1 Introduction
Though the demand for software continues to increase, software development, however, continues to be a time-consuming, expensive and error-prone process. Little progress has been made in greatly improving the software development process. What gains have been made, have been incremental, with minimal profound impact on the state of software. If software development is to improve as a process, we need to be able to provide development managers with the right information so they can make informed decisions in planning and scheduling activities and allocating resources during development.

Software testing is one of the most time-consuming and labor-intensive activities performed during the software development lifecycle. While practically impossible to exhaustively test any software system, it is possible to narrow and direct testing resources to those modules most likely to contain defects. The challenge is in identifying those modules more likely to contain defects, early in development. The prudent application of design measures can accomplish just that.

2 Software Measurement
In the last decade, object-oriented development has become the predominant methodology for developing software. Meyer describes object-oriented development as the “construction of software systems as structured collections of abstract data type implementations”[1]. This is in contrast to top-down, functional design which directs designs to start with the most abstract description of the systems function and then successively decompose into progressively smaller and smaller units of functionality until the system can no longer be decomposed. This leads to a top-down hierarchy of functional units.

There has been much research in the area of software measures for these functional units of software. The emphasis of measurement for functionally developed software has centered around four common categories: size, coupling, cohesion and complexity. Traditionally the size of a unit has been calculated by counting the number of lines of code in the unit, using various standards for the definition of a line of code. The theory being that modules with more lines of code take more time to test and are harder to maintain. Simple lines of code counts, though easy to compute, do not take into account differences in computer languages, nor do they account for the relative difficulty of the various language constructs, i.e. shorter programs that use complex language constructs versus longer programs that may contain several lines of simple expressions.
Another popular measure is that of coupling between modules. Coupling is the measure of interdependence between modules. For functional software, there are four general coupling measures (in order from acceptable coupling to detrimental coupling):

1) Data coupling – in which modules communicate by passing parameters;
2) Stamp coupling – in which two modules accept the same record type as a parameter;
3) Control coupling – in which a parameter passed by one module into another for the purpose of controlling its behavior;
4) Common coupling – in which two or more modules refer to the same global data; and
5) Content coupling – in which one module directly accesses and alters the contents of another module.

Cohesion is another common measure for functional software and describes the functional strength of individual modules. In order words, it measures the degree to which the individual pieces of a module are needed to perform the basic function of the module. There are seven kinds of cohesion and range, from the best in which the module performs a single, well-defined function to the worst in which the module performs several, unrelated functions.

The final popular functional measure is complexity. Complexity is a commonly used term intended to measure the totality of all internal software product attributes. There have been many complexity measures, with the most prevalent ones being those defined by McCabe [2] and Halstead [3]. Each of these complexity measures is an indirect measure and arrives at a single value for complexity by directly measuring various software attributes – nodes and edges for the McCabe measure and operators and operands for Halstead measure.

3 Object-Oriented Software Measures

Though certain aspects of traditional software measures can be applied to object-oriented software, by and large the measurements lose their original intent and are therefore, no longer applicable. The chief reason being that the properties and relationships among object-oriented software are fundamentally different from those of functionally developed software.

Object-oriented software utilizes the principles of localization, encapsulation, information hiding, inheritance and abstraction.

- **Localization** is the process of placing items in close proximity to one another. Functional decomposition localizes information around functions while the object-oriented approach localizes information around objects.

- **Encapsulation** is the packaging of a collection of items. In function decomposition subprograms are the mechanisms for encapsulation while objects encapsulate state and operations. Encapsulation means the basic unit is the object and not the subprogram as in functional decomposition.
• *Information hiding* is the suppression of details - only showing what information that is required. In object-oriented software this leads to both public and private interfaces.

• *Inheritance* is the mechanism by which a object acquires characteristics from one or more other objects. There is no equivalent to inheritance in functionally decomposed software.

• *Abstraction* is the mechanism for focusing on the important details while ignoring the inessential ones.

The fundamental unit or module in object-oriented systems is the class. A class is a template that defines the structure and capabilities of an object instance. A class definition encapsulates both the state data and the behavior for an object of the class. Using the properties above, classes are an abstraction of real-world entities such as a circle or a list. Classes encapsulate all relevant data and operations for these entities and present only the relevant ones in the public interface of the class (information hiding). Classes also localize the operations on an object around the state information for that object. Finally, classes may inherit the state information and the relevant operations from other classes, enabling designers to extend the definition of a class to form a new, related class.

As with functionally developed software, there are three basic attributes of object-oriented software that directly relate software quality: 1) size, 2) coupling, and 3) cohesion. Each of these are described in more detail below.

3.1 Size

This section will include a definition of size metrics for object-oriented software and discuss research that relates these metrics to software quality.

3.2 Coupling

This section will include a definition of coupling metrics for object-oriented software and discuss research that relates these metrics to software quality.

3.3 Cohesion

This section will include a definition of cohesion metrics for object-oriented software and discuss research that relates these metrics to software quality.

4 Application to Ada 95

This section will detail specific size, coupling and cohesion metrics and describe how to apply these metrics to Ada 95. For each metric, data from 4 different subsystems consisting of several hundred Ada classes will be given. The specific metrics are listed below.
4.1 Depth of Inheritance
4.2 Number of Children
4.3 Number of Ancestors
4.4 Number of Operations
4.5 Number of Attributes
4.6 Number of Total Operations
4.7 Number of Total Attributes
4.8 Number of Class-variable (global)
4.9 Number of Class-wide operations
4.10 Message Passing Coupling
4.11 Number of Abstract Data Types
4.12 Abstract Data types References
4.13 Lack of Cohesion of Methods

5 Summary and Future Work
This section will summarize the finding of the paper and outline future work to be done in this area.

6 References

