ISSN: 0128-7680
Pertanika J. Sci. & Technol. 3(1):1-18(1995)

Software Design Criteria for Maintainability

Aziz Deraman and P. J. Layzell[1]
Computer Science Department
Faculty of Mathematics and Computer Science
Universiti Kebangsaan Malaysia,
43600 UKM, Bangi, Selangor Darul Ehsan, Malaysia

[1]*Information System Group*
Computation Department
UMIST
P.O. Box 88, Manchester, M60 1QD, England

Received 20 August 1992

ABSTRAK

Satu daripada isu yang sedang diperkatakan dalam bidang kejuruteraan perisian adalah berkait dengan masalah penyenggaraan perisian. Telah menjadi hakikat bahawa masalah yang timbul disebabkan oleh rekabentuk perisian yang tidak baik dan amalan penyenggaraan yang tidak betul Isu rekabentuk perisian yang tidak baik merupakan tujahan kertas ini. Kita menghujahkan bahawa kebanyakan metadologi rekabentuk perisian sekarang tidak dibina berasaskan kriteria untuk menjadikan kerja penyenggaraan di kemudian hari lebih mudah. Oleh yang demikian, dengan adanya satu set kriteria rekabentuk untuk kebolehsenggaraan, perisian yang dihasilkan dipercayai akan lebih mudah disenggara. Dalam kertas ini kita akan mengenal pasti kriteria berkenaan dan diikuti dengan penilaian terhadap beberapa metodologi rekabentuk perisian.

ABSTRACT

One of the current issues in the software engineering community is related to problems of software maintenance. It is a common belief that these problems are caused by bad software design and poor maintenance practices. The first of these is the concern of this paper. We argue that the existing software design methodologies are not properly developed based on criteria for easy software maintenance at later stages. Therefore, with a set of software design criteria for maintainability, software is believed to be more maintainable. In this paper we shall identify those criteria followed by assessment of several software design methodologies.

Keywords: software design, maintainability, methodology, modelling, prototyping, explicitness, software engineering

INTRODUCTION

One of the current issues in the software engineering community is the problem of software maintenance. Even though it has long been recognized (Swanson

1976), the problems are becoming worse (Zvegintzov 1983; Weiner 1984; Harrison 1987; Linehan 1988). It is a common belief that these problems occur because of bad software design and poor maintenance practices. The first is the concern of this paper.

A vast number of methodologies are being used in software development environment. These methodologies are different in various ways such as the emphasis on the approaches taken to solve a certain problem, the use of design languages, supporting tools and so on. To stem an outbreak of software maintenance problems, methodology producers are trying to update their methodologies so that the software produced are more maintainable. A maintainable software product is one that is understandable, testable and easy to modify. Maintainability can be derived from several maintenance metrics such as consistency, modularity, simplicity, conciseness and self-descriptives (Gilb 1977; Boehm 1978; Perlis 1981; Arthur 1985). However, this strategy has yet to be proven successful since the software maintenance processes are still costly.

Having analysed software maintenance metrics (Perlis *et al.* 1981; Athur 1985), we have come to the conclusion that they can be applied to the software development methodology to produce more maintainable software. Therefore, this paper begins with a brief historical perspective of software development methodology. We then propose several design criteria for maintainability followed by assessments of several contemporary methodologies. Finally we conclude with discussion of future trends based on the proposed criteria for maintainability.

HISTORICAL PERSPECTIVE

In the 1970s, research to improve software development methodology was conducted and the so-called 'structured methods' emerged. During this period most of the proposed methods were based on functional decomposition; these were later followed by methods based on data structure and data flow. Functional decomposition, one of the earliest structured methodologies, was based on functional analysis. Among the successful methodologies using this approach were HIPO (Stay 1976) and SADT (Ross 1977; Dickover 1978). However, as software became larger and more complex and software maintenance gained popularity, these methods were no longer reliable. Then came the idea of software development based on data. It was claimed that, data-oriented designs were more stable and good for maintenance purposes. Among popular methodologies using data-oriented approaches are JSP (Jackson 1975) and Warnierr-Orr methodology (Orr 1977). Methodologies using data flow are structured design (Stevens 1974) and similar methods by Meyers (1975), DeMarco (1978) and Yourdon and Constantine (1979).

