Software Design: An Introduction

David Budgen

1. The Role of Software Design

A question that should be asked (and preferably answered!) at the beginning of an overview paper such as this, is

What exactly is the purpose of design?

and the answer that we will be assuming is along the lines of

"To produce a workable (implementable) solution to a given problem."

where in our context, the eventual "solution" involves producing an artifact that will be in the form of software.

This end goal is one that we need to keep in mind in seeking to provide a concise review of some of the many factors and issues that are involved in designing software-based systems. We also need to remember the corollary to this: that the key measure of the appropriateness of any solution is that of *fitness for purpose.*

The significant characteristic of design as a problem-solving approach is that there is rarely (indeed, almost never) only one solution to a problem. So we cannot hope to identify some systematic way of finding the answer, as occurs in the physical and mathematical sciences. Instead, the designer needs to work in a creative manner to identify the properties required in the solution and then seek to devise a structure that possesses them.

This characteristic can be illustrated by a very simple example of a design task that will be familiar to many, and which is based upon that major trauma of life: moving house! When we move to a new house or apartment, we are faced with a typical design problem in deciding where our furniture is to be placed. We may also be required to assist the removal company by supplying them with an abstract description of our intentions.

There are of course many ways in which furniture can be arranged within a house or apartment. We need to decide in which room each item needs to be placed, perhaps determined chiefly by functionality, and then to decide exactly where it might go in the room. We might choose to focus our attention on getting a good balance of style in one room at the expense of another. We also need to consider the constraints imposed by the configuration of the house, so that furniture does not block doors or windows, and power outlets remain accessible.

So this simple example exhibits all of the main characteristics that are to be found in almost all design problems [1]: no single "right" solution; many factors and constraints to be balanced in choosing a solution; no one measure of "quality;" and no particular process that can ensure that we can even identify an acceptable solution!

1.1 The software design process

An important task for a designer is to formulate and develop some form of abstract design model that represents his or her ideas about a solution. Accepting that these activities that underpin the design process are creative ones, the next question that should be asked is why is it that the task of designing software seems to be even more intractable and less well understood than other forms of design? In [2], Fred Brooks has suggested that some software properties that contribute to this include:

- *The complexity of software*, with no two parts ever being quite alike, and with a process or system having many possible states during execution.
- *The problem of conformity* that arises because of the very pliable nature of software, with software designers being expected to tailor software around the needs of hardware, of existing systems, or to meet other sources of "standards."
- *The (apparent) ease of changeability*, leading to constant requirements for change from users, who fail to appreciate the true costs implied by changes.
- *The invisibility of software* so that our descriptions of design ideas lack any visual link to the form of the end product, and hence are unable to help with comprehension in the same way as usually occurs with descriptions of more physical structures.

Empirical studies of the activities involved in designing software [3, 4, 5] suggest that designers use a number of techniques to reduce the effects of at least some of these properties. These techniques include the use of abstract "mental models" of their solutions, which can then be mentally executed to simulate the final system behaviour; reusing parts of previous solutions; and making notes about future (detailed) intentions as reminders for later stages in development.

Even where designers use a particular strategy to help with developing a design model, they may still deviate from this in an opportunistic manner either:

- to *postpone* making a decision where information is not yet available; or
- to define components for which the information is ready to hand, in *anticipation* of further developments in the design.

The use of an opportunistic strategy should not be taken to imply that design decisions are being made in an *unstructured* manner. Rather, this corresponds to a situation where the designer is making use of his or her own experience and knowledge of the problem domain to help adapt their problem-solving strategy, by identifying those aspects of the solution that need to be given most attention in the early stages [6].

Where a designer lacks experience, or is unfamiliar with the type of problem being solved, then one means of acquiring the experience of others is through the use of a *software design method*. Clearly, to transfer all of the different forms of knowledge that allow the designer to use opportunistic development strategies would be difficult, and design methods are therefore limited to encouraging those forms of design practice that can be prescribed in a *procedural* manner. To do so, they provide:

1. A *representation part* consisting of a set of notations that can be used to describe a design model of the form that the method seeks to develop.
2. A *process part* that describes how the model is to be developed, expressed as a set of steps, with each step representing a transformation of the model.
3. A *set of heuristics* that provide guidance on how the process part should be modified or adapted in order to cope with particular forms of problem. These may consist of alternative procedures, or may identify useful "rules of thumb."

One important point that should be made here: Designing software is rarely a completely *unconstrained* process. The designer not only has to produce a solution to a given problem but must also meet other customer-imposed requirements. These *constraints* may include the need to design a solution that can be implemented in a particular programming language; or one that will work within a particular environment or operating system. Constraints therefore act to limit the "solution space" that is available to the designer.

