Towards an Analytical Approach to Measure Modularity in Software Architecture Design

Morteza Ghasemi[1*], Sayed Mehran Sharafi[1], Ala Arman[2]

[1]Department of Computer Engineering, Islamic Azad University, Najaf-Abad Branch, Esfahan, Iran
[2]Università degli Studi di Milano, Crema, Italy

[*] Corresponding author. Email: morteza.ghasemi@sco.iaun.ac.ir.
Manuscript submitted July 15, 2014; accepted January 20, 2015.
doi: 10.17706/jsw.10.4.465-479

Abstract: Modularization is one of the important subjects in the software design area which leads to increasing the level of quality attributes such as maintainability, portability, reusability, interoperability and flexibility. Therefore, measuring the modularity of a designed architecture is a vital issue to obtain software with a high quality level. Moreover, low coupling between modules, high cohesion of a fine-grained module is two major criteria that could lead to more advanced standard design. In this paper, we introduce an analytical method to calculate modularity considering coupling, granularity and cohesion. To assess the comprehensiveness of the proposed method, the degree of modularity is calculated in a case study using two different architectural designs which shows the architecture's desired quality characteristics in designing the software. The assessment implies that our approach offers a holistic, flexible method considering the type of software application.

Key words: Cohesion, Coupling, Granularity, Software Architecture, Modularity

1. Introduction

Software architecture plays a significant role in constructing software systems. It is the structure or structures of a system that comprises software components and their externally visible properties as well as relations among them. It enables software designers to evaluate the design effectiveness in meeting its stated requirements. Besides, one of the most important factors in gaining a high level architecture standard is modularity that makes a software system intellectually manageable. A large program which consists of a single module, which is called monolithic software, is not easily understandable. Because the number of control paths, span references, the number of variables make complicated software that is almost impossible to grasp. Therefore, it is necessary in almost all instances to break the designed software into some modules, hoping to make it more understandable and decrease the cost of its building as a result [1].

However, there are some other causes that make the concept of modularity more convoluted. By referring to Fig. 1, the cost of developing a single module reduces when the total number of modules increases. Besides, as the number of modules increases, the cost of integrating them rises. Thus, the total cost of developing typical software's curve is like the one shown in Figure1. As can be seen, there is a number M, of modules that would cause the minimum development expenditure. To have software with minimum cost, the number of modules should be within M area. Therefore modularization is an inevitable task in the software structure design.

One of the main items that have an effect on modularity is called coupling which is a qualitative measure of the degree to which modules are connected to one another. It increases as modules become more interdependent. It has been proved that higher coupling leads to greater fault proneness; because, dependencies within the code result in regression faults. Also, if a module has a high faultiness potential, it will have to go through repetitive maintenance. In addition, it is common to modify several modules to fix a single bug. Thus, when a module is fault proneness, it can affect the maintainability of some other modules, negatively [2]. Hence, considering low coupling is an essential subject in the module design. Another vital aspect in the software modularization is cohesion. According to [1], a cohesive module has a small, convergent set of responsibilities and single- mindedly applies methods and attributes to implement those responsibilities. A module which has to fulfill too many small tasks is not useful to other modules using it and produces minimal value consequently. Moreover, a module with too many tasks decreases the possibility of reusability as it provides more desire than other modules.

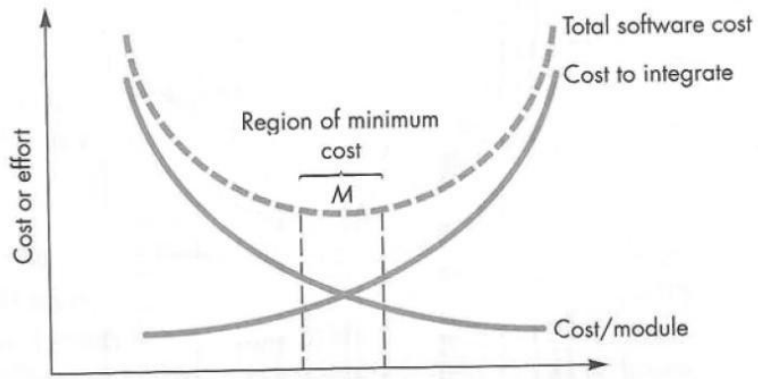

Fig. 1. The relation between the number of modules and developing a module, integration and total] software cost [1].

The last main influence on modularity that we considered in our proposed method is granularity. Module granularity is a metric to assess the module size or complexity. As the granularity size of a module increases, the probability of interchangeability decreases. On the other hand, as the granularity size of a module decreases, the maintenance of the system gets more complex. Therefore, granularity topic is a fuzzy topic in software modularity.

Low coupling between modules and high cohesion of a fine-grained module are two desirable properties in the modular architecture design. Therefore, software engineers attempt to measure the degree of modularity, i.e., the relative decomposition of a complex system into cohesive and loosely coupled parts which are fine-grained comes to attention.

In this paper, we propose a comprehensive technique to measure the degree of modularity by taking into consideration of low coupling and high cohesion as well as the module's fine-granularity. Our technique is a customized version of the software quality measurement method which is interpreted in [3] and [4].

In [3], a methodology has been established to evaluate software architecture on the architectural level. This methodology was used in [4] to assess software modularity which was considered as a quality characteristic based on some quality attributes like maintainability and reusability. These approaches will be elaborated in the next section.

