Software for emergency shut down systems

J.R. Borer

Department of Mechanical Engineering, Brunel University, Uxbridge, Middlesex UB8 3PH, UK

1. BACKGROUND.

The consequences of systematic software errors or 'bugs' in safety critical software can be catastrophic, and there is increasing anxiety concerning such software, particularly in the nuclear industry. Following the Piper Alpha disaster and publication of the report of the inquiry, the causes of this anxiety can be seen to be equally relevant to the offshore industry. The fundamental cause is acknowledged to be almost total inability to discover systematic software errors by testing, and consequential lack of any viable certification process for software, in contrast to hardware systems.

Project specific software for emergency shut down (ESD) systems in nuclear, petroleum and process industries embodies logic relating states of large numbers of detection devices to required states of a large number of shut-down devices such as motorised valves, electrical contactors, motor starters etc. It must be designed and implemented in such a way as to provide the best possible chance that it will always act to prevent a dangerous situation from arising, and never to cause one. This will depend on the reliability of hardware, the known integrity of software and the thoroughness of the HAZOP analysis and specification of system requirements. No satisfactory way has yet been established by which the reliability of an ESD system can be evaluated. Hardware reliability can be assessed in statistical terms such as 'mean time to fail' (MTTF) and 'mean time between failures' (MTBF), but it is not easy to envisage similar parameters with which to quantify the integrity of software, since systematic errors only result in failure when a specific but unknown set of operating circumstances arise.

A report entitled 'Software in safety-related systems', commissioned by the DTI, carried out jointly by the British Computer Society and the IEE and published by the latter in 1989 states, in chapter 2, page 2-7, para 2.29 that: "design errors are a major factor in the overall failure-rate of systems. The errors may be faults in specifications, or failures of the software to meet its specifications;" A consultation document issued in June 1990 by the Interdepartmental Committee on Software Engineering (ICSE) and entitled 'SafeIT' also says, in the Forward, that: "we are constantly being reminded that faulty software is the rule rather than the exception." and in the introduction: "there has been a growing realisation that existing techniques for design and assurance of safety related systems need significant modification and extension to accommodate the new technology;" (IT) ".... and improved techniques of software .... system development are required in
2. INTRODUCTION.

Good project management is of paramount importance in the design of multi-discipline engineering systems, such as an offshore production platform; it has long been recognised by project managers that there is something 'different' about design of software for safety systems within such projects. 'Perfect' specification of the requirements is demanded 'up-front', and this is simply not possible in the real world. Any changes to the ESD system requirements after the start of software design and implementation may and often do result in disproportionate changes to the software already programmed. Moreover, the greater the fraction of the total software which has been completed, the greater the impact of any change initiated by another discipline. This state of affairs is inevitable in any multi-discipline engineering project and the older, more traditional engineering design disciplines have long since learned to accept it. Design changes by other engineering disciplines invariably continue until late into any such project and are bound therefore to affect software design and implementation. This problem is greatly exacerbated by the need for the main design contractor to subcontract design of ESD system software to either a 'software house' or, more usually, to the manufacturer of the system hardware. Software engineers who then design the system on the basis of 'cause and effect' diagrams produced by a petroleum design contractor, are often ignorant of the nature of the other engineering disciplines involved in the overall design of offshore installations. More importantly, they are not used to the continual design changes which inevitably take place within such a multi-discipline engineering design team. However, the start of software design cannot be delayed until a closed ESD system specification is available, because it would not then be possible to complete the platform design in time to meet project schedules. There is therefore a need to change the nature of the software design processes in such a way as to make them tolerant of the exigencies of such multi-discipline engineering design.

3. SOFTWARE DESIGN.

The design process for any engineering system starts with specification of the client's requirements. This should define only what the system is required to do; it should not address itself in any way to how the requirements are to be met. Nevertheless, such a specification would be of little use if it demanded the impossible, and in practice it will inevitably be constrained by such considerations. These constraints are embodied in the established practices of the particular industry; for an ESD system the established practice is that a design contractor produces a client's specification in the form of 'cause and effect' diagrams supplemented by a written description of the philosophy of the system and details of any special requirements which cannot readily be described in diagrammatic form. These documents constitute a 'functional abstraction' of the system which is then used as a starting point to construct a functional specification: however, they do not of themselves constitute a functional specification of the system. As with design of any engineering system, the client's requirements have to be reconciled with constraints imposed by materials and components which the system builder has at his disposal; a functional specification can therefore only be drawn up by the system builder.
The problems with the ESD software system start when the system design is contracted out by the main design contractor to the system builder, an equipment manufacturer or software house. The main design contractor's definition, in abstract terms, of what the system is required to do, is untrammeled by considerations of the system builder's constraints and it is the process of translating these client's requirements into a functional specification of the system which is at the root of most of the trouble. In particular, the trouble is caused by the fact that the specification of the client's requirements, in the form of cause and effect diagrams, will inevitably be subject to repeated and often extensive revision throughout the design and implementation of the software.

