IJARCCE

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
ISO 3297:2007 Certified
Vol. 5, Issue 7, July 2016

A Review on Metrics in SOA

Simardeep Kaur[1], Saloni Khanna[2]

M.Tech (IT), Dept of Information Technology, Adesh Institute Of Engineering & Technology, Faridkot, India [1, 2]

Abstract: A service-oriented architecture (SOA) is an architectural pattern in computer software design in which application components provide services to other components via a communications protocol, typically over a network. The principles of service-orientation are independent of any vendor, product or technology. A service is a self-contained unit of functionality, such as retrieving an online bank statement. By that definition, a service is an operation that may be discretely invoked. However, in the Web Services Description Language (WSDL), a service is an interface definition that may list several discrete services/operations. And elsewhere, the term service is used for a component that is encapsulated behind an interface. Metrics play an important role in empirical software engineering research as well as in industrial measurement programs. The metrics presented in this paper measure the difference between class inheritance and interface programming. The metric values of class inheritance and interface prove which program is good to use. Our goal is comparing the inheritance and interface concepts in object oriented programming through cohesion- metrics. Complexity, Service granularity metrics

Keywords: SOA, Cohesion, complexity, granulity ,Metrics, WSDL, Inheritance, Services.

I. INTRODUCTION

Service-Oriented Architecture (SOA) is emerging as a promising development paradigm, which is based on encapsulating application logic within independent, loosely-coupled stateless services, that interact via messages using standard communication protocols and can be orchestrated using business process languages, The notion of a service is similar to that of a component, in that services, much like components, are independent building blocks that collectively represent an application. However, services are more platform independent, business-domain oriented, and autonomous and hence decoupled from other services as compared with components. Service-oriented systems in conjunction with supporting middleware represent Service-Oriented Architecture (SOA), a more abstract concept which is founded on the idea of discovery and orchestration whereby a business process or workflow can identify at runtime the most suitable services for a particular scenario and dynamically compose them in order to satisfy a particular domain requirement. Moreover, in SOA, enterprises should consider services as enablers of business processes that reflect workflows within and between organizations, rather than treating them simply as interfaces to software functionality. Although SOA is becoming an increasingly popular choice for the development of enterprise software, service-oriented (SO) design principles are not well understood and documented, with contradicting definitions and guidelines making it hard for software engineers and developers to work effectively with service-oriented concepts . Consequently, service-oriented systems are often developed in an ad-hoc fashion potentially resulting in lower-quality software being produced.

An important mechanism in a SOA is the Dynamic Discovery of services:

The interaction model of the basic SOA consists of three key players, the service providers, the service requestors, and the intermediating directory service. First, the service providers register with the directory service, then clients can query the directory service for providers and browse the exposed service capabilities.

Typically a directory service supports:
• A look-up service for clients
• Scalability of the service model: services can be added incrementally
• Dynamic composition of the services: the client can decide at runtime which services to use.

Some of the constraints that apply to the SOA architectural style are given below based on the Fig 1

- Service users send requests to service providers.
- A service provider can also be a service user.
- A service user can dynamically discover service providers in a directory of services.
- An ESB can mediate the interaction between service users and service providers.

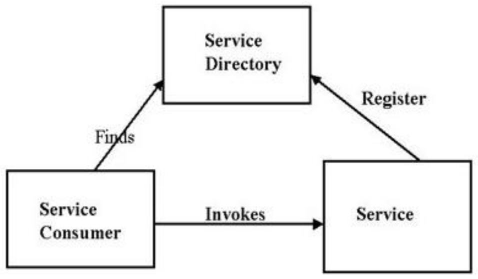


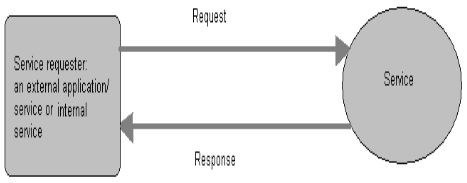
Fig 1 SOA

1.1 Services:

- A service is a logical representation of a repeatable business activity that has a specified outcome (e.g., check customer credit, provide weather data)

- Services are the building blocks of SOA enabled application.
- It is basically an encapsulation of data.
- A service consists of an interface, has an implementation.
- The service interface defines a set of operations, which exposes its capabilities.

