METHOD POINTS: TOWARDS A METRIC FOR METHOD COMPLEXITY

Graham McLeod,
University of Cape Town, Private Bag,
Rondebosch, 7700 South Africa
mcleod@iafrica.com

ABSTRACT

A metric for method complexity is proposed as an aid to choosing between competing methods, as well as in validating the effects of method integration or the products of method engineering work. It is based upon a generic method representation model previously developed by the author and adaptation of concepts used in the popular Function Point metric for system size. The proposed technique is illustrated by comparing two popular I.E. deliverables with counterparts in the object oriented Unified Modeling Language (UML). The paper recommends ways to improve the practical adoption of new methods.

KEY WORDS

Methods, Methods Engineering, Metrics, UML

INTRODUCTION

Methods engineering, as used in this paper, is the process whereby methodologists develop and enhance methods related to information systems in a disciplined way. As defined by James Odell: "Method engineering is the coordinated and systematic approach to establishing work methods" (Brinkkemper, Lyytinen, Welke, 1996). Methods engineers have several objectives when performing this activity. These may include:

- Providing a structured, repeatable process for practitioners to follow in producing a desirable result
- Integration of techniques to produce a comprehensive approach to address a broader problem
- Updating existing methods to include more sophisticated techniques or deal with new requirements
- Specification of capable model notation and representation forms which will be accessible to practitioners and enhance communications between users

Research by the author and others (Bosman R, McLeod G, Tanfield J, 1992; Murphey, Harvey, Mattison, 1990) has shown that success in adoption and use of methods is, in part, determined by the degree of fit of the methods to the problem and the difficulty of becoming proficient in the use of such methods. The latter is directly influenced by the complexity of the method. As yet, there is no published technique to measure the complexity of a method so that quantitative comparisons can be made. This paper is a first attempt at establishing such a metric.

MODELING METHODS

The author previously developed a generic method model which is capable of holding the knowledge describing a method (McLeod, 1992, 1993, 1995). The method description includes definitions of the tasks performed, the deliverables created and the resources required to carry out the tasks. These types of method components each occupy a *facet* within the method model.

McLeod Method Model

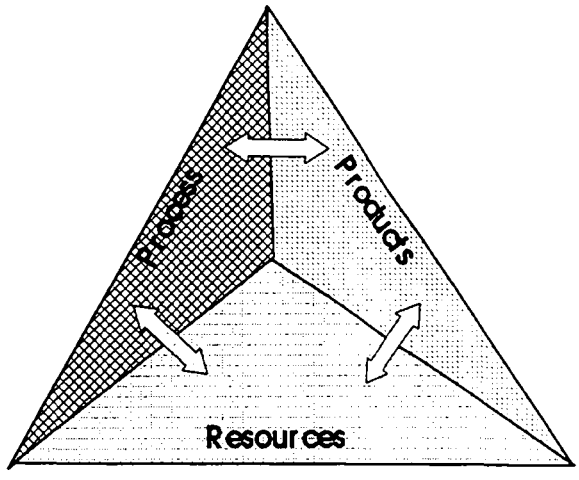

Figure 1 - Method Model Facets

Within each facet, there is a hierarchy of nodes, representing the collation of lower level nodes or the decomposition of higher level nodes. The task (process) facet can thus hold a work breakdown structure, while the product facet may hold a configuration of deliverables and how these are collated into larger products. Components within a facet may be linked by dependency relationships. These form a network similar to a critical path network. Components can also be linked to others across the facets, for example:

- A *product* may be linked to the *task(s)* which create it or use it as input
- A *task* may be linked to the types of *resources* necessary to complete it
- A *resource* could be linked to the *products* it may hold (e.g. a CASE repository which would hold a variety of deliverables identified in the product facet)

There is no restriction on the level or number of these links and they may connect any type of component to any other. The components in each facet are further described by a set of attributes and behaviours, sufficient to fully describe the product, task or resource. Attributes represent data items whose values will describe an instance of that type of component.

