THE
UNIVERSITY
OF HULL

What Is Software?
Draft 5

Darren Stephens
Centre for Internet Computing
University of Hull

June 2001

“Ceci n’est pas un pipe”
René Magritte (1926)

Contents

List of Figures

1 Introduction

In the time since the first digital systems were developed the complexity and power of computer systems have increased exponentially, especially with the advent of microprocessor technologies in the late 1960s. This exponential increase in processing power, known as Moore's Law [Intel2001], has resulted in extremely powerful systems able to cope with the demands of the ever larger tasks for which we use them. Partly as a result of this increase (and partly due to other factors which will be discussed later) the nature of software and the methods used to produce it have changed a great deal in parallel with these hardware innovations.

Almost from the beginning of the computer age, it had been apparent to practitioners that software was not a manufactured good, but an engineered item. However, as distinct from other complex (usually physical) engineered systems, software is highly mutable and subject to rapid and significant change during its lifecycle.

The use of the term "Software Engineering" to describe software and process used to create it at first seems to have occurred at the NATO Software Engineering Conference, held at Garmisch, Germany, October 7-11, 1968. In fact, the Garmisch conference provided the first public acknowledgement of an impending "software crisis". [Djikstra1972b]

In the early days of programming, each program had to be hand-written for specific machines and tasks because of different programming environments and operating systems. Even today, in the era of object orientation and so-called code re-use, large portions of computer programs are still written on an individual bespoke basis.

This paper examines the changing nature of software and to arrive at a meaningful definition of what software actually is. In order to do this, two things must be examined: firstly, the evolution of software as an engineered entity and secondly, the differing conceptual models which describe such artefacts. The intention is not to give a rigorous treatment to any of the subjects but to present a broad overview to allow a sufficient definition to be arrived at.

2 The Evolution of Software

2.1 A Software Timeline

Figure 2.1 shows the evolution over time of software and software systems. The diagram aims to highlight significant milestones in software evolution, such as the development of the early languages that provided the archetype for certain paradigms, for instance Simuila67 for Object Orientation or Pure Lisp (1958) for functional programming. Sammet [Sammet1972] [Sammet1976] has carried out fuller surveys of extant programming languages at various times in the past. These surveys are interesting for historical reasons.

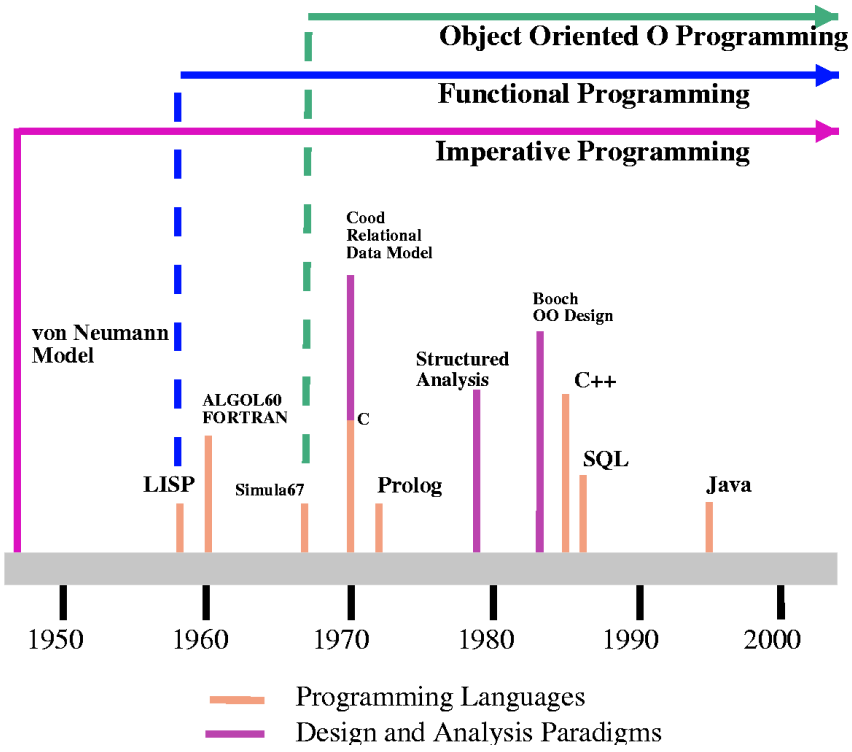


Figure 1: A Software Timeline

The diagram illustrates the following important milestones:

1. 1947 - The developemnt of the von Neumann computer model which dominates to this day.

2. 1958 - LISP, the oringinal functional programming language was developed.

3. 1960 - ALGOL60 and FORTRAN, the prototypical imperative languages were released to the world. The language *Pascal*, developed by Wirth in 1971, was one of their successors, designed as a language to encourage the techniques of **structured programming**. Wirth was a major proponent of this technique.

4. 1967 - Simula67, the first Object Oriented programming language was released. This was succeeded by the language *Smalltalk* in 1972. Its principal use was in the developement of the Dynabook project at Xerox PARC [UCNZ1998].

5. 1970 - Codd's Relational Data Model,the precursor to Object oriented design and analysis, is first published.