The rest of this paper is organized as follows. Section 2 discusses about some related works about our problem. In Section 3, we present our proposed method to measure modularity. Section 4, describes a case study that adopts our technique to calculate modularity. Section 5, discusses about the method advantage and its comparison with other similar method. Finally, we present conclusions and future works in Section 6.

2. Related Works

There has been much research conducted on measuring modularity considering coupling and cohesion which some of them have been offered in [5]–[8]. Most of them suggest several metrics to assess modularity. For example, in [9], a concern-driven measurement framework has been designed to quantify modularity. In this work, some broadly-scoped issues such as exception handling and persistence have been taken into account to find out modularity metrics like the number of interfaces and the number of operations. In [6], a set of metrics has been introduced to measure coupling based on the actual inheritance and other forms of coupling such as HMPI (the set of Hidden Method-Method Procedure Interactions) and VMPI (Visible Method-Method Procedure Interactions). In [8], an intelligent software tool has been characterized to provide heuristic modularization advice to improve the current code. In this work, a unique aspect is measuring design coupling rather than data coupling. Design coupling means that there are too many design links between modules, i.e., modules or classes which share similar assumptions about a design fact such as format and interpretation of a table. As a result, if one of the modules or classes ever needs to be rewritten, the other ones should be checked if they require correlative changes. However, according to [1], data coupling occurs when operations pass long strings of data arguments. Therefore, the bandwidth of communication between classes and components increase consequently. Also, the testing and maintenance of modules get more difficult. There have been other methods to measure modularity, like QESDRP, Panas et al. [10] and PSAEUM, Tvedt et al. [11], but the majority of them focus on the post-architectural design steps of software development processes with artifacts such as source code and real use cases as has been discussed in [3].

3. Proposed Method

It has been discussed in the section I that modularity is one of the most controversial subjects in the software architecture design and a key concept to manage complexity and dependencies. We have designed and examined a holistic approach to measuring modularity in the software architecture. In this section we are going to discuss our proposed approach to measure modularity. Our methodology has been inspired from an approach which has been introduced in [3]. Therefore, first, this method that we call it Deng and Mercado's method and then our methodology will be expressed.

3.1. Deng and Mercado's Method

Deng and Mercado [3], describe their quality model based on a quality notation that has been shown in Fig. 2.

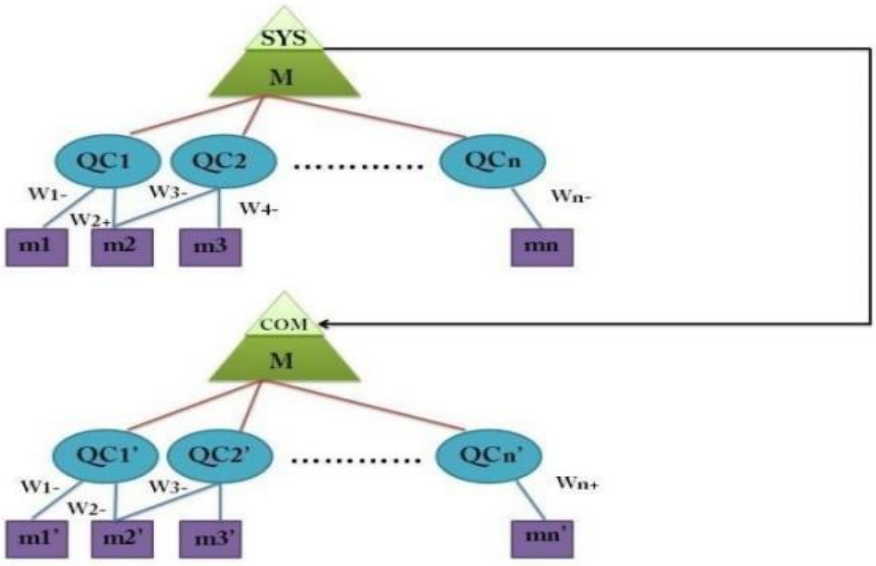

Fig. 2. Deng and mercado's quality notation [4].

As can be seen in Fig. 2, two levels of quality attributes (QAs) have been considered, one for component level and one for system level. They have been illustrated by triangles and are denoted by *com* and *sys* respectively. Each quality attribute has a set of quality characteristics (QCs) which are denoted by QC_n and have some related metrics denoted by m_n. To each metric a weight is allocated denoted by w_n+ and w_n- indicating their impact on its related quality attribute. Deng and Mercado's mathematical method has three steps [3]:

- Integrating both system and component level metrics to their corresponding QCs.
- Integrating component QCs to component QA.
- Integrating system QC and component QA to system QA.