1.2 Design in the software development cycle

Constraints can affect the design process as well as the form of the product. Designing software is not an isolated and independent activity. The eventual system as implemented will be expected to meet a whole set of user needs (reminding us of the criterion of "fitness for purpose"), where these needs are likely to have been determined by some process of *requirements elicitation*. The activities of *analysis* may be used to identify the form of solution that will meet the user's needs, and the designer is then required to provide a solution that conforms to that form. But of course, the activities of all those tasks will interact, largely because each activity is likely to lead to the identification of inconsistencies between requirements and solution, as ideas about the latter develop.

In a like manner, a designer must provide a set of specifications for those who are to construct a system. These need to be as clear, complete, and unambiguous as possible, but of course it is likely that further needs for change will be identified during implementation. The designer also needs to "think ahead" in planning a solution, since few software systems are used for long without being altered and extended. So designing for "maintenance" (a term that is usually a circumlocution for "extensive further development") is another factor that may influence the form of the solution that is adopted.

1.3 Design qualities

The features of a system that may be considered as representative of our ideas of quality are apt to be dependent upon the specific relationship that we have to the system. We began by suggesting that *fitness for purpose* was a paramount need of any system, but of course, this is not an absolute measure of quality, nor one that can be measured in any direct manner. Simply doing the job correctly and within the resource constraints identified may not be enough to achieve fitness for purpose. For example, if it is anticipated that a system will be used for at least ten years, involving modification at frequent intervals, then our notions of fitness for purpose are very likely to incorporate ideas about how easily the structure of the design will accommodate the likely changes. On the other hand, if the need is for a solution that is extremely short-term, but urgent, we may place much more priority on getting a system that works than on ensuring that it can also be modified and extended.

We do not have space here for a discussion of quality factors, but a useful group to note are those that are usually referred to as the *"ilities"*. The exact membership of this group may depend upon context, but the key ones are generally accepted as being *reliability, efficiency, maintainability*, and *usability*. The ilities can be considered to describe rather abstract and "top-level" properties of the eventual system, and these are not easily assessed from design information alone.

Indeed, it has generally proved to be difficult to apply any systematic form of measurement to design information. While at the level of implementation, basic code measurements (metrics) can at least be gathered by counting lexical tokens [7], the variability and the weak syntax and semantics of design notations make such an approach much less suitable for designs. More practical approaches to assessment at this level of abstraction usually involve such activities as design walk-throughs and reviews [8].

2 Describing Designs

2.1 Recording the design model: design viewpoints

In this section we examine some of the ways in which a designer's ideas about the design model can be visualised by using various forms of description.

A major need for the designer is to be able to select and use a set of abstractions that describe those properties of the design model that are relevant to the design decisions that need to be made. This is normally achieved by using a number of representation forms, where such forms can be used for:

- documenting and exploring the details of the design model;
- explaining the designer's ideas to others (including the customer, the implementors, managers, reviewers, and so forth);
- checking for consistency and completeness of the design model.

Because software design methods must rely upon constructing a design model through a fixed set of procedures, they each use an associated set of representations to describe the properties identified through following the procedures. This forms both a strength and a weakness of design methods: The representations support the procedures by helping the designer visualise those aspects of the design that are affected by the procedures; but they may also limit the designer's vision. (Indeed, the act of deviating from the procedures of a method in order to draw some other form of diagram to help highlight some issue is a good example of what was earlier termed *opportunistic* behaviour on the part of a designer.)

The representations used in software design can be grouped according to their *purpose*, since this identifies the forms of property they seek to describe. One such grouping, explored in some detail in [9] is based upon the concept of the *design viewpoint*. A design viewpoint can be regarded as being a "projection" from the design model that displays certain of the properties of the design model, as is shown schematically in Figure 1. The four viewpoints shown there are:

1. The *behavioural* viewpoint, describing the causal links between external events and system activities during program execution.
2. The *functional* viewpoint, describing what the system does.
3. The *structural* viewpoint, describing the interdependencies of the constructional components of the system, such as subprograms, modules, and packages.
4. The *data modelling* viewpoint, describing the relationships that exist between the data objects used in the system.

2.2 Design representation forms

The three principal forms of description normally used to realise the design viewpoints are text, diagrams, and mathematical expressions.

Textual descriptions

Text is of course widely used, both on its own, and in conjunction with the other two forms. We can structure it by using such forms as headings, lists (numbered, bullets), and indentation, so as to reflect the structure of the properties being described. However, text on its own does have some limitations, in particular:

- The presence of any form of structure that is implicitly contained in the information can easily be obscured if its form does not map easily onto lists and tables.
- Natural language is prone to ambiguity that can only be resolved by using long and complex sequences of text (as is amply demonstrated by any legal document!)

