistent user interface throughout the system.

The design of the fine data structures that must hold the long term information constitutes half of the design work: programs become less complicated, more visible, when the data structures are correct.

The methods used for designing a software system are essentially the same as those that can be used for the analysis. It is a matter of shifted accent.

4. This notion of level is advocated by ISO. It is based upon the idea that there are three ways to look at a software system: a conceptual look (understanding what it is all about), an external look (understanding how to use it) and an internal look (understanding how it is built).
IBM's AD-cycle methodology speaks about an enterprise level (perception of a software system as it is articulated in the enterprise reality), a design level (how does it work), a technology level (how is it built).

3) Program Implementation phase

This phase contains two strongly connected activities. One is the creation of the algorithms[5] and of the physical representation of the data-base (if any). The other is the actual writing of the programs and the setting up of the data-base (if any). While on-line algorithms in business DP are usually quite simple, algorithm creation for batch processes calls on all the craftsmanship of the programmer. A great amount of knowledge is required, especially of typical solutions to a vast number of situations. However, if the problem to be solved has been decomposed deeply enough and the associate data structures and resources have been well described, most algorithms will usually be mere transducers and fairly easy to establish. The more difficult ones tend to be isolated; it is not every day that a programmer must solve Hanoi tower problems.

On the other hand, if the problem to be solved is in a specific area, such as text processing or syntax handling, there are sufficient well-documented solution profiles in literature. Writing the algorithms should be straightforward.

There is another important aspect: notwithstanding the claims of formalists, the algorithm's profile depends on the chosen programming language. This is a rather sensitive field: instead of allowing programmers to choose the most appropriate language for a given algorithm, there is usually a company-imposed language such as COBOL. There may be economic arguments for this (although I strongly doubt their validity) but it is a truly sad fact. No language is perfect, so that the mere choice of a language entails the need to use a number of tricks. The major responsibility of the programmer is to produce highly readable code which espouses the algorithm as closely as possible. This is a matter of producing good syntax: bear in mind all the syntactic structuring items the language offers and use them in the cleanest of ways. Don't misuse a structure to simulate something with! Good programming standards (but do they exist?) are certainly helpful. A rule of thumb is to isolate *black boxes* in procedures which are replaceable as a whole (or paragraphs in COBOL). This, incidentally, is the gist (in reverse) of stepwise refinement. And don't forget the programming of the accesses to information (pragmatically: data access). Acceptance of and commitment to the concept of data independence is an absolute must: a programmer should never go physical on data. Instead, use SQL or something similar (and better). And don't ever forget this basic truth: *structured programming is not a goal in itself; a program must be structured so as to reflect (in some way) the structure of the objects it works with (data, knowledge, events, information as it were); structures in the syntax (connectives and the likes) are only a way to divide and conquer, but of course, you must know what to divide and how to divide it!*

5. This sub-phase is sometimes called technical design; the phase which I called design is then called functional design, by contradistinction.

Why lifecycle?

The three phases that we have seen are always present. Their border lines may vary according to the authors or the culture. Some authors speak about the *waterfall metaphor*: one phase flowing into the other; there can also be a flow up stream: from design back to analysis, from implementation back to design, from testing back to implementation. But there is more to it. After implementation, there is the important phase of *testing*. Here a software package is verified for its precision and reliability. This phase is also called the *quality assurance* phase.

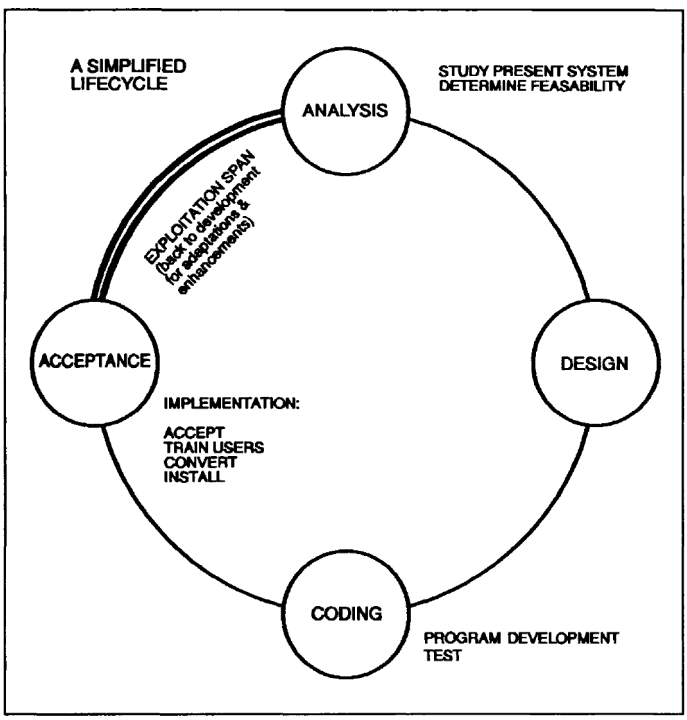


figure 1

Testing takes place at various levels of integration: unit testing regards a program or set of programs, stand-alone testing regards the software system taken by itself, integration testing is the test of the system in its real environment. There are various policies for testing and this constitutes a fascinating domain; it is however beyond the scope of this book.

After the testing the software package is formally accepted[6] and is installed in its production environment. This is when things start happening. Soon (much too soon) parts of the software system must be modified, either because

6. Although rejection is also possible, I have never seen it happen. At this late stage, companies prefer to accept a software system, even if not completely satisfactory, and will learn to live with it.

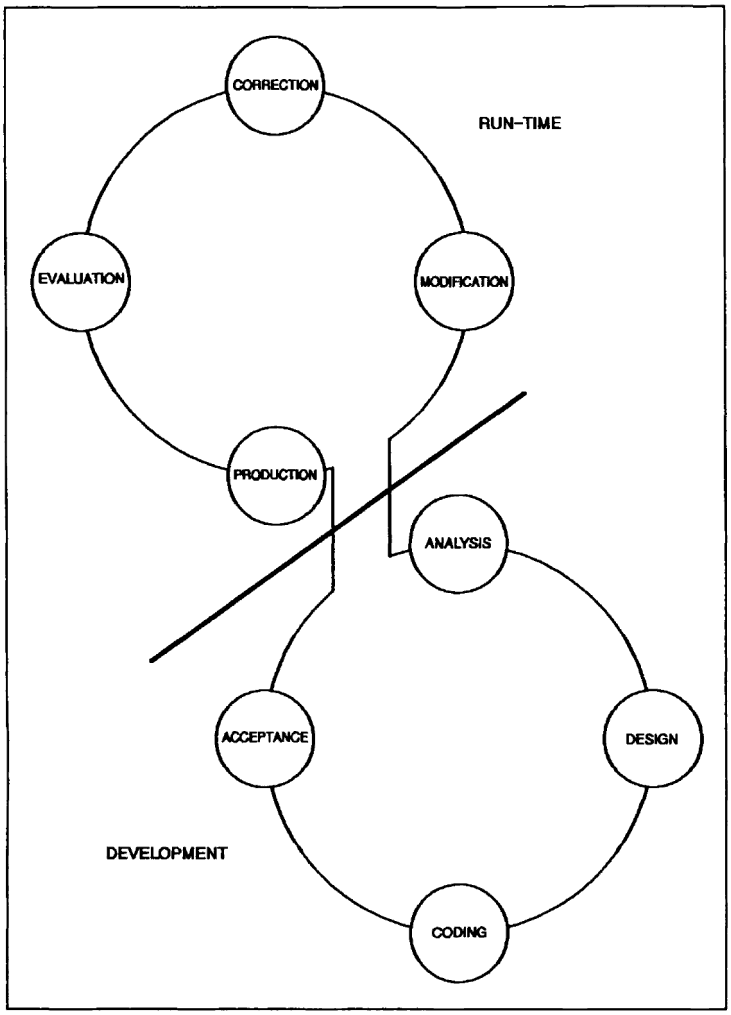


figure 2

errors must be corrected, or because company rules have changed or because new cases must be added. The part of the system that undergoes the change goes back to development, so that we start a new cycle of analysis, design, implementation and testing (see figure 1). This cyclic way of doing things is typical for software[7]: in most other domains one works with fixes or merely throws away the unsatisfactory product. Of course, there are situations in which software will be temporarily fixed (the temporary plaster becoming a definite wart). And it is also clear that the moment will come when the package will be thrown away. But in the meantime many "life" cycle rounds will have been fought through. Yet another (equivalent) way of illustrating the lifecycle notion is given in figure 2. It has the advantage of stressing the different environments: what happens during development and what happens during operations. In

7. It sounds obvious today, but it took DP a long time to come to the lifecycle idea. In the beginning, development was done according to a *code-and-fix* philosophy (not unlike the proverbial trial and error approach). Needless to say, programs were never correct. The idea of phases (analysis, design, code,...) came next. The *waterfall* metaphor emerged as the discipline for going through the phases: one phase feeds the immediatelyfollowing one. An extension was the retrofit possibility: one phase is capable of causing a return to the preceding phase. The cyclic idea of an application going back into blueprint stage came to the minds many years later and it was eventually tempered by the *double lifecycle* philosophy.

the latter environment, the cycle starts by taking the system into *production* and running it. At some moment (possibly because of problem reporting & assessment) an *evaluation* is held which results in the decision to undertake (and the definition of) small scale *corrections* and enhancements, which are then brought into the system as mere *modifications*. The cycle can then proceed into the production phase. After several such rounds, there will be more dramatic enhancements needed, which cause the system to return to the development environment.

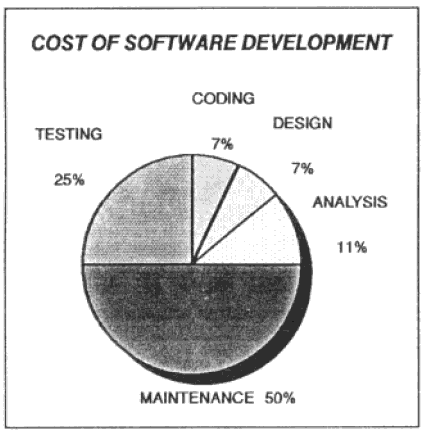


figure 3

According to various sources, the effort involved in the various phases is rather different; it is remarkable that coding amounts to less than 10% of the total effort, whereas analysis and design take the lion's part. But, and this is perhaps more spectacular, 50% of the total effort put in a software system is spent in maintaining the system (see figure 3).

The state of the union

Apart from the *what does it cost?* question, the major pre-occupation of system makers is *end quality*.

Literature indicates certain criteria that allow a qualitative definition of quality. They are:

- Precision: the degree of conformity of the finished system to its initial specifications, assuming, of course, that the specifications are precise themselves;
- Reliability: the degree to which the finished system behaves without errors when it is used according to its original specifications (assuming that the specifications describe such uses);
- Efficiency: the acceptability of the cost of using the finished system (in terms of computer, human and time resources);
- Security: the discipline by which the system avoids unauthorized access;
- Usability: the ease of learning to use the system; this includes the coherence of the user interface (the same microscopic functions of different subsystems have the same look and feel; this applies even across different systems);
- Maintainability: the ease (for the programmer) of bringing corrections to the system;
- Flexibility: the ease (for the programmer) of bringing enhancements to the finished system and the resilience to obsolescence;

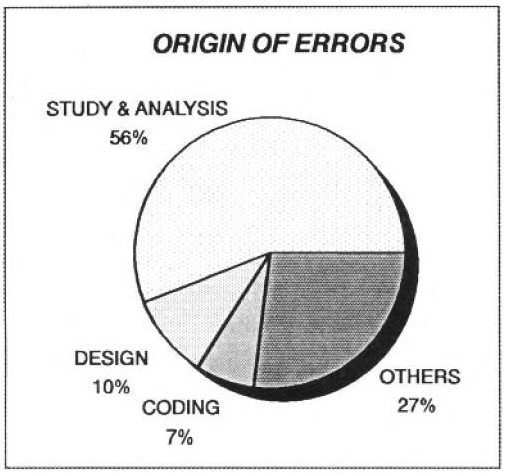


figure 4

– Integratability: the ease of using this system in conjunction with other systems, either over a common data resource (a data-base) or as a sub-system.

When we investigate systems of today against the above criteria, we cannot help but notice that most systems do not comply. The most common complaint is that although the development of the system has taken much more time (and money) than foreseen, it is neither precise nor reliable. The common saying is: *the system has taken twice as long to develop and is twice as bad as expected*! This results in a cascade of effects. Since it has taken longer to develop the system, the human pressure during development was high (deadline pressure) so that the precision and reliability came to suffer even more. Moreover, the time used to develop the system reduced the time for developing other systems so that backlogs of years come into existence (5 to 8 years is an accepted figure). On the other hand, the delivered systems being neither precise nor reliable, they need frequent corrections. Not only are the programs usually not easily maintainable, but maintenance must also be done at catastrophic moments (according to Murphy's law). As a result, programs obsolesce much faster than foreseen, so that the maintenance effort increases

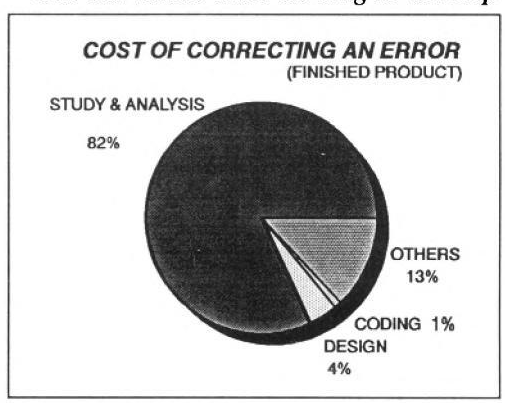


figure 5

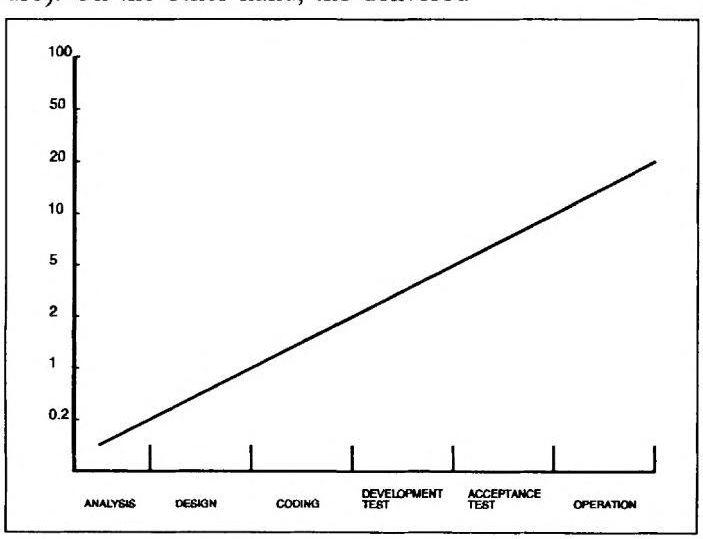


figure 6

over time, causing increased backlogs and faster obsolescence. On the other hand, obsolescence is not readily recognized, and people keep mending programs that are beyond salvation.

The general symptom is that the users are extremely dissatisfied, both because of the number of errors in the systems and their difficulty of use. The backlog is also a factor that users have to live with.

When, trying to do a more scientific survey of the state of matters, one notices that most of the errors in a finished system originate in the analysis phase (see figure 4). The correction of these errors during the productive life of a system amounts to over 80% of the corrective effort (figure 5). In general, the later the phase of the life-cycle in which an error (of any origin) is corrected, the more expensive the correction is (figure 6) and the more obsolescence it tends to create.

The conclusion is obvious: one needs to devote much more attention to the analysis phase. This is where methods should do their utmost to help the software engineer in achieving the required detail.

There are two major problems here that cause managers to do away with a deep analysis stage: the first one is that the analysis is a social exercise: one must understand a human organization for which the automated system has to be made and since this is an activity that takes place in a climate of conflicts of interests, it is avoided; the second one is the cost factor: before being able to set up an estimated cost of a software development, the analysis (and part of the design) must be conducted without any guarantee of return on investment. It is all very well to state the necessity of using analytical modelling; the question is: can one afford it? Both in time and money. The process is lengthy, calls for organization specialists (expensive people!) and produces unverifiable quantifications. Indeed, how could one verify the completeness and thoroughness of a pre-study? Who is to tell?

As can be expected, the analysis phase is more often than not largely sacrificed... But consider what the alternative cost is: according to NASA, 75% of the sizeable software projects fail in delivering what was promised...

Can quality be achieved?

An important aspect of software engineering is quality control. Does the program correctly represent the problem to be solved? Does the algorithm correctly map the reality? Does it meet the requirements? Have all possible exceptions been covered (at least with error exits)? Many formal authors have developed a body of *program calculus* allowing program correctness proofs to be conducted. However, they all rely on one premise: the problem specification is itself formally complete. In reality, this is, unfortunately, never the case. The world of discourse holds its own contradictions, vaguenesses and incompletenesses. Moreover, correctness proofs of a formal nature are extremely difficult. What we need therefore, is to establish *near correctness*, i.e. acceptability defined in economical terms: keep the *risk* of not being correct within accepted margins.

I believe that the most practical way to pronounce near correctness is by having walkthroughs done at the earliest stage, that of (functional) design. Let the author explain his work in detail before his fellows in the team and observers from the user's world. Questions will arise, errors will be detected, alternatives will be suggested. The ideas bouncing in such a session will contribute to correctness. Conversely, the much acclaimed technique of code reviewing does not seem to be very effective (I consider it an expensive exercise in disguised futility!). It is boring, to say the least, and the readers will not spot problems, apart from rather obvious ones. Moreover, since it is done a posteriori, developers will tend to become even more sloppy: they know the code reviewer will be held responsible if errors remain. Instead of code reading it is more effective to set up an extensive test environment, that maps the reality as closely as possible. It then suffices to separate cases of interest into specific test cases. Again, such techniques cannot be exhaustive, but they do serve their purpose and are worth the trouble, even though they are not without problems.

Of course, the whole quality control work is eased by an order of magnitude if a prototype was approved initially and when a solid design language and programming language was chosen and algorithms were well documented (i.e. state their input, state the output, state the long term memory, state the business rules, state the inter-connections with external objects, state default assumptions). A very good way is to use refinement: give a global functional description of which the phrases are refined into program sentences, thereby guaranteeing readability. It might be advisable to use data-driven practices (e.g. decision tables) rather than code-embedded ones, since such programs are definitely more stable, and the external data structures, being very descriptive, also act as the documentation of the process.

There are no miracles! Whatever the means, the realization of a project calls for time, effort and a lot of skill. Most projects degenerate or fail altogether. Once finished, most programs are already (partially) obsolete in their functionality. As a result, more effort has to be put into the lifelong maintenance of an application.

The cause of it all seems to be the incredible amount of misunderstanding throughout the project's life. This is due to communication problems between parties concerned: the user talks to an analyst who talks to a designer and to management; the designer informs the programmer who converses with the quality control specialist. All of these people supply implicit personal views and assumptions. It results in garbling. The only way out is (according to many) *early prototyping*. Show the requesting parties what they'll get. Remember the old saying that *a picture is worth a thousand words*. An early prototype is worth a thousand pictures! There are of course two ways to do the prototyping: either one sets up a *fast and easy* throw-away prototype, or -and this is infinitely more productive- one creates an evolutive prototype which gets gradually refined into the final product, by using fourth gener-

ation tools and practices, thereby ensuring that there is always a *showable* object. When the move towards 5th generation[8] programming is taken, the necessity of prototyping will become even more crucial. Indeed, there is not yet an accepted universal knowledge theory[9]. Thus, development of a knowledge base is done by means of stepwise prototype-based refinement. In this area, conviviality of the workbench is of the essence.

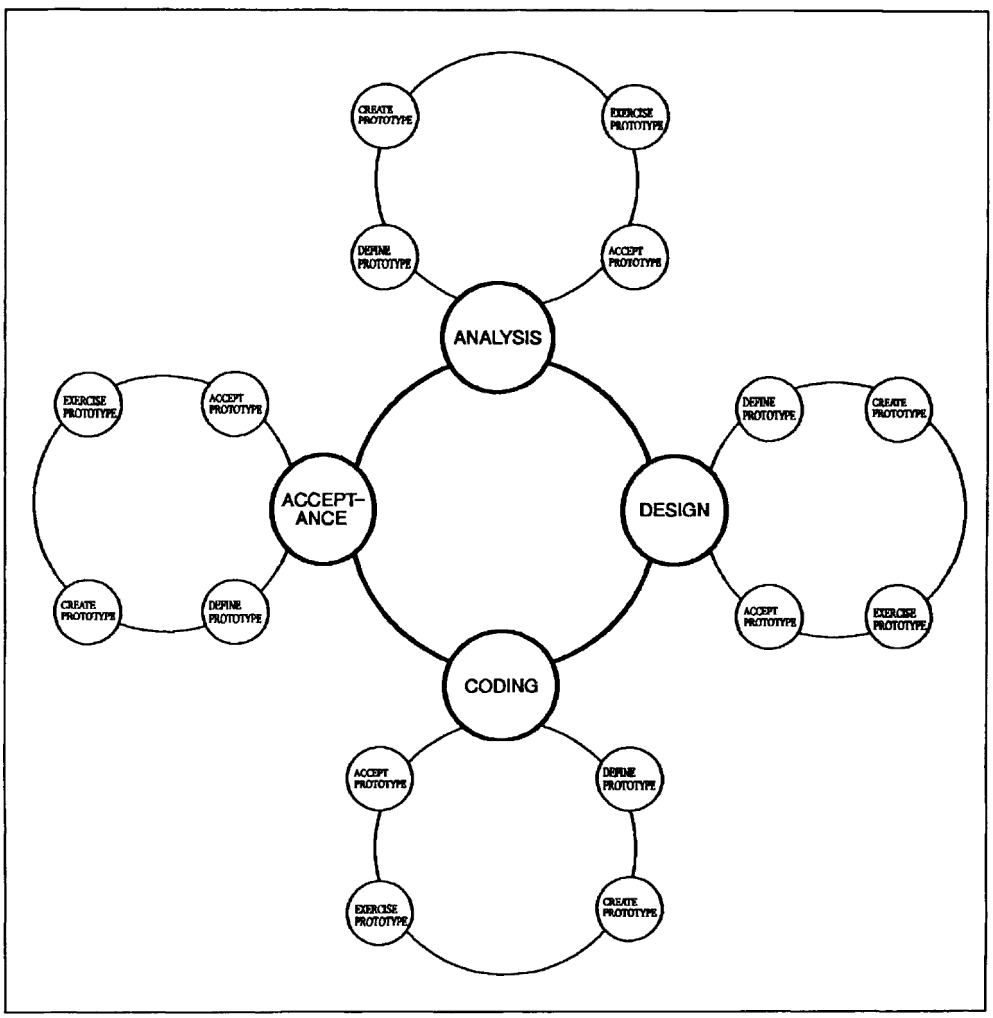


figure 7

8. By that, I am refering to the various techniques that allow designers to go into *Knowledge Engineering*.

9. Although the apparatus of formal logic is, obviously, as complete as it can be, there remains the fundamental aspect of heuristics to be suitably formalized.

There is a caveat, though. Prototyping techniques are not the long-expected miracle. Indeed, practice has shown that programs developed using prototyping are generally of a better functional quality but are less robust under maintenance because they are not so well structured; programs developed using the conventional techniques have a lower functional quality but have a better resilience under maintenance, because they contain solid program structures[10]. A compromise might be the development of a model: a prototype that one throws away at a certain stage, after having translated it into a solid program which gets further refined in the normal way. Whatever the approach may be, prototyping calls for more thinking at earlier stages about the implementation aspects. This may be good or bad; the fact is that the prototyping activities are in fact miniature lifecycles for a prototype nested in the lifecycle of the software system under development (see figure 7). The activities in the prototype lifecycle are the definition, the creation (building), the exercising of and the acceptance of the prototype. The latter is inherent as a termination event of the exercising of the prototype, and need not be represented in the prototype lifecycle.

More about prototyping is said in chapter 6.

Quality: a matter of responsibility

All things being said, quality remains largely un-quantifiable. Attempts at quantification have been made, usually by decomposing quality objectives and requirements into finer statements, so fine in fact that a metrics become possible. Still, there is no satisfying way to ensure that quality is obtained.