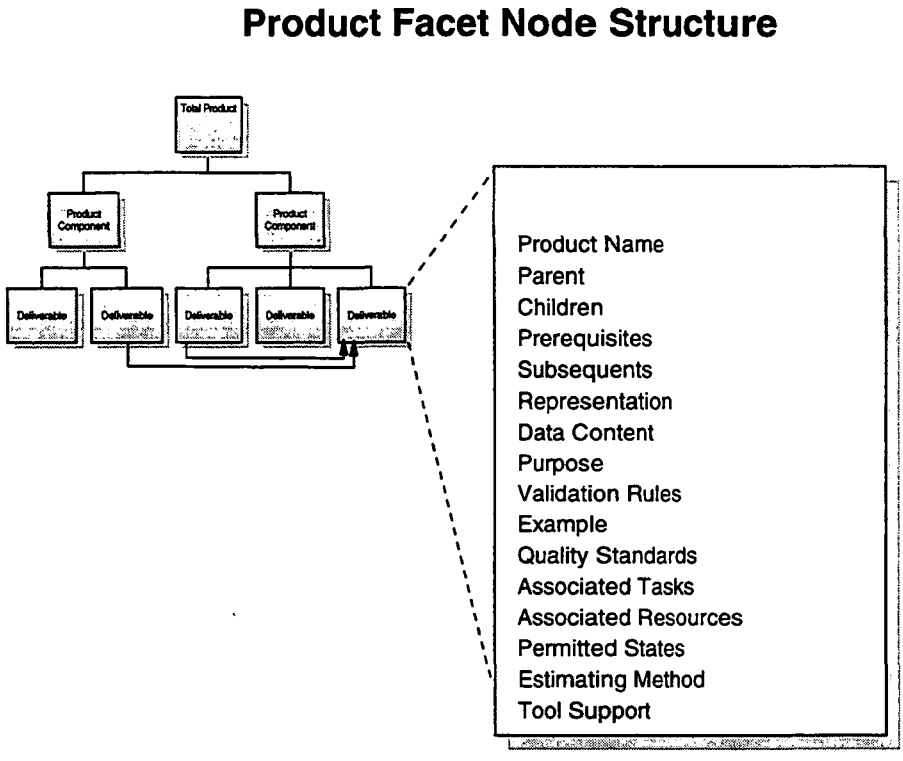

Figure 2 - Product Facet Structure

The model has proven capable of describing a wide variety of methods in a consistent way, and is thus seen as a suitable basis for representation of methods with a view to the application of metrics.

FUNCTION POINTS

The function point metric was first described by Albrecht (1979) at IBM Canada to enable technology-independent estimating of the size (functional weight) of a software product. This was useful in the estimation of software project effort, and in determining relative productivity of development groups. Function point (fp) counting methods were standardised by the International Function Point Users Group (IFPUG), which updates and improves the technique on an ongoing basis (IFPUG,1994). Many organisations have adopted the fp metric, and it has proved to be a useful and fairly consistent (though not perfect) technique.

A fp count for a software product is determined by counting the number of:

- *Inputs* (that add data to the system)
- *Outputs* (that provide information to end users)
- *Queries* (online requests for information)
- *Interfaces* (exchange of data with other systems)
- *Main files* (persistent data maintained by the system)

that the system will provide/maintain, categorising each of these as simple, average or complex, and multiplying the number of deliverables by a function point credit for the type of deliverable and its complexity (e.g. a simple input may be worth 3 fp while a complex input may be worth 5 fp). These scores are totalled to produce a raw fp count.
A second process takes into account adjustment factors which can influence the relative difficulty (hence effort) of delivering the particular project. These include factors such as performance, number of target sites, data distribution, online processing, reusability and others. The influence factors are scored on a scale of 0 to 5. They are totalled and used to calculate a degree of influence, which, in turn, is used to adjust the raw fp count. The adjusted fp count can be used in conjunction with previously measured productivity figures (in fp per person month, or person days per fp) to calculate the likely effort required to create a proposed software product. For completed products, an accurate count can be produced. With data about actual effort expended on development, this can be used to determine the relative productivity of the development process.