Static and Dynamic Services
To invoke a service provider, a service user needs to determine the interface of the service (operations available, expected input and output) and locate the actual service. For static binding, as shown in Fig.2, the service interface and location must be known when the service user is implemented or deployed. The service user typically has a generated stub to the service interface and retrieves the service location from a local configuration file. The service user can invoke the service provider directly, and no private or public registry is involved. For dynamic services, as shown in Fig. 3, a provider must register the service to a registry of services. The registry is queried by service users at runtime for the provider's address and the service contract. After acquiring the required information, the service user can invoke the operations of the service provider.


Request
Service requester: an external application/ service or internal service
Service
Response
Fig 2 Static Binding

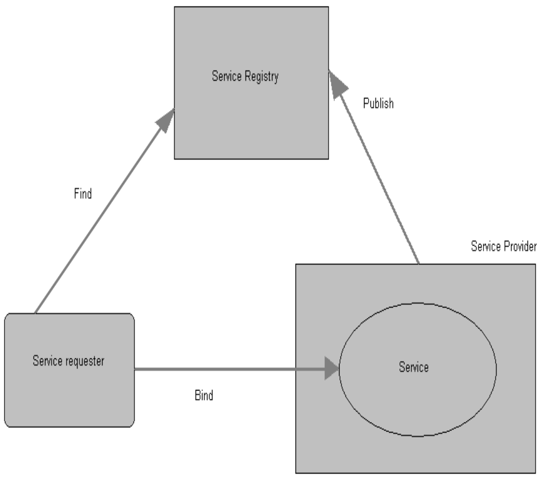


Fig 3 Dynamic Binding

1.2 Software Architecture:
- Its defines as "the structure or structures of a system, which defines software elements, the externally visible properties of those elements, and the relationships among them".
- Examples of such elements could include compilation units and processes, each with its own related structure.
- Software architecture is typically documented using multiple views.
- A "view" is described as "a representation of a set of system elements and the relationships associated with them".

1.3 SOA Layers:
Basically SOA aims at the provisioning of abstract software functionality through services that can be flexibly composed to implement business processes. The five functional layers are as follows (bottom to top) shown in Fig 2
- Operational systems: Represents existing IT assets, and shows that IT investments Are valuable and should be leveraged in an SOA.
- Service components: Realize services, possibly by using one or more applications in the operational systems layer. As you can see on the model, consumers and business processes do not have direct access to components, but just services. Existing components can be internally reused, or leveraged in an SOA if appropriate.
- Services: Represents the services that have been deployed to the environment. These services are governed discoverable entities.
- Business Process: Represents the operational artifacts that implement business processes as choreographies of services.
- Consumers: Represents the channels that are used to access business processes, services, and applications.

The four non-functional layers are (left to right):
- Integration: Provides the capability to mediate, route, and transport service requests to the correct service provider.
- Quality of service: Provides the capability to address the nonfunctional requirements of an SOA (for example, reliability and availability).
- Information architecture: Provides the capability to support data, metadata, and business intelligence.
- Governance: Provides the capability to support business operational life cycle management in SOA.

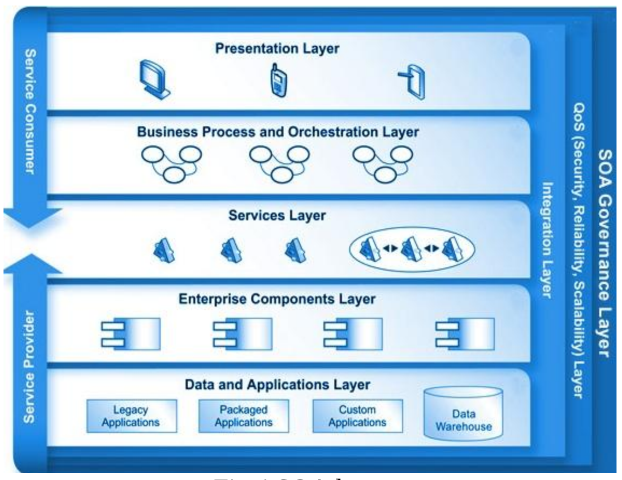


Fig 4 SOA layers

1.4 Web Services in SOA
SOA is an architectural style, whereas Web services are a technology that can be used to implement SOAs. The Web services technology consists of several published standards, the most important ones being SOAP and WSDL. Other technologies may also be considered technologies for implementing SOA, such as CORBA. Although no current technologies entirely fulfill the vision and goals of SOA as defined by most authors, they are still referred to as SOA technologies. The relationship between SOA and SOA technologies is represented in Fig 3