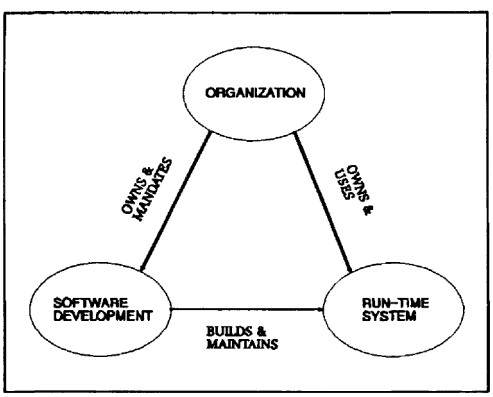


figure 8

It may be that the definition of quality cannot be fully stated. In fact, there is a complicating factor: the quality of a system component is assessed in informal ways by many people, so many that all statements become contradictory, partial and purposely vague. In fact, everyone is concerned but no one is responsible. The blame will always be placed on the next person. A good deal of quality problems will be solved almost spontaneously when each component has a visible responsible *owner*. This will avoid the classic charade: the software engineer blames the end user who didn't express his needs correctly, the end user blames the software engineer who didn't understand the needs; both

10. This statement expresses a positive prejudice. In reality, some of the conventional programs have such an appalling structure that any prototype would be better.

blame the management for not having taken the right decisions. What should not be forgotten is that the company's organization is the requesting party. They are the ones who need actions, processes and functions to be automated. Unsolicited development by the software department is impossible. The organization owns the software department. The software department builds the requested systems under the responsibility of the organization. The resulting system is owned by the organization. This ownership philosophy is illustrated in figure 8.

By placing the responsibilities in that way, even at this macroscopic level, a feeling of concern should prevail and this will be instrumental in improving the quality of the system. The notion of responsibility through ownership is refined in virtually every chapter of this book.

Design revisited or divide and conquer

Involved as the analysis of a problem to be solved may be, the design of a solution is even more complicated an activity. During the design phase, one essentially decomposes a problem into sub-problems, each one of them being decomposed again until manageable units are obtained which we can start implementing. Consider a trivial example: suppose our problem statement is "build a TV set". This problem decomposes into sub-problems which are: {get a wiring diagram}, {acquire the components}, {assemble the components according to the wiring diagram}. What should be noticed here is that the sub-problems are not independent. First of all, all three of them must be solved. Furthermore, they must be solved in order, since acquiring the components can only be done after we have set up the wiring diagram, and assembly can only be undertaken after both the diagram and the components exist. This is an *ordered conjunctive* decomposition. In many cases however, the order of the decomposition is not relevant. For instance, the problem "create a hi-fi installation" decomposes into {acquire a turn-table}, {acquire an amplifier}, {acquire speakers} and each of the sub-problems can be solved in any order. This is a *conjunctive* decomposition. Sometimes a decomposition is even more flexible. For instance, the (sub-)problem {get a wiring diagram} can be solved either by {buy a diagram} or by {create a diagram}. This is a *disjunctive* decomposition. Such decompositions are often bewildering in the sense that one wants to select only one solution, but which one? Of course, it may happen that our original problem requires the alternative to be kept and have the problem of choice solved each time the application actually runs (according to temporary conditions).

The creation of a decomposition is, in fact, also a problem of abstraction in the sense that each level in the decomposition is itself an abstraction of the levels underneath. The lowest level is a (set of) program statement(s), of course, but this is again an abstraction of the machine code underneath the program statements. Decomposition is a top-down approach going from the most abstract level down to the most concrete level (stopping at a body of program statements in a chosen language). The important thing is that a decomposition creates a hierarchical tree, a structure which is conceptually easy to master.

While we perform the decomposition, we will also have to investigate the data structures that our problem requires. Such data structures will also be seen starting with a very abstract view down to a very concrete view. In other words, there is also an important notion of data abstraction. For instance, the sub-problem {get a wiring diagram} uses the data structure *wiring diagram;* of this data structure it is not important, at this level, to know *how* it is implemented nor *how* it is manipulated. What we need is the knowledge of *what* it represents and what logical operations it should be able to undergo (possibly limited to *read, store, delete* a diagram). Data abstraction is essentially a matter of expressing the data in terms of classes and objects with methods. The actual creation of the physical implementation of the data structure and its methods should be seen as a sub-problem in itself, disconnected from the more procedural sub-problems which should never be burdened by the physical data aspects. In other words, we impose an independence between the program realm and the data realm. In this respect it is quite incredible to notice that, especially in the business DP world, people are rarely interested in this approach, although it is considered by most authors as the most valuable contribution made to software engineering in the last fifteen years.

All this being said, the fundamental problem we are now stuck with is : how do we decompose our problem into sub-problems, *so that it makes sense* and can lead to an implementation plan? This is not a trivial feat. Consider indeed an ordered conjunctive decomposition: the sub-problems depend on one another. Can they be designed and implemented in their own right or should they be done one after the other? This depends on the way in which the sub-problems actually connect together. For instance, if the sub-problem {acquire the components} is solved in a very different way depending on the solution of {get a wiring diagram}, we must solve these problems in order. On the other hand, if we can create a solution to {acquire the components} which is general enough (i.e. abstract enough) so that it works whatever wiring diagram was set up, we do not need to work in order. However, for a large system, the first level of decomposition should be of an ordered conjunctive type, so that the sub-problems of this level can be assigned to different team members without creating major human communication problems. For instance, in a human resource system one could create a first level comprising personnel registration, absence registration, payroll, skill management, etc. These problems can very obviously be considered stand-alone. Of course, there is one essential communication area anyway: that of the data structures, which all components will somehow use. It is a good idea to devote the design of them (including their methods and semantics) to yet another team member (or even to a data analyst outside of the team).

Further levels of decomposition should be "workable". What is meant by this is a subjective concept. One quality aspect is in obtaining a tree of sub-problems in levels rather than a network with crisscross lines. Consider as follows: at the lowest level of the decomposition we obtain implementable modules (something a language like ADA is very good at). Of course, each higher level represents a connection together of modules so that super-modules come into existence. The decompo-

sition is good if any module of a given level, say n, connects to *only* modules of the level immediately underneath, n+1 (and to modules of its own level as well). In fact, such an approach allows us to consider each level as an abstract machine of increasing functional power (as we look at a higher level of the diagram), replaceable at any moment by another implementation, provided the same functionality is offered. One of the first advocates of this philosophy was Dijkstra in his famous THE machine views. He also indicated that a module obeying the constraints indicated, had every chance of being proven correct, using formal proof methods. Such a module he called a *pearl*, since like in a necklace, every pearl "connects" to only its predecessor and successor. And a pearl is an object of high quality as well, containing no bugs. A beautiful image! Using this approach, not only is design more easy to conduct, but maintenance will also be much more controllable.

The links between modules should be kept as simple as possible. Simplicity is expressed in terms of data structures and values that are transferred over the links. In fact, a module should never need anything other than these structures, which should be frozen not only throughout the execution of a module, but for the whole life of a module (maintenance permitting). In other words, the quantity of information a module works with (*entropy*) is not a function of time. The advantages of having the rule (of thumb) that entropy should be constant, are obvious: programmers have less human communication problems regarding their modules while they construct them; linearity is improved: a change in one module has less chance of propagating its effects all over the place; understandability of the system by the people responsible for the maintenance is largely improved, since any module can be seen in the context of only one predecessor and one successor level. Also, the modules can be made more general (abstract) and serve more universal functions (universal within their own level), since they need less assumptions regarding their environment.

In fact, the connections between modules should be *weak*, by which is meant that the connection does not depend upon the internal complexity of a module. Another way to formalize this notion is by saying the connection should be standardized. Compare this to the problem of configurating a PC. The PC is composed of a keyboard, a CPU, a monitor and a printer. These are modules and they are rather complex. However, they inter-connect with only a couple of standard cables. The inter-connection is weak. On the other hand we may say that the internal coherence of each module is very *tight*. There appears to be a pragmatic law that says *the tighter the internal coherence of a module, the weaker the connections required and conversely*. This is intuitively true, since lack of tightness of a module refers to the amount of knowledge a module must still gain from its environment over connections whose entropy may vary in time and be largely unplanned.

It is a very positive fact that 4G techniques enforce the creation of modules with only weak inter-connections (a very limited set of standard connectors is available) so that tightness of the modules ensues almost automatically because the designer is forced into creating tight modules. On the other hand, many applications use an

underlying data-base as a background, but this data-base should be seen as a long-term memory, nothing else. In that respect, it is unforgivable that many developers consider the data-base as a connector as well, using various tricks to pass information from one module to another in a totally unstandardized way (flags, exceptions, etc.), thus inducing a serious loss in tightness of the modules.

The houses of Ret Up Moc

Software engineering, notwithstanding all the methods that tend to increase the quality of the job, is most often done in a very "magical" way. Misgivings abound. Lack of perception and insight is commonplace. As a result, a totally biased philosophy has appeared and has led to a DP culture of a rather paradoxical nature. Shortcomings have been turned into accepted inherent properties ("it cannot be different").

Baber gives an amusing parable. He compares the situation of software engineering to the situation of house building in the mythical land of Ret Up Moc, a legendary empire coeval with the Egyptian empire. In this land great advances were obtained in house building. Architects and builders were extremely qualified and did an excellent job. As a result, there was a boom in the building industry and builders were in great demand. The experts had no time anymore to train new experts, so that the schools developed emergency programs to train new people (or re-train older architects). The training was concentrated upon memorizing checklists and recipes of building. No time could be devoted anymore to explain the why of the various rules (let alone that the instructors had this knowledge). A new class of builders emerged, who knew all the rules, but they could not interpret them. So, they set about their building task, and built many houses. As a result, a sizeable proportion of the houses collapsed within their first year or even during construction. Worried about these problems, the rulers of Ret Up Moc created rules for testing a house. Some tests had to be conducted during the construction, other ones at the end. One of the tests, during construction, was pouring tons of sand on top of the house. If it didn't collapse, the sand was removed and work could proceed. At the end some similar tests were conducted and if everything was stable, the owner of the house had to sign a contract discharging the builder of any liability.

Any one who takes a critical look at present day software engineering will have to admit that the situation is very much the same as that of the building engineering in Ret Up Moc. A ridiculous and unacceptable situation of course! What can be done to correct it? One obvious approach is to fundamentally revise the education system. Another one is to create tools that help the software engineer in controlling his activities at the conceptual level (CASE: computer aided software engineering). Such tools might contain the checklists, thus relieving the engineer from manipulating them and allowing him to concentrate again upon the why. The tools should also deliver various perceptions of the design phases by producing "perspectives" or alternate representations. They should enforce the notion of decomposition and weak connection. For instance, drawing a line between two module boxes means

the modules are (and will be) connected in a standard way, which can not be modified into garbled exceptions anymore.

However, I have noticed that many users of CASE tools tend to consider them as even more "blind" checklists and accept without discussion whatever the tool produces after having pushed the button. Clearly, this is not at all what such a tool is meant for.

Many roads lead to Rome

There are many ways to look at an information processing system and its environment. According to Bemelmans, there are four levels of observation in the system. For each of these levels there are development activities that should be realized during the lifecycle of the system. The first level is the description of the *why*. What is the use of data and processes, what is their impact on the organization, which business functions do they execute. What we are concerned with here are the *pragmatics* of the system. Bemelmans calls this level the *systelogical* level. The second level is where we should devote attention to the *what*. What is the meaning of the information, which are the objects and events of the real world that the information is about? What are the transformations that processes bring to the information? What are the information streams? This is the *semantic* aspect, and Bemelmans calls it the *infological* level. The third level deals with the *how*. It comprises the description of data and processes in a suitable implementation language: it is the implementation itself, the program writing so to speak. We are concerned here with the syntactic representation of data and its processing. Bemelmans calls this level the *datalogical* level. Finally, the fourth level is the *wherewith*. It describes the implementation means, like the hardware, the receiving operating system software, the network, the processors and other such paraphernalia. This level is the *technological* level.

Bemelmans formulates the strict requirement that the four views which lead to four models of a system should be kept strictly separate from one another, so that the semantic description of data cannot be influenced by its physical representation, for instance. In other words, if for some reason a physical representation changes this does not mean the semantic description must be revised. Of course, this discipline becomes harder to respect when we look at the relationship between the datalogical model and the technological model: only if the syntaxes used are in fact pseudo-code descriptions will the two levels be independent. Our programming languages of today (most notably Cobol, but even the 4th generation languages) are very heavily impacted by the underlying physical atrocities. This unfortunate aspect tends to evolve as notions like data independence and object-orientedness suggest.

ISO took a slightly different avenue in describing an information system. They look at the system from the standpoint of its users, since that is what a system is for in the first place. Therefore, they state that there is indeed a level where the why and what of a system is described at a suitable level of abstraction, just like in the

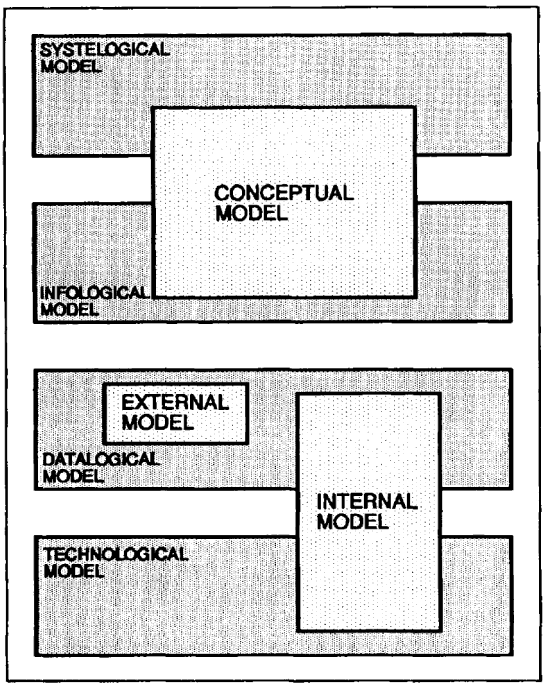


figure 9

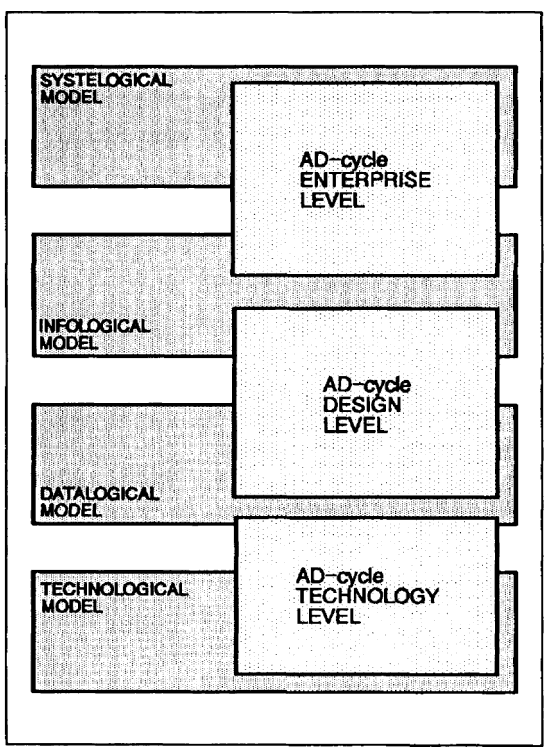


figure 10

systelogical and infological models of Bemelmans. ISO calls this why-what model the *conceptual* model. The real implementation with the programs and the host system aspects is invisible to and inaccessible by the end users of the system, they are reserved for the programmers. For this reason, these aspects constitute the *internal* model (identical to the datalogical and technological levels of Bemelmans). Finally, the implemented system has a presentation interface through which it is seen and operated by the end users. This is the *external* model (not really expressed in Bemelmans).

The correspondence between the four levels of Bemelmans and the three models of ISO is represented in figure 9

In recent years, IBM has set up a set of system development guidelines, assembled in the AD-cycle strategy. Here, three levels of system development activity are recognized. First there is the description of what the organization in which the software system will have to be integrated is all about. This is the *enterprise level*, which speaks about users and their grouping, systems and their decomposition. The level beneath it is the *design level* where the software solutions for the system to be implemented are described. Finally, at the bottom end there is a *technology level* which regroups all the implementation aspects such as programming, physical data-base design and other niceties. The levels in IBM's AD-cycle are more pragmatic than the levels in Bemelmans. In a way, the

enterprise level contains the systelogical level. The datalogical level is found in the design level whereas the technological levels of both approaches overlap rather smoothly. The problem is in the infological level (which encompasses ISO's conceptual model). In reality, this level is the result of the analysis. There is apparently no room for it in AD-cycle. The best equivalence is to see it at the bottom of the enterprise level and at the top of the design level (see figure 10).

The data-driven world

In all methods used today for the development of a software system, two classes of objects are distinguished: objects that carry information and objects that process information. The methods are all working with some form of information model (which becomes a data model by refinement) and of functional model (which becomes a task or process model later on).

When designing an information system, the two aspects of information and processing must be taken into account. However, experience tells us that the description of information is much more permanent than the description of processes. Moreover, processes are justified by the information that they process and not the converse. For this reason, many modern methods are devoting a lot of attention to (and indeed start with) modeling the information. The business world is information driven and therefore the software systems are data driven. This fact is proved very clearly by the commercial success of data-base management systems.

If we want to represent the information as a suitable (implementable) model, there must be a consensus regarding the reality that the information pertains to. This reality is what is called the *universe of discourse*. This universe is a collection of abstract and concrete objects that one manipulates or speaks about. Furthermore there are sentences (propositions in formal logic) that describe the behaviour of objects and their relationships to one another. The very description of the universe of discourse as a set of objects and propositions is the most difficult task that one can undertake. One may indeed think that such objects merely exist and are to be observed in an objective way, so that a given description (set up by one person) is comprehensible to another person *without any difference in meaning*. An invoice is a clear cut object and it has the same meaning for everyone. Experience shows that this objectivist view is a fallacy. The reality (if it exists objectively at all!) is perceived through personal filters. The comprehension of the reality is subjective. Every user has his own view. This is where a vicious circle appears: it is not possible to make a model of the reality based upon the various subjective views other than by creating a consensual objective view, and impose it upon the protagonists. They will, however, still look at it subjectively. Moreover, the objective abstraction that

we make can only be valid for the consensus group. It is not valid for other groups. These facts explain the utter misery of information modeling. Fortunately, there is enough discipline in a company for a consensus to exist about specific subsets of the world of discourse[11]. That groups have their own (subjective) view of reality is a fact that the data analyst must accept. As a result, the world of discourse cannot be represented in just one model, but rather by means of sub-models, one for each consensus. These sub-models can be constructed independently, but must eventually be merged and this calls for conflict resolution rules. Nevertheless, it is important that each consensus group keeps seeing its own sub-model[12].

When we speak about information, we should not forget that there are two aspects to it: *knowledge* and *communication*[13]. Information that resides somewhere (in the head of an end user or in a data-base) is knowledge. This knowledge is not interesting (indeed one may doubt its very existence) if it does not get used. At the basis of using knowledge lies the need to transfer it, to communicate with other persons (or processes). The communication aspect calls for a well formed description of what is valid information and what is not. That is what an information model is for: the various processes in a software system that exchange information can draw their information screening discipline from the structures (the semantics!) that reside in the information model. It is indeed vital that the ultimate receiver of data interprets it as the original sender intended. Throughout the system and with the users, there must be solid rules of interpretation regarding all valid information that the system can process (this is usually called the *Helsinki principle*).

The abstraction rounds

All the lifecycle models try to represent the same mental process and its steps: *abstraction*. According to Brachman, the creation of a system proceeds in two movements: one is a path that starts from the reality (the world of discourse) and creates a very high-level abstraction, the second one takes this abstraction and descends to a concrete implementation. The path up is a path of understanding and conceptualization, the path down is one of design and implementation (see figure 11). Brachman distinguishes five levels of abstraction. At the *linguistic* level, the process that takes place is one of gathering information from the sources that may exist (the end user for instance) and cleaning it up, ridding it of redundancies, contradictions and ambiguities. The next level, the *conceptual* level, represents a work

11. Due to the aforementioned vicious circle, the implemented software system will help in maintaining the consensus. Conversely, breaches in the consensus will contribute to the fast obsolescence of the system.

12. This principle is strongly enforced in an information modeling technique devised by Philips (the Netherlands): INFOMOD.

13. Human speech is very sloppy about knowledge, information and data. These words are used as synonyms. Let us say here that knowledge is stored somewhere; information is a higher concept: it is the explicitation of meaning (thus: knowledge is an amount of information); data is a form of storage for knowledge (and therefore of information).

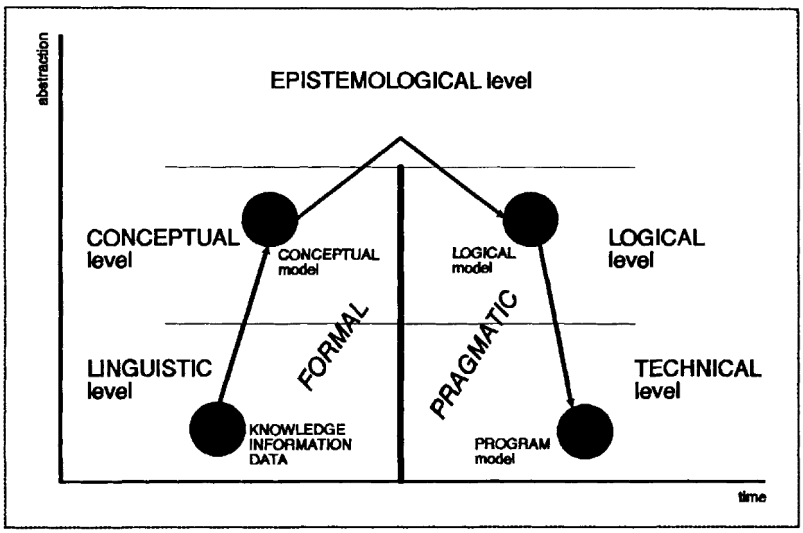


figure 11

of abstraction: concepts are uncovered in the collected information and the information gets classified according to these concepts. In more detail, at the conceptual level, one describes domain information (for instance in a semantic information model), flow information (how information leads to other information, an information flow model), control information (how the flow is controlled by tasks, processes and functions) and strategy information (such as plans, scenarii, major cases, ...).

From the conceptual level, we produce one higher abstraction, at the *epistemological* level. At this level we uncover the essential structuring principles of our model, which we can understand as rules that govern the specific modelling process itself and remain valid throughout all subsequent activities.

When a model of this high abstraction level has been obtained (a conceptual model), we are in a very special situation: the model is correct (or assumedly so), whatever the implementation of it may be. The model does not depend on any technical or system-imposed decision. It is free from programming constraints. The next activity, therefore, is to constrain the model in accordance with the retained implementation formalism. This results in a logical model and takes us to the *logical* level. Finally, the work becomes as concrete as it can be by proceeding to the *technical* level, where the actual program model is designed.

So, what is a method?

By now, it should be fairly clear that a method must meet several criteria. Certainly, a method must *span the lifecycle* of a software system. In other words, its features and benefits must extend from the very (pre-)study of a system down to the ultimate death of the working system, thus including both the development life and the maintenance life. Looking at most existing methods with this fairly trivial criterion reveals that most of them do not even meet the criterion of the full lifecycle.

Next, a method must offer a consistent *framework* within which the various lifecycle phases can be articulated and expressed. The least one may expect here is a consistent terminology. If the method comes with graphical representation means, these must use consistent icons and a unique user interface. This requirement eases the learning effort for the method, but also it avoids representation errors. Methods can be open-ended in the sense that they may accept input from other methods or Case tools, but then there should be some kind of automated conversion feature that controls the up- and downloading between the two representations.

Since a software development project must be monitored according to a chosen metrics, a method must implement a discipline sufficient for allowing this metrics. This does not mean that the metrics must be in the method, but there must be sufficient milestone activities allowing easy assessment. Specifically, the method should enforce a disciplined flow through the lifecycle, disallowing a next step when the previous one has not been terminated. In other words, a disciplined method calls for a project controller and should have software tools that automatize the enforcement of the discipline. In particular, a state of the art method should rely upon a computer based repository in which the various objects that the development activity has to use are represented and controlled. This being said, the discipline may not be a harsh one: for instance, a method that states that prototyping is disallowed is not acceptable[14].

Of extreme importance are the targets and objectives of the method. Whatever the method comes up with should be justified by the objectives. Some methods increase the productivity of the developer and provide the means to achieve this; other methods concentrate on the quality of the end result. Which of the two approaches is best is an open question[15], but it is important that the method states its purpose, so that the user can accept the discipline (or not). There are methods that engulf the developer in a mass of documentation, but, one wonders, to what purpose? No one ever reads the documentation (and I am sure that no one ever controls the documentation for completeness, correctness, consistency,...). So why should we lose much time on a gratuitous exercise?

In the same vein, a method should offer sufficient representation techniques so that the various aspects of the reality that need modeling can indeed be modeled. Some methods offer much in the way of the functions, but leave the information modeling uncovered. Others concentrate on information modeling, but say nothing about the processes. These are incomplete methods which leave the developer out in the cold. Of course, the method should be clear in these matters: too much is too much! There is no need to represent the same portion of reality in more than one model. Alternative representations must be available when needed, that is the important aspect.

