Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X
IMPACT FACTOR: 5.258

IJCSMC, Vol. 5, Issue. 7, July 2016, pg.417 – 424

A Review On Cohesion Metrics In Service Oriented Architecture

Simardeep Kaur

M.Tech, Dept of Information Technology , Adesh Institute of Engineering and Technology, Faridkot, India

E-mail Address: simardeep2789@gmail.com

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. Cohesion metrics play an important role in empirical software engineering research as well as in industrial measurement programs. The Cohesion 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. This paper addresses the Cohesion of service operations for service-oriented systems from the perspective of a service provider.

Keywords- SOA, Cohesion, 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.

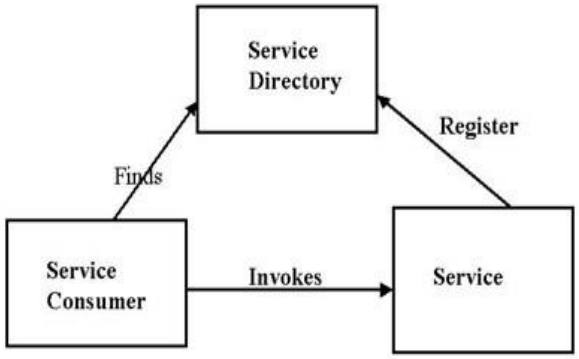

Fig 1 SOA

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.

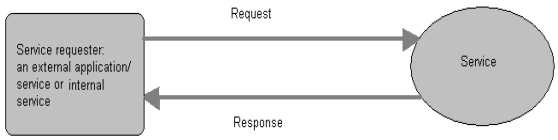

Fig 2 Static Binding

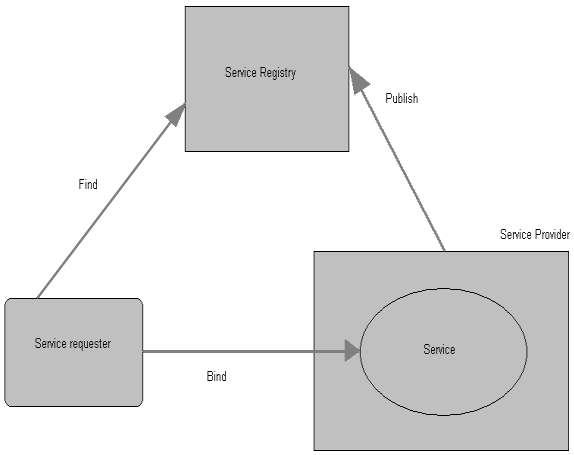

Fig 3 Dynamic Binding

***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 4

- **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

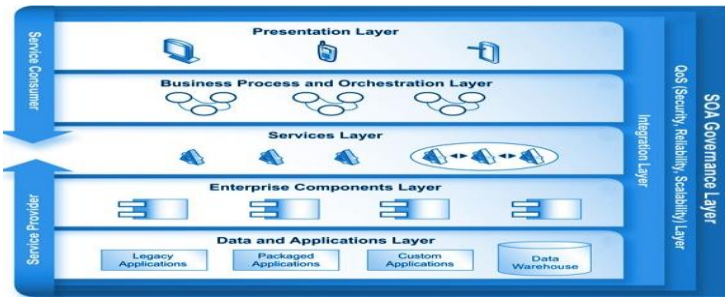

Fig 4 SOA layers

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 5

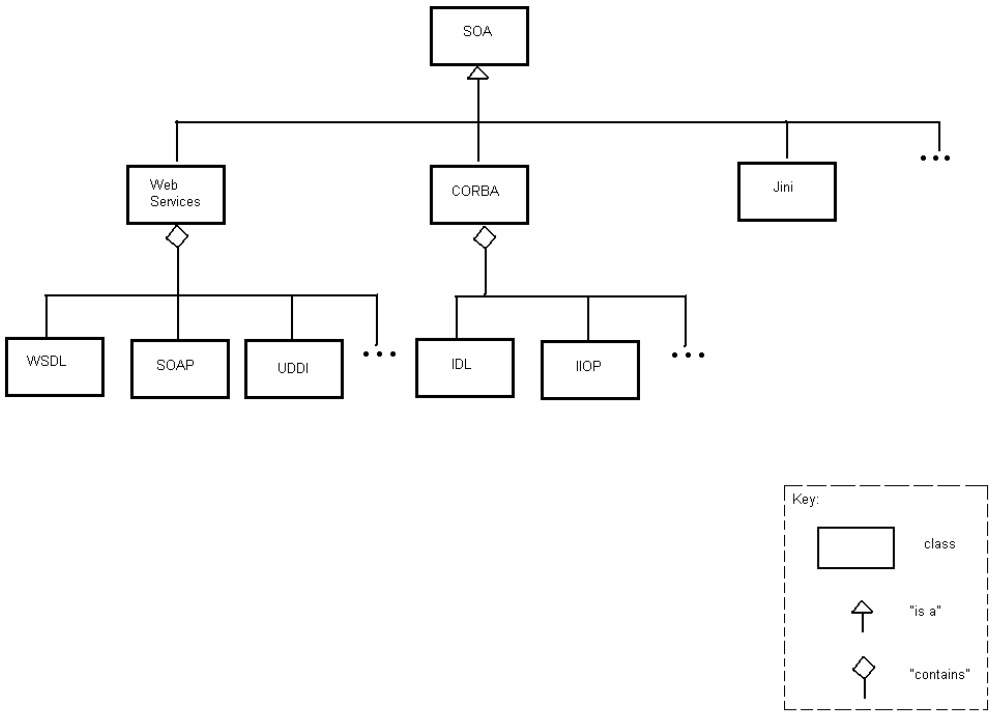

Fig 5 SOA Web Service

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, and 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

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.

IV. COHESION 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}.

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

$P = \{(I_i, I_j) \mid I_i \cap I_j = \phi\}$ and $Q = \{(I_i, I_j) \mid I_i \cap I_j \neq \phi\}$

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 M_1,M_2 and M_3. Let $\{I_1\} = \{p,q,r,s,t\}$ and $\{I_2\} = \{p,q,t\}$ and $\{I_3\} = \{a,b,c\}$. $\{I_1\} \cap \{I_2\}$ is non-empty, but $\{I_1\} \cap \{I_3\}$ and $\{I_2\} \cap \{I_3\}$ 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 are proposed. 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

[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.