Design of any engineering system rests for practical reasons on identification of suitable, available components. If such a component exists, its function is already defined and cannot be changed, or can only be changed in certain very restricted respects. If totally new components are needed, the designer must define their functions so as to meet the system specification. However he will rarely if ever be called upon to design all the components for his system; in today's world this is nearly always uneconomic. Engineering system design can be seen to require a 'bottom up' approach. Conventional software design methods on the other hand are 'top down' by nature; the total design task being divided up on the basis of the functions which must be carried out by the software, on data supplied at 'run time'. This is why there is so little reuse of ESD software.

Procedural source code is in fact widely reused, but the 'modules' bear no relation to any identifiable component of real time systems. Rather they represent one programmer's solution to some specific problem in the implementation of a particular task. A typical example is inversion of a matrix, which is a routine which recurs in many mathematical solutions to engineering problems, including real time problems. The reusable code must be able to accept any data which might be fed to it at run time, which in this case poses no great difficulty, since the data structure can only be a matrix (array) and allowable 'data types' real values. However, reusable code to carry out a search of a 'data base' is a different matter. Different 'database' structures are possible (tables, lists [sorted or unsorted], matrices, arrays, 'trees' etc.) and to be of general applicability, such a procedure must accommodate any possible data type:- literal, integer, real values or even logical (or boolean) values.

In the software sense then a software module will represent a data handling procedure whereas, in an engineering sense, it should correspond to a component part of the physical system. However, software for ESD systems is not, and cannot at present be designed in the form of reusable modules and as a result each system must be considered in its totality for design, testing and certification purposes. The reason why perfect specification of software is demanded up front is thus directly related to design and implementation methods, which do not enable designers to produce reusable software modules in either the engineering or the software sense. This explains why there are the problems of design and project management referred to above, why certification is more or less impossible, and why systematic design faults are so prevalent in contemporary ESD software systems.

Because software for ESD software systems is not designed in the form of reusable modules, each system must be certified in its
software modules so that these rather than whole systems can be certified. Thus it is necessary to define and develop techniques and procedures for design, implementation testing and certification of reusable software modules, each representing a component part of an ESD system, which could be reused in other ESD systems. Such modules would provide a basis for solution of the problems of design, project management and certification of ESD software systems. Not only would changes to software resulting from design changes instigated by other disciplines then be less extensive, but they would be easier to identify and specify.

The concept of a 'software component' which can support a bottom up, rather than a top down design philosophy appears then to offer the best chance to establish such new techniques for design of ESD software. However, as with conventional engineering hardware system components (such as a centrifugal pump and the associated pipework system for instance), ability to interface or match software modules must be essential to establishing design and implementation techniques which can support the concept of reusable software components for real time systems.

A new framework is required under which reusable software modules can be designed, constructed tested and operated. This might be based on a suitable compilable object oriented programming language; application software modules could be designed, programmed, compiled and assembled into machine code, which could be certified to run only on a specific processor. Using a compilable language, object orientation could be applied at all levels; data structures as well as engineering components being defined as objects. For example a linked list and a centrifugal pump could both be perceived as objects. Not only could modules be reusable in a mechanical engineering sense, but also a minimal amount of code would be generated. Each module thus created, could be type tested and approved in compiled and assembled form, to run only on a specified processor. In addition, whilst any subsequent changes to a module would necessitate retesting and recertification, this would require minimal new code. However, an ESD system is a real-time-on-line system and requires extensive 'front end' facilities to support interfacing with measurement sensors and transducers. The development of such facilities typically absorbs thousands of man-hours of programming effort. Thus it may prove more practical to use some existing software environment, which already includes such front end facilities, to implement reusable software components of the ESD system, rather than to develop a new Object Oriented language and then to use this to implement entirely new application software. Whether this approach is possible will depend on whether object oriented design can be implemented in other than Object Oriented languages.