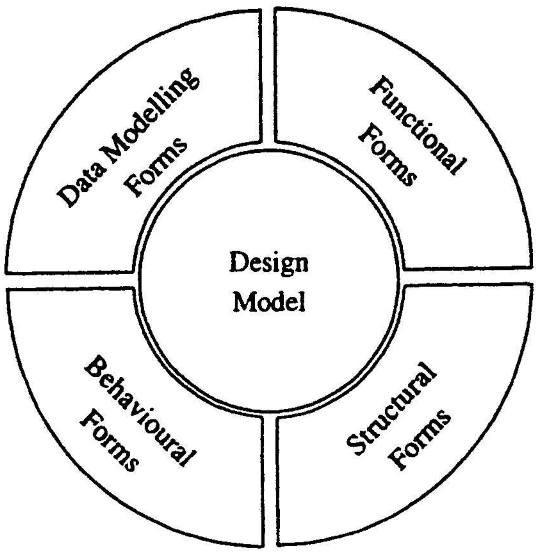

Figure 1. Design viewpoints projected from the design model.

Diagrammatical descriptions

There is a long tradition of drawing diagrams to provide abstractions in science and engineering, and even though the "invisibility" factor makes the form of these less intuitive when used to describe software, they are still very useful. Since they will form the main examples later in this section, we will not elaborate on their forms here, other than to identify the following properties as those that seem to characterise the more widely used and "successful" forms:

- *A small number of symbols.* The symbols in a diagram describe the "elements" that are modelled by that form of diagram, and the number of symbols is often in inverse proportion to the degree of abstraction provided. Most of the widely used forms use only four or five different symbols, including circles, lines (arcs), and boxes.
- *A hierarchical structure.* The complex interactions that occur between software components together with the abstract nature of the components means that diagrams with many different symbols are often very difficult to understand. To help overcome this, many diagrammatical forms allow the use of a hierarchy of diagrams, with symbols at one level being expanded at another level with the same set of symbols, as is shown schematically in Figure 2.

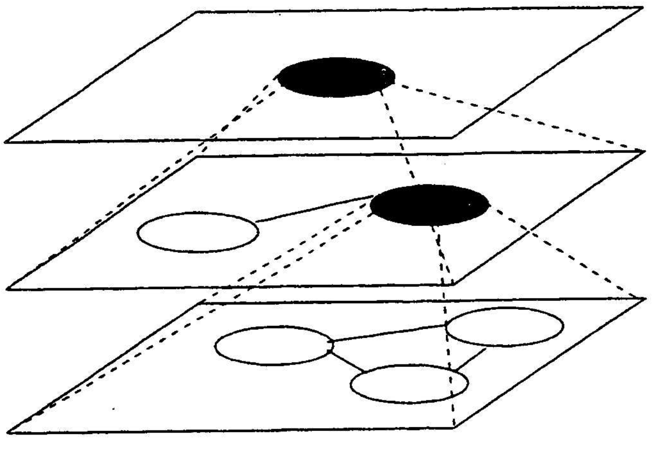

Figure 2. Use of hierarchy in representations.

- *Simplicity of symbol forms*. Ideally, *any* notation should be capable of being drawn using a pencil and paper only (or a white-board). Complicated symbols that require the support of specialised diagram drawing software can limit the ease with which designers communicate their ideas to others.

Mathematical descriptions

Mathematical notations are of course ideally suited to providing concise abstractions of ideas, and so it is hardly surprising that these have been employed in what we generally term the Formal Description Techniques, or FDTs. However, the very terse nature of mathematical descriptions, and the precision that they provide, are not necessarily compatible with the designer's need for abstraction, since they may demand early resolution of issues that the designer may wish to defer.

So far, FDTs have found their main use in specification roles, especially in providing unambiguous descriptions of requirements and descriptions of detailed design features. In both of these roles, their form makes them well suited to exploring the completeness and consistency of the specification, although less so to its development [10].

2.3 Some examples of design representations

To conclude this section we provide some simple examples of diagrammatical notations used for the four design viewpoints. This is a very small selection from the very large range of forms that have been proposed and used (for a fuller survey, see reference [9]). Table 1 provides a summary of some widely used representations, the viewpoints that they provide, and the related design properties that they describe. (It should be noted that the conventions used in these notations do vary between different groups of users.)

The Statechart

Statecharts provide a means of modelling the behaviour of a system when viewed as a finite-state machine, [11] while providing better scope for hierarchical decomposition and composition than is generally found in behavioural representation forms.

A state is denoted by a box with rounded corners and directed arcs denote transitions. The latter are labelled with a description of the event causing the transition and, optionally, with a parenthesised condition. The hierarchy of states is shown by encapsulating state symbols.

A description of the actions of an aircraft "entity" within an air traffic control system is shown in Figure 3. Note that the short curved arc denotes the "default initial state" that is entered when an instance of the entity is added to the system. (For clarity, not all transitions have been labelled in this example.)

The Jackson Structure Diagram

This notation is very widely used under a variety of names. Its main characteristic is that it can describe the ordered structure of an "object" in terms of the three classical structuring forms of sequence, selection and iteration. For this particular example, we will show its use for modelling functional properties, although it is also used for modelling data structure and for describing time-ordered behaviour.

