〔公　開〕

ＴＲ－Ｃ－００６９

Ｓｏｆｔｗａｒｅ　Ｄｅｓｉｇｎ　ａｎｄ　ｉｔｓ Ａｕｔｏｍａｔｉｏｎ　Ｆｉｎａｌ　Ｒｅｐｏｒｔ

Timothy J. Gleeson

１９９１．１０．１９

ＡＴＲ通信システム研究所

Software Design and its Automation
Final Report

Tim Gleeson

October 1991

Contents

Chapter 1

Introduction

This report summarises and reviews a year's research into software system design and its automation.

One of the goals of the ATR Communication Systems Research Laboratories is the development of technologies for the automatic generation of communications software, for example, tools to assist in the storage and reuse of aspects of software design.

The success of a tool depends on a thorough understanding of the problem it seeks to tackle. Thus it is imperative that we gain a better understanding of the design process. This requires both wide research, to gain an understanding of the basic problems involved, and deep research, to ground the research and to demonstrate feasibility. Much of the research reported here comes into the first category.

We summarise the research under the following chapters:

Philosophy First, design philosophy and theories of design in general and design for software in particular are discussed.

Psychology Next, we discuss design psychology. This includes: general psychology, as related to design; psychological activities involved in design; psychological problems and limitations; and knowledge structures and information sources used in design.

Notations One particular information structure we discuss is design notations, and the relationship between internal and external methods of information storage and their relationship.

Decisions The nature of design decisions, making them and recording them are covered.

Future Finally, we discuss the opportunities for design automation and some interesting areas which need more investigation.

These domains are somewhat orthogonal. For example, the issue of design control can be mainly seen in the area of psychological investigation of human designers, but it also has philosophical aspects in the study of design methods and automation aspects when we have to prove these ideas in practice.

A listing of all the papers, notes, surveys, abstracts, reports and reviews that I have produced over the year appear in Appendix A. Throughout this final report these documents are referred to.

There is also a large annotated bibliography, listing all the papers and books that I have read over the year. For many entries there is a short review.

Chapter 2

Philosophy of design

What is design? How do we do design? What are the basic problems of design? These questions all come under the heading of design philosophy. In this chapter, we try to outline some design philosophy and theories of design in general and design for software in particular are discussed.

2.1 What is design?

Design requires the proposal of a function to be achieved, resources with which to achieve it and the production of an artifact (the description of an assembly of those resources) that implements it. Typically, a *design task* arises in some social setting where a function has already been proposed (this is the *requirements*) and a *design process* must be undertaken to produce the required artifact. A theory of design demands a more detailed analysis of all these terms.

"Requirements" or "function" must be interpreted liberally: "goal" or "desire" are other terms. It is certainly wider than the "functional requirements" of a computer system. e.g. "Write some music which pleases me."

Software design tasks tend to be at the hard end of design, i.e. *innovative* and *creative* rather than *routine* [70]. They certainly fall into Simon's [186] class of *ill-structured* problems.

Some other factors in design are:

- Design is carried out in a large space ([28])
- Constraints may be explicit or implicit ([163, 143])
- Constraints come from several domains ([186, 143, 79])
- Requirements may be incomplete, ambiguous and contradictory ([186, 167, 79])
- Evaluation criteria are not well defined ([79])
- Novelty is often involved, so there is no predetermined solution path ([186, 78])

- Design is an *exploration* process: what is relevant only appears as design proceeds ([70])
- Trade-offs between constraints — relaxing, modifying or rejecting some — may be needed [39].

2.2 Definitions of terms

There are many terms used in studies of design. Here we provide a short list of those terms which are not further explained in this document. This list is adapted from "Survey of Design, Design Processes and Information Structures for Design".

2.2.1 constraints

Limits on the valid regions of a design space. Requirements are often given as a set of constraints. For systems design, many constraints may be implicit.

Constraints may be of the form "how a system is built" rather than "what the system does", for example, bottom-up requirements.

2.2.2 design history & rationale

A design history is a record of the activities and products of a design project. These histories are records of the external aspects of design experiences, so they can be at various levels of detail. The least detailed, and most usual, is simply the final product of the project: a program. More detail may be added in the form of requirements documents and intermediate design steps. A significant step is to include details of design decisions (5) relating intermediate design stages. These decisions represent relatively logical and presentable summaries of internal (rather less logical) decision processes.

A design rationale is an idealized version of design history documenting only its current state. This follows the argument of [157] for a document which represents the current state of the design, not the path followed to reach that state. It should represent the design as if it had been produced in an idealised, systematic way. As such, it may be useful for learning about a design.

2.2.3 design methods & tools

A design method is set of guidelines for selecting and sequencing the use of design techniques and design tools in order to construct an artifact. As such, we can regard them as well-structured, high-level design control (3.4) strategies. Examples methods are JSD, VDM and stepwise refinement. The following references discuss design methods: [6, 99, 122, 90, 27].

[67] indicates that different design methods are more or less algorithmic "from precise algorithms on the one hand to loose collections on the other". Software design methods tend to be more heuristic.

Design tools include design notations (4) and clerical aids to support a design method and its design techniques.

Some problems, or subproblems of a general problem, can be seen as search problems. For these, design heuristics and weak problem solving methods are sometimes employed. These are domain independent, but may be very inefficient. They are typically used either when the domain is small, or there is no alternative.

2.2.4 domain

A *domain* is a space, a part of the world isolated from the rest for separate treatment. The need to divide the world up this way arises from both natural and intentional acts of abstraction.

Examples of domains include:

- distributed systems
- databases
- resource allocation systems
- elevators
- libraries

A domain model is a representation of features of a domain. It may describes its relationship to other domains, possibly refinement. There is much more to a domain than just its definition. There are three important questions to be asked about domains:

- what does the domain do
- how is the domain implemented
- what higher level structures can be built using this domain

2.2.5 evaluation

Evaluation occurs in different forms at different stages during the design process:

critiquing is a diagnostic activity, mapping from undesirable behaviour to the structures which generate that behaviour.

simulation is the activity of exercising a model on some data and comparing the results with another model. Thus it is an evaluation exercise. The differences can drive diagnostic and corrective actions to alter the faulty model. Simulation can be purely symbolic or can use scenarios.

verification involves ensuring that a design proposal meets its requirements. One method is simulation.

2.2.6 requirements

Requirements define the goals of a system. They form the starting point of a design task.

Early requirements for software systems may be *incomplete*, *ambiguous* and *contradictory*. Because of this, we need to *elaborate* requirements [78] by inference of implicit requirements and addition of new ones. Inferred requirements are implicit in the requirements and real-world knowledge, but must be deduced and made explicit. Added requirements are desirable additions.

The requirements can be stated in a number of ways, including [24]: specification, constraints (2.2.1) and scenarios. Evaluation (2.2.5) criteria are an implicit form of requirement.

2.2.7 scenario

A scenario describes a sequence of events that a system performs. Scenarios can be given explicitly as problem requirements or may be retrieved from experience (usually about the problem domain) as implicit requirements. Scenarios can aid the understanding of a system specification because retrieval and simulation of them can add new requirements or help to structure requirements. See, for example, [79].

2.3 Design activities

There are an enormous number of different activities involved in software design, many interconnected, many overlapping and with an inconsistent nomenclature. Based on the work [129] we have developed a four point classification of these activities:

formulation — *defining the problem*
: requirements acquisition, understanding, specification, structuring, problem selection

synthesis — *identifying solutions consistent with requirements*
: kernel solution generation, synthesis, programming, integration, documentation

evaluation — *checking a design description for conformance with requirements*
: evaluation, testing, validation, verification, criticism, review, consistency and conformance checking

feedback — *(re)defining the problem and proposed solution*
: refinement, elaboration, maintenance, evolution, debugging, adjustment, redesign

Formulation is both the main entry point for problems, and a recursive entry point for subproblems. Feedback can be taken either iteratively over a single, evolving problem and solution or recursively on subproblems.

2.4 Abstraction and design

When we design and build large systems we must consider both the nature of large systems and of the human element in design. Tools to help users design systems must take into account basic ideas of complexity. Also, the intellectual complexity of construction depends as much on the psychology of the constructor as on the physical size of the problem. Abstraction is a tool to help us in this process.

A number of issues arise when we try to apply the idea of abstraction to the design of large systems. Here we summarise some of those issues, and then expand a little upon them:

- An abstraction depends on both the application requirements and the implementation possibilities.
- Abstraction is a compromise process between the pressures from these two domains.
- We need to understand the interactions between the two domains.
- Implementing and using an abstraction gain us a great deal of understanding.
- Some pressures (from either domain) are more important than others.
- Current specification mechanisms cannot describe many of these characteristics.
- Some non-functional requirements may be more important than other functional requirements.
- Critical issues depend on the interactions that occur.
- Critical issues cannot be abstracted: they must be made visible.
- It is hard to move an abstraction to a new environment because new interactions will make different issues critical.
- Overengineering ensures issues do not become critical.
- Abstraction is a post-hoc explanation mechanism for an already-made choice [157].

2.4.1 Some characteristics more important than others.

An abstraction is an agreement between two parties about what they both consider to be most important. Both parties will have many other considerations in mind and these will come into play when negotiating the abstraction. Each party is able to offer tradeoffs to the other. Each may be prepared to accept non-ideal characteristics of the other so long as they get something more valuable in turn. This negotiating process requires a detailed understanding of tradeoffs.

2.4.2 Critical issues depend on interaction.

An issue is not critical if it can be dismissed from consideration by a simple, general or overengineering argument. Examples of issues and arguments for their non-criticality are illustrated:

- Is loss of power critical to this abstraction?
  No. If we lose power then the whole program dies so it's not an important issue here.

- Is concurrent access a critical issue?
  No. This module is only used in a sequential environment.

- Is the order of computation a critical issue?
  No. We never need to sort more than 10 items, so order of computation is not critical.

Some issues might be critical:

- Is the speed of the abstraction critical?
  This depends partly on the abstraction itself, but mostly upon the environment in which the abstraction sits. We should

It is genuinely hard to determine if an issue is critical or not. However, it is this determination which is at the core of design and abstraction.

It is well known that it is very hard to predict which part of a program will cost the most execution time. This illustrates the fact that frequently designers do not understand much of the interaction involved in a program. An implementation is sometimes the only way to establish the relative importance of parts of a program.

2.4.3 Current specifications are inadequate

There is a strong belief in the formal methods community that many of the problems of software production ensue from inadequate specifications. This is surely true. But it is also surely a fantasy that the writing of a specification can be completely divorced from considerations of the implementation of that specification; writing a specification teaches us about the requirements, but not about the implementation. Unless a specification results from an interaction of requirements and capabilities, we will be incapable of implementing it. A good specification will have hidden most implementation considerations, but only because the specifier has the experience to find a clean abstraction in the messy space of interactions. A poor specifier may produce an implementation which appears to be free from implementation concerns, but this is illusory. A specification must describe all important aspects of the behaviour of a system. Unless a specifier has great experience in the domains of requirements and capabilities he will be unable to predict which aspects of the behaviour of a system will be important. The only way then to remove implementation considerations from a specification will be to *over*-simplify.

Frequently, specifications only document these ideal, oversimplified characteristics. They should also document all characteristics which occur in the negotiation process between application and implementation domains.

Sometimes hard non-functional requirements, such as execution speed, may take precedence over soft functional requirements, for example when a tradeoff between 32 and 31 bit integers makes a significant speed and space saving.

2.4.4 Only uncritical issues can be abstracted

Some abstractional decisions can be made which are almost invisible. For example, it is almost impossible to tell that my integers are represented in two's complement form. Looking at the range of values available and just possibly the timing behaviour under different problems may give us hints. We could X-ray the machine while operating and find more information. However, these are all insignificant aspects of the behaviour of the representation.

If the property concerned is not significant, then we can abstract from it. As we have noted, an abstraction depends on its environment. Thus whether something is invisible or not depends on the kinds of glasses we wear. If we wear X-ray glasses, then new things may become visible.

However, I can tell that the integers I declare occupy something called "memory". If I use lots of integers then my system will fail to work in rather unpredictable ways.

2.4.5 Overengineering

Let us return to our example of modeling the memory required by integers.

A very naive model would say that I can use 10,000 integers before I have any trouble with memory. I wouldn't even bother counting them in my program because I'm sure I have several orders of magnitude less than this. Overengineering requires us having an order of magnitude of leeway in the characteristics we are considering so that a simple model can be built. If we get any closer to our resource limits than this we will have to start building more detailed models. When we can overengineer, we get simple but adequate models of important (?) behaviour.

This kind of overengineering is a vital concern in many other disciplines, such as civil engineering and cooking. It is vital to know which issues are important and/or significant and which are not. If we do not make these distinctions then the morass of details will drown us. The conventional model of rigorous system formal specification and formal development does not allow for this. This kind of overengineering is a special case of what has been called *rigorous* rather than *formal* development [101]. In this case, we use simplistic models and a wide margin to convince ourselves that a formal treatment would be possible.

Overengineering gives protection from change, it makes issues non-critical. As an example, we usually use a 32 bit integer where an 8 bit integer would do.

Another way of looking at this is to say that if I have lots of memory and few integers, I can pretend that memory doesn't exist. If our representation of integers has lots of bits (32, say) and I use few, then I can pretend that overflow is impossible.

2.5 Future research

What this suggests is that it should be possible to build abstract models of the tradeoffs between the characteristics in a domain. It should be possible to do this without explicitly considering the range of implementations which lead to those tradeoffs.

Overengineering seems to be a vital, but understudied, issue.

Much of the material in this chapter was developed for "Survey of Design, Design Processes and Information Structures for Design" and "Decisions in Abstraction and Implementation".

Chapter 3

Psychology of design

Design psychology research concerns:

observation of the behaviour of problem solvers, designers and design teams

explanation of that behaviour by the proposal of systems which can exhibit it

We will look at both of these areas.

3.1 Design behaviour

There are many aspects of design behaviour which are important, but we will concentrate on just three:

- Opportunistic versus balanced design behaviour
- Novice versus expert differences
- Cognitive problems in design

3.1.1 Opportunistic versus balanced behaviour

Many researchers postulate an idealized top-down, breadth-first strategy for design problem decomposition, e.g. [171, Section 2.2]. This is called *systematic expansion* by [2] and *balanced development* by [78] when subproblems are explored to roughly the same depth.

Deviations from systematic expansion have been observed by: [35, 96, 2, 80, 78] [78] explain *opportunistic* design behaviour:

> In terms of its behavioural manifestations, opportunistic design is design in which interim decisions can lead to subsequent decisions at various levels of abstraction in the solution decomposition.
>
> Opportunistic design is characterised by on-line changes in high-level goals and plans as a result of inferences and additions of new requirements.

3.1.2 Novice versus expert differences

Expertise manifests itself in the differences between expert and novice designers, and this is a particularly fertile field for psychological experimentation.

3.1.3 Cognitive level problems

It is useful to study the problems and difficulties we have in design, because this allows us to direct our work to compensating for these problems. This is done both in work on notations, for example external notes, and in automation.

We can break knowledge sources into two big classes:

a priori knowledge either explicit or implicit. This includes previous knowledge and knowledge of the problem specification and environment.

working knowledge for example that used to store partial solutions.

Concerning a priori structures, [80, Section 8] say the main cause of design errors are:

- lack of specialised design schemas
- lack of (or poor) design meta-schema
- lack of problem-domain (i.e., problem-environment) knowledge

Concerning working structures, one major source of breakdown is due to cognitive limitations whereby the designer cannot keep all the design constraints in mind at once, or forgets to come back and deal with them.

[2] suggested that concerns arise during simulation (2.2.5) which are at the wrong level of detail; this sometimes cause tasks to be repeated. Some aspects of opportunistic design (3.1.1) can be explained by misremembering or forgetting deferred goals.

3.2 Basic mechanisms

The basic mechanisms hypothesised to produce general design behaviour are:

- Cognitive processes based on mental models
- A multi-level design mechanism

3.2.1 Mental models

Most psychological research on design assumes that humans build and manipulate *mental models* (also: *conceptual models* or *conceptualisations*) which represent the problem and proposed solutions. These are models because they support some form of simulation.

A description of the human design process from [80] is summarized here:

- Mental models are formed of the problem and solution. Often a kernel idea is rapidly adopted.
- These are refined during design:
  - The accuracy of the models (mental and external)
  - Their relationship

3.2.2 Design as a multi-level activity

Several authors have noted that the design process appears to be carried out at a number of levels. For example, [150, (unread)], cited in [121, 78] and [170] who also cite [149, (unread)]. The following levels are illustrative:

- very short term — explicable at the neural processing level.
- short-term — explicable by "conventional psychology".
- longer-term — explicable by bounded rationality
- very long-term — explicable by social and organisational theory.

At each level we find:

- Different problems
- Different processes
- Different sources of information
- Different models (and structures)
- Different goals and different forms of learning

For example, a "design team" is an entity which needs to learn just like a "designer" does. There is also activity between levels.

3.3 Information sources

Memory systems are conventionally split into *semantic memory*, which records general facts, and *episodic memory*, which records specific events or experiences. They are clearly related and Schank [175] has proposed an integrated model.

There are many models of knowledge storage, but several of them are based on having a rough outline, or schema, of a particular domain:

plans are short, stereotypical program fragments [203, 126, 54]. They are low level, but general.

cases can be episodic (if representing a specific design experience) or semantic (if representing abstracted experiences) [39].

schemas are generalised, abstract knowledge structures [5, 77].

The way we should discriminate between these and other knowledge structures concerns how they are used. Are they used implicitly, data-driven using pattern matching or explicitly, goal-directed and perhaps interpreted?

We have argued (Chapter 2) that the important characteristics that domain models carry into the design process is their models of tradeoffs between behaviours. We have also more narrowly suggested the value of additional requirements behaviour (e.g. scenarios) in synthesis, and implementation options in specification is their contribution to the evaluation of the relative importance of abstraction characteristics.

Domain knowledge can occur at varying levels of detail. It can be declarative or procedural and may be stored in various ways, for example, in books as definitions and discussions, as general knowledge in humans as design schemas, and in machines as domain models (2.2.4), or as exemplars or episodes.

Many authors conject at how domain knowledge is stored in experts and novices: [58] citing [42, 178] and [2] cite [1, (unread)]. The general view is that novices only have structures of low-level, concrete, surface features, while experts have structures involving both abstract and concrete features.

3.3.1 Learning

The study of learning is relevant to design because it shows us how knowledge sources are generated.

There are several classes of learning relevant to design. We can distinguish modes of learning by whether they apply to *working* or *a priori* knowledge:

working Learning relevant to the current design project, in particular the discovery of new knowledge and the creation of solutions.

a priori Learning relevant to later design projects including:

- generalisations, e.g. the development of design schemas (3.3)
- specific episodes

Knowledge discovery includes:

- Domain analysis, done in early stages by experts [116, 117, 58], generates domain information only implicit in the requirements.
- Simulation of solutions can lead to recognition of a solution from another part of the problem.
- A requirement can lead to the recognition of a low-level partial solution before solution decomposition.
- Simulation in the problem domain can lead to recognition of a partial solution in another part of the problem.

[58] say:

> Two very general learning mechanisms — generalization and discrimination — are part of most skill acquisition systems.

3.4 Design control

Control of the design process is a multi-level process. Most significantly we can see:

high-level goal driven behaviour

low-level data driven behaviour

as well as behaviour at higher (social) and lower (neural) levels. These two levels can be used to explain some of the observed behaviour of designers.

Design control includes:

- focusing of the designer's attention: choice of strategy and tactics
- resource (e.g. time and cognitive load) management including monitoring progress to a solution (e.g. how long a given strategy should be used for).

for this it has to

- represent goals and alternatives
- prioritise these

and it operates at every level. The interaction of different levels of control accounts for some of the observed behaviour of designers.

Now we will look at control mechanisms at different levels.

3.4.1 High-level control

High-level design control involves the setting of deliberate goals and strategies with which to solve them. Many of the strategies are design methods (2.2.3), for example, decomposition and case-based reasoning.

However, there are limitations [78]:

> Ill-structured problems, because of their ill-specified goals, prevent the determination of a single and stable high-level goal and of a corresponding initial hierarchical plan of actions to be executed throughout the design process. Ill-structured problems make a goal-directed, top-down process difficult.

3.4.2 Low-level control

The data-driven application of rules is one of the basic, unconscious, low-level processes used in design [148, 153]. It is important because the use of data-driven rules has little cognitive cost, compared with goal-directed behaviour [9].

[78]:

> The interaction of the ill structuredness of a problem with data-driven processing by experts is likely to induce the recognition of partial solutions at various levels of abstraction prior to an overall solution decomposition.
>
> Information that become the focus of attention — partial solutions, problem domain scenarios, requirements and external representations — can trigger knowledge rules.

The rapid shift in design control activities accompanying discovery of partial solutions, rather than making a note and deferring it, can be accounted for [78] by the fact that partial solutions can be easily retrieved and reused and immediately add additional constraints to the problem.

[80] partial solutions are recognised at different levels of abstraction.

3.5 Explanation of behaviour

3.5.1 Opportunistic versus balanced behaviour

[78] thinks unbalanced development can occur when experienced designers already have a good model of a system and when

- there is novelty
- multiple knowledge sources are involved
- when a subproblem appears critical, difficult or has a known solution

3.5.2 Novice versus expert differences

A number of statements are made about what expertise involves:

[79] the application of data-driven knowledge rules ([58] report that [118] empirically investigated novices and experts solving difficult problems. Novices did use means-end analysis but experts never did.)

[58] having the appropriate domain-specific knowledge (citing [134, (unread)])

[80] requires detailed knowledge of the many different domains and techniques involved in design

[80] a sophisticated control of the design process

[80] the ability to consider multiple alternatives before adopting an initial solution kernel

[115] not so much the ability to solve problems but rather the ability to devise (or revise) problems that fit the solutions they already have (citing [180, (unread)])

[96] experts engage in more thorough decompositions

3.6 Future research

We have looked at some cognitive level problems (3.1.3) and we have noted how design is a multi-level activity (3.2.2). We should study the problems of design at levels other than the cognitive, for example, the social level, see e.g. [52].

We looked at cognitive *structure problems* (3.1.3). We should also look at cognitive *process* problems, for example, non-logical reasoning.

Design control at different level was examined (3.4), but the interaction of these deserves much closer study.

Chapter 4

Notations

Notations form an important information structure for design. They allow for the external manipulation and communication of design information. We will examine the nature of external design media, how they interact with internal, mental information storage and formal notations in particular. The most important observation is that we must consider the relationships between different models of a problem and its solution.

4.1 Introduction to external media

Humans already use general purpose tools in design, for example, computer editors, filesystems, pencil, paper and libraries. A better understanding of the use of external media will allow us to construct much more useful computer tools to assist in software design.

We will look at the following areas:

- The characteristics of the systems involved in the design process, i.e. internal and external memory and the design process itself.
- How and why external design media are used and problems with their use.

4.1.1 System properties

There are several significant characteristics that external media possess:

capacity often unlimited

shareability by a group or team

structure paper has pages which effect latency and indexability; hypertext has less structure

bandwith this is really a function of the communication channel between internal memory and external design media consists of our eyes and ears, paper and pencil, keyboard, mouse and screen.

Green, Bellamy and Parker [76] have made some more detailed observations about the properties of external design media:

access window This is the "width" of information that can be accessed at any time. For human short-term memory, this is small. Paper is almost completely accessible (it is very easy to move the eyes to a particular part of a page, or to turn the page). For a machine, for example an editor, the access window is an area surrounding the cursor.

viscosity How hard is it to make local changes in the stored material?

role expressiveness How well does the medium express the purpose of the stored data? Many programming languages do not do this well — much text must be read and then "de-programmed" to understand the intention of the programmer.

4.1.2 Exploitation of external design media

At a certain level of abstraction we can equate storage of information on paper and storage of information in a computer: they are both external to the user. At this level, questions about user interfaces become meaningless and we can concentrate on the basic use of externally stored information.

Data organisation

To be useful, externally stored information has to be *accessible* and this means organising it. For paper, there are many different kinds of organisations, including desks, filing systems, libraries etc. In finding information we use search processes and various forms of *indices*. Many of these forms of organisation include indices at different levels of detail.

Of particular interest is how this media is accessed as an *extension* of the human mind. The root (or the roots) to the information stored externally must exist in the human mind, and perhaps a number of levels of indices and other primary access mechanisms exist there.

Data use

External media are used for a variety of purposes including:

- short-term and long-term storage
- searching
- learning
- reference
- visualisation

- simulation

We use external design media because they have different properties from our minds which we wish to exploit. For example:

dimensionality 2D diagrams, 3D models

animation films

We can consider the differences between paper storage of information and computer storage of information. Some of the capabilities of computers merely advance what could already be done with paper and some of them create entirely new opportunities. In some cases the computer is a worse tool than pencil and paper, for example, in reading experiments [84]. For inferencing, [87] argue external sources are weaker than internal ones.

These new capabilities of computers can considerably change the habits of users. Computers have made the medium of text far more manipulable. They allow the rapid movement of large quantities of text, something which could not be easily done with pencil and paper. But this may not be a good thing; blocks of text can be seen as the *realisation* of ideas; it is the ideas that we wish to move around, but to do this we must mentally translate the blocks of text back to ideas, and this can be seen as *reverse engineering.*

This can also be seen in fields other than that of authoring: syntax-directed editors and interactive debuggers have made the medium of programs much easier to manipulate. It is now very easy to *write* the text of programs but these systems do not encourage programmers to *design* programs.

[80] and others observe that the early adoption of kernel solutions to problems seems almost universal. The urge to generate code as soon as possible can be seen as an attempt to shift the storage of solution issues from limited memory to an external medium. However, a programming language is not a good medium for the expression of high-level solution designs and there will certainly be a lot of "decompiling" up from the language to the high-level issues. See e.g. [76]

4.1.3 Channel and cognitive capacity

We must examine the problems and difficulties associated with using external media and the communication channel which links them with the human memory and human design process.

The channels carrying information from the external world to the internal world are strictly limited in capacity as is the human ability to use that information. This means we cannot observe all the external information all of the time. Thus we must multiplex these channels. Hypertext systems can be said to *multiplex* information if they allow it to be overlaid or rapidly swapped. These tools can potentially provide rapid access to large information bases, but by failing to appreciate limitations in cognitive ability and the channels of communication, they may be nearly useless.

The tradeoff between volume of information, and speed of access to it via narrow communication channels has been recognised [88, 36, 192]. There are two key cognitive observations:

1. Speed of access to information decreases rapidly with the volume of information.

2. Locality of reference to information when performing a particular task.

They notice that, like paging in virtual memory systems, human design activity seems to be centered for long periods around *tasks*, with occasional switches between tasks. When performing a task, the user will concentrate on only a small fraction of all the information available. This locality of reference to information can be used to speed up task switching if large segments of task context (perhaps several windows) can be switched together.

4.2 Building a supersystem

One of the reasons for using external design media is that they can potentially store a large volume of information. However, it needs to be integrated with internal structures to be useful. In Section 3.2.2 we concentrated on the *vertical* nature of multi-level design. Here we are more concerned with the *horizontal* interaction between the different parts of a *multi-agent* design system. The agents may be humans, computers, papers, pencils etc.

4.2.1 Example: Using a library

It may be profitable to take a "systems" view of the use of external media.

Let's look at problem solving. There are a number of problems I can solve with no access to external media. However, cognitive limitations mean that the class of such problems is very small.

However, if I am given pencil and paper, some of these problems vanish and the range of problems I can solve increases.

If I am also given a well-stocked library, with a pencil and paper, then the class of problems I can solve expands enormously. I have not changed. The library has not changed. The combined system of me and the library, however, has far more potential than either of us alone.

If we just examine the human-subsystem, we see that the human-design-process can only work on the human-memory system. However, some new, higher-level design process emerges when we build the human-external-media supersystem. This super-design-process must be distinguished from the human-design-process: it works on the joint memory system. Of course, it is entirely dependent on the human-design-process in much the same way that a Fortran system is entirely dependent on a machine-code system to execute it. It is obvious how to make the distinction in this computing example; we must do the same with the design example. As another example, a "design team" is a system working on "group documents".

4.2.2 Knowledge acquisition and learning

Here we are interested in the acquisition of knowledge by the joint computer-external-media system and not just by the human or just by the computer. This is particularly interesting because the *facts* themselves may already be in the external-media part of the system, for example, in a dictionary, textbook or encyclopedia. However, this knowledge may not be in the system as a whole.

As a system is used, external indices (for example, those in a book or hypertext system) are exchanged for internal indices. This is a dynamic process. As a system is used, it becomes more and more familiar to the user, i.e., more and more of the information (including indices) is internalised.

The dynamic nature of the interaction between people and paper can be seen not just in the long-term assimilation of information by the user, but also as a medium-term interaction whereby paper can act as a buffer for the short-term memory for ideas which may never become assimilated in long-term memory, for example, a list of tasks which must be completed.

4.2.3 Common knowledge

Common knowledge is vital for communication between the elements of a multi-agent design system. [203] in the context of the Programmer's Apprentice (PA) and KBEmacs systems, says:

> The key to cooperation between the programmer and the assistant is effective two-way communication — whose key in turn is *shared knowledge*. It would be impossibly tedious for the programmer to explain each decision to the assistant from first principles. Rather, the programmer needs to be able to rely on a body of intermediate-level shared knowledge in order to communicate decisions easily.

Let us examine group-design, as a special case of multi-agent design. The setting of goals and the realization of those goals in a software design project are usually performed by different people. Sometimes many people are involved. Sometimes many groups are involved.

The common knowledge that these people share is an *external* source of information, since it is created by a group. It is beneficial because it can be used to abbreviate, and more easily communicate, requirements and ideas. It is problematic because people may think they share the same common knowledge when this is in fact not true: they have different internal models of what this common knowledge is. This can lead to confusion, ambiguity, lack of or excess of communication. Thus the internal model of the external model is as important as the external model itself.

There is no reason why an external system should *reflect* what is going on or being represented in the human mind, a property sometimes looked for in hypertext systems. The external system should *complement* the human mind producing a new, joint supersystem which is more powerful than either of its components.

4.3 Solutions

One of the most important uses for external media is for recording solutions and partial solutions. The internal and external representations have many things in common. The representation of solutions and partial solutions should record:

- issues considered
- resolutions of issues
- open concerns

The solution representation must support various activities:

- simulation, both mental (internal) and external
- evaluation
- alteration
- the notation should provide operators for developing solutions

The mechanisms which decompose a problem into partial solutions are related to the solution storage mechanisms:

cognitive A design schema (3.3) helps to decompose a problem and can also be used to organise the storage of solutions.

external An external notation, for example a formal one, is used both to represent a solution and, by its operators, to develop that solution.

4.4 Formal specifications

Incomplete and ambiguous problem requirements are inherent to system design. Two fundamental parts of the design process are *understanding* and *elaborating* these requirements [79]. From the psychological viewpoint of the study of the design process, the writing of formal specifications provides a valuable tool because it enables us to largely separate the activities of problem elaboration from problem understanding.

4.4.1 Value of formal specifications

Understanding partly involves the construction of an internal model which reflects the external problem. It is difficult to build such an internal model because of certain cognitive limitations.

Constructing a formal specification can help, particularly if the formal notation has a calculus of properties, as illustrated by the work of Hoare et al. [89]. It is the existence of such a calculus that not only allows us to ask questions about the

problem (and get objective answers) which increases our understanding, but can also suggest those questions.

Another benefit of formality is the constructive externalisation of information. Externalisation is important because it relieves many of the cognitive problems involved in design. Many non-formal methods also encourage externalisation, but formal methods are particularly constructive in that they have objective criteria for what must be externalised. In addition, of course, the externalisation allows for the use of the formal calculi.

4.4.2 Interaction of internal and external models

A designer still needs to understand a formal specification, even though it may be complete and unambiguous. The internal model that the designer constructs will itself be incomplete and ambiguous. Thus we need a psychological study of the relationship between internal and external models and how they interact. This will be easier if the external model is formal, complete and unambiguous.

One way in which the internal and external models interact is by the process of testing, or comparison (2.2.5), where the internal model is exercised against the external one. When writing formal specifications, a calculus of properties within the specification was one of the advantages of formal specifications mentioned. In the case of using formal specifications, this can be done with both a calculus of properties and a calculus of refinement. The designer can use the internal model to suggest refinement steps, but will always be able to check these against the external model when the formal steps must be made. This is a self correcting system: the external, formal refinement step is necessarily correct, but it leads to the correction of internal model errors. Thus we get the best of both worlds: external formality and rigour and internal non-logical, creative reasoning which is corrected. We must build systems which promote this synergy.

4.5 Future research

The following issues must be addressed:

- We need a wider survey of the current uses of external design media.
- We need a more complete listing and comparison of the various characteristics of internal and external memory systems. This would be both fascinating and useful.
- From these two we must develop a better understanding of external device use. Throughout this chapter, we have seen many examples of internal and external models. The fundamental point we must consider is the relationship between these models. There is one rather basic account of this, [76]. However, they do not consider the extension of indices and access mechanisms out of the human system into the external design media, thus creating a supersystem.

Formal refinement, as practiced in Z [190] and VDM [101], will become more and more important. We can learn from this more about both human design psychology and what parts of the design process need automating.

Chapter 5

Decisions

A study of the nature of design decisions is important in a number of areas:

reverse engineering applications, e.g. [26, 21, 173], which requires the *rediscovery* of possibly age-old and implicit decisions in existing code;

maintenance which relies on the *review* of recorded or retrieved decisions; and in the original

design process itself, where design decisions are initially made [160] but which, at best, can only be tentative.

We regard the construction of a design as the progressive addition of interacting decisions. Structuring mechanisms are required to control both the static scope and dynamic extent of this set of decisions. Programming languages only provide mechanisms for structuring low-level decisions.

5.1 Properties of design decisions

A *design decision* is a tentatative commitment to a smaller design space[1]. It is often made by choosing one space from a set of alternatives. They have several properties:

- Design decisions occur at all levels of design.
- They apply to design products, "let's use quicksort", and also to design processes and in particular design control,"let's try functional decomposition".
- Design decisions are the product of the interaction of design control (3.4) and a particular problem (or subproblem).
- Design decisions, once taken, can later be revoked. Our commitment to them is limited.
- Design decisions and solutions are a form of *working* rather than *a priori* knowledge (3.1.3).

[1]This definition is entirely inadequate.

[95] lists other properties by which different decisions can be compared.

Since design decisions are tentatative, we must record them to make it easier to revoke them later [160].

A *justification* explains why a particular decision was made. In order to make decisions comprehensible, justifications must also be recorded. In order to make decisions easily revokable, rejected alternatives should also be recorded.

5.2 Information control

Abstraction, as a process of design decision hiding [156], has a clear role in computer systems development [124]. We will look at both design decision hiding and design decision revelation.

5.2.1 Scope and extent of design decisions

One way to control complexity is by limiting the *scope* of design decisions. Scoping is a mechanism for statically enforcing abstraction boundaries. Scoping allows us to defer making design decisions without affecting the rest of the design.

Building a complex system involves making a large number of design decisions where later decisions may depend on earlier decisions. However, we know that decisions are only tentatatively made. The dependence between decisions is what we mean by *extent*: when does this decision have to be reviewed?

Both scope and extent find expression in programming (and linker, and software environment) languages.

When designing systems we regard it as important to be able to easily change decisions: for this we must increase the extent of design decisions by reducing their dependencies. In order to easily make design decisions, we need to limit their scope.

5.2.2 Revelation of design decisions

As a program is transformed from its high-level description to a low-level implementation, some design decisions must be revealed, or transferred out of their original scope. We suggest that, at appropriate levels of this transformation, such information can be legitimately employed.

For example, the Modula-2 [213] concept of *opaque* types allows a type name, and its associated operations, to be exported and used, by importing modules, with no knowledge of its implementation.

However, the implementation of an importing module needs to know more about the *representation* of the opaque type $\mathcal{T}$, let's call it $\mathcal{T}_i$. This is because the importing module's implementation must allocate space for $\mathcal{T}_i$ objects. This information has nothing to do with $\mathcal{T}$'s semantics and certainly doesn't appear in the definition module. This information is needed at compile time, when $\mathcal{I}_i$ is produced from $\mathcal{I}$.

The compromise adopted by Modula-2 is not relevant here, but is discussed in "Revealing Design Decisions". The fact that some information must be revealed is relevant.

So, during the vertical transformation (compilation) process, some information must be passed horizontally (between modules).

5.3 Implementation

We will examine a number of issues related to design decisions in implementation: what knowledge is needed, what tradeoffs are involved and what the effect of optimisations is.

5.3.1 Knowledge required

How much do you really need to know about a domain to make intelligent design decisions? For example, we have given a formal specification of sorting (in "All You Ever Wanted to Know About Sorting") but this does not contain the information which is important.

There are a very large number of factors which contribute pressure to an abstraction for sorting. Here we list some of them.

- operations available on input and output data structures and the element type
- number of elements, N, to be sorted (expected maximum and absolute maximum)
- key value range, frequency distribution and location distribution
- cost of comparing keys and moving records
- variability in length of records
- additional internal data structures available
- stability of sort with respect to equal keys (for sequential input structures)
- orderings on the client's use of operations
- simplicity and size of mechanism
- interaction with virtual memory
- extra memory needed by the mechanism, both how much and when
- average-case and worst-case behaviour

Any engineer well versed in sorting mechanisms will recognise these key issues (most of them were extracted by a simple scan of [110]), and will probably have a detailed mental model of how they relate to particular implementation algorithms, data structures and tricks. However, it should be clear that we don't need to understand the *details of any implementations* in order to use an appropriate abstraction but we do need to understand the *range of possible behaviours* and how these can be

traded off against application pressures. Making these tradeoffs is the process of using an abstraction.

Sometimes, an application requirement will be most important and will force us to compromise on other solution behaviours. Sometimes, an aspect of solution behaviour will be most important and will force us to compromise on, or even change, our application level requirements.

Designing an abstraction is a difficult process which involves the interaction of knowledge from two domains: the application domain and the solution domain. The process is one of compromise between flexible behaviours from both domains.

Abstraction has a vital role in designing complex computer systems. When designing an abstraction, for example for sorting, it is not initially clear which interactions will be most critical. Design can be seen as a process of predicting and discovering critical interactions, which must then become part of the interface.

So how much do we need to know about sorting to produce a suitable abstraction for a given application? The answer is: we need to understand tradeoffs in behaviour, and not necessarily anything about particular implementation data structures and algorithms.

5.3.2 The generality/efficiency tradeoff

One way of overcoming the problems of software reuse is to write generic packages, for example, for Sets, or Stacks or double-ended queues, Deques. These packages are generic because instead of hard-wiring element types, we can parameterise the packages by them. This encourages reuse of the package in different contexts and also serves to make explicit the nature of the dependence of a package on its parameter type.

A package must make clear what characteristics it requires from its parameter types, the package is then free to exploit those characteristics. The less a package demands from its parameters, the more generally applicable that package will be. However, the less a package demands to know of its parameters, the less opportunity it will have to exploit their characteristics and produce efficient implementations. There is thus a tradeoff between writing general purpose packages and writing efficient packages.

In their discussion of design decisions, [173] note that generalisation and specialisation decisions have some implications in that it is easier to reuse or adapt generalised components though they may be less efficient and harder to test. The point made here is different: the design decisions to use a particular component has implications for the *rest* of the program, not just that component.

5.3.3 Optimisation

There are two, well-known, opposing forces in design:

- to abstract, modularize and hide design decisions to increase a system's comprehensibility and maintainability

- to reveal, open up, collapse levels and globally view systems to increase their efficiency. A major part of the design process is in making such efficiency inducing, but global and complicating design decisions.

We should seek to understand and control the interaction of these activities.

There are several kinds of optimizations which we can apply to a system. Some of these are familiar from compiler optimisers. These are illustrated in greater detail in "Scope and Extent of Design Decisions".

- Remove generality
- Remove duplicates
- Multiple use

The potential for these kinds of optimisations leads skilled designers to *anticipatory* design.

Overengineered constraints are ideal candidates for tightening up (putting into the bargaining process) when there is strong pressure from other (maybe non-functional) requirements. The availability of optimisations feeds back and affects requirements, for example, integers in CLU [123] and some Standard ML [138, 137] implementations.

Another lesson that we learn is that the relationship of design abstractions to code (or high-level abstractions to low-level abstractions) will not be at all simple. The clean structures we find in textbooks are not the ones we will find in the final code — they may be deliberately complicated. Parnas [156] noted that a clean module structure may not be evident in the final code. Software engineering in general and "design recovery" research [26], relying on the identification of abstractions in code, must carefully consider this phase of the design process.

5.4 Parameterisation: automation of decisions

The difficulties we saw in designing a sort package arise because there is a large mismatch between the general abstraction of sorting and the specific instances of it. The price we pay for generality is more than loss of speed: it is undetermined behaviour for all those "non-functional" characteristics. The only way to bridge this gap is by making design decisions.

A parameterised abstraction is simply an abstraction where we have chosen to limit our degree of variation of a particular characteristic to a well defined, and explicitly stated, set of values. Some design decisions can be effectively parameterised. For example, in the formal specification for sorting, the element type to be sorted, X, and its ordering operation, $\preceq$, are effectively parameters of the specification. Fortunately, it is also very easy to parameterise such things in implementations.

However, the linguistic mechanisms we currently have available, for example, generic types are relatively rigid and weak in the manner in which they can be parameterised. It is possible to parameterise a sorting mechanism by the input and

output selector and constructor operations available, but it is not easy. Conventional parameterisation mechanisms produce an implementation which is a simple composition of base type and parameter type implementations; this may be inadequate if, for example, we want to parameterise by additional operators on the element type, producing a substantially different result.

5.5 Future research

Other areas for further research are the extension of current parameterisation mechanisms and an understanding of why some behavioural characteristics are critical while others are not.

The decisions made dynamically by a human and resulting in hardwired, less general, less maintainable implementations, can in some cases be transferred to the computer. We need to design languages where both manual and automatic optimisation by controlled revelation of design decisions is possible.

We gave an example of where design decision information is passed out of its original scope, during compilation of a module. We need to examine what kinds of design information are useful to other modules.

Chapter 6

Future

What kind of design decisions are automateable? Automation requires having an operational representation of:

- the alternatives available (generated both from user requirements and domain options)
- evaluation criteria
- an optimisation procedure

Decisions are made both by humans and by language systems. As we automate design decisions, we tend not to call them design decisions any more. Instead they become "obvious" or "basic techniques". For example, compilation as an automated activity is no longer regarded as a decision making one, but before compilers existed, it was.

6.1 Automation research

Automated support for design activities depends on the kind of behaviour we want to support. In Section 3.2.2 we considered the various levels at which design occurs. We need to indicate what the most significant problem for design is at each level.