4. SOFTWARE DESIGN METHODS.

The EAR data model provides a basis for conceptual design of real time software which has a counterpart in conventional engineering design techniques in the form of flow diagrams and detailed equipment specifications. This analogy breaks down however because nothing in this model corresponds to an interface between components of a system. For instance, the mechanical engineer must define the matching of a pump with a system of pipework (two components of a larger system). To do so he amalgamates the operating 'characteristics' of both components to generate the
operating characteristic of the system. This involves consideration of such concepts as 'degrees of freedom', 'state variables' etc. The concept of relationships within this common conceptual model for a data structure thus falls far short of the engineering concept of an interface, and this mitigates against parallel development of software and conventional engineering systems. The latter include multi-processor digital hardware systems in which interfaces are designed on the basis of 'protocol' - a set of rules. It is not difficult to appreciate that this lack springs from retention of 'top down' rather than 'bottom up' decomposition of tasks when one-man/one-program became impracticable.

In the engineering sense software modularity must be defined in terms of reusability in different, but generically similar systems. The concept of 'interfacing' must be introduced into the software design process. This in turn requires that the concept of 'transparency', established in the hardware engineering field, be introduced into software design.

5 SOFTWARE IMPLEMENTATION.

Design of software system components should be analogous to design of a conventional engineering system. E-R diagrams to engineering flow sheets and attribute definitions to detailed equipment specifications. Extending this analogy, implementation of software by a programmer should correspond to design and manufacture of the hardware components of a conventional engineering system. The skills of a programmer in designing and implementing software mechanisms defined by the data model, parallel the conventional engineering skills applied to design and construction of hardware components. The 'client' relationship between the component manufacturer and the system constructor can be seen to be crucial.

A processor operates on data which constitutes an 'image' of something real. Software comprises both instructions and data for operations which the processor carries out on data. The concept of a software object is thus of something real represented by software. A class of objects can be perceived as a system component or a data type; as an 'external' or an 'internal' object. An (object) class can be defined either to generate an image of a physical component of a real system or a message to be passed within a system. Both constitute 'instances' of a class, an object. Thus a key to a possible new approach to design and implementation of real time software systems is decomposition of tasks on the basis of objects, rather than functions; functions being defined as 'features' of a class of objects. This approach addresses the problem of the transparent interface identified as crucial to implementation of software components of a real time engineering system. One class of objects, a 'client', can generate a message instructing another class to perform an operation which is one of its features, and so to generate an object which is a data image of something real. Unfortunately, object orientation is in direct conflict with functional decomposition, and for this reason development of software design, which has been predominately along the path of functional decomposition, has only very recently embraced object orientation for either software design techniques or programming languages. However the latter are now seen, at least in some quarters, to provide a possible answer to the problems of creating reusable software modules. The software object class is perceived by software engineers, not as a representation of a physical component of an engineering system, but as a data structure.
However at the same time it is seen as a machine since the term software encompasses instructions as well as data. A linked list for instance, which is normally regarded as a static repository for data, is perceived under object orientation as a machine with its own internal functions for searching, inserting, deleting etc: these are its 'private parts'. This introduces the concept of state to the linked list and provides a basis for an interface or 'public part'. The latter must comprise only what a 'client programmer' needs in order to utilise the list. Thus the concepts of a software object class either as a data structure or as a component of an engineering system appear to have a lot in common. A language which supports reusable data structures is likely also to support reusable engineering components. Whether a software environment, implemented in a non Object Oriented language would be capable of supporting an Object Oriented design, as outlined above, is another question entirely.

6. THE INTERFACE.

Any system component must 'stand alone'; that is it must be possible to interface it with another component without first having to modify it in any way. It must therefore have a 'closed specification': any change in specification after it is closed will result in a new component being created. For a programmer this demands that a module be a 'syntactic unit'; that is a unit which is separately compilable. Also, unless a software module is transparent, its specifications cannot be considered to be 'closed' since it may have to be modified in some way in order to interface it with another module. Internal functions of a 'reusable software module' must thus be 'hidden', whilst its interface with another module must be 'public'. This concept has its counterpart in the mechanical system engineering field. From a 'clients' point of view a centrifugal pump is fully defined by its operating characteristics; he does not need to know anything about design details such as vane angles, vane height or impeller diameters. In the case of a software module the concept of 'information hiding' must be supported, which means that what goes on inside the module need not concern a client. Specification of the public part thus defines an 'interface', which is in many respects analogous to the functions of a 'modem' in a communication system. Just as a pair of modems can confer transparency on a communication link, so the public parts of a pair of software modules can confer transparency on their internal functions. Interfaces between software modules, formed by amalgamation of their public parts, therefore facilitate control of the functions of one module by another without reference to any details of the implementation of those functions. Information hiding can thus be perceived as a technique for separating internal functions from interfacing. The less complex the public part of any module, the greater the chance that a design change affecting its internal functions will not affect another module to which it is interfaced. 'Small' interfaces or weak coupling are therefore said to enhance software reusability. Every reusable software module must have both 'private' and 'public' parts. As with a centrifugal pump which an engineer interfaces with a pipework element, client documentation for a reusable software component must contain only that information which a client will require to enable him to interface the module with another in building a system.