In the first step, the value of each of metric is normalized, which is shown by d [3].

$$x = \frac{d - d_{min}}{d_{max} - d_{min}} \; if \; d! = \; d_{min} \; \wedge \; d_{\max} \tag{1}$$

$$x = 0 \;\; if \; d = d_{min} \;\; AND \;\; x = 1 \; if \; d = d_{max}$$

In equation (1), x is the normalized value of each metric. Then all quality characteristics are calculated considering the metric values. Equation (2) is used for this calculation. Some of the metrics have direct impact and some of them have an inverse impact on their related quality characteristic. Each quality characteristic has direct metrics and inverse metrics. As it has been noted, some weights are given to each metric. The sum of these weight values is equal to 1. If a metric has a reverse relationship with a quality characteristic, its value is subtracted from 1 [3].

$$QC = m_1 W_1 + \; m_2 W_2 + \cdots + \; m_i W_i + \; (1 - m_1^{'}) W_1^{'} + \cdots + \; \left(1 - m_j^{'}\right) W_j^{'} \tag{2}$$

$$\sum_{k=1}^{i} W_k + \sum_{k=1}^{j} W_k^{'} = 1 \tag{3}$$

In the next step, quality attribute value for all components is calculated. Equations (4) and (5) are used for this calculation. For example, we are going to calculate maintainability as a quality attribute. Moreover, there are some quality characteristics for this quality attribute like modularity and testability. Each of these quality characteristics has some values which are shown by QC_{com} that was measured in (2). First the value of quality attribute for each component is measured using (4), and then the quality attribute's value for all components is calculated using (5). The value of QA_{com} corresponds to the degree of how the components meet the requirements of that quality attribute [10].

$$QA_{comp} = \sum_{k=1}^{|QC_{comp}|} QC_{com} W_k \tag{4}$$

$$QA_{complevel} = \sum_{k=1}^{|comp|} \tag{5}$$

$$\sum_{i=1}^{|QC_{comp}|} Wi = 1 \tag{6}$$

Finally, the value of quality attribute at the system level is measured. Equation (7) is used for this calculation [3].

$$QA_{Sys=} \sum_{i=1}^{|QC_{sys}|} QC_{sysi} W_i + QA_{complevel} * W_{|QC_{sys}|+1} \quad (7)$$

To give an example, consider maintainability as a quality attribute, all components have their own maintainability value. All of these values are multiplied by a weight which shows each component importance in the system's maintainability.

3.2. Customization of Deng and Mercado's Method to Measure Modularity

It has been discussed that coupling and cohesion are two main important criteria for modularity. To maintain a high level standard on modularity, high cohesion and low coupling between them should be considered as high priorities. In [12], several factors have been mentioned having the most effects on the cohesion of components and coupling between them. Some of the most important of these factors are as follows:

- **Degree of Separation of Concerns (SoC)**: A large problem is easier to solve if it is divided into some set of sub problems or concern. Technically, each concern gives a definite functionality that can be developed or verified independently of other concerns [1]. Model-View-Controller (MVC) is a design pattern which is an excellent example of separating these concerns for better software maintainability.
- **Degree of Information Hiding and Encapsulation:** Modules should be designed so that **information** (data and algorithms) within a module is inaccessible to other modules that have no need for such information [1]. It gives a level of privacy for a module and its environment.
- **Degree of Concurrency**: Running a module on a single or multiple processors may influence on influence cohesion. A module which is running across multiple CPUs and address spaces is more **complex** and less cohesive in comparison with those that are running on a single process and having a single address space.
- **Synchronicity of Connector:** If two or more components are synchronized, it may influence coupling. Because it is essential to coordinate between synchronized modules. For example, a synchronization mechanism called semaphore restricts the access of multiple processes which want to use a shared resource. Another mechanism which is called "Time Stamp Ordering", allocates a timestamp (t_s) to processes when it starts. Two processes Pi is executed before P_j if $t_s (P_i) < {}_{ts} (P_j)$.
- **The Cardinality of the Interface**: This factor represents a number which represents the number of communication points with other modules. It expresses the degree of complexity and dependency between modules. In this area, there are metrics such as Fan-in, which is the number of other modules that refer to a class and Fan-out which is the number of modules that referenced by a module.
- **Mode of Connection:** There are three types of connection: point-to-point, multicast, and Broadcast. The **degree** of coupling (complexity) in the point-to-point (1 to 1) connections is less than multicast connections (1 to m). Likewise, the broadcast connections are more complex than the two previous cases.
- **Type of Dependency:** There are several dependencies between modules which relate to a change. For example, a shared memory is a dependency; because if there is a change for any shared memory, it may **cause** changes in all programs or knowledge sources. The changes can be at runtime, compile time and source code.
- **Type of Access Control:** This factor, which explains modules' competition to access resources has a direct impact on coupling. A queue can be a good example of access control. When a queue is full,

producer modules are blocked to access the queue. When a queue is empty, Consumer modules are blocked to access the queue.

- **Dependency Direction**: This factor shows the direction of change dependency between components. In [12], three types of change dependencies have been mentioned: Unary, Binary, n-ary. "Unary dependency direction is a dependency change that may require a change in a behavior of one **component** affects the behavior of another component." Binary is a bidirectional dependency change. For instance, any change in a server causes in a change in a client and the other way around. In n-ary dependency change, a change in a component named *A*, results in changes in *n* other related components to *A*.

Although modularity is not a quality attribute and Deng and Mercado's method is an approach to measure quality attributes based on calculating quality characteristics, it is possible to use it to calculate modularity. Therefore, to improve Deng and Mercado's method to measure modularity, first, modularity is considered as a quality attribute and its related important factors, i.e., coupling, cohesion and granularity are seen as Quality characteristic and their correlated metrics are calculated. In other words, taking into consideration of Deng and Mercado's method, modularity is substituted as a quality attribute. Also, coupling, cohesion and granularity are substituted as quality characteristics. It should be noted that there could be more than one metric for coupling, cohesion and granularity. Besides, metric values for coupling and granularity are subtracted from one as they have a negative impact on modularity.

