CHAPTER 2

Literature Review in Object-Oriented Design Metrics

Software Metrics

Software metrics are measures that are used to quantify software, software development resources, and/or the software development process. This includes items which are directly measurable, such as *lines of code*, as well as items which are calculated from measurements [46], such as *earned value*.

Everyone who develops software uses some kind of software metrics. However, when asked what software metrics are, the tendency is to restrict the response to software size measurements, such as lines of code or function points. In reality, software metrics include much more than primitive measures of program size [47].

Software metrics include calculations based on measurements of any or all components of software development [47]. For example, consider the system integrator who wishes to determine the status of a project's test phase. He or she will undoubtedly ask for information on the proportion of tests that have been executed, the proportion that were executed successfully, and the number of defects identified. These measures are all examples of primitive - yet useful - software metrics.

Consider the engineer who is responsible for improving the performance of a software product. He or she will consider items such as memory utilization, I/O rates, and the relative complexity of software components. These are also examples of software metrics.

There is nothing overly complicated about software metrics. In fact, the biggest challenge in establishing an effective metrics program has nothing to do with the formulas, statistics, and complex analysis that are often associated with metrics. Rather, the difficulty lies in determining which metrics are valuable to the company, and which procedures are most efficient for collecting and using these metrics [48].

The SPC conducted research with a number of software developers and managers, many from well-established companies few of whom had much experience with software metrics. A common assumption emerged: software metrics are used simply to measure programmer productivity against an industry standard.

Apart from providing little value, such comparisons are apt to foster resentment among the programming staff, and earn little support for a metrics program.

A healthy metrics program focuses on much more than the measurement of programmer productivity. Consider these areas of software development which can benefit from a well-planned metrics program:

- project management
- product quality

- product performance
- development process
- cost and schedule estimation

The key to the effective use of software metrics within an organization is to prepare a plan describing how metrics will be used to meet strategic management goals [46].

Metric types:

There are 3 metric types:

- ***Direct Measurement:*** like the length of source code or duration of testing process. (These measurement activities involve no other attributes or entities) [49].
- ***Indirect/Derived Measurement:*** like Module defect density = Number of defects/ Module Size. (These are combinations of other measurements) [49].
- ***Predictive Measurement:*** like predict effort required to develop software from the measure of its functionality – function point count. (These measurements are systems consisting of mathematical models together with a set of prediction procedures for determining unknown parameters and interpreting results) [49].

Classifications of software metrics:

Software metrics supply data about different aspects of the software development process. The basic components of software development, i.e. *process*, *product* and *resources* characteristics, define individual application domains for software measurement and allow different measurement strategies. At the same time they serve as a starting-point for a detailed classification of software metrics ([50], [51], [52]).

- ***process metrics***
    - ***maturity metrics***
        - organization metrics
        - resources, personnel and training metrics
        - technology management metrics
        - metrics for documentation standards
        - process metrics
        - data management and analysis metrics
        - process control metrics
    - ***management metrics***
        - milestone metrics
        - risk metrics

        - review metrics
        - productivity metrics
    - ***life cycle metrics***
        - problem definition metrics
        - requirements analysis and specification metrics
        - design metrics
        - implementation metrics
        - maintenance metrics

- ***product metrics***

    - ***size metrics***
        - size of project attributes (e.g. lines of code, number of documentation pages, number of test cases etc.)
        - development time metrics
        - development costs metrics
        - metrics for the consumption of resources
    - ***architecture metrics***
        - number of components
        - number of language paradigms in the product
        - level metrics
    - ***structure metrics***
        - depth metrics
        - width metrics
        - component coupling metrics
    - ***quality metrics***
        - functionality metrics (suitability, accuracy, interoperability, compliance, security)
        - reliability metrics (maturity, fault tolerance, recoverability)
        - usability metrics (understandability, learn ability, operability)
        - efficiency metrics (system behavior over time, usage of resources)
        - maintainability metrics (analyzability, changeability, stability, testability)
        - portability metrics (adaptability, install ability, conformance, replace ability)
    - ***complexity metrics***
        - computational complexity metrics
        - psychological complexity metrics (size, data flow, control flow, clarity, entropy, topology)