Table 1. Design representations and viewpoints.

Representation Form	Viewpoints	Design properties
Data-Flow Diagram (DFD)	Functional	Information flow; dependency of operations on other operations.
Entity-Relationship Diagram (ERD)	Data modelling	Static relationships between subprograms; decomposition into subprograms
Structure Chart	Structural and functional	Invocation hierarchy between subprograms; decomposition into subprograms
Structure Diagram (Jackson)	Functional, data modelling, behavioural	Algorithm forms; sequencing of data components; sequencing of actions.
Pseudocode	Functional	Algorithm forms
State Transition Diagram (STD)	Behavioural	State model describing how events cause transitions in entities.
Statechart	Behavioural	System-wide state model, including parallelism, hierarchy, and abstraction.

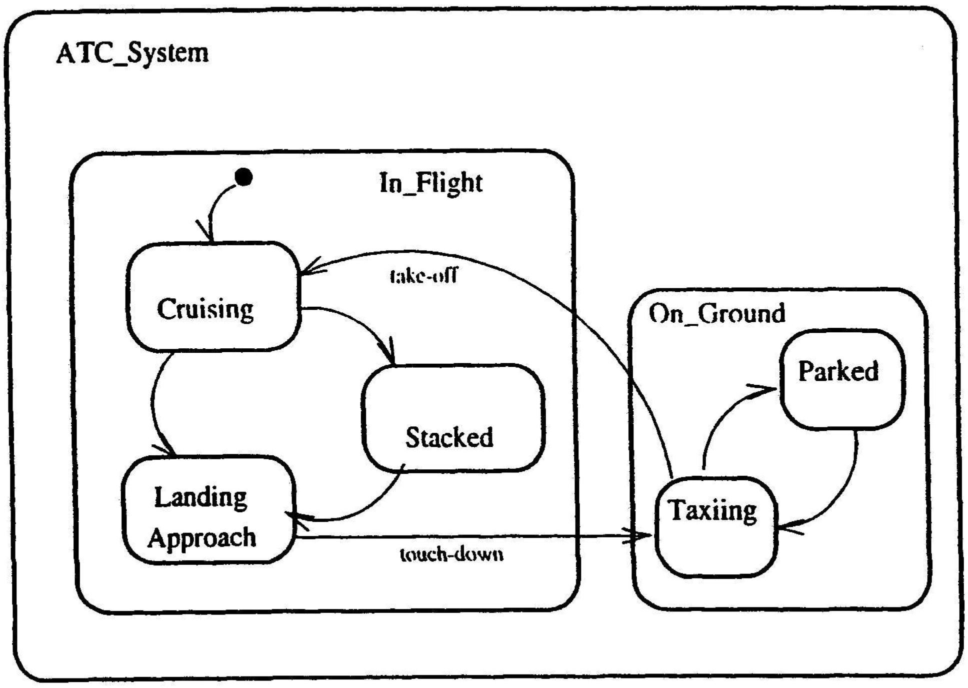

Figure 3. An example Statechart.

Figure 4 provides a simple functional description of the (British) approach to making tea. Points to note are:

- Each level is an expanded (and hence, less abstract) description of a box in the level above.
- Sequence is denoted by an ordered line of boxes, selection by a set of boxes with circles in the upper corner, and iteration by a box with an asterisk in the upper corner.
- The structuring forms should not be mixed in a group on a level. (Hence the action "put tea in the pot" forms an abstraction within a sequence, and this is then expanded separately as an iterated set of actions.)

The Structure Chart

This notation captures one aspect of constructional information, namely the invocation hierarchy that exists between subprogram units. While the tree-like form is similar to that of the Jackson Structure Diagram, the interpretation is very different, in that the elements (boxes) in a Structure Chart represent physical entities (subprograms) and the hierarchy shown is one of invocation (transfer of control) rather than of abstraction. Figure 5 shows a very simple example of this notation. (There are different forms used to show information about parameter passing; this is just one of them.)

The structural viewpoint is concerned with the physical properties of a design, and hence it is one that may need to describe many attributes of design elements. For this reason, no one single notation can effectively project all of the relevant relationships (such as encapsulation, scope of shared information, invocation), and so an effective description of the structural viewpoint is apt to involve the use of more than a single representation.

The Entity-Relationship Diagram

The Entity-Relationship Diagram (ERD) is commonly used for modelling the details of the inter-relationships that occur between data elements in a system, although it may also perform other modelling roles [12]. Figure 6 shows a very simple example of one form of ERD containing two entities (boxes), a relationship (diamond) and the relevant attributes of the entities. Additional conventions are used to show whether the nature of a relationship is one-to-one, one-to-many or many-to-many. (In the example, the relationship is many-to-one between the entities "aircraft" and "landing stack," since one stack may contain many aircraft.)