6. 1972 - Prolog - the first language for logic programming is developed.

7. 1978 - Structured Analysi and Design, the first memthodology to present a unified model for design and to address analysis issues. This was the first time the entire software engineering process had a sound theoretical base.

8. 1982 - Foundation of the ANSI SQL committee and development of SQL, following on from the CODASYL project. The first SQL standard is published in 1986.

9. 1983 - OO design. The early 1980s saw the first movements in applying similar rigour to OO systems as to structured ones by people such as Booch, Rumbaugh, Jacoboson and others Coincided with the beginnings of developemnt of C++ by Stroustrup.

10. 1985 - C++ released [CPlus2001]. C++ was built on the C language (released in 1970) and its ancestors.

11. 1995 - Java is released. Java embodied the principle of *Write Once Run Anywhere* (WORA). Such an approach was not new, having already been used in UCSD p-code compilers during the 1980s The use of network services and the principle of the applet makes it a major milestone in software, particularly in relation to the web.

2.2 Early Programming Methods

The nature of early computer systems, with non-standard and highly individual environments, usually meant that programming on such systems was also a highly individual pursuit tailored to that environment. Early developments in the software area concentrated on making source code itself more readable (i.e., optimising the program conceptual space) to facilitate rapid debugging.

Little or no attention was paid to the process or logical design used to model the program's conceptual space itself. The flowchart, a representation devised by Goldstine and von Neumann [vonNeumann1947] was used to describe logical flow. It was designed to help map out the algorithmic structure present within a software program although, by the mid 1970s, Brooks [Brooks1995a] was unconvinced that the flowchart had any real value at all except in the initial teaching of algorithmic programming.

There appears to be little or no evidence of any other formalised way of mapping out program design in these early years. Each programmer seems to have used his or her own methods and notation to aid their own mental processes when writing code.

From a contemporary vantage point it appears that Knuth's seminal work "The Art of Programming" was a crystallisation of much of the formalism of the early years of computer programming and design. This may in part be because of the nature of computers and their use at this time.

Computing seems to have been a highly academic pastime and a tool to aid mechanistic computation of mathematically based problems. It is unsurprising that the early languages used for such systems were mainly mathematical in nature such as APL). Booch [Booch1994] refers to the work of Wegner to illustrate the progressive development of these high-order languages.

The use of such languages for development did not aid the process of design. Languages such as FORTRAN were highly reliant on the use of constructs such as `GOTO`. Djikstra [Djikstra1968] commented on the fact that such constructs did not encourage good program construction or design. Furthermore, they most certainly did not make maintenance easier. This is hardly surprising as many of the computing systems of the time were not designed for general-purpose use. It is only as the desire for such general use became more widespread that methods to facilitate it became necessary, prompting Djikstra's opinion.

2.3 Evolution from Early Methodologies

In the early 1970s a number of people including DL Parnas, HD Mills [Mills1971], Wirth [Wirth1971], Gauthier & Pont [Gauthier1970] and Hoare et al. [Hoare1972] began to argue the case for "modularisation", decomposing programming problems into smaller, interconnecting, units, originating from abstractions at higher levels. These arguments marked the beginning of the use of *Structured Design*.

In more specific terms, Parnas [Parnas1972] stated that using items such as flowcharts as a tool for design and development were not a useful way of beginning the process of decomposition.This argument identified the problem that, later, DeMarco and others would introduce as Structured Analysis to address, to model the abstract properties of systems before using the Structured DEsign methods to construct them. Parnas' work particularly provoked disagreement from Brooks in his work "The Mythical Man Month". Brooks saw such a methodology as undesirable because it made the workings of processes less transparent for developers.This was an opinion that Brooks would later change [Brooks1995b]. At this stage however, the development of structured methods was confined solely to design. The methods used to define problem domains and modle the conceptual flows of data through systems were not present.

This was the arena into which first Structured Design, then later, Structured Analysis was introduced. The Structured Design methodology successfully tied together disparate parts of existing modelling, such as Codd's work and the use of State Transition, along with newer ideas such as that of the be-

havioural DFD into a unified entity. A structured system could be considered as three separate but related components; Dynamic (State Transition), Behavioural (modelled using the Data Flow Diagram) and Static (the Entity Relation Analysis which cam from Codd). This is mentioned further in Section 4. Due to the rapid growth in size of software systems and related to the growth of the GUI interface, it became apparent that Structured Methods were not wholly suited to modelling such systems.

THe later introduction of Strucutred Analysis was designed to provide a consistent methodological framework throughout the deveopment of a software product, starting from the definition of the problem domain and existing systems (if they existed), to allow new or improved solutions to replace them.

THe structured methodolgies concentrated a great deal on the concept of *data flow*, being primarily concerned with the movement of data around a system and the transofrmations it underwent as it did so. Such methodolgies embodied Wirth's early assertion of "Program = Algorithm + Data".

Unfortunately, data flow and modularisation did not appear to solve some of the serious problems of development with software sysytems of the time. Other approaches needed to be condsidered. The sheer size of most new systems, in tandem with the need for event driven mechanisms to power graphical user environments (and the size overheads they introduced) were among the factors that provoked the growth of Object Oriented Design.