14. Many methods are indeed incompatible with prototyping.

15. Enough quantity with sufficient quality is the best of all worlds.

Last but not least, the method must be clear about its fundamental paradigm. How does the method expect its users to visualize the reality? This is essential, because it determines a number of representation particularities that one must accept without discussion. For instance, there is quite some difference between the paradigm of processes that store and retrieve data and that of processes that exchange messages. These paradigms are more diverse in the realm of action representation than in that of data (information) modeling. In many ways the paradigms are equivalent (they represent the same reality, don't they?), but their finer discipline differs so much that it is very difficult to combine them.

As a result and in summary, a method should not be particularly difficult: it must be comparatively easy to learn (in fact, it should suffice that one understands the basic underlying paradigm). The method should be usable by end users as well as software engineers (not at the same level of detail, but certainly at the conceptual level). The representation means must be rather natural or at least have a reduced number of icons and reserved words. In fact, by using the method's terminology and representation syntax, it should be possible to explain a software system to an external person (training effect) and conversely, the method's syntax must allow interested parties to evaluate a foreign piece of software (bought applications for instance).

The best way to conclude is by stating that a method must be acceptable to ***all*** persons concerned. That this is not without cultural incidence is clear, but one must avoid the appearance of the method fanatic as well as the unconditional rejection attitude. Many methods are worth one's while, but they must be pruned somewhat for certain aspects and extended with company discipline for others. There is no such thing as a *best method*.

Software Engineering is a social exercise

The top-level activities contained in the software engineering exercise take place in the real world, are done with the end users at the *enterprise*[16] level. Therefore, these activities have a rather important social impact, either because they depend on human input or because they dictate human behaviour as a result.

There are behavioural assumptions underlying the work of the software engineer. These assumptions are based on the type of social environment (or vision of the environment) where the work must take place. One such type is that of an ordered society characterized by stability, integration, consensus and functional coordination. The converse type is that of conflict where change, coercion and disintegration reign. Of course, these society models can be combined with an objectivist or a subjectivist view of reality. The behaviour of the analyst is largely determined by the chosen coordinates: *objective+order* or *objective+conflict* or *subjective+order* or *subjective+conflict*. The resulting models are of greater or lesser quality accord-

16. This is AD-cycle terminology, of course.

ing to the coordinate choice (or the imposed coordinates) and the implemented system will display this same degree of quality (or lack of it).

The reality is not to be grasped in an objective way, as experience has amply shown. Subjectivism, hopefully constrained by sub-system consensus, is the rule. As a result, the analyst must display a great degree of relativism: he must perpetually measure the information given by one user against the information given by another user. The role of the analyst is to interact with management in determining which system makes sense, preserving the individual understanding of the users (or groups of users). In solving conflicts of interpretation, the software engineer must permanently work with the users and ease the transition from one (rather false[17]) viewpoint to another (rather correct) one.

When re-positioning the subjective dimension in the society-in-conflict context, it becomes clear that the software engineer tries to resolve conflicts, by taking the side of those the system is destined for: the end users. In all ways the system will allow the emancipation of the users by removing the barriers to rational discourse. In order to do so, the software engineer must acquire knowledge in the technical domain concerned by the system; this is an obvious requirement which is always the basis of the engineer's work of course. But the software engineer must also increase his knowledge in mutual understanding, since the system has this kind of emancipatory implication. Therefore he must devote full attention to the cultural aspects and backgrounds of the users involved in the system. He must understand particular system requirements by comparing them to broadly similar systems[18]. Finally, the analyst must also gain knowledge in the emancipatory aspects themselves. The analyst will elicit a shared understanding of the many obstacles to human communication. He needs to acquire an appreciation of the different viewpoints of the different stake-holders. This cannot be done by external observation: genuine participation is crucial.

While eliciting the various knowledge aspects, the analyst must be aware of factors which can hinder human communication, such as: authority and illegitimate power, peer opinion pressure, time and other resource limitations, social differentiation, biases, loss of power and language barriers. These factors make it difficult to understand the relevance of system requirements. Correct system requirements and objectives can only emerge from a free and open discussion which must lead to a shared agreement without suffering from the indicated barriers.

The realized system must also provide for increasing the knowledge of its users in the three indicated domains: technology, mutual understanding and emancipation. As a result, information systems can facilitate the move towards improved

17. The *rather false* and *rather correct* qualifiers are themselves subjective; they are probably imposed by some form of strategic management.

18. Making sense out of a new situation by comparing it to some similar situation is what *hermeneutics* are about. This is normally the domain of the jurist, but it seems that DP professionals are also great practitioners of this human science.

technical control, better mutual understanding and continued emancipation of the stake-holders. In fact, information systems are developed also to facilitate a wide debate on organizational issues, free of social pressure. This type of ideal discourse is made more easy by a number of realizations during system development that tend to remove speech barriers:

- information modeling contains the representation of semantics, thereby allowing validations that avoid interpretation distortions;
- well balanced project management and logistics can motivate the participants into sharing and eliciting missing information;
- object-oriented designs can help in overcoming educational differences;

The way in which the software engineer profiles and defines his activity is based largely upon his personal culture and the company culture. Nevertheless, it is also a fact that the newer methods for developing software have more and more taken into account the subjective nature of the reality. Most of them still tend to favour the ordered-society view, where every issue is determined by consensus and nothing changes. But even that is not true: changes abound, coercion exists, conflicts of interest are commonplace...

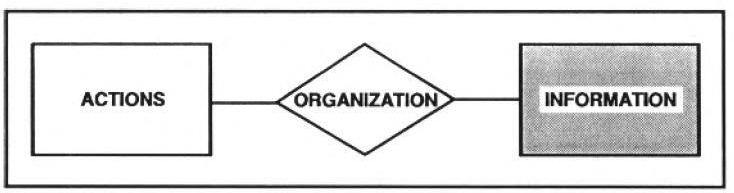
Should a software engineer be a computer scientist? No...

Should he be an organization analyst? No...

Should he be a social worker? Yes, and some of the above as well.

ENTITY 1

INFORMATION



WARNING

The subjects covered in this part of the text are about information modeling and its obligatory sequel: data modeling.
The reader is assumed to have a fair knowledge of the relational data representation, even though this is not used here as the fundamental model. A brief introduction to this model is included in the text, but space constraints prevent me from giving a full criticism.
A fair portion of the text is devoted to semantic modeling, where understanding the information is of the essence, whereas the remainder of the text deals with the actual modeling of the data structures, achieving the (almost) zero-redundancy objective.

The modeling techniques presented are those that -based on personal experience over years- have given me the best basis for achieving quality in the resulting models.

The associate subject of information architecture modeling is covered in another part of the text (it is inculded in enterprise modeling)

CHAPTER 2
The Semantics of Data

Preamble: Codd's relational model

That Data has a structure is an accepted fact. Clearly, all forms of list structures are conceivable for representing data. However, such structures offer too much freedom. Thus, the pragmatic world took another avenue. An avenue first explored by Ted Codd, leading to the famous *relational model*. What Codd stated, as an axiom, was the existence of a group of data fields, i.e. the list of related fields. Fields are called *attributes* (and sometimes *columns*). Such a list was given the name *relation* (sometimes also *table*). Thus, a relation is composed of discrete attributes. The relation has a name which allows programmers to perform operations against it. For the attributes nesting is not allowed, in other words fields cannot be group-fields: a column should not have a finer structure. One says that an attribute is *atomic*. It may, however, have a *binary* structure: the value in the column may be absent or present; an absent value is a *null value*, and this can be allowed or disallowed in any given column. An example: a relation holding employees has a column for the name of the spouse. Bachelors will have a null value in this column.

Codd also defined the *tuple* (sometimes also called *row*): one significant value of all fields of the relation, such that the aggregation of these values constitutes a significant occurrence in the usage of the relation. In more pragmatic terms, a tuple is an *occurrence* of the relation.

In the pure scientific definition, the relation is the set of its tuples (just like a file is not the layout of a record, but the set of all occurrences of a given record layout). Therefore, if we draw it up on a sheet of paper, a relation looks a lot like a *table*, with named columns and filled with rows (these are the tuples). Obviously, this is a rather attractive way of viewing data, isn't it? It corresponds to what data usually is in a non-automated environment. So much so that I will use the term *table with columns* as a synonym of relation with attributes (this has become daily habit in programming circles; strict relationalists do not accept it, however). In fact, a relation is none other than a new name for an old object: the file! The associate discipline is not that of files, however, as we will see presently.

In a way, Codd followed the Pascal philosophy: each column (or field or *attribute*) is defined over a *domain*, another way of saying that it was of a given (strong) type (strong typing refers to the fact that a program using a field cannot assign a value to it that is not allowed by the domain of the field, nor can the field be

assigned to a field with another domain; manufacturers tend to be rather lenient and vague in supporting this kind of discipline; many of them allow some kind of conversion or coercion between domains). This aspect of typing is called *domain integrity*.

Codd did more: he stipulated that identical rows in a tuple were to be banished, as they would be totally indistinguishable and would therefore be unusable. But, if duplicate rows are to be avoided, that means that rows are somehow distinguishable. How? By the fact that any pair of rows differ in the value of at least one field. And, noticed Codd, quite often this is the same field for all rows of the table. So, a table may contain a column that has something special about it: it can be used (or, better, its values can) to *identify* the rows of the table. This means that the column actually *identifies all the other attribute* (values) of the row. Such a column is a *primary key*. Of course, a primary key can be made up of two or more columns taken together. The important rule is that a table must have a primary key, even if it is made up of the concatenation of all columns of the table. This is what Codd called *entity integrity*. That the primary key can be made up of agglomerated fields does not ruin the atomicity of attributes. A program can only access atomic fields: it cannot compute nor compare primary keys. The primary key is reserved for usage in specialized instructions with well defined semantics and no possible deviated usage.

A significant example of concatenated primary key usage can be found when one tries to use the relational representation for higher level tables. Indeed, the relational table as seen corresponds to the single entry table. What if we have a 2-entry table? In a single entry table, the breadth of the table is limited at any moment (it is the list of fields), whereas its height is unlimited (the set of tuples). In a two entry table both breadth and height are unlimited. Consider for instance a situation in which we want to represent the price of parts delivered by various suppliers (at a different price). We will make a 2-entry table, with one row for each supplier# and one column for each part#. Each cell (crossing of column and row) will hold the price of that particular part for that particular supplier. A very familiar representation. It can be easily extended if we want to know more, e.g. the available stock of a part with a supplier. This is added in the same cell as the price, so that the cell becomes a structure of two fields. In general, the cell in a 2-entry table is composed of a fixed (but not limited) number of fields. Clearly, such a table is not relational. But we can turn it into a relational table by defining price and stock as usual columns, by adding a column for (say) supplier# as a primary key, and have as many rows for this particular supplier as there are parts deliverable by him. The table has an integrity violation: its primary key is not unique. This is as expected: we must distinguish the parts supplied, so we add yet another column, part#. What we have now is a single entry table with the columns supplier#, part#, price, stock and the concatenated primary key supplier#+part#. What was done for 2-entry tables generalizes by induction for n-entry tables: they correspond to relational tables having a concatenated primary key of n elements.

This being said, strict relationalists do not allow the primary key to be composed of more than one field; if no such key can be found, they add an artificial column to the relation holding some unique value (e.g. a sequence number). Under this strongly restricted model, there is an equivalent simpler model: the *binary relational model*. A table (A,B,C) where A is the primary key can always be replaced by as many two-column tables (binary tables) as there are non-key attributes, as follows: (A,B) and (A,C). In this simple structuring scheme, null-values are more naturally accepted: if indeed C can have null values in the original table, this merely means that *not* all A values occurring in (A,B) also occur in (A,C).

Some very important properties pertain to primary keys. First of all, they have unique values. This is an essential aspect of the key since it is only by that uniqueness that the key can identify attributes. Consequently, a field that has unique values today by chance, but may have duplicate values tomorrow can never be a primary key. The primary key of a table must have a significant value in all rows of the table, it is never an irrelevant value (*null value* not allowed). If a primary key is composed of many columns, none of them may ever have a null value.

In fact, each column or column combination that has essentially unique values in a table is a potential primary key. And if there are many of these in a table, they constitute *candidate keys*. A problem of choice: only one of them is really to be used as true primary key, the others may be demoted as *alternate keys*.

According to Codd, a relation's tuples should be unordered. It is irrelevant to know that the row with key value ANC is the 768th one, and it is just as irrelevant that therefore row 769 has some specific related meaning. Rows should be interchangeable, without effect upon the applications using them.

Pragmatists allow any field (or group of fields) in a table to be decreed a key allowing access. Such a key is called a secondary key. It has no criterion of uniqueness (in the course of subsequent data modeling case studies we will see that this freedom is really a trap because it contains a chicken and egg paradox). A secondary key will very often serve to sort the table and therefore it does imply an ordering. For that reason, pure relationalists avoid speaking of such keys.

So much for the table as a stand-alone being. In fact, a table is a highly simplified list structure, simple enough to allow its implementation in any kind of file system and any kind of programming language. The point now is: can tables represent all the data structures one needs? They look simple enough to make their usage attractive. But are they coherent? We will see in due course.

Codd's definition of a relation gave a specific meaning to the word *relational*. It stands for "pertaining to Codd's definition of a relation". Therefore, when data is represented by using only relational tables, we will say it is represented as a relational model. A data-base that implements only such relational tables and nothing else may be called a *relational data-base*.

In fact, it would be better to call Codd's definition the *relational paradigm*.

Relational pragmatists have a more "human" definition of the relational database; it is a data-repository with a suitable management software such that:

- the data is stored as relations only (no physical structures are visible at all),
- the entity integrity and domain integrity rules are enforced,
- the access features are free of any physical impact (e.g. they are made up from the relational algebra (PROJECT, SELECT, JOIN,...) or from the relational calculus (e.g. SQL)); moreover they are *set-oriented* in the sense that they operate on a set of rows and not just one row in isolation.

The above definition is not sufficient though.

In the relational model as promulgated by Codd, tables are not to be linked. But, in true life, tables are not independent! A table of employees certainly has some relationship with a table of departments, otherwise how can we express the fact that an employee works in a department? Relationalists said that such a fact must be expressed by means of fields only. So, a table A can *refer to* a table B, if table A contains a field that is a duplicate of a field of table B. This new field in table A is a *foreign key*. Thus a table of employees can refer to a table of departments by containing a department number. The actual meaning is that an employee is working in one and only one department. Of course, if each employee is in two departments, there might be a second column with a department number, and a third, a fourth... The number of such references, i.e. foreign keys, in a table is a static (not a dynamic!) property of the table. This is where tables differ fundamentally from lists.

Obviously, foreign keys play an immense role in the join operations. If we want to print employee tuples with their department information, we will *join* the employee and the department table, using the department number in one table as a link with the other table.

Early relationalists refused to see a foreign key as something special; to them it was a column like any other one. But there they were fundamentally mistaken and this caused the *great debate* which opposed Codd and Bachman.

A foreign key expresses something more than just a value. Obviously, it has a flavour of redundancy, since it duplicates data from another table. If a foreign key is to be at all usable, from the table in which it appears it should refer to one single row in the table referred to. Thus, the foreign key must be the match-duplicate of one of the candidate keys of the other table. The existence of a foreign key-candidate key link between two tables effectively relates the two tables. It does so in a very specific way: any one tuple of the table that has the candidate key is related to zero, one or many tuples of the table that has the foreign key; however, a tuple of the table that has the foreign key refers to only one (or no) tuple of the table that has the candidate key. The stated rule is valid for any given candidate/foreign key pair of course. In other words, the foreign key expresses a 1:n relationship, absolutely identical to the set of Codasyl (semantically speaking of course).

The existence of the link leads to an important integrity aspect. Indeed, what happens when the tuple containing the candidate key referred to (e.g. a department) is erased? Are the tuples containing this value of the foreign key (the employees of the department) affected? Three situations can be envisaged:

1) the foreign key tuples (employees) are also deleted;

2) the foreign key tuples (employees) are modified so that their foreign key value becomes null (the employees have no department any more, but they still exist);

3) the deletion of a candidate key tuple (department) is not allowed if there exist foreign key tuples (employees) referring to it.

Obviously, rule 1 can cause propagation of deletion. All three rules leave the tables consistent. This is a very fundamental problem. Reluctantly, relationalists called it *referential integrity*. Of course, it had already been solved satisfactorily in Codasyl data-bases.

Referential integrity also impacts on INSERT and UPDATE operations. Indeed, can a row (an employee) be stored which contains a foreign key value that has no corresponding candidate key (no department)? Can a foreign key value be changed, possibly so that there is no longer a corresponding candidate key? Or can a candidate key value (department number) be changed, even if it has corresponding foreign keys? Should the change not propagate down the line in that case? Important questions, all of these. The relational model remains silent.

Interestingly, tables can be classified based upon the foreign keys they contain. First of all, there are tables without foreign keys: these are called *kernel* tables. A customer table could be of that nature. Some tables may have a foreign key referring to another table: these are dependent tables. An order table with a customer number reference is such a table. Next, there are tables that may have a primary key of *composite* nature made of a foreign key (possibly of more than one column) *and some sequence number*. Such tables are *detail* tables or *characteristic* tables. For instance, an order line table which has a primary key composed of an order number and a line number, but where the order number is a reference to the order table (the order lines are of a particular order). Then there are tables whose primary key is composed of only foreign keys (each of them possibly comprising many columns), a *compound* key, which are called *associations*. The example is that of two tables: a table of teachers and a table of subjects; a teacher teaches subjects and this is expressed via a third table which has the primary key teacher number composed with subject number; teacher number is a foreign key referring to the teacher table and subject number is a foreign key referring to the subject table.

It is important to notice that kernel tables may *all of a sudden* become associations or characteristics (or just dependent tables) when a new kernel table is created with a primary key composed of (part of) the same primary key as an existing table. For instance, customer is a kernel table; now we create a new table called "activity-sector" with an activity code as a primary key (e.g. banking, insurances,

manufacturer, ...). The customer table is extended with a column referring to the activity sector, so that it has become a dependent table. It is the claim of the relationalists that such a change is trivial in the relational model: a mere addition of a column and no structural change whatsoever! This may look true, but the impact on programs is still important: additional join operations will certainly be needed; additional update constraints will have to be validated. Admittedly, the program changes may be easier to work out with relational access languages than with Codasyl's DML. Another aspect is that a table may very well have to be split into two tables, because of changing business rules. In that case, the resulting program changes are just as difficult with a relational approach as with a network approach. Since relationalists deny the physical truth behind tables, the splitting of a table for physical (tuning) reasons is not even mentioned. The new generation of relationalists, confronting this problem, state that such a split is *never* required, because all accesses are optimized by the use of a system-based access optimizer (little do they know how difficult a task this optimization is; it is not independent of the way in which a program formulates a relational operation; the optimization does not preclude the need to tune tables by splitting; many situations cannot be optimized (there is scientific proof for some such cases)).

The result of a relational retrieval operation is a new table of a temporary nature, residing (at least in principle) in the program's working storage. But of course, this need not be so. It is just as possible to see a relational operation as a way not to actually access the data, but as a definition of the data that is accessible. For instance, we could write something like

select * from customers where city="Nankin"

in order to retrieve all customers living in Nankin (a city located in mythical Lemuria); in fact we could just as well write something like:

declare selectable customers where city="nankin"

and this time we mean to define (but not immediately access) the sub-table of the Nankin customers. This sub-table does not exist physically, not as such at least! To all purposes, a declaration like the above defines a new (fictitious) table that can therefore be accessed by means of relational operations; such a defined table is called a *view*. In order to allow access to a view, it is required to give it a name (and we will polish the syntax somewhat using the word *derive*):

declare nankincust view as derive customers where city="nankin";

and we could select all Nankin customers who have the status "banking":

select nankincust where status="banking"

The declaration (the actual terminology is: *derivation*) of a view is in fact the relational operation that is capable of extracting the sub-table. In other words, any relational operation can define a view. On the other hand, a view is a table, even if it is not residing somewhere explicitly. Therefore, it has all the properties of a table: it has columns and rows, it has a primary key (or at least it should have one), it has

no apparent order and it can be accessed by means of relational access operations like any other table.

Since a view is a table, it is allowed to use it to derive yet another view from it, and this can go to any depth. Most manufacturers are reticent about views, though, and therefore do not offer such luxury. Even the relationalists are rather reserved about the whole concept.

True, there are a number of problems. A first one is: what happens when a table is copied to a table? Reply: the first table is copied tuple-wise into the second table. However, if the receiving table is a view, it does not physically exist. So what happens during assignment? One may envisage refusing the assignment. On the other hand, one could envisage to *invert* the view mechanism and move the tuples to the tables that are the constituents of the views. But this is awkward: can these tables be updated correctly? What if the view is defined as a projection? Then there are missing columns, and can these be *inverted*? Obviously not. The update is not safe. Inversion of the view derivation can be envisaged in most cases where no projection is involved. Indeed: for an intersection view, the tuple offered would need to go to both source tables. For a join view, the tuple must be decomposed (after verification of the join condition) and moved piece-wise to both source tables. For *union* the situation is ambiguous: to which of the two constituent tables should the new tuple be moved?

For the rewriting of a modified tuple (UPDATE operation), one may also envisage a carrying through into the constituent tables as explained above. But what if it is a DELETE operation, and more specifically, for a JOIN view? Should tuples be deleted from both constituents? Or only one of them? There does not yet exist any formal theory about the safe updating of views... However, for each view it is of course possible to determine the ***ad hoc update policy*** (or policies) according to the functional usage of the view. It suffices, therefore, that manufacturers provide an exit level programming language allowing a programmer to write the necessary update routines. Extended in this way, views become formidable means allowing the exact application of what is called *program to data independence*. This type of independence means that changes in the data structures do not impact on programs at all; the only thing that may need to be changed is the view code (and even so, this can be avoided in many cases because of the underlying optimizer technology). Obviously, such chnages to view can be complex and may induce ripple effects. Therefore, each view must be placed under the very strict responsibility of one suitably chosen person. The responsibility aspects pertaining to views are dealt with in chapter 5. A certain number of view usage recommendations can be found in box 2.

BOX 1: SUMMARY OF THE RELATIONAL MODEL AND ITS CLONES

THE LIST MODEL

(as implemented by COBOL structures and PASCAL records)

- Set of members (i.e. fields)
- A member is an atom (elementary field) or a list (group field)

THE RELATIONAL MODEL

- Data is represented only as relations (i.e. tables)
- Tables hold only atomic fields (there are no sub-tables nor arrays)
- Vectors are tolerated as fields ina table
- A table has a primary key (which can be a concatenation of columns)
- A table can have alternate keys and master indexes
- The primary key and all the alternate keys are candidate keys of the table
- Relationships between tables are expressed as foreign keys
- The relationships are of 1:n cardinality
- Reflexive relationships are supported (a table to itself)
- In an A to B (1:n) relationship, the foreign key in table B is the replication of a candidate key of table A
- The model needs rules for referential integrity
- The model has rules for domain integrity
- The model needs rules for user-defined integrity
- Preferable input/output operations are expressed as relational algebra or relational calculus (e.g. SQL)
- As an extension, the model provides for the definition of views
- There could be a specialized programming language allowing the definition of user validation rules in a view

THE TOPOLOGICAL MODEL

(as implemented by CODASYL (for instance))

- Data is represented as lists (records and structures)
- Relationships are represented as links between records (Codasyl sets)
- Relationships are of 1:n (master - detail) cardinality
- The m:n relationship (association) between A and B is expressed as A→X←B (X is a junction record)
- Multiple masters are allowed (they are not allowed in a pure hierarchical model)
- Preferable input/output operations are expressed as DML-programming statements

BOX 2: RULES OF GOOD BEHAVIOUR WITH VIEWS

- Each functional path of a program uses only those data views (user views) which are logically required for the function *and nothing more.*
- A view is the definition of an abstract table-like object made up of columns and rows.
- A view is almost certainly un-normalized (this is an essential property)
- The usage of a data view by the program comprises only the operations of Open and Close of the view and select, update, insert, delete of one or many rows of the view.
- Since most views are theoretically not updatable, there should be a specialized programming language allowing the creation of ad hoc view update policies (as a part of the view definition)
- Only unary relational operations (such as Selection and Projection) can be used in the program-based access operations; the n-ary operations of relational algebra (join, union,...) should not be used since they are not table-independent.
- A view can (but must not) incorporate the integrity rules for domains, entities, references and user-defined constraints.
- Views can be derived from one another by all operations of relational algebra or relational calculus; at the highest level views are very close to the application world; at the lowest level, a view has a one-to-one correspondence with a physical table (or record).
- View definitions may not reside in the programs that use them; they must be seen as objects with methods (information hiding and object-oriented approach).
- The physical tables should be suitably normalized (this is not strictly required, but highly desirable for reasons of update consistency).
- Fields of the tables may be pure attributes, or primary keys, or foreign keys, or alternate keys, or secondary keys; keys may be constituted of concatenations of fields; each table must have a primary key; dimensioned attributes should be allowed (fixed dimensions).
- The lowest level records should be composed of only fields; however, there can be no objection to group fields or fixed bound arrays (vectors), even nested, provided these fields are kept semantically clean; in particular, although any field of a group or vector is visible, no such field may itself be a primary key (or part of it); on the other hand, a group or vector as a whole may be a primary key (or part of it).
- The lowest level should enforce the integrity of all keys (especially entity integrity and referential integrity).
- A view should have a well defined responsible owner (non technical person), who masters the functional life of the view

Data has a life of its own

One of the most important facts about data is that it has a life of its own. This realization somehow came as a shock to Data Processing people: what, data is not something one can conceive according to program needs only? The Data Processing people had grown used to making programs in some self-justified way, they had freedom to structure programs from within, they didn't have to explain the internals of their programs. The same attitude prevailed as far as data was concerned: it was a commodity for the program. But data just exists before any program comes into life. Data is owned by the company for which the programs are developed. Data is a representation of the company. Data is composed of fields and these are the finest objects a company works with, whether there are programs or not! Thus, data has a structure that is company-owned, and Data Processing was going to be forced into accepting that structure.