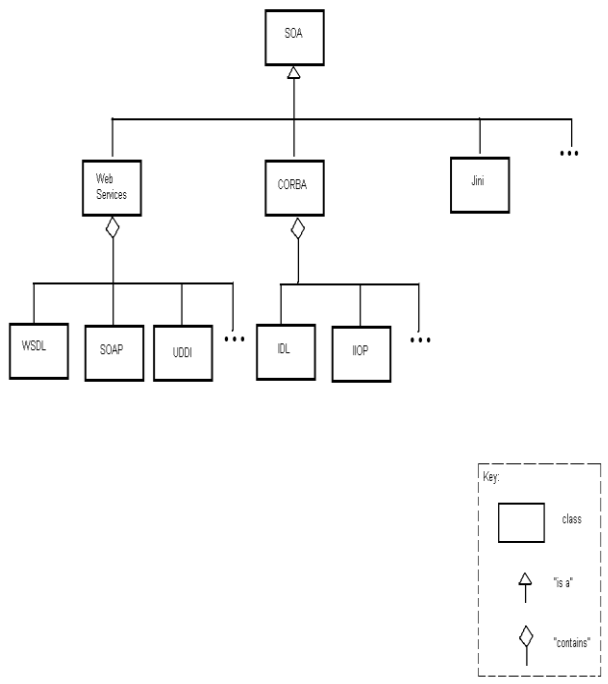


Fig 5 SOA Web Services

1.5 View
A “view” is described by Clements as “a representation of a set of system elements and the relationships associated with them”. Together, these definitions are saying that the software architecture serves multiple purposes and hence cannot be captured in a single model (i.e., a view).Kruchten proposed the use of the five following views:

- The logical view that supports the system’s services provided to the end-user
- The process view that describes the synchronization and concurrency aspects
- The development view that supports construction of the system and management of it development.
- The physical view that maps the elements of the previous three views onto processing nodes a fifth view that ties the other views together by a set of scenarios describing how the elements of the other views cooperate Other sets of views have been proposed. Even more views are possible and necessary. Thus there are multiple abstractions (i.e., elements and their relationships) associated with a given software architecture.

1.6 SOA Implementation in Java EE 6
In this section, we will cover how web services can be realized using Java, one of the most widely-used enterprise technologies. There are several web services implementation in Java technology such as Axis2 and CFX from Apache, Spring Web Services, JBossWS and Glassfish Metro. However, we will only discuss Metro, a reference implementation of Java EE web services technologies.
Metro web services stack is fully supported in Glassfish server which is also a reference implementation of Java EE specifications. It mainly consists of two components: Java API for XML-based Web Services (JAX-WS) and Java API for REST ful Web Services (JAX-RS). Our emphasis will be on the former rather than the latter whose data exchange could be JSON, XML or any other data exchange protocol and whose operations are mainly in the form of HTTP methods such as GET, PUT, POST, or DELETE.

II. SOFTWARE METRICS

Tools for anyone involved in software engineering to understand varying aspects of the code base, and the project progress. They are different from just testing for errors because they can provide a wider variety of information about the following aspects of software systems:

- Quality of the software, different metrics look at different aspects of quality, but this aspect deals with the code.
- Schedule of the software project on the whole. Some metrics look at functionality and some look at documents produced.
- Cost of the software project. Includes maintenance, research and typical costs associated with a project.
- Size/Complexity of the software system. This can be either based on the code or at the macro-level of the project and its dependency on other projects.
-

General uses of Metrics
- Software metrics are used to obtain objective reproducible measurements that can be useful for quality assurance, performance, debugging, management, and estimating costs.
- Finding defects in code (post release and prior to release),predicting defective code, predicting project success, and predicting project risk.
- There is still some debate around which metrics matter and what they mean, the utility of metrics is limited to quantifying one of the following goals: Schedule of a software project, Size/complexity of development involved, cost of project, quality of software.

Types of Metrics
1. Requirements metrics
a. Size of requirements
b. Traceability
c. Completeness

d. Volatility
2. Product Metrics
a. Code metrics
b. Lines of code LOC
c. Design metrics – computed from requirements or design documents before the system has been implemented
d. Object oriented metrics- help identify faults, and allow developers to see directly how to make their classes and objects more simple.
e. Test metrics
f. Communication metrics – looking at artifacts i.e. email, and meetings.