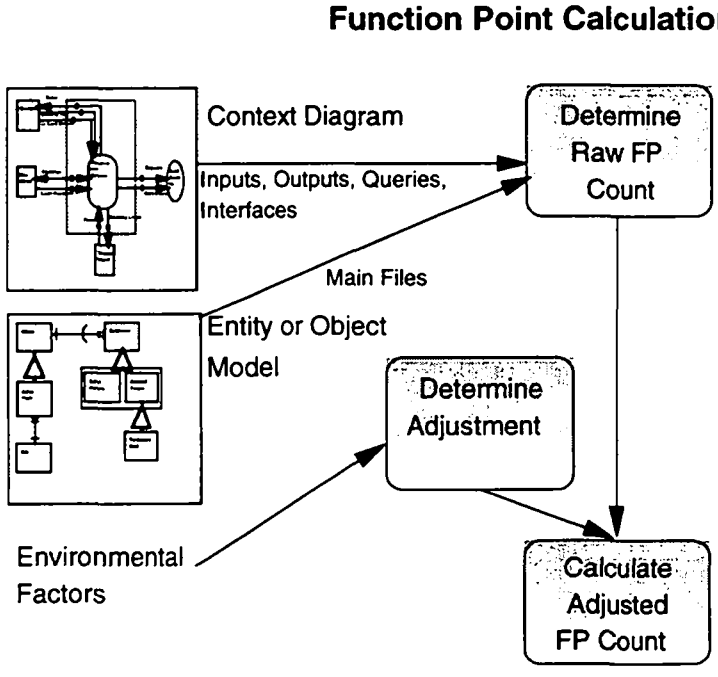

Figure 3 - Calculating Function Points

METHOD POINTS

A method or method fragment can be regarded as behaving in much the same way as a software system:

- It interacts with its environment
- It has users (practitioners) who provide it with information and receive outputs (deliverables) from the processes it performs
- It manages information to support the various processes and outputs

- It interfaces with other methods (or method fragments) which may precede or follow it, or operate in parallel to it

Formal definition of the method in a standard way allows us to collect information for measurement in a similar way to which building a context diagram and entity or object model for a software system allows us to collect information for input to the fp technique. Thus, a model can be built of a method which allows use of a similar process to determine its complexity to that used in function points for a system described in a model.
We recommend that counting should proceed in the following way:

1. *Express the method* to be counted in the method model in terms of Tasks, Resources and Deliverables
2. *Determine the counts for each type of deliverable* specified in the method. We distinguish three types of deliverables:
   - *Graphical Deliverables* such as diagrams and models. Examples would be entity relationship diagrams, data flow diagrams, class hierarchy diagrams and project management network diagrams
   - *Tabular Deliverables* which can be expressed as columns and rows or as records in a relational table. Examples would include the definition of the attributes of a data group, which may have columns for name, type, length and valid ranges; and a system consistency matrix
   - *Textual Deliverables* which include long descriptions and narratives as well as more structured reports and hypertext documents

   We will discuss the counting of each type of deliverable in the following sections.
3. *Determine and add the count for task complexity.* This will normally not come into play, unless the tasks are more complex than would be apparent from the deliverables produced.

Note that for function point counting, an actual deliverable (instance of a description) is used. This yields a *size*. For method points, we consider the structure of a generically described deliverable (class). Since the latter represents a meta deliverable, its variety of options effectively translates not to size, but *complexity*.
As far as possible, a *principle of equivalence* should apply, whereby an equivalent deliverable expressed in a different notation should generate the same count, provided that its semantic content is the same. Likewise, the same deliverable expressed as a graphic model or as text, should yield similar counts.