3.2.1. Metric identification

In the first step, some metrics that affect on coupling, cohesion and granularity are specified. As discussed before, some metrics have positive and some others have a negative impact on. Table I, shows some of the most important of these metrics. It is essential to be mentioned that these metrics could be absolutely different in various projects as they completely depend on software architecture. As can be seen, these metrics have been considered some of the above mentioned factors such as the cardinality of interface, mode of connection, dependency direction and some others has not like type of access control and degree of concurrency. The selection criteria are based on the type of software application. For example, if it is a real-time application, architects should calculate the synchronization factor influence on the coupling, cohesion and granularity.

As has been shown in Table 1, the metrics have been classified based on two categories. The first one which is called Group divides the metrics based their influence on coupling or cohesion and granularity. The second category, which is called Level specifies that which metric has been defined at the system level and which of one has been defined at the component level.

As can be seen in Table I, there are 7 metrics at the component level and 2 metrics at the system level. There are two metrics in Table I named *NumCycles* and *InDepend*. They measure the number of dependency cycles of a component and the number of indirect paths from a component to other components, respectively, which are related to the dependency direction factor which was described before.

3.2.2. Modularity Measurement

To measure the modularity of typical software Architecture, first, the metrics at level component are replaced in *(2):*

$$Modularity_{com\ i} = NUCPCOM * W_1 + (1 - CBCOM) * W_2 + (1 - AC) * W_3 + (1 - EC) * W_4 + (1 - NumCycles) * W_5 + (1 - InDepned) * W_6 + (1 - NCPCOM) * W_7 \quad (8)$$

In the above equation, the value of modularity is calculated for each component. Like Deng and Mercado's method [10], some weights are assigned to each metric so that the sum of the weights is equal to 1. Besides, it is important to note that the values of coupling metrics are subtracted from 1 as they have an inverse

relationship to modularity. The weight of a metric is assigned by the software architect according to factors discussed above and how much these factors contribute to the modularity measurement. In other words, each metric's weight is calculated considering its importance in the modular design for its related component or module. This weight could be too little for some modules, even zero or more for others. In addition, based on Model McCall's method [3], modularity is a quality characteristic for some quality attributes such as maintainability, flexibility, portability, reusability, interoperability, etc. Therefore, software architect can assign a weight to each metric based on the importance of each quality attribute in each application. For example, if maintainability is important for module A more than B, more weight is assigned to maintainability in A compared to B.

Table 1. Metrics Used To Measure Modularity

Group	Level	Metrics	Description
Coupling	System	CBN [3]	The number of dependencies of a physical node to other physical nodes.
Coupling	Component	CBCOM [3]	The number of all dependencies of a component to other components.
Coupling	Component	AC[5]	The number of components that require service from the assessed component
Coupling	Component	EC [9]	The number of components from which the assessed component requires service from them.
Cohesion	Component	NUCPCOM [3]	The number of usecases that a component is involved in.
Coupling	Component	NumCycles	The number of dependency cycles of a component
Coupling	Component	InDepned	The number of indirect paths from a component to other components.
Granularity	System	NCOMPUC [3]	The number of components corresponding to a usecase
Granularity	Component	NCPCOM [3]	The number of classes per module

In the next step, by applying *(5)*, the value of modularity is calculated at the component level:

$$Modularity_{component\ level} = \left(\sum_{j=1}^{n} modularity_{com_j} W_j\right) \quad (9)$$

In (9), first some weights are assigned each component. Then the values of each component modularity which was calculated in (8) are multiplied by these weights and finally these new values are added together. The result is the modularity at the component level.

Lastly, the value of modularity for the entire system should be measured. Equation (10) shows how it is calculated:

$$Modularity_{Sys} = (1 - CBN) * + Modularity_{Sys} = (1 - CBN) * + (1 - NCOMPUC) * W_2 \quad + \quad Modularity_{component\ level} * \mathrm{W3} \quad (10)$$

In (10), the *CBN* and the *NCOMPUC* are the metrics at the system level, which both have a negative impact on the modularity. *W1* and *W2* are two related weights for the *CBN* and the *NCOMPUC* respectively, and *W3* is the corresponding weight for the modularity at the component level.

4. Case Study

In this section, to evaluate the proposed technique, the value of modularity in the architectural design of a typical Applicant Management System is measured considering two architectural design scenario. The first one is described in this section and the second one is discussed in the Appendix A and their results will be elaborated. The goal of the system includes the management of requests that people make for a business and some representative should handle these requests. To do this, first, a requesting person should register his request in the system. Next, his request is investigated by a representative. It includes searching the person and his city information using an external system called Information Management System, issuing some letter(s) to the person as well as setting his request status. Moreover, the representative might forward the request to other organizations or schedule a meeting with the person to handle the request.

The architecture of this system is based on Attribute- Driven Design (ADD) which has been reported in [13]. ADD is a method for the design of a software conceptual architecture [14] of a system or collection of systems based on an explicit articulation of the quality attributes goals. It focuses on a recursive process that decomposes a system or system elements by applying architectural strategies and patterns that satisfy quality attribute requirements [15]. Fig. 3, shows a typical AMS in deployment view which is used to measure the value of the *CBN*.

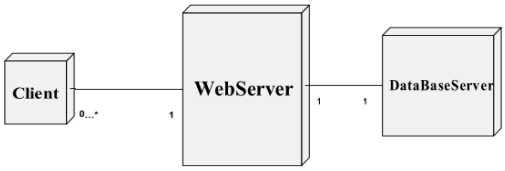

Fig. 3. The AMS in the deployment view.