The idea of Object Oriented systems had existed for some time, proposed by Hoare and Djikstra but were first developed most successfully by Dahl & Nygaard in Simula67, well before the introduction of methodologies for designing software using it. This appears to be a common them throughout the history of software - design and analysis methodologies are predated by the systems they describe, sometimes by many years.

3 Definitions of Software

3.1 An Initial Definition

The Shorter Oxford English Dictionary [OED1993] provides the following definition of software:

> **software**
> *Computing*
> The programs and other operating information used by a computer (opp: hardware) esp. the body of system programs specific to a particular computer.

The word "software" is now also commonly used more generally in the wider population to denote other types of material supplied upon media such as Compact Disc, DVD and videotape whether they are programmatic or not. This definition exists in addition to its more normal meaning attached to computer programs in general. For broad, general use, the previous definition may, in itself, be an sufficient one but it does not describe the subtlety or complexity of a software system nor indeed the nature of the product nor process that produced it. Such a broad definition only serves to dilute the meaning of what software is.

3.2 Formulating a Narrower Definition

In a manner more useful to software practitioners Pressman [Pressman2000] describes software as an "information transformer". We can then, as a useful starting point, present Pressman's definition of software in the following way:

Software Any system or entity (whether singular or composite) which acts to transform information from one state into another.

Diagrammatically this is shown in Figure 3.2

This definition of software is at once both highly descriptive and open to many interpretations. Clearly, it does not apply to electronic or digital systems alone. Somerville defines software in a way that includes not only

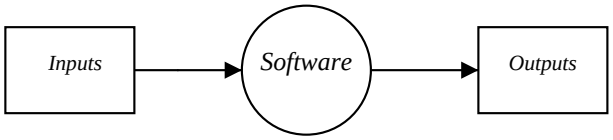


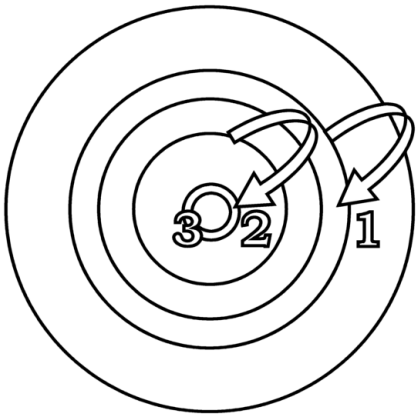
Figure 2: Software as Information Transformer

program code by also any documents or configuration information associated with it [Somerville1989]. Such definitions could quite clearly be applied to mechanical or even to physical systems that serve to transform information. Clearly then, a software system can be described as a complex entity containing both code and non-code components.

3.3 A Conceptual Model for Software - "A Software Onion"

The above description still does not describe a software system in an entirely satisfactory way. Figure 3.3 shows a representation of a layered process desbribing the interactions between each stage of an artefact's development cycle.

A piece of software is (usually) written to be built and executed within a certain environment, either on a real or a virtual machine. This is simply a specialisation of a more general principle: software is highly contextually dependent. A Bourne Shell script without a Bourne shell in which to run it is merely a text file. Again, this model does not have to be machine based. The act of a person reading a document is also a software process, given that an input set is transformed by some process into a final form within the brain of the reader. The transformation may be indeterminate but it has still taken place. We can illustrate the idea of a conceptual model framework for software in the following way: Each succeeding shell (or skin) of the onion is simply a representation (or contextualisation) of the one surrounding it. The move from each space to the next is accomplished by one (or more) trans-



Key:

1. Design Conceptual Space. This contains an abstract space corresponding to a design addressing a particular problem domain.
2. Program Conceptual Space. The expression of the design space in a machine understandable form.
3. Execution Context. The context chosen for the execution of the programmatic representation to produce the final artefact.

Figure 3: A Software Onion

forming processes. It may be helpful to consider two examples to illustrate this process:

1. In the first scenario, the move from level 1 (signifying a UML design in response to a particular problem domain, for instance) to level 2 (a C++ or Java source code structure) may be accomplished by use of a CASE tool. This effects a transformation of a design framework into machine generated code. The transformation from two to three may then be accomplished by compilation or interpretation (for instance by use of the gcc compiler in a Linux environment) to produce final, executable code. This code would only be meaningful as executable code in that particular context. It would not execute within a Windows environment, for example. Hence, in that context it would not be

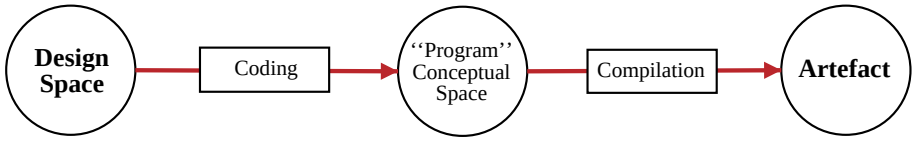


Figure 4: Generic Software Transform Model

software.