COUNTING GRAPHICAL DELIVERABLE TYPES

We identify several components of a graphical model which are relevant to our purpose:

- *Symbol types.* These are unique shapes that represent something in the domain being modelled. Examples include entity boxes in an entity relationship model, process boxes in a data flow diagram, and class symbols in an object model.
- *Link types.* These are unique types of connection between symbols. Examples would include a data flow arrow on a data flow diagram, a relationship line on an entity relationship diagram, and inheritance relationships in an object model.
- *Embelishments.* These are any unique type of modifier which can be added to a symbol, a link or the model canvas. Examples include: text label of a data flow, cardinality indicator on an entity to entity relationship, and indication of a key field identity on a data group symbol. Further examples include: a duplicate marker added to a data store on a data flow diagram, a boundary around symbols indicating geographical location or mutual exclusion, and a business rule related to a symbol on an event model.
- *Decomposition.* It is common that a symbol on one model can be expanded into another model at a greater level of detail. An example would be a process box (representing a complete system) within a context diagram which may be decomposed to a data flow diagram expressed as a separate model.

To determine the count for a graphical deliverable, proceed as follows:

- For each unique symbol type, count 1
- For each unique connector type, count .5
- For each unique type of embellishment count .5
- For each type of symbol which can be expanded into another model (which is cross referenced from this one) count .5. If this diagram (model) can contain a reference to a parent model, count .5.

Counting Tabular Deliverables

Tabular deliverables are those which have a structure including labels and other information which can be expressed in tabular form (e.g as a spreadsheet, or as records in a relational table). Typical examples would include: the definition of the attributes of a relation in a relational data model (attribute name, type, length), and the system consistency matrix(Create, Read, Update, Delete diagram) used in IE to check that data groups have corresponding processes and vice versa.
Tabular deliverables may also have Embelishments. An example would be underlining attributes which comprise the key when defining a relational table. Like graphic models, one tabular model can be expanded into a greater level of detail in another. For example, the value in a cell of one spreadsheet can be the result of a variety of complex factors which are expressed in another spreadsheet.

To determine the count for a tabular deliverable, proceed as follows:

- For each column, count .25
- For each unique type of embellishment, count .5
- For each type of expansion that this table can contain, count .5
- If this deliverable can hold a reference to a parent deliverable, count .5

Counting Textual Deliverables

Textual deliverables include long descriptions, narratives and various structured and semi-structured textual documents. Examples include a feasibility report, a process narrative for a data flow process symbol, and an elaboration of the legal requirements for a process on a process chart. Text deliverables can include hyperdocuments which contain links to other sections of the same document, or to other documents, such as would be found in HTML documents designed for perusal on the World Wide Web or an intranet. Normal textual (not hypertext) documents can contain references to other documents and deliverables.
To determine the count for a textual deliverable, proceed as follows:

- For each identified section which the deliverable should include, count .25
- For each page (A4) estimated to be included in an average deliverable of this type, count .25.
  This is based upon a normal size and font (the reference being Times Roman at 12 point).
- If the document can contain hyperlinks, or references to other documents or deliverables, count .5
- If the document contains an "outline" or "indented list" structure indicative of a hierarchical arrangement, count .5

Compensating for Task Complexity

In most cases, the effort involved in tasks will be adequately incorporated in the count by the inclusion of the resultant deliverables. However, some methods may include tasks of considerable complexity, which result in relatively simple deliverables as measured by the above processes. In these cases, we will need to compensate for the effect of such tasks. This is analogous to using *feature points*, as opposed to function points for systems where there is considerable algorithmic complexity, but relatively little data input, output or storage (Software Composition Technologies, Inc., 1997).
There is a danger that estimating task complexity could be very subjective. We thus stipulate that tasks should be expressed in the method model according to the following criteria:

- Tasks should be completely described, so that no external references are required to complete them
- Each task description should not exceed one A4 page of text. If more is required, the task should be further decomposed.
- Each task as described should be able to be performed by one person expending 5 days (or less) of effort. Where more effort would be required, the task should be decomposed further