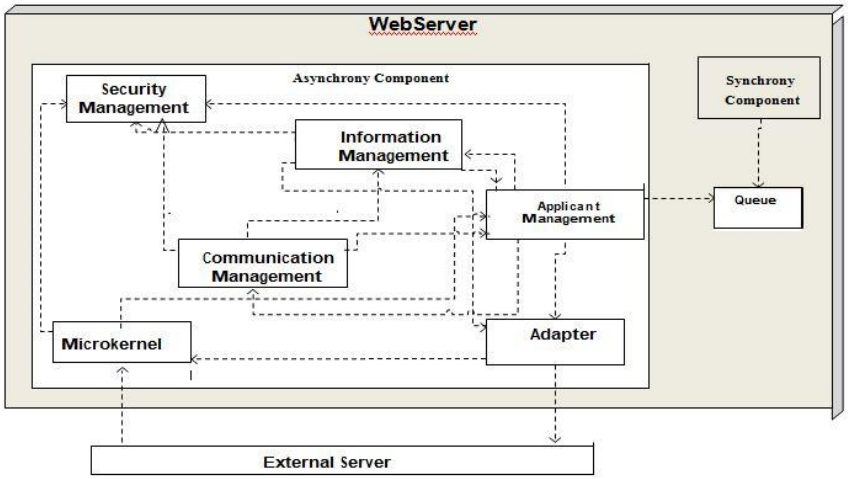

Fig. 4. The AMS in the uses view (the first scenario).

Fig. 4 presents the designed architecture for the AMS in uses view. In this view, dependency relationships between components are shown in more detail. As can be seen, there are several components that some of them have been taken from functional requirements and some others have been taken from some architectural patterns in the design of architecture. For example, components like micro kernel, Adapter and External Server are taken from Micro Kernel architectural patterns and components such as Asynchrony Component, Synchrony Component and Queue are taken from an architectural pattern called Half Synch/Half Asynch. On the other hand,

components such as Information Management, Application Management System and Communication Management show functional requirements. Application Management is the main component that uses other components to handle the request. In addition to this component, the Information management component is used to extract the general information of people from one of the systems in an External Server through Adapter and Microkernel components. The Communication Management component is used to extract their contact information. As we want to keep the reusability potential high for Communication Management and Information Management, they have not been included in the Application Management component which is the main one.

4.1. Measuring the Modularity at the Component Level

To measure modularity of the AMS system, according to our method, the modularity at the component level should be calculated initially. In order to do this, first, some weights are assigned to each metric at the component level. These weights have been presented in Table 2.

Table 2. Metric Weights at the Component Level

Metric	Weight
NUCPCOM	0.1
CBCOM	0.1
AC	0.15
EC	0.1
NumCycles	0.3
InDepned	0.25

Table 3. Metric Values at the Component Level and the Modularity of Each Component (the First Scenario)

List Of Components	*NUCPCOM*		*InDepend*		*NumCycles*		*CBCOM*		*AC*		*EC*		*Modularity*
	Q_Valu	*N_Value*	*Q_Val*	*N_Valu*	*Q*	*N_Valu*	*Q_Val*	*N_Val*	*N_Val*	*Q_Valu*	*Q_Val*	*N_Value*	
Security Management	15	0.88	0	0	0	0	4	0.43	4	0.50	0	0.00	0.87
Applicant Management System	12	0.69	5	0.63	9	1	6	0.71	3	0.38	5	0.63	0.32
Information Management System	6	0.31	6	0.75	5	0.56	4	0.43	2	0.25	3	0.38	0.46
Communication Management	2	0.06	7	0.88	4	0.44	3	0.29	1	0.13	3	0.38	0.47
Adapter	3	0.13	7	0.88	4	0.44	4	0.43	2	0.25	2	0.25	0.46
Microkernel	4	0.19	7	0.88	5	0.56	4	0.43	2	0.25	2	0.25	0.43
External Server	4	0.19	7	0.88	2	0.22	2	0.14	1	0.13	1	0.13	0.59
Synchronization	1	0	1	0.13	0	0	1	0	0	0.00	1	0.13	0.86
Queue	1	0	0	0	0	0	2	0.14	2	0.25	0	0.00	0.85

Then, for each component in uses view, their values at the component level are calculated and replaced in (8). These values have been reported in Table 2.

In Table 3, the value of each metric for each component is calculated and then by applying (1), these

values are normalized. So the values of each metric have been presented in two columns. The first column (*O_Value*) and the second column (*N_Value*) display the original values and the normalized values of the metrics respectively. After that, using (8) as well as assigning weights to the metrics, the value of modularity for each component is calculated. These values have been shown in the *Modularity* column.

Then the total modularity at the component level is measured. First, some weights are assigned to each component and then by applying (9), this value is calculated. Table 4 shows the component weights and how the modularity at the component level is measured.

Table 4. Modularity at the Component Level (the First Scenario)

Component	Modularity	Weight	Weight*Modularit
Security Management	0.87	0.05	0.04
Applicant Management System	0.32	0.4	0.13
Information Management System	0.46	0.2	0.09
Communication Management System	0.47	0.2	0.09
Adapter	0.46	0.05	0.02
Microkernel	0.43	0.1	0.02
External Server	0.59	0	0
Synchronization	0.86	0	0
Queue	0.85	0	0
The value of modularity at the component			0.42

4.2. Measuring the Modularity at the System Level

In this section the modularity of the entire system is calculated. In order to do this, first of all, the values of the *CBN* and the *NCOMPUC* are measured. As has been mentioned before, the value of the *CBN* is calculated based on the deployment view of the AMS which has been shown in Fig. 3. Table V presents the calculation of the *CBN*.