We have noted in (3.6) the need for research on design problems, for example at the social level. Work is already proceeding on attempts to compensate for these, e.g.

cognitive level IBIS

group level groupware

[33] suggest that design proceeds by the cooperation of several more specific problem solvers (or subprocesses or subtasks). [39] discusses propose-critique-modify (PCM) methods. He discusses 3 groups of methods for solution proposal:

- problem decomposition / solution composition

- retrieval of cases from memory
- constraint satisfaction ([191]) (only useful on small, well-defined spaces)

A special case of decomposition knowledge is a *design plan* which specifies a sequence of steps to produce a design — it is a precompiled partial solution to a design goal [68, 166, 97, 140, 33, 39]. Knowledge of how to use a domain is very important when trying to understand a problem involving that domain. The Draco system [151] attempts to ease programming by providing models of domains, such as basic algebra, along with tools for manipulating expressions in that domain. [77] also suggest the use of a library of reusable design schemas (3.3) which model problem decomposition and merging.

These domain models are certainly needed, but example or skeleton solutions only indirectly contribute to problem understanding and problem/solution requirements/behaviour interaction understanding. We can potentially build tools, however, which could indicate and possibly animate the interaction of solution domain properties and constraints.

6.2 Future research

A number of issues arise when considering how we should provide assistance for designers. These need much more research:

- We must make it easier to delay commitment to a decision. For this we must record and allow reevaluation of alternatives. See Section 5.1.
- We must build our computer support systems so they provide more space for human data-driven processing (3.4), while performing more of the goal-directed work themselves. Woods [214] considers a related idea.

6.2.1 Generic packages

It seems possible to make the implementation of a generic package dependent on aspects of the implementation of its parameters. This implies no loss of abstraction at the abstract level. This has not been done, I believe, for several reasons:

- The extra dependence may increase the compilation/link cost associated with making changes and rebuilding the system
- It is rather hard to express the characteristics of an implementation which another implementation might be interested in.

The distinction between the work of a "compiler" and the work of a "linker" seems to rest at the point where local transformations have finished and global transformations have been adopted. I don't wish to challenge this division, merely to suggest that we could try getting the linker to do more, firstly by revealing design decisions which may be useful over a more global context and secondly by removing

decisions which have been bound in at an early stage, for example, that opaque types are restricted to pointers and, more radically, stack structures.

One possible criticism of such an approach is that it will lead to systems which are riddled with dependencies. The contrary argument is that if our systems aren't riddled with dependencies, then they can't be very flexible. Explicitly representing our design decisions in this way will ultimately be helpful.

6.2.2 Tools

Current design tools, such as gIBIS [47], allow designers to record their ideas and understanding of a problem, and how these are elaborated and altered as the design progresses. Such tools can also be used to help design solutions, but the problem elaboration issues may be mixed up with solution design issues. We should attempt to separate the two because they are very different activities and need different kinds of support.

Appendix A

Writings

This chapter lists all the papers, notes, surveys, abstracts, reports and reviews that I have produced over the year.

A.1 Papers

- Tim Gleeson and Toyofumi Takenaka. The Roles of Formal Specification in the System Design Process. In *Information Processing Society of Japan: Spring Conference*, 1991.
- Tim Gleeson and Toyofumi Takenaka. All You Ever Wanted to Know About Sorting. In *Information Processing Society of Japan: Autumn Conference*, 1991.

A.2 Notes

- Notes About Automating Software Design. August 30th 1990.
- Research into Automating Software Design: External Design Media. October 3rd 1990.
- The Z Notation and Software Design. December 11th 1990.
- Scratch Proposal. April 1st 1991.
- Scope and Extent of Design Decisions. May 1991.
- Decisions in Abstraction and Implementation. May 1991.
- Revealing Design Decisions. September 1991.

A.3 Survey

- Survey of Design, Design Processes and Information Structures for Design. August 1991.

A.4 Abstract

- Generic Module Implementation Selection Based on Parameter Type. May 1991.

A.5 Conference Reports

- First Japanese Knowledge Acquisition for Knowledge-Based Systems Workshop (JKAW'90).
  Kyoto, October 25–26 1990.
- 3rd International Symposium on Future Software Environment (ISFSE3).
  Hikone, Shiga-ken, June 12–14 1991.

A.6 Paper Reviews

- PM: A System to Support the Automatic Acquisition of Programming Knowledge. R. G. Reynolds, J. I. Maletic and S. E. Porvin. *IEEE Transactions on Knowledge and Data Engineering*, 2(3):273–282, September 1990.
- Philosophy and Psychology of Design: Two Papers:
  - Herbert A. Simon. The architecture of Complexity. *Proceedings of the American Philosophical Society*, 106(6):467–482, December 1962.
  - Raymonde Guindon, Bill Curtis, and Herb Krasner. A model of cognitive processes in software design: An analysis of breakdown in early design activities by individuals. Technical Report STP–283–87, MCC, Austin, Texas, 1987.

Bibliography

[1] Beth Adelson. Problem solving and the development of abstract categories in programming languages. *Memory and Cognition*, 9(4):422–433, July 1981.

[2] B. Adelson and E. Soloway. The role of domain experience in software design. *IEEE Transactions on Software Engineering*, 11(11):1351–1360, November 1985.

> A design experiment involving: familiar domain, unfamiliar object; unfamiliar domain, unfamiliar object; familiar domain, familiar object. A number of behaviours were observed and contrasted: 1) formation of mental models; 2) simulation; 3) systematic expansion; 4) representing constraints; 5) retrieving labels for plans; 6) note making.

[3] William W. Agresti, editor. *New Paradigms for Software Development.* IEEE Computer Society Press, 1986.

[4] Robert M. Akscyn, Donald L. McCracken, and Elise A. Yoder. KMS: A distributed hypermedia system for managing knowledge in organizations. *Communications of the ACM*, 31(7):820–835, July 1988.

[5] Joseph W. Alba and Lynn Hasher. Is memory schematic? *Psychological Bulletin*, 93(2):203–231, 1983.

> "It is widely agreed that the term *schema* has no fixed definition. It is most often used to refer to the general knowledge a person possesses about a particular domain. A schema allows for the encoding, storage, and retrieval of information related to that domain." Frame theory and script theory from AI are related. "A frame [139] is a schema that contains general knowledge about the structure of a particular event... A script [177] is very similar to a frame... [but] also contains more specific information about the contents of the event." This paper is a review of research results into remembering. "Schema driven encoding of complex information is characterized by four basic processes: selection, abstraction, interpretation, and integration."

[6] C. Alexander. *Notes on the Synthesis of Form.* Harvard University Press, 1967.

[7] Fernando Alonso, Jose Luis Maté, and Juan Pazos. Knowledge engineering versus software engineering. *Data & Knowledge Engineering*, 5(2):79–91, July 1990.

Flowery introduction quoting from Bacon and Carlyle on the use of tools. The review of software engineering history is excellent.

[8] J. R. Anderson. *Language Memory and Thought.* Erlbaum, 1976.

[9] J. R. Anderson. *The Architecture of Cognition.* Harvard University Press, 1983.

[10] J. R. Anderson. Skill acquisition: Compilation of weak-method problem solutions. *Psychological Review*, 94:192–210, 1987.

[11] John R. Anderson. Acquisition of cognitive skill. *Psychological Review*, 89(4):369–406, July 1982.

A framework for skill acquisition is proposed that includes two major stages in the development of a cognitive skill: a declarative stage in which facts about the skill domain are interpreted and a procedural stage in which the domain knowledge is directly embodied in procedures for performing the skill. Knowledge compilation is the process by which the skill transits from the declarative stage to the procedural stage. Once proceduralized, further learning processes operate on the skill to make the productions more selective in their range of applications. These processes include generalization, discrimination, and strengthening of productions. General interpretive procedures with no domain-specific knowledge can be applied to some facts about the domain and produce coherent and domain-appropriate behaviour. The process is slow because interpretation requires retrievals of declarative information from long-term memory and because the individual production steps of an interpreter are small in order to achieve generality. The interpretive productions require that the declarative information be represented in working memory and this can place a heavy burden on working memory capacity. The knowledge compilation process in ACT can be divided into two subprocesses. One, which is called *composition*, takes sequences of productions that follow each other in solving a particular problem and collapses them into a single production that has the effect of the sequence. The second process, *proceduralization*, builds versions of the productions that no longer require the domain specific declarative information to be retrieved from working memory.

[12] Robert S. Arnold, editor. *Tutorial on Software Restructuring.* IEEE Computer Society Press, 1986.

[13] Robert Balzer. A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering*, 11(11):1257–1268, November 1985.

Excellent survey of the field. Probably a first read. Discusses SAFE, Gist, Popart, Paddle and Glitter. They have now reached the following notion of automated compilation: "1) the specifications have to be acquired and validated; 2) this validation requires an operational semantics; 3) an interactive translation facility is needed to obtain the lower-level specification that can be automatically compiled; 4) the decisions employed in that interactive translation must be documented and recorded in a formal structure; 5) this formal development is the basis for a replay facility which enables implementations to be retrieved from the revised specification on each maintenance cycle."

[14] Phil Barnard. Research on human-computer interaction at the MRC applied psychology unit. In *Proceedings of the CHI'90 Conference on Human Factors in Computing Systems*, pages 379–380. ACM.

[15] D. Barstow, H. Shrobe, and E. Sandwall, editors. *Interactive Programming Environments.* McGraw-Hill, 1984.

[16] D. R. Barstow. An experiment in knowledge-based automatic programming. *Artificial Intelligence*, 12(2):73–119, August 1979.

[17] David Barstow. Automatic programming for streams. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 232–237, 1985.

Most automatic programming focuses on terminating programs but stream programs may not terminate. Stream operators are specified. Stream programs are specified in a simple predicate language. There are 3 types of transformations from this to code: algorithm instantiation (basically, pattern match an operator and replace it with a lump of code which is a new process); problem reduction (splitting a postcondition); stream elimination. The instantiations given in the paper may be useful.

[18] David Barstow. Artificial intelligence and software engineering. In *Proceedings of the 9th International Conference on Software Engineering*, pages 200–211, 1987.

Waffle, but with a good bibliography.

[19] David R. Barstow. Domain specific automatic programming. *IEEE Transactions on Software Engineering*, 11(11):1321–1336, November 1985.

A long description of some examples from the domain of oil-exploration system programming. An unexciting model of the interaction of programming knowledge and domain specific knowledge is introduced. Programming is modeled as formalisation and implementation, both of which are transformational activities.

[20] Friedrich L. Bauer. From specification to machine code: Program construction through formal reasoning. In *Proceedings of the 6th International Conference on Software Engineering*, pages 84–91, 1982. Reprinted in [3].

Arguments for formal methods.

[21] David G. Belanger, Ronald J. Brachman, Yih-Farn Chen, Prekumar T. Devanbu, and Peter G. Selfridge. Towards a software information system. *AT&T Technical Journal*, 62(2):22–39, March/April 1990.

Problems are: size, distributed documentation, evolution (which cause structure degradation), structure often invisible. Software information systems have four aspects: acquisition, representation, accessibility, applications. "All the projects are driven... from information about the source code of a software system." CIA extracts (limited) information from C programs and allows access via a relational DB. This may help in restructuring. One interesting metric is weight: the number of entities that an entity depends on and thus the effort required to understand it. Also: cross-coupling, binding strength. In understanding a system, four views are used: architectural, customer, code (files, functions) and domain. LaSSIE is a KBS which attempts to integrate and allow queries of these. As a downstream, reverse-engineering system the three tools all look useful.

[22] I. D. Benest, G. Morgan, and M. D. Smithurst. A humanised interface to an electronic library. In Bullinger and Shackel [34], pages 905–910.

When humans acquire factual knowledge, they tend to store the knowledge together with cues that aid its later retrieval. Finding a book in a library involves four characteristics: (a) the ability to browse quickly through indexes and along bookshelves (b) the ability to handle information in alphabetical order (c) the ability to remember geographically the position of specific items (d) to allow a book's cover design to stimulate recollection or assumption of its contents.

[23] G. D. Bergland. Structured design methodologies. In *Proceedings of the 15th Annual Design Automation Conference*, pages 475–493, June 1978. Reprinted in [25].

Very amusing and instructive. This is an introduction to the ideas of design structuring. In particular, it introduces and compares Functional Decomposition, Data Flow Design Method and Data Structure Design Method (JSP). He cites Jackson [94] as saying: the most critical factor in determining the life cycle cost of a program is the degree to which it faithfully models the program environment. He also says: "Once an optimisation has been cast in code it is like concrete. It is very difficult to undo." Law of continuing change: a system that is used undergoes continuing change until it is judged more cost effective to freeze and recreate it. Law of increasing unstructuredness: the entropy (disorder) of a system increases with time unless specific work is executed to maintain or reduce it.

[24] G. David Bergland, Geoffrey H. Krader, D. Paul Smith, and Paul M. Zislis. Improving the front end of the software-development process for large-scale systems. *AT&T Technical Journal*, 69(2):7–21, March/April 1990.

A model must: support multiple views, multiple levels of abstraction, be machine processable. Three views could be: specification, scenarios, constraints. The paper discusses some vapourware — integrating a number of existing (and some non-existing) CASE tools. It purports to cover front-end design and emphasizes formal specifications, but the support for deliberation is not discussed.

[25] Glenn D. Bergland and Ronald D. Gordon, editors. *Tutorial: Software Design Strategies.* IEEE Computer Society Press, 1981.

[26] T. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Computer*, 22(7):36–49, July 1989.

Design recovery is more general than reverse engineering and needs a domain model. This paper mentions the Desire system, but at the time it had no concept of a data model.

[27] Dines Bjørner. On the use of formal methods in software development. In *Proceedings of the 9th International Conference on Software Engineering*, pages 17–29, 1987.

"We use formalisms: (1) because they appear to help structure more finely the development; (2) because they are the primary means we know of to help guarantee correctness of software; (3) because developments that have used formalisms have been far more productive...; (4) because it is fun." The idea of a graph of development structure is strongly emphasized. "Method" is a set of guidelines for selecting and sequencing the use of "techniques" and "tools" in order to construct an artifact. "Tools" are such things as languages and clerical aids to support a method and its techniques.

"Techniques" are principles used to e.g. (1) specify abstract definitions; (2) transform these into designs and designs into code; (3) prove such transformations correct; (4) discharge proof obligations.

[28] John H. Boose. Knowledge acquisition tools, methods and mediating representations. In Motoda et al. [145], pages 25–62.

[29] Edward M. Bowden, Sarah A. Douglas, and Cathryn A. Stanford. Testing the principle of orthogonality in language design. *Human-Computer Interaction*, 4(2):95–120, 1989.

This paper validates what you already knew about learnability of languages and language design.

[30] Robert S. Boyer and J. Strother Moore. *A Computational Logic.* Academic Press, 1979.

[31] R. Brooks. Towards a theory of cognitive processes in computer programming. *International Journal of Man Machine Studies*, 9(6):737–751, November 1977.

This article discusses a cognitive model based on Newell and Simon's and implemented with rules. He suggests tens or hundreds of thousands of rules may be needed to represent a programmer's knowledge. There are three cognitive processes in design: understanding, method-finding and coding.

[32] Ruven Brooks. Categories of programming knowledge and their application. *International Journal of Man Machine Studies*, 33(3):241–246, September 1990.

The range of knowledge, from programming environment specific to application specific is noted. The following areas are touched on: application domain knowledge, program structure knowledge, interpersonal knowledge, problem-solving strategy knowledge. The article is oriented towards coding. Short.

[33] David C. Brown and B. Chandrasekaran. *Design Problem Solving: Knowledge Structures and Control Strategies.* Morgan Kaufmann, 1989.

A useful introduction to the ideas of "generic tasks". The particular system described is a routine design system. The basic design principle is hierarchical plan selection and detailing. The generic components involved are: specialists, plans, tasks, steps, constraints, failure handlers, redesigners, sponsors and selectors. A language, DSPL (Design Specialists and Plans Language), allows knowledge to be expressed in a problem-solving-type dependent but design-domain independent manner. Failures in design and redesign are given a lot of attention. An example of an air-cylinder designer is

presented. The design database used is pretty simple: each component is represented by a frame which contains attribute-value pairs. There are no component/subcomponent records but there is an extremely simple "kind-of" hierarchy which allows defaults and update triggerable constraints to be represented. Changes are placed on a stack and checkpoints can be made, confirmed and discarded.

[34] H. J. Bullinger and B. Shackel, editors. *Human-Computer Interaction — INTERACT'87.* Elsevier, 1987.

[35] J. M. Carroll, J. C. Thomas, and A. Malhotra. Clinical-experimental analysis of design problem solving. *Design Studies*, 1:84–92, 1979.

[36] S. Card and A. Henderson. A multiple, virtual workspace interface to support user task switching. In *Proceedings of the CHI+GI'87 Conference on Human Factors in Computing Systems and Graphical Interfaces*, pages 53–59.

The authors state a number of properties they want from a system to aid task switching. In VM systems, paging is clustered. They have some evidence that this is also the case for user tasks. (However, I wonder if this is something fundamental about human task working, or whether it is an artifact of current tools, e.g. paper and Macs). Thus they set up independent workspaces (Rooms) which users can switch between. To help navigation, they have a pop-up list of all room names and also an overview. Their thesis is that there is a tradeoff between volume of information (the desk fills up) and access time to that information (it become harder to find things); the tradeoff can be (partly) broken by observing the locality of information access.

[37] S. K. Card, M. Pavel, and J. E. Farrell. Window-based computer dialogues. In Shackel [183], pages 239–244.

A window taxonomy is given: time-multiplexed (scrollable, frame at a time); space-multiplexed (1D, 2D, 2 1/2D); non-homogeneous (icons, bifocals, fisheyes). The functions that windows can serve are: 1) more information; 2) access to multiple information sources; 3) combining of multiple sources of information; 4) independent control of multiple programs; 5) reminding; 6) context (for commands); 7) multiple representations. The computer acts as an external memory and communications medium "Of course, notes on paper scattered about a desk can serve a similar function, but they are not dynamic as with a computer display." Working sets, from VM systems are introduced. Some boring statistics of window working sets are presented.

[38] J. M. Carroll, J. C. Thomas, and A. Malhotra. Presentation and representation in design problem solving. *British Journal of Psychology*, 71:143–153, 1980.

Discusses programming as an example of nonstructured problem solving, which cannot be explained by existing theories.

[39] B. Chandrasekaran. Design problem solving: A task analysis. *AI Magazine*, 11(4):59–71, Winter 1990.

Design has a general structure, consisting of tasks, methods and subtask. This general notion is illustrated on PCM (propose-critique-modify) methods. Design problems in different domains differ in the mixture of subtasks and methods. Most subtasks are not really specific to design.

[40] Thomas E. Cheatham Jr., Glenn H. Holloway, and Judy A. Townley. Program refinement by transformation. In *Proceedings of the 5th International Conference on Software Engineering*, pages 430–437, 1981. Reprinted in [3].

A few syntactic program transformations are give. A little bit more than syntactic sugar.

[41] Patricia W. Cheng and Keith J. Holyoak. Pragmatic reasoning schemas. *Cognitive Psychology*, 17:391–416, 1985.

"The view that people typically reason in accord with formal logic has been overwhelmingly refuted by evidence based on experiments in conditional memory. In its place two major views have been proposed: the specific-experience view and the natural-logic view... We propose that people often reason using neither syntactic, context-free rules of inference, nor memory of specific experiences. Rather, they reason using abstract knowledge structures induced from ordinary life experiences, such as "permissions", "obligations", and "causations". Such knowledge structures are termed *pragmatic reasoning schemas*. A pragmatic reasoning schema consists of a set of generalised, context-sensitive rules which, unlike purely syntactic rules, are defined in terms of classes of goals (such as taking desirable actions or making predictions about possible future events) and relationships to those goals (such as cause and effect or preconditions and allowable action).

[42] M. T. H. Chi, P. J. Feltovich, and R. Glaser. Categorization and representation of physics knowledge by experts and novices. *Cognitive Science*, 5:121–152, 1981.

[43] Mark H. Chignell. A taxonomy of user interface terminology. *ACM SIGCHI Bulletin*, 21(4):27–34, April 1990.

The four main branches are: 1) The Basic Interface Model (the what); 2) Cognitive Engineering; 3) User Interface Engineering (the

how); 4) Applications. Cognitive engineering is split into: 2.1) Cognitive Science; 2.2) Normative Models and 2.3) Descriptive models. The taxonomy is very arguable, but at least it is somewhere to argue from.

[44] Ian A. Clark. Designing a human interface by minimising cognitive complexity. In Bullinger and Shackel [34], pages 101–108.

A graphical notation for user-interfaces is presented. A UI is complex if a "state dependent" transition appears in it.

[45] Lynne Colgan, Paul Rankin, and Maddy Brouwer-Janse. User models of the circuit design process. *ACM SIGCHI Bulletin*, 22(1):33–35, July 1990.

A very short article on some observations of analogue circuit designers using a simple tool. They used a blackboard model from [Whitefield84]. A chunking process was noted. Concerning optimization: "Users had problems thinking in more than two dimensions at once."

[46] A. M. Collins and E. F. Loftus. A spreading activation theory of semantic processing. *Psychological Review*, 82:407–428, 1975.

[47] Jeff Conklin and Michael L. Begeman. gIBIS: A hypertext tool for exploratory policy discussion. *Transactions on Office Information Systems*, 6(4):303–331, October 1988.

[48] Neal S. Coulter. Software science and cognitive psychology. *IEEE Transactions on Software Engineering*, 9(2):166–171, March 1983.

The psychological basis of "software science" is pulled from under its feet.

[49] R. D. Coyne. Design reasoning without explanation. *AI Magazine*, 11(4):72–80, Winter 1990.

This is a short (but a little too long) contrast of the classical cognitive and connectionist models of design. Connectionism can offer innovative design using the method by which it does recollection. Connectionism also offers an inexplicable mode of reasoning.

[50] Bill Curtis. Fifteen years of psychology in software engineering: Individual differences and cognitive science. In *Proceedings of the 7th International Conference on Software Engineering*, pages 97–106, March 1984.

A sound survey. The article surveys psychological research in programming from the two points of individual differences and cognitive science. The former is pretty poor (the work, rather than the article). The latter mentions 7 +/- 2, programmer knowledge bases and

some interesting work that asserts that solving unstructured problems (in this instance computer ones) is very different from solving structured ones. He also asserts that there has been too much work on coding and not enough on design. A good list of references.

[51] Bill Curtis. Implications from empirical studies of the software design process. In *Info Japan '90*, pages 127–134, 1990.

The paper mainly consider group and organizational issues, and coins the term *organizational cognition* for the joint activity carried out by an organization, rather than just by its members.

[52] Bill Curtis, Herb Krasner, and Neil Iscoe. A field study of the software design process for large systems. *Communications of the ACM*, 31(11):1268–1287, November 1988.

Software engineering technologies only marginally improve production — we must instead understand how human and organisational factors effect software development. "The volume of application domain knowledge and lack of good domain models are serious obstacles in current automatic programming systems." "There aren't enough system level thinkers to go around." "The ratio of unresolved issues to the number of issues recorded may be a valuable indicator of design stability and actual progress in the design phase." "Specification should not be formalised any faster than the rate of uncertainty about technical decisions is reduced." "In the dynamics of the team there is only one way [to communicate] — verbal." "Documentation... was often the main source of communication between successive teams." "Customers often generated... scenarios in determining their requirements but did not record them and abstracted them out of the requirements document." There are three views of the requirements problem: how they are understood, how their instability affects design and how they are communicated throughout a project. Three capabilities that we believe must be supported in a software development environment are knowledge sharing and integration, change facilitation and broad communication and coordination.

[53] Norman M. Delisle and Mayer D. Schwartz. Contexts — a partitioning concept for hypertext. *Transactions on Office Information Systems*, 5(2):168–186, April 1987.

[54] Francoise Détienne and Elliot Soloway. An empirically-derived control structure for the process of program understanding. *International Journal of Man Machine Studies*, 33(3):323–342, September 1990.

A schema is a data structure that represents concepts stored in memory. Program plans are program fragments that represent stereotypic action sequences in programming.

[55] Edsger W. Dijkstra. On the cruelty of really teaching computing science. *Communications of the ACM*, 32(12):1397–1414, December 1989.

Another sarcastic Dijkstra Diatribe. He argues that computing should be regarded as radically different from anything encountered before. The two radical novelties are: the complexity of computation and the digital nature of computation (making analogue reasoning useless). He argues that this has not been accepted by industry, academia or even mathematics. "We construct our mechanical symbol manipulators by means of human symbol manipulation." He argues for a radical change in education. There are replies by Parnas, Scherlis, van Emden, Cohen, Hamming, Karp and Winograd. Scherlis says that formal methods must be integrated into software engineering: "The effect is of proving small theorems about large systems rather than large theorems about (inevitably) small systems." *Postscript*: there are a number of very witty and enormously entertaining replies in the letters page of the March 1990 edition of CACM.

[56] Andrew Dillon. A psychological view of "user-friendliness". In Bullinger and Shackel [34], pages 157–163.

Interesting discussion and survey of the nature of mental models, knowledge acquisition and failures in knowledge acquisition. Then the results of some trashy experiment.

[57] Dennis E. Egan, Joel R. Remde, Louis M. Gomez, Thomas K. Landauer, Jennifer Eberhardt, and Carol C. Lochbaum. Formative design-evaluation of SuperBook. *Transactions on Information Systems*, 7(1):30–57, January 1989.

SuperBook is unimpressive, but the first, general part of this article is quite good. SuperBook was (a bit) better than paper when searching for particular items in text but was pretty bad when searching for vaguer, more general ideas.

[58] Renée Elio and Peternela B. Scharf. Modeling novice-to-expert shifts in problem-solving strategy and knowledge organization. *Cognitive Science*, 14(4):579–639, October–December 1990.

An excellent short survey of novice-expert work and representation schemes. EUREKA has 3 components: unorganized textbook knowledge, a means-end problem solver and a P-MOP (Problem Memory Organization Packet) network. Complete problem solving experiences are stored in the P-MOP and problem-type schemas evolve

there. The problem representation is organized around objects, features of objects, and relations between objects. A P-MOP network contains specific problem solving experiences (also called enhanced problem representation) and P-MOPs which represent a collection of common features (domain inference rules, solution methods, and problem features). A P-MOP has a set of *norms* which represent commonalities of the knowledge it organizes. Each difference from a norm is called an *index* which is a predicate-value pair that points to the representation of a specific problem-solving experience, or another P-MOP. The fact that a whole problem solving experience is stored, and not just the problem, is important because this allows the space of descriptors (norms and indices) to be redefined. A P-MOP network is very different from a more conventional model which has a complete, general schema pattern matched to solve each problem(i.e. each schema is a complete cognitive unit). Instead, the pathway of indices from the top node to some terminal node in the network can be regarded as a pattern match, and the knowledge in the P-MOPs themselves (solution trees, inferences, features as norms) along the path can be regarded as the action or conclusion side. A complete top to tip path does not have to be traversed, this means that it is much easier to alter, add and remove MOPs than it is to do similar operations with conventional schemas. Some of the restructuring mechanisms are related to those of [174] and [10]. "We think it is important that the qualitative shift in solution methods comes not from retrieving past solution trees, but from remembering and recalling relevant inferences about the problem scenarios which triggers principles, which in turn have associated contexts that specify conditions of applicability and further tests." EUREKA does no failure driven learning.

[59] C. A. Ellis, S. J. Gibbs, and G. L. Rein. Groupware: Some issues and experiences. *Communications of the ACM*, 34(1):38–58, January 1991.

This is an excellent survey article, with many references. Groupware is "Computer-based systems that support groups of people engaged in a common task (or goal) and that provide an interface to a shared environment." The key words here are goal and shared environment. Communication, collaboration and coordination are vital.

[60] David W. Embley and Scott N. Woodfield. A knowledge structure for reusing abstract data types. In *Proceedings of the 9th International Conference on Software Engineering*, pages 360–368, 1987.

Abstract Data Types (ADTs) are strongly annotated and linked in their system. Some notes on searching for the right ADT.

[61] M. S. Feather. Reuse in the context of a transformation based methodology. In REUSE83 [164], pages 50–58. Reprinted in [66].

Gist is a semi-formal behaviour specification language based on relations, historical reference, constraints and daemons. They describe mechanisms for transforming some of these constructs into (somewhat) lower-level constructs. The transformation process must rely on human guidance. This paper reports on the features of Gist which support reuse (and alteration) of specifications and their translation.

[62] Martin S. Feather. The evolution of composite system specifications. In IWSSD4 [93], pages 52–57.

Recording the evolution of the specification enhances comprehensibility and maintainability.

[63] Stephen Fickas. Automating the analysis process: An example. In IWSSD4 [93], pages 58–67.

"Our interest is in acquiring a problem description from a user, storing it in a form that allows us to reason about it, doing our best to find problems with it before design and coding starts, and finally mapping our representation to a language from which implementation may proceed." A domain representation of resource management systems is constructed including concepts: resources, resource depositories, resource managers, resource users, resource operations, usage scenarios, security operations, resource constraints, resource constraint management, queries, environmental aspects, policies. Policies include "keep a sufficient stock on shelves" and "maintain a user's privacy". "A major goal in our work... is to avoid requiring a user to generate tedious listings of the common objects, actions, and constraints of a domain... a problem description should be able to make use of shared knowledge of the domain to reduce the amount of detail provided."

[64] Gerhard Fischer. Cognitive view of reuse and redesign. *IEEE Software*, 4(4):60–72, July 1987.

Waffle and nothing much about cognitive issues.

[65] Gerhard Fischer and Andreas C. Lemke. Construction kits and design environments: Steps towards human problem-domain communication. *Human-Computer Interaction*, 3(3):179–222, 1987–1988.

This paper describes *construction kits* and *design environments*. The former is a set of building blocks for a domain: examples are a pinball machine creation program and a music construction program. The latter, in addition, contains general knowledge about design for a domain.

[66] P. Freeman, editor. *Tutorial: Software Reusability.* IEEE Computer Society Press, 1987.

[67] Peter Freeman. Fundamentals of design. In Peter Freeman and Anthony I. Wasserman, editors, *Software Design Techniques*, pages 2–22. IEEE, 4th edition, 1983.

A little design philosophy, then more specific software design. This would be a good early article to read.

[68] P. Friedland. Knowledge based experimental design in molecular genetics. In *Proceedings of the 6th International Joint Conference on Artificial Intelligence*, pages 285–287, 1979.

[69] Susan L. Gerhart. Applications of formal methods: Developing virtuoso software. *IEEE Software*, 7(5):7–10, September 1990.

Most of the examples were slanted toward hardware and fixed-function systems, rather than the softer upstream areas where trade-offs and ambiguities reside. It may be intrinsically difficult to portray any significant use of formal methods in the always confusing process of requirements analysis for systems at the user level.

[70] John S. Gero. Design prototypes: A knowledge representation schema for design. *AI Magazine*, 11(4):26–36, Winter 1990.

The description of the relation between function (goals), behaviour and structure is interesting, but design prototypes themselves are not terribly novel or well worked out.

[71] Monika Gerstendörfer and Gabriele Rohr. Which task in which representation on what kind of interface. In Bullinger and Shackel [34], pages 513–518.

Tasks involving complex structural relations with differing views and few sequential operations are best suited for visual spatial encoding and thus pictorial presentations. Procedural operations with strong time dependencies are best suited for natural language presentation i.e. linguistic encoding.

[72] D. J. Gilmore and T. R. G. Green. Are 'programming plans' psychologically real — outside Pascal? In Bullinger and Shackel [34], pages 497–503.

The 'programming plan' is a good description of the knowledge of an expert Pascal programmer, but Basic programmers seem unable to benefit from cues to plan structures. Abstracting the plan structure from a program is important, but other structures, e.g. control are also important. Some languages are 'role expressive' allowing plans to be easily represented.

[73] Allen Goldberg. Reusing software developments. In Richard N. Taylor, editor, *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*, pages 107–119, Irvine, CA, December 1990. Appears as *SIGSOFT Software Engineering Notes* 15(6) December 1990.

Component reuse is contrasted with design reuse. To achieve design reuse, designs must be captured and represented. Software development is formalised as transformations to an annotated abstract syntax tree. Design decisions are made by selecting from amongst a set of applicable transformations. "The transformational methodology supports design reuse in two interesting ways. First the creation of transformations and tactics formalizes general design knowledge in a highly reusable way. Second replay uses design decisions made for related specifications." Commons software development structures are: virtual machines, decomposition, set of rewrite rules, case analysis. Basic operators are composed using the above composition mechanisms to produce composite operators (tactics). This is the approach taken in LCF. It seems to me that derivations and programs are separated a bit too much in this approach. A form of design recovery is used to relate them.

[74] Richard E. Granda, Richard Halstead-Nussloch, and Joan M. Winters. The perceived usefulness of computer information sources: A field study. *ACM SIGCHI Bulletin*, 21(4):35–43, April 1990.

This study asked users about their computer problems and the information sources they used to solve them. The critical observation was that users were in one of three states: (learning) where they are acquiring new concepts and domain building, (solving) where they know critical concepts but still need to build new cognitive structures to solve the specific problem and (refreshing) where dormant cognitive structures need to be activated. The kind of help these users needs varies enormously. This kind of learning is very goal directed, and users will rate the likelihood of finding what they want from the various sources. Refreshers tend to know exactly what they want and where to find it. Humans are highly rated information sources because they: are interactive; are selective; can engage in queries at multiple levels of dialogue and can make assessments.

[75] C. Green and D. R. Barstow. On program synthesis knowledge. *Artificial Intelligence*, 10:241–279, 1978.

(Fragment read). A set of rules for array operations, divide and conquer and recursive to iterative transformation are given and illustrated by developing sorting algorithms.

[76] T. R. G. Green, R. K. E. Bellamy, and J. M. Parker. Parsing and gnisrap: A model of device use. In Bullinger and Shackel [34], pages 65–70.

A problem is expressed in a "task language", but aspects of it can be expressed (and solved) in a "device language". Problem solving concerns building structures in the task domain by performing actions in the device domain. A simple model of IO to a device is given "Make use of the external medium as a temporary store or as a dump when overload threatens them". Gnisrap is the process of creating the external structure and parsing of retrieving it. The strategy used is determined by several factors including: "access window" — head (small), paper (total), machine (cursor centered); "viscosity" — resistance of device language to local change; "role expressiveness" — need for "de-programming" to infer what comes next.

[77] R. Guindon and B. Curtis. Control of cognitive processes during design: What tools would support software design. In *Proceedings of the CHI'88 Conference on Human Factors in Computing Systems*, pages 263–268. ACM, 1988.

An example design schema, for asynchronous service, is given. This paper seems to be a much abbreviated version of [80].

[78] Raymonde Guindon. Designing the design process: Exploiting opportunistic thoughts. *Human-Computer Interaction*, 5(2 & 3):305–344, 1990.

This paper provides a long elaboration on the evidence for and the nature of opportunism in design. She contrast two models of cognitive processes: Hayes-Roth who suggests a blackboardish model and Anderson who suggest a more planned model. However, they can both exhibit opportunistic behaviour. Excellent reading. Suggestion for a computer design environment are: 1) It should not prescribe a fixed order of activities; 2) It should support rapid shifts between tools which represent and manipulate different kinds of objects; 3) It should provide easy navigation between these objects; 4) Informal and formal requirements should be representable; 5) Easy reorganisation of requirements, issues and decisions; 6) The origin of requirements should be supported; 7) Interim and partial design objects should be representable.

[79] Raymonde Guindon. Knowledge exploited by experts during software system design. *International Journal of Man Machine Studies*, 33(3):279–304, September 1990.

I thought this article was going to be very good, but it turned out to be just good. There are one or two new elements, but mostly it is a continuation of her earlier work. The knowledge domains identified and discussed are: problem domain; requirements and their elaboration; design solutions, their representations, simulations and

evaluations; design strategies, methods and notations; problem solving and software design schemas; problem solving and design heuristics; preferred evaluation criteria. The following are all quotes. System design involves the integration of multiple problem domains. Incomplete and ambiguous system requirements (or goals) are intrinsic to system design. As a consequence, system design has two features of ill-structured problems: poorly defined goals and no well-defined criteria to evaluate the solution. Design tasks involve extensive problem structuring. Problem structuring is the process of uncovering missing information and using it to define a problem space. Therefore the design process should encompass the discovery of new knowledge, in particular, discovery of unstated goals and evaluation criteria. The application...of data-driven rules is considered to be automatic and to impose little cognitive cost, in contrast to goal-directed behaviours. Experts organize their knowledge in terms of functional categories in their domains of expertise, whereas novices organize their knowledge in terms of surface features of the problem. Experts are expected to retrieve knowledge rules and the more complex design schemas in a data-driven manner. However, because design problems are ill-structured, the design process cannot be just the retrieval of known solutions, even in experts. The novelty in design and the incompletely specified requirements force even expert designers to punctuate the retrieval of known solutions with the inference of new requirements, the recognition of partial solutions at various levels of abstraction, and the creation of new solutions. The complexity of the design process also forces designers to use design methods and heuristics in a goal-directed manner to constrain the search for a design solution.

[80] Raymonde Guindon, Bill Curtis, and Herb Krasner. A model of cognitive processes in software design: An analysis of breakdown in early design activities by individuals. Technical Report STP–283–87, Microelectronics and Computer Technology Corporation (MCC), Austin, Texas, 1987.

[81] Christina Haas. Does the medium make a difference? two studies of writing with pen and paper and with computers. *Human-Computer Interaction*, 4(2):149–169, 1989.

The two experiments were the composition of persuasive letters and the revising of letters. The three media were pen and paper, PC and workstation. The rates of composition were similar, but the time (and thus length) for workstations was longer than for pen and paper. The quality of workstation and pen and paper composition were better than for PC work. In revision, more planning occurred with pen and paper, but the text was reread more with computers. Two theories for the lack of planning with computers were presented:

users cut short planning time because there is no distraction element (pen chewing) available; second, the medium of more malleable, so writers can begin writing sooner. The extra rereading with computers may be explained by the reading problem with computers, and the lack of "sense of text".

[82] Frank G. Halasz. Reflections on Notecards: Seven issues for the next generation of hypermedia systems. *Communications of the ACM*, 31(7):836–852, July 1988.

This article has an extensive hypermedia bibliography. It also discusses the distinction between *reference* and *composition* in links and cards.

[83] Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, September 1990.

[84] Wilfred J. Hansen and Christina Haas. Reading and writing with computers: A framework for explaining differences in performance. *Communications of the ACM*, 31(9):1080–1089, September 1988.

Experiments showed that paper was superior for reading to either PCs or workstations. For writing, workstations and paper were about the same but better than PCs. The results were explained by seven factors, four primary and three secondary. The primary factors were: page size, legibility, responsiveness and tangibility. The secondary factors, determined by the primary factors, were: sense of directness, sense of engagement and sense of text.

[85] Douglas E. Harms and Bruce W. Weide. Copying and swapping: Influences on the design of reusable software components. *IEEE Transactions on Software Engineering*, 17(5):424–435, May 1991.

An implicit operation in many languages is that of replicating a value, e.g. when assigning. This, coupled with the frequent need to use references leads to aliasing and other problems. These can be reduced if we encourage programmers to use swapping primitives and make ADTs provide explicit replicate() operations.

[86] Barbara Hayes-Roth and Perry W. Thorndyke. Integration of knowledge from text. *Journal of Verbal Learning and Verbal Behaviour*, 18(1):91–108, 1979.

[87] Barbara Hayes-Roth and Carol Walker. Configural effects in human memory: The superiority of memory over external information sources as a basis for inference verification. *Cognitive Science*, 3(2):119–140, April–June 1979.

"The ability to integrate information from diverse texts and to detect logical implications of the integrated information is fundamental to the understanding process... Detection of logical implications

requires that temporally and spatially separate facts be configured together and related according to deductive rules... Peoples' ability to detect and use logical configurations of related facts apparently derives from highly developed memory mechanisms." An *integrated* memory representation is hypothesised [86]. "It is extremely difficult for people to detect configural information in an external information source, such as a text. It is much easier for people to detect configural information if the source information has been committed to memory. The model proposed to account for these effects assumes that: (a) Subjects use search procedures that are inadequate to detect configural information; and (b) Automatic memory mechanisms organize acquired facts in memory structures that make configural information salient and highly accessible."

[88] D. A. Henderson and S. K. Card. Rooms: The use of multiple virtual workspaces to reduce space contention in a window-based graphical user interface. Technical report, Xerox PARC Intelligent Systems Laboratory, July 1986.

[89] C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin. The laws of programming. *Communications of the ACM*, 30(8):672–687, August 1987. See Corrigenda in *Communications of the ACM*, 30(9):770.

[90] A. Holgate. *The Art in Structural Design.* Oxford University Press, 1986.

[91] E. Hollnagel and D. Woods. Cognitive systems engineering: New wine in new bottles. *International Journal of Man Machine Studies*, 18(6):583–600, June 1983.

[92] Ellis Horowitz and John B. Munson. An expansive view of reusable software. *IEEE Transactions on Software Engineering*, 10(5):477–487, September 1984. Reprinted in [66].

OK. They identify: A) Reusable code; B) VHL program producing systems 1) Reusable design (e.g. Draco) 2) Reusable processor (MODEL language) 3) Reusing transformations 4) Application generators (business) 5) Prototyping; C) PL support (ADTs)

[93] *4th International Workshop on Software Specification and Design*, 1987.

[94] M. A. Jackson. *Principles of Program Design.* Academic Press, 1975.

[95] M. Jackson. *System Development.* Prentice-Hall, 1983.

[96] Robin Jeffries, Athea A. Turner, Peter G. Polson, and Michael E. Atwood. The processes involved in designing software. In J. R. Anderson, editor, *Cognitive Skills and their Acquisition*, pages 255–283. Erlbaum, 1981.

The authors do some experiments on getting people to design programs and the differences between different skilled subjects is assessed. Their theory says that general design work is embodied in a "design schema", but they do not elaborate how the "decompose problem into subproblem" bit works.

[97] L. Johnson and E. Soloway. PROUST: Knowledge based program understanding. *IEEE Transactions on Software Engineering*, 11(3):267–275, 1985. Biggerstaff gives the authors in the other order.

[98] P. N. Johnson-Laird. *Mental Models.* Cambridge University Press, 1983.

The general thesis is that a particular structure, the mental model, provides an explanation of many psychological questions such as inference, meaning and understanding.

[99] J. C. Jones. *Design Methods.* Wiley, 1980.

[100] Capers Jones, editor. *Tutorial: Programming Productivity: Issues for the Eighties.* IEEE Computer Society Press, second edition, 1986.

[101] C. B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall, 1986.

[102] W. P. Jones. 'As we may think'?: Psychological considerations in the design of a personal filing system. In Raymonde Guindon, editor, *Cognitive Science and its Applications for Human-Computer Interaction*, pages 235–287. Erlbaum, 1988.