Once the tasks required are expressed in the form described above, we can add a count for task complexity as follows:

- Count .5 for each task in the hierarchy
- Subtract 1 for each deliverable produced

- If the result is greater than 0, add it to the method point score; if not, discard it (this is to bring the task complexity factor into play only after a certain default level of effort associated with deliverables is exceeded)

Sample Usage - Information Engineering and UML deliverables

To demonstrate the use of the technique, we examine and contrast two components of the information engineering approach (Martin, McClure, 1988) with two components of the newly published Unified Modeling Language (UML) from the collaborators (Booch, Rumbaugh, Jacobsen) at Rational (Rational, 1997a-d). UML has been submitted to the Object Management Group (OMG) for consideration as a possible standard object oriented method. It is thus useful to obtain a benchmark complexity measure for it (and its components) which can be used in comparison with other methods (and possibly other submissions for OMG consideration).
We have chosen two key deliverables of UML for analysis, viz:

- *The Static Structure Diagram*, which can represent many static aspects of the system being modelled including: classes (types), packages, relationships and instances. These will be contrasted with Entity Relationship Models (ERM).
- *Use Case Diagrams*, which are used to capture the interaction of users with facilities provided by the system. These will be contrasted with the I.E. context diagram.

The above method components were chosen because they are frequently used, represent static and dynamic perspectives, and perform similar functions within the respective approaches.

Analysis of the Static Structure Diagram

UML allows for some variation in presentation of the static structure diagram. We have selected for analysis the option which shows only the names of classes included, not their expanded notation. The expanded notation could be treated separately as a tabular deliverable type. The static structure diagram can include the following unique symbol types:

- *Class*: A descriptor for a set of objects with similar structure, behaviour and relationships
- *Type*: A descriptor for objects with abstract state, concrete external operation specifications, and no operational implementations.
- *Template*: A parameterised class which describes common characteristics for a family of classes. It must be bound to a set of parameters to form a class.
- *Ternary* (and higher order) *Association*: showing a relationship between more than two classes via a diamond symbol.

These are counted at 1 point each, for a total of 4.

The following unique link types are permitted:

- *Interface*: indicates externally visible behaviour of a class, component or other entity (e.g. Package)
- *Imports*: indicates the dependency of one package upon a class definition in another
- *Binary Association*: an association between two classes
- *Generalisation*: an inheritance relationship
- *Dependency*: a dependence of one model element upon another
- *Refinement*: indicating the expansion of an element to a greater level of detail

These are counted at .5 each, for a total of 3.

The following embellishments are identified:

- *Bound Element*
- *Utility modifier*: an indicator added to a class to indicate that the attributes and operations of the class are globally visible.
- *Metaclass modifier*: an indicator added to a class to indicate that the instances of the class are classes.
- *Pathname modifier*: used to show that a class included in the model actually resides in another package.
- On associations:

- association *name*
- association *class symbol*
- association *role* (indicates multiplicitly, ordering, qualifier, navigability, aggregation, rolename)
- *OR* association indicator
- dashed line, indicating *optional* association
- *composition*

- *Details*: an ellipsis indicating that not all items are shown on the current diagram
- *Constraints*: overlapping, disjoint, complete or incomplete modifiers on relationships between subclasses
- *Derived element* indicator: showing that an element is derived and need not be included in the implementation
- *Navigation expression*: indicating the navigation path in the class model

These are counted at .5 each for a total of 7.

All of the symbols can be decomposed, and the model itself can be a decomposition of a higher level model, so we add 4*.5 + .5 = 2.5

Our basic method point score based on deliverables only is thus:
4 + 3 + 7 + 2.5 = 16.5

Since UML currently does not specify a standard process for the creation of the deliverables, we cannot perform the task complexity augmentation. It will be instructive to perform this once variants of the UML are popularised and endowed with process elements.