- ***resources metrics***

    - ***personnel metrics***
        - programming experience metrics

        - communication level metrics
        - productivity metrics
        - team structure metrics
    - ***software metrics***
        - performance metrics
        - paradigm metrics
        - replacement metrics
    - ***hardware metrics***
        - performance metrics
        - reliability metrics
        - Availability metrics

Object Oriented Metrics

Object-Oriented Analysis and Design of software provide many benefits such as reusability, decomposition of problem into easily understood object and the aiding of future modifications. But the OOAD software development life cycle is not easier than the typical procedural approach. Therefore, it is necessary to provide dependable guidelines that one may follow to help ensure good OO programming practices and write reliable code. Object-Oriented programming metrics is an aspect to be considered. Metrics to be a set of standards against which one can measure the effectiveness of Object-Oriented Analysis techniques in the design of a system.

OO metrics which can be applied to analyze source code as an indicator of quality attributes. The source code could be any OO language.

Metrics for OO Software Development Environments [53], [54]

System Level - Standard View

There is only the Standard view at this level. This view shows the numbers of Packages, Classes, Methods and Statements in the system. It also shows the Total and Average (per method),Cyclomatic complexity, the Total Halstead Effort and the two forms of the maintainability index.

Package Level - Standard View

There is only the Standard view at this level. This view shows the numbers of Classes, Methods and Statements in the package. It also shows the Total and Average (per method) , Cyclomatic complexity, the Total Halstead Effort and the two forms of the maintainability index.

Class Level - Standard View

This view shows the numbers of Methods and Statements in the class. It also shows the Total and Average (per method) and maximum Cyclomatic complexity and the Total Halstead Effort. There are also a number of other class-related standard metrics -

LCOM: Lack of cohesion of methods - This metric measures the correlation between the methods and the local instance variables of a class. High cohesion indicates good class subdivision. Lack of cohesion or low cohesion increases complexity. Classes with low cohesion could probably be subdivided into two or more subclasses with increased cohesion. It is calculated as the ratio of methods in a class that do not access a specific data field, averaged over all data fields in the class. The LCOM version that we use is LCOM* as defined by Henderson-Sellers. This algorithm should produce answers in the range 0 to 1 with zero representing perfect cohesion (each method accesses all attributes), however we have noticed that some values exceed 1. This can occur in three scenarios:

1. Where there are no instance attributes defined in the class (i.e. it inherits them all from super classes). We view this as legitimate since the inherited class may be modifying or adding behavior - in this case we set LCOM to 0.
2. Where there are no references to instance attributes but there are instance attributes defined. We view this as bad - we have created attributes that are not referred to within the class - this can only mean that they are either not used or are used in a subclass - if the latter then they should be defined in the subclass. We therefore set LCOM to the number of instance variables. The only other case where attributes are defined but not used are classes which define static constants for use by other classes. We don't make an exception for this but examination of the class by eye (or the class name - most people put 'Constants' in the name of such classes) should confirm if this is the case. Some implementations of LCOM don't count static instance variables - we took the decision to leave them in (mainly because naming conventions cover the legitimate case above)
3. Where the number of references to attributes is less than the number of attributes. In this case some of the attributes are not being referenced. This is really a less extreme case of 2 so we let the algorithm take its course and let LCOM ride up above 1.

UWCS: This is calculated from the number of methods plus the number of attributes of a class. Smaller class sizes usually indicate a better designed system reflecting better distributed responsibilities. In other words you didn’t just stuff all the functionality into one big class. It’s difficult to set hard and fast rules about this but you should look carefully at classes where UWCS is above 100.

Number of queries: number of methods which return a value.

Number of commands: number of 'void' methods i.e. methods which do something but don't return a variable.

Class Level - Design View

This shows the two forms of the maintainability index at the class level as well as the following other details:

Number of External Methods Called, No. of Methods Called in class hierarchy and No. of local methods called: these give some idea of the level and complexity of interaction between the class and other classes (both in the class's hierarchy and external to it). These figures are used in calculating metrics such as RFC, LCOM, MPC, Fan In and Fan Out.