As software maintenance problems became more apparent, efforts to improve the existing methodologies continued into the early 1980s. During

this period, data-based modelling, especially the E-R approach (Chen 1976) and functional analysis modelling, were used together to get a good system model (Yourdon 1984). Recent methodologies are more comprehensive and integrated (i.e. cover almost all aspects of the software development life-cycle) and are generally accompanied by several development tools. Such methodologies are information engineering (Macdonald 1986), NIAM (Verheijen 1982), SSADM (Downs *et al.* 1988; Longworth 1989), JSD (Jackson 1983) and RUBRIC (Layzell and Loucopaulos 1988).

However, chaos in software maintenance is still far from over. Contemporary methodologies with powerful graphical tools and some with good formal practices such as VDM (Hekmatpour and Ince 1988) do not seem to have overcome the maintenance problems. There are also some suggestions on revised methodologies with emphasis on maintenance, but many of these ideas are based purely on a mixture of the old methods and management practices (see e.g., Brice and Cornell 1983; Connell and Brice 1984; Tinnirello 1984; Longworth 1985). We believe that the emphasis on subjective areas such as management contributes little towards maintenance since it can be easily ignored in a real world environment. On the other hand, a major factor that should be considered now is the actual contents of the design itself and how they are represented.

DESIGN CRITERIA FOR MAINTAINABILITY

There are two types of criteria that can contribute towards producing maintainable software. The first, or primary criteria, will determine the ease of maintenance during a software's life-time, especially adaptive maintenance. The second, or secondary criteria, is the one that will ultimately determine the quality of the software produced, which will help to meet user requirements. These criteria will therefore reduce problems especially for corrective and perfective maintenance.

Among primary maintainability criteria in software development methodology are:

- Real world modelling;
- Independence of specification modelling (which includes the following models: process, entity, event, constraint, task and human-computer interface);
- Explicitness;
- Modularity.

The secondary criteria are:

- Data dictionary;
- Uniformity;
- Prototyping;
- User involvement;

- Documentation;
- Computer-aided tools.

The above criteria are expected to contribute towards software maintenance generally in the following areas:

◊ Reducing effort for software understanding by high-quality design deliverables;
◊ Reducing corrective maintenance effort by meeting all user requirements;
◊ Producing software to accept changes by anticipating future requirements early in the design;
◊ Making maintenance tasks simpler by the adoption of simple and mechanizable techniques during software development.

The following are detailed descriptions of the above criteria:

Real World Modelling

The need for users to anticipate current and future requirements during software development is crucial. To achieve this, a methodology should first provide a way to construct a real world model of an application. With the model, users are expected to understand more formally and thoroughly about their world, and therefore can further anticipate their future requirements. With this understanding, users can perhaps express their requirements more accurately and therefore help reduce maintenance effort. With less effort needed to specify users' requirements, the analyst can spend more of his time working on flexible design so that adaptive maintenance can be done easily as users view their future world.

Independence of Specification Modelling

The specification phase in software development is very important since it helps to bridge the real world (user perception) and the system world (analyst perception). This phase will specify the user's requirements as perceived by the analyst. Since the product of this phase becomes a critical resource during development and maintenance, it is very important to model various specification elements explicitly. As mentioned earlier, there are six elements that should be modelled separately to ensure greater maintainability of software.

i) Process models: This model describes and represents processes involved in the system. The model will show the interrelationship between processes used to convert input into output. To a certain extent, the model will also show the detailed action of each process.
ii) Entity models: Entity modelling is concerned with the description of data behaviour. Every distinct object in the enterprise will be identified

together with all their relationships (each relationship can also be regarded as an entity). These entities and relationships will then be used to construct an entity model(which represents data). To further refine the entity model, attributes for each entity type can be identified and more detailed entity models can be produced.

iii) Event models: Event modelling is concerned with the happenings in an enterprise which will change the state of data in the enterprise. Occurrences of the events eventually trigger a certain process or action in the system .

iv) Constraint models: Constraint modelling is concerned with rules or constraints that are applied to an entity throughout its life-time in a system. These rules therefore determine the behaviour of the entity that is difficult to represent in the entity model.

v) Task models: This model describes the sequence of user interactions with the system. The model will show the logical order of processes from the user's perspective.

vi) Human-computer interface (HCI) models: HCI modelling is concerned with the way users will interact with the system. It provides details about how each task will be performed by the user.