ANALYSIS OF THE ENTITY RELATIONSHIP MODEL

Based on the standard Martin/McClure (1988) notation without sub and super-types, we find that there is just one unique symbol type: the *Entity Box*. Count 1
There is just one unique type of link, a *Relationship Line*. Count .5
Embellishments are present as follows:

- *Labels* on the relationships
- *Cardinality Indications* on the ends of relationships as follows:
  - *Minimum Number* of associated instances (a O or I symbol)
  - *Maximum Number* of associated instances (a I or crow's foot symbol)
  - *A numerated cardinality* where numbers are used instead of the symbols

At .5 each, we count 2.

An entity symbol can be decomposed into another diagram, which could be referenced to the parent, so we count 1.

Our method point score for the ER model is thus:
1+.5+2+1=4.5

ANALYSIS OF THE USE CASE DIAGRAM

Use cases (Rational, 1997) show the relationship among actors and use cases within a system. They are frequently used in early analysis to identify user-system interactions and the transactions and functions which must be present in the system under consideration.
Unique symbols include:

- *Actors*: represent persons or organisational units in the environment who will interact with the system
- *Use Case*: an ellipse containing the name of the use case

We count 2.

Unique link types include:

- *Communication*: links an actor to a use case
- *Extension*: indicating that one use case may include behaviour of another
- *Uses*: indicates that one use case makes use of behaviour in another
- *Refinement*: indicating the expansion of an element to a greater level of detail

count these at .5 each = 2.

Embellishments:

- A *boundary* is drawn to indicate the bounds of the system. It contains the use cases, and does not include the actors.

counts .5.

Hyperlinks/references:

- A use case can be decomposed into another model (more detailed use case, state machine, collaboration)

We count this at 1 (.5 for reference to a detailed model, and .5 for reference to a parent model)

Our total for the use case diagram is thus:
2 + 2 + .5 + 1 = 5.5

ANALYSIS OF THE I.E. CONTEXT DIAGRAM

A context diagram contains the following unique symbols:

- The *actors* in the environment which provide inputs to the system, or make use of outputs
- A *process box* in the centre, representing the system itself

At 1 each, count 2.

Connectors are limited to the arrows indicating communication of actors with the system. Count .5

Embellishments are confined to labels on the *arrows* and the *boundary indication.* Count .5 each, total 1.

The context diagram can be (and usually is) decomposed into further models. It does not have a parent diagram, since it is the highest level model. Count .5

Total for the context diagram is thus:
2+.5+1+.5 = 4

FINDINGS

The UML SSD rates as considerably more complex than the IE ERD. This bears out the experience of practitioners interviewed with respect to the difficulty of teaching, learning and applying each technique. There is much less difference between the UML Use Case and the IE Context Diagram, although the UML technique is rated slightly more complex. Again, this is borne out by interviews with practitioners who found relatively little difficulty in moving to the Use Case approach. A large number of surveys and case studies have found significant resistance to the adoption of object oriented techniques (Graham, 1994). These techniques are often richer and more comprehensive than the preceding IE techniques, however, ways will have to be found to teach and apply them in such a way that their increased complexity does not become a barrier to their adoption and successful use.
One approach is the use of incremental model completion, where only certain aspects of the full model are completed during a phase. In this way, the comprehensiveness and cohesiveness of the full model are not compromised, while reducing the difficulty encountered by project staff in dealing with too many aspects concurrently (Inspired, 1997). This approach is catered for in the notion of states in the method model.

FURTHER WORK

A complexity metric alone will be of little use. It is like knowing the fuel consumption of a vehicle without knowing its load carrying capacity or speed of travel. We thus propose that a *comprehensiveness* metric for methods is also required. This would measure the spread of capabilities of the method. The efficiency of competing methods could then be compared by determining the ratio between comprehensiveness and complexity. A comprehensiveness metric should embody:

- *Lifecycle coverage* (from and to where in the overall process does the method provide guidance)
- *Dimensions* (data, process, timing, quality, geographic distribution, technology, human interface) addressed by the method
- *Integration* (degree to which method is seamlessly integrated)

We intend in future work to suggest such a metric.