III. COHESION METRICS

Cohesion refers to the degree to which the elements of a module belong together. Thus, it is a measure of how strongly related each piece of functionality expressed by the source code of a software module is. Cohesion is an ordinal type of measurement and is usually described as “high cohesion” or “low cohesion”. Modules with high cohesion tend to be preferable because high cohesion is associated with several desirable traits of software including robustness, reliability, reusability, and understandability whereas low cohesion is associated with undesirable traits such as being difficult to maintain, test, reuse, and even understand. Cohesion is often contrasted with coupling, a different concept. High cohesion often correlates with loose coupling, and vice versa. The software metrics of coupling and cohesion were invented by Larry Constantine in the late 1960s as part of Structured Design, based on characteristics of “good” programming practices that reduced maintenance and modification costs. Structured Design, cohesion and coupling were published in the article Stevens, Myers & Constantine (1974) and the book Yourdon & Constantine (1979); the latter two subsequently became standard terms in software engineering.

Cohesion is increased if:
The functionalities embedded in a class, accessed through its methods, have much in common. Methods carry out a small number of related activities, by avoiding coarsely grained or unrelated sets of data.

Advantages of high cohesion (or “strong cohesion”) are:
Reduced module complexity (they are simpler, having fewer operations).Increased system maintainability, because logical changes in the domain affect fewer modules, and because changes in one module require fewer changes in other modules. Increased module reusability, because application developers will find the component they need more easily among the cohesive set of operations provided by the module. While in principle a module can have perfect cohesion by only consisting of a single, atomic element – having a single function, for example – in practice complex tasks are not expressible by a single, simple element. Thus a single-element module has an element that either is too complicated, in order to accomplish a task, or is too narrow, and thus tightly coupled to other modules. Thus cohesion is balanced with both unit complexity and coupling.
Types of cohesion
Cohesion is a qualitative measure; meaning that the source code to be measured is examined using a rubric to determine a classification. Cohesion types, from the worst to the best, are as follows:
Coincidental cohesion (worst)
Coincidental cohesion is when parts of a module are grouped arbitrarily; the only relationship between the parts is that they have been grouped together (e.g. a “Utilities” class).
Logical cohesion
Logical cohesion is when parts of a module are grouped because they are logically categorized to do the same thing, even if they are different by nature (e.g. grouping all mouse and keyboard input handling routines).
Temporal cohesion
Temporal cohesion is when parts of a module are grouped by when they are processed - the parts are processed at a particular time in program execution (e.g. a function which is called after catching an exception which closes open files, creates an error log, and notifies the user).
Procedural cohesion
Procedural cohesion is when parts of a module are grouped because they always follow a certain sequence of execution (e.g. a function which checks file permissions and then opens the file).
Communications/informational cohesion
Communicational cohesion is when parts of a module are grouped because they operate on the same data (e.g. a module which operates on the same record of information).
Sequential cohesion
Sequential cohesion is when parts of a module are grouped because the output from one part is the input to another part like an assembly line (e.g. a function which reads data from a file and processes the data).
Functional cohesion (best)
Functional cohesion is when parts of a module are grouped because they all contribute to a single well-defined task of the module.
METRICS
Cohesion can be defined as the intra-modular functional relatedness of a software module. As previously stated, we can categorize cohesion is into seven levels (ranging from low cohesion to high cohesion).

Static Cohesion Metrics
There are a lot of alternative measures which are being proposed for measuring cohesion. A broad survey on the current state of cohesion measurement is carried out by Briand et al. [8] in object-oriented systems and he provided fifteen separate measurements of cohesion. Following is a review of these measures in the following subsections.
Chidamber and Kemerer
The Lack of Cohesion in Methods (LCOM1) measure was first suggested by Chidamber and Kemerer
[5].Given n methods M1, M2, …, Mn contained in a class C1 which also contains a set of instance variables {Ii}.

IJARCCE
ISSN (Online) 2278-1021
ISSN (Print) 2319 5940
International Journal of Advanced Research in Computer and Communication Engineering
ISO 3297:2007 Certified
Vol. 5, Issue 7, July 2016

Then for any method Mi we can define the partitioned set of

$$P = \{(Ii, Ij) \mid Ii \cap Ij = \phi\} \text{ and } Q = \{(Ii, Ij) \mid Ii \cap Ij \neq \phi\}$$

then LCOM = |P| - |Q|, if |P| > |Q| =0 otherwise
LCOM is a count of the number of method pairs whose similarity is zero.
Example: Consider a class C with three methods M1,M2 and M3. Let {I1} = {p,q,r,s,t} and {I2} = {p,q,t } and {I3} = {a,b,c }. {I1} ∩ {I2} is non-empty, but {I1} ∩ { I3} and { I2} ∩ { I3} are null sets. LCOM is the (number of Null intersections - number of non-empty intersections), which is 1 in this case. LCOM is considered as an Inverse cohesion measure. An LCOM value of zero specifies a cohesive class.