2. In a second scenario, the move from level 1 to 2 may be accomplished by devising a DTD or schema for an XML document type to be used for describing a particular set of content. This may describe the structure of some real world entity like a technical manual. This XML entity may then be "executed" in the execution context by a browser interpreting the entity and outputting the transformed entity in some form. This would only be possible in a context where the DTD or schema was understood by the interpreting agent

Both of these examples are illustrations of a more general porcess at work. This can be illustrated in figure 3.3:

The second scenario allows for the possibility that a document can also be considered a piece of software. In this case, it is only distinguished by the method of transformation from programmatic space to execution context (i.e. a browser acting as an interpreter, having inputs and output in a visual form). This is an idea that can be explored at a later point.

The initial working definition requires refinement to accommodate this principle. This is done by extending the definition to say that program code (for want of a better expression) and the process that describes its production is a conceptual structure for facilitating the transformation of information. The code itself (i.e. the product) is merely a context dependent manifestation of this conceptual structure, requiring both its context and input(s) to

function. Software engineering, the process of building software, makes this distinction between the product and process of software development.

3.4 The State Model of A System

State has been widely discussed for many years as a computing concept, most notably beginning with the ideas of the Turing Machine and the Finite State Machine [Kleene1956]. The von Neumann machine model could itself be described as a Finite State Machine, giving a formal mathematical basis for the use of the computer for problem solving in mathematics. This was of course, the first real application of computing systems.

An imperative model of a software system implies that for a transformation to take place, the information undergoing the transformation (and possibly the transforming agent itself) experience one or more changes in their state. This also happens in declarative programming but at a much lower level, hidden from the programmer. Although this was not a key part of methodologies such as DeMarco's [DeMarco1979] it quickly came to be seen as vital to others like McMenamin & Palmer [McMenamin1984] or Ward & Mellor [Ward1985]. The former especially discussed the concept of state within complex systems. This approach showed that a system could be modelled not only in terms of a flow of data but also in terms of the changing state of such a system as this flow happens.

Initially, state considerations, such as McMenamin & Palmer's event partitioning model, which identifies input and outputs for process as well as the stores required, was temporally unfocused or undefined. Ward & Mellor (and others such as Pirbhai and Hateley [Hateley1988]) refined this idea to connect state with time constraints for modelling real-time systems, mainly by adding temporally information and dependencies to the behavioural model (DFD). The structured approach to software thus proposed a tripartite model:

- The Dynamic Model models the behaviour of a system in response to outside stimuli or inputs. This model describes the various states in which such a system can be configured at some point. This part of the Structured Model was accomplished by the use of the State Transition Diagram (STD). The dynamic model is still an important of part of

object oriented methodologies, specifically in relation to event driven systems.

- The Static Model models the relationship between data entities with a system. This also incorporates the Data Dictionary; effectively the static structures of the data components themselves. This part of the Structured Model was accomplished by use of Entity Relation Analysis (ERA)

- The Behavioural Model models the flow of data through the system. Its implementation in the Structured method is accomplished using the Data Flow Diagram (DFD). In the traditional sense, this part of the model encapsulates the algorithmic properties of a software system. This is slightly harder to show in Object-Oriented systems although it is done more indirectly by showing dependency information within the data model.

4 A Taxonomy of Software

4.1 The Software "Family"

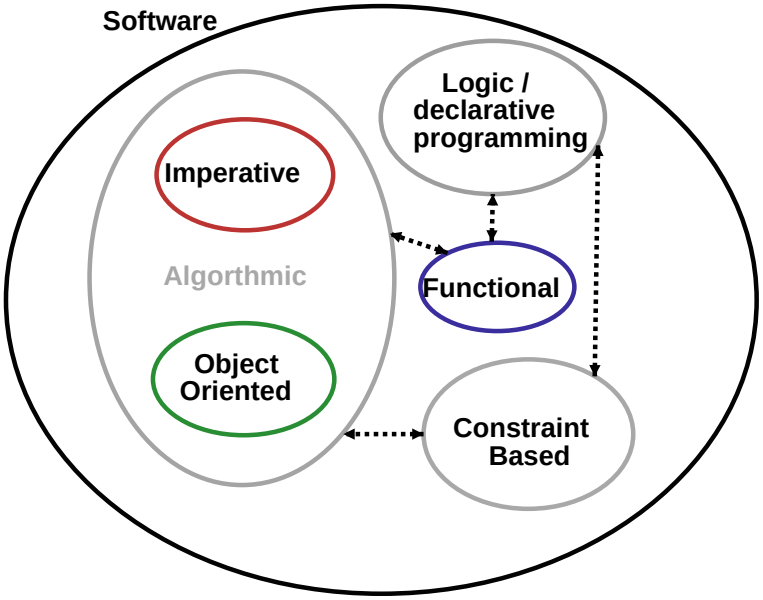


Figure 5: The Software Family

Bobrow and Stefik [Bobrow1986] define the following kinds of programming style:

- Procedural programs are algorithmic and generally concerned with data flow. This form also includes scripting languages and can also include functional programming.
- Object Oriented programs, where code may have algorithmic elements but there is more concern with mapping software objects to real world ones where possible.
- Declarative prgrams, encapsulating logic and rule based programming. Declarative systems may also utilise imperative or functional components but have much in common with O-O systems.