A 'transparent module' then must 'encapsulate' a routine for every possible combination of data structure and data type, together with a selector mechanism whereby the module rather than the client programmer would match the data structure and type to
the correct data handling routine. The concept of a 'software package', as supported by such programming languages as Algol and Modula-2, goes some way to solving this problem. By definition, a 'package' may comprise more than one routine, together with declarations of data type, constants and variables, to constitute a complete data structure together with its associated operations. Such a package might comprise, not just the table search routine required, but a whole structure for the table, its creation, deletion and addition operations, as well as a search mechanism. Using a technique known as 'name overloading' the compiler can be made to select the correct routine. This enables client programmers to write 'client code' which will invoke a selected routine, but the technique falls short of transparency since it is the client who must specify the required routine. Thus the concept does not offer a basis for implementation of truly reusable modules. Neither does it exploit commonalities between groups of implementations; it merely lumps them all together in one large package! In any case, such procedurally coded modules are likely to comprise large amounts of program and to execute very slowly, if indeed they are usable at all. Any new client requirement would necessitate a redesign of this single large package which in any case, far from reducing the total amount of code required, would greatly increase it, thus negating one of the main advantages expected of reusability.

That reusable software is not practicable merely on the basis of procedurally coded packages goes a long way to explaining why it is that project specific software for an ESD system is still produced more or less from scratch. The prime requirement for a reusable software module which can be equated with a component of a system in engineering terms can be seen to be a fully transparent interface.

7. OBJECT ORIENTATION AND CONCEPTS OF ENGINEERING COMPONENTS

In order to simulate an engineering component, such as a centrifugal pump, the software module must have features which model the internal operations of the pump. These are the attributes, and the procedures which constitute the methods of the class. All software comprises and operates on data and the programmer perceives the internal object as a list or record which is a representation in code of the external object. Internal functions or methods correspond to operating functions of the engineering component represented by the object; whilst its interface represents its coupling with other components. For instance, computer simulation of a pump involves description, in data, of the 'state' of the pump, and since the simulation must be discrete, it can be appreciated that the pump can represent a class of objects the instances of which in this case are instantaneous values of its state, computed by applying the methods of the class to data representing values of some of its attributes. Thus objects can be but are not always transient. Introducing the concept of state to software modules in this way makes it possible for a data structure to have a real time dimension.

Invoking the methods of an object, or in Smalltalk jargon 'sending a message to the object', may result in generation of another object which could represent a new state of the external object (the pump for instance). A recent paper describing an object oriented data base management system SO2 (1.1), makes distinction between 'active' and 'passive' objects, with the declared intention to support asynchronous communication between objects in a real time system such as a process control system.
The concept of active objects is of objects which can change state by internal operations, whilst that of passive objects is of objects the states of which can only be changed by interaction with other objects. The internal state of an active software module can thus be changed by its internal methods, on a time dependant basis for instance, or by receipt of a message via its interface. If objects are to 'lead a life of their own' as engineering components, communication between them must be asynchronous. SO2 supports the concept of a 'blackboard'. Messages can be left on the blackboard by one object for another to pick up in its own time; a sort of software 'poste restante'.

Although the software still needs hardware to run on, it can be seen that in a very real sense an active software object class can indeed live a life of its own. The concept of software sub-components is therefore realistic, which it is not without object orientation! The concept of an object can be seen to embrace that which is lacking from the concept of an entity as defined in the EAR data model; an interface. What can be 'encapsulated' into a software object can thus parallel the real world in a way that is impossible with conventional top down functional decomposition design methods, supported by a conventional procedural programming language. However, it is essential that interfaces between objects should be 'transparent', by which it is meant that the object sending a message need know nothing about the methods of the object which is to receive the message. This directly addresses the concept of client relationship. A manufacturer of pipework components for an engineering system needs to know something about the attributes of a centrifugal pump which is to be interfaced to it, such as the connection size and the working pressure, but he does not need to know anything about how it functions (methods). At present this is not true of software modules: a client for a program (or functional module) normally has to make decisions relating to how the functions shall be performed! The relationship between two objects is always that of a 'client' and the provider of certain 'services'. The entities in an E-R diagram could be replaced by objects, in which case relationships can be seen to be constrained by the definition of the interfaces between them.