The ME system is designed to improve the user interface to a personal database by actively modeling the user's own memory for files and for the context within which these files are used. A strong... argument asserts that the basic nature of an information retrieval task does much to determine the design of any system that would accomplish this task — whether the system is realized in the human brain or in a digital computer. It is quite possible that fundamental limitations in the use of electronic storage may be psychological rather than economic. In part, we use a filing system as an external memory that serves to extend, complement, and overcome the limitations of our own internal memories. Electronic storage... has an initial property of *impressibility* permitting the rapid transcription of detailed information and a subsequent property of *indelibility* promoting the preservation of this information, in its exact form, for long periods of time. Since all links involving a file object are bidirectional, a file's in-links and out-links are one and the same... This is not true for context objects. A context object... has a representation (it has out-links leading to terms) but no indexing (no in-links). In models of human memory, there has been an evolution away from

feature set models towards associated network models (e.g. [8, 9], [46]). The decay is a continuous version of displacement theories of forgetting in human short-term memory. The decay mechanisms of the current ME system produce an exponential forgetting function. While such a function produces a good approximation of short-term forgetting in people... long-term forgetting is better described by the power-law function. The spreading activation mechanism of the ME system is a limited, more primitive version of a spreading activation retrieval mechanism used in the ACT model of human information processing, [8, 9].

[103] William P. Jones. On the applied use of human memory models: The memory extender personal filing system. *International Journal of Man Machine Studies*, 25(2):191–228, August 1986.

The Memory Extender (ME) is based on the ideas of multi*term* indexing of *objects*. This is bidirectional allowing greater recallability and recognizability. A *context* models that of the human; this is a set of weighted terms and effectively specifies a working set of objects; this can be rapidly shifted by adding and deleting terms. It can be seen to generalise the idea of a current working directory. Files are both found and stored by context. The current context is partially defined by the files a user has recently used, and thus gradually wanders. A file's representation is also gradually altered by the contexts in which it has been accessed. A context decay mechanism keeps the context's strength constant as new terms are added. A global decay mechanism periodically operates so file strengths average to a common value. Files are deleted, after notifying the user, if they drop below a threshold. A spreading activation matching process is used. Single-term input from the user is processed in 1-2 secs. This proceeds in two stages: from query to terms and from terms to files. This can be extended to four stages: from found files back to more terms an from them back to more files. Information regarding term similarities (and synonomy) is thus inferred from usage patterns. In use, only 1 to 3 terms are generally needed to get any file from a 600m file UNIX system into the top 10 list. The system uses implicit psychological knowledge: where you go and what you do (and where you don't go and what you don't do) says something about what the files mean to you and how they relate to each other.

[104] William P. Jones. How do we distinguish the hyper from the hype in non-linear text? In Bullinger and Shackel [34], pages 1107–1113.

Snappy title, but this is just a brief survey of selection techniques

[105] Elaine Kant. Understanding and automating algorithm design. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages

1243–1253, 1985. A paper with same title appears in *IEEE Transactions on Software Engineering*, 11(11), November 1985.

The convex hull problem. The processes involved in design observed were: 1) understand the problem; 2) select a problem to work on; 3) plan a solution around a kernel idea and refine or elaborate the kernel structure; 4) execute the partially specified algorithm; 5) notice and formulate any difficulties or opportunities; 6) verify that the structure is a solution; 7) evaluate the solution (e.g. for efficiency). Execution can use scenarios or be symbolic. It can be viewed as a technique for selectively propagating constraints by moving them around in the order in which steps of the algorithm are automated. Designers work in multiple spaces.

[106] Elaine Kant and David Barstow. The refinement paradigm: The interaction of coding and efficiency knowledge in program synthesis. *IEEE Transactions on Software Engineering*, 7(5):458–471, September 1981. Reprinted in: [15] and [3].

A refinement tree is built and two cost-estimates (space-time product) are associated with each node: a lower bound (optimistic) and an upper bound (achievable) cost. Coding rules cover: representation techniques for collections, enumeration techniques for collections, representation techniques for mappings. Small trees of collection and mapping representations are shown. The internal program representation format is discussed in [16]

[107] Shmuel Katz, Charles A. Richter, and Khe-Sing The. PARIS: A system for reusing partially interpreted schemas. In *Proceedings of the 9th International Conference on Software Engineering*, pages 377–385, 1987.

"A partially interpreted program *schema* is syntactically a program or independent module in some programming language, but contains abstract entities such as abstract functions, predicates, constant symbols, unspecified domains or unrealized program parts, each represented by free variables in the abstract *entities* set. Each schema is accompanied by its *specification*, which includes: 1) Applicability conditions... 2) Section conditions [on free variables]... 3) Result assertions." The system has only been demonstrated for CSP. PARIS uses the Boyer Moore theorem prover [30] as a subsystem but this is not completely integrated. Finding schemas is not completely solved yet, a keyword mechanism related to the prover is envisaged, but the prover must always act as a final check. "An extension in which the Theorem Prover could be restricted, called "problem statement by interrogation," can be incorporated into later versions of PARIS. In that mode, once a potential schema has been selected by a few

basic criteria, the system would interrogate the user in order to internally construct a problem statement similar in form to the schema specification." "A problem statement consists of three components: an entity list, applicability conditions, and result assertions. These three components look similar to, and have the same syntax as the first three components of a schema entry in the library, except that they contain no abstract entities."

[108] Mark T. Keane. *Analogical Problem Solving.* Ellis Horwood, 1988.

This seems to be just a discussion of Holyoak's model of induction, though it's not bad for that. Lots of references.

[109] Charles Kellogg, Robert A. Gargan Jr., William Mark, James G. McGuire, Michael Pontecorvo, Jon L. Schlossberg, Joseph W. Sullivan, Michael R. Genesereth, and Narinder Singh. The acquisition, verification, and explanation of design knowledge. *ACM SIGART Newsletter*, (108):163–165, April 1989. Special issue on Knowledge Acquisition.

This is a very short report on work involving the Stanford HELIOS system. Design constraints, rationale and success or failure of verification are used to explain parts of a design.

[110] Donald E. Knuth. Sorting. In *The Art of Computer Programming*, chapter 5. Addison-Wesley, second edition, 1973.

[111] Janet L. Kolodner and Christopher K. Reisbeck, editors. *Experience, Memory, and Reasoning.* Erlbaum, 1986.

[112] Janet L. Kolodner and Robert L. Simpson Jr. Problem solving and dynamic memory. In Kolodner and Reisbeck [111].

Experience has 2 roles in problem solving: refinement and modification of the reasoning process (using both positive and negative results); provides a set of exemplars (analogies to previous cases guide and focus later decision making). MOPs are discussed. These are generalized episodes compiled from individual experiences. Individual experiences are indexed within these structures by features which differentiate them. When two experiences differ from a generalized episode in the same way, *reminding* occurs. *Analogy* occurs when predictions based on the first episode are used to analyze a new one. *Generalization* occurs when similarities between two episodes are compiled to form a new schema. A "problem solving with followup" model is introduced. (1) The problem is classified (2) a plan for resolution is generated and consequences predicted (3) results are compared with prediction; whatever the outcome, knowledge is updated, but when a failure occurs (4) results must be explained

and (5) failure corrected. Previous experience can aid in (1) classification (2) planning (3) explanation and recovery. A previous case can (1) predict additional features to be investigated (2) can suggest alternate classifications.

[113] Seiichi Komiya. Automatic programming by composing program components and its realisation method. *Future Generation Computer Systems*, 5(1):151–161, August 1989.

"A well-defined target domain does not require large numbers of standard program components". The PAPS system uses the "composing components" model of automatic program generation. The differences between code generation (from specifications) and code libraries (searched by specifications) are noted.

[114] Richard J. Koubek, Gavriel Salvendy, Hubert E. Dunsmore, and William K. LeBold. Cognitive issues in the process of software development: Review and reappraisal. *International Journal of Man Machine Studies*, 30(2):171–191, February 1989.

A bit thin, but lots of good references. There is a review of a few different models of different areas of software development. Programming subtasks include: requirements interpretation, solution design, coding, comprehension, testing, debugging, documentation, modifications. "Experts form a mental representation of the solution program while novices concentrate on specific functions of the problem."

[115] Marianne LaFrance. The quality of expertise: Implications of expert-novice differences for knowledge acquisition. *ACM SIGART Newsletter*, 108:6–14, April 1989. Special issue on Knowledge Acquisition.

"Experts not only know more *quantitatively* than those with less expertise but ...they know what they know in *qualitatively* different ways from those possessing less knowledge." Six differences are noted: 1) experts categorize problems in terms of known types, while novices use surface object and syntactic pattern matching similarities; 2) experts focus on goals rather than effects; 3) experts' knowledge is more functional than that of novices; 4) experts can recognise larger chunks and their chunks are structurally related; 5) experts' knowledge is more complex than novices' knowledge; 6) experts' knowledge is more often used unconsciously; 7) experts have greater episodic memory than novices.

[116] J. H. Larkin. Processing information for effective problem solving. *Engineering Education*, 70:285–288, 1979.

[117] J. H. Larkin, J. McDermott, D. P. Simon, and H. A. Simon. Models of competence in solving physics problems. *Cognitive Science*, 4:317–345, 1980.

[118] J. H. Larkin. The role of problem representation in physics. In Dedre Gentrer and Albert L. Stevens, editors, *Mental Models*, pages 75–98. Erlbaum, 1981.

[119] Jill H. Larkin and Herbert A. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11(1):65–100, January-March 1987.

> A "sentential" representation is a sequentially indexed data structure. A "diagrammatic" representation indexes information by location in a plane, allowing colocation and numerous "adjacencies". A diagram can be superior to a verbal description for solving problems because: "* Diagrams can group together all information that is used together, thus avoiding large amounts of search for the elements needed to make a problem solving inference. * Diagrams typically use location to group information about a single element, avoiding the need to match symbolic labels. * Diagrams automatically support a large number of perceptual inferences, which are extremely easy for humans." "The advantages of diagrams, in our view, are computational... because the indexing of this information can support extremely useful and efficient computational processes."

[120] M. M. Lehman. Process models, process programs, programming support. In *Proceedings of the 9th International Conference on Software Engineering*, pages 14–16, 1987. Response to an ICSE9 keynote address by L. Osterweil [155].

[121] Clayton H. Lewis. A research agenda for the nineties in human-computer interaction. *Human-Computer Interaction*, 5(2 & 3):125–143, 1990.

> A survey of different ideas on the central goals of HCI research. Excellent introduction.

[122] T. Y. Lin and S. D. Stotesbury. *Structural Concepts and Systems for Architects and Engineers*. Wiley, 1981.

[123] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual*, volume 114 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

> The language CLU is based on abstractions. The three abstraction mechanisms provided are procedures (for procedural abstraction), iterators (for control abstraction) and clusters (for data abstraction). Objects are of fundamental importance, and in this respect all the above mechanisms produce objects which can be manipulated like the more familiar integers and bools etc. Variables denote objects; assignment cause a variable to denote a new object; the old object

may be garbage collected if it is no longer referenceable. Parameters are passed by assigning actual to formal parameters. Because of this mechanism, there can be a huge amount of aliasing and it is important to be aware of any side effects you have on objects.

[124] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development.* MIT Press, 1986.

This is very much a book about programming; how you can specify bits of program, how you can decide on program abstractions, and how you can prove your program correct. The formal methods is rather toned down, but the whole book is clearly strongly influenced by them. The most interesting part is the semi-formal discussion of abstraction functions and representation invariants.

[125] Ralph L. London and Kathleen R. Milsted. Specifying reusable components using Z: Realistic sets and dictionaries. *ACM SIGSOFT Software Engineering Notes*, 14(3):120–127, May 1989. This also appears in the Proceedings of the Fifth International Workshop on Software Specification and Design, Pittsburgh, Pennsylvania, May 19–20, 1989.

The Smalltalk implementation of sets and dictionaries is specified and one or two interesting things appear.

[126] M. T. Lubars and M. T. Harandi. Knowledge-based software design using design schemas. In *Proceedings of the 9th International Conference on Software Engineering*, pages 253–262, 1987.

Unexciting arguments about how good the idea of design schemas is. Some ideas on schema library searching.

[127] Mitchell D. Lubars. Schematic techniques for high level support of software specification and design. In IWSSD4 [93], pages 68–75.

Vague and waffly. "Useful software designs can be abstracted into reusable design schemas [HarandiLubars85] that apply to related classes of design problems". The particular example discussed is a library system, considered as an instance of an inventory control system.

[128] B. Maher and D. H. Sleeman. Automatic program improvement: Variable usage transformations. *ACM Transactions on Programming Languages and Systems*, 5(2):236–264, April 1983. Reprinted in [12].

The Leeds Transformation System is a simple source code improver. It is data driven, having both a set of transformations and BNF for a few languages.

[129] Mary Lou Maher. Process models for design synthesis. *AI Magazine*, 11(4):49–58, Winter 1990.

She discusses: decomposition, case-based reasoning and transformation and gives some references to AI work.

[130] Ashok Malhotra, John C. Thomas, John M. Carroll, and Lance A. Miller. Cognitive processes in design. *International Journal of Man Machine Studies*, 12(2):119–140, February 1980.

Some rather disparate experiments lead to a simple 3 subprocess model of the design process and the suggestion of tools, particularly a domain-specific design information system. "The important reasons for the premature introduction of solution [by a designer, during a client-designer dialogue] seems to have to do with the designer helping the client articulate and elaborate his goals." The design process consists of 3 subprocesses: * *goal elaboration*, leading to functional requirements, alternating with * *design generation*; a component of which is * *design evaluation*, which may uncover new requirements. A suitable representation is important for design generation, perhaps by allowing the important features of the design to be represented. "A better method of communicating goals from the ultimate users or buyers of programs and the designer could *very significantly* impact the cost of software." "In many cases a user will be able to tell whether or not a design is acceptable even though he may or may not be able to articulate the basis on which he makes the decision."

[131] S. Marcus, J. McDermott, and T. Wang. Knowledge acquisition for constructive systems. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 637–639, 1985.

[132] Daniel D. McCracken and Michael A. Jackson. A minority dissenting view. In W. W. Cotterman et al., editors, *Systems Analysis and Design — A Foundation for the 1980's*, pages 551–553. Elsevier, 1981. Also appears in [3].

Life cycle bashing

[133] John McDermott, Geoffrey Dallemagne, Georg Klinker, David Marques, and David Tung. Explorations in how to make application programming easier. In Motoda et al. [145], pages 134–147.

[134] K. B. McKeithen, J. S. Reitman, H. H. Reuter, and S. C. Hirtle. Knowledge organization and skill differences in computer programmers. *Cognitive Psychology*, 13:307–325, 1981.

[135] William Mettrey. A comparison of expert system tools. *IEEE Computer*, 24(2):19–31, February 1991.

A review and comparison of: ART-IM, CLIPS, KES, Level 5, and VAX OPS5. There is a performance comparison, but the technical, documentation and support provided are also compared.

[136] James R. Miler, William C. Hill, Jean McKendree, Michael E. J. Masson, Brad Blumenthal, Loren Terveen, and Jay Zaback. The role of the system image in intelligent user assistance. In Bullinger and Shackel [34], pages 885–890.

Graphical interfaces may be especially sensitive to misconceptions for exactly the reason that they are often easier to learn and use — their ability to activate different kinds of prior knowledge, and thereby suggest conceptual models and instruction techniques to the user.

[137] Robin Milner and Mads Tofte. *Commentary on Standard ML.* MIT Press, 1991.

[138] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[139] M. Minsky. A framework for representing knowledge. In P. H. Winston, editor, *The Psychology of Computer Vision.* McGraw-Hill, 1975.

[140] S. Mittal, C. Dym, and M. Morjaria. PRIDE: An expert system for the handling of paper systems. *IEEE Computer*, 19(7), 1986.

[141] Naomi Miyako. Constructive interaction and the iterative process of understanding. *Cognitive Science*, 10:151–177, 1986.

A Function/Mechanism hierarchy, giving more and more details about a sewing machine, is proposed. In understanding (analysis), a person will move down the hierarchy with the following activities: (1) F identified (2) F questioned (3) M searched for (4) M proposed (5) M confirmed. Within this process the person is said to be in a state off "understanding" or "non-understanding" of the system. It was suggested that people do not skip levels as they proceed down. The conceptual point of view (CPOV), a spatial viewpoint, is introduced. This appeared to be stable during points of understanding and to shift frequently at points of non-understanding. Pairs of people interactively tried to understand the sewing machine, but it seemed they still had different understandings. "In two-person constructive interactions, the person who has more to say about the current topic takes the task doer's role, while the other becomes an observer, monitoring the situation. The observer can contribute by criticizing and giving topic-divergent motions, which are not the primary role of the task-doer."

[142] Carroll Morgan. Procedures, parameters and abstraction: Separate concerns. *Science of Computer Programming*, 11:17–27, 1988.

This is an excellent article. Abstraction identifies a coherent algorithmic activity that can be split from the main development process; conventionally, a procedure call is left at the point of abstraction, and its necessary properties become the specification of the procedure body. Instead, we leave the specification itself at the point of abstraction, with no *a priori* commitment to a procedure call. Procedure call is treated as Algol60 text substitution, not caring whether we substitute programming language code (as we would in the final program) or a specification (as we would in a high-level design). Parameterisation is treated as a substitution mechanism that can be applied uniformly to specifications or to program language code, whether or not a procedure call occurs there.

[143] J. Mostow. Toward better models of the design process. *AI Magazine*, pages 44–57, 1985.

[144] Jack Mostow and Donald Cohen. Automating program speedup by deciding what to cache. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 165–172, 1985.

Automatically inserting caches into lisp code isn't easy because of things like side-effects. There is a small complexity model.

[145] H. Motoda, R. Mizoguchi, J. Boose, and B. Gaines, editors. *Proceedings of the First Japanese Knowledge Acquisition for Knowledge-Based Systems Workshop — JKAW'90*, 1990.

[146] Barbee T. Mynatt and Katherine N. Macfarlane. Advanced organizers in computer instruction manuals: Are they effective? In Bullinger and Shackel [34], pages 917–921.

When semantic information is presented prior to procedural information, the semantic information is called an *advanced organizer.*

[147] Robert Neches, Bob Balzer, Neil Goldman, and David Wile. Transferring users' responsibilities to a system: The information management computing environment. In Shackel [183], pages 421–426.

The IM system is built on a uniform database model, accessing objects by description but with rules to allow the computer to do some things automatically. The practice is a bit wet, but they do consider the human model of a problem, how it is acquired, how it is applied and how some of it can be transferred to a system.

[148] A. Newell. Heuristic programming: Ill structured problems. In J. Arorofsky, editor, *Volume 3 in Prog. Oper. Res.*, pages 360–414. Wiley, 1969.

[149] A. Newell. *Unified Theories of Cognition.* Harvard University Press, 1990.

[150] A. Newell and S. K. Card. The prospects for psychological science in human-computer interaction. *Human-Computer Interaction*, 1:209–242, 1985.

[151] J. M. Neighbors. The Draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, 10(5):564–574, September 1984. Reprinted in [100] and [66].

The basic idea seems interesting, but this is a rather boring paper. Draco is a system based on having an (extensible number of) well-defined domains. Domains capture the result of a domain analysis and consist of: 1) Parser 2) Prettyprinter 3) in-domain transformations 4) refinements to other domains 5) domain specific procedures.

[152] Jakob Nielsen. The art of navigating through hypertext. *Communications of the ACM*, 33(3):296–310, March 1990.

A two-level map navigation system was employed. This is a pleasant, illustrative overview of the application of a hypertext system and some of the problems which still need to be tackled. Good bibliography.

[153] H. P. Nii. Blackboard systems: Blackboard application systems, blackboard systems from a knowledge engineering perspective. *AI Magazine*, pages 82–106, Summer 1986.

[154] Judith Reitman Olson and Gary M. Olson. The growth of cognitive modeling in human-computer interaction since GOMS. *Human-Computer Interaction*, 5(2 & 3):221–265, 1990.

Scanned. A review of the GOMS model of HCI: its strengths and weaknesses and how it has been extended.

[155] Leon Osterweil. Software processes are software too. In *Proceedings of the 9th International Conference on Software Engineering*, pages 2–13, 1987.

Process programming.

[156] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

There is no doubt that modularity of systems is a good thing, but how to achieve it is another problem. Traditionally, systems are split up according to program steps, the order in which processing is to take place. Parnas argues that such systems will only be comprehensible as a whole, and will have to respond in whole to changes in requirements and representations. Instead, systems should be split up along "information hiding" lines where every module is characterised by its knowledge of a design decision which it hides from

others. He finishes by a discussion of hierarchical structure and concludes that hierarchical structure and "clean" decomposition are two desirable but independent properties of a system structure.

[157] David Lorge Parnas and Paul C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12(2):251–257, February 1986.

Though we will never succeed in rigidly following a design process that will lead us to an ideal solution, along with appropriate documentation, we should produce our documentation as if that had been the case. The authors do give their own design process and reasons why it can never be rigidly followed (poorly understood requirements, difficulties in modularising it, changing requirements, preconceived ideas etc.), but they argue that documentation is vital: "The documentation is our medium of design and no design decisions are considered to be made until their incorporation into the documents". They make some analogies with mathematical proofs which may start of tortuously difficult and eventually be reworked: "The simpler proofs are published because the readers are interested in the truth of the theorem, not the process of discovering it". Though they note that design alternatives must be documented, their insistence on "the" document representing the current best state seems a bit Stalinist.

[158] H. Partsch and R. Steinbrüggen. Program transformation systems. *ACM Computing Surveys*, 15(3):199–236, September 1983. Reprinted in [12].

Excellent.

[159] Stephen J. Payne. Complex problem spaces: Modeling the knowledge needed to use interactive devices. In Bullinger and Shackel [34], pages 203–208.

The yoked state space hypothesis: The user of any input devices must construct and maintain two separate state spaces: the goal space, and the device space, and a mapping between the two.

[160] Colin Potts and Glen Bruns. Recording the reasons for design decisions. *IEEE Transactions on Software Engineering*, 10(418–427), 1988.

This paper argues the need for design decisions to be recorded, looks at the Liskov and Guttag text formatter. They describe a rather thin system which proposes to join a Prolog "type" checker to a hypertext system.

[161] D. J. Pullinger, T. I. Maude, and J. Parker. Software for reading text on screen. In Bullinger and Shackel [34], pages 899–904.

Previously used media, for example stone, papyri and paper, have shared one thing in common; it is apparent what one does with them. The media carries the general operational message.

[162] Brian J. Reiser. Knowledge-directed retrieval of autobiographical memories. In Kolodner and Reisbeck [111].

To retrieve episodic memory, one must guess the mental context used to encode the event, for example, constructing a plausible scenario for an occurrence of that type of event. Processing is not purely automatic, but requires strategic reasoning mechanisms to direct the search. "Activities" seem to be the best cues and strategies rely on causal reasoning.

[163] W. R. Reitman. *Cognition and Thought.* Wiley, 1965.

[164] *ITT Proceedings of the Workshop on Reusability in Programming*, September 1983.

[165] R. G. Reynolds, J. I. Maletic, and S. E. Porvin. PM: A system to support the automatic acquisition of programming knowledge. *IEEE Transactions on Knowledge and Data Engineering*, 2(3):273–282, September 1990.

[166] C. Rich. A formal representation for plans in the programmer's apprentice. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, pages 1044–1052, 1981.

[167] Charles Rich, Richard C. Waters, and Howard B. Reubenstein. Toward a requirements apprentice. In IWSSD4 [93], pages 79–86.

The Requirements Apprentice (RA) is an idea for an upstream tool to help in writing requirements. Early requirements are: incomplete, ambiguous and contradictory. "The RA... will contain an extensive library of knowledge about the particular domain of the requirement to be constructed." A list of general features of informal requirements communication (extending Balzer's work): abbreviation, ambiguity, poor ordering, incompleteness, contradiction, inaccuracy. Engineers use: "their previous experience, in the form of knowledge of the commonly occurring structures (combinations of primitives) in the domain. The term *cliché* is used here to refer to these commonly occurring structures.... Formally a cliché consists of a set of *roles* embedded in an underlying *substrate*. The roles of a cliché are the parts which vary from one use of the cliché to the next. The substrate of the cliché contains both fixed elements of structure (parts that re present in every occurrence) and *constraints*." Similar ideas to clichés are found in several other places. "A central underpinning of the RA will be an extensive library of clichés in the particular domain of the requirement to be constructed." Example clichés are:

repository, information system, tracking system. The requirements evolve using information from 3 sources: explicit statements, clichés and inferences. The RA is useful because of its: support for clichés, propagation of information and contradiction detection.

[168] Lance J. Rips. Mental muddles. In Myles Brand and Robert M. Harnish, editors, *The Representation of Knowledge and Belief*. University of Arizona Press, 1986.

The author differentiates "figurative mental models" from "literal mental models" and "more familiar formats (e.g., networks and propositions)." Rather unconstructive.

[169] Robert S. Rist. Variability in program design: The interaction of process with knowledge. *International Journal of Man Machine Studies*, 33(3):305–322, September 1990.

Unexciting.

[170] Paul S. Rosenbloom, John E. Laird, Allen Newell, and Robert McCarl. A preliminary analysis of the Soar architecture as a basis for general intelligence. *Artificial Intelligence*, 47(1–3):289–325, January 1991.

Soar is split into a number of levels. Level 1 is memory. Items are retrieved from production-memory into working-memory. Knowledge can be coded into production-memory in any way desired. Level 2 is the decision level which has a fixed decision procedure. It operates in an elaborate-decide cycle. Elaboration consists of repeatedly accessing memory until the system stabilises. After quiesence, the decision procedure selects one of the retrieved actions based on acceptability and preference information also retrieved from memory. Level 3 is the goal level. Whenever an impasse in decision making is reached, the architecture generates a subgoal of resolving the impasse. This leads to a subgoal stack. Learning can occur by chunking of subgoals. An example of multi-column subtraction is presented. Soar has a uniform architecture: it has only one memory structure, allowing learning to be general; it has only one task representation (goals) allowing smooth shifts from search to procedural behaviour; it has only one type of decision procedure. Excellent article.

[171] Mary Beth Rosson and Sherman R. Alpert. The cognitive consequences of object-oriented design. *Human-Computer Interaction*, 5(4):345–379, 1990.

An introduction to design. An introduction to OO. Speculation. Not very exciting.

[172] Winston W. Royce. Managing the development of large software systems. In *Proceedings of IEEE WESCON*, pages 1–9, August 1970. Reprinted in Proceedings of the 9th International Conference on Software Engineering.

This is the original waterfall model article (though he doesn't mention the word waterfall). It is short, well written and well diagrammed. His 5 key points are: 1) complete program design before analysis and coding begins; 2) documentation must be current and complete; 3) do the job twice if possible; 4) testing must be planned, controlled and monitored; 5) involve the customer.

[173] Spencer Rugaber, Stephen B. Ornburn, and Richard J. LeBlanc, Jr. Recognizing design decisions in programs. *IEEE Software*, pages 46–53, January 1990.

Nothing exciting. A few good references.

[174] D. E. Rumelhart and D. A. Norman. Accretion, tuning and restructuring: Three modes of learning. In J. W. Cotton and R. Klatzley, editors, *Semantic Factors in Cognition*. Erlbaum, 1978.

[175] R. C. Schank. Language and memory. In D. A. Norman, editor, *Perspectives on Cognitive Science*. Erlbaum, 1981.

[176] Roger C. Schank. Explanation: A first pass. In Kolodner and Reisbeck [111].

Original and refreshing. Explanation is a much more pervasive phenomenon than either reminding or learning. "People want to understand the world — personnaly, socially and physically. They do this by constantly creating and modifying explanations and indexing memories by the explanations they caused to be formed."

[177] R. C. Schank and R. Abelson. *Scripts, Plans, Goals and Understanding*. Erlbaum, 1977.

[178] A. H. Schoenfeld and D. J. Herrmann. Problem perception and knowledge structure in expert and novice mathematical problem solver. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 8:484–494, 1982.

[179] Wolfgang Schönpflug. Internal representation of externally stored information. In F. Klix and H. Wandke, editors, *Man-Computer Interaction Research: MACINTER-I*, pages 125–130. Elsevier, 1986.

"Retrieval from internal memory is mostly an automatic process... Retrieval from an external store is not automatic... The user has to build up an internal representation of the information stored externally, the storage device used, and the relations between information and device." *Source knowledge* refers to *focal knowledge*. "Effective use of source knowledge requires both internal representation of external sources, and internal representation of the information to which the source refers."

[180] S. Scribner. Thinking in action: Some characteristics of practical thought. In *Practical Intelligence: Origins of Competence in the Everyday World.* Cambridge University Press, 1985.

[181] Maung K. Sein, Robert P. Bostrom, and Lorne Olfman. Conceptual models in training novice users. In Bullinger and Shackel [34], pages 861–867.

Mental models are internal conceptual representations of a given system with which a user is interacting. They aid the user in: making inferences about the system; reasoning about it; guiding his actions. Opinion is divided on whether users spontaneously build (useful) mental models. The consensus is that users should be provided with (external) *conceptual* (primitive) model that will help them build mental models. Support for the efficacy of analogical conceptual models is weak. The role of conceptual models is to act as advance organizers. In their experiment, for "near transfer" tasks analogical conceptual models proved superior. For "far transfer" (more complex) tasks, abstract conceptual models proved superior. Also, subjects with low visual ability (to create and manipulate images) were handicapped by the abstract model.

[182] V. Sembugamoorthy and B. Chandrasekaran. Functional representation of devices and compilation of diagnostic problem-solving systems. In Kolodner and Reisbeck [111].

Device functionality can be represented in many dimensions, including: causal, temporal, interaction. A hierarchical structure records five significant aspects at each level: structure, function, behaviour, generic knowledge, assumptions. Understanding devices involves understanding the function-structure relation and the ability to use this in problem solving e.g. troubleshooting and prediction of the results of changes (what will happen if — WWHI).

[183] B. Shackel, editor. *Human-Computer Interaction — INTERACT'84.* Elsevier, 1984.

[184] Mary Shaw. Toward higher-level abstractions for software systems. *Data & Knowledge Engineering*, 5(2):119–128, July 1990.

Thin.

[185] J. I. A. Siddiqui and B. Ratcliff. Specification influences in program design. *International Journal of Man Machine Studies*, 31(4):393–404, October 1989.

Keywords in a problem specification (informal) affect the structure of the solution. Unexciting.

[186] H. A. Simon. The structure of ill-structured problems. *Artificial Intelligence*, 4:145–180, 1973.

[187] Stephen Slade. Case-based reasoning: A research paradigm. *AI Magazine*, 12(1):42–55, Spring 1991.

Case-based reasoning developed out of psychological models of episodic memory and the technological impetus of AI. This paper has a good bibliography and a reasonable introduction to the memory research.

[188] J. G. Snodgrass and D. Y. Y. Yun. Software requirements specification from a cognitive psychology perspective. In *Proceedings of the 1988 International Conference on Computer Languages*, pages 422–430, Miami Beach, Florida, October 1988. IEEE.

The authors assert that the reuse of standard designs and scenario execution are fundamental cognitive processes in design. Writing requirements does not make use of these underlying knowledge structure, so designers don't want to write requirements. Many references.

[189] Elliot Soloway, Jeannine Pinto, Stan Letovsky, David Littman, and Robin Lampert. Designing documentation to compensate for delocalized plans. *Communications of the ACM*, 31(11):1259–1267, November 1988.

In reading documentation, subjects use micro- and macro- strategies. Micro-strategies consist of Read, Question, Conjecture, Search and Answer episodes. Such inquiry episodes are triggered by unfulfilled expectations. Macro-strategies can either be systematic or as-needed, but the systematic approach only works for small problems. The as-needed strategy is problematic because it fails to detect delocalized plans (long range interactions). In design reviews, 34% of time is spent attempting to reconstruct design rationales, 37% is spent ensuring code-spec consistency. Problems with delocalized plans arise because of *lack* of expectations.

[190] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, 1989.

[191] M. Stefik. Planning with constraints. *Artificial Intelligence*, 16:111–140, 1981.

[192] M. Stefik, D. G. Bobrow, G. Foster, S. Lanning, and D. Tatar. WYSIWIS revised: Early experiences with multiuser interfaces. *Transactions on Office Information Systems*, 5(2):147–167, April 1987.

[193] David Steier. Creating a scientific community at the interface between engineering design and AI. *AI Magazine*, pages 18–22, Winter 1990.

Report of a workshop held at EDRC Engineering Design Research Center at CMU. Short, but a good summary of some current ideas.

[194] Louis I. Steinberg. Design as refinement plus constraint propagation: The VEXED experience. In *Proceedings of the 6th AAAI*, pages 830–835, 1987.

A VLSI design tool is described. It has a complete functional model of the system behaviour and a number of refinement rules hardwired in. At each stage it suggests applicable refinements and the user chooses. Limited domain.

[195] P. David Stotts and Richard Furuta. Petri-net-based hypertext: Document structure with browsing semantics. *Transactions on Information Systems*, 7(1):3–29, January 1989.

The authors use petri nets rather than DAGs to model a hypertext system. While I'm slightly dubious about petri nets, their ideas do seem useful. Petri nets allow the author to put much more structural and control information into a network. For example, a button can make several windows appear or several disappear. Access control can also be fitted in.

[196] A. G. Sutcliffe and A. C. Old. Do users know they have user models? some experiences in the practice of user modeling. In Bullinger and Shackel [34], pages 35–40.

Taxonomy of user models is given

[197] Bill Swartout. The GIST behaviour explainer. In *Proceedings of the 3rd AAAI*, pages 402–407, 1983.

"Two of the major impediments to understandability are the unfamiliar syntactic constructs of the language and non-local interactions between parts of the specification". This system allows scenarios to be simulated and the trace behaviour of the system summarised and justified.

[198] W. Swartout and R. Balzer. On the inevitable intertwining of specification and implementation. *Communications of the ACM*, 25(7):438–440, 1982. Also appears in [3].

Short and unconvincing. One point is noteworthy: the systems we specify and build are complex and we cannot forsee all the implications until we implement it and then those realisations can be fed back up to alter the specification. They confuse "low level requirements" with "implementation decisions".

[199] Hideaki Takeda, Paul Veerkamp, Tetsuo Tomiyama, and Hiroyuki Yoshikawa. Modelling design processes. *AI Magazine*, 11(4), Winter 1990.

Several models of the design process are possible: descriptive, cognitive, prescriptive and computable. They propose a very heavy mathematical descriptive model. Then there is a small cognitive model based on the idea of design cycles which consist of 5 stages: 1)

Become aware of a problem; 2) Suggest concepts needed to solve it; 3) Develop candidate solutions; 4) Evaluate the candidates and 5) Adopt a solution. At stages 3 and 4 new problems may emerge. Two "levels" of design are observed: the object level and the action (meta) level. They then merge the two models to produce a computable model which uses deduction, abduction and circumscription at the object level and simple deduction based on rules at the action level.

[200] Perry W. Thorndyke and Barbara Hayes-Roth. The use of schemata in the acquisition and transfer of knowledge. *Cognitive Psychology*, 11(1):82–106, January 1979.

"A memory schema... is a cluster of knowledge (a set of concepts and associations among the concepts) that defines a more complex and frequently encountered concept." Interest in memory schemata has been promoted by AI work which seeks representations of knowledge which involve knowledge clustering, such as *frames* [139] and *scripts* [177]. It has also been promoted by psychological research into memory for discourse. Some common features of the many models presented are: "(1) A schema represents a prototypical abstraction of the complex concept it represents... (2) Schemata are induced from past experience with numerous exemplars of the complex concept it represents... (3) A schema can guide the organization of incoming information into clusters of knowledge that are "instantiations" of the schema. This represents a goal-directed focusing of processing by active memory elements... (4) When one of the constituent concepts of a schema is missing in the input, its features can be inferred from "default values" in the schema..." Attempts to formulate general schema theories of memory have: (1) been vague and general and fit any data (2) poorly specified and cannot be used predictively. The schema must be retrieved in order to retrieve a detail that instantiates it. This paper details the costs and benefits associated with the use of schemata in memory.

[201] Kenneth Utting and Nicole Yankelovich. Context and orientation in hypermedia networks. *Transactions on Information Systems*, 7(1):58–84, January 1989.

This article describes WebView, an addition to Intermedia to help in reducing disorientation. The first part is a good introduction to the problems of using hypertext: disorientation and cognitive overload of which the major factors are: determining where a link leads; estimating the size of the web and returning to a previous state. The way other systems help to overcome problems of spatial and temporal context are reviewed. They tried a map with fixed positions,

but it didn't work, and in fact gave up on trying to automatically produce any sort of global map "essentially, the link structure of the web has no inherent correlation with the user's concept of how documents are related." Such maps may have to be created with human intervention. The system they came up with is a graphical display which: (TEMPORAL) lists documents recently visited and (SPATIAL) lists documents connected to the current one. This is a good paper, but the results aren't earth shattering.

[202] Willemien Visser. More or less following a plan during design: Opportunistic deviations in specification. *International Journal of Man Machine Studies*, 33(3):247–278, September 1990.

"Design consists in transforming a problem representation into another: it always starts with 'requirements' and produces 'specifications'". "The plan which guides the activity is used in an opportunistic way, that is, only when no more opportune actions arise". Actions are selected first by cognitive cost (e.g. schemas are cheap, easy to access information) and also by importance (e.g. fixing omissions, importance of object). Processing of a component X may lead to the activation of a component Y because of a relationship between the components: analogy, prerequisite, interaction, opposites. "Acting according to a plan is a concept-driven activity. Starting from a goal imposed by the plan, the engineering will look for the information needed to achieve this goal. However, most deviations observed stem from data-driven processing, such as starting from information which the engineer has at his disposal and which allows a goal not imposed by the plan to be achieved." "If the design activity is opportunistically organized, a system which support — and therefore imposes — a hierarchically structured design process will at the very least constrain the designer and will possibly even handicap him".

[203] Richard C. Waters. The programmer's apprentice: A session with KBEmacs. *IEEE Transactions on Software Engineering*, 11(11):1296–1320, November 1985.

The most important aspects of machine assistance is "shared knowledge" with the human: in KBEmacs this takes the form of a set of "cliches". KBEmacs does simple program reasoning, for Lisp and Ada, and cliche combinations. A "plan" is used to represent the structure of a program and knowledge about cliches. Cliches are operated on using a primitive, informal mix of Larch-ish keywords and Z schema calculus operators. The system has "constraints" in the form of Ada functions (an interpreter is used, I think) which work directly on the internal plan representation. The user can edit the resultant code, but the systems has to analyse this and generally

loses track of the cliches that was used. The implementation is a 40K Lisp monster. KBEmacs has been used to construct programs more than 100 lines long. It seems to me that upstream design does not have such a set of cliches. Upstream design is a process of coining cliches, and then seeing if they are useful. Future versions of KBEmacs will replace the "plan formalism" with the "plan calculus" allowing general purpose automated deduction.

[204] Richard C. Waters. Program translation via abstraction and reimplementation. *IEEE Transactions on Software Engineering*, 14(8):1207–1228, August 1988.

A translator from Cobol to Hibol (Satch) and from Pascal to PDP11 (Cobbler) are discussed. Both produce abstract program descriptions before doing translation. Satch is limited and Cobbler does not exist. In Cobbler, design decision inferred from the program are represented in terms of transformations. The KBEmacs plan representation, and not the plan calculus, is used.

[205] Richard C. Waters. Automatic transformation of series expressions into loops. *ACM Transactions on Programming Languages and Systems*, 13(1):52–98, January 1991.

Waters considers the restrictions necessary on the language of streams to allow transformation into efficient looping constructs, considering both Lisp and Pascal. A sequence expression is optimizable if and only if: (1) It is a straight-line computation; and (2) It is statically analyzable; and (3) Every procedure called by it is preorder; and (4) Every non-directed data-flow cycle in it is on-line. Series functions can be divide into three categories: collectors, which compute non-series values from series; scanners, compute series from non-series values; and transducers, which computer series from series. "In situations where optimization is impossible, it is usually better to represent a sequence as a vector or list than as a series." In the transformation process, loop fragments must be represented and then combined. The representation has the following parts: inputs, outputs, local variables, labels, loop prologue, loop body, loop epilogue.

[206] Dallas E. Webster. Mapping the design information representation terrain. *IEEE Computer*, pages 8–23, December 1988.

Downstream software development (code, documentation) is rather formal and stylised, whereas upstream development is informal and fuzzy. "It is not enough to capture data; one must capture it effectively so it can be interpreted and used." Good bibliography.

[207] Peter Wegner. Varieties of reusability. In REUSE83 [164], pages 30–44. Reprinted in [66].

Broad but shallow survey of reusability

[208] Mark Wesier and Ben Shneiderman. Human factors of computer programming. In Gavriel Salvendy, editor, *Handbook of Human Factors*, pages 1392–1416. Wiley, 1982.

Unexciting.

[209] Charles Wiecha and Max Henrion. Linking multiple program views using a visual cache. In *INTERACT87*, pages 689–694.

A visual cache is a pop-up panel which displays a limited amount of information for a short time. The cache reduces the number of times attention must be shifted between windows. The number of windows in a cluster is reduced by eliminating those that provide only the information now in the cache. As a result, disorientation is reduced by allowing attention to focus on the task at hand rather than on the details of navigating through a complex set of displays.

[210] David S. Wile. Program developments: Formal explanations of implementations. *Communications of the ACM*, 26(11):902–911, November 1983. Also appears in [3].

Transformations accomplish two tasks: implementation and optimisation. Current problems with the transformational approach are: * building a library of transformations * indexing transformations * validating transformations * verifying enabling conditions for transformations * automation * describing design decisions * scaling up. A development structure is built simultaneously with program transformations. This is done in the POPART system. The development structure has its own language, independent of the program itself, and is built using the Paddle language. The development structure must record both decisions and motivations (goals). Goals must be structures, either a priori or a posteriori, with structures such as "divide and conquer". The system is manually driven.

[211] Jeannette M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–24, September 1990.

A formal method should possess a set of guidelines or a "style sheet" that tells users the circumstances under which the method can and should be applied as well as how it can be applied most effectively. Formal methods provide frameworks within which people can specify, develop and verify systems in a systematic, rather than ad hoc, manner. In practice, you must usually deal with incomplete

specifications... any incompleteness in the specification should be an intentional incompleteness... any intentional incompleteness can be captured succinctly as a parameter in the interface.

[212] Terry Winograd and Fernando Flores. *Understanding Computers and Cognition: A New Foundation for Design.* Ablex, 1986.

Knowledge is *always* the result of interpretation, which depends on the entire previous experience of the interpreter and on situatedness in a tradition. If we begin with the implicit or explicit goal of producing an objective, background-free language for interacting with a computer system, then we must limit our domain to those areas in which the articulation can be complete (for the given purposes). A design constitutes an interpretation of breakdown and a committed attempt to anticipate future breakdowns. For a typical complex computer program, there is no intelligible correspondence between operations at different levels... Furthermore, in going from level to level there is no preservation of modularity. There remains the logical possibility that a computer could end up operating successfully within a domain totally unintended by its designers or the programmers who constructed its programs. The key to design lies in understanding the readiness-to-hand of the tools being built, and in anticipating the breakdowns that will occur in their use. A system that provides a limited imitation of human facilities will intrude with apparently irregular and incomprehensible breakdowns. There are surprisingly few basic conversational building-blocks (such as request/promise, offer/acceptance, and report/acknowledgment) that frequently recur in conversations for action. A systematic domain is a structured formal representation that deals with things the professional already knows how to work with, providing for precise and unambiguous description and manipulation. The critical issue is its correspondence to a domain that is ready-to-hand for those who will use it. The computer is ultimately a *structured dynamic communication medium* that is qualitatively different from earlier media such as print and telephones. Communication is not a process of transmitting information or symbols, but one of commitment and interpretation. A human society operates through the expression of requests and promises among its members. There is a systematic domain relevant to the structure of this network of commitments, a domain of 'conversation for action' that can be represented and manipulated in the computer. In all situations where systematic domains are applicable, a central (and often difficult) task is to characterize the precise form and relevance of the domain within a broader orientation.