Other Static Cohesion Metrics
Briand et al. classify a set of cohesion measures for object-based systems [9,10] which are adapted in [11] to object-oriented systems. For this adaption a class is viewed as a collection of data declarations and methods. A data declaration is a local, public type declaration, the class itself or public attributes. There can be data declaration interactions between classes, attributes, types of different classes and methods.

They categorized the different cohesive metrics based on the above principle into following categories:
1. Ratio of Cohesive Interactions (RCI)
2. Neutral Ratio of Cohesive Interactions (NRCI)
3. Pessimistic Ratio of Cohesive Interactions (PRCI)
4. Optimistic Ratio of Cohesive Interactions (ORCI).
Run-time/Dynamic Cohesion Metrics
Despite extensive research work conducted in the measurement of static cohesion, only a few metrics have been proposed for the measurement of cohesion at runtime.

Gupta et al. Metrics

Bieman and Ott [13,14] proposed the concept of Strong Functional Cohesion (SFC) and Weak Functional Cohesion (WFC) and then Gupta et al.[15] redefined these module cohesion metrics. Gupta et al.[15] commence the dynamic cohesion measurement using program execution based approach on the basis of dynamic slicing (dynamic slice is the set of all statements whose execution had some effect on the value of a given variable).

They use dynamic slices of outputs to measure module cohesion. According to them module cohesion metrics based on static slicing approach have got some insufficiencies in cohesion measurement. Their approach addresses the limitations of static cohesion metrics by considering dynamic behavior of the programs and designing metrics based on dynamic slices obtained through program execution. They defined SFC as module cohesion obtained from common defuse pairs of each type common to the dynamic slices of all the output variables and WFC as module cohesion obtained from defuse pairs of each type found in dynamic slices of two or more output variables.

Dynamic Metrics for GUI Programs
Though Graphical User Interfaces (GUIs) make the software easier to use from user's viewpoint however they Increase the overall complication of the software since GUI programs unlike conventional software are event based systems. The special characteristics of a GUI program imply that the traditional methods of evaluating complexity statically may not be the suitable ones as static analysis of source code emphasize only on the probability that what may happen when the program is executing whereas a dynamic analysis attempts to enumerate what actually happened during program execution.

Mitchell and Power [16] outline a new technique for collecting dynamic trace information from Java GUI programs and a number of simple runtime metrics areproposed. The exPubMet.Ob metric gives an estimation of level of coupling present in a GUI program and The priMet.ob metric shows that simple programs devote a greater proportion of their method access to the internal Working of their classes than the GUI program.
exPubMet.Ob: measure of the level of coupling within a program at runtime.

= Number of External Public methods called/Total Number of Objects created
priMet.ob : measure of the level of cohesiveness within a program.
= Number of Private methods called/Total number of objects created

IV. COUPLING METRICS

There are mainly seven different levels which we can use to find the characteristics of complexity of software product by establishing the correlation and interdependence between them.

These levels are as follows:
a) Control Structure
b) Module Coupling
c) Algorithm
d) Code
e) Nesting
f) Module Cohesion
g) Data Structure

Among all of these, "Coupling" and "Cohesion" are considered to be the most important attributes. Coupling and cohesion are the attributes which measure the degree or the strength of interaction and relationships among elements of the source code, for example classes, methods, and attributes in SOA software systems. One of the main objectives behind Object Oriented analysis and design is to implement a software system where classes have high cohesion and low coupling amongst them.

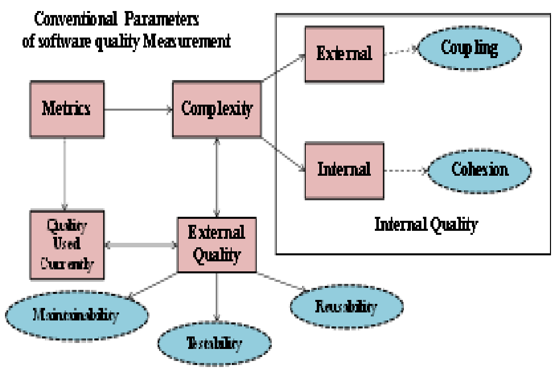


Fig 6 External and Internal attributes of a software product

Coupling in existing literature
Coupling or dependency is the degree to which each program module relies on each one of the other modules. Stevens et al. [17] first introduced coupling in the context of structured development techniques. According to them "coupling is the measure of the strength of association established by a connection from one module to another". In their opinion, the complexity of the software product will be dependent upon the interconnection and interdependence between the modules. As coupling is the degree of interdependence among the modules so that degree can be high as well as low depending on their bonding level.
The following is the set of different types of coupling in the order of the precedence from highest degree to the lowest one:

1. Content Coupling (high)
2. Common Coupling
3. External Coupling
4. Control Coupling
5. Stamp Coupling
6. Data Coupling
7. Message Coupling (low)

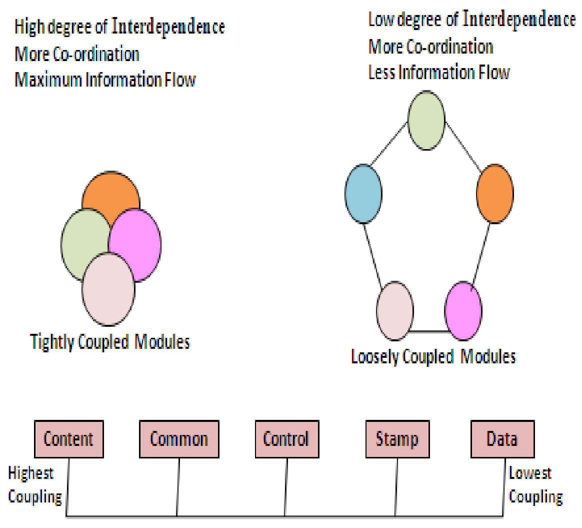


Fig 7 shows variety of coupling and the interdependence among modules

Coupling Measure
To determine the complexity, it is very important and useful to measure the coupling between modules. The higher the inter object coupling, the more scrupulous the testing needs to be. There are several matrices using this concept . Number of children metric defines number of sub-classes subordinated to a class in the class hierarchy. Coupling between Number of Objects is that two classes are said to be coupled if the methods of one class use the methods or attributes of the other class. Number of Dependencies IN is defined as the number of classes that depend on a given class [18]. Number of Dependencies OUT metric is defined as the number of classes on which a given class depends. Number of Association metric was suggested by Brian in which he stated that the number of association per class metric is the total number of associations a class has with other classes or with itself. Direct Dependency is direct association between services. This kind of dependency may exist between services explicitly when a service itself calls other services or a service is called by other services [19]. Indirect Dependency between services may occur in two cases. In the first case, when an indirect or transitive connection or association between the services is present. In the second case when the services share global data.

Static Coupling Metrics
There exists a large variety of measurements for coupling. A comprehensive review of existing measures performed by Briand et al. [20] found that more than thirty different measures of object-oriented coupling exist. The most prevalent ones are explained in the following subsections:
Chidamber and Kemerer suite of Metrics
Chidamber and Kemerer propose and validate a software metrics for object-oriented systems for the following basic purposes:
(a) To measure the unique aspects of Object Oriented approach.
(b) To measure the complexity of the design.
(c) To improve the development of the software.
The most accepted and commonly used coupling metrics amongst them are:

- Coupling Between Objects (CBO)
- Response for class (RFC)

Coupling Between Objects (CBO)
Chidamber and Kemerer first define a measure CBO for a class as, a count of the number of non-inheritance related couples with other classes [5]. If the methods of one class use the methods or attributes of the other that implies that the objects of both of the classes are coupled with each other. To improve the modularity of a software the inter coupling between different classes should be kept to a minimum. Beside reusability a high coupling also has a second weakness, a class that is coupled to other classes is susceptible to changes in those classes and as a result it becomes more difficult to maintain and becomes more error-prone. Additionally it is also harder to test a heavily coupled class in isolation. The class becomes so ambiguous that it is quite difficult to understand it. Therefore the number of dependencies should be kept at a

IJARCCE

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
ISO 3297:2007 Certified
Vol. 5, Issue 7, July 2016

minimum. They further refined this definition by saying that CBO for a class is a count of the number of other classes to which it is coupled.