'Functional abstraction' is the cornerstone of software design using structured procedural languages. It supports description of function without concern for implementational detail (programming). Object orientation applies the concept of abstraction to data types rather than to procedures, thus enabling data structures to be used without concern for the details of their implementation. In an object oriented language, a class of objects is perceived as a collection of data and operations (methods) in generic form. Specification of a class of objects comprises the definition of its attributes and methods as features, just as specification of a centrifugal pump includes definition of its design details and performance. So features of a class are generalisations of the data fields (attributes) and procedures (methods) which determine any specific instance, or object. The class does not change when an object is created, but an old object may or may not be discarded. Consider the application of an object oriented language to a dynamic computer graphics display of a process equipment system comprising pipework, valves and pumps for example. Together with an 'icon', the display of a pump would typically show current real time values of parameters such as discharge pressure, which are attributes of the pump. There may be a number of pumps in the engineering system which could be represented by an identical class of objects. (that is, have the same generic attributes such
as inlet and discharge pressures, and identical operations to compute their states) However, the values of these attributes must be different for each pump; A different class definition for each pump which could specify different transducers as the source of specific values for each generic attribute such as mass flow rate. Each instance of any of these class definitions would then represent a new state of one of the pumps. However, given a mechanism which could provide identities of specific mass flow rate transducers, suction pressure transducers etc. at run time, only one class definition would be required. This would obviously provide great economy of code. Such a mechanism necessitates support for the concept of dynamic data abstraction or 'Polymorphism'.

Graphics displays are essentially dynamic illusions, and Kay defined a dynamic mechanism of data abstraction within Smalltalk for this type of application. Polymorphism is construed as the ability of an 'entity' to take more than one of a number of possible dynamic 'types', at run time. This requires that the object oriented language support the concept of 'sharing', which in turn necessitates support of a type of 'data field' which is rarely if ever supported by a procedural language - a reference field. Type declarations define fields in the usual way but a reference field refers to object classes as though a class were a type of data. More than one class may thus contain within its reference field, the same data, which is a reference to a third class. In a reference field, units of data are known as entities: an entity providing the means whereby one class can refer to another. (Thus the meaning of the term entity is completely different from that of the same term as it is used in EAR data modelling.) Object Oriented Entities are declared with a 'static' value but are given a 'dynamic' value at run time as part of the operation of creating an object. This dynamic value is not itself an entity, but is an 'heir' to the entity of its class. Classes, sub-classes and super-classes can all have entities, declared in reference fields.

Applying the concept of Polymorphism to the earlier example of dynamic computer graphics display of a process equipment system, the reference field of the class 'pump' would contain as an entity, a static reference for each measurement transducer connected to it. These would then be replaced by a dynamic values at run time thus determining the specific measurement value to be displayed on the pump icon defined by the object or instance to be created. Only one class definition would then be required however many pumps there were, provided that the generic form of each pump was identical. At run time, as part of the create operation an entity defined statically as 'the identification number of suction pressure transducer' would be given the dynamic value corresponding to a pressure transducer connected to the pump in question. The graphic symbol created would then represent the latest state of that pump.

Another example of polymorphism can be provided by the simulation of operation of a plant which has a number of centrifugal pumps. An entity of static value 'centrifugal pump type A' referring to a class which has, by reason of its design features, a certain operating characteristic, might take as its dynamic type at run time either 'pump No 123', or 'pump No 456', both of which, being of the same design, have the same operating characteristic. Each such pump has quite different operating states however and must be represented in the simulation by a separate object, an instance of this class. At each iteration of the simulation, a new object would be created for each pump on receipt of a message containing a reference to the pump. This reference would result
Software Quality Management

in the operation of the internal methods of the class on a database appropriate to that pump. This data would of course be totally different in the case of each individual pump. The previous object might be discarded, or recorded as historical data, and the latest object created would then represent the current state of the particular pump referred to. Thus each pump would live a life of its own within the simulation, but only one class need be defined. Attributes of this class would be 'statically' defined in generic form and take the 'dynamic' values appropriate to a particular pump at run time.

Abstraction of data by support of polymorphism can thus be seen to be a way to economise in the amount of code a programmer has to generate for any particular application. Because the data remains in generic form (data abstraction) right up to the time when an object is required to perform its function/s (method/s), the same class definition can be used at run time to represent many 'external' objects which have identical generic attributes. Application software does not need to encapsulate the associated coding more than once (in generic form).