Explicitness

Every decision (or assumption) made during software development processes should be formally stated and explicitly recorded. This explicitness criterion is very important for development and subsequently for maintenance because it will ensure the availability of complete information about any design element at any time. This feature will greatly assist the maintainer to understand and analyse the software during maintenance activities.

Modularity

Software can be partitioned into modules (hence its coding). The structure of the modules may reflect the process of data refinement and functional decomposition. Modules may be related to each other in either a hierarchical or a flat network. Modularity (structured design) is very important in determining software maintainability. If software is modular, it is easier to understand, to track a certain part of the code and its related design deliverables as well as to perform all maintenance tasks. The structure of a module both in data and functional structures is also important for maintenance since most of the new requirements will involve these two structures.

Coupling is one of the criteria for modularity and is concerned with the connections between modules. The simpler the connections, the weaker the coupling, implying high maintainability. During maintenance, modification of a certain module has a low risk of affecting other modules if the coupling is weak. This is very significant for maintenance, since the task of detecting

ripple effects of a modification will eventually create another maintenance problem. Weak couplings also imply that simple information is used for interfacing between modules and therefore will reduce software complexity.

Binding or cohesion is another criterion for modularity and is concerned with the activities within a module. In contrast to the need for low coupling, a high binding within a module is desirable. According to the classification given by Stevens *et al.* 1974, functional binding is required the most where one module implements only one task or one function. This criterion is important for maintenance purposes, since it is easier to identify and to understand a module with a single function rather than a module with several unrelated functions.

Data Dictionary

Data dictionary is a very important aspect of software development and maintenance. It is used to define and describe all data structures used in the system. A good data dictionary will ensure the simplicity of maintenance tasks. During maintenance, references about data definition, usage etc will be made easier with a centralized data store in the data dictionary. All consequences to the data affected by maintenance tasks can be comprehensively located and properly managed.

Prototyping

A methodology that allows prototyping is good for maintenance since all the user requirements can normally be met before the software is delivered. With prototyping, users can be given a sample of the product, either in the form of a screen report or interaction between them and the system. In this way users can verify whether the given reports are what they really wanted or not. Any disagreement can be solved earlier in the design stage. Prototyping can also help users' and designers' understanding of the system as sometimes problems are not well understood until they are implemented. This can lead to another related criterion, i.e. experimentation. Prototyping can perhaps assist a designer to produce several solutions through experimentation and users can choose the best solution to meet their requirements.

Documentation

To complement the requirement for a data dictionary, good documentation practices and tools are needed throughout the software development lifecycle. When all development deliverables are well documented, understanding a software program is made easier. Good documentation will also encourage more users' involvement in the software development. With the supporting documentation users will find easier to understand and to follow the software development process. In fact, good documentation and uniformity enforcement are interrelated.

Computer-aided Tools

This criterion is considered compulsory for modern development methodology. Besides speed and consistency, computer-aided tools will provide a very lively environment when dealing especially with graphical works. The tools also can be very useful aids in calculating and verifying various quality metrics. All these will ensure the production of good quality software. Furthermore, these tools can also be used in a maintenance environment.

User Involvement

More user involvement during software development will produce software that more accurately meets requirements. This will greatly reduce effort at least in corrective maintenance. Therefore, a methodology should explicitly define where and what role users should play during the software development life-cycle.

Uniformity

There is a need for uniformity in all procedures and tools used during software development such as diagrams, structured text and naming conventions. For example, a square will represent an external entity in all stages of software development. Consistency in applying these aids will ensure the integrity and correctness of the design (hence the code). As a result of these practices, maintainers will find that a software product is much easier to understand since all related deliverables are based on a uniform structure.

METHODOLOGY ASSESSMENT

Five contemporary software development methodologies have been chosen for assessment against the proposed design criteria for maintainability. Information engineering (IE) and SSADM are selected to represent the newest structured methods, JSD, to represent the very special structured methods (since it is an extension of the well-known JSP), NIAM, to represent the very near formal method and RUBRIC, to represent the rule-based development paradigm. Details of these methods can be found in references given in the earlier section.