Response for class (RFC)
The response set (RS) of a class is a set of methods that can potentially be executed in response to a message received by an object of that class. RFC is used to measure the number of different methods that can be executed when an object of that class receives a message (when a method is invoked for that object). There are some other important metrics discussed in this direction which measure the degree of coupling among different classes and hence are useful to determine the complexity of the software product. Message Passing Metrics (Li and Henry) recognizes a number of metrics that can predict the maintainability of a design. There are two measures, message passing coupling (MPC) and data abstraction coupling (DAC). Message Passing Coupling measures the numbers of messages passing among objects of the class. Data Abstraction Coupling Metric (DAC) measures the number of Abstract Data Types defined in a class. This metric is used to measure the number of instantiations of other classes within the given class. Also the Afferent and Efferent Coupling Metric is given by Martin. Afferent coupling is harder to determine and much more valuable. It measures how many other classes use the current class. Efferent coupling determines how many number of classes the current class references. It is easy to find out via simple inspection: open the class in question and count the references (in fields and parameters) to other classes.

V. SOFTWARE COMPLEXITY

Software complexity, deals with how difficult a program is to comprehend and work with [21]. Software maintainability [21], is the degree to which characteristics that hamper software maintenance are present and determined by software complexity. Software complexity is based on well-known software metrics.
Various software complexity metrics invented and can be categorized into two types:

1) Static metrics
Static metrics are obtainable at the early phases of software development life cycle (SDLC). These metrics deals with the structural feature of the software system and easy to gather. Static complexity metrics estimate the amount of effort needed to develop, and maintain the code.

2) Dynamic metrics
Dynamic metrics are accessible at the late stage of the software development life cycle (SDLC). These metrics capture the dynamic behavior of the system and very hard to obtain and obtained from traces of code.
Software Complexity Measures: Attributes
Software complexity metrics can be distinguished by the attributes used for measurement. In this paper, we are concentrating on static measure which can be classified into three types:

1) Size based metrics
Size is one of the most essential attributes of software systems. It controls the expenditure incurred for the systems both in man-power and budget, for the development and maintenance. These metrics specify the complexity of software by size attributes and helps in predicting the cost involvement for maintaining the system. Size based metrics measures the actual size of the software module. Metrics is originated from the basic counts such as line numbers, volume, size, effort, length, etc.

2) Control flow based metrics
Control flow based metrics measures the comprehensibility of control structures. These metrics also confine the relation between the logic structures in program with its program complexity. These metrics are originated from the control structure of a program [21].

3) Data flow based metrics
Data flow based metrics measure the usage of data and their data dependency (visibility of data as well as their interactions) [21].Structural testing criteria consider on the knowledge of the internal structure of the program implementation to derive the testing criteria. Test cases are generated for actual implementation, if there is some change in implementation then it leads to change in test cases. They can be classified as, complexity, control flow and data flow based criteria. The complexity based criterion requires the execution of all independent paths of the program; it is based on McCabe’s complexity concept. For the control flow based criteria, testing requirements are based on the Control Flow Graph (CFG). It requires the execution of components (blocks) of the program under test in condition of subsequent elements of the CFG i.e. nodes, edges and paths. Another method is number of unit tests needed to test every combination of paths in a method. In Data Flow based criteria, both data flow and control flow information are used to perform testing requirements. These coverage criteria are based on code coverage. Code coverage is the degree to which source code of a program has been tested. Test coverage is measured during test execution. Once such a criterion has been selected, test data must be selected to fulfill the criterion.

Complexity of software is measuring of software code quality; it requires a model to convert internal quality attributes to code reliability. High degree of complexity in a component like function, subroutine, object, class etc. is consider bad in comparison to a low degree of complexity in a component. Software complexity measures which enables the tester to counts the acyclic execution paths through a component and improve software code quality. In a program characteristic that is one of the responsible factors that affect the developer’s productivity [8] in program comprehension, maintenance, and testing phase. There are several methods to calculate complexity measures were investigated, e.g., Nesting Level, different version of LOC, NPATH , McCabe’s cyclomatic number

,Data quality, Halstead's software science, Function Points, Token Counts, Chung's live definition etc.

V. CONCLUSION

This paper focuses on two very significant factors of complexity measurement of software which are coupling and cohesion. An extensive study of approximately all types of coupling and cohesion metrics has been reported in this paper. The major categories are static coupling and cohesion metrics and dynamic coupling and cohesion metrics. From this in-depth study we find that the static metrics are comparatively easy and simpler to collect because there is no need to execute the software. Static metrics can be obtained at very early stages of program development this is the reason these metrics are widely available. Static metrics are satisfactory to measure the quantity attributes such as size and complexity but as far as quality attributes such as reliability and testability are concerned, use of static metrics are not accurate because static metrics are evaluated only by means of static inspection of the software artifact. Dynamic metrics are calculated on the basis of the data collected during actual execution of the system, and thus reinforce the quality attributes explicitly such as chances of fault occurrences, performance. Thus keeping in view the above limitations of static metrics we see that the dynamic metrics are more precise to use for complexity measurements. However the computation process of dynamic metrics is difficult in comparison to the static once. Also very little work has been done in areas of dynamic coupling and cohesion metrics and need further more investigations. So we can conclude that to avoid computational efforts and also for qualitative measurements, a hybrid approach of static and dynamic metrics can proved to be beneficial one.

