Chapter VII. Conclusion: Design Styles.

One of the myths most tenaciously propagated through the oral culture which constitutes the Artificial Intelligence Community of the world is the idea that the problems tackled in this field have an intrinsic complexity which defies rational design methods. A.I. programs are considered to be so complex that they can not be designed from "behind the desk" but need to grow out of an incremental "trial and error" process which at no stage requires the programmer to have a real grasp of what has been built so far. Problem solving programming languages such as PLANNER and QA4 and knowledge representation languages like KRL and KZONE are designed to support such processes. The result of this sort of interaction between programmer and system is, hopefully, a working program which shares with human intelligence the feature of being mysterious: no one knows how it really works.

This "unstructured programming" method is just as unrealistic and idealistic as any other preconception about the design process. No programs are designed strictly in this way. In fact, the more successful programs are constructed according to considerably less fanciful methods. As an ideal, however, this programming myth exerts a real influence – which explains, among other things, the dismissals of formal logic and structured programming one often finds\(^1\) and the emphasis placed in A.I. on very highly interactive and intelligent programming tools.\(^2\)

This book has demonstrated the viability of another approach, at least for the case of a large scale natural language question-answering program. The point of departure for the PHLIQA1 system was a thorough analysis of the problem of computational question-answering followed by a careful design process aimed at separating sub-problems as clearly as possible in order to allocate them to independently operating modules with well defined interfaces. The resulting program displayed possibilities for compile-time checking of program data and separate testing of modules – useful features which are most unusual for an A.I. program. As a consequence, the

\(^1\) See, e.g., Winograd (1972) and Winograd (1974).

\(^2\) Rulifson's (1971) desiderata for the QA4 programming language constitute a most revealing testimony of this. It is the purpose of QA4 "...to provide a method whereby one can construct programs without having to understand the whole problem or even to have worked out a global structure to the solution process. We expect the programs to grow interactively and to be continually refined and improved. We feel that the programmer has a notion of how the program is to work, but does not understand enough of the notion to write algorithms. If he must express his ideas in standard formal languages the strict formality inhibits his intuition and the ideas are lost. By using QA4, he can express these ideas, ambiguous though they may be." (p. 15)
implementation phase of the final program was unusually unproblematic and fast. The debugging phase was, in fact, almost non-existent.

The approach taken here may thus be seen to have certain affinities with the ideas behind "structured programming". It is perhaps worth pointing out, therefore, that the actual design process of any large complex program differs considerably from the "structured programming ideal".

Usually, structured programming is discussed in the context of designing algorithms for mathematical or otherwise precisely defined tasks. Developing a structured program is then defined as the decomposition of a "higher-level" program written in terms of the same algorithmic constructions (such as iteration, if-then-else, and the like) as the final "low level" program – but containing unanalysed procedures which are spelled out later in terms of other primitives. For mathematical, arithmetic or business applications, this view of developing a properly structured program may be adequate, because one operates within a well-defined formal framework. So when one talks about "unanalysed primitive procedures" whose task is specified without their algorithm being spelled out, one can tacitly assume that this task definition has a certain kind of preciseness.

We can not make this assumption, however, in designing an A.I. program. The complexity and ill-definedness of the problems attacked by such programs call for extensions and refinements of the function-decomposition technique.

A theoretical investigation of the notions involved in the task of the program has therefore played an important role in the design process. Explicit characterizations of the meanings of questions, answers and data bases were developed, to make it possible to define the tasks of program modules with mathematical precision, independently of the design of the algorithms for carrying out the tasks. Because of this, the work on the design of PHLIQA1 has dovetailed, to some extent, with work on theoretically relevant issues in formal linguistics and data base theory.

During the course of the process of designing a question answering system, the "question-to-answers" function is successively described at three different levels:
- informally
- mathematically, by specifying a formal relation between input and output


4) A mathematical definition of a function may be a non-deterministic program, for example. (A case in point would be the use of an Augmented Transition Network (Woods, 1970) to define the task of a parser.) But we need not exclude the possibility of explicating the task of a procedure by means of a genuinely non-constructive definition – as when the task of a parser is defined by means of a generative transformational grammar.
– as a program (an effective algorithmic specification of the mapping from inputs to outputs).

The design of a program starts from an informal specification of a function which maps informally defined kinds of input data to informally defined kinds of output data. The design process consists in getting more specific about the data as well as about the function.

Thus, our approach can be seen to combine the idea of function-decomposition as the basic design process (as in Dijkstra, 1972) and the distinction of three different levels of abstraction in the description of the program data (Hoare and Dahl, 1972).

In hindsight, the work reported on here may therefore be seen as a defence and implementation of a particular style of software design for complex problems. An important, though largely implicit, claim of this book, then, is that the "structured design" process which resulted in the PHLIQA1 system could be most fruitfully applied to computational problems beyond the issues of question-answering and natural language modelling treated specifically here.