Information Engineering (IE)

- *Real world modelling*
  IE uses the idea of information strategy planning where an information strategy plan is produced to represent a real world model. This includes information architecture and business system architecture. This will enable users such as managers to describe their objectives, requirements and priorities for system developments.

- *Independence of specification modelling*
  For this criterion, IE clearly produces only two separate specification models during business area analysis, i.e. process models and entity models. Process models describe business functions, the process of making up the functions and the process dependencies. Entity models describe entity types, relationships and attributes, with their properties and their usage patterns in the business processes. Both of these models are supported by diagrammatic techniques such as process dependency diagrams, state transition diagrams and action diagrams. IE does not mention anything about constraint models but for event modelling, it is implicitly defined in the process modelling. This can be found in the process dependency diagram which shows how an event can trigger a certain process (Macdonald 1986).

Task and HCI models are addressed by IE in the business system design stage. At this stage one of the tasks is concerned with user-oriented, and behavioural aspects of the system. Among the products of this stage is business system specification where user procedures (task models) are defined for each business process and dialogue and other user interfaces (HCI models) are defined for each computer procedure.

- *Explicitness*
  IE ensures this criterion by using an encyclopedia which keeps track of all the design decisions during software development and automates some aspects of the modelling activities. The use of eleven principal diagram types is also helpful to ensure the explicitness of the design within IE.

- *Modularity*
  IE ensures the modularity of the design by employing various features of structured design techniques such as the use of data flow diagrams, decomposition diagrams and data structure diagrams. These diagrams are used in conjunction with several techniques such as entity and function analysis, interaction analysis, and process logic analysis. However, the method does not explicitly address coupling and cohesion of the modules.

- *Secondary maintainability criteria*
  In general, all the secondary criteria have been satisfied by the IE methodology. The data dictionary system is represented by the use of the encyclopedia. All the information relevant to the development process is stored in the encyclopedia, enabling the methodology to manage and manipulate development products systematically. Prototyping is fully supported by IE as needed. The use of various diagrammatic tools to represent models and designs provides comprehensive support in terms

of documentation. This documentation and the encyclopedia are in fact intersupported.

Among the objectives of IE is the automation of its procedures as much as possible. Therefore IE provides various computer-aided tools to capture and manipulate diagrams, to interact with the encyclopedia and other management related tasks.

In terms of user involvement, IE stresses the maximum involvement of end users in the specification of requirements. In fact, during the design stage, the use of a user-oriented approach shows how concerned the methodology is towards user involvement throughout the development process. This is also supported by the simplicity of the techniques, and tools, which are suitable for user understanding. The matter of uniformity is also a great concern of IE methodology, which provides consistent meaning to symbols employed by the tools. This will encourage thorough understanding by both users and designers.

Structured System Analysis and Design Method (SSADM)

- *Real world modelling*
  There is no special technique to describe a real world model except to use data flow diagrams (DFD) to record the current physical system. The model will show what is done, and also the existing physical aspect of where and how it is done and who does it. The existence of physical resources flows in the DFD perhaps can improve the understanding of a real world model.

- *Independence of specification modelling*
  SSADM defines very clearly the existence of explicit specification modelling elements. For process modelling, SSADM uses data flow diagrams (DFDs) to model the functional aspect of the specification. To represent entity models, SSADM essentially uses the concept of E-R modelling which is known as logical data structure (LDS). LDS modelling provides another way of viewing system requirements or specifications. Another model for system specification is entity life histories (ELH) which is used to model events. ELH model in fact complements the DFD models because DFD can only show events by inference from the data flows and these events are not in order.

SSADM models HCI by the construction of a logical dialogue outline (LDO) for each event or inquiry found in ELH (see Downs *et al.* 1988). These LDOs are then linked together to show a series of ordered tasks that should be performed, and are represented by logical dialogue controls (task

models). However, for constraint modelling there is no such model defined in SSADM. Perhaps this model is narratively and implicitly explained in the problems/requirements lists (PRL) which are used to present all the problems, requirements and system objectives.

- *Explicitness*
  Within SSADM, the existence of several distinct phases in development activities with their appropriate tools will ensure this criterion. Further, the cross-checking feature introduced in the SSADM is also a major factor that will determine the explicitness of the design activity. For example, the data model is inter-cross-checked by top-down logical data structuring technique (LDST) and bottom-up relational data analysis (RDA).