- Constraint Oriented programs. including self-organising systems such as neural networks and genetic algorithms . Software of this type can use code generated using any of the other styles for resons which will be explored later.

Some of the above families are covered in more detail by Bal and Grune [Bal1994a]. To this list, it is proposed that another category could be added, given the conceptual space model described for software earlier.

- "Semantically" Oriented
  here there is no "flow" as such. This style is concerned with the organisation of information in order to be semantically meaningful. This form includes mark-up languages and formatting languages such as Postscript and TEX. It can be argued, however, that such a paradigm is merely a variation on a declarative one. Each element could be said to describe a clause and the tree derived from parsing maps the relationships between such clauses for each document.

Bobrow and Stefik state that a style is simply a way of organising programs based on a conceptual programming model and an associated language to make the programming style clear [Bobrow1986].

Each style in this sense is interpreted as equally valid and indeed, each particular style has applications and context in which they may be most successfully used. This goes part of the way to describing the onion-skin model described earlier.

4.2 Imperative Programming

One interpretation of the idea of transformation is that software involves a functional or algorithmic process. This is a key element of views of software such as Structured Analysis [DeMarco1979] that treats the information transformation process as a flow of data through a number of functional components. It corresponds with the view taken by Functional Programming (q.v.), albeit in a slightly different way, This also corresponds with our starting definition as we can define the application of functional components of a software process in well-understood manner.

For instance, consider a tranfsormation of the form, transofrmaing an inital set *x into a final set* y:

$$y = f(x)$$

We can consider a software process to describe a flow through a number of such operations that could be applied sequentially, viz.

$$x_a = f_a(x)$$

$$f_{a+1}(x_{a+1}) = f_{a+1}(f_a(x_a))$$

$$\vdots$$

$$y = f_n f_{n-1} \ldots f_{a+1} f_a(x)$$

Figure 6: Functional consideration of an algorithmic process

This representation, however, could be interpreted in two different ways: imperatively, by being able to create assignments at each intermediate stage of evaluation and functionally by considering the whole without these intermediate actions. In this context, imperative systems may also encounter problems because of evaluation order: this is a problem that would not occur in the functional context.

4.3 Algorithms

In many views of programming and software production until the introduction of logic programming (q.v.), the role of the algorithm is of vital importance. Knuth [Knuth1969], states five properties that define an algorithm:

1. **Finiteness**
   The algorithm has a finite number of iterations or repetitions. That is, there must be some defined exit condition.

2. **Completeness**
   The algorithm encapsulates the problem domain completely.

3. **Definiteness**
   The algorithm encapsulates ONLY the problem domain and does not focus on irrelevant concerns.

4. **Input**
   The algorithm requires input conditions or states to work from, or must be able to define them.

5. **Output**
   The algorithm should produce discernible results or outputs.

6. **Effectiveness**
   The algorithm should, in some way, effectively address the domain of the problem.

It is arguable whether this final item, although extremely desirable is a defining characteristic of every algorithm. It is possible to define a set of algoirthms to solve a particular problem which would have markedly different levels of effectiveness. Each would, however, still be a "valid" algorithm in as much as it provides a working solution.

The main implication here again is that software, being algorithmic, is essentially a sequential, deterministic process (specifically in terms of the finiteness condition). This determinism *only* applies to the algorithm itself, *not* to the results of applying it. In other words, the algorithm maps a set of possible conditions and outcomes. The results of applying them are not necessarily known at the time of execution. This, in fact, as discussed later, may be considered the major difference between imperative (algorithmic) and logic programming

4.4 Object Oriented Systems and Programming

The development of Relational Data Modelling by Codd [Codd1970] and others in the early 1970s had successfully introduced the idea of Entity Relation Modelling as an effective tool for data modelling. In fact, in the abstract of his 1970 paper, Codd himself proposed that one of the main reasons for developing his relational model was to hide the underlying structure of data from the user.

Even in the early 1970s this is a common concern shared with Parnas ideas on modularisation. The Parnas idea posited that, instead of simply decomposing programs as subprograms, that each module should have a "responsibility assignment". In other words, Parnas was proposing information hiding (along with Codd) as well as being an early proponent of the abstraction principles that object orientation would expand upon. Many considered this system to be a powerful tool for data modelling as it closely mapped with human traits of characterisation by categorisation. Codd himself recognised this fact.

Structured Analysis integrated ERA to provide a description of the static model of the software system. Software practitioners such as Booch, Rumbaugh and Jacoboson (separately at first, together later in the UML methodology [UML1999] as well as Yourdon (again) all saw the promise of using such a system more prominently in software design. Previously, software systems had been seen as large entities able to be split into parts and developed separately to model the flow of data around a system. Each of the components or modules could manage a part of the flow. This modularisation process had begun with Wirth, Mills and Parnas, as described earlier.

The O-O system departed from this and (like the model used for Unix system commands) postulated that systems were better modelled as smaller, communicating independent entities. To use the object oriented phraseology; such systems should be componentised, cohesive and lowly coupled.