Table 5. The Value of *CBN* Metric

Node	O_Value	N_Value
Client	1	0
Web Server	2	1
Data Base Server	1	0
Average	1.33	0.33

Fig. 5, shows the usecase diagram of the AMS. It is used to calculate the value of the *NCOMPUC*. This metric calculates the number of applied components of each usecase.

There are 17 use cases for the AMS. These usecases, their values and the *NCOMPUC* calculation method have been shown in Table 6.

Finally, the modularity of the entire system is measured considering (10). Like the previous steps, some weights should be assigned to the *NCOMPUC* and the *CBN* metrics and the modularity at the component level.

$$Modularity_{Sys} = (1 - CBN) \times W_1 + (1 - NCOMPUC) \times W_2 + Modularity_{component\ level} \times W3$$

$$Modularity_{component\ level} = 0.42\,, CBN = 0.33, NCOMPUC = 0.48\,,\ \text{W1=0.1, W2=0.1, W3=0.8}$$

$$Modularity_{Sys} = (1 - 0.33) \times 0.1 + (1 - 0.48) \times 0.1 + 0.42 \times 0.8 = 0.455$$

In the proposed method, the value of modularity is varied between 0 and 1.Lower values indicate more dependency between the components and less system's modifiability as a result. In fact, our approach helps the architect to assess the modularity of the designed architecture. However, the decision of that how desirable is the modularity value depends on his/her opinion - and of course, the stakeholders requirements. For example, if we calculate the modularity considering a different uses view, the value of modularity will be equal to 0.561 (see Appendix A). Although it shows less dependency between the components, the new architecture would not

enough meet the requirements of some quality characteristics like reusability. It is possible that the reusability is not important as much as the dependency in this application. Hence, this architecture design is more suitable from the architect's point of view.

Table 6. Usecase Values and the Value of NCOMPUC Metric (the First Scenario)

Usecase	O_Value	N_Value
Authentication	1	0
Define Representative Activities	1	0
Define Representative Communications	7	0.75
Manage Communications	7	0.75
Search People Information	5	0.5
Search City Information	5	0.5
Send Request	6	0.63
Define Corporate personhood and Natural	2	0.13
Mange Representative Information	2	0.13
Follow-Up Request	7	0.75
Define Users	1	0
Assign Work Groups to Users	1	0
Issue Letters (to the Requesting Person)	4	0.38
Forward Request	7	0.75
Investigate Request	7	0.75
Set Request Status	9	1
Schedule Meeting	7	0.75
Average	4.56	0.48

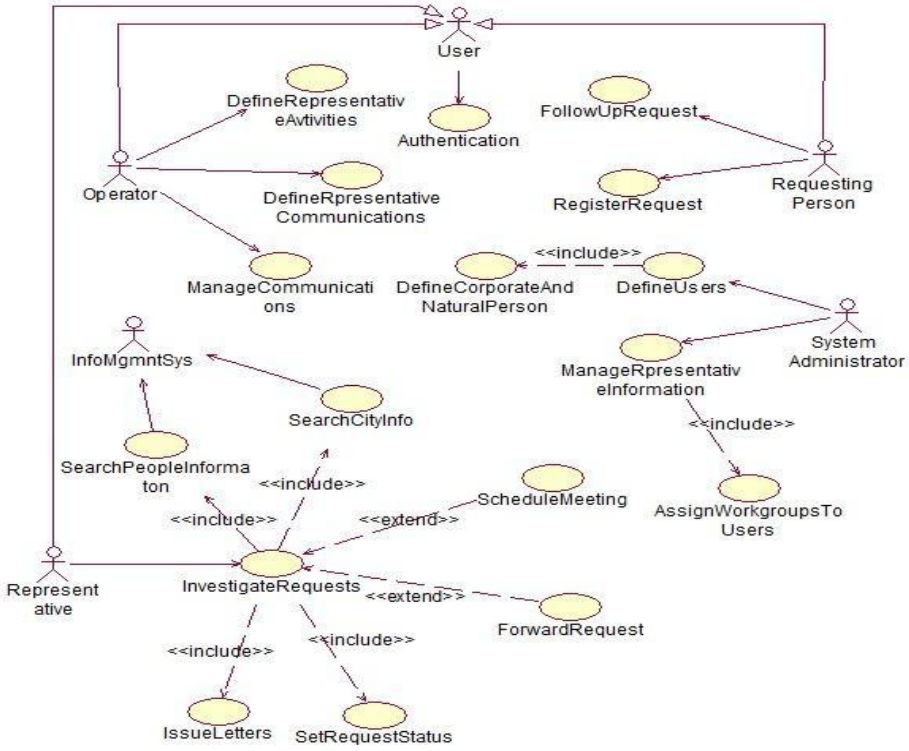

Fig. 5. Use Case View of AMS.

Therefore, the desirability of the level of the modularity measured is completely up to the architect.