- *Modularity*
  Modularity is supported by three key techniques of data flow diagrams, logical data structures and entity life histories.

- *Secondary maintainability criteria*
  SSADM generally satisfies all the criteria under this category except data dictionary. Data dictionary is not defined within SSADM, therefore the developer has to rely on existing practices within the organization. For prototyping, SSADM provides support in the early stage of software development. The simplest form of prototyping can be done through LDO and LDS, which can give some pictures of human computer interaction. The method further supports simulation of the business to show how the system is proposed and this can be done as part of the process of selecting a business system option (BSO).

In terms of documentation, SSADM is essentially a diagram-oriented method in which most of the language used for communication between users and developers is in the form of diagrams. This will inevitably create various levels of documentation to support subsequent stages in the software development and maintenance. To ensure users' involvement throughout the development period, SSADM provides a very detailed check list of what users should do. Simplicity in using the SSADM techniques is achieved by a set of simple but effective symbols/diagrams used during software development. This will also attract more users' involvement.

Since SSADM is considered a comprehensive but simple method, it can be easily interfaced with computer-aided tools. One such tool, Automate+ provides support in several areas. It provides facilities to integrate and interface between various tools and techniques within the method, and it also support prototyping of the user interface dialogue. Automate+ also provides support for documentation cross-referencing, which is crucial for software maintenance environment.

Jackson System Development (JSD)

- *Real world modelling*
  In principle, JSD is a real-world modelling based methodology. The first task in JSD is to model a real world (about happening/event) without much concern to the aspect of function and data. Real world model is identified as a model process and is represented by process structure diagram (PSD) and is used to describe an entity life history.

- *Independence of specification modelling*
  It is difficult to judge whether such an independence of specification modelling exists in JSD. Modelling activity in JSD can be considered an iterative and continuing process throughout modelling and network stages. JSD firstly identifies its entity model by means of a process structure diagram (PSD) which is similar to ELH model in SSADM. Within this model, each entity is modelled together with all processes that might affect the entity as well as their time ordering. Through several refinements, the PSD will show all events or processes for each entity defined in the real world (entity life history) and at this stage, this entity model can also be considered an event model for that entity.

Process modelling is done during preparation of a system specification diagram (SSD). In SSD, communication (or connection) between processes is established using data streams connection and state vectors connection. At the end, SSD becomes an initial model showing processes with inputs and outputs. During the elaboration phase, SSD is further expanded by adding different types of function processes again using data streams and state vectors connection and finally SSD becomes a very comprehensive process model (see Sutcliffer 1988).

HCI modelling is not directly addressed by JSD. However, the occurrence of this activity is identified when system input is specified from action attributes. By means of input filters, HCI model is constructed within the PSD for each entity, whereby this model is also used for data validation purposes. JSD does not define task and constraint models explicitly. Both of these models can be found implicitly in PSDs and SSDs.

- *Explicitness*
  The only way JSD supports this feature is perhaps by showing a list of elementary operations for every SSD in the design.

- *Modularity*
  Modules correspond to entity and function processes in SSD in which binding rules for a single function within a module can be easily

achieved. Coupling rule is also addressed by JSD during development of process networks. For example, state vector connection does not enforce close coupling between processes since it merely provides a facility to access information about the related processes.

- *Secondary maintainability criteria*
  Not much emphasis has been given to the criteria under this category. Data dictionary is not part of the methodology, and the developer must adapt other methods to include this facility. Prototyping is also not part of the method of implementation in JSD, but some authors like Sucliffe (1988) have suggested a possible prototyping cycle in JSD. With regard to documentation, like most structured methods, JSD also creates large volumes of diagrammatic documentation which are considered very well defined. Unfortunately, this documentation seems to be quite formal, therefore many authors claim that it is difficult to understand, especially by novice users. Also, the method does not explicitly deal with user involvement in the development except during fact finding. The way the method approaches problems and the type of tools provided also tend to discourage users from becoming involved, especially those who have no experience in structured methods and in particular, experience with JSD.

JSD itself does not provide computer-aided tools, but there are some tailor-made tools available to support JSD. For example, JSD specifications can be documented using Speedbuilder and PDF (program development facility) can be used to design and document process structure diagrams. To further automate the development processes, JSP-COBOL can be used to produce COBOL code from PDF specification and JSD-FRAME can support the development life-cycle management.