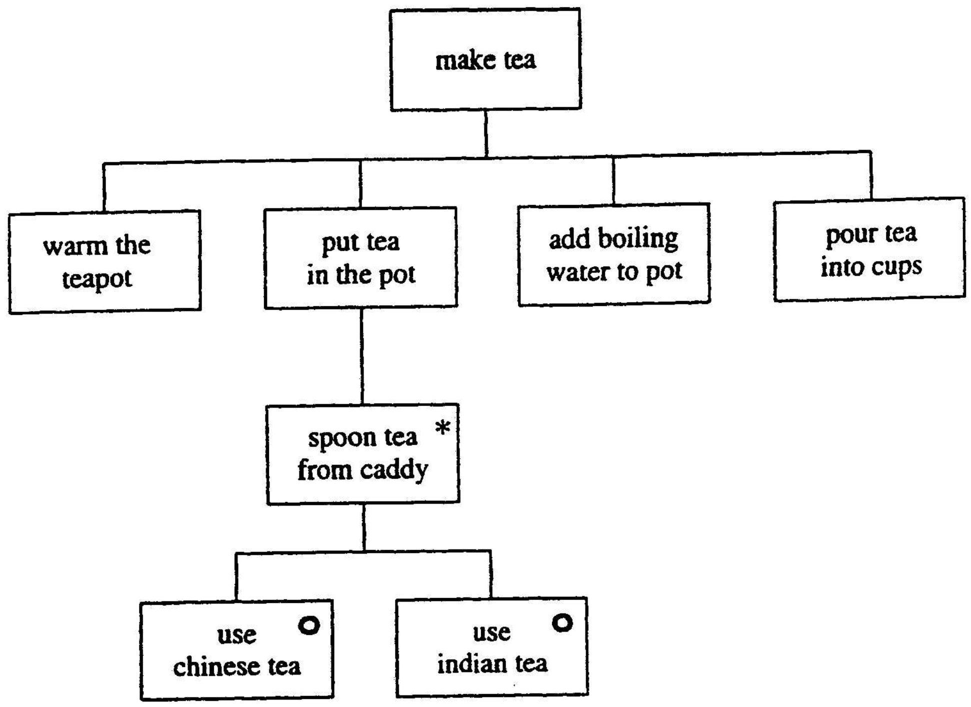

Figure 4. An example of a Jackson Structure Diagram.

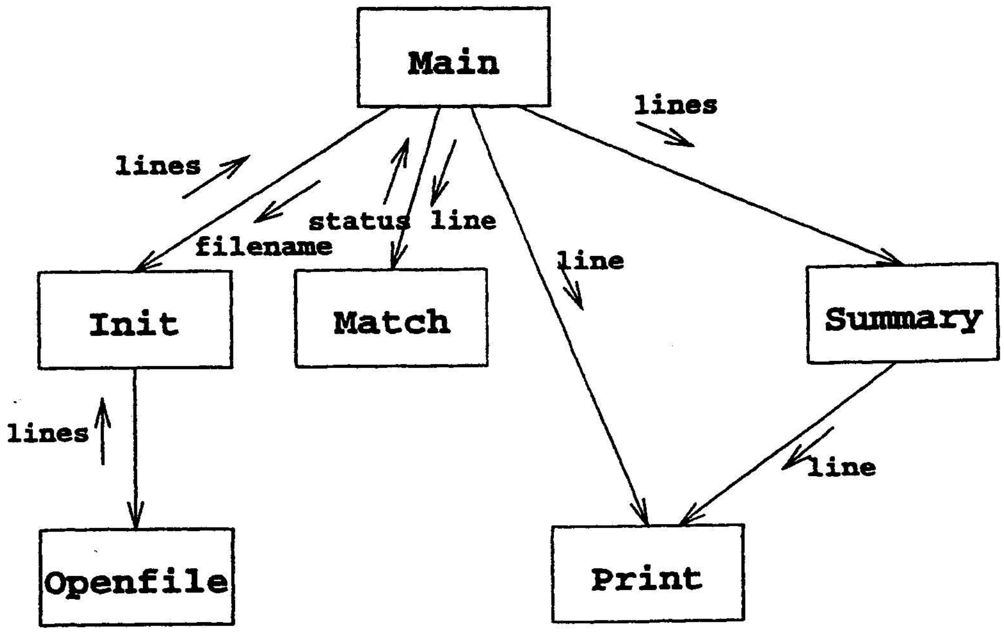

Figure 5. A simple Structure Chart.