5. Discussion

In [4], quality attributes such as maintainability and reuse has been used to measure modularity. In other words, first, these attributes are calculated at the component level and then at the system level. In the next step, based on the mentioned method, they are added together and the value of modularity is measured. In this

method, metrics at the component and system levels have been categorized in two maintainability and usability groups and the value of these quality attributes has been calculated considering these metrics. Therefore, the drawback of this method is that software architects cannot include other metrics like the type of access control or the synchronicity of connector to measure modularity.

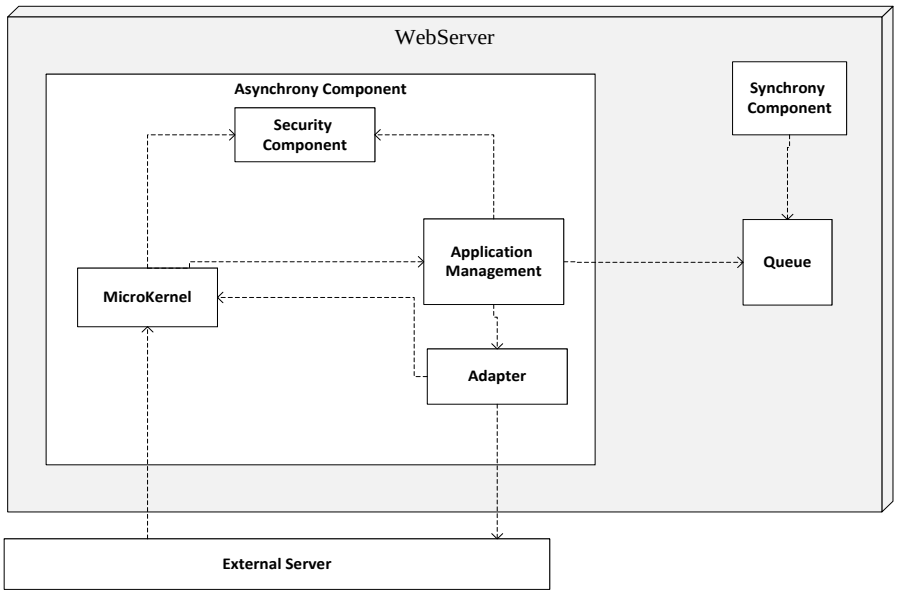

Fig. 6. The AMS in the uses view (the second scenario).

However, in our proposed method, the architect can use other metrics than ours to calculate his/her desired factors. The standard of factors' selection could be different. For example, if the architect is going to measure the modularity in a real time system, s/he should choose metrics that calculate coupling and cohesion based on the synchronization factor which was discussed before. As a result, the goal of the proposed method is to offer a holistic way to measure modularity in various software application types, so that enables the architect to consider his/hers preferred factors considering the type of application.

6. Conclusion and Future Works

In this paper, a comprehensive analytical approach was proposed to measure modularity considering coupling, cohesion and granularity as well as some quality attributes such as maintainability, portability, reusability, interoperability and flexibility. We assessed our approach using a case study considering two architectural designs which represented the architecture's desired quality characteristics in designing the application. The assessment implied (according to section V) that the proposed method offers a flexible, holistic approach as architects can customize it with their own chosen metrics.

In our future work, we intend to study some more factors that influence on modularity, such as type of dependency, mode of connection, etc. in more detail to improve the measurement of modularity with higher accuracy. Moreover, there are some other items like security, performance, deployment, maintenance, etc. that should be considered in measuring modularity. Therefore, another future work could be working on these factors and their influence on modularity.

Appendix A

In this section, we are going to measure modularity of the AMS considering a new uses view. As can be seen in Fig. 6, in the new architecture design, we merged the Information management, the Application Management, and the Communication Management components into one component called Application Management System.

Table 7, presents the metric values at the component level and the modularity of each component.

Table 7. Metric Values at the Component Level and the Modularity of Each Component (the Second Scenario)

List Of Components	NUCPCOM		InDepend		NumCycles		CBCOM		AC		EC		Modularity
	O_Value	N_Value	O_Value	N_Value	O _Value	N_Value	O_Value	N_Value	N_Value	O_Value	O_Value	N_Value	
Security Management	15	0.88	0	0	0	0	2	0.2	0.33	2	0	0	0.92
Applicant Management System	15	0.88	5	0.83	2	0.33	4	0.6	1	0.17	3	0.5	0.54
Adapter	2	0.06	5	0.83	2	0.33	3	0.4	1	0.17	2	0.33	0.50
Microkernel	3	0.13	5	0.83	2	0.33	4	0.6	2	0.33	2	0.33	0.46
External Server	4	0.19	5	0.83	1	0.17	2	0.2	1	0.17	1	0.13	0.60
Synchronization	1	0	1	0.17	0	0	1	0	0	0.00	1	0.13	0.84
Queue	1	0	0	0	0	0	1	0	2	0.33	0	0	0.85

Table 8 shows the component weights and how the modularity at the component level is measured.

Table 8. Modularity at the Component Level (the Second Scenario)

Component	Modularity	Weight	Weight*Modularity
Security Management Component	0.92	0.05	0.05
Applicant Management System	0.54	0.08	0.44
Adapter	0.5	0.05	0.02
Microkernel	0.46	0.1	0.05
External Server	0.60	0	0
Synchronization	0.84	0	0
Queue	0.85	0	0
The value of modularity at the component level			0.55

The value of *CBN* (i.e., the number of dependencies of a physical node to other physical nodes.) is not changed. But the value of the NCOMPUC (i.e., the number of components corresponding to a usecase) is changed. Table IX, shows the calculation of the NCOMPUC.