In a very pragmatic way, it all amounts to this: data should be investigated and structured according to its *significance* in the company, and not because some program wishes to use it in some very local way! The next statement came as a real blow: the significance of data is known by those people who actually work with the data daily: the *end users* themselves.

The need to understand the meaning of data caused a new job to appear: *data analysis*. And a data analyst[1] need not be a programmer. He must be an organization expert, rather. Someone who knows how the company operates and what it operates with. Someone who can interrogate the end user, and come to know all the tricks. Thus, setting up correct data structures creates two requirements: ability to *see* structures (on the part of the data analyst) and end user *involvement*. The end users concerned *must* participate. Indeed, they are the only people who really know all that there is to know. They know which fields they work with and what these fields mean. They know the validity constraints of the fields. But do they know about structure as well? Not really. Structure is perceived implicitly by the end users. In fact, when they work with data (and that is all the time), they are not curious about it. End users do not work with data, in fact. They operate with a more abstract being: *information*. Information is extremely hard to define. It is part of reality, indeed it represents reality. In this context, reality is often also called the *universe of discourse*. However, information in business is reliable, procedural, systematic, repeatable. End users are trained to work with it in a frozen corset: that of *forms* and *documents*. An order is a form. The registration-card of a customer is a form. A bill is a form... End users are very good at manipulating such forms. Forms are the basis of all that happens in the company. Forms are vital. So vital that they should be taken as the basis of analysis. Let me therefore call such forms (or documents) *user views*. Collecting user views is the first part of the data analyst's work. The next step certainly is investigating what forms are used *for* (i.e. what *operations*

1. Often also called *data administrator* or *data designer*.

are done upon/with them). Indeed, a form is significant only if it gets used. One may say that information acquires a meaning only because it is related to other information, and that the relationship is actually used. Quite a change for Data Processing: programs are the actual acceptors or generators of forms! The programs are *information-driven*... A radical inversion of the work.

When Bachman produced his views about the relationships between records, as they were adopted by Codasyl, he was more concerned about the topology of the relationships and he formulated disciplinary rules about networks, trees, self-referencing, multiple members and multiple owners.

Eventually, Codasyl used these rules for the implementation of the so-called network data-base, emphasizing by means of the set the relationships that records have to one another. Little did they know that a major step had been taken.

Indeed, it soon became apparent that the meaning of data in the environment it was being used in lay not in the records, but in the relationships. Of course, the record is important enough: if we know that there is a customer #123, then the record informs us about this customer's name, address, and other such vital items. But knowing that a customer #123 exists is not sufficient. Indeed, what can we do with that knowledge? Surely, this can only be an initial situation... Customer #123 becomes meaningful as soon as he gets related to orders, when he can be sent invoices, ... In other words, the essential information is in the relationships[2]. This realization caused the emergence of representation techniques that stressed the weight of relationships. All of them are based upon *Bachman's entity diagram.*

Bachman diagramming

What we need to do in order to understand data is to make the information it contains explicit. Therefore, we will have to investigate the usage of data in the organization so that the various aspects of the data are clarified. In doing so, we will endeavour to recognize *entities.*

An entity is an information packet in an organization, which one can sensibly talk about, which all people involved have at least the same rough perception of, which one is convinced must be managed, and, above all, for which there exist registration forms of wide use in the company. In other words, an entity is "something" we want to store (and keep) data about.

Examples of obvious entities are: customers, employees, bills, pay-slips, machines,... Such entities have a clear physical existence[3] and are called *tangible.* Other

2. Funny as it may seem, the information given by a record is said to be *existential,* whereas the information held in the relationships is *relational.* This usage of the word *relational* has nothing to do with the relational model though. A very confusing situation. Even more so when one considers that relation and relationship are in fact synonyms.

3. Some entities, like bill or pay-slip, or are so much *linked* with their paper form that one calls them tangible also.

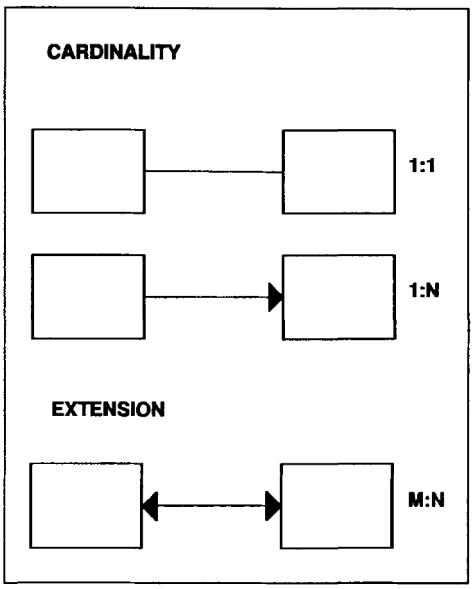


figure 1

entities are *non-tangible* and live only by agreement; by and large, they are part of a company's culture and may very well have no meaning in other companies: a profit-centre, a job, a function, a debt,... One thing is clear: there is something vague about entities: we know about customers, but the actual detail matters little. A slightly amazing fact is that all (business) companies use a largely identical set of entities, and not that many of them: 200 to 300...

Our next step is to discover the relationships between entities and represent them. We will consider entities to be related if they are used together in some process of the company. Moreover the relationship must make sense. An example: a bill is related to a customer, and is related to a product or a service or a cost. In a Bachman diagram, such relationships are depicted as links between rectangles that represent the entities. In Bachman modelling, the links have a limited *cardinality*: they express only a 1:n relationship[4] (including the degenerate 1:1 situation[5]). Although it is an illegal extension on pure Bachman diagramming, some authors represent the m:n relationship as well, see figure 1. As is indicated in figure 2, many authors have devised extended icons, especially in the representation of relationships and their cardinality. Bachman called the two ends of the 1:n relationship *owner* and *member*. The

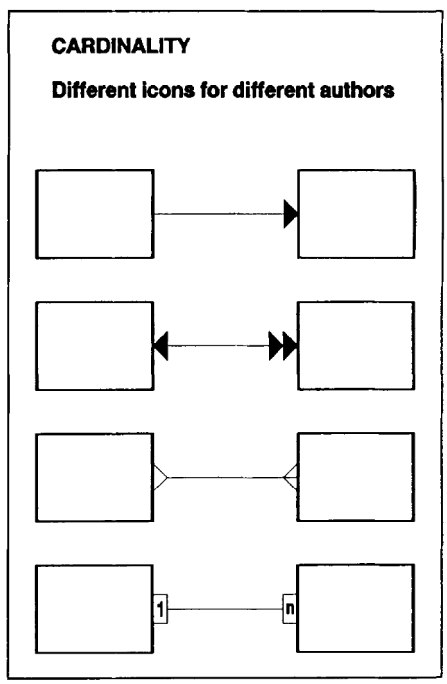


figure 2

4. The reason for this restriction is historical: there are no physical data bases that can implement a m:n relationship straightforwardly. Today, semantic modelling takes place without any consideration of implementability, it serves only to understand the semantics of the information. If a semantic model must lead to implementation, then the well known technique of normalization (and equivalent techniques) will lead to 1:n relationships.

5. It is a 1:n link with an arbitrary direction.

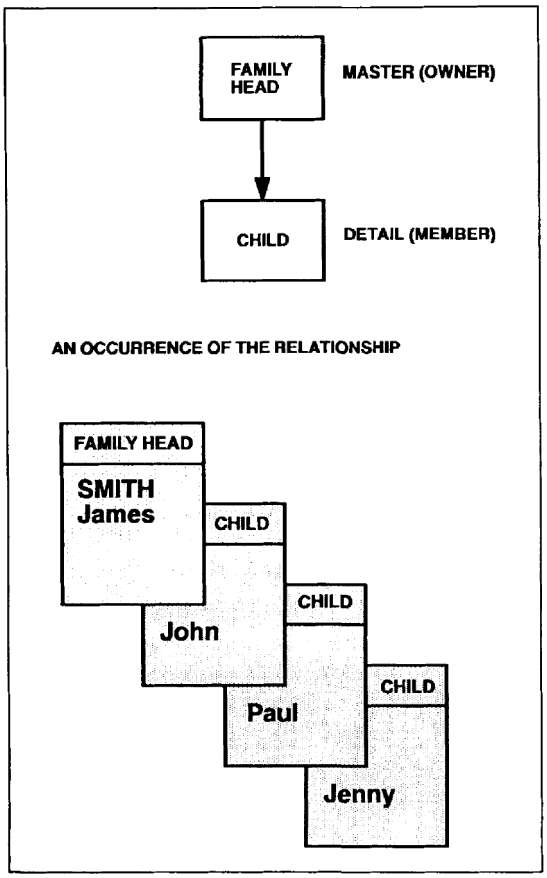


figure 3

Bachman relationship is an implementation of the *master* and *detail* paradigm. At the basis, this paradigm states that the detail belongs unambiguously to the master, of which it constitutes low level repeating data. In other words, it is unlikely that detail belongs to more than one master *occurrence* and therefore a master detail relationship cannot evolve to a m:n relationship.

As a result of this definition Bachman modelling is done according to the following rules:

1. An entity should not contain *repeating groups*; such groups should be expelled from the entity and must appear as a new, linked, entity as in figure 3; entity A_1 is now a *master* and X is *detail* or *characteristic*.

2. Many to many links between entities are not acceptable and must be replaced by a supplementary entity called a *bridge*, or an *association*, or a *relationship*, or a *junction* (all synonyms), linked to the owners by means of master/detail relationships (see figure 4). In fact, in Bachman modelling, the m:n relationship is seen as a *two entry table* linking the entities, see the example of a product-supplier situation in table 3. This table is itself interpreted as an entity linked to the supplier and the product entities.

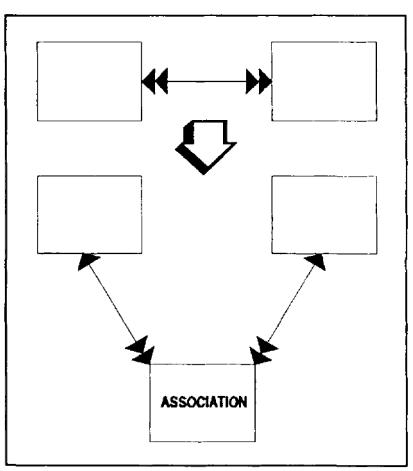


figure 4

------Part Supplier	*PART 34*	*PART 45*	*PART 566*	*PART 92*
ABC	70	45	.99	.21
KLM	128	99	.67	.77
XYZ	131	101	.77	.82

table 3

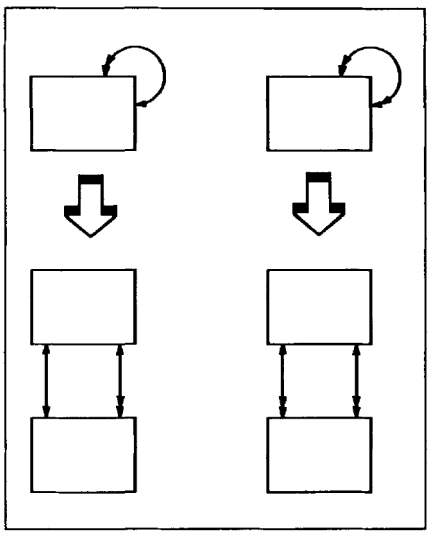


figure 5

3. Nests (i.e. links of entities with themselves) are not allowed because an entity cannot be its own detail; they should be replaced by junctions, see figure 5.

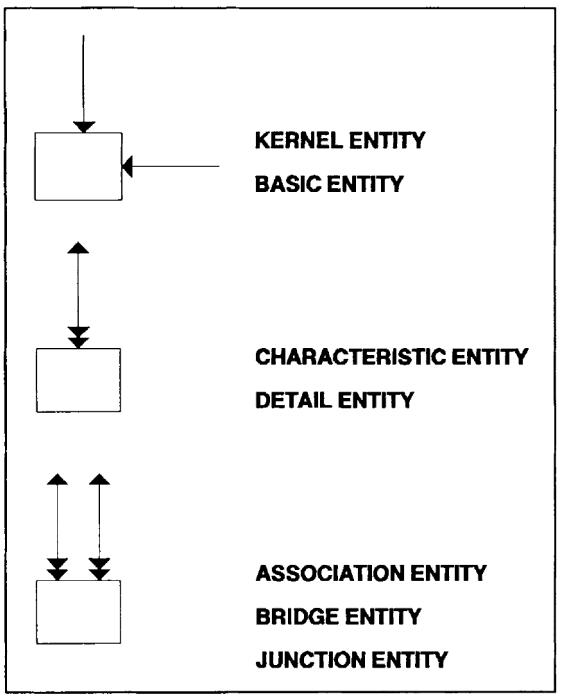


figure 6

In using the 1:n relationship for all possible cases, Bachman modelling is unduly restrictive and loses many of the semantics of the m:n relationships. Nevertheless, Bachman deserves credit for having been the first author to understand that relationships have semantics.

The fact that entities can only be master or detail introduces a concept of *depth* for entities. This is a relative concept, which allows us to set up a possible classification of entities (figure 6). Entities that are detail of no other entity are *kernel* (or *basic*) entities. Entities that are detail of only one master are *characteristic* entities (they express a characteristic aspect of that master). Entities that are detail of many masters are *associations* (or bridges, or junctions): they can express that the masters are associated via the common detail.

One of the important aspects is that relationships are manageable objects as well. In particular, just like entities, relationships have occurrences (see figure 3). In Bachman modelling, an occurrence of a relationship is defined as one occurrence of the owner with all its member occurrences. Thus, a relationship has as many occurrences as there are owner occurrences and even owner occurrences that connect to no member occurrences are the owner of a relationship occurrence, a so-called empty one. The fact that there are two sets of occurrences involved has led some authors to call *base set* the set of occurrences of an entity and *fan set* the set of member occurrences in one occurrence of the relationship.

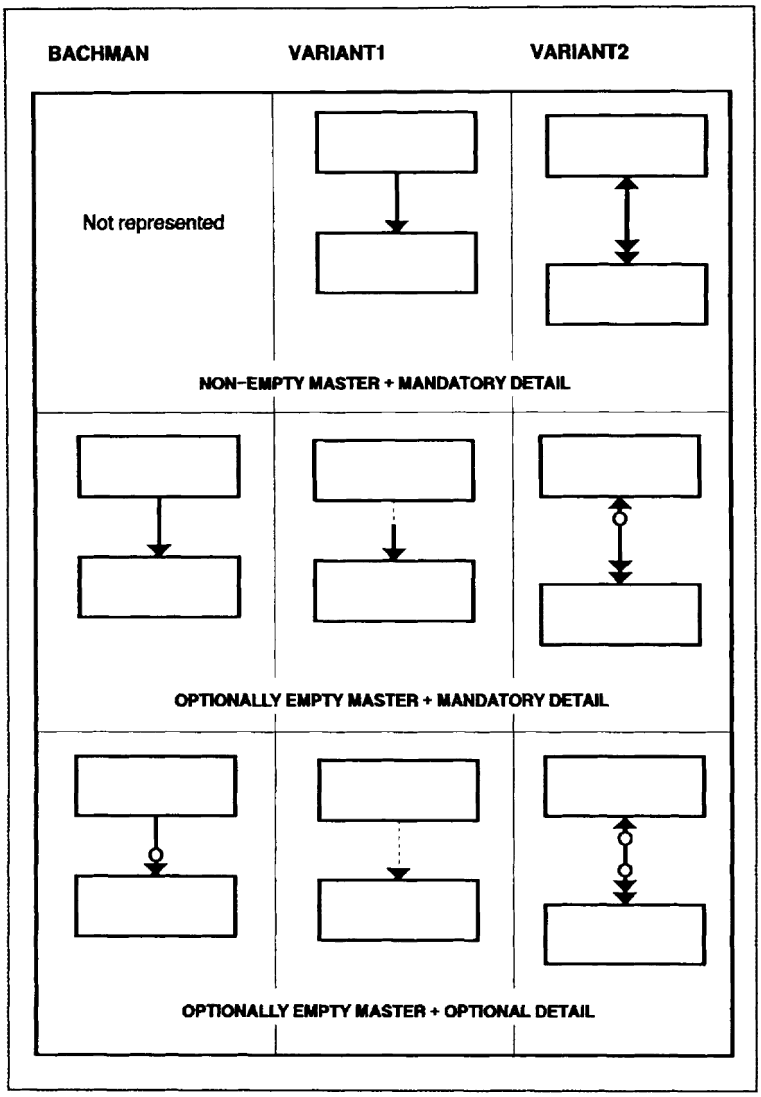


figure 7

All authors now agree that the relationships express the semantics of data; this statement means that we must indicate important constraints (rules) on the relationship. In order to do so, we must observe that entities are abstractions which stand for a set of similar occurrences. In 1:n Bachman diagrams, the semantics of the relationships are the constraints that apply to the detail entities. As far as these constraints go, the model (and diagram) expresses how "strongly" detail links to its master. We will say that a relationship is *mandatory* if an occurrence of detail is always linked to its owner on this relationship; in a true master/detail situation, this is the only possibility. However, it became soon necessary to allow weaker relationships: for instance in a 1:n binding of family head and child, what about orphans and found children? These are children without master. If we allow such situations, then the relationship is *optional*. The notation for these cases is given in figure 7. The situations where we have associations (junctions) cause other deviations to the strict master/detail paradigm[6]. Junction entities

6. Not unexpectedly: as examples will demonstrate, there is an important semantic difference between m:n relationships and master/detail situations.

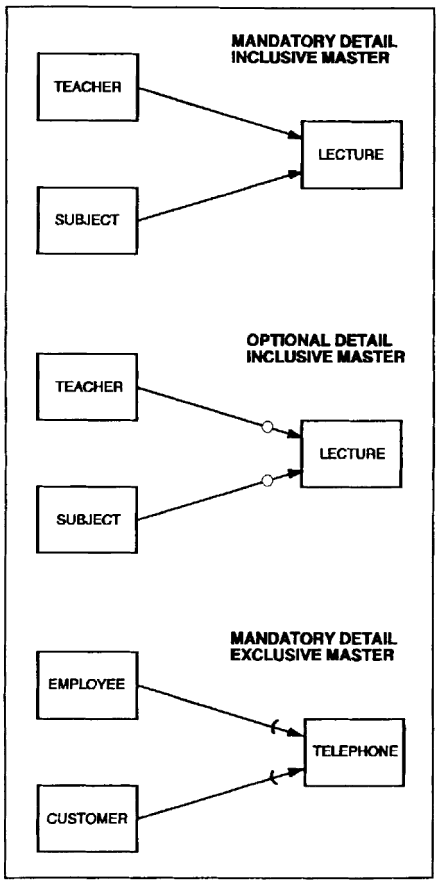


figure 8

a teacher and concerns a subject;
what we have here is a complex situation linking three entities; the two relationships ***taken together*** link a *mandatory detail entity* to its two masters in an *inclusive* way;

- in the same situation, suppose classes can be planned for subjects without the teachers having been assigned yet: the teacher-class relationship is optional, the subject-class relationship is mandatory; again, we have a three entity structure and the two relationships taken together link a (partially) *optional detail* entity to its two masters in an *inclusive* way;

are in fact detail of more than one master. In this case however, because the association expresses a m:n link between the masters, there is something particular about the common detail: it is mandatory in all the relationships. Indeed, an occurrence of a junction plays the role of a bridge between the masters and therefore it needs to be linked to a master occurrence at one end and another one at the other end. A dangling bridge is not acceptable.

Apart from such associations, an entity may be detail of two masters (or more) without being a junction[7]. In that general case we may have some combinatorial questions: does an occurrence of the detail always link to both masters, sometimes both masters, at least one master, just one master (either one), at most one master? Clearly, these situations are semantically quite different. Let us consider some examples (figure 8):

- teachers and subjects are the masters of the lecture entity; a lecture is conducted by

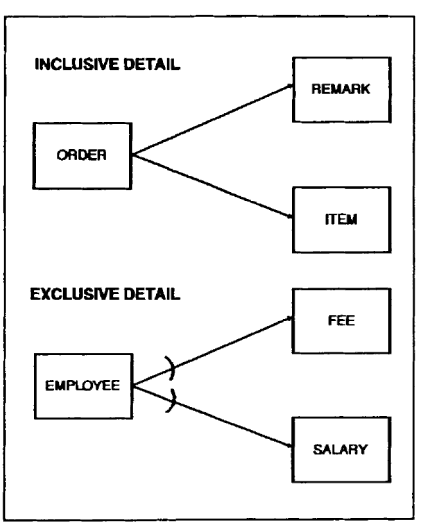


figure 9

7. Bachman modelling does not distinguish these cases; on the other hand, there can be a hidden m:n situation anyhow in many cases of common detail.

- the employee entity and the customer entity have a common detail: telephone; however, a telephone number is either of an employee or of a customer and is never of no one; once more a three entity situation in which the two relationships taken together link a *mandatory detail* entity to its two owners in an *exclusive* way.

As can be seen, these are rather important semantic aspects of relationships which even tend to regroup more than one relationship in a kind of super-relationship. What we have seen for a common detail can also be said about a common master. Suppose indeed that we have a master with two detail entities: we may wonder whether a master occurrence has occurrences of both detail entities or only of one detail entity linked to it. For instance, an employee in a company receives either a salary (if he is under contract) or a fee (if he is acting in a free-lance capacity) but not both. This is a three entity structure (one owner and two detail entities) where the two relationships link two *exclusive detail* entities to a single master. Similarly we can have an *inclusive detail* situation, where the master has always occurrences of both detail entities (see figure 9).

In pure Bachman diagramming, one ends up with a diagram where no link has a higher cardinality than 1:n. And this allows us to structure the diagram hierarchically. Indeed, we will consider that a detail entity is of a deeper (i.e. less important) level than the entity that sends out the link. Of course, there are conflict situations, due to our diagram being a network and not a tree: an entity can be detail of more than one master entity. These ties must be solved according to significance: one should choose the depth of an entity according to the more relevant or the more *natural* link. One rule of thumb here: when an entity is detail in more than one relationship, then those relationships that have the entity as mandatory detail are more significant than those that have it as optional detail, because the notion of detail belonging to an owner is much more strong in the mandatory link. Obviously, the notion of depth is a relative one, but it allows us to order entities by relative depth and this yields a nice readable structure, as the example in figure 10 suggests.