Proponents of O-O claimed that it offered numerous benefits, including:

- Code reuse
- Component based systems to allow code "manufacture"
- Modular abstraction
- Scalability

These benefits were achieved by use of the fundamental concepts used to build object-oriented systems:

- **Abstraction** Following on from the concept of modularity, objects should connect in well-defined visible ways. Objects need not know

about internal implementation issues within other objects with which it communicates. In theory this allows reusable components to be built and allows them to be used to build very large projects more easily. In other words, objects promote scalability.

- **Encapsulation** Pbjects should not be thought of as data structures, to be fully formed an object should also define what actions it can perform, how it communicates with other objects.
- **Polymorphism and Inheritance** This is possibly the key feature of object orientation; objects can form hierarchies and have the ability to be classified according to type. New types can be easily derived from existing ones.

Booch also made the distinction between algorithmic decomposition of the kind described by Wirth and others and Object decomposition, providing a level of abstraction in defining the entities participating in a software system. In many senses, this did indeed echo the entity-based approach to relational data analysis begun by Codd.

The communicating entities approach had also been used in the idea of semaphoring between modules in real-time systems, particularly by Ward and Mellor. This model allowed for large systems to be built more easily and, according to its supporters has encouraged code reuse and highly component-based systems.

4.5 Alternatives to the Imperative Model

As can be seen from Bobrow and Stefik's taxonomy, one of the styles of programming they describe is that of declarative programming. Declarative programming re-interprets one of the fundamental assumptions made so far in this discussion of software and software systems: software is essentially an algorithmically based entity. The early proponents of software design and programming had thought this way, giving rise to ideas like the flowchart and later, other analysis tools as has already been mentioned. However, this is not a wholly valid assumption to make.

Imperative programming works by executing a sequence of defined commands (be they singular or grouped). Part of this process involves changing the values of variables within the domain of the program using explicit assignment operations. One result of this is that the order of command execution within a program can affect the values of variables in different ways. In effect, the programmer has to exert some low-level control over the program conceptual space, explicitly controlling the state of the system in the code.

In declarative programming the possibility of confusion regarding evaluation order does not happen. Declarative programming was developed to hide this lower level process away from the programmer. It was seen as both a source of error in thinking and an incorrect level of abstraction when considering software problems, mainly because the act of assignment made evaluation an important consideration. Functional programming developed to allow a propositional calculus to be implemented in a machine environment. A propositional calculus describes the functional process in more rigorous mathematical terms.

The nature of this style, lent (and still lends) itself to design using formal methods which provide a more mathematically rigorous and complete design framework for such software to operate within. Indeed, the illustration shown in Figure 4.2 corresponds more to this view of software than to purely imperative programming. The programmer thus has little control over the state of the system once the operational parameters are defined and boundary conditions specified.

Declarative programming is not new, So-called "Pure" LISP, although not strictly a purely functional programming language, certainly contained many such features. It was first introduced in 1958 [MacLennan1990].

"Common" Lisp, introduced later however, contained many imperative programming constructs. By the middle of the 1970s, languages such as Lisp and LOGO [Chakrabary1999] were in widespread use. To abstract the process of explicit assignment from the programmer the programming environment itself must be responsible for it. One of the main methods of doing this is by use of recursion, placing great emphasis on stack operations.

By the early 1980s many were concerned that, to increase computing power, systems would have to become much more adept at processing in parallel. To handle this underlying complexity, authors of software should be divorced

from the low-level problems of algorithmic programming as such processes were prone to error. Kowalski [Kowalski1979] (amongst others), proposed a fundamental rethink in the way programming was pursued. These systems relied much more on logical than functional composition, leading to the birth of logic programming.

Where functional programming removed the need for low level assignment, logic programming sought to remove the need for functional considerations entirely. Systems could be devised that would operate under conditions determined only by the relationships between data entities. Systems of this nature involve the use of a predicate calculus, most famously described in the first instance by Alonzo Church [Church1936], to represent these relationships. The engine which evaluates these rules may contain functional (or even imperative) components at lower levels but seeks to make the programmer think in purely abstract terms.

Logic Programming relies on the use of the *Horn Clause* [Bal1994b]. A Horn Clause is one which states that a particular condition is true if zero or more other conditions is also true. A further definition is that a horn clause with no conditions is a fact. Unlike imperative logical evaluations, however, each of these other conditions could be evaluated in parallel. The goal of the logic programmer is to reduce a problem domain into a set of horn clauses which can be evaluated. As a consequence of parallelism, however, logic programming is non-deterministic in the sense that evaluation orders are not necessarily known.

Work by Eisenstadt and Brayshaw [Eisenstadt1988] suggested however, that there was a conceptual link between algorithmic, functional and object-oriented approaches. A more rigorous discussion of logic programming will ont be undertaken in this paper

4.6 Self-Modifying Systems

Self-modifying systems, which include Genetic Algorithms (GAs) and Neural Networks), unlike the algorithmic methods described earlier are essentially non-deterministic. GAs stem form initial work by Holland [Holland1975] but ideas about evolutionary programming in general can be found as far back as Fogel [Fogel1966]

A diagrammatic representation of this process is shown in 4.6 [1].