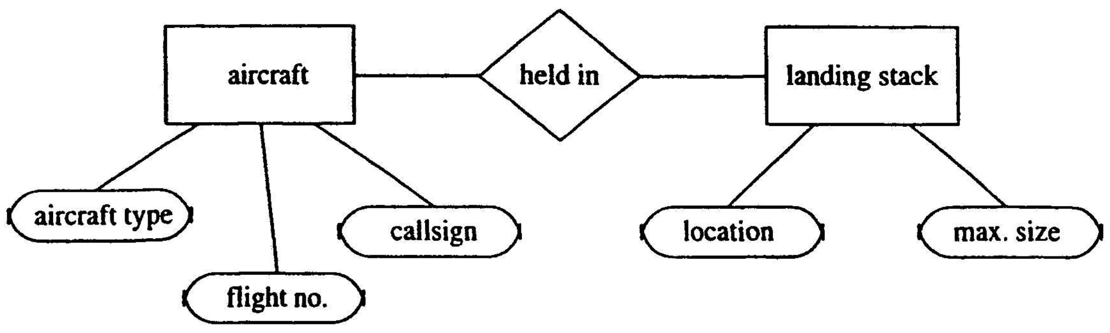

Figure 6. A simple Entity-Relationship Diagram.

3 Software Design Practices and Design Methods

Section 1.1 introduced the concept of a *software design method* as a means of transferring "knowledge" to less experienced designers. This section explores this concept a little further.

3.1 Rationale for software design methods

The use of "methods" for software design has no parallel in any other stage of software development. We do not have "testing methods" or even "programming methods." When teaching programming, we commonly provide the student with a set of "programming metaphors," together with a set of application paradigms such as trees and stacks that make use of these.

The partial analogy with programming points to one of the problems that hinders teaching about design, namely that of scale. Novice programmers can use the abstractions provided in a programming language to construct actual programs, and in the process receive feedback that can assist with revising their ideas and understanding during both compilation and execution of the programs. In contrast, novice designers have no equivalent sources of feedback to indicate where their ideas might be inconsistent, and have little or no chance of comparing an eventual implementation against the abstract design ideas. So "method knowledge" may be our only practical means for transferring experience, however inadequate this might be. (As an example, it is rather as though programmers were taught to solve all needs for iteration by using only the FOR construct.)

Other roles for software design methods include:

- Establishing common goals and styles for a *team* of developers.
- Generating "consistent" documentation that may assist with future maintenance by helping the maintainers to recapture the original design model.
- Helping to make some of the features of a problem more explicit, along with their influence upon the design.

Constraints that limit their usefulness are:

- The *process part* of a method provides relatively little detailed guidance as to how a problem should be solved. It may indicate how the design model is to be developed, but not what should go into this for a given problem.
- The need to use a procedural form (do this, then do that, then...) leads to practices that conflict with the behaviour observed in experienced designers. Decisions have to be made on a method-based schedule, rather than according to the needs of the problem.

So, while at the present time software design methods probably provide the most practical means of transferring design knowledge, we cannot claim that they are particularly successful.

3.2 Design strategies

Design methods embody strategies (indeed, this is where they particularly diverge from the practices of experienced designers, since the latter are observed to adapt a strategy opportunistically in order to meet the needs of a problem). Four widely-used strategies are:

Top-down: As the name implies, this is based upon the idea of separating a large problem into smaller ones, in the hope that the latter will be easier to solve. While relatively easy to use, it has the disadvantage that important structural decisions may need to be made at an early stage, so that any subsequent need for modification may require extensive reworking of the whole design.

Compositional: As implied, this involves identifying a set of "entities" that can be modelled and which can then be assembled to create a model for the complete solution. While the earlier stages may be simple, the resultant model can become very complex.

Organisational: A strategy of this form is used where the needs of the development organisation and its management structures impose constraints upon the design process. This may require that project (and design) team members may be transferred at arbitrary times, so that the method should help with the transfer between old and new team members. A good example of this form is SSADM [13].

Template: Templates can be used in those rare cases where some general paradigm describes a reasonably large domain of problems. The classical example of this form is that of compiler design, and indeed, this is probably the only really good example.

Whatever the strategy adopted, the process part of a method is usually described as a sequence of *steps*. Each step alters the design method, either by *elaborating* the details of the model, or by *transforming* them, adding new attributes to create new viewpoint descriptions. Steps may involve many activities, and provide a set of milestones that can be used to monitor the progress of a design.

The choice of design strategy and associated method has significant implications for the resulting solution structure, or architecture. Shaw's comparative review of design forms for a car cruise-control system [14] demonstrates the wide range of solution architectures that have been produced from the use of 11 different design methods. But of course, this still leaves open the question of how to know in advance which of these is likely to be the most appropriate solution, and hence the most appropriate method! Any choice of a design method may also be further influenced by prior experience, as well as by social, political, historical, business and other nontechnical factors, none of which can be easily quantified (or accommodated in this review).

4. Features of Some Software Design Methods

A review paper such as this cannot examine the workings of software design methods in any detail. So in this section, we briefly review the significant features of a number of well-established design methods, chosen to provide a reasonable range of examples of strategy and form. For fuller descriptions and references, see [9].

Our review starts with two first-generation design methods that provide examples of both compositional and decompositional strategies. After that, we look at two examples of second-generation methods, which typically exhibit much more complex design models.