Table 9. Usecase Values and the Value of the NCOMPUC Metric (the Second Scenario)

Use Case	O_Value	N_Value
Authentication	1	0
Define Representative Activities	2	0.16
Define Representative Communications	5	0.66
Manage Communications	5	0.66
Search People Information	5	0.66
Search City Information	5	0.66
Send Request	5	0.66
Define Corporate personhood and Natural Person	2	0.16
Mange Representative Information	2	0.16
Follow-Up Request	5	0.66
Define Users	1	0
Assign Work Groups to Users	1	0
Issue Letters (to the Requesting Person)	5	0.66
Forward Request	3	0.33
Investigate Request	5	0.66
Set Request Status	7	1
Schedule Meeting	5	0.66
Average	3.76	0.46

Considering the modularity at the component level, the values of the *CBN* and the *NCOMPUC*, and their weights, the modularity at the system level is measured.

$$Modularity_{component\ level} = \sum_{j=1}^{n} modularity_{com\,j} W_j = 0.55\ ,$$

$$CBN = 0.33, \qquad NCOMPUC = 0.46\ ,$$

$$W1=0.1,\ W2=0.05,\ W3=0.8$$

$$Modularity_{Sys} = (1 - 0.33) \times 0.1 + (1 - 0.46) \times 0.1 + 0.55 \times 0.8 = 0.561$$

As can be seen, the modularity has been increased around 10%, which indicates that this architecture offers less dependency among components and more system's modifiability compared to the first scenario. From now on, it is architecture's responsibility to decide which architecture is more desirable to be applied based on the system's specification. If s/he believes that dependency is more important in the application, s/he might choose the architecture of the first scenario and if she believes that modifiability is more important in the application, s/he might choose the architecture of the second scenario.

References

[1] Pressman, R., & Maxim, B. (2014). *Software Engineering: A Practitioner's Approach*. McGraw-Hill Education.

[2] Schach, S. R. (2005). *Object-oriented and Classical Software Engineering*. 6th ed. McGraw-Hill Higher Education.

[3] Deng, C., & Mercado, H. (2008). A method for metric-based architecture level quality evaluation.

[4] Johansson, H. H. P. (2010). On the Modularity of a System.

[5] Abdeen, H., Ducasse, S., & Sahraoui, H. A. (2011). Modularization metrics: Assessing package organization in legacy large object-oriented software. *World Council for Renewable Energy*, 394–398.

[6] English, M., Buckley, J., & Cahill, T. (2007). Fine-grained software metrics in practice. *Empirical Software Engineering and Measurement*, 295–304.

[7] Oliveira, M. F. S., Redin, R. M., Carro, L., Lamb, L. D. C., & Wagner, F. R. (2008). Software quality metrics and their impact on embedded software. *Proceedings of the 2008 5th International Workshop on Model-based Methodologies for Pervasive and Embedded Software* (pp. 68–77).

[8] Schwanke, R. W. (1991). An intelligent tool for re-engineering software modularity. *Proceedings of the 13th International Conference on Software Engineering* (pp. 83–92).

[9] Sant'Anna, C., Figueiredo, E., Garcia, A., & Lucena, C. (2007). On the modularity assessment of software architectures: Do my architectural concerns count?. *Aspects in Architectural Description (AARCH).* 1–4.

[10] Panas, T., Lincke, R., Lundberg, J., & Löwe, W. (2005). A qualitative evaluation of a software development and re-engineering project, 66–75.

[11] Tvedt, R. T., Lindvall, M., & Costa, P. (2002). A process for software architecture evaluation using metrics. *Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop* (pp. 191–194).

[12] Reza, H., Carver, J., & Grant, E. (2008). Assessing the complexity of software architecture using coupling and cohesion. *Software Engineering Research and Practice*, 150–156.

[13] Bachmann, F., & Bass, L. (2001). Introduction to the attribute driven design method. *Proceedings of the 23rd International Conference on Software Engineering* (pp. 745–746).

[14] Hofmeister, C., Nord, R., & Soni, D. (2000). *Applied Software Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co.

[15] Wojcik, R., Bachmann, F., Bass, L., Clements, P. C., Merson, P., Nord, R., & Wood, W. G. (2006). Attribute-Driven Design (ADD).

M. Ghasemi received his MSc in software engineering from Islamic Azad University- Najaf-Abad Branch, Iran. His research interest is in the area of software engineering.

S. M. Sharafi is an assistant professor in the Faculty of Computer Engineering at the Najafabad branch of Islamic Azad University (IAUN) where he has been a faculty member since 2000. Dr. Sharafi completed his Ph.D. in Tehran research and science branch of Azad University in Iran and his M.Sc. and undergraduate studies at IAUN. He was the head of the Faculty of Computer Engineering Department in years between 2008 and 2015.

He was a software project manager and the head of Informatics Department at Azar Refractories Corporation in Esfahan, Iran between years 2001 and 2006. His research interests include software engineering, software architectures, and software validation and verification.

A. Arman received his MSc in software engineering of distributed systems at the KTH (Royal Institute of Technology), Stockholm, Sweden, 2012. He is now a PhD researcher at the Università degli Studi di Milano.

His research interests include cloud computing, distributed systems, software engineering and information security.