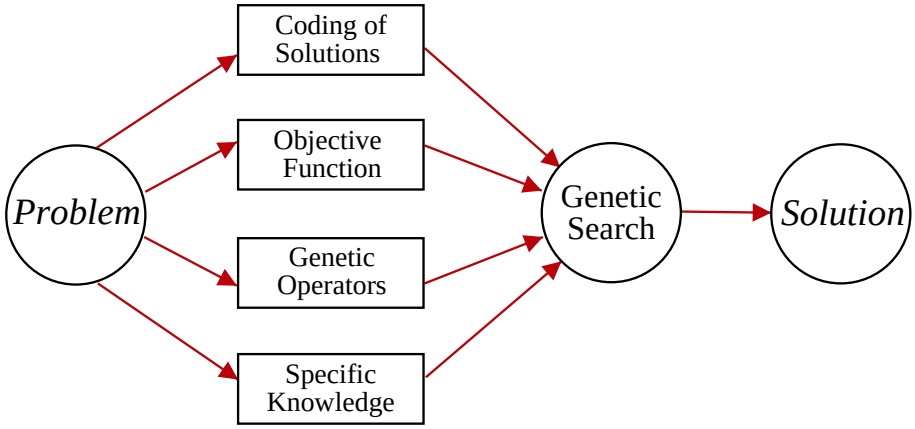


Figure 7: Genetic Algorithm Heuristic

Instead of well-predicted algorithmic systems, the GA system involves the use of a "fitness function" to determine the evolution of the algorithm. Thus, a programmer has no control over the algorithm or the system's state until completion. The fitness function itself does not have to be algorithmic, rule based or functional tests are also allowed (and in some cases may be preferable).

The genetic search is an iterative sequence (a generation) through the following stages:

- Fitness assignment
- Selection
- Replication
- Recombination Crossover
- Mutation

[1] This diagram is adapted from The Genetic Algorithm Toolbox [GAT2001]

Replications that do not meet the speciifed fitness criteria are then discarded. In theory this produces successive generations of solution that satisfy the fitness function more closely. The self-modiying system is therefore non-deterministic. Not only are the outcomes unpredictable but the algorithms used to arrive at that outcome are also not known at the outset as these solutions are generated during the execution cycle.

A more detailed consideration is beyond the scope of this document but readers are advised to refer to the work of Fogel [Fogel1994] for a more rigorous discussion.

5 Conclusions

A full definition of software is a particularly difficult task to specify satisfactorily in the space available and doubtless this work could be expanded at a later date to include areas that have not been covered in greater depth here.

There, is, however, a distinction between a software artefact (or product) and the process that produces it, even if some of this information is incorporated as part of the final deliverable artefact itself.

As has been seen, however, software systems do exhibit common characteristics:

1. A software system transforms information from one state into another.
2. The result of the application of a software process is an artefact. But the items used to create that artefact can also be described as software. The artefact itself may be transient, as is the case with scripts.
3. A software system must have inputs and outputs, even if they are implicit or self-generated.
4. A software system must specify its nature and a context in which it is to take place. For a software system to be classed as such, the context **must** be specified.
5. Software defines a conceptual space (or series of conceptual spaces) which provide a representation of a solution to a particular problem domain.
6. A software artefact does not necessarily have to be algorithmic; the software can take a number of forms (as described by Bobrow and Stefik's taxonomy.

There is also no necessary paradigmatic system for software development. Most of the paradigms developed in the past 50 years have, to this day, some appropriate or useful application at some point or in some project. Again, the choice is a matter of problem context and representation.

Central to these conclusions is the assertion of the importance of context to the perception of software. In this context, a software system can be considered to be machine based, although this does not have to be the case.

Should an artefact satisfy the conditions above it could then be considered to be software, meaning it corresponds to the following definition:

> **Software**
> A system or entity, whether singular or composite, defining a conceptual space which can be applied in a particular context to effect the transformation of information from an initial state to a final one.

References

[Backus1978] Backus J. *Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs*, CACM 21(8) Aug 1978 pp613-641

[Bal1994a] Bal HE, Grune D, *Programming Language Essentials*, Addison-Wesley 1994 ISBN 0-201-63179-2 p8

[Bal1994b] *ibid.* p166

[Bobrow1986] Bobrow DG, Stefik MJ, *Perspectives on Artifical Intelligence Programming*, Science, vol 231, Feb 1986, p951

[Booch1994] Booch G, *Object Oriented Analysis and Design With Applications (Second Ed.)*, 1994 Benjamin/Cummings, ISBN 0-8053-5340-2 pp28-29

[Brooks1995a] Brooks FP, *The Mythical Man Month (Anniversary Edition)*, Addison Wesley, 1995 p271

[Brooks1995b] Brooks FP, *The Mythical Man Month (Anniversary Edition)*, *ibid.*

[Chakrabary1999] Chakrabary A, Graebner R, Stocky T, *The Conception of LOGO*, 6.933J, Dec 1999, MIT (http://mit.edu/6.933.www/LogoFinalPaper.pdf) pp13-16

[Church1936] Church A, *A Note on the Entscheidungsproblem*, Journal of Symbolic Logic, Vol 1, pp40-41, 1936

[Codd1970] Codd EF, *A Relational Model for Large Shared Data Banks*, CACM 13(6) June 1970, pp377-387