4.1 Jackson Structured Programming (JSP)

JSP was one of the earliest design methods and provides a useful paradigm for the method-based approach to transferring and developing design knowledge. It is deliberately aimed at a very tightly constrained domain of application, and hence can be more prescriptive in its process part than is normal. In addition, it is an algorithm design method, whereas the other methods are aimed at larger systems and produce structural plans. On occasion, JSP may therefore be useful for localised design tasks within a larger system design task. JSP uses a compositional strategy.

The *representation part* of JSP is provided by the ubiquitous Jackson Structure Diagram. This is used both for modelling data structures and also for functional modelling.

The *process part* of JSP is summarised in Table 2. For each step we describe its purpose and also whether it elaborates or transforms the design model.

The heuristics of JSP are highly developed. *Read-ahead, back-tracking,* and *program inversion* have been widely documented and discussed. Between them, they provide a set of "adaptations" to the basic process, and they can be used to resolve some of the more commonly-encountered difficulties in applying JSP to practical problems.

Table 2. Summary of the JSP process part.

Step 1	Draw Structure Diagrams for inputs and outputs	*elaboration*
Step 2	Merge these to create the program Structure Diagram	*transformation*
Step 3	List the operations and allocate to program elements	*elaboration*
Step 4	Convert program to text	*elaboration*
Step 5	Add conditions	*elaboration*

4.2 Structured Systems Analysis and Structured Design

Like JSP, this is a relatively old method, and can be considered as an extension of the functional top-down approach to design. It consists of an "analysis" component and a "design" component. (For the purposes of this paper, the activities of analysis are considered to be integral with those of design.)

During the analysis phase (Steps 1 and 2), the designer constructs a functional model of the system (using "physical" Data-Flow Diagrams) and then uses this to develop a functional model of the solution (using "logical" DFDs). This is usually supplemented by some degree of data modelling, and some real-time variants also encourage the development of behavioural descriptions using State Transition Diagrams (STDs).

In the design phase (Steps 3 to 5), this model is gradually transformed into a structural model, based on a hierarchy of subprograms, and described by using Structure Charts. There is relatively little support for using ideas such as information hiding, or for employing any packaging concepts other than the subprogram.

The *representation part* therefore uses both DFDs and Structure Charts for the primary notations, and sometimes involves the use of ERDs and STDs.

The *process part* is summarised in Table 3, using the same format as previously.

The heuristics are far less well-defined than those of JSP. One of them is intended to help with determining which "bubble" in the DFD acts as the "central transform," while others are used to help restructure and reorganise the solution after the major Transform Analysis Step (Step 4) has generated the structural viewpoint for the design model.

As a method, this one has strong intuitive attractions, but suffers from the disadvantage of having a large and relatively disjoint transformation step.

4.3 Jackson System Development (JSD)

JSD encourages the designer to create a design model around the notion of modelling the behaviour of active "entities." In the initial stages, these entities are related to the problem, but gradually the emphasis changes to use entities that are elements of the solution.

A characteristic of second-generation design methods is that they involve constructing much more complex design models from the start, usually involving the use of more than one design viewpoint. As a result, they generally use a sequence of elaboration steps to modify the design model, rather than providing any major transformation steps.

The *representation part* of JSD makes use of *Entity-Structure Diagrams* (ESDs) to model the time-ordered behaviour of long-lived problem entities. (These diagrams use a different interpretation of the basic Jackson Structure Diagram.) The function and structure of the resulting network of interacting "processes" is then modelled using *System Specification Diagrams* (SSDs).

The *process part* can be described in terms of three stages [9,15], which can be further subdivided to form six major design activities. Table 4 provides a very basic summary of these activities, using the same format as before.

Table 3. Summary of the process part of SSA and SD.

Step 1	Develop a top-level description	*elaboration*
Step 2	Develop a model of the problem (SSA)	*elaboration*
Step 3	Subdivide into DFDs describing transactions	*elaboration*
Step 4	Transform into Structure Charts	*transformation*
Step 5	Refine and recombine into system description	*elaboration*

Table 4. Summary of the JSD process part.

1. Entity Analysis	Identify and model problem entities	*elaboration*
2. Initial Model Phase	Complete the problem model network	*elaboration*
3. Interactive Function Step	Add new solution entities	*elaboration*
4. Information Function Step	Add new solution entities	*elaboration*
5. System Timing Step	Resolve synchronisation issues	*elaboration*
6. Implementation	Physical design mappings	*elaboration*

The *heuristics* of JSD owe quite a lot to JSP, with both back-tracking and program inversion being recognisable adaptions of these ideas to a larger scale. An additional technique is that of state vector separation which can be used to increase implementational efficiency via a form of "reentrancy."

4.4 Object-Oriented Design

The topics of "what is an object?" and "how do we design with objects?" are both well beyond the scope of this paper. Some ideas about the nature of objects can be found in [16] and in [17].