'Dynamic binding' (1.2 & 1.3) is recognised as one of the most important properties of object orientation. The concept enables methods which are features of a class, as well as attributes, to take different forms in different sub classes; the appropriate form only being specified at run time as part of the create operation. For example, the class centrifugal pump may be a 'parent' to a number of sub-classes, each having not only different inlet and outlet diameters, impeller diameters, suction pressures etc., but also design differences such as, for instance, shrouded or unshrouded impellers. This means that the procedure to be used to calculate say discharge pressure, though similar, need not be the same for any two pumps. At run time, appropriate procedures for calculating discharge pressure, mass flow rate etc. will be selected dynamically as part of the process of creating an instance of the appropriate sub-class. In the examples given above, not only the dynamic values of the attributes but, given support for dynamic binding, correct operations (methods) could be decided at run time. Not only attributes, but also methods can remain in generic form until run time.

As an example of how this might work, take the case of simulation of the hydraulic behaviour of a process plant comprising several pumps, valves and pipework. An executive procedure organises the periodic computation of an updated set of values representing the 'states' of each component in the system. Pump No. 123 happens to be of one possible 'type': centrifugal_pump_type_A. The 'executive' procedure selects a data base for pump_No.123, a LIST, from memory and passes it to the create procedure for the latter to create a new object, which in this application will represent a new state of pump No.123. Most 'methods' of centrifugal_pump_type_A are inherited from its parent centrifugal_pump but one, 'discharge_pressure' is different:-

a.create (LIST 123: REAL);  
[run 'create' procedure, taking data from the operating data list for pump No 123, to create an object of the class centrifugal_pump_type_A]

c := a;  
[assign dynamic value 'a' to the entity statically declared as 'c']
x := c.discharge_pressure

['method' to calculate
latest value of variable
'discharge pressure'
uses form of procedure
specific to the class:-
centrifugal_pump_type_A]

The declared (static) value of entity c: centrifugal_pump, takes
the (dynamic) value a: centrifugal_pump_type_A, at run time;
[There may or may not be more than one type A pump] then at run
time an object is created, which is the instantaneous 'state' of
the type A pump No.123. There can be no question which of a
number of possible versions of the procedure 'discharge_pressure'
is being called; it is the version for the pump class 'a',
encapsulating the correct procedure (method) to be applied to
data for this type of pump; in this case pump No.123. A class 'a'
pump can be seen to be an 'heir' or descendant of the more
general class of centrifugal pumps 'c', and will therefore
inherit from class 'c' any 'features' common to class 'a' pumps,
but not those which are common only to other classes. One feature
which can never be inherited is the 'create procedure', since
invoking this procedure produces an instance (object) of the
class of which it is always a unique feature. The create
procedure thus creates objects which inherit features from
superior classes with the addition of other features and/or
substitution of specific versions of generalised (generic)
operations (methods) defined in a parent class.

'Assertions' can be included as additional features of a class in
order to constrain or limit operations (methods). Assertions are
statements, included in the code, which express some specified
requirement of the module behaviour which can be checked during
execution: pre/post assertions are checked prior to and after
code has been run. Assertions are currently used mainly to help
the programmer check the correctness of the code he/she is
writing. ('programming by contract') However they constitute an
obvious way to 'model' limit conditions in real physical systems,
and inclusion of appropriate assertions could also enable the
behaviour of the software to be monitored. This latter use could
help to eliminate undiscovered 'bugs' which, as recognised by the
HSE guide, are the principal reason why the behaviour of software
cannot be assured at present. However 'on-line' use of assertions
is still in the development stage: mechanisms to ensure fault
tolerance and/or recovery from failure must be developed before
this technique can be generally adopted.

8. INHERITANCE AND THE ENGINEERING SOFTWARE COMPONENT.

The concept of inheritance is supported by both Object Oriented
systems and knowledge based systems. In its static or
specification form, it is a formalisation of hierarchical
relationships between the most general form, through more
specific, to the most specific form of a generic object class.
Inheritance in object oriented systems can be seen to be a
parallel concept to that of entities and super entities as
defined in the EAR data model. For instance, many features
(attributes and methods) of a centrifugal pump may be common with
those of another. However, all single stage pumps will have more
in common with each other than any will with a multistage pump,
and members of any sub-group of single stage pumps will have even
more in common. Applying the concept of inheritance at the
specification stage, only the differences between a new pump and
an existing one need to be addressed. Translating the external
object class, or pump specification, into the internal object class, (code representing it in a computer) the amount of new code required could be drastically reduced in almost all cases. From the viewpoint of testing and certifying software object classes this clearly reduces the potential for systematic errors arising in a new class defined as a modification of a generic 'parent'.