[213] Niklaus Wirth. *Programming in Modula-2.* Springer-Verlag, 1982.

[214] D. D. Woods. Visual momentum: A concept to improve the cognitive coupling of person and computer. *International Journal of Man Machine Studies*, 21:229–244, 1984.

> The "getting lost" and "keyhole" phenomena... do not represent human limitations... Across-display processing difficulties are the result of a failure to consider man and computer together as a cognitive system [91]. The advantage attributed to parallel over serial data presentation is based on the characteristics of human perception and attention, rather than the mode of data presentation. When display system structure provides a visual frame of reference that describes the relationships among data points as well as the data points themselves, the viewer's attentional mechanisms can identify highly "informative" areas (high visual momentum). When no perceptual clues are available... the user must rely on other mental processes, of more limited capacity. Memory problems are not the cause of performance difficulties like "getting lost" phenomena; instead they are symptomatic of the mismatch in the man-machine cognitive system represented by low visual momentum. Consequence of low visual momentum can include: (a) cognitive tunnel vision; (b) impaired ability to locate "important" data; (c) getting lost in display networks; (d) memory bottlenecks due to increase in mental workload; (e) decreases in problem-solving performance. Visual momentum can be increased by using perceptual context to help the user construct and maintain a cognitive map, or schema of the data structure — the spatial framework reflects the semantic structure. Mechanisms are: long shot (overview), perceptual landmarks, display overlap (physical and functional), spatial representation, spatial cognition. The latter uses maps, i.e. analogical representations where all points are simultaneously available. This equiavailability principle is shown by (a) the ability to generate specific routes (b) the ability to traverse new routes (c) orientation abilities. Route knowledge becomes available in parallel, not because it's simultaneously available in the user's short term memory (a catalog of itineries) but because analogical representation supports a route generation process (in the sense of a mental skill). Memory bottlenecks occur when there are problems in the route generation mechanism; for example, because the user has no perceptual or internal map. Ergonomic guidelines on the design of computer displays generally attempt to ensure that human sensory limits are not strained. However, human performance problems in display systems such as the getting lost phenomena and cognitive tunnel vision demonstrate that the potential to see/read data does not guarantee successful user information extraction.

Notes about automating software design

Tim Gleeson

Aug 30th 1990

1 Basic terms

There seem to be two fundamentally separate issues:

- develop a cognitive model of the design process (perhaps just in a limited domain) by analysing design histories;
- provide machine support for this cognitive process.

The former seems to be a far larger problem than the latter.

I think "Automated method of extracting design strategies from recorded design histories" is an even harder problem. It presupposes:

- design strategies have a particular form
- we know that form
- features of that form can be readily identified in recorded design histories.

The tractability of the problem will depend on the size of the domain we are going to consider. We may be able to automate some work in small domains but not in large ones. In very small domains, of course, it is easy; for example, in the world of regular expressions it is trivial to automate the translation to finite state machines.

Tactics and strategies

A distinction between design *tactics* and *strategies* has to be made, as they fulfill different tasks. Guindon's model of design identifies "design meta-schemas", which are strategies and control the overall design process, and tactics based on specific knowledge and experience which he calls "design schemas". Design schemas are keyed by specific design situations but meta-schemas are relatively independent of the domain.

Perhaps design schemas can be identified, stored in some sort of database and keyed for reuse. However, I'm unsure that these are the most important elements in design. Perhaps meta-schemas, procedural aspects of design, are more important,

for example, meta-control: I've worked long enough on subproblem A, its time to do some work on subproblem B. These could be stored but they are really aspects of design that are *used* (all the time) rather than *reused.*

Domain specific knowledge may be amenable to storage and reuse, noting that this is probably only possible for small domains. However, we know that we can solve problems even if we know little about the domain. Of course, it takes us longer because we have to *discover* for ourselves much about the problem and domain.

Meta-schemas

Guindon[8.1 and elsewhere] emphasises that two fundamental problems in design are:

- lack of specialised design schema
- lack of, or poor, meta-schema

The latter is a planning problem. Machine help could be provided by a system which encourages users to separate issues, prioritise them (according to importance), allows them to explore multiple-levels of the problem, reminds them of outstanding issues etc.

Design schemas

For the former, we need a better understanding of how specialised design schemas are employed, i.e. what makes a design schema useful? It is unclear to me how much of this could be automated. (Guindon suggest automation in 8.3) Stored design schemas could be keyed and matched on an abstract description of their applicability. However, suppose the design schema retrieved is new to the designer: the designer now has 3 things to understand:

- the original problem
- the retrieved design schema
- how the above two can be related to each other

If the designer can understand all these, and express the relationship in a machine processable way, then the machine can use the design schema to suggest a solution plan to the designer. An essential aspect of reusability is parameterisation: what this means here is that the designer must parameterise the design schema with the particular details of the problem. However, part of the power of a design schema is that it is internalised (in the designer) and provides an essential framework in which the designer can hang tentative, intermediate solutions and ideas at a low cognitive cost. It is well known that experts in a field can perceive and remember far more than novices, because they have an internalised structure on which to hang the new information; novices have to build that structure. In order to make machine

assistance useful, the machine would have to perform this function for a designer not familiar with the design schema.

How do we identify design schemas? Observing programmers may give us more information about meta-schemas than about design schemas. In order to find design schemas we should perhaps look in "Collected Algorithms from the ACM" or IEEE SAC. It might be interesting research to see how we can convert an algorithm or a data structure (in general, an abstraction) into a design schema and eventually a machine usable design schema.

At what level in a design should an attempt be made to pattern match in a database of existing design schemas? Probably at the highest, fuzziest, most upstream point possible; then previous ideas and experience can be immediately used (perhaps by analogy) both to further understand the problem and also to start work on solutions.

How good does a match for a design schema have to be? Finding even a roughly similar problem in the database would be useful because it is always easier to adapt and edit an existing design than to start a new one from scratch. (Question: does anybody write a "makefile" completely from scratch?)

2 Understanding the problem

What does "understanding and elaborating requirements" mean? The former means building a mental model, structuring information, generalising. The latter means specialising, filling in omitted, elided or "obvious" information.

Understanding the problem is very important. It consists of forming mental models. This can be aided by simulation of the model, finding difficulties with them and revising the mental model. However, tools to aid the understanding process can be independent of any attempted solutions: i.e. we can simulate the *problem*.

Stored design strategies could be used as templates to encourage designers to think about the problems involved. For example, we have a problem, we see an analogy to "scheduling systems" then our stored knowledge of scheduling systems encourages us to ask specific questions about the problem: is real-time behaviour important, what is the fairness criterion?

We have a large LTM (long term memory) model of the problem, and we use STM (short term memory) to focus on a portion of it. Potentially a machine can store and focus on the whole, or much more, of the problem. We must elaborate our LTM model to the machine.

Guindon asserts that designers move frequently between different levels of abstraction. Why? One possible answer is that people are using the various design schemas that they know, which will be at various levels of abstraction, to investigate *any* area of the problem or solution. This is not so much top-down-design, bottom-up-design or middle-out-design but familiar-unfamiliar-design.

Guindon[8.1] says that a crucial part of design is the simulation of aspects of the problem and solution. However, this is both mentally taxing and not well supported by pencil and paper. Two solutions are possible:

- provide better simulation tools, particularly those which display temporal and dynamic issues
- help users to better design "aspects" which are easier to mentally manipulate

3 Limitations in cognitive ability

Because of limitations in working-memory (STM), only a fraction of the internal model of a problem can be focussed on at any time. A goal of a software support system must be to encourage users to:

- built mental models that can be easily split for focusing purposes
- focus on appropriate (easy? crucial?) fragments.

gIBIS (TOOIS 6(4):303–331 October 1988) forces a strong separation of *issues* Can a system be built which forces strong separation of levels of abstraction? Such a system would be based on information limitation. Some of the metrics produced by the CIA system (Belanger et al 1990) might be useful: for example, the ideas of weight and cross-coupling between nodes. These indicate how connected (or conversely, how separate) nodes are, and thus how complicated they are to understand. A "weight trigger" might be invoked if a node got sufficiently heavy, i.e. complicated.

In the KMS hypermedia system (CACM 31(7):820–835 July 1988) they have a fixed size for a "frame" (1132 x 805 pixels). The system allows for the display of either a whole frame or two half size frames, but never more than two. They state that scrolling of large frames is an inefficient way of moving through a database and that navigating through a hierarchy of smaller units is better. HyperCard, for the Mac, is also a card-per-screen system. This shows that a system built on information limitation can work. On the other hand, many systems are being built with the explicit goal of displaying *more* information.

It might be useful to perform a literature survey exploring the facilities for abstraction and information limitation in existing design tools. This would also look at systems whose goals are to expand information display. This might take the same form as the article "Mapping the design information representation terrain" but be called "Mapping the design abstraction terrain".

Several systems, e.g. gIBIS (TOOIS 6(4):303–331 October 1988) and NoteCards (CACM 31(7):836–852 July 1988), have discussed the need for *filtering*, i.e. displaying an interesting subset of the information in a database. gIBIS uses "primary and secondary" links and NoteCards suggests embedding a search/query mechanism more deeply into the interface to allow specification of what is interesting. These two approaches differ in that gIBIS is relatively static and NoteCards is relatively dynamic. Neptune (TOOIS 5(2):168–186 Apr 87) seems also to have link types and a link filtering mechanism.

Much computing research can be seen as identifying dynamic structures (e.g. GOTOs and labels) and replacing them with static structures (e.g. WHILE loops) which can be reasoned about more easily. Two interesting issues arise from these

observations: first, can we find a mechanism to statically specify all interesting structures; second, at different times, different subsets of the information will be interesting, thus multiple views need to be supported. The idea of multiple views is probably an extension of that found in databases which I believe, though I'm very ignorant about this, is concerned with record subsetting and intersection. It is an extension because it concerns alternative *structural* views rather than alternative *data* views. For example, when the X window system is installed, there is a directory "X" under which is found the three subdirectories "bin", "lib" and "include". Under the directory "/local" is also found directories "bin", "lib" and "include". The contents of "X/bin" must appear in "/local/bin" and so on. When I am primarily interested in binaries, I will look in "/local/bin" and when I am primarily interested in X, I will look in "X". The strict UNIX filesystem hierarchy cannot express this change in interest and links must be used. A directed acyclic graph (DAG) could be used.

When we develop an upstream system, a lot of our work is in understanding that system. We build mental structures and structures on paper to represent it. Periodically we identify more fundamental constructs than we were previously aware of. These constructs provide simpler explanations of what we already knew and are more useful for predictive purposes. These changes in understanding structure are what a lot of science is all about. A computerised design-aid must support this activity. Simple restructuring takes place when we identify a repeated idea; we can abstract a principle from this, and then instantiate it when needed. It may be that we can identify such patterns by the linkages and clusterings in a machine representation. The machine could then prompt the user to investigate further. The ideas of weights and clustering from CIA are useful and other work is Hutchins and Basili ("System structure analysis: clustering with data bindings", IEEESE 11:749–757 Aug 1985).

4 Miscellaneous notes

Versions are like refinements

Potts and Bruns ("Recording the reasons for design decisions" SE10 1988 418–427) noted the need for the "documentation of the design process" i.e. refinements. This vertical design structure must also be supported by a horizontal design structure which records design *changes*. Individual versions can be supported in the same way as individual design results, but just as important are the reasons for and the descriptions of (deltas) the design changes.

Working sets

An analogy has been drawn between working sets in a VM system and windows on a desk (e.g. WYSIWIS (Stefik et al TOOIS 5(2):147–167 Apr 1987) who cites Rooms (Henderson and Card, Xerox PARC ISL TR, Jul 1986)). This idea needs more exploration than the simple treatment it has been given. Though reducing the

number of *window faults*, as suggested in WYSIWIS, may be useful, we really need to explore the relationship between STM, LTM and the window working set. Does a window working set act as a cache, or as an intermediate level of storage between STM and LTM? What are its properties? Perhaps it has better store and delete access than LTM, but retrieval is much slower? We should start by examining why pencil and paper are useful.

Systems which seek to expand the window working set are:

WYSIWIS

NoteCards with its Browser and FileBox cards

Neptune (TOOIS 5(2):168–186) with its Graph Browser

gIBIS with its text index and "zoomed out" global view

KMS claims it supports "time multiplexing", rather than the conventional "space multiplexing" (lots of windows), because it can quickly switch between nodes. The terms are due to Card (Card et al, "Window Based Computer Dialogues", Proc. 1st IFIP Conf. on Human-Computer Interaction (Interact '84) 1984). These approaches seem psychologically very different. Can we investigate their characteristics?

Hypertext

Hypertext is like machine-code: it is a low-level, efficient, completely flexible *mechanism* for structuring text. However, like machine-code it is too flexible; application specific constraints need to be added to make it more comprehensible. These constraints are domain specific *policies*. [Reference to policy-mechanism separation].

Ordinary text can be compared to a programming language with no branching structures. Adding links and nodes enormously increases the flexibility of a text system, just as GOTOs and labels add to the flexibility of a programming language. However, we soon see the need to restrict ourselves to more controlled structures such as LOOPs and procedures. In a hypertext system, the structures we add will depend on the domain of use. For authoring we would use chapters, sections, footnotes, marginal notes, diagrams etc. In a program development system we would need specifications, refinements, updates, documentation etc.

The notion of typing in hypertext systems needs investigating. This is the same old programming language, operating system argument, e.g. LISP for flexibility or PASCAL for security. Delisle and Schwartz ("Contexts" TOOIS 5(2):168–186 Apr 1987) note that since many documents are tree structured, many hypertext systems give special support to this kind of organisation.

The discussions of where links should be allowed to point to is very similar to discussions held long ago about where labels can be placed and jumps can be made (out of loops? into loops? into procedures?). The answer lies in a better definition of the semantics of the domain in which the system is going to be used.

Research into automating software design: External Design Media

Tim Gleeson

Oct 3rd 1990

Contents

1 Introduction

A goal of the ATR Communication Systems Research Laboratories is to develop technologies for the automatic generation of communications software, for example, tools to assist in the storage and reuse of aspects of software design.

The success of a tool depends on a thorough understanding of the problem it seeks to tackle. Thus it is imperative that we gain a better understanding of the design process. There have been experimental psychological investigations of many aspects of human design, for example [GCK87, GC88]. Humans already use general purpose tools in design, for example, computer editors and filesystems. The tools of pencil and paper and libraries have been used for thousands of years so we should also look at how the various aspects of the human design process interacts with external media. A better understanding of the use of external media will allow us to construct much more useful computer tools to assist in software design.

1.1 Document purpose

The purpose of this document is twofold.

Firstly, to sketch out some of the domain of external design media. Many of the notes here are poorly structured and incomplete, but I hope they will indicate the bounds of the domain. There are many references given which remain to be read.

Secondly, this document provides a proposal for further work. There are many section headings which have no text and others which are very short or lack detail. These are left to indicate the need for work in that area.

1.2 Document structure

This document looks at the following areas:

- The characteristics of the systems involved in the design process, i.e. internal and external memory and the design process itself.
- How and why external design media are used and problems with their use.
- Other cognitive issues.

2 System properties

We must examine the basic characteristics of the various systems that interact when a design is being made. These include:

- The internal memory systems of the designer
- External media, e.g. paper and computer
- The communication channel between the above
- The design process itself

Green, Bellamy and Parker [GBP87] identified a number of properties of external design media ("device language") that have an impact on design strategy. These properties can also be applied to internal memory:

access window This is the "width" of information that can be accessed at any time. For human short-term memory, this is small. Paper is almost completely accessible (it is very easy to move the eyes to a particular part of a page, or to turn the page). For a machine, for example an editor, the access window is an area surrounding the cursor.

viscosity How hard is it to make local changes in what has been stored?

role expressiveness How well does the medium express the purpose of the stored data? Many programming languages do not do this well — much text must be read and then "de-programmed" to infer the intent.

2.1 Internal memory systems

2.2 External media

There are several significant characteristics that external media posses:

- They have an almost unbounded storage capacity
- They are shareable between designers

Paper-based systems have a certain degree of structure that cannot be avoided — the paper must be rolled or cut into pages — this also imposes some constraints on use such as the kind of indexes possible and how rapid it is to access particular items (latency). Computer-based systems are almost completely structure-free, as can be seen in hypertext systems. A structure-free system has penalties as well as gains, however.

2.3 Communication channel

The communication channel between internal memory and external design media consists of the screen, our eyes and ears, the keyboard and the mouse. This channel has limited bandwidth. Much work on human-computer interaction (HCI) has focussed on *interfaces*, for example, the structure of menus, whether scrolling a large text is better than paging through it or the wording of error messages. Though this work is useful, it is rather shallow; we need a much deeper understanding of the basic possibilities of interaction with external media.

2.4 Design process

There have been experimental psychological investigations of many aspects of human design, such as the use of design schemas (domain knowledge) and meta-schemas (design strategies), for example [GCK87, GC88] and also the different processes used by novices and experts.

3 Exploitation of external design media

We will look at the following issues related to the use of external design media:

- data storage (kind of data and its structuring)
- use of the data
- problems in use of the data

At a certain level of abstraction we can equate storage of information on paper and storage of information in a computer: they are both external to the user. At this level, questions about user interfaces become meaningless. Though computers can do much more than pencil and paper can, we still don't know what we can do with pencil and paper. Thus instead of starting our investigations with human-computer interactions, we can start by investigating human-paper interactions. This has a number of advantages:

- Paper is a very simple technology
- It is very well developed, having been used for thousands of years

3.1 Data storage

3.1.1 Kinds of data stored

3.1.2 How we access information

To be useful, externally stored information has to be *accessible* and this means organising it. For paper, there are many different kinds of organisations, including desks, filing systems, libraries, little sticky pieces of yellow paper glued in seemingly random profusion around an office. In finding information we use search processes and various forms of *indices*[1]. Many of these forms of organisation include indices at different levels of detail.

Of particular interest is how this media is accessed as an *extension* of the human mind, for example, we know in the design of computer filing systems that the idea of a *root* or origin is very important. Here, the root (or the roots) to the information stored externally must exist in the human mind, and perhaps a number of levels of indices and other primary access mechanisms exist there. This issue is discussed more in Section 4.1.

3.2 Data use

External media are used for a variety of purposes including:

- short-term and long-term storage

[1] I seem to use the word "index" for all access mechanisms.

- searching
- learning
- reference
- visualisation
- simulation

We can say that the properties of the media are being *exploited.* For example, in visualisation we exploit the two-dimensional character of paper (a characteristic not found in our heads) to explore aspects of a system. We could also build three-dimensional models and make films (animation), again exploiting the characteristics of the external media.

We can consider the differences between paper storage of information and computer storage of information. Some of the capabilities of computers merely advance what could already be done with paper and some of them create entirely new opportunities. In some cases the computer is a worse tool than pencil and paper, for example, in reading experiments [HH88][2]. These new capabilities of computers can considerably change the habits of users.

For example, in the field of *authoring*, computers allow the rapid movement of large quantities of text, something which could not be easily done with pencil and paper. This meant that pencil and paper users had to:

- organise their ideas
- express those ideas as blocks of text.

The text-shifting capabilities of computers has meant that this order of working is not so necessary now; computers have made the medium of text far more manipulable. But this may not be a good thing[3]; blocks of text can be seen as the *realisation* of ideas; it is the ideas that we wish to move around, but to do this we must mentally translate the blocks of text back to ideas, and this can be seen as *reverse engineering*, which I think should be avoided. However, computers have completely changed my mode of working and I now find it hard to write without the aid of one.

This can also be seen in fields other than that of authoring: syntax-directed editors and interactive debuggers have made the medium of programs much easier to manipulate. It is now very easy to *write* the text of programs but these systems do not encourage programmers to *design* programs. A friend close to Apple asserted that the Macintosh operating system was not designed, rather, it was "debugged into existence".

[2]Why do programmers continue to use printed listings of their programs unless they have some properties which are better than VDU listings?

[3][Haa87] cited in [HH88] discusses this issue.

There is something about the almost limitless flexibility of such systems that gives their users a sense of unease and a craving to get something expressed, something stable and concrete, as soon as possible. This effect has also been observed within the design process itself; [GCK87][4] observe that the early adoption of kernel solutions to problems seems almost universal. They conject that sophisticated software tools might allow this to be delayed and the problem space to be investigated more. I also hope this is the case, but it will require a different kind of software from that which we have at the moment, which seems to encourage premature decision making.

3.3 Problems

We must examine the problems and difficulties associated with using external media and the communication channel which links them with the human memory and human design process.

3.3.1 Channel and cognitive capacity

The channels carrying information from the external world to the internal world are strictly limited in capacity as is the human ability to use that information. This means we cannot observe all the external information all of the time. Thus we must multiplex these channels. For example, a desk is a reusable resource: today it holds information about subject X, tomorrow it will hold information about subject Y.

Computerised "information tools" take varying viewpoints on how much information to display and what information to display. These polices can be summarised as: random, unconscious and conscious.

Many hypertext systems try to present as much information as possible to the user, and this means more screens and bigger screens. The KMS hypermedia system [AMY88] provides (basically) two cards per screen, thus a limited amount of information. However, they assert that their system supports *time multiplexing*, [CPF84], of information (because cards can be accessed rapidly) rather than the more conventional *space multiplexing* (where many cards can be simultaneously displayed). These are completely different ways of interfacing the external design medium to the user and this interaction deserves investigation. However, neither of these approaches makes any policy decision on what information to present to the user, or provides tools for the user to explicitly make such choices. These tools can potentially provide rapid access to large information bases, but by failing to appreciate limitations in cognitive ability and the channels of communication, they may be nearly useless.

The tradeoff between volume of information, and speed of access to it via narrow communication channels has been recognised [HC86, CH, SBF+87]. They notice that, like paging in virtual memory systems, human design activity seems to be centered for long periods around *tasks*, with occasional switches between tasks. When

[4] And other papers as well.

performing a task, the user will concentrate on only a small fraction of all the information available. This locality of reference to information can be used to speed up task switching if large segments of task context (perhaps several windows) can be switched together.

In a virtual memory system, we would like to switch between the working set of pages of different tasks, here we wish to switch between the window working sets of tasks. They call these segments *rooms*, [HC86], each room holds the context for a given task. However, this approach only helps to tackle the problem of access to large volumes of information, it does not solve it. It is a two-tiered approach: windows within rooms; the EMS system[5] has pages in hierarchical chapters.

This approach is based on two simple cognitive observations:

1. Speed of access to information decreases rapidly with the volume of information.

2. Locality of reference to information when performing a particular task.

It is unclear whether this principle of task organisation is basic to human activity or is an artifact of current tools, as, for example, are the current modes of authoring and system construction described in Section 3.2.

Another view is that the only way to tackle the problem of rapid access to large information base is by a careful and rigorous discipline of conscious, constructive abstraction of the information. Each abstraction should be small enough to be comprehensible. However, we must be careful here, the exercise should be to reduce *cognitive* complexity and not necessarily *system* complexity, though the two are of course related.

3.3.2 Context

When moving through a large base of information, the question "where am I?" often appears. Poor structuring of the information within a design tool, or poor indexes into it lead to the problem which has been called "disorientation" and "lost in hyperspace", [Ber88]. This problem seems to be specific to hypertext systems and does not appear in paper systems to any great extent. Perhaps this is because we cannot travel very fast within paper systems, or perhaps it is related to the kind of structures found in paper systems. Why do we not get lost in the inner-space of our own memory systems, which must be far more complex than hypertext systems, and which we can navigate through much faster? Again, we must examine the relationship between internal and external design media if we are to build better tools.

More fundamentally, we must ask why we want to know where we are. I think knowing where you are means knowing where you are near to, and how the place you are at relates to those other nearby places. Getting lost is symptomatic of not understanding the region you are in and its relations to nearby regions. This is particularly the case when the region is densely connected. The problem here is poor

[5] According to [UY89], but see [Fei88, FNvD82]

abstraction — we should only ever need a small context to work in, i.e. we should build systems which are tightly compartmentalised and loosely connected. The only connections we should need to understand are those of abstraction (what function is this system implementing?), refinement (how is this system implemented?) and perhaps change (how did this system get to be like it is?)

[UY89] have identified *spatial* and *temporal* context. We refine and extend their ideas of context as follows:

where we are, our current position (spatial)

how we got there, and where else we could have gone (spatio-temporal)

why we wanted to get there, a higher-level plan

The idea of "task" can be seen as incorporating the context we need to work in. The unconscious locality of reference to information, as seen in infrequent task switches, must be supplemented by a consciously structured world. However, we don't want to structure the world too much because this will prevent serendipitous work which requires looser, ad-hoc, vague connections.

3.3.3 Moving through information

There are several ways of moving through a large information base. The conventional model of paper books with linearly arranged pages is obviously extremely profitable. An even older model, recently revived, is that of linear scrolling. Node jumping has been made popular by hypertext systems and folding editors provide another metaphor.

There are arguments about which of these are the most effective, but I think these are really issues of user-interface and are not deeply significant from a psychological viewpoint.

4 Cognitive issues

4.1 Building a supersystem

One of the reasons for using external design media is that they can potentially store a large volume of information. However, there are problems with how this is structured, how it can be displayed, and covering both of these, how it can be integrated with internal structures.

4.1.1 Me and my library

It may be profitable to take a "systems" view of the use of external media.

Let's look at problem solving. There are a number of problems I can solve if locked up in a room with no external media. However, limitations in my

- cognitive ability

- knowledge of the problem domain
- general problem solving skills
- memory (for the solution)

mean that the class of such problems is very small.

However, if I am given pencil and paper, some of these problems[6] vanish and the range of problems I can solve increases.

If I am further locked up in a well-stocked library, with a pencil and paper, then the class of problems I can solve expands enormously. I have not changed. The library has not changed. The combined system of me and the library, however, has far more potential than either of us alone.

When we discuss systems, we must carefully identify their boundaries. We must carefully avoid associating the behaviour of a system merely with the behaviour of its components. Thus I would thus like to consider "me-and-my-pencil-and-paper" and "me-and-my-library" as being new systems. The latter is constructed out of the components "me" and "my library", but it is an entirely distinct and new system. It deserves study in its own right.

This study must explore a number of issues:

- the nature of the component subsystems, both the human (is he an expert or a novice?) and the external component
- the nature of the supersystem we are going to build
- how the properties and connection of the former lead to the latter — an issue of refinement

I have suggested thinking about a new, combined system, a supersystem, where "me" is one of the components. The other component could be "pencil and paper" or "library" or "hypertext". In all these cases, the other system is external to the human. Part of this study must be to discover the way the human component interacts with the external component. When we understand more about this, we will be able to design better external components, better than libraries and hypertext systems.

There is no reason why an external system should *reflect* what is going on or being represented in the human mind, a property sometimes looked for in hypertext systems. The external system should *complement* the human mind producing a new, joint supersystem which is more powerful than either of its components.

We discussed the design process in Section 2.4. If we just examine the human-subsystem, we see that the human-design-process can only work on the human-memory system. However, some new, higher-level design process emerges when we build the human-external-media supersystem. This super-design-process must be distinguished from the human-design-process: it works on the joint memory system. Of course, it is entirely dependent on the human-design-process in much the same

[6] And others such as difficulties in visualisation and simulation

way that a Fortran system is entirely dependent on a machine-code system to execute it. It is obvious how to make the distinction in this computing example; we must do the same with the design example.

4.2 Learning

As a system is used, external indices (for example, those in a book or hypertext system) are exchanged for internal indices. This is a dynamic process. As a system is used, it becomes more and more familiar to the user, i.e., more and more of the information (including indices) is internalised. For example, when I first started to use LaTeX I had to refer almost constantly to the contents page and the index of the manual to locate the information I needed. Now, in many cases, I know the information I want, or if not, I know almost exactly which part of the manual to turn to.

The dynamic nature of the interaction between people and paper can be seen not just in the long-term assimilation of information by the user, but also as a medium-term interaction whereby paper can act as a buffer for the short-term memory for ideas which may never become assimilated in long-term memory, for example, a list of tasks which must be completed. A study of the relative characteristics of human memory systems and paper storage systems would prove both fascinating and useful.

5 Related fields

This work is related to several other fields.

Knowledge acquisition

Here we are interested in the acquisition of knowledge by the joint computer-external-media system and not just by the human or just by the computer. This is particularly interesting because the *facts* themselves may already be in the external-media part of the system, for example, if the whole of the Encyclopaedia Britannica was on-line. However, this knowledge may not be in the system as a whole. Knowledge acquisition will thus involve investigating (reading, browsing, studying) the external-media and building indices, which may be both internal and external. As an analogy, we can gain new information from existing facts by thinking.

Miscellaneous

[ERG+89] report on some work [Fur, FLGD83, GL84] which discusses indexing schemes, though mostly keyword based. The idea of latent semantic analysis can be found in [DFL+].

6 Proposal for research

The following issues must be addressed:

- We need a more complete characterisation and comparison of the characteristics of internal and external memory systems. Computing metaphors such as multi-level memory systems, caches and virtual memory systems should be considered.
- We need a more complete description of the current uses of external design media.
- From these two we must develop a better model of external device use. There is one very primitive model that I am aware of, [GBP87], but this simply models human memory as dumping to external memory when it overloads and concentrates on the decoding (parsing) and encoding (gnisrap[7]) of the information stored externally. They do not consider the extension of indices and access mechanisms out of the human system into the external design media, thus creating a supersystem.

There are many other important questions we must ask. Who are we trying to help, experts or novices or both? What tasks are we trying to help them with? What problems are we trying to solve? We cannot yet make firm decisions on these issues, but must wait until we understand the domain better.

Finally, this research would benefit greatly from having contact with a concrete project, perhaps the writing of a program.

7 Notes on sources

The United Kingdom (UK) Medical Research Council (MRC) Applied Psychology Unit (APU) in Cambridge does research into memory and reasoning. In particular they are looking into the interaction of humans and real world objects, including documents [Bar].

The following journals exist at ATR, though many of the collections are incomplete:

- ACM SIGCHI
- Cognition
- Cognition and Emotion
- Cognitive Psychology
- Cognitive Science
- International Journal of Man Machine Studies
- Journal of Experimental Psychology: Learning, Memory and Recognition
- Memory and Cognition

[7] *gnisrap* is *parsing* spelled backwards à l'Algol68.

References

[AMY88] Robert M. Akscyn, Donald L. McCracken, and Elise A. Yoder. KMS: A distributed hypermedia system for managing knowledge in organizations. *Communications of the ACM*, 31(7):820–835, July 1988.

[Bar] Phil Barnard. Research on human-computer interaction at the MRC applied psychology unit. In *Proceedings of the CHI'90 Conference on Human Factors in Computing Systems*, pages 379–380. ACM.

[Ber88] M. Bernstein. The bookmark and the compass: Orientation tools for hypertext users. *ACM SIGIOS Bulletin*, 9(4):34–45, October 1988.

[CH] S. Card and A. Henderson. A multiple, virtual workspace interface to support user task switching. In *Proceedings of the CHI+GI'87 Conference on Human Factors in Computing Systems and Graphical Interfaces*, pages 53–59.

[CHI] ACM. *Proceedings of the CHI'88 Conference on Human Factors in Computing Systems.*

[CPF84] S. K. Card, M. Pavel, and J. E. Farrell. Window-based computer dialogues. In Shackel [Sha84], pages 239–244.

[DFL+] S. T. Dumais, G. W. Furnas, T. K. Landauer, S. Deerwester, and R. Harshman. Using latent semantic analysis to improve access to textual information. In CHI88 [CHI], pages 281–285.

[ERG+89] Dennis E. Egan, Joel R. Remde, Louis M. Gomez, Thomas K. Landauer, Jennifer Eberhardt, and Carol C. Lochbaum. Formative design-evaluation of SuperBook. *Transactions on Information Systems*, 7(1):30–57, January 1989.

[Fei88] S. Feiner. Seeing the forest for the trees: Hierarchical display of hypertext structure. In *Proc. Conf. on Office Information Systems*, pages 205–212. ACM, 1988. (EMS).

[FLGD83] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. Statistical semantics: Analysis of the potential performance of key-word information systems. *Bell Systems Technical Journal*, 62(6):1753–1806, 1983.

[FNvD82] S. Feiner, S. Nagy, and A. van Dam. An experimental system for creating and presenting interactive graphical documents. *ACM Transactions on Graphics*, 1(1):59–77, January 1982.

[Fur] G. W. Furnas. Experience with and adaptive indexing scheme. In *Proceedings of the CHI'85 Conference on Human Factors in Computing Systems*, pages 131–135. ACM.

[GBP87] T. R. G. Green, R. K. E. Bellamy, and J. M. Parker. Parsing and gnisrap: A model of device use. In H. J. Bullinger and B. Shackel, editors, *Human-Computer Interaction — INTERACT'87*, pages 65–70. Elsevier, 1987.

[GC88] R. Guindon and B. Curtis. Control of cognitive processes during design: What tools would support software design. In CHI88 [CHI], pages 263–268.

[GCK87] Raymonde Guindon, Bill Curtis, and Herb Krasner. A model of cognitive processes in software design: An analysis of breakdown in early design activities by individuals. Technical Report STP–283–87, Microelectronics and Computer Technology Corporation (MCC), Austin, Texas, 1987.

[GL84] L. M. Gomez and C. C. Lockbaum. People can retrieve more objects with enriched key-word vocabularies. but is there a human performance cost. In Shackel [Sha84], pages 257–261.

[Haa87] C. Haas. *How the Writing Medium Shapes the Writing Process: Studies of Writers Composing with Pen and Paper and with Word Processing.* PhD thesis, CMU, 1987.

[HC86] D. A. Henderson and S. K. Card. Rooms: The use of multiple virtual workspaces to reduce space contention in a window-based graphical user interface. Technical report, Xerox PARC Intelligent Systems Laboratory, July 1986.

[HH88] Wilfred J. Hansen and Christina Haas. Reading and writing with computers: A framework for explaining differences in performance. *Communications of the ACM*, 31(9):1080–1089, September 1988.

[SBF+87] M. Stefik, D. G. Bobrow, G. Foster, S. Lanning, and D. Tatar. WYSIWIS revised: Early experiences with multiuser interfaces. *Transactions on Office Information Systems*, 5(2):147–167, April 1987.

[Sha84] B. Shackel, editor. *Human-Computer Interaction — INTERACT'84.* Elsevier, 1984.

[UY89] Kenneth Utting and Nicole Yankelovich. Context and orientation in hypermedia networks. *Transactions on Information Systems*, 7(1):58–84, January 1989.

The Z Notation and Software Design

Tim Gleeson

December 11th 1990

Introduction

Z is a formal specification notation developed at the Programming Research Group (PRG), Oxford University. This document summarises Z work which is relevant to the design of software and the storage and retrieval of designs. It is a selection of references from:

[Z:Bowe90] Jonathan P. Bowen. Z bibliography. Oxford University Computing Laboratory, April 1990.

available from the PRG archive server: `archive-server@prg.oxford.ac.uk`. The PRG Z bibliography currently contains more than 300 references, so this document is a very small selection of Z work. The [Z:Bowe90] is the LaTeX cite key which can be used to refer to the above article from the PRG Z bibliography.

General

The following references are general introductions to Z and a related formal method, VDM. There are also some references to refinement which is (or at least was) still actively under theoretical research, rather than development. Refinement is particularly relevant because it formally encodes design decisions.

[Z:VDM90] D. Bjørner, C.A.R. Hoare, and H. Langmaack, editors. *VDM and Z - Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*, Kiel, Germany, 1990. Springer-Verlag.

[Z:Haye87] Ian J. Hayes, editor. *Specification Case Studies.* International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire HP2 4RG, UK, 1987.

[Z:King90b] Steve King. Z and the refinement calculus. In D. Bjørner, C.A.R. Hoare, and H. Langmaack, editors, *VDM and Z - Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*, pages 164–188, Kiel, Germany, 1990. VDM-Europe, Springer-Verlag.

[Z:King90] Steve King. Z and the refinement calculus. Technical Monograph PRG-79, Oxford University Computing Laboratory, 11 Keble Road, Oxford, UK, February 1990.

[Z:McDe89b] John A. McDermid. Special section on Z. *Software Engineering Journal*, 4(1):25–72, January 1989.

[Z:McDe89] John A. McDermid, editor. *The Theory and Practice of Refinement: Approaches to the Formal Development of Large-Scale Software Systems.* Butterworths, London, UK, 1989.

[Z:Morg88d] C. Carroll Morgan. Procedures, parameters, and abstraction: Separate concerns. *Science of Computer Programming*, 11(1), October 1988.

[Z:Morg88e] C. Carroll Morgan, Ken A. Robinson, and Paul H.B. Gardiner. On the refinement calculus. Technical Monograph PRG-70, Oxford University Computing Laboratory, 11 Keble Road, Oxford, UK, October 1988.

[Z:Reed90] Joy N. Reed and Jane E. Sinclair. An algorithm for type-checking Z: A Z specification. Technical Monograph PRG-81, Oxford University Computing Laboratory, 11 Keble Road, Oxford, UK, March 1990.

[Z:Spiv89] J. Michael Spivey. *The Z Notation: A Reference Manual.* International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire HP2 4RG, UK, 1989.

[Z:Spiv90] J. Michael Spivey and Bernard A. Sufrin. Type inference in Z. In D. Bjørner, C.A.R. Hoare, and H. Langmaack, editors, *VDM and Z – Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*, pages 426–438, Kiel, Germany, 1990. VDM-Europe, Springer-Verlag.

[Z:Spiv90c] J. Michael Spivey and Bernard A. Sufrin. Type inference in Z. In John E. Nicholls, editor, *Z User Workshop, Oxford 1989*, pages 6–31, Oxford, UK, 1990. Springer-Verlag.

Methods

Z and other formal methods can be used for design and documentation. The following all describe methods of using Z. This section also contains references to structuring and reuse of specifications.

[Z:Bera88] S. Bera. Structuring for the VDM specification language. In G. Goos and J. Hartmanis, editors, *VDM – The Way Ahead. Proc. 2nd VDM-Europe Symposium*, volume 328 of *Lecture Notes in Computer Science*, pages 2–25, Dublin, Ireland, 1988. VDM-Europe, Springer-Verlag.

[Z:Bowe88c] Jonathan P. Bowen. Formal specification in Z as a design and documentation tool. In *Proc. Second IEE/BCS Conference on Software Engineering*, volume 290, pages 164–168, Liverpool, UK, July 1988. IEE/BCS.

[Z:Brya90] T. Bryant. Structured methodologies and formal notations: Developing a framework for synthesis and investigation. In John E. Nicholls, editor, *Z User Workshop, Oxford 1989*, pages 229–241, Oxford, UK, 1990. Springer-Verlag.

[Z:Garl90] David Garlan and Norman Delisle. Formal specifications as reusable frameworks. In D. Bjørner, C.A.R. Hoare, and H. Langmaack, editors, *VDM and Z – Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*, pages 150–163, Kiel, Germany, 1990. VDM-Europe, Springer-Verlag.

[Z:Lond89] Ralph L. London and Kathleen R. Milsted. Specifying reusable components using Z: Realistic sets and dictionaries. *ACM SIGSOFT Software Engineering Notes*, 14(3):120–127, May 1989.

[Z:Nich89] John E. Nicholls, editor. *Z in the Development Process*, Oxford, UK, June 1989. Oxford University Computing Laboratory.

[Z:Wood89] James C.P. Woodcock. Calculating properties of Z specifications. *ACM SIGSOFT Software Engineering Notes*, 15(4):43–54, 1989.

[Z:Wood89d] James C.P. Woodcock. Structuring specifications in Z. *Software Engineering Journal*, 4(1):51–66, January 1989.

Tools, CASE and PSE

Z has been used in the specification and design of many tools to aid in the construction of large systems. Many of these tools are designed to aid the use of Z itself. The most significant of these are Z type checkers.

[Z:Albu85] Harold Albuquerque. Dynamic cross-referencing for Z design documents. Master's thesis, Oxford University Computing Laboratory, 11 Keble Road, Oxford, UK, 1985.

[Z:Brow90] D. Brownbridge. Using Z to develop a CASE toolset. In John E. Nicholls, editor, *Z User Workshop, Oxford 1989*, pages 142–149, Oxford, UK, 1990. Springer-Verlag.

[Z:Flyn90] Mike Flynn, Tim Hoverd, and David Brazier. Formaliser – an interactive support tool for Z. In John E. Nicholls, editor, *Z User Workshop, Oxford 1989*, pages 128–141, Oxford, UK, 1990. Springer-Verlag.

[Z:Fox87] Catriona Jane Fox. The implementation of a type-checker for Z in a rule-based programming language. Master's thesis, Oxford University Computing Laboratory, 11 Keble Road, Oxford, UK, September 1987.

[Z:Kand84] Javed S. Kandloosi. Formal specification and implementation of a version control database. Master's thesis, Oxford University Computing Laboratory, 11 Keble Road, Oxford, UK, 1984.

[Z:Kary84] Ioannis Karydas. Formal specification and implementation of a compilation manager. Master's thesis, Oxford University Computing Laboratory, 11 Keble Road, Oxford, UK, September 1984.

[Z:King84b] Steve King. Specification and implementation of a Z-schema manipulation tool. Master's thesis, Oxford University Computing Laboratory, 11 Keble Road, Oxford, UK, 1984.

[Z:Reed88] Joy N. Reed. Semantics-based tools for a specification support environment. In *Mathematical Foundations of Programming Language Semantics*, volume 298 of *Lecture Notes in Computer Science*. Springer-Verlag, 1988.

[Z:Spiv87] J. Michael Spivey. Printing Z with LaTeX. Oxford University Computing Laboratory, January 1987.

[Z:Spiv88] J. Michael Spivey. *The fuzz Manual.* Computing Science Consultancy, 2 Willow Close, Garsington, Oxford OX9 9AN, UK, 1988.

[Z:Sufr86c] Bernard Sufrin, James Woodcock, Pavel Grossman, Patick Marriott, and David McGlade. Towards the formal specification of a simple programming support environment. In Darrell C. Ince, editor, *Software Engineering: The Decade of Change*, volume 8 of *IEE Computing Series*, pages 100–114. Peter Peregrinus, 1986.

[Z:Sufr87c] Bernard A. Sufrin and James C.P. Woodcock. Towards the formal specification of a simple programming support environment. *Software Engineering Journal*, 2(4):86–94, July 1987.

[Z:Sufr89] Bernard A. Sufrin. Using the Hippo system. Oxford University Computing Laboratory, 1989.