Nijssens Information Analysis Method (NIAM)

- *Real world modelling*
  In NIAM, the area of concern is referred as an object system where a real world model is described in a very simple way. The model is constructed using a simple hierarchical block diagram which contains all activities performed in the object system.

- *Independence of specification modelling*
  In general, NIAM provides only three explicit specification models (abstraction system), i.e. process, entity and constraint models. Process models are represented by information flow diagrams (IFDs). This model is created for each function and the subsequent subfunction defined using a functional decomposition diagram. The decomposition

continues until each function can describe its transformation and information flows (in IFDs) in detail. Therefore IFD is quite similar to DFD.

Entity is described (for each information flow) during information analysis and NIAM uses a sentence model. This model (i.e. each sentence) is made up of objects (that can be classified into two classes, lexical and non-lexical objects) and predicates (known as rules in NIAM). Therefore, sentence models are quite similar to E-R models. NIAM concept of sentence models may be visualized by graphical notations which are called information structure diagrams (ISDs). With these diagrams, all types of sentence models can be precisely represented.

To complete the description of an abstraction system, NIAM includes rules which prescribe the behaviour of the object system. In NIAM, these rules are called constraints. Most of NIAM constraints have a graphical notation which can be used to construct constraint models. However in practice, constraint models are generally incorporated in ISDs (see Verheijen and Van Bekkum 1982).

- *Explicitness*
  NIAM satisfies this criterion by formally expressing all design decisions using its conceptual grammar language in which checking and validation can be easily done. This is then supported by the activity of modeling all real world events using ISD which can present explicitly all the relevant activities and design decisions during software development.

- *Modularity*
  NIAM ensures modularity by enforcing the use of functional decomposition in constructing its IFDs.

- *Secondary maintainability criteria*
  NIAM is essentially a method to analyse information (not a complete and comprehensive method), therefore most of the criteria under this category cannot be evaluated and discussed. However, NIAM provides very comprehensive documentation. Apart from its diagrammatic representation, its conceptual grammar using formal language called RIDL (referential idea language) is very useful to represent an abstraction system in the formal way.

In the area of computer-aided tools, NIAM analysis is fully supported by ISDIS, which is the meta-information system (also called information dictionary) that can be used to support all activities of creating and storing diagrams, sentences, show the implication and consequences of the specified conceptual grammar, and compile the conceptual grammar to make it suitable for the enforcer of the implementation system.

RUBRIC

- *Real world modelling*
  Within the RUBRIC paradigm, there is no model that explicitly addresses the real world.

- *Independence of specification modelling*
  Among the objectives of the RUBRIC project is to understand more clearly the essential components of the system. To achieve this objective, within the RUBRIC paradigm there exist four explicit specification models. RUBRIC identifies these models in two modes of requirements, static and dynamic.

The static mode pertains to data. Data structures are represented by structural components which describe the basic objects of a system, in terms of entities, relationship between entities and domains. This entity model is similar to the E-R modelling. Another aspect of the data is the existence of some constraints imposed on a particular entity. Some of these static constraints cannot be expressed using E-R notation, therefore constraint models are developed explicitly to complete the modelling of the data.

Process model (dynamic aspect) is represented by behavioural unit models which are used to describe discrete units of behaviour exhibited by entities within a system. In this aspect, there also exists another model, an event model which is known as dynamic rules in RUBRIC. Dynamic rules describe the events that trigger the execution of behavioural units and the precondition that must be specified prior to execution.

- *Explicitness*
  The main idea behind the RUBRIC paradigm is to explicitly separate the business policy from other aspect of the design. Therefore, we believe that explicitness is one of the main concerns of the RUBRIC producer.

- *Modularity*
  Modelling of structural components and entity behaviour units facilitate the organization of modules during design and implementation of the system.

- *Secondary maintainability criteria*
  The RUBRIC project is still at the development stage. Therefore, many of the requirements under this category are still at the research stages. However, in the area of computer-aided tools, RUBRIC is well ahead. Many of the implementation aspects are planned to be supported by automated tools such as application controller, MMI management tools and rule processor.