The concepts of object orientation can be applied to design of a new type of centrifugal pump having such attributes as the physical parameters:-

- inlet diameter;
- outlet diameter;
- impellor diameter;
- impellor inlet vane angle;
- impellor outlet vane angle;

and operating parameters:-

- suction pressure;
- discharge pressure;
- speed of rotation;
- mass flowrate;

(which could equally well represent a simulation of the same pump) A new class would not inherit all of the computational operations (features) by which the set of attributes (impellor diameter, vane angles etc.) are computed from the operating design parameters of a generic parent.

The 'internal object' resulting from running the software of its class of objects with specific values for the operating parameters specified (creating an instance of the class - an object) would represent in data form, the 'external object', the design of a specific pump of the type represented generically by the class. A different set of values for the operating parameters would yield a different pump design, but one which is of the same type. If any of the procedures for computing the attributes impellor diameter, vane angles etc. from the operating parameters suction/discharge pressures, mass flowrate etc. are different, a new class must be defined before an object representing the specific design of a new type of pump can be 'created'.

If the programming language supports inheritance, only the differences between existing code representing the parent class and that required to represent the new class need be specified and codified. The object code representing the parent class would be inherited at compile time, becoming the starting point for modification of class specification. It is not necessary that every feature of the parent class be inherited: some attributes and methods can take a different form from the parent if necessary. This is known as redefinition. Instances of the new class, once created represent the design of a new type of pump; different from its parent but not radically so. The nearest example of an existing pump object class would be chosen as the parent from a group of possible pumps. It in turn might have inherited some of its features from a more generic design. Multiple-inheritance if supported by a programming language, would allow features from more than one 'parent' class to be inherited selectively.

Inheritance can be seen to be fundamental to the engineering concept of reusable software, because it makes possible the creation of a new design (of pump) as a variation of a more general form. Using object oriented design supported by a suitable object oriented programming language it would be
possible to define a sub-class having a shrouded impellor for instance, which therefore inherits a modified form of the method to calculate discharge pressure, from a general class, centrifugal pump. Other methods such as calculation of electrical power consumption, might not differ from the parent class. The program code for the parent class can be inherited by subclasses and it is no longer necessary to reinvent the wheel (or pump) for every new pump design. If these mechanisms could be made to operate dynamically, very considerable economies could be made in the amount of code to be written for say a simulation application. The design of a class of objects to represent a specialised type of pump could be synthesised at run (not compile) time by inheriting features from one or more parent super-classes and by redefinition. If the language supports the concepts of polymorphism and dynamic binding this is indeed possible!

Thus the term inheritance covers such concepts as redefinition (the possibility for a class to override some of its proper ancestor’s features); multiple inheritance (ability of a class to be formed from more than one parent); polymorphism (association of references with instances of different classes at run time); and dynamic binding (selection at run time of a specific variant of an operation). Inheritance can therefore be perceived as having two separate aspects:-

Specification or static inheritance;
& Implementation or dynamic inheritance;

The former relates to the application of the principles of inheritance at the specification stage and therefore does not depend on run time concepts such as polymorphism and dynamic binding. Implementation inheritance however implies that a specification is only established at run time as part of the create operation. This necessitates support of those concepts which address data handling and make it possible to construct software from related classes at run time. It is this aspect of inheritance that is perceived by a programmer or software engineer as a means to reuse and extend code by imitation, refinement or combination. Without implementation inheritance, each pump must be represented by a different module. However, mechanical and software engineers have different views of the reusability of software. Perceiving the software module as a component of a system, the former may be content if such basic components can be reused as specified as components of different systems or sub-systems. Whilst such components or reusable software modules must always be transparent, it does not appear to be essential that inheritance be supported in the implementational sense. Implementation inheritance demands a rigidly defined syntactic basis and must support polymorphism and dynamic binding in order to ensure type consistency, and dynamic redefinition (and even perhaps multiple inheritance) at run time. It would enable the programmer to reuse code, written to implement one class of objects, to synthesise a specialisation of it, by merely defining small modifications to this established module. The total code required for any application could thus be greatly reduced, and most important of all, the amount of new code subjected to retesting and recertification could be minimised.

Multiple inheritance, if it can be supported, enables a programmer to combine existing classes as a technique for meeting new requirements. However, definition of multiple inheritance presents difficulties which are not fully resolved at the present time, as in the example given in ref 1.4., in which two classes Y
1 and Y 2 both inherit features from superclass Z. Such difficulties are currently being researched and it is therefore not possible to be sure that any programming language available will generate software modules which are reusable in a totally transparent sense.

9. SUMMARY AND CONCLUSIONS.

Conventional top down design techniques, supported by structured procedural programming languages, have been shown to be incompatible with the management of bottom up design of large multi-discipline engineering projects. In this respect software engineering is fundamentally out of step with the rest of the engineering profession. Problems caused by continual changes of software specification during the design period currently make testing and certification of systems impracticable if not actually impossible. It cannot be seriously in question that this is the root cause of the lack of reliability and robustness of the software identified in the IEE/BCS report of 1989 (Safety in safety related systems).