[Z:Wool87] Simon G. Woolhouse. A Z specification database and support tool. Master's thesis, Oxford University Computing Laboratory, 11 Keble Road, Oxford, UK, September 1987.

Prototyping and execution

Z specifications need not be constructive, thus in general some effort has to be made to execute them.

[Z:Batt86] Peter Batty. The implementation of Z specifications in Orwell. Master's thesis, Oxford University Computing Laboratory, 11 Keble Road, Oxford, UK, September 1986.

[Z:Baxt88] S. Baxter. Executing Z specifications. Research and Technology memorandum RT31/009/88, British Telecom Research Laboratories, Martlesham Heath, Ipswich, Suffolk, UK, 1988.

[Z:Dick90] A.J.J. Dick, P.J. Krause, and J. Cozens. Computer aided transformation of Z into Prolog. In John E. Nicholls, editor, *Z User Workshop, Oxford 1989*, pages 71–85, Oxford, UK, 1990. Springer-Verlag.

[Z:Faza87] Pauline Fazackerley. The development of a system for solving sets of constraints. Master's thesis, Oxford University Computing Laboratory, 11 Keble Road, Oxford, UK, September 1987.

[Z:Grif88] Jacqueline S. Griffin. Formal specification of a prototyper interface. Master's thesis, Oxford University Computing Laboratory, 11 Keble Road, Oxford, UK, September 1988.

[Z:John90] M. Johnson and P. Sanders. From Z specifications to functional implementations. In John E. Nicholls, editor, *Z User Workshop, Oxford 1989*, pages 86–112, Oxford, UK, 1990. Springer-Verlag.

[Z:McNe87] Iain R. McNeil. Rapid prototyping of Z specifications in standard ML. Master's thesis, Oxford University Computing Laboratory, 11 Keble Road, Oxford, UK, September 1987.

Scratch proposal

Tim Gleeson

October 29, 1991

1 Introduction

This note proposes the construction of a design-assistant system, Scratch, for the generation of circular-queue modules. It will investigate the storage of circular-queue designs and the design decisions that need to be made when choosing a particular implementation.

1.1 Background

It has been said, [Wat85], that the most significant advance made in programming was the creation of high-level languages with compilers and interpreters. They are useful for two reasons:

- They remove many machine details and idiosyncrasies and provide a higher level of abstraction making programs more comprehensible by delegating low-level decisions[1] to a compiler (or interpreter).
- They generate nearly optimal code[2].

The work on optimizing compilers seeks to both make such code *more* nearly optimal, and to extend the range of source language constructs which can be optimally translated. For example, [Wat91] shows how (a subset of) the very expressive language of series expressions (also known as streams or sequences) can be automatically translated into more efficient, but less expressive, loop constructs.

We can broadly consider programming language design and compiler work as encompassing three tasks:

Relationship Rigorously defining the relationship between two languages: source and target.

[1] Lots of scope for enumerating these.

[2] This depends very much on how you define "optimal". It could be relative to some absolute machine time and space definition of optimal, or relative to what a human could produce. That different compilers produce code varying in efficiency by a factor of 10 I don't consider very significant.

Operationality Making this relationship operational, i.e. the system can actually translate from source to target.

Optimality Understanding optimality of translation.

If we remove both constraints of optimality and operationality, then we have a textbook defining the relationship between languages. An example of such a book is Knuth who defines, for example, the relationship between the abstract idea ("notation" or "language") of sorting and many concrete implementations of it.

If we just relax the optimality constraint: that the system must understand the optimal translation path[3], but instead provide a map indicating what paths are, where they lead, and how constraints and functions are balanced and traded off at each turning point, then we reach human assisted compilers, or machine assisted coding assistants if you prefer. We may describe this as an automated Knuth.

Returning to Waters's work on series [Wat91], the theme of this project is not automatic optimization. However, Waters says:

> In situations where optimization is impossible, it is usually better to represent a sequence as a vector or list than as a series.

This *is* the theme of this project: how to decide when and whether to change algorithm or representation when automatic optimization is not possible.

1.2 Examples

Here we give some examples of how design decisions will affect the code that Scratch produces.

If we specify that the element type in the circular-queue is small, then the system produced should directly allocate storage for the elements[4]. If we specify that the element type is large, or if we don't specify it at all, the system produced must use a pointer implementation and call upon a storage mechanism associated with the element type to allocate and free storage for elements.

If we specify that we need a circular queue with a small, bounded number of entries, and that the element type is small, then the system should produce an array implementation — otherwise known as a bounded buffer.

1.3 Design

This text has been deliberately written to indicate that the use of computer design assistants is a special case of the use of any kind of design assistant. Thus, for exam-

[3]This relaxation may be because our definition of optimal becomes more blurred, rather than because it is not possible to calculate. Such blurring may occur if we introduce more and more unstated, external evaluation functions which must be simultaneously balanced.

[4]Though at the abstract level of describing circular queues we need to know very little about the element type (basically, that it has assign and dereference operations) when we build an implementation we can make use of much more information about it. We *are* breaking abstraction principles here, but in a controlled way and only after we have used abstraction to design a clean and comprehensible system; then we can relax abstraction for efficiency.

ple, the phrase "man-machine" communication is avoided, communication "between designers" is used instead.

This project falls into the class of *group design*: a number of active entities cooperatively engaged in a design task. these entities may be humans or computer design assistants. The designers may be functionally specialised. We can stretch the definition a little and even consider libraries and pencils and paper to be special, passive design assistants, because they share many of the same properties of active design assistants.

2 Goals

This project has many goals, but most of them are concerned with investigating and understanding parts of the design process:

I. The nature of design decisions in general, and software system design decisions in particular, and the communication of design decisions between a human and a design assistant.

II. The nature of the knowledge shared by a human and a design assistant.

III. The representation and processing of a detailed model of a narrow domain.

IV. The nature of implicit, unstated and non-functional requirements, their representation, communication and processing.

The central theme of the project is the construction of a design assistant, Scratch. This has a number of additional, well-defined functional goals:

V. The generation of implementations into a number of programming languages.

VI. The ability to incrementally change the requirements input to Scratch.

VII. The ability to operate at, make decisions at and freely move between any level of abstraction in the domain.

VIII. The recording of a design history, but the production of a design rationale (including justification of decisions made), an abstract specification of the software including all non-functional requirements, the software itself and constraint and dependency information for its use in a wider context.

The functional goals require direct mechanisms to implement them, but there are other indirect mechanisms needed:

IX. An operational representation of the many specifications and relations involved in the domain.

X. Parameterisation and transformation between specifications at many levels.

XI. Property and refinement calculi; simulators[5] and program proving.

XII. Search, lookup and pattern matching of specifications.

XIII. Propogation of constraints within and across levels. No "reverse engineering" solutions should be needed.

There are some long-term future possibilities:

XIV. Domain database input and update facilities[6].

3 Discussion

3.1 Goals

Goal I

Communicating design decisions between designers is a major problem in group design, particularly upstream design where these decisions may be vague and tentative. We need to understand more about how design decisions can be communicated across the man-machine barrier and how we can best encourage humans to record design decisions.

Goal II

For cooperation, designers must have some shared knowledge. We need to investigate the shared knowledge that designers have about a domain. A computer design assistant will certainly have to represent shared knowledge.

Computers have precise representations. Humans have vague, unelaborable evaluation criteria. Thus a profitable division of labour would be to have a computer teach it to the client to convince him it is the right thing for the job[7]

Goal III

When designing an abstract data type (ADT), we should design a human readable representation for the ADT, and associated procedures for conversion, which will be useful for debugging. Similarly, Scratch will be dealing with a single domain. We should provide domain specific exploration, particularly animation, facilities.

Goal IV

Scratch may be considered as the middle levels of a very high-level language compiler. For this to work, the system needs sensible defaults and most critically a range

[5]A spreadsheet can be considered halfway between a simulator and a constraint propagation.

[6]In the same way that the literature on container systems gradually gets improved and patched.

[7]Computer as salesperson.

of module parameters (which must be well specified) to control non-functional requirements such as time and space tradeoffs (and inter-operation tradeoffs e.g. for read-mostly systems). Use of this could be in stages:

1. Just accept the default implementation the system gives[8].
2. Observe or predict usage and set parameters.
3. If this module is a bottleneck, use an interactive or off-line session with Scratch to add more and more design decisions to get it to come up with a carefully tailored, fine-tuned implementation.

Many of the "goals" and "evaluation" functions will be constant across a project or even an organisation[9]. For example, the full generality of a circular queue will not be needed in every enterprise. In fact it is rarely needed, except as an abstract exercise. Most enterprises will already have narrowed the range of types they wish to store in it, the range of operations and the frequency ratio of operations and the language they are going to use. With just this default information, and no human assistance, a much more specific and efficient system can be built. We must factor these out and make them globally accessible to use as system defaults[10].

As well as non-functional requirements, we also need to investigate design given different levels of details of requirements.

Goal V

When does domain knowledge move from being programming language independent to being programming language dependent? This question is examined in [Gle91].

Goal VI

Design and programming are iterative activities. It is important that we support redesign while reusing as much of our previous work as possible, e.g., to allow radical shift of high-level objectives while preserving low-level decisions which have not been invalidated.

Suppose our higher-level goals and constraints change after we have produced a package. We want to edit (incrementally change) the existing design rationale and, much more cheaply than the first time, get a program. We do not want to have to enter all of the design decisions again. For example, we made the (big) decision to use a hash table. We then made many little decisions. We then changed our (high level) decision and decided to use a binary tree. We want to salvage as many low level decisions as possible.

---

[8]Presumably after having given it some primitive decisions, such as the fact that your using C. Also, site and project specific decisions (use gcc, calling conventions, when storage is allocated (static, dynamic)) could be recorded and accessed from a database.

[9]This would be a very rich and fertile area for research. When are significant design decisions taken. Many of these high level decisions are taken implicitly and never explicitly. When we choose an algorithm from Knuth we have already made a huge number of decisions which affect our choice.

[10]Maybe like default X resources in .Xdefaults?

Goal VII

Display and movement between any level of abstraction in the domain. Editing[11] should always be performed at the appropriate level. Therefore no reverse engineering. Ease of movement between levels needs a sophisticated editor, like a folding editor, but at a much finer grain. We must be able to move up from the result of a constraint, e.g. 64, to its expression and then out to its subexpressions. Perhaps we can highlight (colour) expandable (and contractible?) text.

Goal VIII

A design or programming assistant will never be able to do everything, except in trivial domains. It will always be embedded in a larger enterprise. It will never create a whole program so it will have to both produce packages and dump all its information, such as dependencies.

Goal XIII

This mechanism is needed to support functional goal VII.

The interaction of constraints at different levels of detail includes upward propagation of low-level constraints and the propagation of constraints within and across type boundaries.

It is necessary for the domain, and all possible decisions, to be completely elaborated if we want to prevent users from feeling the need to alter the output code, and thereby prevent subsequent operability within Scratch.

In such a system we must have access at every level of detail, but we should be given help at that level and understand the repercussions of our decisions (constraint propagation) at higher and lower levels. We must[12] be able to specify a low-level constraint and see its effects at higher levels.

3.2 Other discussion

One of the criticisms of very high level languages is that although they relieve the programmer of much work, the efficiency of the constructs they generate is not transparent[13] It is always important that a programmer be able to think about efficiency issues if he wants to. Scratch offers this ability in that efficiency arguments, tradeoffs and balances are essential elements of the domain model. Many of the design decisions represented in the domain model are of this form.

---

[11] We ought to be able to ban this word. The only activity should be *deciding*, and maybe revoking decisions.

[12] Do this as an example.

[13] Hoare makes a comment on the need for this somewhere.

4 Justification

It is not yet clear how we can structure, access and use design information in a thoroughly understood domain (such as circular queues) let alone in a new, volatile, upstream design situation. The problems and goals we have set in the well understood domain of circular queues are probably too big, but would become impossible in a fuzzy, upstream domain.

Scratch is clearly more restrictive than even a routine design system.

5 Problems avoided

- Machine learning. Computer assisted teaching is difficult, but machine learning is very much harder. Thus we should preprogram the machine with a large database about a field of computing that we do understand well, rather than building a system which tries infer things about domains we don't understand well.

- Deciding what constitutes a "useful" subroutine, cliche or abstraction. This is a non-trivial problem for both high-level (e.g. an inventory system) and low-level (e.g. a hash table) abstractions. We avoid this by choosing an existing, well defined, useful abstraction to model. Some work[14] [Der81] and [Der85, (unread)] exists on generalizing from specific instances to abstractions, but it is still very limited.

- Searching in an ill-defined space, for example in upstream design. The space in Scratch is very well defined, but still large enough to be interesting.

- By choosing a small, well understood domain we can ignore flexibility in the domain database. Though innovations in well understood domains like circular-queues will continue to appear, the domain database will be read-mostly.

- Failure handling. Scratch always has the most general solution available which will meet the given problem constraints — unless those constraints are logically contradictory and unsatisfiable. There are no design failures and no need for redesign.

6 Comparison to other systems

Scratch takes ideas from many areas. Here we list and discuss its relation to some of them.

[14] Maybe this stuff would be useful in DIG?

6.1 KBEmacs and the Programmer's Apprentice (PA)

KBEmacs, [Wat85], is part of the Programmer's Apprentice project. It is a semi-expert system, embedded in Emacs, which helps programmers by allowing them to compose chunks of programs, *cliches* from a database.

- The principal role of KBEmacs is the "construction of a program by combining algorithmic cliches". The principal role of scratch is "the construction of module plans by decision making". Scratch does not have a library of cliches; only one very thoroughly understood and described domain.
- KBEmacs has a rather limited form of cliche instantiation. Scratch will examine more general issues of parameterisation and transformation.
- KBEmacs works at two fixed levels: plans and text. Instantiation occurs on plans. We will allow multiple levels and instantiation at any level, repercussions to be propagated up and down.
- Cliches are represented in a language dependent form. The majority of Scratch will be programming language independent
- KBEmacs only handles [Wat85] "basic design decisions which underly the program". Scratch focuses on design decisions at multiple levels. Scratch will have to indicate subpart, implementation and other relationships between design fragments. The *plan formalism* of KBEmacs does not represent any information about specifications of programs or cliches. It also does not represent anything about the relationships between cliches or between cliches and design decisions. Both these, [Wat85] cites [Ric81], will be fixed by using the *plan calculus.*
- KBEmacs [Wat85] stresses the need for access at the code level as well as at the cliche level. The problem is that there is no strong relation between the two. Scratch emphasises the strong semantic relation between levels — a change at any level is automatically propagated to all levels[15]. In KBEmacs, editing text makes it subsequently harder to work at the plan level. Scratch has as one of its principles the ability to move smoothly between levels. [Wat85] says the next PA will be able to recognise cliches in a text. Scratch, however, as a principle, avoids all reverse engineering.

6.2 PARIS

PARIS,[KT85, (unread)] and [KRT87], is a system which allows for the formally verifiable instantiation of program schemas from a library.

- PARIS uses schema instantiation. Scratch has a more general model of transformation.

[15] An interesting categorisation of design decisions thus emerges: some are local to a level, others cross levels.

- PARIS has a simple library of schemas. Scratch has a detailed,, richly interconnected model of a specific domain.
- PARIS focuses on formal proof of instantiation validity. Scratch will work with a lower degree of formality.
- PARIS requires as input a formal specification of the problem and outputs some code. Scratch focuses on the decisions made while deciding what is needed. Scratch outputs both code and a specification — which should be a summary of all decisions especially including non-functional ones.

6.3 DIG

- DIG is a much larger system than Scratch.
- DIG operates at a much earlier stage of design.
- DIG is concerned with knowledge growth.
- DIG operates over a much wider range of design areas than Scratch.

6.4 Other systems

Other related systems include:

- Program transformation
- More general program generation from formal specifications.
- Formal refinement.
- Libraries of modules, particularly formal libraries, abstract data type (ADT) libraries, and object-oriented libraries.
- Software module interconnection mechanisms.
- Algorithm design systems.
- Languages which allow relations between modules to be stated.

7 Schedule

- Exploration and definition of the domain
- Development of a domain representation
- Exploration of the nature of design decisions
- Development of a decision representation

8 Tough cookies

What if we want a container system that can be concurrently accessed?

References

[DerS1] Nachum Dershowitz. The evolution of programs: Program abstraction and instantiation. Technical Report UIUCDCS-R-81-1011, Dept. of Computer Science, University of Illinois at Urbana-Champaign, June 1981.

[Der85] Nachum Dershowitz. Program abstraction and instatiation. *ACM Transactions on Programming Languages and Systems*, 7:446–477, July 1985.

[Gle91] Tim Gleeson. Programming languages as categories of design decisions. Working notes, 1991.

[KRT87] Shmuel Katz, Charles A. Richter, and Khe-Sing The. PARIS: A system for reusing partially interpreted schemas. In *Proceedings of the 9th International Conference on Software Engineering*, pages 377–385, 1987.

[KT85] S. Katz and K. The. A preliminary report on the PARIS system: An implementation of software reusability concepts. Technical Report STP-114-85, MCC, October 1985.

[Ric81] C. Rich. A formal representation for plans in the programmer's apprentice. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, pages 1044–1052, 1981.

[Wat85] Richard C. Waters. The programmer's apprentice: A session with KBEmacs. *IEEE Transactions on Software Engineering*, 11(11):1296–1320, November 1985.

[Wat91] Richard C. Waters. Automatic transformation of series expressions into loops. *ACM Transactions on Programming Languages and Systems*, 13(1):52–98, January 1991.

The Scope and Extent of Design Decisions

Tim Gleeson

May 1991

Revision : 1.11

Keywords: abstraction, refinement, information hiding, efficiency, design decisions, software reuse, modularisation, module selection, module dependence

1 Introduction

A study of the nature of design decisions is important in a number of areas: for "reverse engineering" applications, e.g. [Big89, BBC+90, ROL90], which requires the *rediscovery* of possibly age-old and implicit decisions in existing code; in maintenance, which relies on the *review* of recorded or retrieved decisions; and in the original design process itself, where design decisions are initially made [PB88] but which, at best, can only be tentative.

Here we examine some of the more basic properties that design decisions have, their *scope* (visibility) and *extent* (lifetime), and the affects these have on program design and redesign.

We regard the construction of a design as the progressive addition of interacting decisions. Structuring mechanisms are required to control both the static scope and dynamic extent of this set of decisions. Programming languages only provide mechanisms for structuring low-level decisions.

The intent of this paper is not to produce a notation for the expression of design decisions and their rationale, but to explore their basic characteristics and thereby to suggest what needs to be recorded.

One of the aims of this paper is to promote discussion on the nature of design decisions and thereby to stimulate the production of programming languages which take conscious consideration of them.

This paper is organised as follows: in Section 2 we introduce the idea of the scope and extent of design decisions and illustrate them with some examples. These are cases of abstraction. However, in Section 3 we show the need to break this principle when selecting an implementation of a generic module. This is explained in terms of the scope and extent of design decisions. More general optimisations are discussed later.

2 Nature of design decisions

[ROL90] have identified some categories of design decisions based on an analysis of programming constructs. These include:

- composition and decomposition
- encapsulation and interleaving
- generalisation and specialisation
- representation selection

Here we try to step back a little and identify some general properties of design decisions which lead to these particular categories. We are particularly interested in the *scope* (visibility) and *extent* (lifetime) of design decisions.

Decisions are made both by humans and by language systems. As we automate design decisions, we tend not to call them design decisions any more. Instead they become "obvious" or "basic techniques". For example, compilation as an automated activity is no longer regarded as a decision making one, but before compilers existed, it was.

2.1 Scope of design decisions

The biggest challenge we face in computer systems design is dealing with the necessary complexity of these systems. We can only design systems if we can fully appreciate the context in which our design decisions are made, that is, the relevance of previous design decisions. What we call "complex" is a system where there are many interdependencies, where each decision we make depends on an unmanageable number of previous decisions. We need to structure our design decisions, and thus the system that we build, so that their interdependence is minimized. This is an exercise in controlled ignorance, we purposefully limit what we need to know to perform a task, and is called *abstraction.*

One way to achieve this is by limiting the *scope* of design decisions. Scoping is a form of conceptual, and in its representation, spatial, separation or hiding of design decisions. Scoping is a form of structuring. Scoping allows us to defer making design decisions without affecting the rest of the design.

2.2 Extent of design decisions

Building a complex system clearly involves the making of a large number of design decisions. Later decisions may depend on earlier decisions. There is some sort of ordering involved. Decision making is a form of commitment. However, commitment is a relative term. As humans we demand the right to change our minds. If commitment means anything, we must interpret it as bounded commitment. Subsequent, dependent decisions must appreciate this. This is what we mean by the *extent* of a design decision: when does this decision have to be reviewed?

Extent expresses the degree and nature of the interdependence (or independence) of design decisions. It finds expression in programming (and linker, and software environment) languages.

When designing systems we regard it as important to be able to easily change decisions: for this we must increase the extent of design decisions by reducing their dependencies. In order to easily make design decisions, we need to limit their scope.

2.3 Examples of design decisions

We will examine a few small examples to illustrate these ideas.

To implement a new type UID for Unique IDentifiers:

decision: use types Integer and Set

category: representation decision

made by: programmer

scope: textually abstracted to this type: invisible outside

extent: until the UID implementation is next edited

To implement type Integer

decision: use two's complement, 32 bit words on a 68000 architecture

category representation decision

made by: compiler

scope: abstracted within the compiler: invisible outside (it may appear in Appendix F of the local guide to the compiler)

extent: until the compiler is changed

2.4 Decision making activities

Can we explain some of the programming language based categories of design decisions identified by [ROL90] in terms of scope and extent.

Encapsulation is not a post-hoc clean-up activity, as suggested when it is described as "collecting together components", but is one facet of decision making. At least this is the case in an "idealised" [PC86] development. Encapsulation follows from the premise that the context in which the original (to-be-implemented) construct sits in is satisfied by its operation. Subsequent implementation decisions do not need to be revealed to a wider context.

Reinterpretation. How do actual decisions become transformed when we subsequently reinterpret the development in an idealised form? This is certainly an important question because we can only *record* the actual development, but are encouraged to *present* an idealised [PC86] development.

Generalisation may be seen as a cleanup activity, later reinterpreted as oracular encapsulation and subsequent instantiation.

3 Generic module selection

One way of overcoming the problems of software reuse is to write generic packages, for example, for Sets, or Stacks or double-ended queues, Deques. These packages are generic because instead of hard-wiring element types, we can parameterise the packages by them. This encourages reuse of the package in different contexts and also serves to make explicit the nature of the dependence of a package on its parameter type.

3.1 Dependence of a generic package on its parameter

A package must make clear what characteristics it requires from its parameter types, the package is then free to exploit those characteristics to the hilt. The less a package demands from its parameters, the more generally applicable it will be. However, the less a package demands to know of its parameters, the less opportunity it will have to exploit their characteristics and produce efficient implementations. There is thus a tension between writing general purpose packages and writing efficient packages.

The representation we choose for a generic type, for example a Set type, can depend on the characteristics of the parameter type. If we know nothing about the parameter type, except that it has an equality test, then we will probably have to build an implementation which stores pointers to elements. If we know that parameter values can fit in a word, or that there are only four values, for example from an enumerated type[1], then we may store the value representations directly. This is a more efficient implementation.

As another example, of algorithm choice, if we have a list of element types to be sorted[2], then knowing nothing about the element types except a simple comparison operation, then we cannot do better than an O(n log n) sort, for example Quicksort or a heap sort. If we know about the structure of the element type, we may be able to use a potentially faster radix sort.

Some languages allow some operations on parameter types, for example ordering relations, to be required, e.g. CLU [LAB+81], Alphard [Sha81] and Clear [BG81, Gog84]. However, the expression of these dependencies should not primarily be to promote and aid separate compilation; this is only a fringe benefit, albeit a considerable one. The main reason is that a program is fundamentally incomprehensible without them. The nature of a unit in a program is only explicable in terms of the forces that created it — the needs it must fulfill and the restrictions under which it must operate. If these change, the unit must change or be replaced.

[1] This is an extremely interesting example. The external representation may use a whole word, but our internal representation may be completely different, using only two bits. Reference [Whi83] for coexisting implementations.

[2] The sort operation comes, implicitly or explicitly, with any generic list type which demands an ordering operation from its elements.

3.2 The generality/efficiency tradeoff

The choice a designer needs to make when designing or choosing a package can be characterised as follows:

- Choose a more general implementation. This may be less efficient than possible, but it means that it is much easier to revoke preceding design decisions upon which this package is dependent.
- Choose a more specific implementation. This will be a more efficient implementation, but it makes it harder to change our minds.

In their discussion of design decisions, [ROL90] note that generalisation and specialisation decisions have some implications in that it is easier to reuse or adapt generalised components though they may be less efficient and harder to test.

The point made here is different: the design decisions to use a particular component has implications for the *rest* of the program, not just that component.

3.3 Design decisions in package choice

There already exist "implementation module choice" systems, such as PSI/SYN [KB81] and the "type-dependent transformations" of [CHT81], but these use a very different source of information on which to base their choice. For example, PSI/SYN may consider expected usage of the module to determine that, for example, a hash-table representation of a mapping will be more efficient than a property list. This information may be informal and non-functional but it is *in the scope* of the decision. "Type dependent transformations" use visible parameter type information to transform implementations. Our concern is with extending the scope of existing decisions so that future ones can be affected.

Suppose we are parameterising a Stack module with type Wombat. At some point, a decision is made on the representation for type Wombat. This decision may be made by:

- the user, who programs the implementation;
- the user, who selects the implementation from a library;
- the compiler.

Once this decision is made, why cannot users of type Wombat, in this case Stack, have access to it? In particular, why cannot the stack module itself exploit Wombat's representation decision in order to produce a better implementation?

The design decision, about the representation of type Wombat, is usually hidden by lexical scoping to prevent dependencies forming on it. If this decision is changed then it affects nobody.

Here we have the key. The Stack module can *use* information about the representation of type Wombat so long as it does not become *dependent* on it in a way which requires human intervention.

Generic packages are not an idle example. One of the major obstacles to software reuse is the identification of an appropriate library module[3]. As we have seen, this choice affects the modifiability of the rest of the system.

The decisions made dynamically by a human and resulting in hardwired, less general, less maintainable implementations, can in some cases be transferred to the computer. We need to design languages where both manual and automatic optimisation by controlled revelation of design decisions is possible.

If we regard compilation as a process of automated design decision commital, then we can see that a careful ordering or compilations, and pushy brinkmanship by later modules, will allow them to exploit the decisions revealed by earlier commitals as they themselves commit.

4 Optimisation

In his landmark paper [Par72], Parnas encouraged us to modularize our systems based upon hiding design decisions within modules. However, he also noted the potential inefficiency of direct implementations.

He gave two examples of a KWIC index system: the first based on a standard control flow decomposition and the second based on design decision hiding principles. A direct implementation of the former appeared to be more efficient, because of lower procedure call overhead.

We can account for this by noting that the control flow decomposition has explicitly used cost of control flow in its design. The second modularisation, based on decision hiding, explicitly avoided making such decisions globally visible. Parnas [Par72] says:

> To successfully and efficiently make use of the second type of decomposition will require a tool by means of which programs may be written as if the functions were subroutines, but assembled by whatever implementation is appropriate. If such a technique is used, the separation between modules may not be clear in the final code.

We are constantly and explicitly encouraged to abstract, modularize and hide design decisions to increase a system's comprehensibility and maintainability. We are also implicitly encouraged to reveal, open up, collapse levels and globally view systems to increase their efficiency. A major part of the design process is in making such efficiency inducing, but global and complicating[4] design decisions. We should not try to suppress either activity. We should seek to understand and control their interaction.

Compilers can directly only scratch the surface of this need. Languages must be designed which allow both for decision hiding in design and controlled decision revelation for optimisation.

---

[3] Cite?

[4] Structure destroying?

Another lesson that we learn is that the relationship of design abstractions to code (or high-level abstractions to low-level abstractions) will not be at all simple. The clean structures we find in textbooks are not the ones we will find in the final code — they may be deliberately complicated. Software engineering in general and "design recovery" research [Big89] relying on the identification of abstractions in code must carefully consider this phase of the design process.

4.1 Applicable optimisations

There are several kinds of optimizations which we can apply to a system. Some of these are familiar from compiler optimisers. We illustrate them here with examples of how they apply to both car design and computer systems design. We have already seen a specific instance of the first one in the form of module selection.

Remove generality

- Because we have a gearbox, the engine only needs to turn in one direction.
- Some applications never remove elements from a set. However, a general Set data type implementation includes a delete or remove operation. We should be able to get a more efficient implementation that does not have such operations.

Remove duplicates

- One battery can be used for all the lights in the car.
- Though an application uses instances of Set of Integer and Set of String it may be possible to satisfy them with one instance of the code of the Set data type.

Multiple use

- The shell keeps the rain off. It can also support the engine, so we don't need a chassis.
- We use a whole 32 bit word in representing an Integer, but we only need a limited range of values. We can use the top bit as a garbage collection flag, and then don't need a separate flag.

Some of these optimisations may seem so profoundly obvious that to even consider mentioning them, let alone omitting them, would be considered the mark of a seriously deranged programmer. But because they are so obvious, it is both hard and necessary to elaborate them. Dijkstra makes similar observations in [Dij72].

4.2 Removing duplicates

Duplication is not a problem in the abstract world of ideas, but it is a problem in real computer systems. However, it usually only manifests itself in non-functional ways. Reasoning about non-functional behaviour at design time is difficult, yet every decision we make has such an effect.

Dijkstra[5] urges limits on the desire to seal off levels to avoid duplication. This is one of the lowest-level, and most essential, kinds of optimizations where global views are necessary.

The somewhat more general problem of removing *redundancy* can be problematic because different applications have different requirements. These may be satisfied by a single general solution, but two specialised solutions may be more applicable. Complex tradeoffs are required.

The need to remove duplicates is satisfied by *virtual systems*, such as virtual memory and virtual machines. Code sharing is an implementation mechanism.

Of course, removal of redundancy does not always follow design, it can lead it. Though the gains of redundancy removal may be non-functional they can be very significant. In the example of car design, one approach would be to design the lights, the radio and the starter motor entirely independently; this may lead them to require different voltage sources. But we don't normally design this way. In anticipation of removal of redundancy we design our components to use the same power source; this may mean the voltage is lower than optimal for the starter motor, and higher than we would want for the radio, but the benefits we get from redundancy removal are far more significant, and thus drive the design process.

A complex feedback of design decisions is needed. This can either be done iteratively, or with foresite in the case of skilled engineers. It poses problems both for design and redesign. This may make later justifications of idealised development harder unless we have some abstract model of the low-level behaviour.

5 Building flexible systems

5.1 Modifiability

Design decisions are not cast in stone, or at least if we want easily modifiable systems they're not. Obviously the modifiability of a system depends on how well set a decision is: if many subsequent decisions depend upon it, then it is well set. A good design will emphasize separability of decisions, but we are often called upon to entangle them, usually in the name of efficiency and in the form of the optimisations just discussed. In these cases we should attempt to automate these optimisations so that earlier decisions do not become too set in.

It is clear that we need to make explicit representations of as much as possible of the implicit, conceptual information that we possess about software systems. As a minimum, we need to express the *code* of a solution. We can also record the

[5] In [Dij72], I think.

specification of a system and the *intermediate* stages of its development. We should further try to record *design decisions* (including rejected ones) made between these stages and also the *reasons* for design decisions [PB88]. We can proceed further and record the *alternative generation methods* and *evaluation criteria* that we use.

Writing down the design decisions themselves and the "reasons" for design decisions[6] is not enough in these cases. For example, it is not enough to build systems which merely encode design decision dependencies, such as utilities like "make" [Fel79]. If human intervention is required, then not enough information has been provided. Complete decision making procedures need to be encoded.

These optimisations themselves can be localised, and thus understood easily, but their effect on the structure of the visible code is complicating. It is better to try to understand the original structure and the complicating decision than to consider the resultant code.

5.2 Example: Stack structure

Decisions appear at every level of a system. For example, can you mix Fortran and C in your system? This depends on link-level decisions made by your operating system, linker and the compilers, e.g. how are functions and data named at this level? what are the stack building conventions? This is an example of a globally implicit decision: all compilers will compile all modules with the same call block layout and with their stacks growing in the same direction. This allows a significant optimisation in that all stacks can be built together on one common stack, and not use a less efficient heap.

However, this decision is made very early, at the time of the compiler's construction, and is very hard to revoke. Since this decision has global significance, if we are going to have flexibility, it should be one of the *last* design decisions that is made. This is probably only possible using a sophisticated linking mechanism.

6 Current and future support

It is difficult, using current languages, to design a generic package with a variety of selectable implementations. This can be because the language does not give the package programmer access to the right information at the right time: the *package* implementation choice needs to be made dependent on the information about the types of arguments, but this information is usually not available to the package and cannot be manipulated.

How do current generic package systems produce code at the end of the day? Some of them effectively use a macro-substitution process: the generic package is macro expanded with its parameter just before compilation. This can lead to multiple copies of nearly identical code for, for example, Stack of Wombat and Stack of Wigit. Another method is to make the connection at link time by establishing pointers to the parameter specific code. Another mechanism is to force objects of

[6] Refs?

the parameter type to carry around, at run time, indications of their type and the locations of their type specific operations.

However, all these methods rely on the abstract composite behaviour of a parameterised module being realised by a corresponding composition of the separate concrete implementations of module and parameter[7]. A viable alternative is to have different module implementations composed in different ways, with and depending on the parameter, but still achieving the same abstract behaviour.

In contrast, run-time decisions, for example the value of a parameter, can be manipulated, e.g. with IF and WHILE statements. However, the only common method of selection before compile-time is with the primitive mechanism of conditional compilation. Conditional compilation can be seen as a very crude and restricted form of generic module parameterisation: the parameter is the conditional compilation flag, e.g. "SYSV" or "BSD". This is the kind of *mechanism* we are interested in, generic module implementation selection based on parameter type, but its linguistic expression is very limited. The distinction between these two kinds of decision making needs to be blurred, but we also need sound semantics for expressing these decisions at all levels.

In the future, a package will be truly generic if it can process and use all the design decisions which precede it. For example, a sorting package will know about quick sorts and radix sorts and will know the information it needs to decide on the most specific implementation. The client should be able to inform it that he does not require stable sorting.

The "package language" needs to be able to say things like:

```
IF    the parameter type can fit in a pointer
THEN  use an inline implementation
ELSE  use pointer indirection
ENDIF
```

Ideally, the THEN and ELSE implementations should come from a more general implementation which is either parameterised by or transformed by that design decision.

One reason why such constructs have not appeared in programming languages is because they add nothing to, functional, "algorithmic success". As an engineering discipline, software engineering must be concerned with non-functional issues such as efficiency and maintainability. When designing systems, we *do* ask these questions, but only implicitly and not on paper let alone computer. We need to express these decisions in an operational form if we are to achieve software reusability — this is because these questions arise every time we do an implementation and must be answered every time. We should encode these decisions and their answers for reuse.

[7]As soon as we give up the idea of any abstraction having *an* (i.e., one) implementation, we must give up this simple idea of composition. A language system should be able to *choose* between the set of implementations possible.

7 Miscellaneous

- If we use automatic module selection, then we need some pretty detailed NFRs specifying the behaviour. My module selection is different (I think) from algorithm synthesis because it is driven by the level of knowledge, c.f. Mostow caches.
- The degree of "globality" of information flow differs between each optimisation.
- Generic packages in C are hard.
- Arguments about transparency of cost of constructs: Hoare.
- Effect of top-down and bottom-up pressures: We need to store 1000000 items; discs behave very differently from main store.
- The transition from structured to flat is important.
- Some programmer design decisions are "globally implicit".
- Other work: Need a Parnas quote on postponement [PS75, Par76]. Maybe also a Lampson "don't hide power" [Lam83] and maybe a Fabry [Fab76].
- The kind of structure-destroying transformations here are potentially globally complicating, unlike other transformations which might be locally scoped [MC85].
- This paper seeks to explain the concepts of level collapsing and information retrieval in terms of changes in the scope and extent of design decisions.
- More on behaviour and subtyping.

A linker has no respect for the abstract structure we have carefully created. By the time it is set to work, it sees only a flat terrain of procedures and procedure references. Our careful structure has collapsed[8]. The linker only includes one copy of a library routine even if it is required many times. This decision is nearly transparent to the user. "Nearly" because it makes the system smaller than it would otherwise be, and this is visible to the user.

Why is abstraction inefficient[9]? How does this manifest itself as "duplication" or "excess generality". Why are these problems?

[8] In this light, the attempts to use hardware mechanisms of virtual memory and virtual machines at the program level can be seen as attempts to preserve the abstract program structure within the machine. However, even here the use of "shared libraries" can be seen as level collapsing. Maybe get a HYDRA reference in here.

[9] It isn't conceptually inefficient.

Revelation and revocation: what, when, to whom

What kinds of "design decisions" are useful to another module? Presumably read-only information. How about "the sequence is ordered"? As another example, a coordinate transformation system can be made much more efficient if it can assume that all its arguments will be normalised before it is given them. *When* does a programmer make this decisions, if at all?

The kinds of design decisions that programmers make when constructing a module may not be the kinds of decisions that a package needs to know. The latter information may only be implicit in the former. An example might be a change of representation for enumerated types which we introduced earlier.

At the level at which a design decision is hidden, the package must be prepared for the revocation of that decision. The package must be prepared for revocation over some well-defined[10] time-span. E.g. "until I am next compiled, I guarantee that type Wombat will fit in 64 bits." This is a very useful timescale: in particular, it allows the compiler-system of the using module to select just one of the implementations that it has.

The compile-time mechanisms needed to deal with this are the same as the run-time mechanisms needed to deal with tagged unions.

These ideas require us to be prepared to make dynamic shifts in our view of what a system is. "Dynamic" can apply over a number of timescales, for example, compile-time, where we are choosing an implementation, and run time, where we inspect union tags.

Delocalising plans

The *delocalised plans* investigated in [SPL+88] are illustrative of the effects of the delocalising optimisations such as we have discussed. However, it is not enough to document these tricky systems, we must show how they were developed. Delocalised plans arise for very good reasons, so we should not attempt to banish them, but rather understand both the nature of the cognitive problems involved in their use and the forces that generate them. It is not so much delocalised plans which need to be recorded, but *delocalising planning* which needs to be studied.

Related work

This work can be seen as related to replay systems [Gol90] which record design decisions which may be reusable after other changes in the system. The difference we are proposing is that such automisation should:

- be in the source code
- be more generally applicable, especially for library modules, by being parameterised

[10] How defined?

References

[Agr86] William W. Agresti, editor. *New Paradigms for Software Development.* IEEE Computer Society Press, 1986.

[BBC+90] David G. Belanger, Ronald J. Brachman, Yih-Farn Chen, Prekumar T. Devanbu, and Peter G. Selfridge. Towards a software information system. *AT&T Technical Journal*, 62(2):22–39, March/April 1990.

[BG81] R. M. Burstall and J. A. Goguen. An informal introduction to specifications using Clear. In Robert S. Boyer and J. Strother Moore, editors, *The Correctness Problem in Computer Science*, pages 185–213. Academic Press, 1981.

[Big89] T. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Computer*, 22(7):36–49, July 1989.

[BSS84] D. Barstow, H. Shrobe, and E. Sandwall, editors. *Interactive Programming Environments.* McGraw-Hill, 1984.

[CHT81] Thomas E. Cheatham Jr., Glenn H. Holloway, and Judy A. Townley. Program refinement by transformation. In *Proceedings of the 5th International Conference on Software Engineering*, pages 430–437, 1981. Reprinted in [Agr86].

[Dij72] E. W. Dijkstra. Notes on structured programming. In O. -J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors, *Structured Programming.* Academic Press, 1972.

[Fab76] R. S. Fabry. How to design a system in which modules can be changed on the fly. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 470–476, October 1976.

[Fel79] S. Feldman. Make — a program for maintaining computer programs. *Software Practice and Experience*, 9(4):255–265, April 1979.

[Gog84] Joseph A. Goguen. Parameterized programming. *IEEE Transactions on Software Engineering*, 10(5):528–543, September 1984.

[Gol90] Allen Goldberg. Reusing software developments. In Richard N. Taylor, editor, *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*, pages 107–119, Irvine, CA, December 1990. Appears as *SIGSOFT Software Engineering Notes* 15(6) December 1990.

[KB81] Elaine Kant and David Barstow. The refinement paradigm: The interaction of coding and efficiency knowledge in program synthesis. *IEEE Transactions on Software Engineering*, 7(5):458–471, September 1981. Reprinted in: [BSS84] and [Agr86].

[LAB+81] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual*, volume 114 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[Lam83] Butler W. Lampson. Hints for computer systems design. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 33–48, October 1983. Also appears in *IEEE Software*, pages 11–28, January 1984.

[MC85] Jack Mostow and Donald Cohen. Automating program speedup by deciding what to cache. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 165–172, 1985.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[Par76] D. L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2(1):1–9, March 1976.

[PB88] Colin Potts and Glen Bruns. Recording the reasons for design decisions. *IEEE Transactions on Software Engineering*, 10(418–427), 1988.

[PC86] David Lorge Parnas and Paul C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12(2):251–257, February 1986.

[PS75] D. L. Parnas and D. P. Siewiorek. Use of the concept of transparency in the design of hierarchically structured systems. *Communications of the ACM*, 18(7):401–408, July 1975.

[ROL90] Spencer Rugaber, Stephen B. Ornburn, and Richard J. LeBlanc, Jr. Recognizing design decisions in programs. *IEEE Software*, pages 46–53, January 1990.

[Sha81] Mary Shaw, editor. *Alphard: Form and Content*. Springer-Verlag, 1981.

[SPL+88] Elliot Soloway, Jeannine Pinto, Stan Letovsky, David Littman, and Robin Lampert. Designing documentation to compensate for delocalized plans. *Communications of the ACM*, 31(11):1259–1267, November 1988.

[Whi83] John R. White. On the multiple implementation of abstract data types within a computation. *IEEE Transactions on Software Engineering*, 9(4):395–411, July 1983.

Decisions in Abstraction and Implementation

Tim Gleeson

May 1991

Revision : 1.5

Keywords: requirements, specification, abstraction, refinement, design, design decisions, software reuse, modularisation, module selection

1 Introduction

"Abstraction" is widely used as a sales attraction for the latest in program design fashion. In this sense it is rather like the "environment friendly", "low in fat" or "high in fibre" labels used in consumer industries, though considerably less measurable than the latter two.

It is vital to recognise that the success of an abstraction depends on its environment. We view abstraction as a compromise between flexible application requirements and implementation behaviours. This search is based on identifying which parts of the interaction are critical to this particular construction. Once a stable compromise has been found it can be presented in a clean, straightforward way [PC86].

We are forced to examine in more detail the nature of application requirements, and in particular the fact that some are more important than others. Understanding the interactions between requirements and possible implementation behaviours, and trading them off against each other, are vital principles in this iterative process.

The kinds of information used in these processes and the kinds of decisions made are examined. Current specification techniques are shown to be inadequate for all but the most idealised development process.