Bachman diagrams are easy to draw structures that contain much information. They are an instance of the saying *a picture is worth a thousand words*. Indeed, an entity diagram as in figure 10 is very comprehensible and can be used by various people for various discussions. In brief, it is a valuable representation. It has shortcomings, however. In particular, it ignores the m:n relationship and the nests. Many authors have suggested extensions to the diagram icons so as to incorporate more semantics. Not surprisingly, the most common extension is the m:n relationship. This is usually represented by means of a line with an arrow at both ends. Similarly, the 1:1 relationship is represented by a line without arrows. Another extension regards the optionality of the master in a relationship. Indeed, in our examples above, we have only allowed for optional detail: a detail occurrence can live without being linked to a master. Conversely, we may want to express the fact that a master must have at least one detail occurrence (such as: an order has at least one item); this is a *non-empty master*. If this constraint does not exist, we will speak of an *optionally*

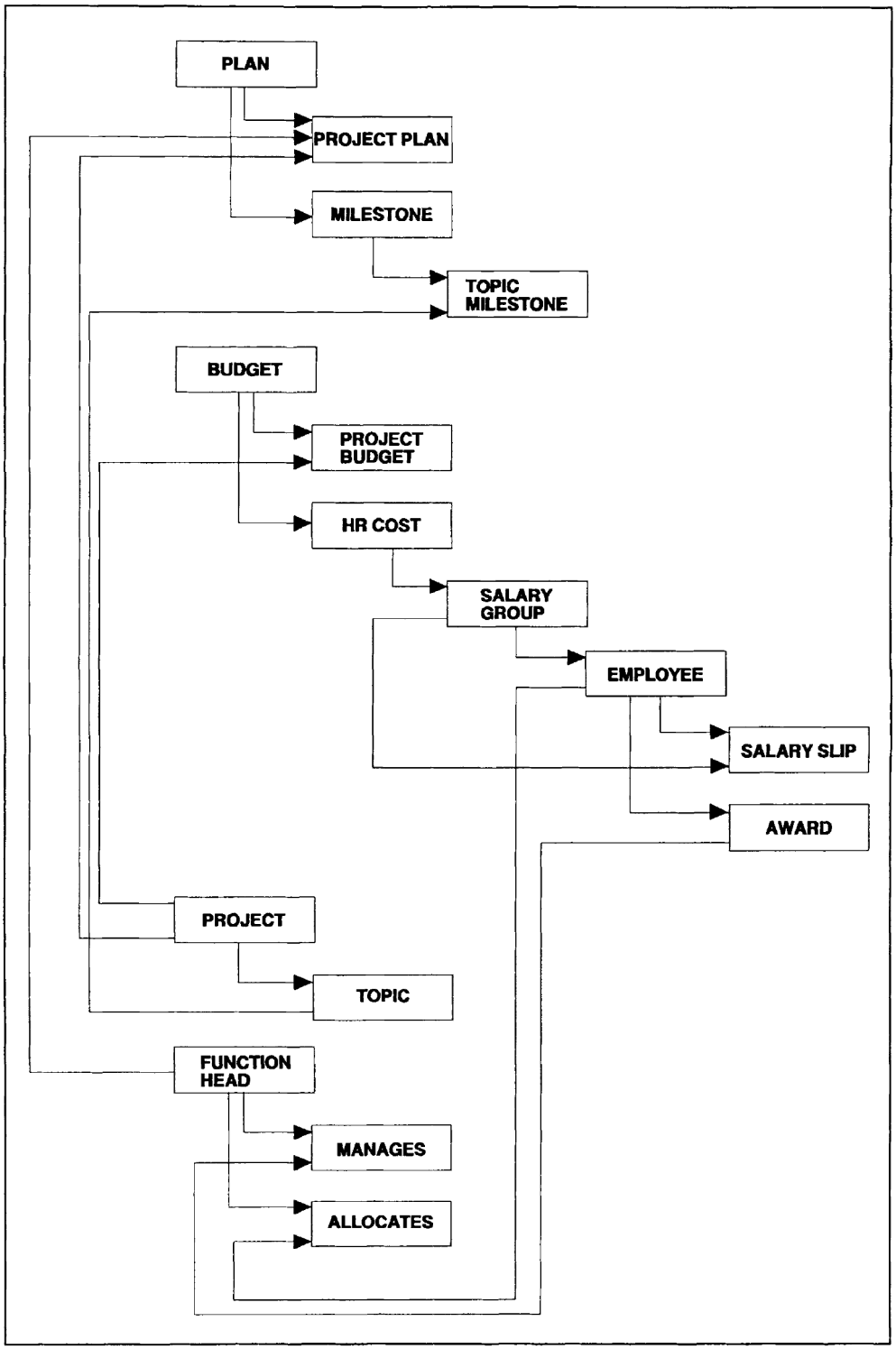


figure 10

empty master. The icons used to represent such relationships are different for different authors (figure 7)

Bachman example: Order Entry

The time has come for an example. It will be a simple one, as it is intended only for illustrating the concepts. Let us try and represent the information model underlying a rather straightforward Order Entry system. In a very informal way, the business rules are the following:

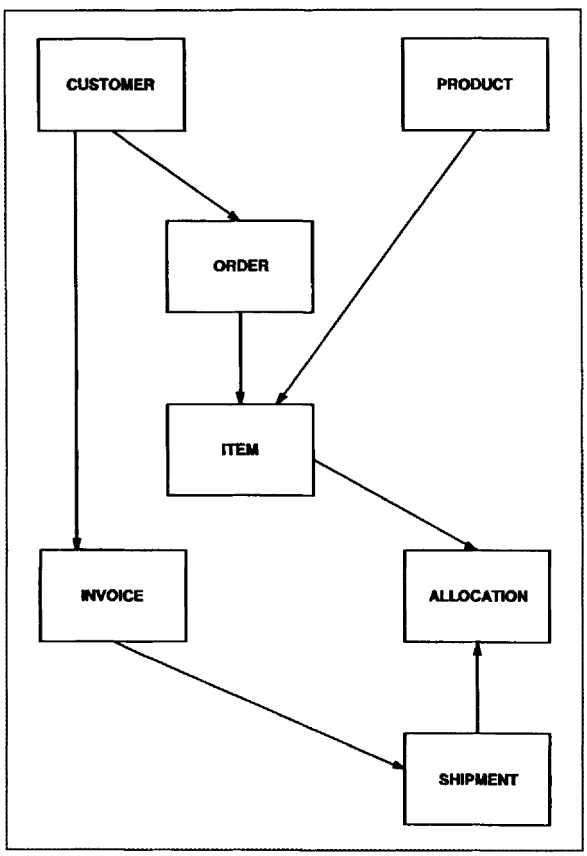


figure 11

* The company is in relation with a number of customers.
* Customers can enter orders for any quantities of any of the products that the company sells.
* Orders are stored for subsequent processing. The processing implies that allocations are made. An allocation actually allocates a certain quantity of an ordered product to a given order.
* Bundles of allocated products will be shipped to the ordering customer.
* Invoices are made to a customer by regrouping delivered shipments.

A rather obvious Bachman diagram results from these rules. It is left to the reader to understand the semantics of the diagram (figure 11).

Subject data-bases go corporate

Entity diagrams à la Bachman can be used to find the list of subject data-bases that are needed to represent the corporate information model.

What indeed is the definition of a Subject Data-Base? The following is a fair try: an SDB is the agglomeration of all data that is strongly related to an organizational subject. Notice the adjective *strongly*. One thing is left open: what is a *subject? An*

BOX 4: OVERVIEW OF THE SUBJECT DATA-BASE ARCHITECTURE

When data-bases came about, they were acclaimed as the long expected solutions to all the problems presented by conventional files. Wasn't the cause of the problems the plethoric growth of the collection of files and the consequent uncontrollable decentralization? Therefore, the data-base idea was a simple one: put data together in some kind of super file and make sure that a program uses only that super file. Develop a super access method so to speak. The trick was done. Suddenly, it became fashionable to have a data-base. Every one wanted one. Brave new world!

Some years later, more specifically when DP went on-line and the volume of applications increased by an order of magnitude, the truth hit home: data-base technology did not, indeed could not, keep its promises. A sad fact.

So what went wrong? Symptoms existed a-plenty; huge maintenance effort (corrective maintenance) was needed. Data-bases were complicated beings which needed a court of specialists to keep them purring: the data-base administrators. And data-base administrators became the most heavily stressed professional category. Most users complained about performance (in terms of response time) if not quality. A general idea pervaded the field; data-base is a necessary evil. One needs it, otherwise complex data relationships are not representable, but the data-base will not be very reliable. A fact one appears to be prepared to live with. This failure is very similar to that of programming techniques. And it has the same cause: a total lack of (or at least severe shortcomings during) conceptualization of the data (analogous to program analysis). Data analysis and design is done (if at all!) in a very intuitive way, without any investigation of meaning. We still create our data-bases as we create second generation files, with the left-hand so to speak. Data is placed in a data-base according to the needs of the programs. Such a data-base has no chance of remaining consistent, in the long run.

At the other end of the spectrum, one finds the normalization fanatics. They will create data-bases with incredible refinement: a great number of records of very small size with numerous foreign keys or sets linking them. This approach is similar to that of the Cobol PERFORM fanatic: he sets up programs that use only PERFORM, with the result that many paragraphs consist of only one statement.

Even if the analysis and design of the data-base has been performed with love and care, the result will still be doubtful. And why is that? Because of something nasty that also affects programs: aging. A data-base lives and therefore interacts with an evolving reality. But can it cope with this evolution? Obviously, only if the evolution has been foreseen. Doing so is an awkward task to say the least. It calls for immense skills on the part of the data analyst to *feel* the stability of data-base items.

Apparently small changes at the logical level may have unexpected effects: the changes in the data-base will be dramatic, and they will cause deep changes in the programs as well (this is so unless the programs have been made independent from the data base structure, something that is possible only partially by using a view-based technology). The major question is: how can one protect a data-base against such mishaps? It is sad but there is no way to cope... And it doesn't matter whether you have a relational data-base or a network data-base.

There is another cause of the misery as well: in a number of cases, program maintenance will adversely impact the data-base. Some programmers will add *technical* fields to the data-base, *flags* as it were. I tend to call this utter brutality. But there is a domain where it is less easy to dismiss such *improvements*: performance and tuning. The same old story. One will sacrifice data-base structures to the performance goddess. This will result in data redundancy and structural redundancy. Causing update problems, of course, which can only be solved by program discipline . In the long run, however, such discipline will not be upheld. The usual compromise that turns sour sooner or later.

As if this were all! Data is not free to live its own life; data is constrained in many ways by the environmental business rules. Although there is no real way to include such constraints into the data-bases of today, one usually tries to represent as many constraints as possible in the relational (static) structure of the data-base. The remaining ones are left to program discipline (the alternative solution is by using some rule-like clauses in an underlying technical view; unfortunately, however, this possibility is NOT offered by SQL). Some manufacturers offer a technology of more evolved relational *views*, but relational fanatics deny them the label "relational". Isn't it an amazing world? Now there are new causes for worry: one problem is the programs breaching the discipline, as may be expected. But there is a more subtle danger: constraints that were frozen into the data-base structure

OVERVIEW OF THE SUBJECT DATA-BASE ARCHITECTURE (continued)

may have to evolve, with the immediate effect of invalidating the structures. The data analyst must investigate the acceptability of any constraint: why does the constraint exist, what does it express, is it arbitrary?

In brief: data-bases are not the solution to our problems. Or, maybe, our way to use data-bases is not the right one?

Let us thus stop and ponder: what is a data-base? A first answer we could try is: *it is the structured set of all data needed by an application* (which leaves us with the need to define the concept of *application*, but I will leave that to intuition: it does not greatly matter in this context). Now, if we have, say, three applications: a payroll, a purchase system and an invoicing system, we would have three similarly denominated data-bases (one for each application). That sounds fine. But is it? Let us see:

- the invoicing system will work with customers, products, orders, services, expenses and accounts receivable;
- the purchase system will work with products (which are bought in to be resold), back-orders, accounts-payable, suppliers;
- the payroll works with employees, salaries, expenses (made by technicians to serve customers), accounts, ...

As can be seen there are *connections* between the three data-bases; they are expressed by overlapping items: products are used for invoicing and for purchasing. Expenses are considered in payroll and in invoicing. Such connections can be very involved because of their implicit nature. In fact, we have a situation where the apparently clean 1:1 link between programs and data-bases is corrupted by the spaghetti bowl of uncontrolled links the data-bases have between each other. These (unforeseen but so predictable) implicit connections are realized usually by additional programs that perform upload-download operations, possibly using intermediate files. These programs are tricky, ill-conceived, unstable, to say the least. They operate with a fragment of reality, according to unverified assumptions. More dramatically, many such programs are developed at moments of catastrophe, as Murphy's law commands. As a result, chaos will slowly but unavoidably result. And what if the data structures of one data-base evolve? Shouldn't such an evolution be considered against all other data-bases? Pathetic question, expressing all the drama of this particular approach to data-base philosophy, an approach that we will call the *application data-base*.

When the need is high, solace is near. People anonymous came up with the solution: since the problem was due to the redundancy caused by separate data-bases it sufficed to banish the mere idea of separation. *Put all data (and its structures) together in one single data-base*. Furthermore, design this data-base as if there were only one gigantic data processing application using it, the union of all applications. Behold the *Corporate Data-Base!* Clearly, such a data-base would be perfect. It would be complete. It would be stable.

Or would it? This corporate data-base approach was doomed from the start: its mere volume and complexity made it infeasible. One would be busy forever constructing the data-base. And supposing one could achieve the goal at all, it would take so much time to reach it that the result would be obsolete anyhow. Utopian thinking! A corporate data-base can't be made.

Still, the idea is nice enough. Question: isn't there a way to realize a corporate data-base *stepwise*? And each step should of course be productive, i.e. allow a sizeable application to be exploited... A *modular* data-base design, as it were. And wonder above wonder, it is feasible. The idea was brought by James Martin's *Subject Data-Bases (SDB)*. A corporate data-base can be defined as a network of loosely connected subject data-bases, the net being realized incrementally. Easy to say, but not so easy to do.

The major question is: what is a subject data-base? Part of the answer is given by the fact that it is a portion of a corporate data-base. If we remember that the corporate data-base holds the data and structures that represent the information the whole organization works with, then we can infer that a subject data-base will hold a part of this information. The partitioning should however be done *from the top*, that is with the organization in mind (and ***not*** any particular application). In other words, a subject data-base is the repository of all information regarding one high-level organizational subject. As an example, the subject *customers* will do: this is clearly a high level concept, of which detail informa-

OVERVIEW OF THE SUBJECT DATA-BASE ARCHITECTURE (continued)

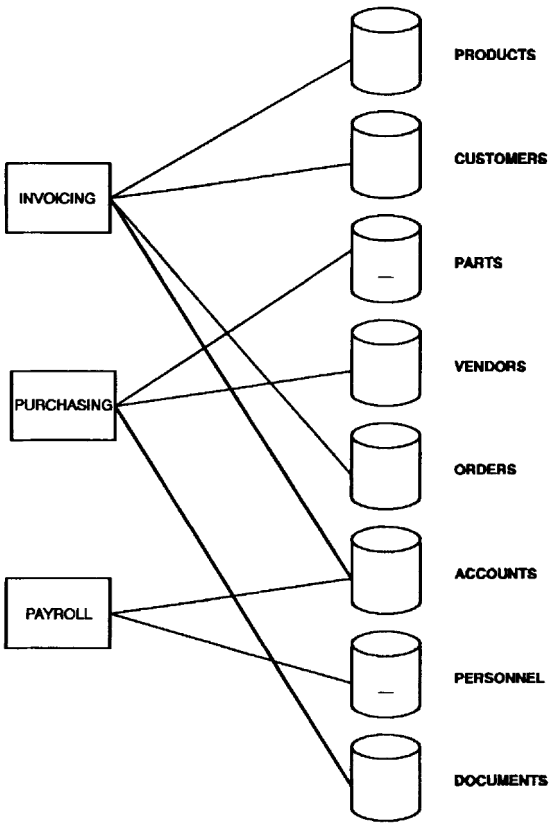


figure 12

tion is used throughout the company, in many applications. These applications do not even all belong to the same department.

The essential idea is that each subject data-base can be developed on its own, in total ignorance of the weak links between the data-bases. It will of course need to be developed by analyzing the use within the organization of each of the contained items, and this should be investigated throughout the company, not only the application on hand. Next, when one sets up programs, it is clear that a given program will have to access more than one subject data-base. Consider the same applications as before, but with subject data-bases for each of the following subjects: products, customers, parts, vendors, orders, accounts, personnel and documents. We then obtain the architecture of figure 12. The most visible effect is that the access links are now between the programs and the data-bases and therefore they are part of program design and development. They can be carefully conceived, instead of having them appear unexpectedly during program maintenance cycles.

Of course, nothing prevents a particular data-base administrator from putting some (or all) of our subject data-bases into only one physically implemented data-base.

It all sounds marvellous, doesn't it? But what is the drawback? What is the price? Since one receives nothing for nothing... Well: there is more work involved. One must first represent the company as a network of Subject Data-Bases. Next one must plan which subject data-bases to develop. And this means investigating *all* departments of the company that use data in that particular subject data-base. This kind of analysis may very well get stuck in conflicts of interest and other organizational unpleasantness... And there can be no compromise: doing a sloppy Subject Data-Base analysis leads to results that are much worse than those obtained by a sloppy application data-base design... The fortunate effect, however, is that after very few implemented systems, all Subject Data-bases have been created, so that there is no supplementary effort to be invested any more. Therefore, there is a high initial cost but in the medium term, there can be a sizeable return on investment.

At a more technical level, subject data-bases, because of the modular approach, cause the same difficulties as those that were encountered, years before, in bare modular programming: the difficulty of defining relevant portions to be implemented. There are no miracles...

organizational subject is the principal entity in a set of strongly related entities. A somewhat circular definition maybe, but it will serve our purpose.

We can start from a Bachman entity diagram. We also want to know how *strong* a link is. In other words, how important it is for us that two entities are linked or not. Now this strength can be expressed as a ratio between the number of times that the entities are used together and the total number of times that they are used at all (over some significant interval), or any other such measure of some significance. This is sometimes called *entity affinity*. Suppose we order this affinity on a scale from 1 to 5 points. Let us do it on our example diagram of figure 10, and we redraw the diagram once more, regrouping those entities that have a high strength relationship (defining high strength, as, for instance, higher than 2). The diagram becomes as in figure 13. So now we see *groups* appearing, containing only strongly related data. And if the groups are connected at all it is by weak links only. Such groups are in fact Subject Data-Bases. What is the rationale behind it? Each subject data-base can be developed in its own, in total ignorance of the weak links. It will of course need to be developed by analyzing the use of each of the entities it contains, and this should be investigated throughout the company, not only the application on hand. Next, when one sets up programs, it is clear that a program will have to access more than one subject data-base (indeed, consider figure 12, which indicates that the access links between the applications and the (subject) data-bases are now part of program design and development). These links, because they are visible, can be carefully conceived: they will not appear unexpectedly during program maintenance cycles. And of course, nothing prevents us from putting some (or all) of our SDB's into only one physically implemented data-base. It all sounds marvellous, doesn't it? But what is the drawback? What is the price? Since one receives nothing for nothing... For one thing, there is more work involved. One must first represent the company as a network of Subject Data-Bases. Next one must plan which SDB's to develop. And this calls for investigating *all* departments of the company that somehow use entities (data) in that particular SDB. This kind of analysis may very well get stuck in conflicts of interest and other organizational unpleasantness... And there can be no compromise: doing a sloppy Subject Data-Base analysis leads to results that are much worse than those obtained by a sloppy application data-base design... In fact, subject data-bases, because of their modularity, have the same difficulties as those that were encountered in *modular programming*. There are no miracles...

More semantics: Chen modelling

Bachman entity diagrams had stressed the relationship as an important component of an information model. They were restrictive in that they only accepted the master/detail paradigm. As a result, m:n relationships were largely ignored (and replaced by the ill-famed junction entity). Peter Chen extended and formalized the concept of relationship even more. He created the entity-relationship modelling (E/R) with the intention of capturing more semantics, especially in the relation-

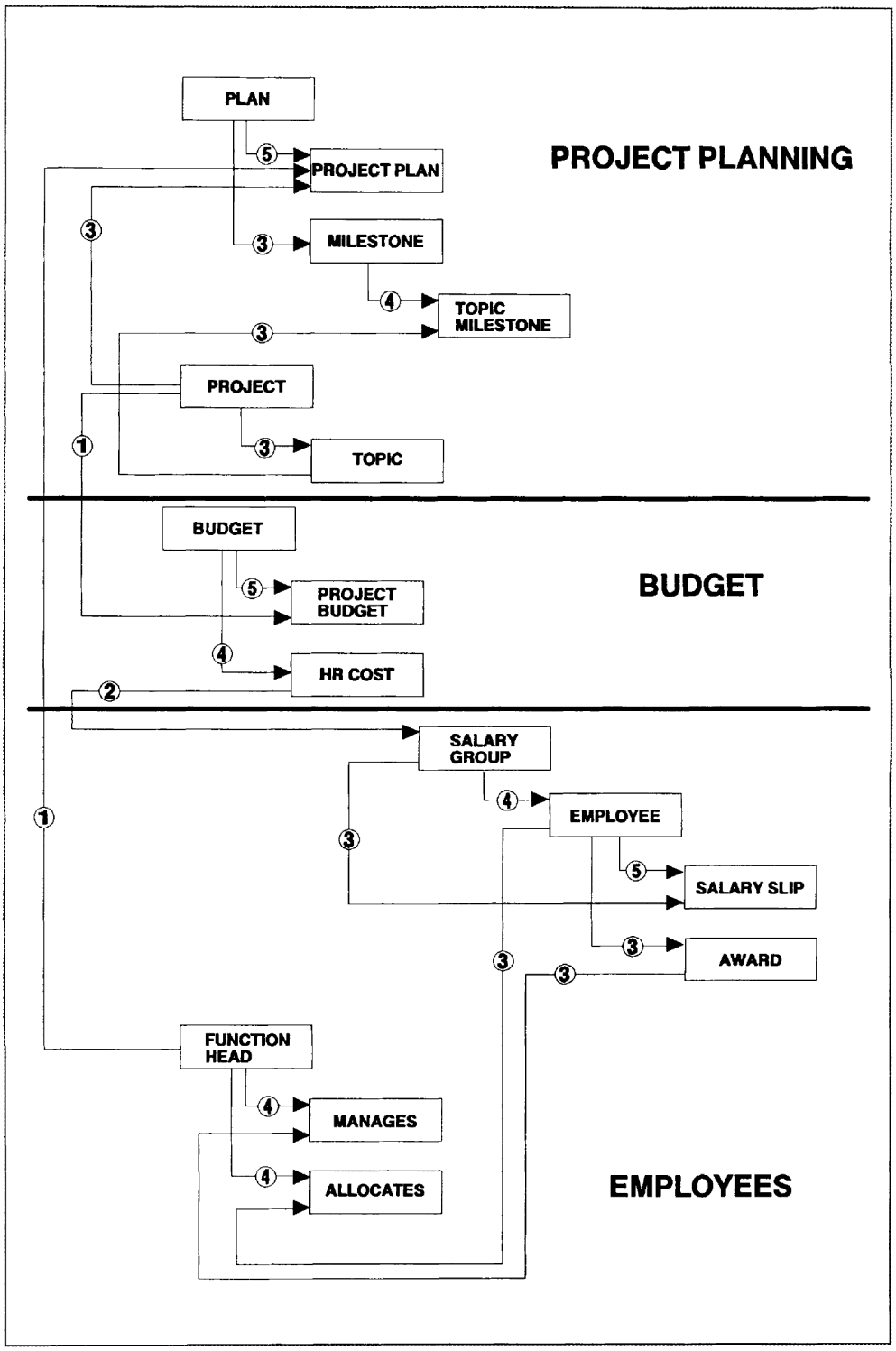


figure 13

ships. Basically, in E/R the building stones are those of Bachman: entities and relationships plus the constituents of entities: *attributes*. Attributes are not especially important in semantic modelling, so we will devote attention to them at a later stage. Nothing more is to be said about the entities. However, the relationships deserve some more attention. The relationship is now considered an association in all cases. The master/detail paradigm is abandoned and all relationships are (or at least can be, can evolve to) m:n relationships. The two entities that are linked have an existence of their own which is not a priori conditioned by the relationship. There is no notion of depth for entities. Although in this the Chen model is a little bit too general for my taste[8], the model introduces powerful new possibilities. First of all the icon for the relationship has become more "visible": the simple line was replaced by a lozenge connected to the two entities that the relationship links. Moreover, the orientation implied in a Bachman relationship (1:n from master to detail) was dropped: because it is an association, a relationship links two entities, period. As a result, the *meaning* of the relationship can now be brought into the model: the relationship appears as an object in its own right, because of the lozenge, and it must have a name (figure 14).

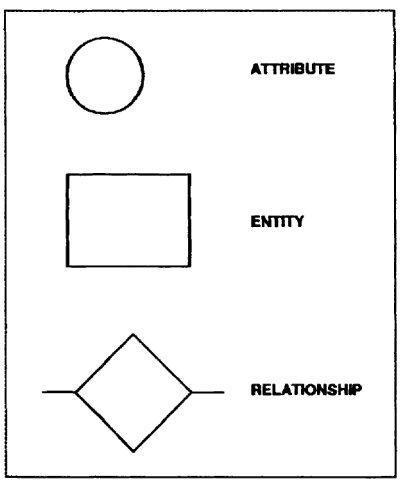


figure 14

The first consequence of the enhanced view was that the cardinality of relationships was defined differently. The cardinality of a Chen relationship is a two-valued item: if the relationship links two entities A and B, then we consider occurrences of the relationship as occurrences of the combination A+B and the cardinality states how many times an occurrence of A appears in occurrences of the relationship, along with how many times an occurrence of B appears in occurrences of the relationship. This cardinality is represented by means of a simple (one occurrence) or a double (many occurrences) arrow going from the entity to the lozenge (figure 15). The representation is definitely richer than that of Bachman: not only is 1:1 explicitly representable, but also m:n is now a valid case. The question is: don't we take a risk in allowing the m:n relationship? After all, if Bachman said that it had to be replaced by a junction entity, there was probably a good reason for this. The reason is physical: the junction entity, implemented as a record or table, allows us to store the data that a m:n relationship will very often require (e.g. the price of a product which is different for the various suppliers). In fact, at the semantical level, this data is data associated with the

8. I like to represent true master/detail as well as associations, because they are different in their behaviour.

relationship; the junction entity is an artificial physical solution. In E/R modeling we allow a relationship to have attributes just as an entity has attributes, in other words, the lozenge is not empty.