Design and implementation techniques for ESD system software should support bottom up design of software systems and employ reusable modules rather than monolithic programs. Reusable software modules could thus provide a basis for the "improved techniques of software... development..." demanded by the 1990 ICSE SafeIT report. The capability to design and construct such reusable software modules would mean that changes to platform design, imposed by other disciplines, could be accommodated by changing a minimal number of such software modules (perhaps only one in many cases) without any effect on other modules. Functional testing, and as a result certification of modules rather than entire systems would be a practicable possibility and a safety lifecycle could be established for each such reusable module rather than for an entire system. In this way reliability and robustness of software could be greatly improved, and project management facilitated.

It is important to make a distinction between software design techniques and programming languages. A number of new concepts are implicit in both, and it is not possible to take full advantage of object orientation without support for the concepts of 'data abstraction', 'polymorphism', 'dynamic binding' and 'implementation inheritance'. However, the principle requirement of such modular software systems is for transparent interfacing between modules and this concept must be supported by any implementational technique adopted. For instance Modula-2, which is often put forward as a suitable language in which to implement object oriented design, can indeed support the construction of generic classes: however these cannot be used without first replacing 'formal generic parameters' with 'actual generic parameters' to create an instance: polymorphism and dynamic binding cannot therefore be supported. This might be found to be acceptable, as it appears that only specification inheritance is essential for ESD applications. However, under Modula-2 it is also necessary for a 'client' program to contain a procedure to carry out this substitution at run time, which makes it impossible to achieve 'transparency'. This deficiency is definitely not acceptable; transparency is essential for ESD systems!

The concept of inheritance offers the possibility to economise greatly in the amount of code which has to be generated for each application. Support of the concept of redefinition and, if
possible, multiple inheritance would greatly facilitate the testing/certification procedure by limiting the changes involved in 'updates' by increasing scope for imitation and extension. Finally, support for implementation inheritance, which entails support for the concepts of polymorphism and dynamic binding could be expected to further reduce the amount of code required for any application and the changes which would be subject to retesting and recertification. It is indeed a paradox that modules written in a standard procedural language correspond more closely to the engineering concept of components in that they, like their 'real' equivalents, are of fixed structure.

The concept of object orientation appears to provide the opportunity to construct reusable software modules which correspond to components of the software system in a systems engineering sense. However there does not appear to exist, at present, any one language which will support all the concepts identified above. Moreover, an ESD system is a real-time-on-line system which requires extensive 'front end' facilities to support interfacing with measurement sensors and transducers. Development of such facilities typically absorbs thousands of man-hours of programming effort. An alternative to a compilable object oriented language could be a suitable software environment, which, by reason of maturity, would improve software quality. Such an environment, which is marketed as a real time expert system development tool, has been identified. Using the rule based facilities, it has been found possible to define protocols by means of which the essential concepts identified, though not implementation inheritance, can be supported. This approach appears to offer considerable advantages over the present methods which are based on top-down design and implementation under procedural languages. It is also realisable using existing technology. However, in the long run a suitable object oriented language is the best answer to the problems of ESD software systems. It could provide maximum economy of code. Security could depend entirely on testing and certifying individual closed software modules as components of more than one system, and not partly on maturity of a non object oriented software environment. Finally the on line use of 'assertions' would help greatly to ensure the absence of systematic coding errors or bugs'.
472 Software Quality Management

References:

1. **SO 2:- A Flexible Object Oriented Data Base Management System.**
   Authors:- A. Attouri & M. Schneider
   Of:- Laboratoire d'Informatique, Universite de Clermont-Ferrand II
   Presented at:- The Second International Conference on Reliability and Robustness of Engineering Software.

2. Object-Oriented Software Construction. (page 227 sec. 10.1.9)
   Author:- Bertrand Meyer.
   Published by:- Prentice Hall.- 1988.

   (page 122 section 9.4)
   Authors:- J.A. Hewitt & R.J. Frank.
   Published by:- Macmillan

4. Inheritance in Object Oriented Programming Languages. (sec. 4)
   Paper from:- Inheritance hierarchies in knowledge representation and programming languages.
   Edited by:- Maurizio Lenzerini, Daniele Nardi & Maria Simi.
   Published by:- Wiley

   Published jointly by the BCS and IEE. 1989.

6. 'SafeIT'
   Published by the Interdepartmental Committee on Software Engineering (ICSE) 1990.