2 What is abstraction?

It is often said that deciding on an abstraction is the hardest part in design, and that subsequent use and implementation is relatively easy. This is an ideal. In practice, we have to both implement and use an abstraction before we can decide what the best abstraction is. This is an iterative process.

Abstraction is an ideal post-hoc explanation mechanism [PC86]. In these cases we know which issues are important and which are unimportant. It's use as a design

tool must be separated from this, however. Design is the *search* for important, stable characteristics in a large, constrained space. Design is the search for abstractions in a space determined by requirements and resources.

The description of an abstraction being a straight line conjures up rather too rigid an image. The most important property is that it is a surface, i.e. it has two sides. Those aspects we are interested in we will certainly want to hold rigid, like a plane, but other aspects will move to and fro as the implementation changes. Imagine holding some parts of the surface of a balloon rigid whilst the pressure inside changes.

This image illustrates the distinction between nearly decomposable [Sim62, Sim73] and effectively decomposable systems [Gle89]. We cannot hold steady *all* aspects of a system and expect to be able to change the implementation. However, we can hold steady all those aspects of interest.

The beautiful stability of the watch subsystems, which Simon discusses, and the watch itself, are not stable under many conditions, for exemple, an inquisitive seven year old, a weighty hammer or high temperatures.

2.1 What can we / should we abstract?

When we abstract, we have to leave something out, but what can it be or what should it be? The issues which are critical depend on the interaction between the system and its environment.

Some abstractional decisions can be made which are almost invisible. For example, it is almost impossible to tell that my integers are represented in two's complement form. Looking at the range of values available and just possibly the timing behaviour under different problems may give us hints. We could X-ray the machine while operating and find more information. However, these are all insignificant aspects of the behaviour of the representation.

For the kinds of interactions we are interested in, the property concerned is not significant. If this is the case, then we can abstract from it. As we have noted, an abstraction depends on its environment. Thus whether something is invisible or not depends on the kinds of glasses we wear. If we wear X-ray glasses, then new things may become visible.

However, I can tell that the integers I declare occupy something called "memory". If I use lots of integers then my system will fail to work in rather unpredictable ways.

It is very hard to model exactly when my system will fall down, in terms of the numbers of integers I use. If possible, we should use a rough model and overengineer. We discuss overengineering as a design tool in Section 4.5.

Absolute space and time are not the only characteristics to be considered when making a choice. A procedure to reverse a list on tape may transiently use a large amount of internal storage, and then return it all. When writing a sort procedure in a functional, list-based language we must consider the indirect overhead of the amount of garbage we generate.

2.2 What makes a good abstraction?

An abstraction is an agreement between two parties about what they both consider to be most important. Both parties will have many other considerations in mind and these will come into play when negotiating the abstraction. Each party is able to offer tradeoffs to the other. Each may be prepared to accept non-ideal characteristics of the other so long as they get something more valuable in turn. This negotiating process requires a detailed understanding of tradeoffs.

An abstraction is the result of the interaction of requirements and available mechanisms. Sometimes one, sometimes the other dominates the process.

We can consider available mechanisms and work back to (the very large range of) supportable specifications. This mode of working is necessary because constraints may well be "how" constraints, e.g. "using the Tortoise Systems Vector Processor" or "using C".

An abstraction depends on the environment in which it is placed. For example, in some environments we don't need to worry about overflow. This is related to the issue of overengineered components and rose coloured spectacles.

We cannot necessarily take an abstraction created for one environment and expect it to work in another. For example, we cannot expect an abstract data type developed for a sequential environment to work in a concurrent environment.

The process we are involved in is taking a domain and producing an abstraction. The abstraction we produce depends as much on the environment/application as it does on the solution domain. In choosing an abstraction we are looking for what is important in this application, and often we don't know this when we start.

- *An abstraction depends on its environment*

2.3 How far can we separate requirements and implementation?

A formal specification is an application of abstraction — synthesis knowledge should be irrelevant to requirements and requirements knowledge should be irrelevant to synthesis. However, because neither requirements nor implementations are entirely fixed, they always have soft characteristics, this ideal is not realisable.

Sometimes it is useful to build a solution to understand the relative importance of requirements[1] and sometimes it is useful to hypothesize the range of requirements[2] when building a solution.

---

[1]The understanding that we gain about requirements is their relative importance in the space of possible implementations.

[2]i.e. client behaviour characteristics (e.g. scenarios). What we are doing is optimising within a set of rigid constraints (the spec). There seem to be 3 classes of issues:

- rigid constraints in the spec
- soft constraints which we then try to optimize
- (nearly) invisible characteristics

Though the ideal is not realisable, it is still useful. After separating requirements and implementations by a specification, we can *reintroduce* some of the elaborations[3].

There is a strong belief in the formal methods community that many of the problems of software production ensue from inadequate specifications[4]. This is surely true. But it is also surely a fantasy that the writing of a specification can be completely divorced from considerations of the implementation of that specification[5]. Unless a specification results from an interaction of requirements and capabilities, we will be incapable of implementing it. A good specification will have hidden most implementation considerations, but only because the specifier has the experience to find a clean abstraction in the messy space of interactions. A poor specifier may produce an implementation which appears to be free from implementation concerns, but this is illusory. A specification must describe all important aspects of the behaviour of a system. Unless a specifier has great experience in the domains of requirements and capabilities he will be unable to predict which aspects of the behaviour of a system will be important. The only way then to remove implementation considerations from a specification will be to *over*-simplify.

2.4 Example tradeoffs

3 Requirements and characteristics

I will use the following terminology:

requirements which may be *hard* or *soft*

specification which meets at least the hard requirements

implementation which meets the specification, but has other observable behaviour

Requirements can be classified in a number of ways:

- *functional* and *non-functional*
- *soft* and *hard*
- *free* and *determined*

Requirements are usually stated beforehand, but sometimes they take the form of an evaluation of determined characteristics. For example, speed is usually a determined characteristic; it is determined by all the other choices we make. We rarely state the speed that a program should run at, but we certainly do evaluate this determined characteristic.

A soft requirement may be "the program must run in under an hour".

[3] e.g. in formal parameters to generic modules?

[4] At any level of abstraction in the development of a computer system.

[5] Writing a specification teaches us about the requirements, but not about the implementation.

Sometimes hard non-functional requirements, such as execution speed, may take precedence over soft functional requirements, for example when a tradeoff between 32 and 31 bit integers makes a significant speed and space saving.

Behaviour can be split into two classes:

- Client behaviour
- Implementation behaviour

The characteristics of these two systems are what we must consider when making design decisions. Some user properties can be made into a functional, though rather rough and abstract. For example, the parameter types to a generic module carry implicit information about client behaviour e.g.:

- has $<$
- has radix form
- has $hash()$
- fits in a WORD

Some of these characteristics are sufficiently functional that they can be used in automatic or semi-automatic implementation choice systems. See section 4.3

Examples of properties of a client are: *ExpectedMax* and *AbsoluteMax* on the number of elements in a storage structure, orderings on the use of operations e.g. "static searching structures" [BS79] where all inserts precede all queries and relative frequencies of operation calls.

Examples of implementation behaviour are: e.g. bottom up requirements such as language structures, RAM, CAS, discs

4 Processes in choosing an abstraction

The variables we work with are not free. They are subtly interdependent by the range of possible implementations. Having fixed one set of variables, our hard requirements, we can alter the set of implementations and see how the other variables change. Many of these will be of great interest to us. They may cause us to alter our set of soft requirements.

There will be a backwards and forwards interaction of flexible requirements and flexible implementations — the balloon will alter and change its shape until the requirements are *adequately* satisfied.

Initial, upstream design is difficult because *adequately* is a subjective term. It usually requires human evaluation.

What is *adequate* depends on what is *possible*, and in innovative design we never know this. Once we have thoroughly understood a domain we know what is possible, and more importantly we know the tradeoffs involved. However, we are then no longer concerned with innovative design.

A very simple model of the process might be as follows:

What I want. Specify the ideal, desired functional behaviour of the system[6].

What I have. This delimits a (potentially) large range of possible implementations. They will probably not meet exactly the desired functional behaviour. They will certainly add extra requirements, specify additional behaviour and suggest possible tradeoffs, e.g. you can't have an arbitrarily large stack, please give me some idea of how big it will be.

Is this OK. Can the caller meet these extra requirements? Do the tradeoffs offered allow an acceptable compromise? Can the caller operate under the suggested limits? The caller is forced to reexamine his requirements, and their relative importance. A greater understanding of requirements is established.

4.1 Criticality

An issue is not critical if it can be dismissed from consideration by a simple, general or overengineering argument. Examples of issues and arguments for their non-criticality are illustrated:

- Is loss of power critical to this abstraction?
  No. If we lose power then the whole program dies so it's not an important issue here.
- Is concurrent access a critical issue?
  No. This module is only used in a sequential environment.
- Is the order of computation a critical issue?
  No. We never need to sort more than 10 items, so order of computation is not critical.

Some issues might be critical:

- Is the speed of the abstraction critical?
  This depends partly on the abstraction itself, but mostly upon the environment in which the abstraction sits. We should

It is genuinely hard to determine if an issue[7] is critical or not. However, it is this determination which is at the core of design and abstraction.

1. Is this issue critical?
2. How can we tell if an issue is critical?
3. If an issue is not critical, we can overengineer

[6]This may involve making the overengineering assumptions we have made, i.e. an order of magnitude (OoM) calculation allows us to assume perfection. In fact, we usually do it the other way round: assume perfection and later do an OoM calculation. If this fails, the issue becomes *critical.*

[7]What is an issue?

4. We need a model of behaviour and tradeoffs by which to evaluate if an issue is critical.

5. This may be a superabstract model and perhaps this is what develops with experience.

N.B. We tend to remember the critical issues about a domain and not the details. I know Insertion sort is O(n) and Quicksort is O(n log n), but with a larger constant factor, but dragging up an implementation is genuinely hard. I have to look in a book, but I know which book and where it is. Thus we do not remember a domain, we remember superabstract, critical issues about that domain.

It is well known that it is very hard to predict which part of a program will cost the most execution time. This illustrates the fact that frequently designers do not understand much of the interaction involved in a program. An implementation is sometimes the only way to establish the relative importance of parts of a program.

4.2 Available optimisations

The design process may in fact be driven by available optimisations, e.g. "multiple use" optimisation of 32 bit words: 1 bit for a garbage collection flag and 31 bits for an integer.

4.3 Automation

What kind of design decisions are automateable? Automation requires having an operational representation of:

- the alternatives available (generated both from user requirements and domain options)
- evaluation criteria
- an optimisation procedure

4.4 Design decisions

Why are some design decisions interdependent? Why do some design decisions delocalise information?

Design decisions appear at at least two different times:

- making
- review

When initially making design decisions we are not always aware which issues are critical and how issues interact. This is a model of the design process as "list issues, evaluate issues, choose alternative" is too simplistic. While designing we learn more about the issues involved and their evaluation. Thus we need a more dynamic issue based system to reflect the learning processes involved in design.

4.5 Overengineering

Let us return to our example of modeling the memory required by integers.

A very naive model would say that I can use 10,000 integers before I have any trouble with memory. I wouldn't even bother counting them in my program because I'm sure I have several orders of magnitude less than this. Overengineering requires us having an order of magnitude of leeway in the characteristics we are considering so that a simple model can be built. If we get any closer to our resource limits than this we will have to start building more detailed models. When we can overengineer, we get simple but adequate models of important (?) behaviour.

This kind of overengineering is a vital concern in many other disciplines, such as civil engineering and cooking. It is vital to know which issues are important/significant and which are not. If we do not make these distinctions then the morass of details will drown us. The conventional model of rigorous system formal specification and formal development does not allow for this. This kind of overengineering is a special case of what has been called *rigorous* rather than *formal* development [Jon86]. In this case, we use simplistic models and a wide margin to convince ourselves that a formal treatment would be possible.

Overengineering gives protection from change, it makes issues non-critical. As an example, we usually use a 32 bit integer[8] where an 8 bit integer would do. Overengineered constraints are ideal candidates for tightening up (putting into the bargaining process) when there is strong pressure from other (maybe non-functional) requirements. The availability of optimisations feeds back and affects requirements, for example, integers in CLU [LAB+81] and some Standard ML [MTH90, MT91] implementations.

Another way of looking at this is to say that if I have lots of memory and few integers, I can pretend that memory doesn't exist. If our representation of integers has lots of bits (32, say) and I use few, then I can pretend[9] that overflow is impossible.

5 Conclusions

What this suggests is that it should be possible to build abstract models of the tradeoffs between hard and soft characteristics in a domain. It should be possible to do this without explicitly considering the range of implementations which lead to those tradeoffs.

5.1 Related work

This work goes beyond the simplistic ideas of issue based information systems [PB88] and explores in more detail the specific character of software design decisions. In particular, we have shown some of the nature of the debate, the issues and the considerations which are needed. [PB88], and others, consider models and tools

[8] But we don't use BigNum.

[9] i.e. use an instantaneous, in the head, order of magnitude, rough calculation to demonstrate.

for design deliberation. These are intended to help in the production and review of solutions and do not directly cover the generation of abstractions. They do, however, require some form of specification as input.

"Issue Based" systems seem useful for recording deliberations based on existing understanding of systems. However, innovative design requires *developing* an understanding of a system. This is a different process.

This is a more specific model of "design schemas" that that presented by Guindon [GCK87]. We have argued that the important characteristics that domain models carry into the design process is their models of tradeoffs between behaviours. We have also more narrowly suggested the value of additional requirements behaviour (e.g. scenarios) in synthesis, and implementation options in specification is their contribution to the evaluation of the relative importance of abstraction characteristics.

[GC88] suggest the use of a library of reusable design schemas which model problem decomposition and merging. This is certainly needed, but example or skeleton solutions only indirectly contribute to problem understanding and problem/solution requirements/behaviour interaction understanding. We can potentially build tools, however, which could indicate and possibly animate the interaction of solution domain properties and constraints.

6 Don't read me

Though abstract data types are not ideal for the analysis of design decisions, for one thing, they are frequently at a lower level than we are interested in and in some cases are subject to automation (and area we will touch upon). However, they do illustrate a number of points clearly:

alternatives Given an issue, how do we decide between the alternatives? Here the alternatives are the (potentially unbounded) set of all possible implementations. Decisions are based on (understanding of) behaviour.

issue What specification are we trying to implement? This is as much a problem as implementation choice.

Also related: SETL choice of implementation by sophisticated analysis of the code.

If we are going to allow the abstraction to shift and change, why then do we bother to have it at all? (balloon).

[Lam83] gives an example of this. "Static analysis", the detection of implicit decisions, can be used to increase performance. A joint decision by the file system designer (to allocate data sequentially on the disc) and the application designer (to access the data sequentially) can lead to substantial performance benefits.

6.1 Psychology and formal methods

Recent work in computer systems design psychology has begun to move away from its origins in programming activities to higher-level or "upstream" research [Gui90a,

Gui90b]. However, for two decades there has been a vigorous promotion of formal methods both in low-level activities [Hoa69, Dij76, Gri80] and for higher-level specification activities [GHW85, Hoa85, GH86, LG86, Jon86, Spi89]. This work has not yet had much influence on computer systems design psychology research [GT91].

Between the extremes of the formal methods strict separation of specification and implementation and the apparent continuum adopted in the psychology research, there is an interesting band of work. The conscious use of abstraction is useful, but the way it is used in reality is very different from the idealised development suggested by formal methods practitioners. Consideration of abstractions provides a valuable tool in understanding and examining some of the activities involved in design.

References

[BS79] Jon Louis Bentley and Mary Shaw. An Alphard specification of a correct and efficient transformation of data structures. *IEEE Transactions on Software Engineering*, 6(6):572–584, April 1979. This is a revised version of a paper from Proc. IEEE Conf. on Specifications of Reliable Software, April 1979. It also appeared as a CMU technical report in December 1978.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[GC88] R. Guindon and B. Curtis. Control of cognitive processes during design: What tools would support software design. In *Proceedings of the CHI'88 Conference on Human Factors in Computing Systems*, pages 263–268. ACM, 1988.

[GCK87] Raymonde Guindon, Bill Curtis, and Herb Krasner. A model of cognitive processes in software design: An analysis of breakdown in early design activities by individuals. Technical Report STP–283–87, Microelectronics and Computer Technology Corporation (MCC), Austin, Texas, 1987.

[GH86] J. Guttag and J. J. Horning. Formal specification as a design tool. In N. Gehani and A. D. McGettrick, editors, *Software Specification Techniques*, pages 187–207. Addison-Wesley, 1986.

[GHW85] John V. Guttag, James J. Horning, and Jeannette M. Wing. The Larch family of specification languages. *IEEE Software*, 2(5):24–36, September 1985.

[Gle89] Tim Gleeson. *Aspects of Abstraction in Computing.* PhD thesis, Cambridge University Computer Laboratory, December 1989.

[Gri80] David Gries. *The Science of Programming.* Springer-Verlag, 1980.

[GT91] Tim Gleeson and Toyofumi Takenaka. The roles of formal specifications in the system design process. In *Information Processing Society of Japan: Spring Conference*, 1991.

[Gui90a] Raymonde Guindon. Designing the design process: Exploiting opportunistic thoughts. *Human-Computer Interaction*, 5(2 & 3):305–344, 1990.

[Gui90b] Raymonde Guindon. Knowledge exploited by experts during software system design. *International Journal of Man Machine Studies*, 33(3):279–304, September 1990.

[Hoa69] C. A. R. Hoare. An axiomatic basis of computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[Jon86] C. B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall, 1986.

[LAB+81] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual*, volume 114 of *Lecture Notes in Computer Science.* Springer-Verlag, 1981.

[Lam83] Butler W. Lampson. Hints for computer systems design. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 33–48, October 1983. Also appears in *IEEE Software*, pages 11–28, January 1984.

[LG86] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development.* MIT Press, 1986.

[MT91] Robin Milner and Mads Tofte. *Commentary on Standard ML.* MIT Press, 1991.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[PB88] Colin Potts and Glen Bruns. Recording the reasons for design decisions. *IEEE Transactions on Software Engineering*, 10(418–427), 1988.

[PC86] David Lorge Parnas and Paul C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12(2):251–257, February 1986.

[Sim62] Herbert A. Simon. The architecture of complexity. *Proc. American Philosophical Society*, 106(6):467–482, December 1962.

[Sim73] Herbert A. Simon. The organisation of complex systems. In Howard H. Pattee, editor, *Hierarchy Theory: The Challenge of Complex Systems*, pages 1–27. Braziller, New York, 1973.

[Spi89] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, 1989.

Revealing Design Decisions

Tim Gleeson

Aug 1991

Revision :

Keywords: abstraction, design decisions, efficiency, information hiding, Modula-2, modularisation, module selection, module dependence, optimisation, refinement, software reuse

1 Introduction

Abstraction, as a process of design decision hiding [Par72], has a clear role in computer systems development [LG86]. The revelation of design decisions has not received so much treatment. We attempt to show why this may be interesting and useful.

2 Background

Scoping is a mechanism for statically enforcing abstraction boundaries. However, the compilation and linking process is responsible for removing the boundaries and producing a flat implementation suitable for execution. This process:

- Removes the boundaries between source level abstractions. In the source, the semantics of the language describe the meaninglessness of an identifier taken out of scope. In the object module, there is a relatively[1] flat space of addresses.
- Performs vertical transformations from high-level abstractions to low-level abstractions, perhaps passing through many intermediate level abstractions.
- But, there are no great horizontal transformations which take place.

However, it was suggested a long time ago that this need not be the case, [Par72]:

> To successfully and efficiently make use of the ... decomposition [based on information hiding principles] will require a tool by means of which

[1] Caveat lector: segmented object architectures are not like this.

programs may be written as if the functions were subroutines, but assembled by whatever implementation is appropriate. If such a technique is used, the separation between modules may not be clear in the final code.

As a program is transformed from its high-level description to a low-level implementation, some design decisions must be revealed, or transferred out of their original scope. We suggest that, at appropriate levels of this transformation, such information can be legitimately employed.

3 Modula-2 opaque types

Let us consider an example of compilation. The ideal of abstract data types (ADTs) [LG86] can be approached in the Modula-2 language [Wir82] by using the concept of *opaque* export of types from a definition module. This allows a type name, and its associated operations, to be exported and used, by importing modules, with no knowledge of its implementation.

Let us examine the information communicated. An importing module $\mathcal{I}$ needs to know the semantics of an opaque type $\mathcal{T}$ exported by module $\mathcal{E}$. A definition module is a workable mechanism of communicating semantics:

- compiler-usable semantics, basically, just type checking information
- an informal semantics for the programmer to read

However, the implementation of $\mathcal{I}$, let's call it $\mathcal{I}_i$, needs to know more about the *representation* of the opaque type $\mathcal{T}$, let's call it $\mathcal{T}_i$. This is because $\mathcal{I}_i$ must allocate space for $\mathcal{T}_i$ objects[2]. This information has nothing to do with $\mathcal{T}$'s semantics and certainly doesn't appear in the definition module. This information is needed at compile time, when $\mathcal{I}_i$ is produced from $\mathcal{I}$.

Modula-2 adopts a quick-and-dirty (others might say, well engineered compromise) solution: opaque types are essentially limited to pointer types. Thus the importing module implementation allocates a WORD (or however big an ADDRESS is) for values of the imported type. So, the information is passed by making a global announcement of the size of such an object. This is an example of very early binding, or very early information revelation. This compromise simplifies compilers and linkers considerably, and at a small cost of awkwardness to the programmer, is perfectly workable.

Such a restriction has some important consequences. The most important is the removal of dependence of $\mathcal{I}_i$ on $\mathcal{T}_i$, so recompilations of $\mathcal{E}$ do not force recompilations of $\mathcal{I}$; in Modula-2 terms, $\mathcal{I}$.mod depends on $\mathcal{E}$.def and not on $\mathcal{E}$.mod.

[2]Actually, it merely generates a call to $\mathcal{T}_i$.StackAllocate, an extra procedure provided by $\mathcal{T}_i$ which does not appear in $\mathcal{T}$. This is exactly the information which is "revealed".

4 The revelation of design decisions

We thought that abstraction allowed us to contain decisions. Now we see that some decisions, how type $\mathcal{T}$ is represented as $\mathcal{T}_i$, must be revealed during the transformation process from abstract to concrete program descriptions.

The example we have seen illustrates the fact that the Modula-2 concepts of definition and implementation modules do not ideally correspond to abstraction principles. They show a (reasonable) compromise between: decision hiding and revelation, early and late binding of decisions and the relative economies of compiling and linking.

However, it has also shown that during the vertical transformation (compilation) process, some information must be passed horizontally (between modules).

Here we want to suggest that, without compromising abstractions, the revelation of hidden design decisions can be used to good effect.

5 Generic packages

The behaviour of a generic package clearly depends, in a very well defined way, on the behaviour of the actual parameters which it is given. There is no dependence on the implementation of those parameters. It seems possible, however, to make the implementation of that generic package dependent on aspects of the implementation of its parameters. This implies no loss of abstraction at the abstract level. This has not been done, I believe, for several reasons:

- The extra dependence may increase the compilation/link cost associated with making changes and rebuilding the system
- It is rather hard to express the characteristics of an implementation which another implementation might be interested in.

Preserving abstraction requires that abstract dependencies be clearly stated. The decisions made explicit by the compilation of one unit can be used in dependent units in roughly two ways:

object generation We have already seen that the production of $\mathcal{I}_i$ from $\mathcal{I}$ requires knowledge of the size of $\mathcal{T}_i$.

source transformation Maybe, knowing the size of $\mathcal{T}_i$, we will use a different algorithm or data structure in I.

Some of the dependencies we may wish to introduce can be simply expressed as a substitution, for example, the amount of space to reserve on a stack for a given object. Conventional parameterisation mechanisms fulfill this need. Other dependencies require more significant changes.

6 Conclusion

The distinction between the work of a "compiler" and the work of a "linker" seems to rest at the point where local transformations have finished and global transformations have been adopted. I don't wish to challenge this division, merely to suggest that we could try getting the linker to do more, firstly by revealing design decisions which may be useful over a more global context and secondly by removing decisions which have been bound in at an early stage, for example, that opaque types are restricted to pointers and, more radically, stack structures.

One possible criticism of such an approach is that it will lead to systems which are riddled with dependencies. The contrary argument is that if our systems aren't riddled with dependencies, then they can't be very flexible. Explicitly representing our design decisions in this way will ultimately be helpful.

References

[LG86] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development.* MIT Press, 1986.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[Wir82] Niklaus Wirth. *Programming in Modula-2.* Springer-Verlag, 1982.

Generic Module Implementation Selection Based on Parameter Type

Tim Gleeson

May 1991

Revision : 1.1

Keywords: specification, abstraction, refinement, implementation, software reuse, modularisation, module selection

This is the introduction for a paper that I might write. I would be very grateful for your comments on it.

1 Introduction

We show how the parameter type passed to a generic module can be used to select one of a number of possible implementations for that module. This scheme depends on having a well-defined behavioural subtype relationship over parameter types which can be examined before link time by the generic module. The selection mechanism can be automated or can prompt the programmer with the alternatives available.

We further regard "implementation" as being a special, statically determined case of "subtyping". By breaking the traditional distinction between the two notions, further information can be used in module selection. The costs of this appear in the form of an increased dependency between module and parameter, but is acceptable when automated.

Suggestions for language features to support these ideas are given.

November 1st 1990
Tim Gleeson

Report on

First Japanese Knowledge Acquisition for Knowledge-Based Systems Workshop JKAW'90

Kyoto, October 25th – 26th 1990

Workshop Structure Theme and Attendance

JKAW'90 was one of a series of Knowledge Acquisition workshops, previous ones having been held in Canada and Europe. It was in two parts, the first part was a two day open workshop (at the Kyoto International Conference Hall) and the second was a three day closed workshop (at the Hitachi Advanced Research Laboratory at Saitama). There was probably sightseeing between the two parts.

This note reports on the first part, the open workshop, which I attended. It is very subjective.

The workshop concerned knowledge-based systems (KBS), particularly expert systems (ES), and addressed the problem of how to get knowledge into them. Knowledge acquisition (KA) seems to be a bottleneck in the development of larger and "better" knowledge-based systems.

All of the speakers were invited, most of them from the US and only two from Japan. My (ungenerous) interpretation is that a number of them only came for the sightseeing, and that their research did not directly address the problems of knowledge acquisition. Nearly all of the attendees (200, maybe) were Japanese. The only gaijin I met were living and working in Japan.

Brian Gaines (U. of Calgary, Canada) Foundations of Knowledge Acquisition

The earliest methods of knowledge acquisition were interviewing experts. Now there are computer-aided tools, such as *repertory grids*, *behaviour modelling* and *text analysis*. Future work will integrate these (though it all seems rather *ad hoc* to me) and provide standards for knowledge representation.

John Boose (Boeing) Knowledge Acquisition Tools, Methods, and Mediating Representations

Mediating representations — problem modelling languages (or notations) that help bridge the gap between experts and computer implementations are essential. These are of varying levels of formality, processability and abstraction (some higher level ones e.g. decision trees can be transformed into lower ones e.g. rules).

Knowledge-based application problems can be divided into *analysis* (interpretation) and *synthesis* (construction) problems. *Design* is a synthesis problem which involves configuring collections of objects under constraints in relatively large search spaces. General methods for solving synthesis problems are sparse.

B. Chandrasekaran (Ohio State U.)
Real-Time Problem Solving and Knowledge Acquisition

This talk discussed what real-time control is, using safety-systems as examples. One theme was that you should try to do as much as possible at "compile time" (i.e. not at run time), i.e. think about the problem domain and task a lot. There wasn't much mention of knowledge-acquisition.

W. Clancey (Inst. for Research on Learning, USA)
Implications of the System-Model-Operator Metaphor for Knowledge Acquisition

Expert systems can be viewed as manipulating a number of models. At the lowest level there is a *domain* model which discusses classes of objects and their properties, e.g. diseases and immune systems. Next there is a *situation specific* model which models a particular enterprise, e.g. one patient, his symptoms and the reasons for them. Finally, communication with the world, e.g. the purpose of this expert system, what the user wants from it. From this viewpoint, *control knowledge* is the set of procedures which actually construct, extend and develop the situation specific model, at run-time, as driven by what the external world (the user) wants.

Analogies are drawn with Wirth's maxim:

Programs	= Data structures	+ Algorithms
Expert systems	= General model	+ Modelling operators
	= Relational network	+ Graph manipulation operators

This talk also drew analogies with blackboard systems.

Hiroshi Motoda (Hitachi)
An Architecture for Knowledge Acquisition by Interview Based on Dynamic Analysis

Knowledge can be acquired *statically*, by asking experts for it and then using it, or *dynamically* by evaluating the performance of the system as it runs, identifying the lacking knowledge and then adding it. Two important issues are *failure detection* and *failure classification and resolution*. This talk introduced a specific dynamic analysis system and discussed general architectures for them.

J. R. Quinlan (U. of Sydney)
Inductive Knowledge Acquisition from Structured Data

One method of knowledge acquisition consists of inductively deriving rules from a set of examples. This has been well demonstrated for simple domains where objects can be classified by a fixed number of attributes. The problem is essentially one of searching the rule space. Complexity and information based heuristics are used to guide the space.

However, a fixed number of attributes is a severe limitation in the kinds of domains which can be modelled. This talk described extending the work to use relations and first order logic to derive rules.

Shigenobu Kobayashi (Tokyo Inst. of Technology)
Research Activities of Knowledge Acquisition and Learning in Japan

This talk covered a lot of knowledge acquisition research in Japan. The accompanying paper has a large bibliography. ASTEM (Advanced Software Technology & Mechatronics Research Institute of Kyoto) is working on very large engineering knowledge bases.

John McDermott (DEC)
Explorations in How to Make Application Programming Easier

McDermott discussed a program generator being developed. This has three parts: Spark, Burn and FireFighter. Spark interviews the user to determine the nature of the program to be developed. Spark searches in a (tree-structured) library of pre-defined computational *mechanisms* for a collection of them which will solve the task. This set of mechanisms is passed to Burn. Burn executes knowledge acquisition tools associated with each mechanism to fill in the domain specific knowledge for the problem. The constructed program is then executed under the eye of FireFighter. FireFighter determines whether the constructed program fulfills the user's task, and if not determines whether new mechanisms or new knowledge is required.

McDermott thinks a collection of 100–1000 domain-independent, general purpose, abstract, mechanisms (e.g. classification, comparison, selection, transformation) will cover most problems. Currently they have 17 and have built several different programs with the system, with some of the mechanisms being reused.

The paper contains a good summary of the results of previous research projects whose ideas the authors used.

Thomas Gruber (Stanford)
Justification-Based Knowledge Acquisition

Knowledge-acquisition consists of building models. The difficulties are due to *representation mismatch*, i.e. the domain language (in which people talk about the problem) and the modelling language (inside the computer) are very different. We need to bring the modelling language nearer to the level of the domain language.

Strategic knowledge is control knowledge at the knowledge level rather than at the search level. It determines what to do next to acquire more knowledge, rather than how to search that knowledge.

Summary

Before I attended this workshop I knew nothing about knowledge-acquisition and not much about expert systems, though I was very sceptical about them. I am still very sceptical about expert systems, but I think there is some work which is relevant. There seems to have been little work on *synthesis* problems, such as software design, but I think we can benefit from ideas on representation of problems and control and acquisition. Perhaps we should talk to some people from the AI department?

The proceedings contains most of the papers given above, and many others from the second part of the workshop which I didn't attend. For an introduction to knowledge-acquisition, read Gaines's paper. Boose has a catalogue of computer-based tools for knowledge acquisition, with

a short description of each, and a very large bibliography as well. McDermott's paper gives summaries of several systems.

June 21st 1991
Tim Gleeson

Report on

The 3rd International Symposium on Future Software Environment (ISFSE3)

Hikone, Shiga-ken
June 12 – 14 1991

Symposium Structure Theme and Attendance

The 3rd International Symposium on Future Software Environment follows from two previous ones which were held in Japan (Kyoto) and the United States (Boulder, Colorado). The next one will also be held in the United States. The total attendance was about 45 people, of whom about 2/3 were Japanese. The symposium was sponsored and organised by the Software Designers Associates (SDA) Consortium. Many of the attendees belonged to SDA or the University of Colorado or both.

The symposium was split into 4 sessions. Each session had a theme, a chairman and 3 speakers. The presentations mostly consisted of the work the speakers were engaged in.

Several of the speakers talked about "process programming" and process based environments. A "process program" is supposed to represent of the steps that are required to design some software. A process program is executed by a design team, perhaps with machine support. It appears that process programs are typically given by a set of rules, i.e. preconditions and actions. See [Ost87] "A software environment is best viewed as a vehicle for the specification of process programs, and for their compilation and interpretation" and for a contrasting view [Leh87] "In terms of our current understanding...process programs are more likely to divert attention from the real problems of software engineering than to help solve them".

Session 1: Approaches to FSE

One of the issues raised in this session was the relation between software design and architecture. It was noted that software design is not "well understood" in the sense that architecture is. An example, illustrating this difference, is the question: what is the software equivalent of a "load bearing wall"?

Leon Osterweil (UC Irvine)

Software environments can be classified into 3 generations:

toolset An environment is a collection of tools and provides help in "tool grabbing"

data centred Interoperability in an environment is enhanced by considering common or compatible data formats

process centred The design process determines tool use and data formats

Three other principles for FSEs were mentioned:

- A "process program" is created by a similar development process to a "computer program". This meta-design must also be supported.
- It is impossible to get requirements right, and requirements change with time. We should regard software development as a *service industry* and not a *product industry*.

- *Development* should be regarded as a *maintenance* process. Initial development is maintenance of an empty item.

Hideyuki Nakashima (ETL)

Nakashima-san discussed some early ideas on "New models for software architecture". His main thesis was that we need more flexible software components: object-oriented components are too inflexible. One way to achieve this is to have software modules adapt to their environment: defaults for a system currently come from inside that system, instead they should be determined by the environment.

This work follows on some recent ideas from Stanford concerning "Situation theory" and in particular "Situated environments".

Gerhard Fischer (U. of Colorado)

Fischer listed a number of claims concerning FSEs:

- "Software engineering" is an inappropriate term; we should be more concerned with "architecture".
- Future software environments will be domain oriented..
- There are no optimal solutions in design, only compromises and tradeoffs.
- Object-orientedness is necessary but not sufficient.
- "Problem setting" and "problem solving" need to be integrated. The "owner" of a problem has to be included somewhere in the design loop, because the problem must evolve as well as the solution.
- Human attention is a scarce resource.

He listed a number of domains that he had been working in:

- architectural floor plan design
- computer network design
- voice mail applications
- user interface design

Session 2: Human aspects of FSE

Heimbigner's talk generated most questions on the subject of "what is an unanticipated event for a process program?".

Jonathan Grudin (U. of Aarhus, Denmark[1])

Grudin's presentation concerned groupware and CSCW. He distinguished them by saying that groupware concerned groups, roles and products whilst CSCW was much broader and included the workplace. He presented 8 design and evaluation challenges for groupware and CSCW systems:

1. Do the people who do the work in the system get the benefit from the system?
2. A certain number of people are needed to make a system work: critical mass and prisoner's dilemma.

[1] Grudin used to be at MCC, Austin, Texas.

3. Social and motivational factors in the workplace must be considered.
4. Exception handling and non-routine work must be considered.
5. Low-frequency events must be considered.
6. Evaluation of results is needed.
7. We must overcome poor intuitions in design.
8. A system must evolve. Customers must be prepared for change.

Dennis Heimbigner (U. of Colorado)

Heimbigner is implementing a system based on the idea of a "process program". He gave some rather general comments on the different possible roles for computers and humans in such a system:

- Machine in control, executing process program. Human as subroutine.
- Human in control. Machine event-driven by human.
- Human as program counter, directing machine.

He suggested that some sort of intermediate system was possible, maybe using some sort of shared memory blackboard like model.

Tetsuo Tamai (U. of Tsukuba)

Tamai-san reported on a (4 year old) design experiment. He asked 3 teams, each with 3 members, to build a system which helped in checking housing loan applications. The experiment was supposed to assess cooperation and competition in small teams. He spent most of the time describing the experiment, and very little time on the results.

Session 3: Conceptual issues in FSE

Takuya Katayama (TITECH)

Katayama-san's talk was titled "Specifying the software process". He identified 3 different aspects which needed to be specified:

Functional: artifacts and their (static) relationships

Behavioural: order of activities (dynamic aspects)

Enactional: real world, project and organisational issues

It is straightforward to use formal methods for specifying functional aspects. Behavioural aspects are harder and enactional aspects are harder still. He reported on an experiment using the HFSP formalism at the Mitsubishi Space Software Company.

Akira Kumagai (Fujitsu)

He said we need to support abduction and intelligent activities, but I didn't understand any of the rest of this "philosophy" talk.

Brian Nejmeh (INSTEP[2])

This was a very solid survey and attempt to analyse current software design environment (SDE) infrastructures. Their taxonomy revealed two important, orthogonal aspects of SDE infrastructures: *functional* and *structural*. Functional aspects are:

- data management
- distribution and communications
- user interaction
- coordination

Structural aspects, starting from the top and moving down, are:

- tools
- tool common services (e.g. widget sets for X)
- substrate (e.g. object management and Xlib)
- platform (e.g. hardware, host operating system)

Particular SDEs can be located on a grid formed by these two dimensions. The key point raised was that we should not built new infrastructures but should try to use those that we already have to build new environments on top of.

Session 4: Supporting technologies for FSE

David Notkin (U. of Washington[3])

An FSE cannot be a monolithic system. The many components of an FSE will need to be integrated, but "integration" has many definitions. However, some sort of "implicit invocation" mechanism will be needed to decouple components. This mechanism will be more like broadcast communication than point to point communication. There are very many different kinds of systems around which possess these characteristics and we need to study their similarities and differences more; ad hoc construction of implicit invocation mechanisms is not sufficient for efficient development.

There are a number of characteristic questions which we can ask about such implicit-invocation systems:

- What kind of components can announce and receive events?
- Is the set of events defined by the system, or can new ones be added by the programmer?
- Can events pass parameters?
- What kind of synchronisation and concurrency facilities are available for multiple receivers?

[2] Innovative SofTware Engineering Practices

[3] Currently at the end of a 1 year sabbatical with Torii-san at Osaka University.

Rick Selby (UC Irvine)

This talk was titled "Measurement driven analysis and feedback systems". This was a report on part of the "Amadeus" project which integrates measurement tools into a software environment. The principles adopted in the project are:

- Active use of measurements
- Integration of these measurements into process programs

These ideas can be used, for example, to detect fault-prone components in a system as it is being developed. The measurement tools in this case might be connectivity analysis.

Takahira Yamaguchi[4] (Shizuoka U.)

This was a rather futuristic project entitled "Software process model inference system using CBR and CIGOL". Case-based reasoning (CBR) is used to build a relatively low-level database of software components. CIGOL[5] is used on this database to try to raise its level by making generalisations. The system requires quite a lot of human intervention.

Summary

This was not an introductory workshop on FSEs. Most of the discussion assumed an understanding of, and belief in, the ideas of rich software environments and process programming. Basic principles were not discussed, some specific issues were mentioned, but the symposium lacked focus. Brian Nejmeh's talk was the most useful but this only discussed environment infrastructures and not environments themselves.

Future software environments still lie some time in the future.

References

[Leh87] M. M. Lehman. Process models, process programs, programming support. In *Proceedings of the 9th International Conference on Software Engineering*, pages 14–16, 1987. Response to an ICSE9 keynote address by L. Osterweil [Ost87].

[Ost87] Leon Osterweil. Software processes are software too. In *Proceedings of the 9th International Conference on Software Engineering*, pages 2–13, 1987.

[4] Naoki Tamura (Mitsubishi Electric Corporation) also spoke.

[5] An "inverted logic" system.

September 28th 1990
Tim Gleeson

Review of

PM: A System to Support the Automatic Acquisition of Programming Knowledge

R. G. Reynolds, J. I. Maletic and S. E. Porvin
IEEE Transactions on Knowledge and Data Engineering,
2(3):273–282, September 1990

1 Introduction

The PM (Partial Metrics) system is a semi-automated system for analysing and storing parts of existing programs.

The project attempts to store what the authors call *fundamental implementation knowledge*; this is acquired from example programs in computer science textbooks. The authors use an inverse of refinement: the input is code and the output is a hierarchy of aggregates (HAG) of sections of the code which may have corresponded to stages in the design of the code. *Metrics* are used to control the aggregation process. These aggregates are then stored in a frame-based hierarchy of "knowledge".

2 Details

Refinement is regarded as a process of converting pseudocode in code: *stubs* in the pseudocode are expanded into code. This is a process of complexity addition. The metrics used apply to both pseudocode and code and measure aspects of complexity. The languages considered are Ada, Pascal, C, OPS5 and Prolog. The complexity metrics depend both on the language and on cognitive issues.

Their system is based on a process of aggregation which they regard as an inverse of refinement. A section of code (an aggregate) is identified and replaced with a stub. The metrics are used to identify aggregates of appropriate complexity, i.e. the kind of complexity that would have been added in the refinement process. The process of aggregation is repeated until a hierarchically nested sequence of code aggregates (HAG) is obtained; each aggregate will satisfy the metric constraints.

This process tries to infer what the refinement process was, but it may not actually reflect the design decisions made.

Next, the code aggregates are clustered, if necessary, to reduce inter-cluster dependencies. A metric is used in this process to identify how independent the aggregates are.

Knowledge is represented in a case (frame-based) tree. The organisation of this tree "reflects the way knowledge might be organised in a basic programming text". Each aggregate is stored at an appropriate place in the tree based upon:

- Its role in the refinement process (i.e., how it relates to the other aggregates).
- Its functionality (what it actually does).

Insertion of aggregates into the tree seems to be mostly a manual process.

3 Comments

- The system is not completely automated.
- The use of the acquired "knowledge" is not discussed.
- This is a *reverse engineering* system which attempts to infer design decisions from code.
- The structure of the case tree is decided before any code is examined. It seems arbitrary.

Philosophy and Psychology of Design
Two Papers

Tim Gleeson

Sep 13th 1990

This document is a short review of two papers. The first, *The Architecture of Complexity* by Herbert Simon [Sim62], is a philosophical description of complex systems. The second, *A Model of Cognitive Processes in Software Development* by Guindon, Curtis and Krasner [GCK87], is a psychological study of design.

We will look at each of these papers in turn, and then draw some conclusions.

1 The Architecture of Complexity [Sim62]

This is a very old paper, but one that is still frequently cited. Simon's main interest is the nature of complex systems and, in particular, the nature of hierarchic systems. Many of the ideas developed here seem very familiar to us now, but the paper is still worth reading.

Complex and hierarchic systems

Simon defines a *complex* system as one made up of a large number of parts that interact in a nonsimple way. He contrasts this with systems with large numbers of parts that interact in simple ways, for example, crystals, polymers and, in computing, systolic systems. The patterns of interactions in these systems are all simple and regular.