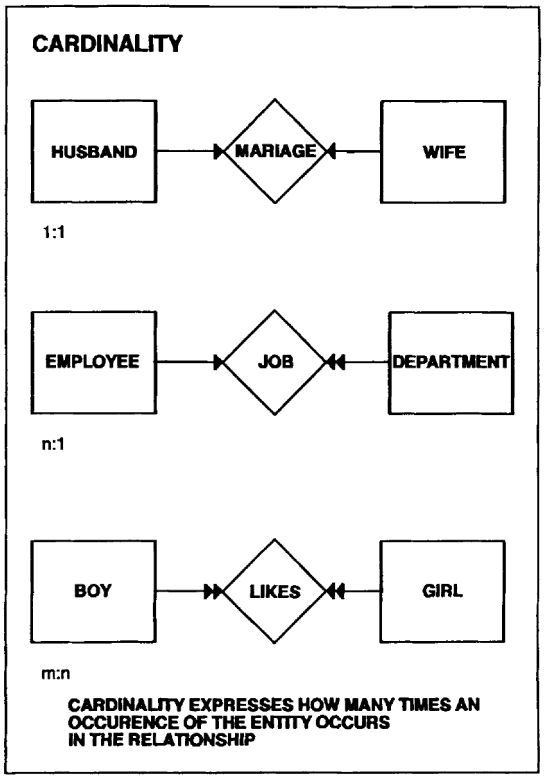


figure 15

How does E/R modeling represent semantics (of the relationships)? First of all, by giving an explicit "active" name to a relationship (in the lozenge), it certainly adds meaning to the relationships. We should observe that relationships are of different types according to this meaning. The classic master-detail is a relationship that expresses the concept of *has-many* (e.g. a journey has steps) with an implication of unambiguous ownership. The m:n relationship, however, usually expresses a much more operational notion: a teacher teaches subjects (and conversely), implying that there is most certainly a program actions involved in the dynamics of the relationship. The term *association* used to designate m:n relationships is very well chosen. There are many 1:n relationships that also have this dynamic aspect; the symptom is that such relationships can very easily become m:n relationships (this is not the case when we are in a true master-detail situation).

Bachman's relationship indicates both a cardinality and a dependency (an orientation, so to speak). E/R has no implication of dependency (orientation) for a relationship. It may however be desirable to represent some kind of *existential dependency* by which we indicate that the occurrences of one entity can only exist if they are related to occurrences of another entity (this is in fact a generalization of the master-detail situation). This is done by inserting an arrow à la Bachman, but without any implication of cardinality. In figure 16 an example is given in which we represent by means of the arrow from product to price that a price can only exist for a product (via an offer), and that a team can only exist when it has employees. In a way this says that a product must exist before one can specify a price and employees must be hired before one can organize teams. An extension to this existential dependency constraint, making it much stronger, is saying two entities are so related that each of them is existentially dependent on the other one: an order can only exist if it has items, an item can only exist if it is on an order. This calls for an arrow at both ends of the relationship. Such a strong existential dependency con-

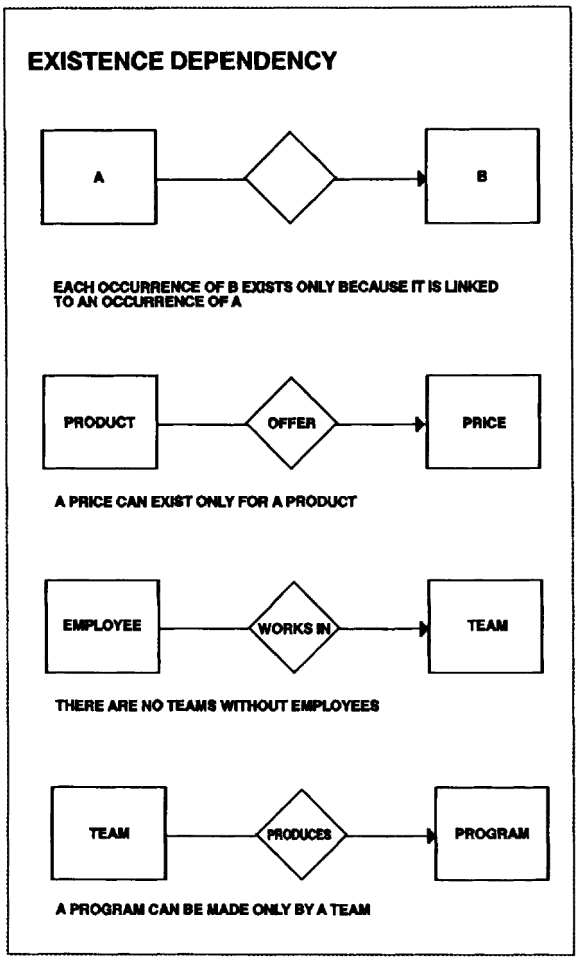


figure 16

straint contains a chicken and egg problem which manifests itself at the moment new occurrences are stored. Indeed, when a new order is to be stored, how should we do this? Storing first the order occurrence will fail because there are as yet no items stored for this order; storing first item occurrences will fail because there is not as yet an order for these items! The answer lies in a temporary (automatic?) relaxation of the constraint, but how temporary is temporary? Or it lies in the definition of a new macroscopic operation that stores orders with items, without splitting them[9].

A concept that comes very close to that of existential dependency is the *participation class* of an entity in a relationship, also called *totality* of an entity. What we want to express here is the quantification of whether *all* or *some* occurrences of an entity occur in the relationship. When all occurrences of an entity occur in a relationship, the relationship is said to be *total* with respect to that entity (or, in short: the entity is total); otherwise it is *partial* (or, in short: the entity is partial). The notation used for the expression of the participation class is a black dot on the side of the total entity (some authors use a bar on the side of the partial entity). Clearly, if an entity has existence dependency, then it is total: all prices are for a product; all teams have employees. The participation class of an entity in E/R models is equivalent to the optionality of a detail (mandatory detail is total) or the emptiness of a master (non empty master is total) in Bachman diagrams. Looking for totality is not always easy: it may be disguised in human speech. For instance: a price can only exist for a product (price is total), there are no prices which are not for a product (price is total), there are no teams without employees (team is total), all teams have em-

9.. None of these options is present in SQL.

ployees (team is total), any team produces at least one program (team is total),... See figure 17 for the graphical representation.

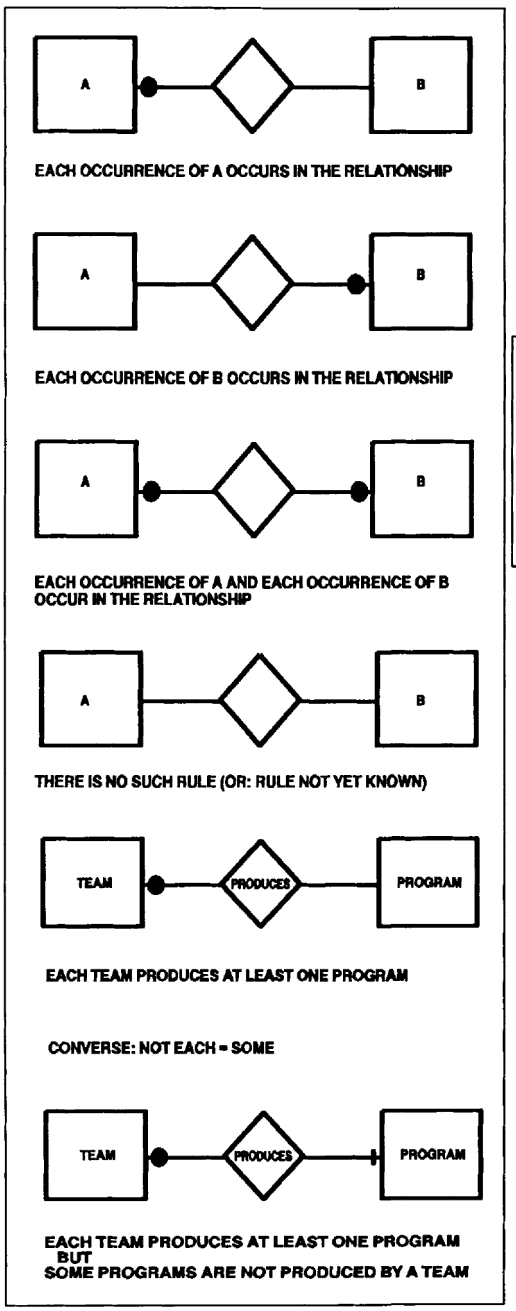


figure 17

Some authors also particularize those cases where the participation of an entity occurrence in a relationship is submitted to conditions. For instance, teams are made of employees, but it is not just any employee who can be member of any team; figure 18 indicates one of the possible notations. The nice thing is that such a case is recognized, but the unfortunate aspect is that it does not go far enough: where indeed is the actual condition (or business rule) represented?

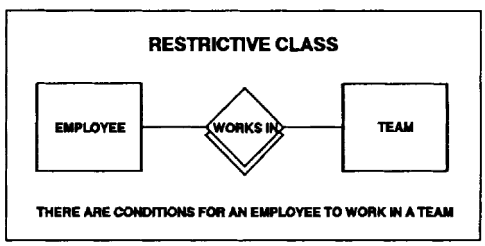


figure 18

E/R example 1: Order Entry

Our order entry diagram (of which the Bachman representation is in figure 11), is now re-drawn using the E/R model. The resulting diagram is given in figure 19. At the current high level of abstraction we work at we have an ORDER-for-PRODUCT relationship which has a m:n cardinality (in Bachman odelling we were forced to create ITEM as a replacement for the m:n relationship). An allocation is about orders (which are split into allocations) and contains products. A shipment groups allocations and an invoice groups shipments.

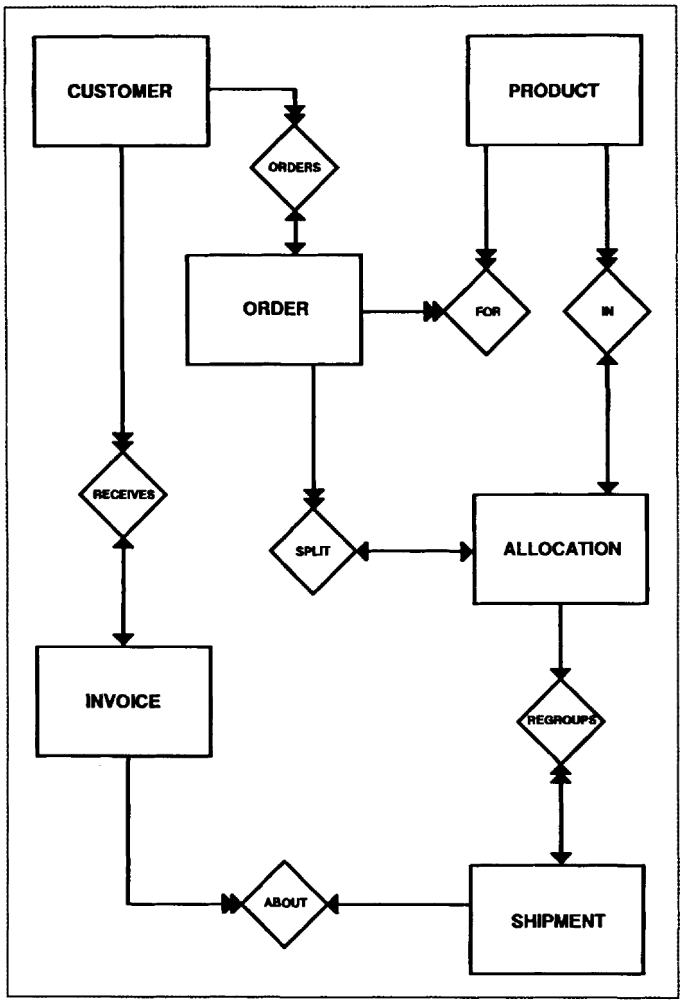


figure 19

The diagram also indicates the dependency (equivalent to totality) constraints.

Higher order extensions

Strangely enough, the E/R model of our order entry misses something that the Bachman model had. In the E/R solution, an allocation can be about anything, even a products not appearing on any order. In the Bachman solution this could not happen because of the link with ITEM. However, we must observe that we are speaking here of links that are not just between entities. In order to model the required semantics, the E/R model has been extended: relationships may link an entity and another relationship. In figure 20 we see the case where one relationship expresses the fact that a teacher teaches a subject and this relationship itself is linked to the students who attend the subject being taught. A relationship may also link two relationships. In figure 21 we find the example of a doctor prescribing only medicine he knows and of a patient having a disease, with the additional information that a doctor prescribes drugs from the medicines he knows, for a patient who has a certain disease.

When we compare Bachman modeling and E/R modeling, the question arises as to which one to prefer. E/R modeling is more recent and more fashionable, and expresses more semantics. The (visual) expression power of E/R models is increased by the lozenge icon, even though a Bachman junction entity also represents a m:n relationship. Bachman modelling is much in favour when the chosen database implementation is a network or a relational one: indeed, neither of these data-

base organizations can represent relationships of m:n cardinality otherwise than with an auxiliary record or table. Of course, a network data-base is very strong in representing 1:n relationships and is therefore much more akin to Bachman diagrams than E/R diagrams. The translation from one representation into the other one is however very straightforward.

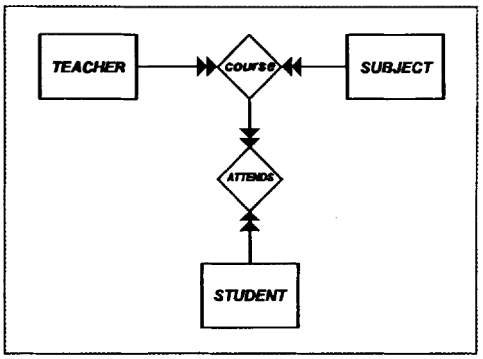


figure 20

E/R example 2: The Transportation Company

Problem statement

The intention of the company is to transport consignments by truck. It works according to the following rules:

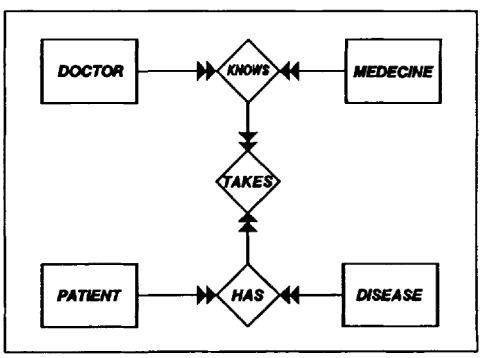


figure 21

- The company will manage itineraries and loads of all trucks and will combine truck loads optimally. It does not own the trucks, which are the property of independent owners.
- Trucks can only transport containers, which contain consignments.
- Trucks perform international journeys (identified by means of a chronological number) and typically will stop at various city warehouses (only one per city) to load or unload containers. In order to perform their job correctly, the chauffeurs receive a journey plan containing the route and the list of containers to load/unload in each city, with a copy of the waybills required. Once a container gets off a truck, the truck continues its journey.
- A journey's itinerary does not pass through the same city more than once.
- Consignments are never delivered C.O.D., so that the truck company will never have to bother with cash flow, nor will it perform any consignee billing as this is left in the hands of independent freight agents.
- Consignment delivery from the warehouse (where the container was unloaded) to the consignee is in the hands of independent (local) delivery companies, with which the transportation company has set up a number of agreements. The delivery date is the date at which the consignment is received by the consignee.
- Delivery companies are not unequivocally linked to the warehouse city.

- In case of journey, truck or consignment problems, the company will be in touch with the freight agents. It will never contact either the consignee nor the sender of the goods.

- In order to have goods transported, a sender has to submit his consignments at de-centralized freight offices (manned by freight agents), from where they are brought to the nearest warehouse.

- A consignment is any grouping of items as defined by the sender. The maximum allowed size of a consignment is that of the container.

- A consignment is described by one waybill.

- The warehouse keeper will stack consignments into containers. Containers may contain more than one consignment. Each container therefore is associated with as many waybills as there are consignments within, describing the composition of the consignment (i.e. the items which compose the consignment).

- A copy of the waybill is given to the sender, one is glued on the container, another will be given to the consignee, yet another will be given to the chauffeur, one is kept by the freight agent, and finally one is transmitted to the local delivery company that will do the final delivery to the consignee.

- The planners will schedule consignments. To that effect, they first determine (or create) a truck journey. Next they determine how to fill the containers (as they can obtain them) with consignments. These containers are then scheduled to the journey of the truck and their load/unload activity in the right city warehouse is determined. Typically, a container going to a certain city will contain goods (consignments) destined to places in the neighbourhood of the warehouse (destination) city. The planners also determine the delivery company that will be used.

- The date at which a container is loaded onto a truck will be considered identical to the date at which the truck leaves the warehouse. Similarly, the unload date of a container is the date at which a truck enters the warehouse.

The Chen model representing our problem statement is in figure 22. It is rather self-explanatory, even though it contains a higher order relationship which links the relationship between containers and steps (of journeys) to the waybills. Note that only one of the two relationships (LOAD and UNLOAD) links to the waybills, since the consignments that are in the container at load time are the same as those that are in it at unload time.

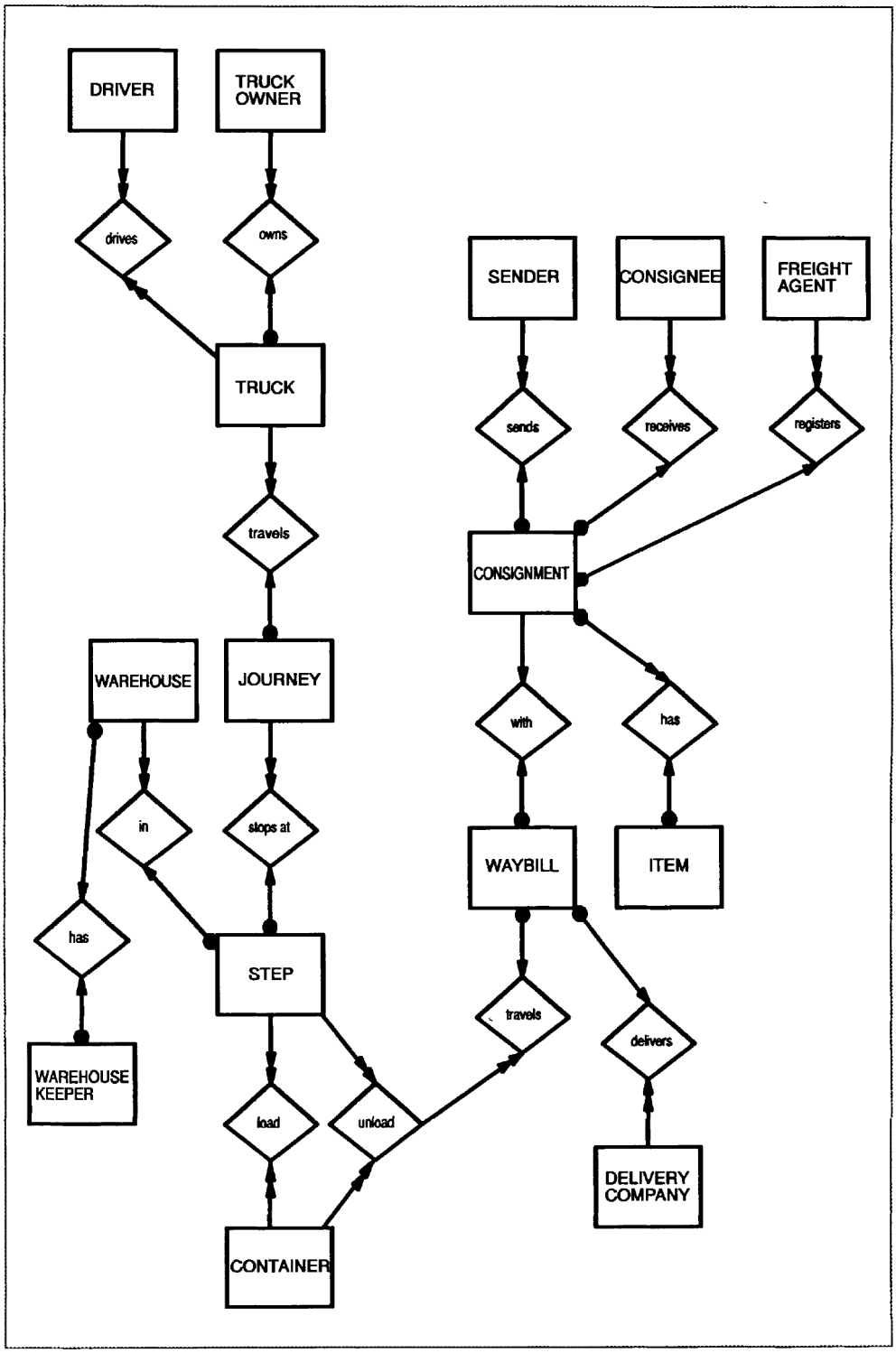


figure 22

Brno" there is a specific non-lexical object customer described by the lexical objects 123 and Brno. By semantic classification a set of similar lexical objects (e.g. all occurrences of an employee name) is called a lexical object type (LOT) and a set of similar non-lexical objects (e.g. all occurrences of employees) is called a non-lexical object type (NOLOT).

NIAM also recognizes relationships, but calls them *facts*[11]. A distinction is made between facts linking entities (NOLOTs) which are called *ideas* and facts linking an entity (NOLOT) and an attribute (LOT) which are called *bridges*. The representation is by means of a double rectangle that is connected to the two objects linked by the fact (figure 26). Interestingly, NIAM makes explicit the fact that a relationship is symmetric: it can be used in one direction as well as in the other one. Therefore, a fact has two *roles* indicated in the rectangles. The left-side role expresses the role of the left-side object in the fact, whereas the right-side role expresses the role of the right-side entity in the fact. The fact that links employees and departments has the two roles: an employee *works-in* a department and a department *occupies* an employee. The representation of facts with roles carries a great degree of semantics, of course. A fact has occurrences which are constituted of the combination of a suitable identifier from each of the linked objects. The set of occurrences is called the population of the fact (figure 27). In a population diagram, we find under the role of an object all occurrences of (a key of) the object that can play this role. For our current example, under the works-in role we find employee references, under the occupies role we find department references.

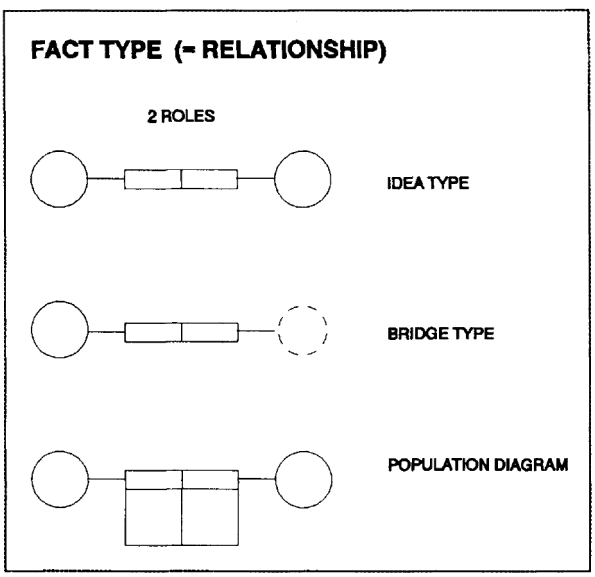


figure 26

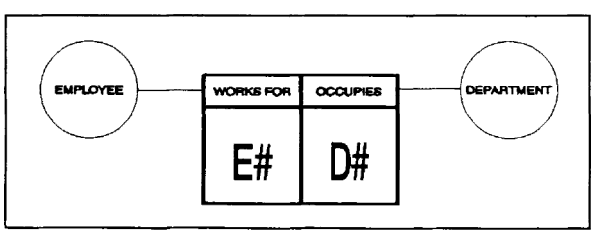


figure 27

In NIAM, just as in all other models, the first concern about facts is the cardinality. NIAM expresses this as rules of

11. More correctly, a relationship is called a *fact type*, whereas an occurrence of the fact type is a *fact*. However, in relaxed speech the word fact represents the two interpretations.

uniqueness associated with the roles of a fact. Saying that a role is unique means that in the population of a role a same reference can only occur once. If a role is not unique, a same reference can occur many times in the population of the role. Uniqueness of a role is indicated by a superscripted flat arrow (figure 28). A unique role states that the entity on that side occurs once in the relationship; a non-unique role states that the entity on that side occurs n times in the relationship. The various combinations of unique roles allow the representation of 1:1, 1:n, n:1 and m:n relationships. Furthermore, a distinction can be made in the m:n cases: cases where each occurrence of the fact is unique (concatenation of roles is unique) and cases where this is not so.