The practicality of a method in an industrial setting is also influenced by availability of the calibre and quantum of resources required for its execution. It may be useful to amend the calculated method point based upon the resource demands of the method. This would allow favouring of methods with lower requirements in terms of practitioner skill or number of discrete resource types needed.

CONCLUSION

We have proposed a metric for method complexity and applied it to commonly used deliverables from two major schools of systems development. It appears to provide a useful measure of relative complexity which is consistent with the experience of practitioners interviewed. Further experimental work will be necessary to accurately determine the weight of the factors used for each type of device in the method representation. We would encourage other researchers to undertake studies to facilitate this. The basic framework provides for incorporation of deliverable and process complexity. We have suggested that a resource intensity adjustment might also be introduced. The method point approach can provide methods engineers with a useful evaluation tool.

We encourage the development of a comprehensiveness measure which will yield a score in the same range as the complexity measure for the current "benchmark" method set, viz IE. A method efficiency score of 1, (calculated by dividing the comprehensiveness measure by the complexity measure) would equate to the current IE practice. This factor could then be calculated for new approaches. Scores higher than 1 would indicate progress towards more efficient methods, while those less than 1 would indicate a regress. The author is keen to collaborate with other researchers in the evolution of these techniques.

REFERENCES

Albrecht, A.J., 1979. **Measuring Application Development Productivity**, Proceedings IBM Share/Guide symposium, GUIDE International Corp., Chicago.

Bosman R, McLeod G, Tanfield J, 1992. **The influence of project-method fit on the success of system development methodology usage**. UCT Empirical Project Report, Dept of Information Systems, University of Cape Town, 1992.

Brinkkemper S, Lyytinen K & Welk J (Editors), 1996, **Method Engineering: Principles of method construction and tool support**, Chapman & Hall, 1996.

Graham I, 1994. **Migrating to Object Technology**, Addison-Wesley, Wokingham, U.K. 1994.

IFPUG, 1994. **Function Point Counting Practices Manual**, Release 4.0, International Function Point Users Group.

Inspired, 1997. **Advanced Systems Engineering with Objects**, Inspired, P O Box 384, Howard Place, 7450, South Africa, 1996/7.

Martin J, McClure C, 1988. **Structured Techniques, The Basis for CASE**, Prentice Hall, Englewood Cliffs, NJ, 1988.

McLeod G, 1992. **A Model for Representation, Integration and Management of Methods**. South African Computer Journal.

McLeod G, 1993. **A generic method model for the Representation, Comparison, Integration and Management of Methods**, Proceedings: 1st International Conference on Information Systems, Henley, U.K.

McLeod G, 1995. **MetaTool: A tool for the engineering and management of methods**, Proceedings: 1st International conference on Meta-CASE. Sunderland, U.K., Jan 1995.

Murphy S, Harvey S, Mattison K, 1990. **A Status Review of Development Methodologies in South Africa.** Empirical Project, UCT Information Systems, Nov 1990.

Software Composition Technologies, Inc., 1997. **Frequently Asked Questions (and Answers) Regarding Function Point Analysis.** Updated 28 Feb 1997.
http://ourworld.compuserve.com/homepages/softcomp/fpfaq.htm

Rational Software Corporation, 1997a. **Unified Modeling Language Summary**, vsn 1.0, 13 January 1997.

Rational Software Corporation, 1997b. **Unified Modeling Language Semantics**, vsn 1.0, 13 January 1997.

Rational Software Corporation, 1997c. **Unified Modeling Language Notation Guide**, vsn 1.0, 13 January 1997.

Rational Software Corporation, 1997d. **Unified Modeling Language Glossary**, vsn 1.0, 13 January 1997.