In computing, we are all familiar now with hierarchic systems. Simon cites examples of hierarchic systems from social systems, biological and physical systems and symbolic systems.

The importance of hierarchic systems

Simon then goes on to discuss why he thinks hierarchic systems are so common and so important. In this paper (and also in [Sim73]) he tells the story of the watchmakers, Hora and Tempus, who are frequently interrupted by telephone calls. When this happens, the work they were doing is lost. Hora builds his watches by building subassemblies, which he them assembles. Tempus builds watches as single assemblies. Tempus almost never completes a watch: his work is lost every time he gets a phone call. Hora, on the other hand, only loses subassemblies when the

phone rings. Since subassemblies are quite small, Hora is more likely to complete one before a phone call. Eventually he completes enough subassemblies to build a complete watch.

Simon use this story to indicate the enormous advantages of intermediate stable states, and hierarchy in general, to the process of evolution:

> Among possible complex forms, hierarchies are the ones that have the time to evolve.

We should thus expect to see many hierarchic systems in nature: they are *selected for*.

Near-decomposability

One of the most important observations that Simon makes concerns what he calls *near-decomposability*:

> Hierarchies have the property of near-decomposability. Intra-component linkages are generally stronger than intercomponent linkages. This fact has the effect of separating the high-frequency dynamics of a hierarchy — involving the internal structure of the components — from the low frequency dynamics — involving interaction among components.

Most interactions decrease in strength with distance, and this will occur between levels and between subsystems at a given level in a hierarchy. This has important consequences for:

- ease of understanding a system
- ease of designing a system

Though we may never be able to build *completely-decomposable* systems, nearly-decomposable systems are very useful. We can comprehend the components individually, and then comprehend how they are connected. In design, we can build subsystems and then assemble them without concern for the inner functioning of the subsystems.

Simon elaborates this work in a later paper, [Sim73], in which he discusses the idea of *information hiding* as a principle for system design. In the computing field, similar ideas were stated at about the same time by Parnas [Par72].

2 A Model of Cognitive Processes in Software Development [GCK87]

This paper reports on a psychological experiment in software design. The goal was to discover more about the cognitive processes involved in *upstream* design. The paper is long (83 pages), has many typographical errors but is an excellent account of a thorough psychological experiment. The results are certainly significant for

those constructing tools to help in the design process. If reading 83 pages is too hard, [GC88] is a good 6 page summary.

We will examine:

- The experiment
- The cognitive model they developed
- Their observations about the design process

2.1 The Experiment

The experiment aimed to discover more about *upstream* design, that is, the early stages of design when the problem is not well understood and there are no obvious solutions. *breakdowns* often occur in upstream design. These can be:

- Bad design choices or decisions
- The results of bad decisions
- The inability to make good decisions
- Bottlenecks in the design process

Upstream design is important because breakdowns made at this stage have a large impact in later stages of design. Upstream design is also not well understood.

The authors studied in detail 8 designers, with a broad range of backgrounds and experience. They were asked to design the logic for an n-lift[1] system serving m-floors, given a set of constraints on the behaviour of the lifts. The designers were given two hours, but none of them completely solved the problem.

The authors carefully describe:

- how the experiments were carried out
- how their results were analysed
- the limitations of the experiment
- the validity of the results

They note that this experiment was carried out on *individuals* and that the design of large projects is a *group* activity[2] They review other empirical investigations and it is clear that much of it concerns *downstream* development.

[1] *elevator* in American English

[2] Curtis, Krasner and Iscoe [CKI88]. assert that software engineering technologies, which are mostly aimed at individuals, only marginally improve the production of large software. They say that an understanding of human and organisational factors is essential if we want to improve production in large projects.

2.2 The model

Before starting their experiment, they listed a number of issues that a cognitive model of upstream development should cover:

1. Breakdowns
2. The nature of expertise
3. The collection of skills involved (e.g. domain knowledge, general computer science knowledge, problem solving skills)
4. How the type of problem affects the design activities
5. How tools might influence the process

The components of their model are listed in Figure 1.

- Knowledge Sources
  - Problem domain knowledge
  - Specialised design schemas
  - Design techniques, notations and concepts
- Design Process Control
  - Design meta-schemas
  - Design heuristics
  - Design techniques notations and concepts
- Representations
  - Internal: short-term memory and long-term memory
  - External: available media and tools

Figure 1: Main components of the cognitive model

Specialised design schemas and design meta-schemas

I think the most interesting elements of the model, as far as storage and reuse of software are concerned, are the *specialised design schemas* and the *design meta-schemas.*

A specialised design schema is an outline (a sketch, a rough description) of how to solve a particular problem. Specialised design schemas are internal representations of software designs, they can be parameterised (specialised) to particular cases. They help the designer both in producing a solution and in understanding the problem.

The design meta-schema provides control for the design process, deciding where to spend time, which subproblems to tackle, which strategies to use, whether more knowledge about the problem is required etc.

The paper discusses how the use of specialised design schemas and design meta-schemas interact; these observations were made possible because designers with widely differing experience were used in the experiment.

Two dimensions of the cognitive model

The cognitive model that the authors produce shows how internal and external representations of the problem and its solution are related and how the design process builds and alters this structure. The internal representation is the model that the designer has; the external representation is any information stored externally, for example, on paper or on computer. The model is split along two dimensions:

- the problem domain / solution domain dimension
- the internal representation / external representation dimension

This leads to a picture with four quadrants, as illustrated in Figure 2. Much of the paper is involved with explaining the activities within each quadrant, how they relate to activities in other quadrants, and how this model is justified by the observations.

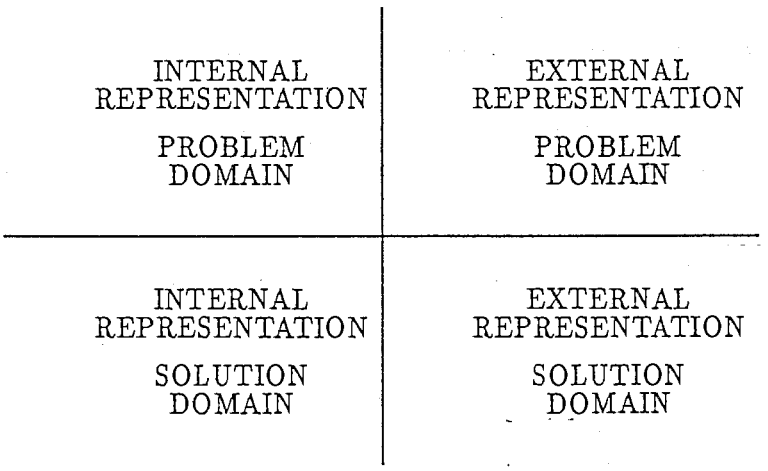

Figure 2: "Map" of the cognitive model

One of the most striking observations about the model, and this observation is not made by the authors, is that there is no direct communication between the external representation of the problem and the external representation of the solution — all interaction goes via the internal representations. This observation may be obvious, but it is very significant. The goal of automatic software generation can be seen as removing the human element in design. In this example, it means completely removing any internal representations — all interaction is done solely in the external representation, in the software tool perhaps.

2.3 Observations

Upstream design is an iterative, interleaved, loosely ordered activity. They noticed that late in the session designers were still involved in understanding the problem.

Some design is *goal-driven*, e.g. "I have to solve subproblem X", however, much of it is not so systematic. *data-driven* design is often used. One example of this is *serendipitous*[3] design where designers notice aspects of the problem for which they already know partial solutions. This can occur at any level of abstraction and happens quite accidentally. This form of design can be seen as proceeding from well-known areas into unknown areas.

External representations (of both problems and solutions) seemed to be used for the following purposes:

- To alleviate limitations in working memory, in particular to serve as a reminding tool for constraints on the solution. Constraints are often forgotten
- A graphical representation can show relationships which are hard to understand internally
- For simulations

Simulation is very important in design. It is carried out in all four quadrants of the model. Limitations in cognitive ability make it hard to simulate systems, especially if they have dynamic and temporal behaviour. It was noted that pen and paper seemed to be inadequate for many simulations. This is one area where computerised design tools might help.

The importance of the ideas of specialised design schemas and design meta-schemas is emphasised by the authors. Many breakdowns in upstream design are due to:

- lack of knowledge
- lack of control of the design process
- cognitive limitations

The authors observed a near-universal trend to develop a *kernel solution* very early on, and they cite other papers which observe the same trend. They conject that this may be a general problem solving heuristic which helps designers to cope with the enormous space of possible solutions. Much greater exploration of the problem domain and of alternative solutions would be beneficial. However, if time is limited, its use must be balanced. They say:

[3] *serendipity* means "happy accident" or "fortunate unexpected discovery". The word comes from the Persian fairy tale *The Three Princes of Serendip*. *Serendip* was the ancient name for Sri Lanka. *serendipity* is the noun and *serendipitous* is the adjective. This is not a common English word.

> We propose that providing tools and methods to permit greater exploration of alternatives (design processes or solutions) and exploration of the problem environment before adopting an initial solution be a priority in research to support the cognitive aspects of software design.

Several other suggestions are made for computer tools to support upsteam design, these all attempt to alleviate the problems in design discovered in the experiment.

3 Conclusions

When we design and build large systems we must consider both the nature of large systems and of the human element in design. Tools to help users design systems must take into account basic ideas of complexity. Also, the intellectual complexity of construction depends as much on the psychology of the constructor as on the physical size of the problem.

References

[CKI88] Bill Curtis, Herb Krasner, and Neil Iscoe. A field study of the software design process for large systems. *Communications of the ACM*, 31(11):1268–1287, November 1988.

[GC88] R. Guindon and B. Curtis. Control of cognitive processes during design: What tools would support software design. In *Proceedings of the CHI'88 Conference on Human Factors in Computing Systems*, pages 263–268. ACM, 1988.

[GCK87] Raymonde Guindon, Bill Curtis, and Herb Krasner. A model of cognitive processes in software design: An analysis of breakdown in early design activities by individuals. Technical Report STP–283–87, Microelectronics and Computer Technology Corporation (MCC), Austin, Texas, 1987.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[Sim62] Herbert A. Simon. The architecture of complexity. *Proc. American Philosophical Society*, 106(6):467–482, December 1962.

[Sim73] Herbert A. Simon. The organisation of complex systems. In Howard H. Pattee, editor, *Hierarchy Theory: The Challenge of Complex Systems*, pages 1–27. Braziller, New York, 1973.

Survey of Design, Design Processes and Information Structures for Design

Tim Gleeson

August 1991

Contents

Chapter 1

Introduction

Aims

The aims of this document are:

- To survey, list and give references to existing, research on design, design processes and information sources used by designers.
- To survey approaches to machine modeling of these processes and sources
- To separate, disambiguate, classify and provide a uniform set of terminology
- To continue this work by filling in missing areas and extending existing ones

Limitations

This survey is not complete and is never likely to be completed because the field it attempts to cover is so large and constantly changing.

A more thorough survey would not include references to papers unread by the compiler. Many such references are included here as potentially useful, but unevaluated, pointers into the literature.

This document has evolved rather than been designed. Periodic overhauls have allowed some structure to remain in it. However, there are many cases where the structuring is inadequate, for example, there are many forward references. It is also unclear how some material should be structured. The section on information structures for design, Section 4, particularly shows this inadequacy.

Structure

There are many ways we can analyse design. Like many previous authors we regard design as an information processing activity, but here we particularly choose to focus on the multi-level nature of design.

In Section 2, "Nature of design", we look at the nature of design and the general activities and structures which occur at every level of design. We try to identify what design is, i.e. to sketch a theory of design. Then in Section 3, "Processes of design", we look at the particular special forms design processes take and in Section 4, "Information structures for design", we look at the information and knowledge structures used. The

split between "process" and "data" is of course somewhat arbitrary, especially in a multi-level interpreter system, but it is useful nonetheless. Section 5 contains extra notes which have not been integrated into the main body of this survey.

In the glossary, page 36, we give a list of important words and phrases and some short definitions of them. There is then a large bibliography, page 39, and an index, page 47.

Chapter 2

Nature of design

Here we examine the nature of design, in particular software system design. Our focus is on design as a multi-level activity and the processes and structures which occur at all levels.

2.1 What is design?

Design requires the proposal of a function to be achieved, resources with which to achieve it and the production of an artifact (the description of an assembly of those resources) that implements it. Typically, a *design task* arises in some social setting where a function has already been proposed (this is the *requirements*) and a *design process* must be undertaken to produce the required artifact. A theory of design demands a more detailed analysis of all these terms.

"Requirements" or "function" must be interpreted liberally: "goal" or "desire" are other terms. It is certainly wider than the "functional requirements" of a computer system. e.g. "Write some music which pleases me."

It is not often stated, but we should bear in mind that not all design tasks are soluble. Some problems may be soluble in theory, but are just too hard in practice, or maybe just too hard for the given designer.

[Sim73] first characterised design tasks as *ill-structured* if they:

- had incomplete and ambiguous goal specifications;
- no predetermined solution path, and
- knowledge had to be integrated from multiple domains.

[Ger90] distinguishes *routine design* where all functions (goals) and structures are known before design begins. The design space is well bounded before design begins. This is simple constraint resolution or prototype instantiation. *Non-routine design* can be divided into *innovative design* when the possible design structures are not known and *creative design* when neither structures nor the allowable design functions are known. Innovative design operates in a well-bounded space, but adaption is required within some of the parameters. Creative design operates in a more open space where variables can be introduced and removed.

Arrangement or configuration tasks [MF89] are dominated by subproblem specification and solution recomposition.

Design is often regarded as a subclass of general problem solving and as such search and exploration are important (search in the space of possible subassemblies or components). However, this analogy is weak when the search space is ill-defined.

[Sim81], cited by [Gui90a], says that the design process is not a natural phenomenon but a human artifact.

Some other factors in design are:

- Design is carried out in a large space ([Boo90]])
- Constraints may be explicit or implicit ([Rei65, Mos85])
- Constraints come from several domains ([Sim73, Mos85, Gui90b])
- Requirements may be incomplete, ambiguous and contradictory ([Sim73, RWR87, Gui90b])
- Evaluation criteria are not well defined ([Gui90b])
- Novelty is often involved, so there is no predetermined solution path ([Sim73, Gui90a])
- Design is an *exploration* process: what is relevant only appears as design proceeds ([Ger90])
- The set of primitive components (and connections) for the solution may be large and only implicitly specified ([Cha90])
- Trade-offs between constraints — relaxing, modifying or rejecting some — may be needed[1] ([Cha90])

2.2 Design as a multi-level activity

Design is a multi-level activity. At each level we find:

- Different problems
- Different processes
- Different sources of information
- Different models (and structures)
- Different goals and different forms of learning

and there is also activity between levels. For example, a "design team" is an entity which needs to learn just like a "designer" does.

The design process appears to be carried out at a number of levels. [NC85, (unread)], cited in [Lew90, Gui90a], partition human behaviour in general into[2]:

- very short term — explicable at the neural processing level.

[1] Need to separate design problem characteristics from design process characteristics.

[2] These, and finer, categories are crucial. Automated support for design activities depends fundamentally on the kind of behaviour we want to support. This grading can be carried upwards if we consider team and group work. We need to indicate what the most significant problem for design is at each level.

- short-term — explicable by "conventional psychology".
- longer-term — explicable by bounded rationality[3].
- very long-term — explicable by social and organisational theory.

A similar set of levels is provided by [RLNM91] who also cite [New90, (unread)]:

Rational band	>10sec	
	~10sec	Goal attainment
Cognitive band	~1sec	Simple operator composition
	~100msec	Elementary deliberate operations
	~10msec	Symbol accessing
Neural band	<10msec	

Communication is vitally important. This must occur both between levels and also within a level. As examples of the latter: at the cognitive plan level, a problem can be split into subproblems and their solutions recombined; at the group level, a problem can be split between a number of designers and their solutions combined. Communication is necessary whether the subtask execute in parallel or nor and whether they are peer or specialised subtasks, as for example in the relationships between human designers and computer design. Communication is thus a necessary consequence of design control processes (3.3) and relies on information structures (4).

2.3 Classifying information sources

Information sources used by designers have been documented before, e.g. [Gui90b] who identified the following "knowledge domains":

- Problem domain
- Requirements and their elaboration
- Design solutions, their representations, simulations and evaluations
- Design strategies, methods and notations
- Problem solving and software design schemas
- Problem solving and design heuristics
- Preferred evaluation criteria

This work is wider, firstly because it goes beyond the human-centered idea of "knowledge domain", and includes external sources of information and secondly because we wish to show that there are many ways of classifying information sources. There are questions we should ask about each of these sources:

Location Where is the information stored

[3]Bounded rationality is itself explained by the interaction of rational endeavours and basic psychological mechanisms.

Storage How is the information stored (and its relation to other information and information sources)

Acquisition How it came to be stored there (for humans: how it was learned; for external media: how it was written)

Retrieval How is it retrieved (e.g., searching method, keys)[4]

Use How is it cognitively (or otherwise) used[5]

Stage What part of the design process it is used in

Limitations Limitations on any of the above, e.g. the potential for cognitive overload

These questions implicitly define classification schemes and they are also questions we should ask of any computer support system.

For example, problem domain knowledge and solution notation knowledge may be stored in the same way (they are both example of domain definitions (2.7)) but are used in very different ways.

We can break knowledge sources into two big classes:

- *a priori* knowledge, either explicit or implicit. This includes previous knowledge and knowledge of the problem specification and environment.
- *working* knowledge, for example that used to store partial solutions.

A priori knowledge is just a source for designers while working knowledge is both a source and product.

2.4 Domains

A *domain* is a space, a part of the world isolated from the rest for separate treatment.

[GC88] give examples of domains:

Technology domains e.g., distributed systems, scheduling systems, algorithms, databases

Application domains e.g., resource allocation systems, inventory control systems

Problem domains e.g., elevators, banking, libraries

Domain knowledge can occur at varying levels of detail. It can be declarative or procedural and may be stored in various ways, for example, in books as definitions and discussions (2.7), as general knowledge in humans as design schemas (4.3), and in machines as domain models (4.2), or as exemplars (2.10) or episodes (4.6).

2.5 Problem domain

[Gui90b, Gui90a] defines the *problem domain* as:

> A subset of the real world with which a computer system is concerned, but not the design solution describing the computer system itself.

This has also been referred to as *task domain* and *application domain.*

[4] Both "structure" and "function" are used in retrieval of design knowledge.

[5] This particularly concerns the cognitive level at which it is used, e.g. low-level, unconscious, automatic; high-level, conscious; social, managerial. See (3.1).

2.6 Solution domain

The key focus in the solution domain will be the notation for the representation of solutions.

[Gui90b] says:

> Design notations play the dual role of vehicle for expressing the design solution and of operators for developing the solutions. Simulations of design solutions... helped uncover [inconsistencies and incompleteness of the solution]. The solution simulations often triggered the inference of new requirements and the discovery of partial solutions.

Imaginal representations [GP89] aid simulation and analogical reasoning.

The solution representation must support solutions and partial solutions and their (mental and external) simulation, evaluation and alteration. Managing an evolving solution is considered by [RA90]. The representation should record: issues considered, resolutions of issues and open concerns. The mechanism which produced the decomposition (maybe a design schema (4.3)) can be used to organize, store and retrieve intermediate solutions.

The urge to generate code as soon as possible[6] can be seen as an attempt to shift the storage of solution issues from limited memory (4.1) to an external medium. However, a programming language is not a good medium for the expression of high-level solution designs[7] and there will certainly be a lot of "decompiling"[8] up from the language to the high-level issues.

2.7 Domain definition, use and implementation

Domain definitions are the "what" part of a domain. Examples are:

- Pascal (User Manual and) Report
- CCITT protocols
- UNIX system calls
- Rules of chess (and go)

However, there is a lot of other useful information about domains in the form of descriptions and discussions of domains. Domain discussions come in two forms: *how* you can implement this domain, using other domains, and *what* you can implement with this domain. Domain models (4.2) usually contain *how* information, but in bottom-up design *what* information is much more important.

Examples of use of domains are:

- Pascal User Manual (and Report)
- Using CCITT protocols

[6] Need some references here.

[7] Big questions are: what needs to be represented and designing an external medium which is both expressive and fluid/mutable/malleable/low-viscosity.

[8] See e.g. [GBP87].

- Programming the UNIX system
- Openings (joseki) and endgames

Examples of implementations of domains are:

- The Pascal P4 Compiler
- Implementation of CCITT protocols
- The 4.3BSD operating system
- Implementing competitive games programs

Discussions usually involve a goal-directed element. The goal-directed element may be a solution description for a problem: to implement a finite table in Pascal, use an array. Many domains are not as well defined as those given above. Such domains cannot have formal definitions, but they can have discussions of use, e.g. the books:

- Ben-Ari, *Principles of Concurrent Programming*
- Gries, *The Science of Programming*

Knowledge of how to use a domain is very important when trying to understand a problem involving that domain. The Draco system [Nei84] attempts to ease programming by providing models of domains, such as basic algebra, along with tools for manipulating expressions in that domain.

When a domain does not have a good definition, it may be illustrated by examples (2.10) or maybe scenarios (2.11).

2.8 Design history

A design history is a record of the activities and products of a design project.

These histories are records of the external aspects of design experiences[9], so they can be at various levels of detail. The least detailed, and most usual, is simply the final product of the project: a program. More detail may be added in the form of requirements documents and intermediate design steps. A significant step is to include details of design decisions (2.14) relating intermediate design stages. These decisions represent relatively logical and presentable summaries of internal (rather less logical) decision processes. Some of these internal activities can be recorded by asking the designer to articulate his thoughts. The value of such records is discussed in [ES80, (unread)].

2.9 Design rationale

[PC86] argue for a document which represents the current state of the design, not the path followed to reach that state. It should represent the design as if it had been produced in an idealised, systematic way. As such, it may be useful for learning about a design. Such a document is certainly useful, but it is only a fragment of a design history (2.8).

Such a *design rationale* represents an Orwellian-Stalinist view of history: it is an idealised view of how and why the system was built.

[9] New topic?

The importance of documentation is emphasised by [PC86] who see documentation as the "medium of design". This relates both to design decisions 2.14 and to the importance of documentation in communication 4.4.

2.10 Examples

Examples can occur in many forms:

- code (for dispatching in the UNIX kernel)
- algorithm sketch (for heap sort)
- element from set (e.g., "blue" from the set "colours")
- deriving a loop invariant

Examples are specialisations. We should not really say "example" but should say "example of..." where "..." represents the abstract domain of discourse. Note that we can have examples of processes as well as of artifacts.

Given an abstract description of a domain (e.g., sorting) it may be hard to conceive of a specific instance from that domain, i.e., to specialise. On the other hand, given a specific example (e.g., bubble sort) it may be hard to *appropriately* generalise to the concept of sorting. This is because it must be done by non-rational induction. We must give both abstractions and examples so users can build appropriate mental models. Where a domain is ill-defined, we may only have examples and no domain definition.

Scenarios (2.11) are examples.

How are examples used? They are found by analogy (the analogy is done at the abstract level) and then modified.

2.11 Scenario

A *scenario* is a test case, or example, or script for a system. A scenario describes a sequence of activities that a system must perform. Scenarios may be given explicitly in the problem requirements (2.12) or they may be retrieved from episodic memory (4.6) as implicit or added requirements, particularly if they are problem domain scenarios. Scenarios can aid the understanding of a system specification because retrieval and simulation of them can add new requirements or help to structure requirements.

[KN87, (unread)] discuss translating scenarios into specifications.

[Gui90b] says:

> Designers interleave problem domain scenarios with solution development throughout the design process. These problem domain scenarios lead to the unplanned discovery of new requirements and of partial solutions in various points and abstraction levels in the solution decomposition.

The retrieval and simulation of problem domain scenarios serve several purposes. [Gui90a, Gui90b] list:

- Simulation to get new ideas. A similar idea was noted in [JTPA81] and called problem solving by understanding. Novices can be expected to do this.

- Disambiguate requirements.
- Confirm the correctness of a discovered partial solution. [KN84] note this behaviour.
- Confirm the relevance of an inferred requirement.

2.12 Requirements

Requirements define the goals of a system. They form the starting point of a design task (2.1). Evaluation criteria are an implicit form of requirement.

Early requirements for software systems may be *incomplete*, *ambiguous* and *contradictory* [RWR87, Gui90b]. Because of this, we need to *elaborate* requirements [Gui90a] by inference of implicit requirements and addition of new ones.

[Gui90b] says:

> Inferred and added requirements mainly serve two purposes:
>
> 1. they lessen the incompleteness and ambiguity inherent in the specification of the requirements; and
> 2. they decrease the range of possible design solutions by acting as simplifying assumptions.

Inferred requirements are implicit in the requirements and real-world knowledge, but must be deduced and made explicit. Added requirements are desirable additions.

[Gui90a]:

> Inferences and additions of new requirements occur throughout the design solution development and are triggered from many sources... [including] associations between related concepts[10], external diagrammatic representations, and prerequisites for the application of a design process strategy.

Inference and addition of requirements can lead to rapid shifts in design control (3.3).

The requirements can be stated in a number of ways, including [BKSZ90]:

- specification
- constraints
- scenarios

This is an idealized view. The specification should be a complete description of the system behaviour with the constraints and scenarios aiding understanding, and thus solution development. The following satisfaction conditions should hold, where $X \supseteq Y$ means X satisfies Y:

- specification $\supseteq$ scenarios
- specification $\supseteq$ constraints
- scenarios $\supseteq$ constraints

[10] Spreading activation.

but since the requirements are ambiguous and incomplete these relationships may not hold. Part of the design process is to *modify* the requirements so that they meet these relationships. However, we are then tackling a different design task, and only the social context in which the task arose can determine whether this is an appropriate course of action.

The requirements may contain constraints (2.13) which are of the form "how a system is built" rather than "what the system does", for example, bottom-up requirements. These requirements may not be visible to the average user, but they will be visible to the maintenance and accounts departments of the customer. Thus we really have multiple "users" of a system, with different views and overlapping requirements.

It is possible that such requirements, especially if they are low-level ones constraining our implementation, will force or influence earlier refinement decisions. These low-level pressures can sometimes force their way all the way up to the system specification and requirements. This is an example of where building a system is an exercise in discovering consequences.

Part of software system design is requirements *validation* against user's desires [RWR87]. This is done in several ways, e.g. simulation, symbolic execution, analysis. This is a particular case of the more general validation involved in design (3.2).

2.13 Constraints

Constraints represent limits on the regions of space which are valid for a given design task.

Constraints can be stated not just in terms of the product of design, "it has to be black", but also on the design process itself, "do it in less than a week". They can occur at any level of the design process.

[GCK87] list the following examples of constraints, many of which will be implicit:

- the problem
  - (perhaps informal) functional specification
  - limitations in solution environment
  - performance and usage requirements
  - form of the artifact (e.g., maintainability, reliability, simplicity)
- the design process itself
  - time available
  - allowable costs
  - tools available
  - organization (team, methods, social, marketing, group)
- the designer
  - knowledge of application domain
  - knowledge of system class (i.e., specialized design schemas)
  - design process knowledge (i.e., design meta-schema)
  - cognitive and motivational attributes

Implicit constraints may come from the domain knowledge of the designer rather than the problem specification, e.g. safety issues. Making them explicit is an important part of the design process and is necessary for effective communication (4.4).

If constraints are, or can be, formally stated, then we may be able to *prove* they hold for a given system. Theorem positing and proving could be major tools in problem understanding.

2.14 Design decisions

A *design decision* is a tentatative commitment to a smaller design space[11]. It is often made by choosing one space from a set of alternatives.

[BC89, 2.2] say:

> The process of design can be thought of as searching for a series of design commitments that result in a goal state.

Design decisions occur at all levels of design. They do not just apply to design products (4), e.g. "let's use quicksort", but also to design processes (3) and in particular design control (3.3), e.g. "let's try functional decomposition".

Design decisions are the product of the interaction of design control (3.3) and a particular problem (or subproblem). Design decisions, once taken, can later be revoked. [BC89] report that [MMW85] discuss a *propose* and *revise* strategy. A commitment is proposed but may later be revised if it was no good. Since design decisions are tentatative, we must record them to make it easier to revoke them later [PB88]. [Fre75], cited in [Fre83] as promoting the recording of design decisions and [Fre83] as discussing representations.

A *justification* explains why a particular decision was made. In order to make decisions comprehensible, justifications must also be recorded. In order to make decisions easily revokable, rejected alternatives[12] should also be recorded.

[Jac83] lists some properties that different decisions have:

- How liable is it to error?...
- What is its subject matter?...
- How easily can it be made?...
- How soon will we find out if the decision is made wrongly?...
- How much later development depends on this decision?...
- Does a choice here imply some other choice elsewhere?...
- Is this an independent decision, or must it be made in conjunction with others?

[Fre83] says that each decision should be evaluated as to its impact (if any) on each goal and the relative importance of goals, and:

> We first identify what decisions must be made, then generate alternatives among which to decide, next evaluate these alternatives to form the basis for a rational decision, and finally choose one alternative.

[11] This definition is entirely inadequate.

[12] Question: How are alternatives generated?

Design decisions and solutions (2.6), belong in the realm of *working* rather than *a priori* knowledge (2.3).

Chapter 3

Processes of design

We will now look at some of the processes involved in design, especially cognitive processes and control.

3.1 Design process models

According to [FD89, (unread)], cited in [TVTY90], design process models can be categorized into:

descriptive How design is done. What design is from a theoretical view. We have examined the general nature of design in (2) and in this section we examine processes of design.

cognitive Explains the cognitive mechanisms which exhibit themselves as the designer's behaviour, e.g. [Gui90a] accounts for deviations from "balanced development". There should really be many models here, explaining the many levels of the design process: neural, cognitive and social.

prescriptive How design should be done, e.g. top-down design. These models have generally been based on observations of the nature of design, e.g. complexity control. They should really also consider limitations involved in the various levels of design, e.g. cognitive limitations and social communication limitations.

computable Computer models or simulations of design processes.

3.2 Design activities

There are an enormous number of different activities involved in software design, many interconnected, many overlapping and with an inconsistent nomenclature.

[Mah90] splits the general design process into 3 (perhaps recursively applied) phases:

formulation Defining the problem

synthesis Identifying solutions consistent with requirements

evaluation Checking a design description for conformance with requirements

At a very gross level, this set of 3 phases is acceptable. However, it may be better to deliberately introduce the idea of feedback. This allows us to show that earlier parts of design are used in later parts of design.

Here we list the relatively high-level and directed stages of software system design using the above categories and feedback:

formulation requirements[1] acquisition, understanding, specification, structuring and problem selection

synthesis kernel solution generation, synthesis, programming, integration, documentation

evaluation evaluation, testing, validation, verification, criticism, review, consistency and conformance checking

feedback refinement, elaboration, maintenance, evolution, debugging, adjustment, redesign

One important development of this simple model is as follows. A problem is formulated, several solutions are synthesised and then comparatively evaluated and the best solution is adopted. See the *design cycle* of [TVTY90]. This can be seen as a recursive application, where the subproblem is comparative evaluation.

[BC89] suggest that design proceeds by the cooperation of several more specific problem solvers (or subprocesses or subtasks). These can be split into two groups: those that "generate" design commitments and those that "test". In the first class are:

- decomposition
- use of design plans (skeletons)
- critiquing and modifying almost correct designs
- constraint processing

These require some "auxiliary" processes:

- subproblem constraint generation
- recomposition of subparts
- design verification
- design criticism

[Fre83] lists:

> Discovering the structure of the problem means understanding the interaction between [the most significant] factors[2] and the desired functional characteristics of the system.

---

[1] requirements can be considered the top-level problem

[2] Determining relevance and importance seem to be very important factors. We should try to compartmentalise constraints.

> Review takes place at all stages of the development cycle of course, but it is most critical to the design phase... [It] goes beyond just determining if something that has been previously spelled out has indeed been done — it is an integral part of the process of discovering the nature of the problem and the proper structure of the solution.

It has been suggested that evaluation may not be needed if we have:

1. Produced a formal specification defining the problem, and
2. Used a sound synthesis method.

This is not true. A formal methodology can only constrain a finite, predetermined set of characteristics. There will always be other, derived characteristics to evaluate. The use of formal specification in design is discussed in [GT91].

Critiquing is a diagnostic activity, mapping from undesirable behaviour to the structures which generate that behaviour. Designs with causal indices that explicitly relate structure and intended function are useful [GC89].

Verification involves ensuring that a design proposal meets its requirements. One method is *simulation*: this requires component behaviour descriptions: it takes structural descriptions and produces behavioural descriptions which can be checked against requirements.

3.3 Design control

Design control includes:

- focusing of the designer's attention, e.g. choice of strategy
- resource management e.g. time and cognitive load. [Gui90a] cites [Sim81] as discussing design as a resource allocation problem.

for this it has to

- represent goals and alternatives
- prioritise these

Design control focuses the designer's attention and does resource (e.g., time, cognitive load) management and monitors progress to a solution. It concerns:

- Which strategy should be used next.
- How long should it be used for

Part of its role is to prioritise constraints, many of which will be implicit and not all of which the designer can keep in mind at once (3.6). To do this it has to represent goals and alternatives.

Design control can and does operate at a number of levels[3], for example, those listed in (2.2).

[3] An analogy to multi-level schedulers in operating systems is possible. We must also discuss control at the group/social level.

- At the bounded rationality level, resources are deliberately managed. At a high level, design control can be conscious. Design methods (3.10.1) can be regarded as well-structured, high-level design control strategies.

- Cognitive structures for design control are referred to as *design meta-schemas* by [GCK87].

A particular designer may fail on a given design task. This may be because of inadequate domain knowledge or because of poor design control[4] Thus, in a very loose sense, design control can be regarded as heuristic because it does not guarantee success.

[Gui90b] says:

> An aspect of the control of the design process is delaying firm commitment to a design decision and re-evaluating tentative commitments as additional information is acquired.

The fundamental challenge in the study of design control is the explanation of observed behaviour. For this, we must consider the interaction of different levels of control. One particular observation we must account for is *opportunistic* behaviour.

[Gui90a] explain *opportunistic* design behaviour:

> In terms of its behavioural manifestations, opportunistic design is design in which interim decisions can lead to subsequent decisions at various levels of abstraction in the solution decomposition.
>
> Opportunistic design is characterised by on-line changes in high-level goals and plans as a result of inferences and additions of new requirements.
>
> A blackboard-based model of opportunistic planning could account, parsimoniously, for both opportunistic and systematic design behaviours.
>
> Opportunistic design behaviours, however, do not necessarily imply an opportunistic model of planning...[And83] argued that behaviours that appear to violate hierarchical planning may actually be due to simple failures of working memory.

The latter can happen either by misremembering goals or forgetting deferred goals (3.6).

Now we will look at some models of multi-level control behaviour.

[TVTY90] distinguish between two levels in the design process: the object level and the action level. The former concerns objects, their properties and their behaviours. In the latter, the designer thinks about how to proceed, e.g. by selecting a knowledge base, scheduling reasoning. In their model,reasoning in the former requires deduction, abduction [Poo88, (unread)] and circumscription [McC80, Lif85, (unread)], but action level reasoning only needs deduction, such as rule-based reasoning because (they say) most of this type of knowledge is procedural.

[Gui90a] discuss the difficulty of high-level control and the behaviour of low-level control:

> Ill-structured problems, because of their ill-specified goals, prevent the determination of a single and stable high-level goal[5] and of a corresponding initial

[4] What is poor design control?

[5] This sounds like a design process "important factor".

> hierarchical plan of actions to be executed throughout the design process. Ill-structured problems make a goal-directed, top-down process difficult. On the other hand, human expertise is associated with the application of data-driven rules. The interaction of the ill structuredness of a problem with data-driven processing by experts is likely to induce the recognition of partial solutions at various levels of abstraction prior to an overall solution decomposition. [This supports] the hypotheses of Newell and Nii linking data-driven processing and ill-structured problems. Information that become the focus of attention — partial solutions, problem domain scenarios, requirements and external representations — can trigger knowledge rules. As these data-driven rules are applied, the problems become better structured. In fact, the data-driven recognition of partial solutions is advantageous. The designer increases the number of constraints on the solution and decreases the daunting size of the solution problem space at very little cognitive cost.

[Gui90a]

> Designers tend to develop immediately the partial solution corresponding to the inferred constraint, leading to a change in goal and a shift to another part of the solution decomposition. Until a designer has discovered the design solution decomposition, it is advantageous to evaluate immediately the impact of a new inferred constraint on the solution rather than take note of it and handle it later.

In a footnote in (3.11) we note the relation of rapid, opportunistic shifts in design strategy to interrupt and prioritised systems. If we combine this with the multi-level nature of design activities we get a multi-level prioritised system. The priorities should be based both on the cognitive cost and the expected gain. This *can* have a high-level, goal-directed top-down strategy but which also accounts for opportunistic behaviour arising at lower-levels. Such a model is worthy of exploration and possible simulation to validate its sufficiency. This explains the "interaction" noted above.

3.4 Cognitive level processes

Most psychological research on design assumes that humans build and manipulate *mental models* (also: *conceptual models* or *conceptualisations*) which represent the problem and proposed solutions. These are models because they support some form of simulation. [AS85] hypothesize that mental models may be formed of at least two representations: one operational, supporting simulation, the other assertional.

A description of the human design process from [GCK87] is summarized here:

- Mental models are formed of the problem and solution. Often a kernel idea is rapidly adopted.
- These are refined during design:
  - The accuracy of the models (mental and otherwise)
  - Their relationship
- Specific memory structures are used: to suggest solutions and to add problem constraints

[Mah90] distinguishes 3 activities by the different kinds of design knowledge they employ.

- decomposition of complex problems into simple problems
- case-based reasoning which employs experience in the form of specific episodes (4.6)[6] rather than general schematic (4.3) knowledge
- transformation[7]

Fox, cited in [Ste90], lists other (higher-level) processes including: filtering, analogy, composition and constraint satisfaction.

We must identify the basic unconscious processes being used in the above stages of design. Some of the following attributions come from [Gui90b]:

[New69, Nii86] the data-driven application of rules is fundamental to human abilities to solve ill-structured problems such as design.

[And83] the use of data-driven rules has little cognitive cost, compared with goal-directed behaviour[8].

[RN81, SAE88] knowledge is abstracted from previous problem solving experience.

[GCK87] partial solutions are recognised at different levels of abstraction.

[Gui90a] the associative nature of human memory and spreading activation[9] working memory limitations.

Many researchers postulate an idealized top-down, breadth-first strategy for design problem decomposition, e.g. [RA90, 2.2]. This is called *systematic expansion* by [AS85] and *balanced development* by [Gui90a] when subproblems are explored to roughly the same depth.

An important aspect of observed behaviour to account for is deviations from systematic expansion. This has been observed by:

[CTM79]

[JTPA81] who observed "critical component" deviations

[AS85] hypothesize this is needed so that solutions can be mentally simulated

[GCK87] who observed deviations due to problem exploration and serendipitous recognition of known partial solutions

[Gui90a] thinks unbalanced development can occur when experienced designers already have a good model of a system and when

---

[6] [Mah90] gives a lot of references on AI systems and case-memory organization and search but doesn't directly tackle case-based reasoning as a model for design synthesis.

[7] With reference to grammars: [MR90]

[8] We must build our computer support systems so they provide more space for human data-driven processing, while performing more of the goal-directed work themselves. Analogy with visual momentum [Woo84].

[9] We should discuss this somewhere and put a reference in here.

- there is novelty
- multiple knowledge sources are involved
- when a subproblem appears critical, difficult or has a known solution

3.5 Simulation

Simulation is the activity of exercising a model on some data and comparing the results with another model. Thus it is an *evaluation* exercise. The differences can drive diagnostic and corrective actions to alter the faulty model. Simulation can be purely symbolic or can use scenarios (2.11).

[AS85], citing [DB81, RMG+76, (unread)], suggest that a mental model is not *runnable* until it has reached a sufficient level of specificity.

3.6 Cognitive level problems

We classify knowledge structures into *a priori* and *working* structures (2.3). We can separately analyse cognitive problems in these two structures[10].

Concerning a priori structures, [GCK87, Section 8] say the main cause of design errors are:

- lack of specialised design schemas
- lack of (or poor) design meta-schema
- lack of problem-domain (i.e., problem-environment) knowledge

and they note [GCK87, Section 8.2.1]:

> We hypothesize that the use of a meta-schema for design is particularly useful if the designer lacks more specialized relevant design schemas.

The meta-schema is Guindon's model of human design control (3.3). Design control occurs at many levels, so even if we can't alter a poor designer's mental design control we can provide control at higher levels, e.g. group control of which a computer system might be part of.

Concerning working structures, one major source of breakdown is due to cognitive limitations whereby even if all the constraints were explicit, the designer cannot bear them all in mind at once, or forgets to come back and deal with them.

Mental models (3.4) are a form of working knowledge. One major problem is the difficulty in returning to postponed tasks — forgetting them. [AS85] suggested that concerns arise during simulation (3.5) which are at the wrong level of detail, this sometimes cause tasks to be repeated.

[AS85] observed the importance of external sources, notes, in this respect. Extensive notes were made when designing an unfamiliar artifact in a familiar domain[11], but few

[10] These seem to be cognitive *structure* problems. We must also analyse cognitive *process* problems, e.g. non-logical reasoning.

[11] Because you know the domain, you know exactly which things to write down.

notes were made when designing a familiar artifact in a familiar domain or when designing an unfamiliar artifact in an unfamiliar domain[12].

3.7 Heuristics

Heuristics are rules of thumb that reduce the complexity of a problem. Heuristics do not ensure success, but increase its likelihood.

[GCK87], citing [Jon80, Fai85], note 3 kinds:

General problem solving heuristics for example:

- consider a simpler problem,, maybe by adding simplifying assumptions or ignoring certain requirements
- search for a related (solved) problem
- decompose into subproblems
- if progress is slow, change viewpoint or explore the problem more
- try a new representation, e.g. draw a picture

General design heuristics for example:

- design the most critical component first
- keep the design solution as simple as possible
- keep the design solution parts as consistent as possible

Software design heuristics These can be divided into *application heuristics* and *design technique heuristics.* The former are based on heuristics of the form: *if the application is X, then use method (technique) Y.* The latter are based on heuristics of the form: *if the software design environment is X, then use method (technique) Y.*

3.8 Weak problem solving methods

Weak problem solving methods completely explore a given space, based only on the surface structure of that space. They are domain independent problem solving schemes. They include:

- generate and test
- means-ends analysis

This situation involves making decisions in an unfamiliar domain. When only the goal and available actions[13] are known, and they lack additional knowledge about goal decomposition or proposed action ranking. This situation is like classical puzzle problems: *problem domain* represents the structure of the available actions and *search methods* are the way space is explored [New80]. [NS72] say they are operating in the *basic problem space.*

[12] Because you don't know the domain, you don't know what to write down. Part of your work is domain understanding, and part is solution design.

[13] We may not know either of these!

[BC89] note that weak problem solving methods can be useful in part of the design process. For highly interdependent parameters, an initial heuristic guess is followed by evaluation, assignment of credit (or blame) and adjustment. This could be a form of hill-climbing and can be used in "tweeking" designs.