[CPlus2001] CPsusPlus.com, *History Of C++*, Available online at: http://www.cplusplus.com/info/history.html Accessed June 22, 2001

[DeMarco1979] DeMarco, T, *Structured analysis and System Specification*, Prentice-Hall, 1979

[Djikstra1968] Dijkstra EW, *Go To Statement Considered Harmful*, CACM, 11(3), March 1968, pp147-148

[Djikstra1972a] Djikstra EW, *The Humble Programmer (1972 ACM Turing Award Lecture)*, CACM 15(7) July 1972 p863

[Djikstra1972b] *ibid.* pp859-866

[Eisenstadt1988] Eisenstadt M and Brayshaw M, *The Transparent Prolog Machine (TPM): An Execution Model and Graphical Debugger for Logic Programming*. Journal of Logic Programming, 5(4), 1988.

[Fogel1994] Fogel, DB, *An Introduction to Simulated Evolutionary Optimization*. IEEE Trans. on Neural Networks, vol. 5, No. 1, pp3-14, 1994.

[Fogel1966] Fogel LJ, Owens AJ. and Walsh MJ, *Artificial Intelligence through Simulated Evolution*, John Wiley, 1966

[GAT2001] *Genetic and Evolutionary Algorithm Toolbox for use with MATLAB*, Online version at http://www.geatbx.com/docu/index.html (Version 1.92, last modified November 1998, accessed March 30th 2001)

[Gauthier1970] Gauthier, R,Pont,S. *Designing Systems Programs*, Prentice-Hall, 1970

[Hoare1972] Hoare CAR, Dahl O, Djikstra E, *Structured Programming*, Academic Press 1972

[Holland1975] Holland, JH, *Adaptation in natural and artificial systems*. The University of Michigan Press, 1975.

[Hateley1988] Hateley DJ and Pirbhai IA, *Strategies for Real-Time System Specification*, Dorset House Publishing, New York USA, 1988.

[Intel2001] *Moore's Law*, Online version , http://www.intel.com/intel/museum/25anniv/hof/moore.htm

, Intel Corp - Accessed 2.3.2001 last date of modification not specified

[Kleene1956] Kleene, SC, *Representation of events in nerve nets and finite automata*, in C. E. Shannon and J. McCarthy, eds, Automata Studies, Annals of Mathematics Studies No. 34, Princeton University Press, 1956, pp. 3-42.

[Knuth1969] Knuth, D, *The Art of Computer Programming*, Vol1: Fundamental Algorithms, Ch1 pp4-6

[Kowalski1979] Kowalski, RA, *Algorithm = Logic + Control*. CACM 22(7): 424-436 (1979)

[McMenamin1984] McMenamin, S Palmer, J, *Essential Systems Analysis*, Prentice Hall, 1984

[MacLennan1990] MacLennan, *Functional Programming: Practice and Theory*, Addison Wesley, ISBN 0-201-13744-5, 1990, pp23-26

[Mills1971] Mills HD, *Top Down Programming in Large Systems*, R Rustin ed. Prentice-Hall, 1971

[OED1993] *The Shorter Oxford English Dictionary*, Oxford University Press, 1993, 4th Edition ISBN 0-19-861134-X

[Parnas1972] Parnas, DL, *On the Criteria To Be Used in Decomposing Systems into Modules*, CACM 15(12), December 1972 pp1053 - 1058

[Pressman2000] Pressman, RS, Software Engineering, *A Practitioner's Approach (European Adaptation) Fifth Edition*, McGraw-Hill, 2000 ISBN 0-07-709677-0

[Rumbaugh1991] Rumbaugh, J et al. *Object-Oriented Modelling and Design*, Prentice Hall International, 1991

[Sammet1972] Sammet J, *Programming Languages History & Future*, CACM 15(7) July 1972 pp601-610

[Sammet1976] Sammet J, *Roster of Programming Languages*, CACM 19(12) Dec 1976

[Somerville1989] Somerville I, *Software Engineering - Third Edition*, Addison-Wesley 1989, ISBN 0-201-17568-1 p3

[UCNZ1998] University of Cantebury, Christchurch, New Zealand *Basic Aspects of Squeak and the Smalltalk-80 Programming Language* Available online at http://kaka.cosc.canterbury.ac.nz/ wolfgang/cosc205/smalltalk1.html#history, Last modified June 22, 1998

[UML1999] Rumbaugh J, Jacobson I, Booch G, *The Unified Modelling Language Reference Manual*, Addison-Wesley, 1999

[vonNeumann1947] von Neumann J & Goldstine HH, *Planning and Coding Problems for an Electronic Computing Environment*, Part II vol I, report for US Army Ordinance Department, 1947 - Reprinted in von Neumann J, Collected Works, AH Taub ed. Vol. V, Macmillan pp80-151

[Ward1985] Ward PT & Mellor SJ, *Structured Development for Real-Time Systems*, Volumes 1-3. Vol. 1: Introduction and Tools, Yourdon Press (Prentice-Hall) 1985.

[Wirth1971] Wirth, N, *Program Development by Stepwise Refinement*, CACM, 14(4), April 1971, pp221-227