It can be argued that object-oriented analysis and design techniques are still evolving (perhaps not as rapidly as was once hoped). The Fusion method [12] provides a useful example of one of the more developed uses of these ideas, and one that has brought together a number of techniques (hence its name).

The *representation part* of such methods is often a weakness, being used to document decisions at a later stage, rather than to help model the solution. Fusion seeks to make extensive use of diagrammatical forms, and especially of variations upon the Entity-Relationship Diagram.

The *process part* is described in Table 5 and includes both analysis and design activities. There are no identifiable heuristics available for such a recent method. (A fuller methodological analysis of Fusion as well as of the other methods described in this section is provided in reference [18].) A problem with object-oriented methods is that they do encourage the designer to make decisions about "structure" at a much earlier stage than "process"-oriented methods (including JSD), and hence bind the design to implementation-oriented physical issues before the details of the abstract design model have been fully worked through.

5. Conclusion

This paper has sought to review both our current understanding of how software systems are designed (and why that process is a complex one) and also how current software design methods attempt to provide frameworks to assist with this. As can be seen, even the second-generation design methods still provide only limited help with many aspects of designing a system.

We have not discussed the use of support tools. Many design support tools still provide little more than diagram editing facilities and support for version control. In particular, they tend to bind the user to a particular set of notations, and hence to a specific design process. Inevitably, this is an area of research that lags behind research into design practices.

Overall, while our understanding of how software is designed is slowly improving [19], it seems likely that this will provide an active area of research for many years to come.

References

[1] H.J. Rittel and M.M. Webber, "Planning Problems are Wicked Problems," N. Cross, ed., *Developments in Design Methodology*, Wiley, 1984, pp. 135–144.

[2] F.P. Brooks Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, Apr. 1987, pp. 10–19.

[3] B. Adelson and E. Soloway, "The Role of Domain Experience in Software Design," *IEEE Trans. Software Eng.*, Vol. SE-11, No. 11, Nov. 1985, pp. 1351–1360.

[4] R. Guindon and B. Curtis, "Control of Cognitive Processes during Software Design: What Tools are needed?" in *Proc. CHI'88*, ACM Press, New York, N.Y., 1988, pp. 263–268.

[5] W. Visser and J.-M. Hoc, "Expert Software Design Strategies," in *Psychology of Programming*, Academic Press, New York, N.Y., 1990.

[6] B. Hayes-Roth and F. Hayes-Roth, "A Cognitive Model of Planning," *Cognitive Science*, Vol. 3, 1979, pp. 275–310.

[7] N.E. Fenton, *Software Metrics: A Rigorous Approach*, Chapman & Hall, 1991.

Table 5. The Fusion design process.

Phase	Step	Action
Analysis	1.	Develop the Object Model
Analysis	2.	Determine the System Interface
Analysis	3.	Development of the Interface Model
Analysis	4.	Check the Analysis Models
Design	5.	Develop Object Interaction Graphs
Design	6.	Develop Visibility Graphs
Design	7.	Develop Class Descriptions
Design	8.	Develop Inheritance Graphs

[8] D.L. Parnas and D.M. Weiss, "Active Design Reviews: Principles and Practices," *J. Systems & Software*, Vol. 7, 1987, pp. 259–265.

[9] D. Budgen, *Software Design*, Addison-Wesley, Wokingham, Berkshire, 1993.

[10] G. Friel and D. Budgen, "Design Transformation and Abstract Design Prototyping," *Information and Software Technology*, Vol. 33, No. 9, Nov. 1991, pp. 707–719.

[11] D. Harel, "On Visual Formalisms," *Comm. ACM*, Vol. 31, No. 5, May 1988, pp. 514–530.

[12] D. Coleman, et al., *Object-Oriented Development: The Fusion Method*, Prentice-Hall, Englewood Cliffs, N.J., 1994.

[13] E. Downs, P. Clare, and I. Coe, *SSADM: Structured Systems Analysis and Design Method: Application and Context*, Prentice-Hall, Englewood Cliffs, N.J., 2nd ed., 1992.

[14] M. Shaw, "Comparing Architectural Design Styles," *IEEE Software*, Vol. 12, No. 6, Nov. 1995, pp. 27–41.

[15] J. Cameron, *JSP & JSD: The Jackson Approach to Software Development*, 2nd ed., IEEE Computer Society Press, Los Alamitos, Calif., 1989.

[16] G. Booch, *Object-Oriented Analysis and Design*, Benjamin/Cummings, Redwood City, Calif., 1994.

[17] A. Snyder, "The Essence of Objects: Concepts and Terms," *IEEE Software*, Jan. 1993, pp. 31–42.

[18] D. Budgen, "'Design Models' from Software Design methods," *Design Studies*, Vol. 16, No. 3, July 1995, pp. 293–325.

[19] B.I. Blum, "A Taxonomy of Software Development Methods," *Comm. ACM*, Vol. 37, No. 11, Nov. 1994, pp. 82–94.