CONCLUSION

In this paper we have presented our view on software development methodology in relation to maintenance problems. We have shown that contents and presentation of the design especially modelling aspects, are the most important factors that can facilitate maintenance tasks. Therefore, we have established several design criteria for maintainability.

We have also made assessments of several methodologies against the proposed criteria for maintainability. The following are summaries of those assessments (see Table 1):

TABLE 1
Software design criteria vs
software development methodology

CRITERIA	METHODOLOGY				
	E	NIAM	SSADM	JSD	RUBRIC
Real World Modelling	H	M	M	H	L
Independence of Specification Modelling	M	M	H	L	M
Explicitness	H	H	H	L	H
Modularity	M	M	H	H	M
Data Dictionary	H	L	L	L	L
Uniformity	H	L	H	H	L
Prototyping	H	L	M	L	L
User Involvement	H	L	H	L	L
Documentation	H	H	H	M	L
Computer-aided Tools	H	M	M	L	H

Legend: Score for the Software Design Criteria
H: High M: Moderate L: Low

- *Real World Modelling*
  In this area, JSD places a great deal of emphasis on real world modeling but the technique is not easy to adapt. NIAM and SSADM use common techniques of hierarchical block structure and DFD to model their real worlds. Therefore, we believe the most promising concept of real world modelling is provided within IE. In spite of common model of real world, IE also includes aspects of management features with the model which can further provide elements of strategic planning for future requirements.

- *Independence of Specification Modelling*
  All methodologies have explicitly defined the two basic models, i.e. process and entity models. Event model has been clearly modelled only in SSADM and RUBRIC. IE incorporates its event model in the process model while JSD derives this model from the entity model. Constraint model has been addressed only by NIAM and RUBRIC. NIAM provides a very comprehensive and detailed classification of this constraint model.

Task and HCI are the two models which have been explicitly addressed only by SSADM, which defines these two models early in the specification stage together with other specification models. These models can also be found in IE and JSD, but they are either indirectly addressed or are produced late in a detailed design specification. In our view, SSADM is the only methodology that fully satisfies this criterion.

- *Explicitness*
  We believe that this criterion can be practically achieved if a methodology can provide detailed guidelines together with appropriate tools throughout the process of software development. Therefore, methodologies such as IE, SSADM and NIAM fit best within this criterion. On the other hand, RUBRIC has the potential to fit this criterion, but unfortunately JSD only provides specification modelling tools to exercise this criterion.

- *Modularity*
  All methodologies assessed are classified as 'structured methods'. Therefore, modularity of the design is also one of their primary concerns.

- *Secondary Maintainability Criteria*
  Within these criteria, we believe that IE is the most promising method. It not only has clear and concise guidelines to perform development tasks, it also provides adequate tools to ensure the simplicity and comprehensiveness of the method to be used. SSADM is also good, but without a properly defined data dictionary, it is difficult to manage, especially design deliverables, for easy referencing and documentationing purposes. Although JSD claims to be a complete methodology, the lack of most of these criteria makes it difficult for use on its own.

From the above discussion, it can be seen that there is a trend for recent methodology producers to produce software development methodology which satisfies our criteria for maintainability (for examples, SSADM and IE). We believe methodologies such as SSADM and IE will be the main preference for methodology users, because of their clear guidance, detailed

modelling activities, simplicity of use (this reduces training requirements) and comprehensive supporting tools (for example, UK Government has adopted SSADM as a standard method for all IT projects).

Finally, we believe that the proposed criteria for maintainability will support the creation of software development methodology which can produce maintainable software. The criteria can also be used to evaluate existing methodology in order to justify any necessity for change especially to cope with more sophisticated software requirements.

REFERENCES

ARTHUR, L. J. 1985. *Measuring Programmer Productivity and Software Quality.* New York: Wiley.

BOEHM, B. W., J. R. BROWN, H. KASPAR, M. LIPOW, G. J. MACLEOD and M. J. MERIT. 1978. *Characteristics of Software Quality.* Amsterdam: North Holland.

BRICE, L. and J. CONNELL. 1983. A methodology for minimizing maintenance costs. *AFIPS 1983 National Computer Proceedings.* Arlington, Virginia: AFIPS Press. p. 113-122.

CHEN, P.P.S. 1976. The entity relationship model - towards a unified view of data, *ACM Transactions on Database Systems.* **1(1):** 9-36.