Number of instance variables, No. of modifiers, No. of interfaces implemented and No. of packages imported: these give additional information about the class's level of semantic complexity

Response for a class (RFC) - This metric measure the response set of a class which is defined as all the local methods of the class and all the methods called by local methods of the class.

Message Passing Coupling (MPC): This metric measures the numbers of messages passing among objects of the class. A larger number indicates increased coupling between this class and other classes in the system. This makes the classes more dependants on each other which increases the overall complexity of the system and makes the class more difficult to change.

Fan-In and Fan-Out - The Fan In of a function is the number of unique functions that call the function. The Fan Out metric counts the number of distinct non-inheritance related class hierarchies on which a class depends. This dependency is called coupling. Excessive coupling increases sensitivity to changes in other parts of the design and makes a module harder to understand and therefore to change since it is interrelated with other modules. Designing systems to reduce coupling between modules improves modularity and promotes encapsulation.

Reuse ratio - calculated as - (number of super classes above this class in the class hierarchy)/(total number of classes in the class hierarchy)

Specialization Ratio - Calculated as (number of subclasses below this class in the class hierarchy)/ (number of super classes above this class in the class hierarchy)

Number of super classes - Used to calculate Specialization and Re-use ratios. Excludes the class Object – which all classes ultimately inherit from.

Number of subclasses - Used to calculate Specialization and Re-use ratios.

Method - Standard View

Calculated Complexity: This allows you to provide your own method of calculation to assess the complexity of your code. You can re-implement this method providing you have bought the source code version of the product.

Number of Arguments: If a method has a large number of arguments it may be a sign that the method is trying to do too much. It also means that this method signature has a greater propensity to change. You should check that all the arguments are used. If the arguments are primarily the arguments of another object then pass in the object. If the reason that you haven't used the object is that you don't want the class that the method is in to know about it then create an interface for the object and get it to implement it. Create an object to hold the arguments.

Number of comments: Comments are a matter of taste. If the method is complex and has no comments this is probably bad but an accessor method without comments is probably fine.

Variables Declared: if a lot of variables are being declared then your method may be doing too much, if that is the case the method should be split using the splitting strategies.

Variables Referenced: if a lot of variables are being referenced then your method may be doing too much, if that is the case the method should be split using the splitting strategies.

Number of Expressions: This is an alternative measure to the number of statements. If the average number of expressions per statement is high this may indicate a complex method.

Max nesting: too deep nesting indicates complexity. Exceptions to this general rule would include iterating across multi-dimensional arrays. The total depth of nesting is also shown as a large number of nested statements may make the method difficult to read unless care has been taken with the formatting. You can reduce the depth of nesting by splitting the functionality of the method using the 'Nested chunks' strategy.

Number of Casts: casts are expensive in terms of performance. Casting is almost as expensive as creating a new instance of an object. You should program to avoid casting if at all possible. One of the main reasons for using casting is to extract data from untyped collections.

Number of loops: loops are an important place to look for performance issues. Code that is only run infrequently is unlikely to have a big impact on performance.

Number of operands - used in calculation of Halstead measurements. These are tokens (parsed atoms of the code) that are not operators, they would include variables, numbers, strings.

Number of operators: used in calculation of Halstead measurements. Operators include mathematical operators (e.g. * + -), keywords and message calls.

Number of external methods called - The more external methods that a class calls the more tightly bound that class is to other classes. If you feel that a class is too tightly bound then this column will help you to identify the methods that are responsible for external calls. You can see the actual methods called and the number of times that they are called by double clicking on the selected method record to bring up the method details dialog

Number of classes referenced - This is similar to the situation with the external methods - you can use this to identify methods that reference external classes and identify them in the method details dialog.

Number of Exceptions Thrown and Number of Exceptions Referenced - These columns simply allow you to identify methods that throw and catch Exceptions. You can identify these in the method details dialog.

How to split methods

There are a number of good reasons to split methods:

1. It makes the methods easier to read and therefore to understand their function
2. It is harder for bugs to 'hide' in the code
3. By separating functionality out into smaller methods there is more likely that the code will be re-used

The actual process of splitting depends on what you are trying to do and the particular algorithm that you are trying to implement.