In a way similar to E/R modeling, NIAM also expresses totality. If an object participates totally in a fact, then this is indicated by a capital A on the side of that object (examples are given in figure 29).

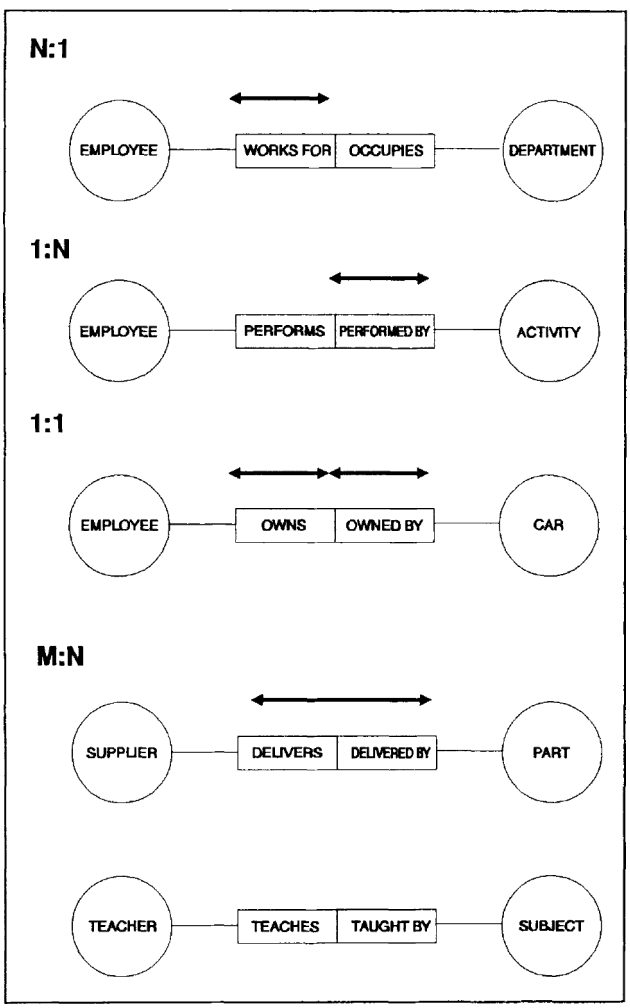


figure 28

The power of NIAM modeling resides in the representation of constraints. We have already seen some constraints (also in other modeling techniques) such as cardinality uniqueness and totality. But the more interesting cases are the constraints that somehow span facts (relationships). Examples will provide the necessary insight in constraint types:

- **Example 1 (role equality)**: employees are in salary groups and employees have a function; obviously, all employees who are in a salary group have a function and vice-versa. The employee populations of the two facts are equal: the same occurrences of employees are in both (figure 30).
- **Example 2 (role implication)**: only employees who have a function can

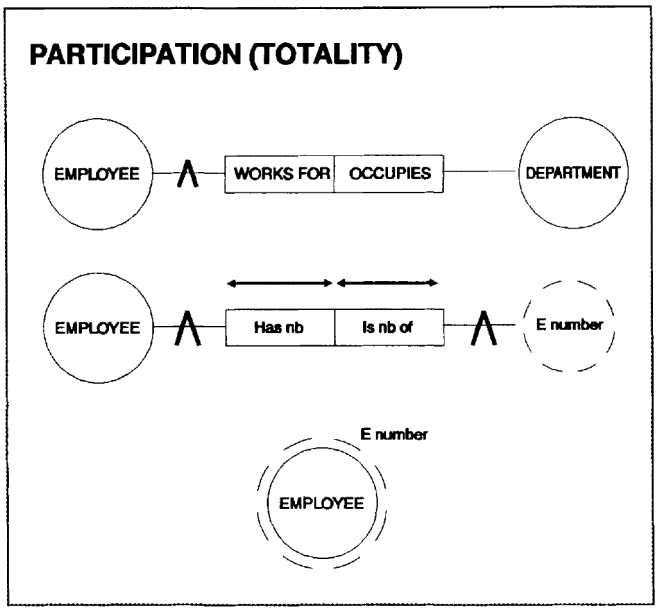


figure 29

be assigned tasks, but not all employees with a function are working on a task. Here the has-task role is a subset of the has-function role of the employees (figure 31). In more boolean terms: the has-task role implies the has-function role.

- **Example 3 (role exclusion)**: a company's employees are either manual workers or intellectual workers, but not both. As a result, they fill different time sheets (figure 32). The two employee populations of the facts are mutually exclusive.

- **Example 4 (fact implication)**: authors write papers. Sometimes they also give a public presentation of some of their papers. There are two relationships between author and paper, one that expresses authorship (we assume there are no joint authorships) and one that expresses the presentation of a paper by its author. The two relationships are dependent by *implication* (or sub-setting)... What we have here is a constraint between occurrences of facts and not only occurrences of roles. Another notation is needed for this case, as indicated in figure 33.

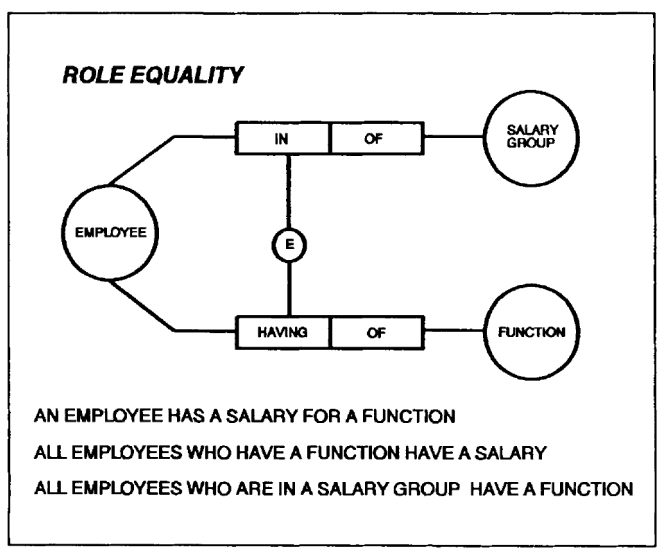


figure 30

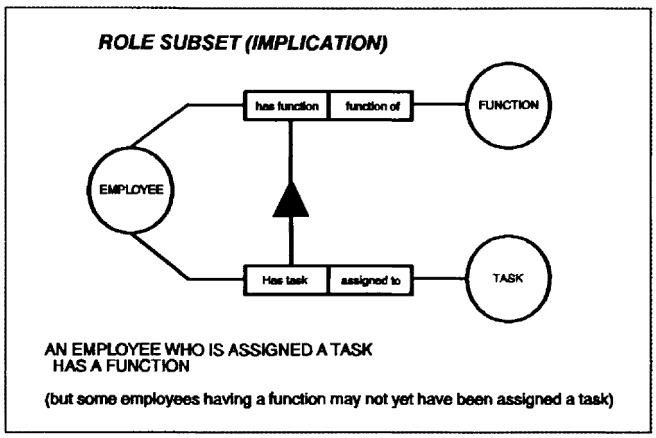


figure 31

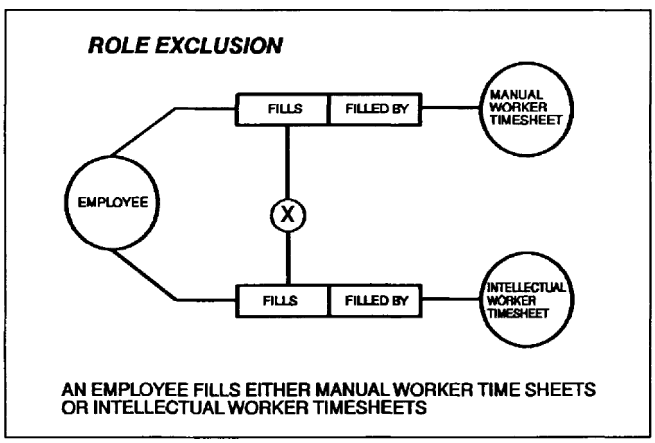


figure 32

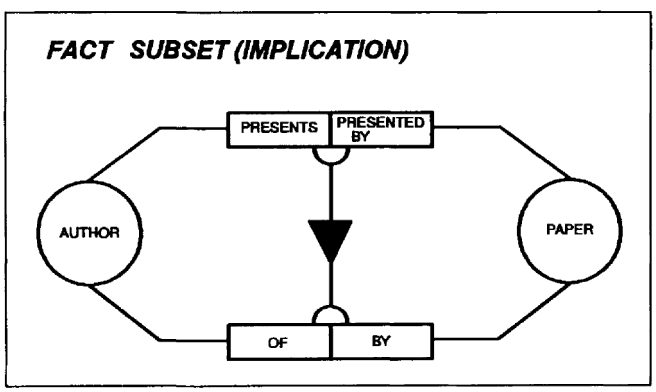


figure 33

- **Example 5 (fact implication):** a department has employees and it has also a secretary. The secretary is a member of the department. This is once more fact implication (see figure 34).
- **Example 6 (fact implication and exclusion):** a school has a number of departments and teachers attached to a department. Some of the teachers actually teach, others are department heads. Thus, there are two relationships between department and teacher. However, per constraint, the following situations can be ruled in or ruled out: *either* the head teachers of a department are always active teachers of the department (implication or subset), *or* the head teachers are never active (exclusion, see figure 35), *or* the head teachers of a department may be active teachers of other departments. Again, these are constraints between the two facts.
- **Example 7 (fact implication and exclusion):** There is a number of facts about warehouses and goods. One fact expresses the catalog of goods that should be in the warehouse. Another fact expresses those goods that are actually in stock. Obviously,

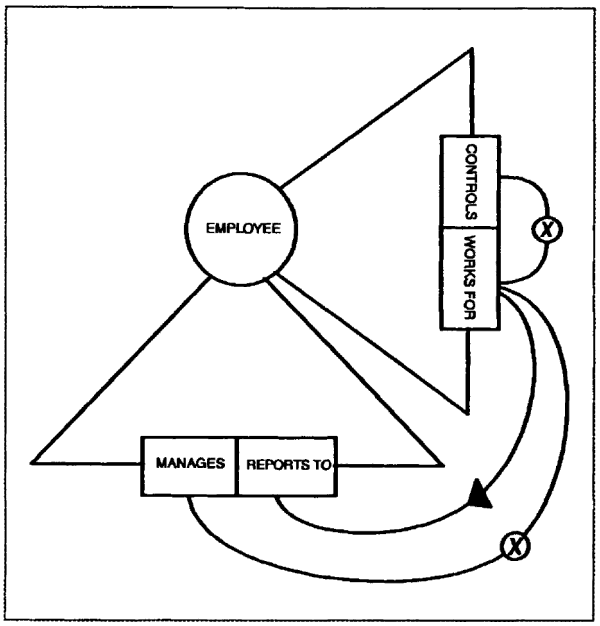


figure 40

Now, employees can work in more than one office (obviously on a scheduled basis), however, in each office they work in they are assigned only one desk. This is a fact about three objects: employee, office and desk. In this fact, the combination of the roles office-for and desk-of is unique (since it is given to only one employee). The representation is given in figure 42 . In pure NIAM such ternary facts are forbidden, so we need an equivalent representation with only binary facts. Our example will need two facts, one linking employees and offices and the other one linking employees and desks. Apparently we lose the constraint that the combination office-for/desk-of is unique. This is restored by indicating a uniqueness constraint upon combined roles, as indicated in figure 43[12]. The resulting diagram is much less readable, once more proving that higher order facts are more than welcome.

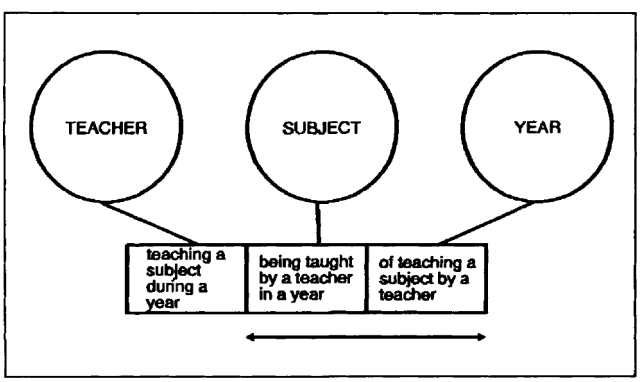


figure 41

Another, rather natural, example of role combination uniqueness is to be found in figure 44.

A last interesting aspect is that NIAM also allows unary facts. If we want to express the fact that a given employee is a salesman, this is indeed a fact about an employee, but it links to nothing since there is no object salesman. Sa-

12. In the strict formulation of NIAM, only binary facts are allowed; higher-order facts must be decomposed into binary equivalents. This strict discipline may be acceptable when the modeling is refined into an actual data-base, but for representing semantics of information all types of facts are to be accepted.

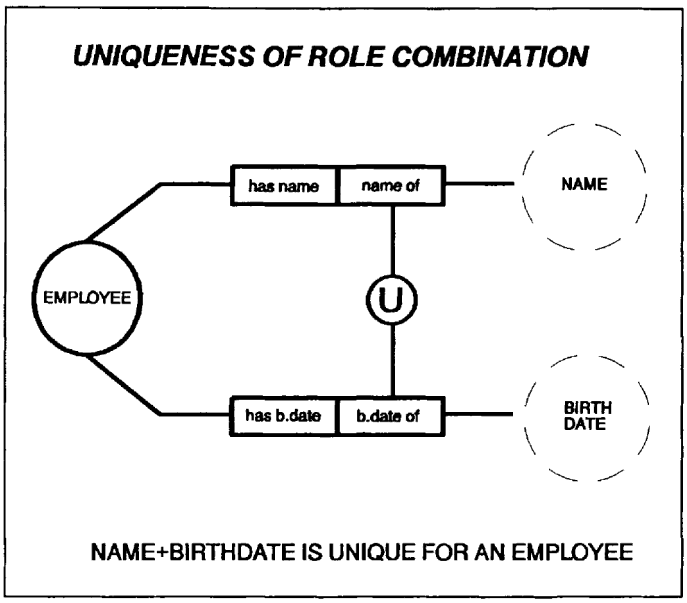


figure 44

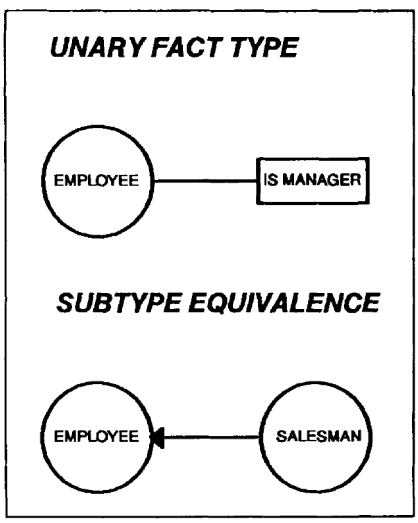


figure 45

– or we create a new object, the office-desk which relates to office on one hand and desk on the other, but which specifically has a fact with employee (this resembles the junction entity in Bachman diagramming), see figure 48.

Let me conclude: business rules expressed as constraints abound in companies. They may be reasonably complicated, but they are definitely part of the data structure, and must be defined as early as possible, and as precisely as possible. Now, since no data-base system is yet able to incorporate such constraints (or allows it to a very limited extent only), most analysts do not represent them. This is a serious mistake, which increases the risk of data corruption. The graphical syntax of NIAM is certainly a good way to represent constraints, but usage of it very quickly leads to overwhelmingly complicated diagrams. Unfortunate but unavoidable.

The question of how to install constraints in a physical data-base is intriguing. Ideally, a data-base definition language should allow such installation and possibly generate some black-box code that would be executed automatically as soon as a program touches the constrained field or relation. There are some data-base manufacturers who implement features that come close (rules and triggers linked with the data definitions)[14].

14. Incidentally, the problems of updating relational views could also be expressed by constraints governing the (conditional) back-propagation of an updated view tuple to the source tuples of the view's constituent tables.

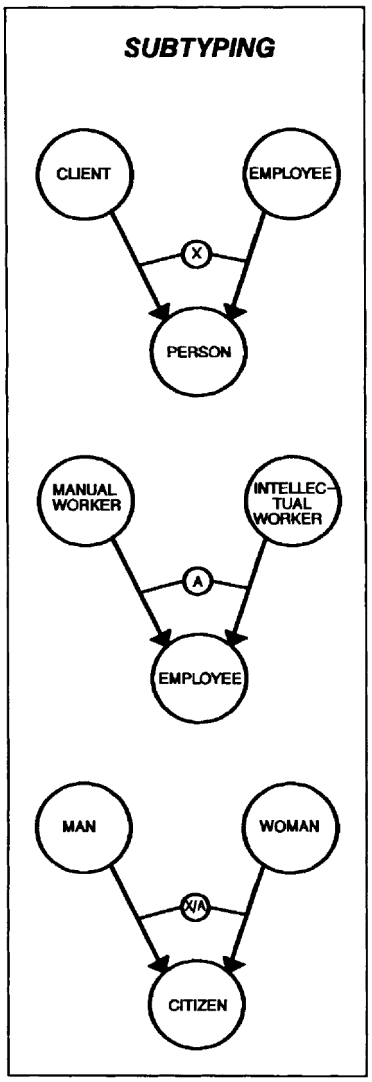


figure 46

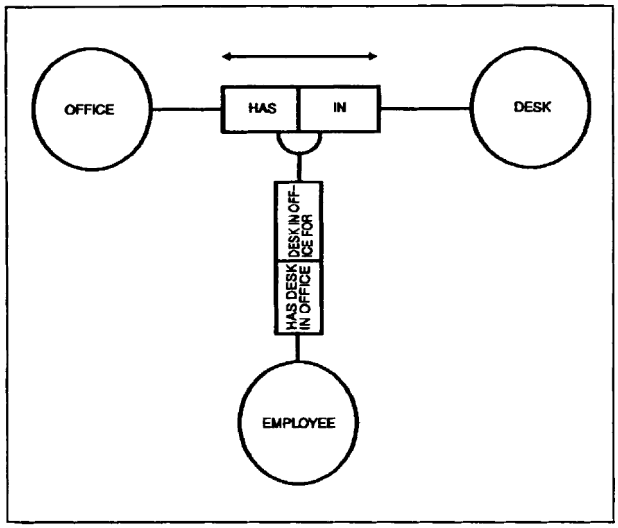


figure 47

More constraints

NIAM expresses much semantic information as constraints, and it does so in a graphical way. This produces a picture which is immensely better than a free style text, however involved the picture may be. The constraints are all in the area of the referential integrity. However, there are many more constraints that one must describe along with the data. These have no graphical representation. To name one: the domain integrity of LOTs[15]. Others concern null values allowed or not allowed for fields[16]. There are also the combinatorial constraints (involving many fields) and the "row level security" constraints such as: in a Bill of material structure expressing company hierarchy a person cannot be his own manager and there exists at least one person who does not report to another person. Finally there are the chronological constraints, for instance, all items (of an order) will eventually be allocated, but not immediately.

Such constraints must still be written down in script form.

15. It is possible to have a graphical representation though: that of semantic networks (see a further section).

16. Again it is possible to represent those occurrences of the entity having a null value for the field as a subtype with that field, the super-type entity not having the field.

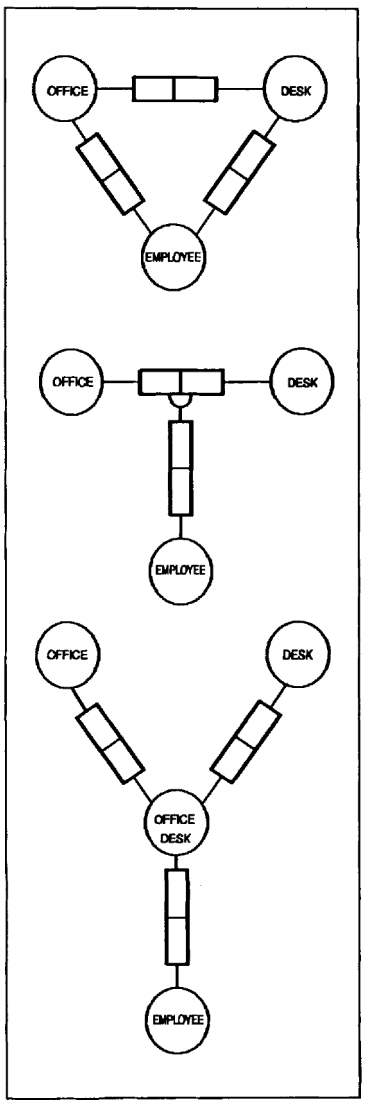


figure 48

NIAM example 1: Order Entry

We will now take up our order entry system once more. It can be translated straightforwardly from the E/R representation to a NIAM representation, which yields figure 49. The uniqueness and totality constraints added are fairly obvious, so that no explanation is required.

The subset constraint between the two customer roles expresses the rule that a customer can only be invoiced if he has received goods (i.e. he has entered orders, since all invoices are for shipments and all shipments are for allocations and all allocations are for orders; what is missing -and is not easy to represent- is the constraint that a customer cannot be invoiced for goods on an order of someone else). The subset constraint on the product roles represents the rule that a product that gets allocated is a product that is ordered in the first place. These two rules may seem fairly trivial, but they are not. What we mean is this:

* (**R1**) a product allocated in an allocation for an order is necessarily a product of that order
* (**R2**) an invoice to a customer is about shipments pertaining to orders of that customer.

These rules are far from easy to represent in a structural chart, not because the graphical representation has shortcomings, but because the rules are fairly complex. Let us have a look at R1; we do not say merely that an allocation is about a product and also about an order, but rather that it is for a product ordered on that order. Therefore, the fact type that we need is between the allocation and the fact that links a product and an order (see figure 50 which shows a higher-order fact structure and the equivalent binary structure where a new NOLOT was introduced; this NOLOT is actually an order item).

Rule R2 is much more involved. It says that shipments (on an invoice) are about allocations for (items of) orders of a given customer. Thus, the fact type that we are looking for links the invoice, the customer, the order (and item), the allocation and the shipment. It is a 6-ary fact, which we can represent as in figure 51. This figure also indicates the required constraints: the customer invoiced for the shipments is the customer the invoice goes to; the customer's orders for which the shipments are invoiced are a subset of the orders of that customer; the order lines on the orders

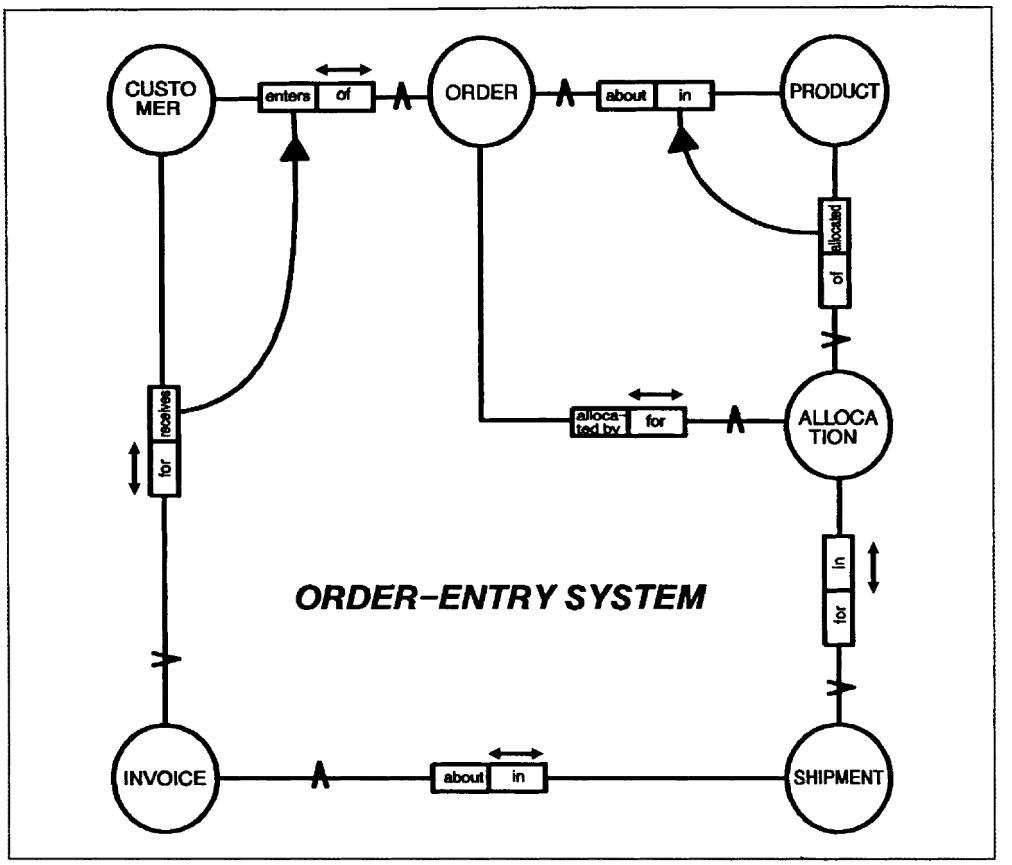


figure 49

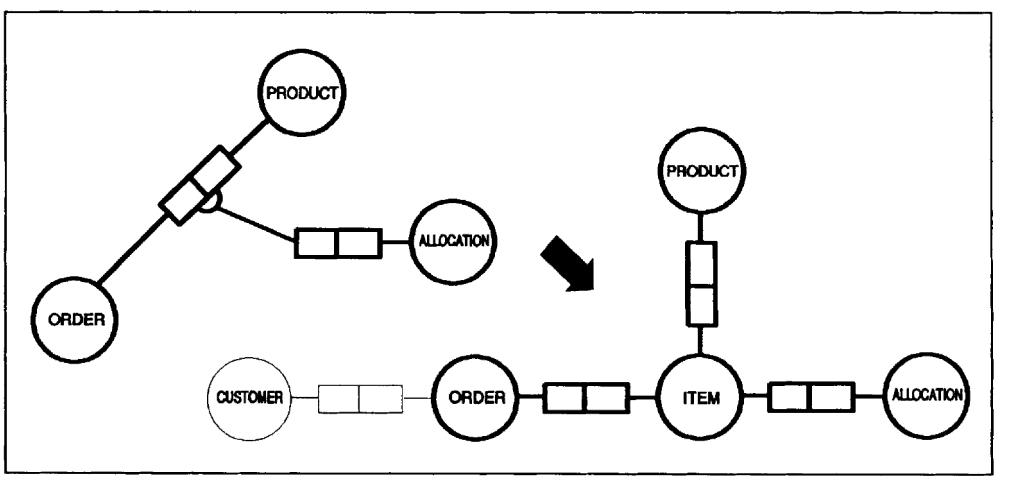


figure 50

- Course: the fact that a subject is taught once by a given teacher. If a teacher teaches the same subject more than once in a given year, these are different **courses**. A teacher teaching a course can have several teachers to replace him sporadically.
- Lecture: one *moment* of a course, held at a given location for a class of students.
- Class: the group of students that is registered for a course.