Use

[GCK87] say these can be resorted to when specialised design schemas are lacking, but their large space of operation produces poor results [LRN86, (unread)].

[ES90] report that [Lar81] empirically investigated novices and experts solving difficult problems. Novices did use means-end analysis but experts never did. [ES90] suggest that maybe intermediate level problem solvers do mix general weak methods with remembered domain-dependent solution methods.

3.9 Expertise

Expertise manifests itself in the differences between expert and novice designers, and this is a particularly fertile field for psychological experimentation.

A number of statements are made about what expertise involves:

[Gui90b] the application of data-driven knowledge rules

[ES90] having the appropriate domain-specific knowledge (citing [MRRH81, (unread)])

[GCK87] requires detailed knowledge of the many different domains and techniques involved in design

[GCK87] a sophisticated control of the design process

[GCK87] the ability to consider multiple alternatives before adopting an initial solution kernel

[LaF89] not so much the ability to solve problems but rather the ability to devise (or revise) problems that fit the solutions they already have (citing [Scr85, (unread)])

[JTPA81] experts experts engage in more thorough decompositions

Many authors conject at how domain knowledge is stored in experts and novices: [ES90] citing [CFG81, SH82] and [AS85] cite [Ade81, (unread)]. The general view is that novices only have structures of low-level, concrete, surface features, while experts have structures involving both abstract and concrete features.

3.10 Design methods, techniques and tools

The distinctions made in this section are mostly derived from [Bjø87].

3.10.1 Design methods

A *design method*[14] is a set of guidelines for selecting and sequencing the use of design techniques (3.10.4) and design tools (3.10.2) in order to construct an artifact. As such, we can regard them as well-structured, high-level design control (3.3) strategies.

[Cha90] says:

> A method can be described in terms of the operators it uses, the objects it operates on, any additional knowledge about how to organize operator application to satisfy the goal... a set of related proposals about organizing computation.

[Gui90b] says:

> A design method provides a sequence of operators and associated tests to apply these operators, for the transformation from an informal specification of the requirements to a design solution. This knowledge is specific to software tasks, but independent of the problem domain. Design methods and design notations can be part of the knowledge searched during design[15].

[Fre83] says that a design method specifies three things:

1. What decisions are to be made
2. How to make them
3. In what order they should be made

He applies these parameters to: top-down, outside-in, inside-out, bottom-up and most-critical component first methods.

Examples of design methods are:

- JSD
- VDM
- structured design
- top-down or stepwise refinement
- object-oriented design

[Fre83] indicates that different design methods are more or less algorithmic "from precise algorithms on the one hand to loose collections on the other". Software design methods tend to be more heuristic. Many design methods are very vague and poorly defined, yet are frequently used. Design methods, however, frequently use non-constructive operators, for example "then find a widget such that it meets constraints X, Y and Z".

JSD follows certain methodological principles[Jac83]:

- Easier decisions should be made before difficult decisions
- The most error-prone decisions should be deferred as long as possible

[14] Language note: *methodology* is the study of *methods*.

[15] Such a search and choice is a matter of higher-level control.

- Implicit decisions should be avoided
- If a decision is error-prone, it should be subjected to the earliest possible confirmation or refutation
- Whenever possible, decisions should be independent of one another

[Cha90] discusses propose-critique-modify (PCM) methods. He discusses 3 groups of methods for solution proposal:

- problem decomposition / solution composition
- retrieval of cases from memory
- constraint satisfaction ([Ste81]) (only useful on small, well-defined spaces)

As we have already noted when discussing weak problem-solving methods (3.8), decomposition and case-based methods reduce the search space by using previously compiled knowledge.

[Mah90], using rather flexible terminology, says that design methods, such as those described in [Ale67, Jon80, LS81, Hol86, (unread)] prescribe what a designer should do, but not how. She says that the decomposition model of design follows directly from design methodology and that specific languages have been developed for machine representation of knowledge to support this method. "Decomposition" can be of either objects or functions.

Problem decomposition maps a problem to a set of subproblems. There may be several possible decompositions, so a choice is needed. A choice is also needed in the order in which to tackle subproblems and this often depends on the dependencies between subproblems. *Constraint generation* (from problem constraints to subproblem constraints) this may alternate with partial design of others involving commitment and backtracking, e.g. propose and revise [MMW85].

At one extreme of decomposition methods we find transformation methods [Bal81] — the human often resolves control issues.

A special case of decomposition knowledge is a *design plan* which specifies a sequence of steps to produce a design — it is a precompiled partial solution to a design goal [Fri79, Ric81, JS85, MDM86, BC89, Cha90].

3.10.2 Design tools

Design tools are such things as design notations (3.10.3) and clerical aids to support a design method (3.10.1) and its design techniques (3.10.4).

3.10.3 Design notations

Design notations include:

- Data flow diagrams, state transition diagrams, structure charts
- Pseudocode, Pascal, predicate logic, CSP

They allow for the expression of system specifications and design solutions, and in this regard they are used externally.

3.10.4 Design techniques

Design techniques are principles. They apply to different pairs of steps in a design. Examples of formal techniques are those used to:

- specify abstract definitions
- transform these into designs and code
- prove such transformations correct
- discharge proof obligations

[Bjø87] suggests there are many well developed (formal) techniques but not enough work has concerned putting these together as design methods (3.10.1).

3.11 Learning

There are several classes of learning relevant to design. We can distinguish modes of learning by whether they apply to *working* or *a priori* knowledge (2.3):

working Learning relevant to the current design project, in particular the discovery of new knowledge and the creation of solutions.

a priori Learning relevant to later design projects, i.e. experience (4.6) including:

- generalisations, e.g. the development of design schemas (4.3)
- specific episodes

[Gui90a], citing [KN84], says *knowledge discovery* is:

> Characterised by the sudden emergence of new knowledge, without apparent planning, which subsequently plays an important role in the solution attempt

[ES90] say that at the start of problem solving:

> Experts first engage in a kind of qualitative analysis [Lar79]. This qualitative analysis was marked by domain inferences that generated additional useful information about the problem situation that was not explicitly stated in the problem statement. [LMSS80] label this a "knowledge development strategy", because it leads to an enriched representation of the problem situation that clarifies the underlying principles involved in the relevant solution method.

This reminds me of Jones's work [Jon78].

Knowledge discovery can lead to the addition of new requirements or new partial solutions and can be followed by drastic shifts in design activities.

- Simulation of solutions can lead to recognition of a solution from another part of the problem.
- A requirement can lead to the recognition of a low-level partial solution before solution decomposition.

- Simulation in the problem domain can lead to recognition of a partial solution in another part of the problem.

The rapid shift in design control (3.3) activities accompanying discovery of partial solutions, rather than making a note and deferring it, can be accounted for [Gui90a] by the fact that partial solutions can be easily retrieved and reused and immediately add additional constraints to the problem[16].

[ES90] say:

> Skill acquisition can be regarded as a type of inductive learning, because every new episode of performing some task or solving a problem can be viewed as an example... Two very general learning mechanisms — generalization and discrimination — are part of most skill acquisition systems.

Learning is a matter of transferring information between the various domains we have identified. On the way it is processed:

- specialised
- abstracted
- used to make inferences (proofs, new knowledge)
- cross-referenced

[And82] proposes a model (ACT) for skill acquisition. There are two major stages:

> A declarative stage in which facts about the skill domain are interpreted and a procedural stage in which the domain knowledge is directly embodied in procedures for performing the skill... Knowledge compilation is the process by which the skill transits from the declarative stage to the procedural stage... Once proceduralized, further learning processes operate on the skill to make the productions more selective in their range of applications. These processes include generalization, discrimination, and strengthening of productions.

the most notable features of this theory are:

- The interpretive mechanisms which exists are general purpose and are not domain specific
- Knowledge compilation occurs whilst solving specific problems using the interpretive mechanism[17]
- Generalisation occurs after proceduralisation

This should lead us to understand that part of the use of external design media (e.g. books about Pascal) is for interpretive purposes. For inferencing, [HRW79] argue external sources are weaker than internal ones.

[16] This sounds like a high-priority interrupt, or process, within the control structure. Perhaps design activities can be prioritized and addition of constraints is high-priority.

[17] This suggests that a machine must "learn" rather than be "taught" knowledge (at least compiled knowledge). The Soar system [RLNM91] also performs its generalisation, or chunking, while engaged in problem solving. Winograd and Flores [WF86] argue the impossibility of this for real-world domains.

The creation of schemas, schema-abstraction or prototype formation are discussed in [HRHR77, MS78, AKB79, EA81, (unread)].

Schank has worked on learning based on reminding. Explanation plays a major role [Sch86b] and [Sch86a, (unread)].

Chapter 4

Information structures for design

Cognitive sources of knowledge must either be *innate* in humans or *learned* (3.11). There are a number of models of aspects of human knowledge. two which focus on learning are those of *experience* (4.6) which discusses direct encoding of episodes, and schemas (4.3) which discusses generalisation of knowledge. Focusing more on the use of knowledge we have *heuristics* (3.7), *weak problem solving methods* (3.8) and design control (3.3).

4.1 Memory

Psychological models of memory were split into *semantic memory* [Qil68] and *episodic memory* [Tul72, Tul83]. Schank [Sch81] proposed *conceptual memory* which attempted to integrate them.

It's unclear to me that there is a discontinuity for two reasons: specific episodes are just one end of a continuum of specificity, the other end being very general knowledge; secondly, episodes may well be retrieved via schemas anyway, e.g. MOPs (4.6).

4.2 Domain model

A *domain model* is a representation of features of a domain and the relationship of that domain to other domains, possibly refinement relationships.

A number of domains have been modeled or suggested for modeling. The following list is derived from [GCK87] and [RWR87]:

- communication systems
- resource management systems, e.g. inventory control systems
- scheduling systems
- distributed systems
- tracking systems

As an example of the kinds of things which need to be included in a domain model [Fic87] discusses the storage and use of a model of the domain of resource management systems. This includes the concepts (with examples):

resources physical resources, borrowable resources

resource depositories buildings, class rooms, libraries

resource managers staff, editors

resource users attendees, students, borrowers

resource operations add resource, remove resource

usage scenarios resource browsing, resource acquisition

security operations give authorisation, check authorisation

resource constraints maximum size, time limits

resource constraint management waiting lists, fines

queries resources by attribute, usage statistics

environmental aspects staff available, performance constraints

policies maximize available resources, maintain user privacy

Searching a design space clearly depends strongly on the representation used. There are many representations that have been explored in AI and they cannot be sharply distinguished.

Semantic networks (first defined by [Qil68]) are graphs made of labeled objects and relations (e.g. "is-a", "instance-of", "a-part-of") including the idea of inheritance of relations.

A *frame system*, [Min75], is a graph whose nodes are *frames* which have a number of named slots or attributes holding information about an object. References are made to other frames by slots and a hierarchical structure and property inheritance can be implemented.

Schemas (4.3) can also be used to model a domain. *Prototypes* is another common word.

Many of the AI systems noted in this section are attempting to provide comprehensive, detailed, machine models, taxonomies, classifications of and processes of significant domains. I have big doubts that this is or ever will be, possible, [WF86].

The Spark, Burn, FireFighter system of [MDK+90] includes a library of mechanism[1]:

> Spark will select, from a library of pre-defined computational mechanisms, one or more mechanisms that can collectively produce the results desired.

In his presentation, McDermott said a collection of about 100–1000 domain-independent, general purpose, abstract, mechanisms would cover most problems, giving as examples:

- classification
- comparison
- selection
- transformation

[1] But does it have domain models, or are these implicit in the computational mechanisms?

They currently have 17 such mechanisms.

The Requirements Apprentice (RA) [RWR87] is based on having a library of *clichés* in the particular domain required:

> Formally a cliché consists of a set of *roles* embedded in an underlying *substrate.* The roles of a cliché are the parts which vary from one use of the cliché to the next. The substrate of the cliché contains both fixed elements of structure (parts that are present in every occurrence) and *constraints.*

An example cliche might be "sequential search". [Wat85] says:

> Design decisions will be introduced into the new system by making them the basis for the organization of the cliche library... While there are several thousand important cliches, there are probably only a few hundred design concepts... It is the design concepts which form the key vocabulary which the programmer and the system must have a mutual understanding of.

[Ger90] discusses a particular kind of design schema, *design prototypes*, which derive from work on prototype theory [OS81, (unread)][2] and scripts [SA75, (unread)]. It brings together all the sources of knowledge for a specific design situation.

Retrieval

It is possible that several domain models may be retrieved. A control mechanism for selecting between them is needed.

In recovering from failure of a plan during design, [BC89, 4.2.2] suggest that plan *redesign* be used first. If this doesn't work, plan selection is used, i.e. the current plan is abandoned and a new one adopted.

Use

Domain models are used by *case-based* design methods (3.10.1). This use of domain models can be sharply contrasted with the use of weak problem solving methods. The latter are based on surface structure, are brute force and completely cover a space. The former are compiled models of deep structure, allowing critical and rapid access to relevant parts of a space.

Domain knowledge can be used during at least two different design activities:

elaboration A domain model is a more general case of a particular problem. By matching in the problem, we can get suggestions of missing objects, actions and constraints and of conflicts.

refinement A domain (e.g. storage data structures) contains heuristics which allow us to make refinement choices.

[Bar87] notes the ways in which application domain and target software knowledge are used:

- decision making
- inference and analysis
- communication

[2] In prototype theory, membership of a concept is determined by similarity to the concept's best exemplar.

4.3 Design schemas

A *schema*[3] is a generalised, abstract knowledge structure, rather than a record of a particular event or experience. Schema theory has developed in two areas: in AI, which seeks clustered knowledge representations such as *frames* [Min75] and *scripts* [SA77], and in research into human memory, e.g. [AH83] who say:

> It is widely agreed... that the term *schema* has no fixed definition. It is most often used to refer to the general knowledge a person possesses about a particular domain. A schema allows for the encoding, storage, and retrieval of information related to that domain.

[THR79] provide a good, general introduction; they list the following generally agreed properties:

1. A schema represents a prototypical abstraction of the complex concept it represents...
2. Schemata are induced from past experience with numerous exemplars of the complex concept it represents...
3. A schema can guide the organization of incoming information into clusters of knowledge that are "instantiations" of the schema. This represents a goal-directed focusing of processing by active memory schemata.
4. When one of the constituent concepts of a schema is missing in the input, its features can be inferred from the "default values" in the schema...

[GC88] (and [GCK87, Gui90b]) discuss *specialised design schemas*. I think the word "specialised" is redundant because schemas are always representations of particular domains. "design schemas" are more specific than "schemas" because the knowledge they encode includes design knowledge.

Similar ideas are discussed by [HL85, Lub87, LH87, DS90]. *Programming plans*, which represent stereotypic action sequences in programming, are related.

A memory or domain model does not *need* to have an explicit notion of schemas. Production rule systems, and in particular Soar [RLNM91, 4.3.1], use more primitive representations which may be built dynamically into schema like constructs.

A design schema has two parts:

Preconditions for this design schema to be valid. These are in terms of the problem domain.

Solution plan a skeleton outline of how the problem should be decomposed.

[Gui90b]:

> On the basis of [the preconditions] the design schema is retrieved, in a data-driven fashion. The design schema also specifies a high-level decomposition of the software system into subsystems... Once the design schema is retrieved, it sets up a goal directed top-down processing (in the sense of building the system following a known structure) to design each of the subsystems.

[3] Language note: the plural of *schema* can be *schemas* or *schemata*.

[Gui90b] argues that design schemas can vary in complexity and granularity. She uses the terms *data-driven rules*, *knowledge rules* and *knowledge schemas* in similar ways.

The MOPs model[4] of [KSJ86] is related: the model integrates the storage of episodes (4.6) and schemas. Knowledge is organized around abstract commonalities and differences. MOPs are generalized episodes compiled from individual experiences. Individual experiences are indexed within these structures by features which differentiate them. When two experiences differ from a generalized episode in the same way, *reminding* occurs. *Analogy* occurs when predictions based on the first episode are used to analyze a new one[5]. *Generalization* occurs when similarities between two episodes are compiled to form a new schema.

Schemas are often used as *domain models* (4.2). It is possible that schemas can be used for other purposes, and domains can certainly be modeled in other ways. It is also unclear that human schematic knowledge represents domain knowledge.

4.4 Communication

If we are to build multi-agent design systems[6] then one of the major goals is to make implicit problem constraints explicit so:

1. the other active entities can use and benefit from them
2. by understanding *what* these constraints are, we can understand *how* they are used, and thus more about the design process.

Concerning structures built up during the design process, [RA90, 4.3] citing [Bar83, (unread)] say:

> Researchers in cognitive science point to the ubiquity of ad hoc categories, highly contextualized concepts that are created on the spot, in the service of immediate needs.

This may make the communication of design ideas from man to computer difficult.

4.5 Common knowledge

Common knowledge is vital for communication (4.4).

[Wat85] in the context of the Programmer's Apprentice (PA) and KBEmacs systems, says:

> The key to cooperation between the programmer and the assistant is effective two-way communication — whose key in turn is *shared knowledge*. It would be impossibly tedious for the programmer to explain each decision to the assistant from first principles. Rather, the programmer needs to be able to rely on a body of intermediate-level shared knowledge in order to communicate decisions easily.

---

[4] [ES90] say MOPs originally came from [Sch80, Sch82, Rie81].

[5] See [Car86]

[6] This term is adapted from [Cur90] "multi-agent problem solving" and "multi-agent cognitive process" terms. This may be group design, in which case all the agents are human, or a computer may act as one of the agents, performing some active role.

The setting of goals and the realization of those goals in a software design project are usually performed by different people. Sometimes many people are involved. Sometimes many groups are involved.

The *common knowledge*[7] that these people share is beneficial because it can be used to abbreviate, and more easily communicate, requirements and ideas. It is problematic because people may think they share the same common knowledge when this is in fact not true: they have different internal models of what this common knowledge is[8]. This can lead to confusion, ambiguity, lack of or excess of communication.

Concerning external representations, [Gui90b] says:

> Designers also relied on external representations to keep track of the requirements, but in this case, under the form of a list of notes. The notes were also used to highlight the requirements they felt were most critical[9].

4.6 Experience

Experience is a very important factor in design [Kol83b, (unread)], but this is a very general term which needs to be refined.

Storage

[LaF89]:

> Experience gets encoded into *episodic memory*. In contrast to *semantic memory*, which can be described as the knowledge of facts, hierarchically arranged, episodic memory is the knowledge of situations culled from experience.

Use

[KSJ86] and [LaF89] identify two roles for experience in problem solving:

- refinement and modification of the reasoning process (with both successful and unsuccessful experiences) by changing and elaborating existing memory structures
- via episodic memory it provides a set of exemplars: analogies to previous cases guide and focus later decision making

A *design case* (this term comes from [Cha90]) is an instance of successful post design problem solving, either by an individual or a community. They can be episodic (4.6) or be the result of abstraction and generalisation of many episodes. Design plans can be regarded as abstracted cases. They key issues is how to choose an existing case as close as possible to the original problem. Goals must be prioritized and case indexed with "features". Case-based reasoning is strongly connected to analogical reasoning.

---

[7] Common knowledge is an *external* source of information, since it is created by a group.

[8] An internal model of an external model. This is the same situation as we have with external devices such as paper and computers.

[9] What kind of requirements are the most critical? The most abstract ones? The ones the designer knows least or most about?

Retrieval

How are episodes retrieved? [Rei86] says that to retrieve episodic information, one must access the mental context used to encode the event (for example, constructing a plausible scenario for an occurrence of that type of event) and that the processing is not purely automatic but requires strategic reasoning mechanisms to direct the memory search[10]. This builds on [Kol83a, Kol84, (unread)].

Note that cases/episodes may not just be retrieved from distant episodic memory, but also from memory of earlier in the given design project, i.e. from working rather than a priori knowledge (2.3). This is called reuse of already constructed components or solutions.

[10] If we try to computerise exemplar search, the human may still have to input strategies for the search.

Chapter 5

Miscellaneous

In evaluating which of these design sources to provide automation for, we must not just be concerned about which are the most easily automateable, but also with which are the most useful. We can do this by looking at which areas are the most crucial and the most susceptible to breakdown [GCK87, Section 8] (3.6). In this context, we must consider whether the tools we provide are meant to aid novices or experts. In order to do this we need to understand the differences between novices and experts, [Kol83b, (unread)] (3.9).

Appendix A

Glossary

This is a list of important words with short definitions for them. Some of these definitions are my own and some are stolen from other authors. A reference is given in brackets to the section which discusses that word. Synonyms are also given. It is certainly not complete or correct.

constraints (2.13) Limits on the valid regions of a design space. Requirements (2.12) are often given as a set of constraints. For systems design, many constraints may be implicit.

design (2.1) The production of an artifact which meets some needs. The space and constraints of a design task are given as requirements (2.12). For system design, the constraints come from multiple domains, may be explicit or implicit and are often incomplete, ambiguous and contradictory. Evaluation criteria are also not well defined.

design control (3.3) is concerned with the control of the design process itself, focusing the designer's activities and doing resource (e.g., time, mental load) management.

design decision (2.14) is a tentatative commitment to a smaller design space. It is often made by choosing one space from a set of alternatives.

design history (2.8) A record of the activities and products of a design project.

design knowledge

design method (3.10.1) A set of guidelines for selecting and sequencing the use of design techniques (3.10.4) and design tools (3.10.2) in order to construct an artifact. e.g. JSD, VDM.

design rationale (2.9) An idealized version of design history (2.8) documenting only its current state.

design tools (3.10.2) Such things as design notations (3.10.3) and clerical aids to support a design method (3.10.1) and its design techniques (3.10.4).

design schema (4.3) General, abstract representations of aspects of design domains, including decomposition skeleton. Abstracted from previous design experience. Vary in complexity. Contains or is a mental domain model (4.2).

domain (2.4) Any space: we should not use this term alone when we really mean some particular domain.

domain definition (2.7)

domain discussion (2.7)

domain model (4.2) A representation of features of a domain. May describes its relationship to other domains, possibly refinement.

examples (2.10)

experience & expertise (4.6,3.9) Very general terms which include most internalised forms of knowledge, including domain knowledge, design methods, design schemas and design control.

heuristics (3.7) are rules of thumb that reduce the complexity of a problem. They include:

- general problem solving heuristics
- general design heuristics
- software design heuristics.

mental model (3.4) (also: conceptual model) An internal, mental representation of the current state of the design, i.e. understanding of problem and proposed solution. Supports some form of simulation (3.5).

problem (also: task, application, goal)

problem domain (2.5) (also: task domain, application domain) The domain in which the task or purpose or application of the desired computer system is described.

requirements (2.12) (also: specification) define the goals of a system. They are often incomplete, ambiguous and contradictory for software system design.

scenario (2.11) A scenario describes a sequence of events that a system performs. Scenarios can be given explicitly as problem requirements (2.12) or may be retrieved from experience (4.6) (usually about the problem domain) as implicit requirements.

simulation (3.5) is the activity of exercising a model on some data and comparing the results with another model.

solution domain (2.6)

specifications

weak problem solving methods (3.8) Goal-driven, complete space searches e.g., "means-ends analysis" and "generate and test".

Appendix B

Changes

Since February 7th 1991

- There is a major emphasis on the multi-level nature of design.
- Document split into "Nature of design" (2), "Processes of design" (3) and "Information structures for design" (4) sections.
- Added new "Design decisions" (2.14), "Expertise"(3.9), "Common knowledge" (4.5) subsections and "Miscellaneous" section (5).

Since April 9th 1991

- New "Simulation" (3.5) and "Memory" (4.1) sections.
- Added index.

Bibliography

[Ade81] Beth Adelson. Problem solving and the development of abstract categories in programming languages. *Memory and Cognition*, 9(4):422–433, July 1981.

[AH83] Joseph W. Alba and Lynn Hasher. Is memory schematic? *Psychological Bulletin*, 93(2):203–231, 1983.

[AKB79] J. R. Anderson, P. J. Kline, and C. M. Beasley. A general learning theory and its application to schema abstraction. In G. H. Bower, editor, *The Psychology of Learning and Motivation, Volume 13*, pages 277–318. Academic Press, 1979.

[Ale67] C. Alexander. *Notes on the Synthesis of Form.* Harvard University Press, 1967.

[And81] J. R. Anderson, editor. *Cognitive Skills and their Acquisition.* Erlbaum, 1981.

[And82] John R. Anderson. Acquisition of cognitive skill. *Psychological Review*, 89(4):369–406, July 1982.

[And83] J. R. Anderson. *The Architecture of Cognition.* Harvard University Press, 1983.

[AS85] B. Adelson and E. Soloway. The role of domain experience in software design. *IEEE Transactions on Software Engineering*, 11(11):1351–1360, November 1985.

[Bal81] Robert Balzer. Transformational implementation: An example. *IEEE Transactions on Software Engineering*, 7(1):3–14, January 1981.

[Bar83] L. W. Barsalou. Ad hoc categories. *Memory and Cognition*, 11:211–227, 1983.

[Bar87] David Barstow. Artificial intelligence and software engineering. In *Proceedings of the 9th International Conference on Software Engineering*, pages 200–211, 1987.

[BC89] David C. Brown and B. Chandrasekaran. *Design Problem Solving: Knowledge Structures and Control Strategies.* Morgan Kaufmann, 1989.

[Bjø87] Dines Bjørner. On the use of formal methods in software development. In *Proceedings of the 9th International Conference on Software Engineering*, pages 17–29, 1987.

[BKSZ90] G. David Bergland, Geoffrey H. Krader, D. Paul Smith, and Paul M. Zislis. Improving the front end of the software-development process for large-scale systems. *AT&T Technical Journal*, 69(2):7–21, March/April 1990.

[Boo90] John H. Boose. Knowledge acquisition tools, methods and mediating representations. In Motoda et al. [MMBG90], pages 25–62.

[Car86] J. G. Carbonell. Analogy in problem solving. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning. An Artificial Intelligence Approach. Vol. II*, pages 371–392. Kaufman, 1986.

[CFG81] M. T. H. Chi, P. J. Feltovich, and R. Glaser. Categorization and representation of physics knowledge by experts and novices. *Cognitive Science*, 5:121–152, 1981.

[Cha90] B. Chandrasekaran. Design problem solving: A task analysis. *AI Magazine*, 11(4):59–71, Winter 1990.

[CTM79] J. M. Carroll, J. C. Thomas, and A. Malhotra. Clinical-experimental analysis of design problem solving. *Design Studies*, 1:84–92, 1979.

[Cur90] Bill Curtis. Implications from empirical studies of the software design process. In *Info Japan '90*, pages 127–134, 1990.

[DB81] J. DeKleer and J. S. Brown. Assumptions and ambiguities in mechanistic mental models. In Anderson [And81].

[DS90] Francoise Détienne and Elliot Soloway. An empirically-derived control structure for the process of program understanding. *International Journal of Man Machine Studies*, 33(3):323–342, September 1990.

[EA81] R. Elio and J. R. Anderson. Effects of category generalizations and instance similarity on schema abstraction. *Journal of Experimental Psychology: Human Learning and Memory*, 7:397–417, 1981.

[ES80] K. A. Ericsson and H. A. Simon. Verbal reports as data. *Psychological Review*, 87(3):216–254, 1980.

[ES90] Renée Elio and Peternela B. Scharf. Modelling novice-to-expert shifts in problem-solving strategy and knowledge organization. *Cognitive Science*, 14(4):579–639, October–December 1990.

[Fai85] R. Fairley. *Software Engineering Concepts*. McGraw-Hill, 1985.

[FD89] S. Finger and J. R. Dixon. A review of research in mechanical engineering design. Part I: Descriptive, prescriptive, and computer-based models of design processes. *Research in Engineering Design*, 1(1):51–67, 1989.

[Fic87] Stephen Fickas. Automating the analysis process: An example. In IWSSD4 [IWS87], pages 58–67.

[Fre75] Peter Freeman. Toward improved review of software designs. In *AFIPS*, 1975.

[Fre83] Peter Freeman. Fundamentals of design. In Peter Freeman and Anthony I. Wasserman, editors, *Software Design Techniques*, pages 2–22. IEEE, 4th edition, 1983.

[Fre87] P. Freeman, editor. *Tutorial: Software Reusability.* IEEE Computer Society Press, 1987.

[Fri79] P. Friedland. Knowledge based experimental design in molecular genetics. In *Proceedings of the 6th International Joint Conference on Artificial Intelligence*, pages 285–287, 1979.

[GBP87] T. R. G. Green, R. K. E. Bellamy, and J. M. Parker. Parsing and gnisrap: A model of device use. In H. J. Bullinger and B. Shackel, editors, *Human-Computer Interaction — INTERACT'87*, pages 65–70. Elsevier, 1987.

[GC88] R. Guindon and B. Curtis. Control of cognitive processes during design: What tools would support software design. In *Proceedings of the CHI'88 Conference on Human Factors in Computing Systems*, pages 263–268. ACM, 1988.

[GC89] A. Goel and B. Chandrasekaran. Functional representation of designs and redesign problem solving. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 1388–1394, 1989.

[GCK87] Raymonde Guindon, Bill Curtis, and Herb Krasner. A model of cognitive processes in software design: An analysis of breakdown in early design activities by individuals. Technical Report STP–283–87, Microelectronics and Computer Technology Corporation (MCC), Austin, Texas, 1987.

[Ger90] John S. Gero. Design prototypes: A knowledge representation schema for design. *AI Magazine*, 11(4):26–36, Winter 1990.

[GP89] V. Goell and P. Pirolli. Motivating the notion of generic design within information-processing theory: The design problem space. *AI Magazine*, 10(1):18–38, 1989.

[GT91] Tim Gleeson and Toyofumi Takenaka. The roles of formal specifications in the system design process. In *Information Processing Society of Japan: Spring Conference*, 1991.

[Gui90a] Raymonde Guindon. Designing the design process: Exploiting opportunistic thoughts. *Human-Computer Interaction*, 5(2 & 3):305–344, 1990.

[Gui90b] Raymonde Guindon. Knowledge exploited by experts during software system design. *International Journal of Man Machine Studies*, 33(3):279–304, September 1990.

[Ham89] K. Hammond. *Case-Based Planning: Viewing Planning as a Memory Task.* Academic Press, 1989.

[HL85] M. T. Harandi and M. D. Lubars. A knowledge-based design aid for software systems. In *Proceedings of SOFTFAIR II*, pages 67–74, San Francisco, CA, December 1985.

[Hol86] A. Holgate. *The Art in Structural Design.* Oxford University Press, 1986.

[HRHR77] B. Hayes-Roth and F. Hayes-Roth. Concept learning and the recognition and classification of exemplars. *Journal of Verbal Learning and Verbal Behaviour*, 16:321–338, 1977.

[HRW79] Barbara Hayes-Roth and Carol Walker. Configural effects in human memory: The superiority of memory over external information sources as a basis for inference verification. *Cognitive Science*, 3(2):119–140, April–June 1979.

[IWS87] *4th International Workshop on Software Specification and Design*, 1987.

[Jac83] M. Jackson. *System Development.* Prentice-Hall, 1983.

[Jon78] C. B. Jones. Constructing a theory of a data structure as an aid to program development. *Acta Informatica*, 11:119–137, 1978.

[Jon80] J. C. Jones. *Design Methods.* Wiley, 1980.

[Jon86] Capers Jones, editor. *Tutorial: Programming Productivity: Issues for the Eighties.* IEEE Computer Society Press, second edition, 1986.

[JS85] L. Johnson and E. Soloway. PROUST: Knowledge based program understanding. *IEEE Transactions on Software Engineering*, 11(3):267–275, 1985. Biggerstaff gives the authors in the other order.

[JTPA81] Robin Jeffries, Athea A. Turner, Peter G. Polson, and Michael E. Atwood. The processes involved in designing software. In Anderson [And81], pages 255–283.

[KN84] E. Kant and A. Newell. Problem solving techniques for the design of algorithms. *Information Processing and Management*, 28:97–118, 1984.

[KN87] Van E. Kelley and Uwe Nonnermann. Inferring formal specifications from episodic descriptions. In *Proceedings of the 6th AAAI*, pages 127–132, 1987.

[Kol83a] J. L. Kolodner. Reconstructive memory. *Cognitive Science*, 7(4):281–328, 1983.

[Kol83b] J. L. Kolodner. Towards an understanding of the role of experience in the evolution from novice to expert. *International Journal of Man Machine Studies*, 19:497–518, 1983.

[Kol84] J. L. Kolodner. *Retrieval and Organizational Strategies in Conceptual Memory.* Erlbaum, 1984.

[KR86] Janet L. Kolodner and Christopher K. Reisbeck, editors. *Experience, Memory, and Reasoning.* Erlbaum, 1986.

[KSJ86] Janet L. Kolodner and Robert L. Simpson Jr. Problem solving and dynamic memory. In Kolodner and Reisbeck [KR86].

[LaF89] Marianne LaFrance. The quality of expertise: Implications of expert-novice differences for knowledge acquisition. *ACM SIGART Newsletter*, 108:6–14, April 1989. Special issue on Knowledge Acquisition.

[Lar79] J. H. Larkin. Processing information for effective problem solving. *Engineering Education*, 70:285–288, 1979.

[Lar81] J. H. Larkin. The role of problem representation in physics. In Dedre Gentrer and Albert L. Stevens, editors, *Mental Models*, pages 75–98. Erlbaum, 1981.

[Lew90] Clayton H. Lewis. A research agenda for the nineties in human-computer interaction. *Human-Computer Interaction*, 5(2 & 3):125–143, 1990.

[LH87] M. T. Lubars and M. T. Harandi. Knowledge-based software design using design schemas. In *Proceedings of the 9th International Conference on Software Engineering*, pages 253–262, 1987.

[Lif85] V. Lifschitz. Computing circumscriptions. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 121–127, 1985.

[LMSS80] J. H. Larkin, J. McDermott, D. P. Simon, and H. A. Simon. Models of competence in solving physics problems. *Cognitive Science*, 4:317–345, 1980.

[LRN86] J. E. Laird, P. S. Rosenbloom, and A. Newell. Chunking in Soar — the anatomy of a general learning mechanism. *Machine Learning*, 1(1):11–46, 1986.

[LS81] T. Y. Lin and S. D. Stotesbury. *Structural Concepts and Systems for Architects and Engineers.* Wiley, 1981.

[Lub87] Mitchell D. Lubars. Schematic techniques for high level support of software specification and design. In IWSSD4 [IWS87], pages 68–75.

[Mah88] M. L. Maher. Engineering design synthesis: A domain independent representation. *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, 1(3):207–213, 1988.

[Mah90] Mary Lou Maher. Process models for design synthesis. *AI Magazine*, 11(4):49–58, Winter 1990.

[McC80] J. McCarthy. Circumscription — a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.

[MDK+90] John McDermott, Geoffrey Dallemagne, Georg Klinker, David Marques, and David Tung. Explorations in how to make application programming easier. In Motoda et al. [MMBG90], pages 134–147.

[MDM86] S. Mittal, C. Dym, and M. Morjaria. PRIDE: An expert system for the handling of paper systems. *IEEE Computer*, 19(7), 1986.

[MF89] S. Mittal and F. Frayman. Towards a generic model of configuration tasks. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 1395–1401, 1989.

[Min75] M. Minsky. A framework for representing knowledge. In P. H. Winston, editor, *The Psychology of Computer Vision.* McGraw-Hill, 1975.

[MMBG90] H. Motoda, R. Mizoguchi, J. Boose, and B. Gaines, editors. *Proc. First Japanese Knowledge Acquisition for Knowledge-Based Systems Workshop — JKAW'90*, 1990.

[MMW85] S. Marcus, J. McDermott, and T. Wang. Knowledge acquisition for constructive systems. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 637–639, 1985.

[Mos85] J. Mostow. Toward better models of the design process. *AI Magazine*, pages 44–57, 1985.

[MR90] S. Mullins and J. R. Rinderle. Grammatical approaches to design. In *Proceedings of the First International Workshop on Formal Methods in Engineering Design, Manufacturing and Assembly*, pages 42–69, 1990.

[MRRH81] K. B. McKeithen, J. S. Reitman, H. H. Reuter, and S. C. Hirtle. Knowledge organization and skill differences in computer programmers. *Cognitive Psychology*, 13:307–325, 1981.

[MS78] D. L. Medin and M. M. Schaffer. A context theory of classification learning. *Psychological Review*, 82:207–238, 1978.

[MSM88] S. Marcus, J. Stout, and J. McDermott. VT: An expert elevator designer that uses knowledge based backtracking. *AI Magazine*, 9(1):95–114, 1988.

[NA77] G. S. Novak, Jr. and A. Araya. Representations of knowledge in a program for solving physics problems. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, pages 286–291, 1977.

[NC85] A. Newell and S. K. Card. The prospects for psychological science in human-computer interaction. *Human-Computer Interaction*, 1:209–242, 1985.

[Nei84] J. M. Neighbors. The Draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, 10(5):564–574, September 1984. Reprinted in [Jon86] and [Fre87].

[New69] A. Newell. Heuristic programming: Ill structured problems. In J. Arorofsky, editor, *Volume 3 in Prog. Oper. Res.*, pages 360–414. Wiley, 1969.

[New80] A. Newell. Reasoning, problem-solving, and decision processes: The problem space as a fundamental category. In R. Nickerson, editor, *Attention and performance VIII*, pages 693–718. Erlbaum, 1980.

[New90] A. Newell. *Unified Theories of Cognition.* Harvard University Press, 1990.

[Nii86] H. P. Nii. Blackboard systems: Blackboard application systems, blackboard systems from a knowledge engineering perspective. *AI Magazine*, pages 82–106, Summer 1986.

[NS72] A. Newell and H. A. Simon. *Human Problem Solving.* Prentice Hall, 1972.

[OS81] D. N. Osherson and E. E. Smith. On the adequacy of prototype theory as a theory of concepts. *Cognition*, 9(1):35–58, 1981.

[PB88] Colin Potts and Glen Bruns. Recording the reasons for design decisions. *IEEE Transactions on Software Engineering*, 10(418–427), 1988.

[PC86] David Lorge Parnas and Paul C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12(2):251–257, February 1986.

[Poo88] D. Poole. A logical framework for default reasoning. *Artificial Intelligence*, 36:27–47, 1988.

[Qil68] M. R. Qillian. Semantic memory. In M. Minsky, editor, *Semantic Information Processing*, pages 227–270. MIT Press, 1968.

[RA90] Mary Beth Rosson and Sherman R. Alpert. The cognitive consequences of object-oriented design. *Human-Computer Interaction*, 5(4):345–379, 1990.

[Rei65] W. R. Reitman. *Cognition and Thought.* Wiley, 1965.

[Rei86] Brian J. Reiser. Knowledge-directed retrieval of autobiographical memories. In Kolodner and Reisbeck [KR86].

[Ric81] C. Rich. A formal representation for plans in the programmer's apprentice. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, pages 1044–1052, 1981.

[Rie81] C. K. Riesbeck. Failure-driven reminding for incremental learning. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, pages 115–120, 1981.

[RLNM91] Paul S. Rosenbloom, John E. Laird, Allen Newell, and Robert McCarl. A preliminary analysis of the Soar architecture as a basis for general intelligence. *Artificial Intelligence*, 47(1–3):289–325, January 1991.

[RMG+76] E. Rosch, C. B. Mervis, W. D. Gray, D. M. Johnson, and P. Boyes-Braem. Basic natural categories. *Cognitive Psychology*, 8:382–439, 1976.

[RN81] D. E. Rumelhart and D. A. Norman. Analogical processes in learning. In Anderson [And81].

[RWR87] Charles Rich, Richard C. Waters, and Howard B. Reubenstein. Toward a requirements apprentice. In IWSSD4 [IWS87], pages 79–86.

[SA75] R. C. Schank and R. Abelson. Scripts, plans, and knowledge. In *Proceedings of the 4th International Joint Conference on Artificial Intelligence*, 1975. [SA77] may be more comprehensive.

[SA77] R. C. Schank and R. Abelson. *Scripts, Plans, Goals and Understanding.* Erlbaum, 1977.

[SAE88] E. Soloway, B. Adelson, and K. Ehrlich. Knowledge and processes in the development of computer programmers. In M. T. H. Chi, R. Glaser, and M. J. Farr, editors, *The Nature of Expertise*, pages 129–152. Erlbaum, 1988.

[Sch80] R. C. Schank. Language and memory. *Cognitive Science*, 4:243–284, 1980.

[Sch81] R. C. Schank. The structure of episodes in memory. In D. G. Bobrow and A. Collins, editors, *Representation and Understanding*, pages 237–272. Academic Press, 1981.

[Sch82] R. C. Schank. *Dynamic Memory: A Theory of Learning in Computers and People.* Cambridge University Press, 1982.

[Sch86a] R. C. Schank. *Explanation Patterns: Understanding Mechanically and Creatively.* Erlbaum, 1986.

[Sch86b] Roger C. Schank. Explanation: A first pass. In Kolodner and Reisbeck [KR86].

[Scr85] S. Scribner. Thinking in action: Some characteristics of practical thought. In *Practical Intelligence: Origins of Competence in the Everyday World.* Cambridge University Press, 1985.

[SH82] A. H. Schoenfeld and D. J. Herrmann. Problem perception and knowledge structure in expert and novice mathematical problem solver. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 8:484–494, 1982.

[Sim73] H. A. Simon. The structure of ill-structured problems. *Artificial Intelligence*, 4:145–180, 1973.

[Sim81] H. Simon. *The Sciences of the Artificial.* MIT press, second edition, 1981. Originally published 1969.

[Sla91] Stephen Slade. Case-based reasoning: A research paradigm. *AI Magazine*, 12(1):42–55, Spring 1991.

[Ste81] M. Stefik. Planning with constraints. *Artificial Intelligence*, 16:111–140, 1981.

[Ste90] David Steier. Creating a scientific community at the interface between engineering design and AI. *AI Magazine*, pages 18–22, Winter 1990.

[THR79] Perry W. Thorndyke and Barbara Hayes-Roth. The use of schemata in the acquisition and transfer of knowledge. *Cognitive Psychology*, 11(1):82–106, January 1979.

[Tul72] E. Tulving. Episodic and semantic memory. In E. Tulving and W. Donaldson, editors, *Organization of Memory*, pages 381–403. Academic Press, 1972.

[Tul83] E. Tulving. *Elements of Episodic Memory.* Oxford University Press, 1983.

[TVTY90] Hideaki Takeda, Paul Veerkamp, Tetsuo Tomiyama, and Hiroyuki Yoshikawa. Modelling design processes. *AI Magazine*, 11(4), Winter 1990.

[Wat85] Richard C. Waters. The programmer's apprentice: A session with KBEmacs. *IEEE Transactions on Software Engineering*, 11(11):1296–1320, November 1985.

[WF86] Terry Winograd and Fernando Flores. *Understanding Computers and Cognition: A New Foundation for Design.* Ablex, 1986.

[Woo84] D. D. Woods. Visual momentum: A concept to improve the cognitive coupling of person and computer. *International Journal of Man Machine Studies*, 21:229–244, 1984.

Index