There are three main types:

1. **Sequential chunks**: in this case you separate each chunk out into a method and your original method just calls each of the methods in succession, handling the interim variables that need to be passed from one method to another.

2. **Nested chunks:** this is the case where you have a number of nested loops each of which (apart from the deepest) does some preparatory work (optionally), calls the inner loop then (optionally) does some work after calling the loop. In this case each of the inner loops becomes a separate method and your original method contains only the outer loop and the call to the first nested loop
3. **Conditional chunks**: this is where you have a number of alternative paths in your algorithm depending on condition(s) passed in or accessed by the method. Each conditional chunk should be implemented as a separate method and the original method should simply handle the outer conditions and decide which of the methods to call.

The need of the formal models:

The formal model is to capture the essence of component-based software at a high level

Formal model definition:

The definition has been implemented in a prototype development environment in which simple component based programs can be built. The pictures used to illustrate the definitions were generated by this prototype. Before presenting our definitions in Section 3, we give a brief overview of component models.

Software reuse has long been one of the major issues in the world of software engineering, where code reuse is seen as the key to many benefits, such as increased productivity, improved reliability, and ease of maintenance. As a result, many software reuse technologies have been developed over the past few years.

Software reuse was first realized in the late 50's with the development of libraries of pre-compiled subroutines such as large numerical libraries for engineering and scientific computation. Reuse via subroutines has limited applicability, however, because of the difficulty of encapsulating high-level functionality in subroutines.

A major step forward was made with the advent of object-oriented programming (OOP), which provided inheritance as its primary reuse mechanism. As OOP research and practice has progressed, other reuse techniques have been devised, such as object composition. More significantly, object-oriented application frameworks have emerged, providing the means to capture very large application design patterns in the form of "inverted libraries" which call the code supplied by the application developer, rather than the other way round as with traditional libraries [55].

Another popular reuse technique involves design patterns, which provide guidance during the design phase of software by suggesting high-level

organizations for the necessary abstractions [56]. Design patterns provide methodology but not tools.

A recent trend in software engineering is towards software development using components. This development methodology is based on the observation that hardware design and development has evolved well beyond the “build-from scratch” era. Hardware these days is built from “off-the-shelf” components which are customizable to a degree, cheap because they are manufactured in large quantities, are well tested and reliable, and have a well defined interface. The aim of component-based software development is to attain reuse, economy, reliability and so forth by creating large catalogues of software components for assembling into software systems. This approach has some other desirable consequences as well, such as standardization, resulting in software products from different developers working correctly together.

The analogy with hardware components on which the notion of software components rests, leads naturally to a visualization of component-based software as a network of boxes communicating with each other via connecting wires.

In recent years several software development tools have appeared that provide visual programming based on components. Visual Age for Java [57], Parts for Java and Java Studio use Java Beans as the underlying component mechanism. In Visual Age and Parts, the visual programming by wiring components together is largely limited to programming the user interface. Components that implement user interface items are represented in the visual program by their usual interface appearance, and wires connect them to indicate the flow of messages between these visible components and others which have no visual representation. A more direct representation of the box-and-wire metaphor is provided by Java Studio (no longer available), in which icons representing the components are wired together.

Another visual manifestation of components can be seen in Microsoft Office products where “objects” of differing types (word processor document, spreadsheet, picture) can be embedded in each other. The objects are components, and are linked together and pass messages to each other according to a standard protocol, Object Linking and Embedding (OLE).

Apple Computer spent several years designing and building a similar but more flexible system called OpenDoc, in which documents consisting of communicating components could be built by dragging components into a workspace [58]. Like many other good ideas, the concept of components has developed in a rather *ad hoc* way. Even though the various component technologies have very similar capabilities, they are all different in detail, and few of them are defined in clear, concise terms the way programming languages are (or can be).

A Formal Model for Component-Based Software

- Provides a simple but precise characterization of components which may help dispel confusion resulting from the rapid evolution of many subtly different component technologies.

- Describes an underlying structure on which the syntax of component-based languages can be based, as well as a semantics for such languages.

- Provide a basis for a formal testing and verification methodology.

- Allows recursive components which none of the existing component technologies provide;