CONNELL, J. and L. BRICE. 1984. Prolonging the life of software *AFIPS 1984 National Computer Procedings,* Arlington, Virginia: AFIPS Press. pp 243-249.

DEMARCO, T. 1978. *Structured Analysis and System Specification.* New York: Yourdon Press.

DICKOVER, M. E., C. L. MCGOWAN and D. T. ROSS. 1978. Software design using SADT, Infotech State of the Art Report, Structured Analysis and Design, Vol II. pp 101-114. Maidenhead, England: Infortech International.

DOWNS, E. *et al.* 1978. *Structured Systems Analysis and Design Method: Application and Context.* Hemel Hampstead: Prentice Hall.

GILB, T. 1977. *Software Metrics.* Cambridge: Winthrop.

HARRISON. R. 1987. Maintenance: giants sleeps undisturbed in federal data centers, *Computerworld* **March 9:** 81-86.

HEKMATPOUR, S and D. INCE. 1988. *Softwareprototyping: Format Methods and VDM.* Addison-Wesley.

JACKSON, M.A. 1975. *Principles of Program Design.* Academic Press.

JACKSON, M. 1983. *Systems Development.* Prentice-Hall International.

LAYZELL, P. J. and P. LOUCOPOULOS. 1988. A rule-based approach to the construction and evaluation of business information systems. In *Proc. of Conference on Software Maintenance-1988.* New York: Computer Soc. Press, IEEE.

LINEHAN T.F. 1988. Application software configuration management and testing in a pharmaceutical laboratory automatic environment. In *Proc. of Conference on Software Maintenance-1988,* pp 178-182. New York: Computer Soc. Press, IEEE.

LONGWORTH, G. 1985. *Designing System For Change.* Manchester: NCC .

LONGWORTH, G. 1989. *Getting the System You Want: A User's Guide to SSADM.* Manchester: NCC.

MACDONALD, I.G. 1986. Information engineering: an Improved, automatable methodology for the design of data sharing system. In: *Information System Design Methodologies: Improving the Practices* ed. T. W. Olle, H. G. Sol and A. A. Verrijin-Stuart. p. 173-224. Amsterdam: North-Holland.

MEYERS, G. J. 1975. *Reliable Software Through Composite Design.* Petrocelli Charter.

ORR, K. T. 1977. *Structured System Development.* New York: Yourdon Press.

PERLIS, A., F. SAYEARD and M. SHAW (eds). 1981. *Software Metrics: an Analysis and Evaluation. Boston,* MA: MIT Press.

ROSS, D. T. 1977. Structured analysis (SA): a language for communicating ideas, *IEEE Transaction on Software Engineering* **3:** 16-34.

STAY, J. F. 1976. HIPO and integrated program design. *IBM System Journal* **15:** 2

STEVENS, W., G. MEYER and L. CONSTANTINE. 1974. Structured Design. *IBM System Journal* **13(3):** 115-139.

SUTCLIFFE, A. 1988. *Jackson System Development.* New York: Prentice-Hall.

SWANSON, E. B. 1976. The dimension of maintenance. In *Proc of 2nd International Conf. on Software Engineering, San Francisco, Oct 1976.* p. 492-497.

TINNIRELLO, P.C. 1984. Software maintenance in fourth-generation language environments. In *AFIPS 1984 National Computer Proceedings. pp 251-257.* Arlington, Virginia: *AFIPS Press.*

VERHEIJEN, G. M. A. and J. VAN BEKKUM. 1982. "NIAM: an information analysis method. In: *Information System Design Methodologies: A Comparative Review* ed. T. W. Olle, H. G. Sol and A. A. Verrijin-Stuart. Amsterdam: North-Holland.

WEINER, R. and R. SINCOREC. 1984. *Software Engineering with Modula-2 and Ada.* New York: Wiley.

YOURDON, E. 1984. The structured paradigm - a perspective. In *Structured Method: State of The Art Report* 12:1. p. 141-151. Pergamon Infotech Ltd.

YOURDON, E. and L. CONSTANTINE. 1979. *Structured Design.* Englewood Cliffs, N. J: Prentice-Hall,

ZVEGINTZOV, N. 1983. Nanotrends. *Datamation* **August** 106-116.