REFERENCES

[1] Ashutosh Mishra,vinayak srivastva (2012) "Conceptual and Semantic Measures for Cohesion in Software Maintenance "International Journal of Computer Applications (0975 – 8887)Volume 47– No.22, June 2012
[2] Rupinder.S and Hardeep.S(2010),"On Formal Models and Deriving Metrics for Service-Oriented Architecture", JOURNAL OF SOFTWARE, VOL. 5, NO. 8, AUGUST 2010
[3] Varsha Mishra(2013), "Better Object Oriented Paradigm Inheritance and Interface through Cohesion Metrics", International Journal of Computer Applications (0975 – 8887) Volume 66– No.21, March 2013
[4] Varun Gupta(2010), "Dynamic cohesion measures for object-oriented software", Journal of Systems Architecture 57 (2011) 452–462
[5] Himanshu Dua and Puneet Jai.K(2014), "Dynamic cohesion metrics for aspect oriented programming" , International Journal of Emerging Technologies in Computational and Applied Sciences (IJETCAS).
[6] R. Martin. OO design quality metrics: An analysis of dependencies. In Proceedings Workshop on Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, 1994.
[7] W. Li and S. Henry. Object-oriented metrics that predict maintainability. The Journal of Systems and Software, 23(2):111-122, 1993.
[8] L.C. Briand, J.W. Daly, and J.K. Wust. A unifed framework for cohesion measurement in object-oriented systems. Empirical Software Engineering: An International Journal, 3(1):65-117, 1998.
[9] L.C. Briand, S. Morasca, and V. Basili. Measuring and assessing maintainability at the end of high-level design. In International Conference on Software Maintenance, pages 88-97, Montreal, Canada, 1993.
[10] L.C. Briand, S. Morasca, and V. Basili. Defining and validating high-level design metrics. Technical Report CS-TR 3301, Department of Computer Science, University of Maryland, College Park, MD 20742, USA, 1994.
[11] L.C. Briand, J.W. Daly, and J.K. Wust. A unified framework for cohesion measurement in object-oriented systems. Empirical Software Engineering: An International Journal, 3(1):65-117, 1998.
[12] E. Arisholm, L.C. Briand, and A. Foyen. Dynamic coupling measures for object- oriented software. IEEE Transactions on Software Engineering, 30(8):491–506, 2004.
[13] Bieman J M, Ott L M. Measuring functional cohesion. IEEE Transactions on Software Engineering, 1994, 20(8): 644-657.
[14] Ott L M, Bieman J M, Kang B K. Developing measures of class cohesion for object oriented software. In Proc. The 7th Annual Oregon Workshop on Software Metrics, Oregon, USA, 1995.
[15] Gupta N, Rao P. Program execution based module cohesion measurement. In Proc. the 16th International Conference on Automated on Software Engineering (ASE 2001), San Diego, USA, Nov. 26-29, 2001, pp.144-153.
[16] Yacoub S, Ammar H, Robinson T. A methodology for architectural-level risk assessment using dynamic metrics. In Proc. 11th Int. Symp. Software Reliability Eng, San Jose, Oct. 8-10, 2000, pp.210-221.
[17] W.P. Stevens, G.J. Myers, and L. L. Constantine. Structured design. IBM Systems Journal, 13(2):115-139, 1974.
[18] Marcela Genero, Mario Piattini and Coral Calero, "A Survey of Metrics for UML Class Diagrams", Object Technology Journal, Vol. 4, No. 9, Nov-Dec 2005.
[19] R. Martin. OO design quality metrics: An analysis of dependencies. In Proceedings Workshop on Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, 1994
[20] L.C. Briand, J.W. Daly, and J.K. Wust. A unifed framework for coupling measurement in object-oriented systems. IEEE Transactions on Software Engineering, 25(1):91-121, Jan/Feb 1999.
[21] W. Harrison, K. Magel, R. Kluczny, and A. Dekok, Applying Software Complexity Metrics to Program Maintenance Compute, vol. 15, pp. 65-79, 1982