The NIAM model for this problem statement is to be found in figure 53. It is rather straightforward, even though it contains a number of (obvious) assumptions, which must certainly be verified against the reality. Note that the model is presented without any history (registration of data in the past): it is left to the reader to fill this part in[18]. There are a number of remarks, though. We have a constraint that says that a student gets assigned to a course in each subject he registered for; this constraint cannot be easily represented. A weaker form is saying that all students registered to subjects are assigned to courses, but this does not guarantee that the course to subject matching is correct. Our constraint is really a fact about the three objects student, course and subject. We can therefore represent a ternary fact (see figure 54) and state by constraints that the student-subject role combination is equal to the student-subject population expressing registration and also that the subject-course combination in the ternary fact is a subset of the subject-course fact expressing the courses scheduled for subjects. Of course, the ternary fact occurrence can only be created when a course is scheduled and a student assigned to it for a subject he registered for. A similar problem is in expressing that a student must register for all subjects that are mandatory for the degree he enrolls for. It appears that we had better not represent a fact that links students and subjects, and rely on the fact between students and courses instead. But this creates another problem: how can we then keep track of a student registering for a subject, if the relevant course has not yet been created? Finally, the facts (absence and presence) relating a student and lectures are only for the lectures in courses the student is assigned to.

Meaning, awareness and visibility

When drawing up entity/relationship models (using a combination of all the methods described), one may never be blinded by the method. The deliverable of the type of modelling that we undertake in this chapter is a diagram, certainly, but it is before anything else the statement that the demander (the user) and the data analyst have come to a mutual understanding of what the information is all about.

18. Hint: many binary facts become ternary because they will contain a role for the year.

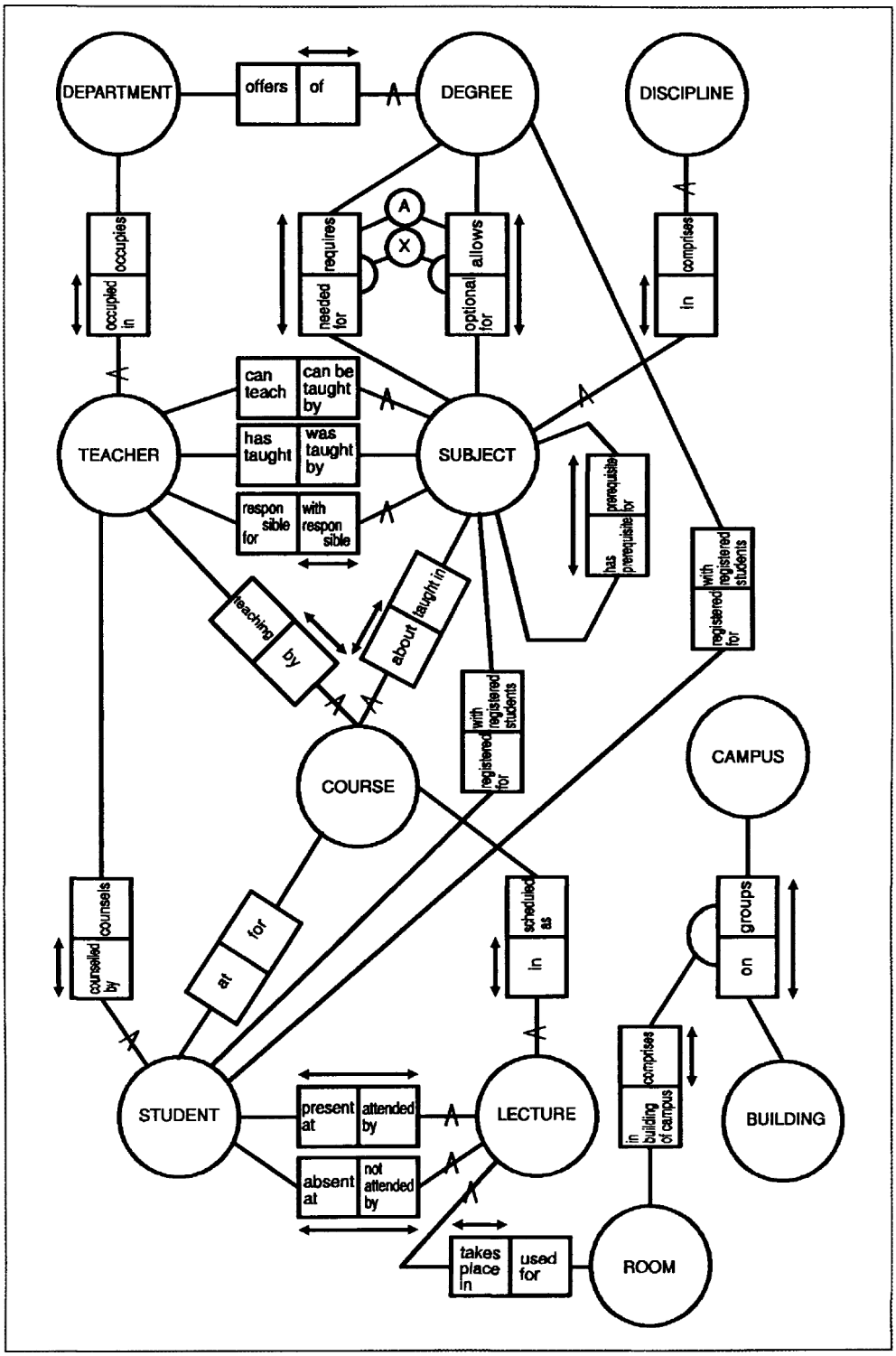


figure 53

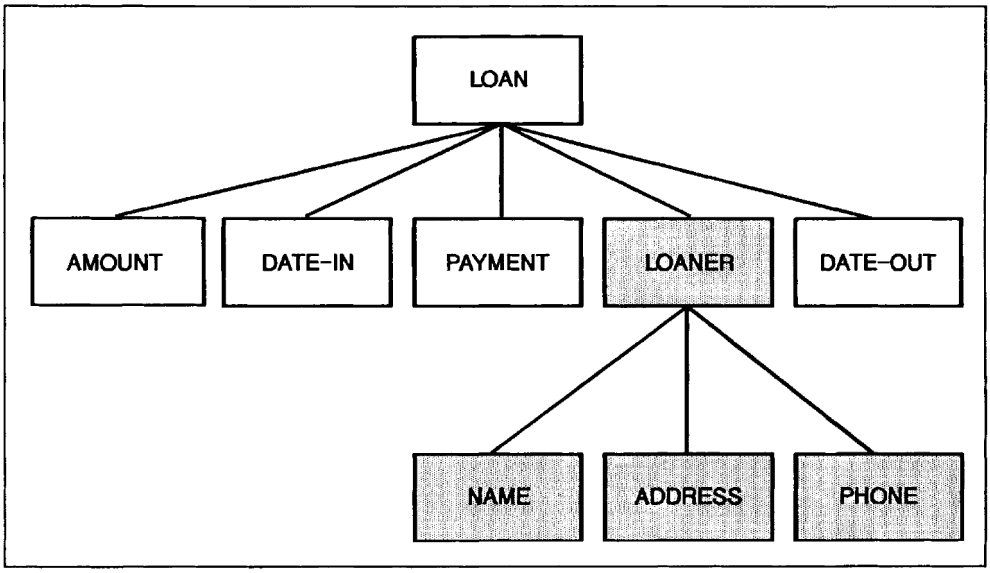


figure 64

class brings is that we can speak about the class without having to speak (or even think) about its properties.

But there is more: the icon for class and property is the same. Would that mean that class and property are actually the same thing? The answer is yes. And this is where the kind of diagram that we are creating now goes much further than E/R and NIAM. Indeed, a class can be a property of another (aggregating) class. Suppose that we have a class called LOAN (figure 64). This class has a number of fairly obvious properties: AMOUNT, DATE-IN, REIMBURSED, DATE-OUT. But it has a LOANER as well. Thus LOANER is a property of LOAN. And LOANER is itself a class. The structure can be extended in this way so that it becomes a network of classes. This (complemented with the components described below) is precisely what we mean by *has-part network.*

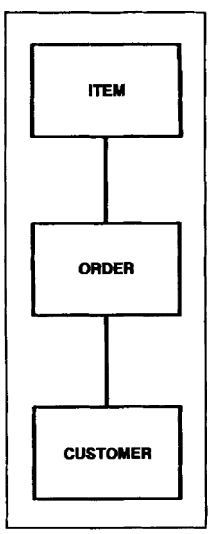


figure 65

Now, let us take a look at cardinalities. A loan has only one amount; the same amount may be that of many loans. A loan has one date-in; the same date-in may be of many loans. More importantly perhaps: a loan has one loaner, but this same loaner may have many loans. From these examples we infer the basic rule: the has-part link is in fact a 1:n relationship from the property to the class. Using ter Bekke's convention, has-part diagrams are drawn in a hierarchical fashion, so that no arrows are needed: the 1:N relationship is from bottom to top. The existence of such a relationship allows us to try and convert from E/R modeling to has-part networks. Let us first look at the

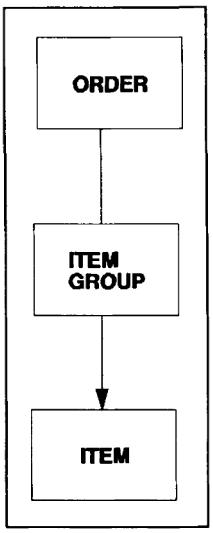


figure 66

master/detail situation, for instance customers entering orders. In a has-part network, we express that the class ORDER has a property CUSTOMER (itself a class), as in figure 65. That this calls for a reversal of our thinking is true, but the situation is fairly obvious. We might be more surprised when we try to represent the sentence: an order has order lines (items). In a has-part network we should express this as: *an item has an order*, so that we represent it as in figure 65. This is bewildering only at first sight. Indeed, looking at the LOAN class, we infer from the structure that a loan can only exist if the property *loaner* exists, i.e. a loan can only exist for a loaner. More generally: each loan has a loaner. The same is true for an item: an item exists only if it is on an order. Moreover, when we process the information in a program, we process the items as the essential information; the order is used later as a regrouping feature, and is therefore more subordinate. Still, many will prefer to use a covering: an order has an item-group which covers items (figure 66, where an arrowed line expresses the covering).

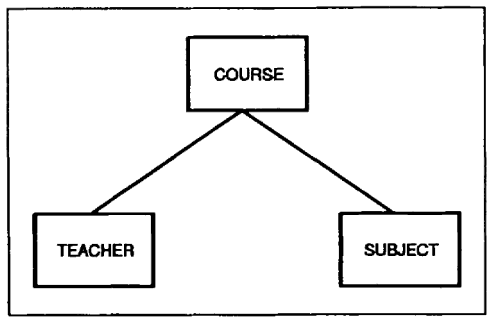


figure 67

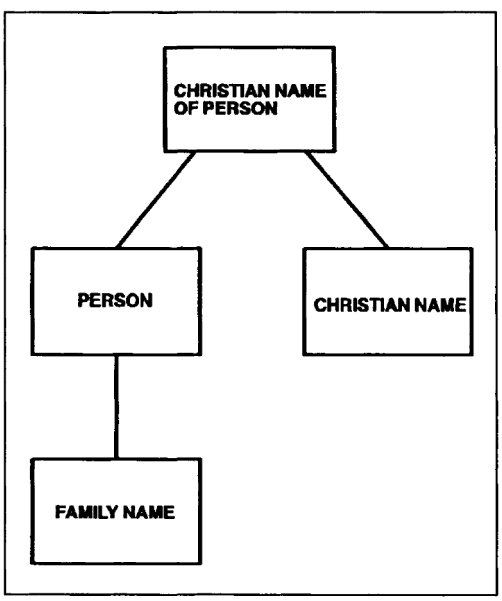


figure 68

The situation for a m:n relationship is just as easy to handle: in Bachman diagramming we replaced it by a junction entity and two 1:n relationships. This equivalence is carried over into the has-part network, so that a teacher-to-subject relationship is represented as in figure 67. Again, this is bewildering at first sight. But just think: what is important is not the fact that a teacher can (or will or may or should) teach a subject; the important thing is that there exists a *course* which is about a subject and is taught by a teacher. *Course* is the aggregating class which has the two properties *teacher* and *subject*! Has-part networks bring a very coherent way of looking at the mysteries of

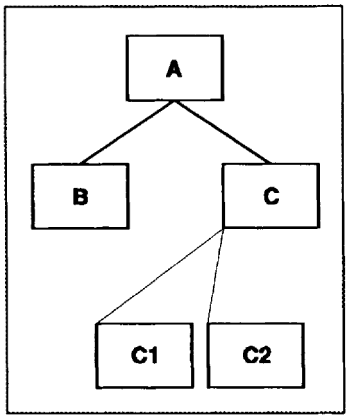


figure 70

easy to state that these are specializations of the more general class *date*. There is nothing wrong with that, as such. But, at physical implementation time, one must realize that date is a type and not an object; in other words, there is no field called *date*, but there are fields called *date-of-birth*, etc.

Another application of generalization/specialization comes about when a class has a property that is a super-class (see figure 70, where property C has two "formats": C1 and C2). This means effectively that the class has a property that can have more than one look, a situation that is identical to Pascal's record variants and C's unions. We shall call it a *categorization*: the class comprizes a number of *categories*. It should be noted that the Bachman entity model provides for categorization in combination with covering by means of the multi-member relationship. An example is the student entity which is the master of reports, but these exist in two categories: lab reports and exercise reports. The representation is in figure 71: the dot indicates the categorization; notice that there is only one relationship, even though there are two arrows. Some authors have devised an entity notation that expresses categorization in a more explicit way, see figure 72: X is the fixed (common) part of the entity whereas B and C are the two (there may be more) mutually exclusive variant parts. Each of these parts may take part in a relationship with another entity, but one must be careful to keep the model meaningful.

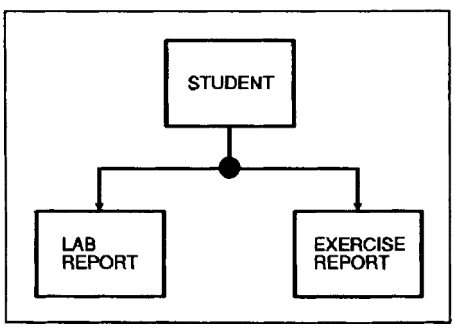


figure 71

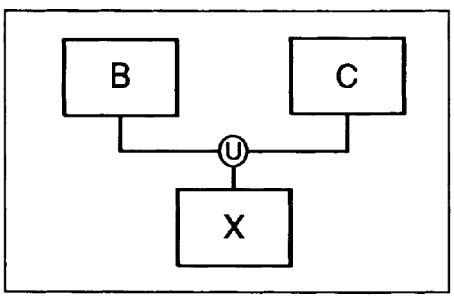


figure 72

The fourth abstraction is that of covering. In fact, covering allows for a class to have properties that are not limited to scalar occurrences, but include sets of occurrences. This is done via a covering class, though, and not directly. An example is: a department of a school has (many, possibly no) teachers. A possible graphical representation is given in figure 59, showing the teacher-group class. Note that the property that gets covered is a class itself, so that it can be an aggregate.

If we concentrate on the two important semantic operations that relate classes (aggregation and generalization), we cannot but notice that the relationship is one

in all cases. This is certainly not to be seen as an axiom, but the principle is very realistic in the type of information modeling we are interested in.

Another essential question regards the finer structure of an object. Since we need data stored with an object, it is clear that an object has fields (called *properties*). But can it have more involved structures? Can the fields be themselves groups of fields? More generally, can an object be composed of yet other aggregates or coverings? The answer here is no: an object should not contain deeper structures than a list of fields. As a result, an object is submitted to the same rules of coherence as an entity[25]. Of course, nothing prevents us from modeling at a high level of abstraction, but eventually, when drawing the final model, we must come down to only acceptable objects. As a matter of terminology, we will call those objects *normalized objects*.

The statement (and requirement) that an object contain no sub-structures has to do with coherence of manipulation. Indeed, manipulations upon a highly structured object may in fact concern only some of the nested sub-structures and therefore create the risk that non-relevant fields are nevertheless manipulated also. A number of examples will certainly clarify the need for normalized objects.

Let us consider a purchase order. This is an object, obviously. However, it contains a list of items (a repeating group). The items are existentially dependent on the order and it goes even further: there are no orders without items. In other words, for most data modellers, the items are an integral part of the order object, and one is tempted to represent the aggregate as only one object. But take a closer look at a purchase order, more specifically at which manipulations it undergoes. There is the producer side: a customer who issues a purchase order does not just write out an order. Instead, the various users enter item purchase requests. It is only later, possibly in another department, that items to be purchased are bundled into one order, the bundling criterion being (for instance) the supplier we want to order from. It is clear therefore that the objects that are manipulated on the customer (producer of the order) side are the items, and that the order is created merely as a grouping feature serving only for easy subsequent reference. Anticipating definitions given in chapter 3, the order is an indexing feature, called a *master index*. On the other hand, such a grouping feature may be used to factorize those fields of items that have the same value: for instance, an item can have an order date and this has the same value for all items bundled in an order; the same is true for the reference to the supplier. The order date will be added to the order and removed from the item[26]. As a result the order carries data, and therefore it has

25. We can therefore speak of object normalization, analogous to entity normalization, which removes from an object any covering (repeating group) and any nested aggregate (transitive dependency); an object that would contain such sub-structures will be exploded into several objects.

26. Anyone now observing that one should indeed promote the order data to the order because otherwise the item information would not be in second normal form is right, of course; however, he could very well be accused of normalization fanaticism.

many companies, this payment is not really associated to the invoice; rather it serves to cover any debts for unpaid items previously invoiced. So, the payment is fragmented and attached to the items. If this is not the case, it can be attached to the invoice, and we may have a valid reason to turn the invoice into an object after all. Similarly, credit notes that correct billed amounts, are usually not attached to an invoice as such, but to items. The conclusion is that in many cases, entities that result from calculation and formatting are not really objects.

Another typical example of an entity that is (probably) not an object is a payslip.

Our final example is the waybill in the transportation company (problem statement on page 64). We have the consignment as the real world object. In the system it is most probably represented by a consignment registration form (not described in the problem statement though), which is the system object consignment. But there is also a waybill which is attached to one consignment. The object waybill is an artefact that also represents the consignment. Therefore, consignment registration and waybill are two faces of the same object, and it would be a good idea to replace both by only one document (see also the discussion of view 4 in the data modeling of the transportation company, chapter 3).

```
FRAME restaurant
is-a:  business-establishment
types: range:(fast-food,cafetaria,
     seat-yourself,wait-to-be-seated)
  default: wait-to-be-seated
  if-needed:if plastic-orange-counter
     then fast-food
     if stack-of-trays
     then cafetaria
     if wait-for-waiter-sign
     or reservation made
     then wait-to-be-seated
     otherwise seat-yourself
location: range:type address
  if-needed:[look at menu]
name:  if-needed:[look at menu]
food-style: range:(burgers,chinese,
       continental,seafood,french)
  default: continental
  if-added: [update range of food-style]
```

figure 76

An object model is different from an entity model in one more aspect: for each object type the licit manipulations must be described as well. These manipulations are a characteristic of the object, they are defined by the object. In the world of objects they are called *methods*. The ideas behind methods are fully described in chapter 7. Methods are the static aspect of what is allowed, but there is also a dynamic aspect: what is the sequence of events an object can undergo? These aspects and their modeling are described in chapter 5.

The semantics in frames

Complex objects that depict situations meaningful as a whole are called *frames*. Consider the (classic) example of describing a restaurant; a frame is defined for this stereotype in figure 76.

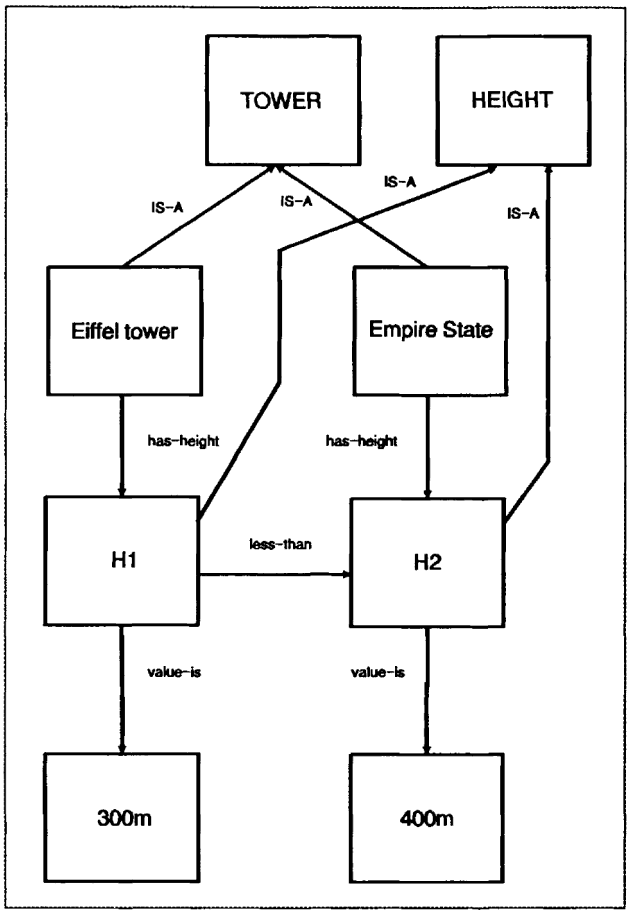


figure 79

than H2. The objects *300m* and *400m* have a meaning by themselves, as pure numbers; they are instantiations of the *value* property of the object instances H1 and H2. These objects could have other value properties as well, for instance the height in feet rather than meters.

It has been stipulated that semantic networks, built of *is-a* links, *has* links and constraint-expressing links, can represent anything, even quantified assertions such as *Everyone has read at least one book.* However, such assertions introduce a level of complexity in the representation that seems to make semantic networks less interesting[29] in that area.

A repository of structures: the meta-model

The techniques that are described in this chapter regarding the semantic modeling of information can be used for the representation of the information about the semantic model itself, of course. Indeed, what do semantics comprise? Various objects, which differ somewhat according to the representation used, but which can be summarized as follows:

- entities (or NOLOTs, or objects),
- relationships between entities (is-a, has, Bachman, E/R, facts or other),
- constraints between relationships (or roles or facts),
- relationships between relationships,
- relationships between objects and relationships.

29. At least for information modelling purposes.