Journal of Software Engineering and Applications, **2011, 4, 244-252**
doi:10.4236/jsea.2011.44027 Published Online April 2011 (http://www.SciRP.org/journal/jsea)

Assessing Internal Software Quality Attributes of the Object-Oriented and Service-Oriented Software Development Paradigms: A Comparative Study

Yaser I. Mansour, Suleiman H. Mustafa

Department of Computer Information Systems, Yarmouk University, Irbid, Jordan.
Email: yamansour@live.com, smustafa@yu.edu.jo

Received March 16th, 2011; revised April 3rd, 2011; accepted April 6th, 2011.

ABSTRACT

Service-Oriented Architecture (SOA) is becoming the dominant approach for developing and organizing distributed enterprise-wide applications. Although the concepts of SOA have been extensively described in the literature and industry, the effects of adopting SOA on software quality are still unclear. The aim of the paper is to analyze how adopting SOA can affect software quality as opposed to the Object-Oriented (OO) paradigm and expose the differential implications of adopting both paradigms on software quality. The paper provides a brief introduction of the architectural differences between the Service-Oriented (SO) and OO paradigms and a description of internal software quality metrics used for the comparison. The effects and differences are exposed by providing a case study architected for both paradigms. The quantitative measure concluded in the paper showed that a software system developed using SOA approach provides higher reusability and lower coupling among software modules, but at the same time higher complexity than those of the OO approach. It was also found that some of the existing OO software quality metrics are inapplicable to SOA software systems. As a consequence, new metrics need to be developed specifically to SOA software systems.

Keywords: *Service-Oriented Architecture, Object-Orientation, Internal Software Quality Attributes, Software Quality Metrics*

1. Introduction

The large explosion of business demands and enterprise-wide applications has created the need for different approaches to software development in order to facilitate business collaboration and growth. This has led to the adoption of Service-Oriented Architecture (SOA) for building highly distributed and integrated enterprise-wide software systems that use web services as its building blocks. A web service can be thought of as an autonomous software component that implements specific business rules and logic to perform a particular functionality. The ability to encapsulate business rules and logic into web services that can be accessed by applications or other web services provides a high level of separation of concerns and greater opportunities of reusability.

SOA has been widely described in research and industry, but little work has been done so far to analyze the impacts associated with adopting the service-oriented approach as a paradigm to software systems development. Hence, the purpose of the study is to provide an empirical comparison and evaluation of how encapsulating business logic and rules into web services can affect internal software quality attributes. Such attributes involve size, complexity, coupling, and cohesion. Consequently, the study addresses the following three basic research questions:

1) Does a software system developed using the SOA approach requires new software quality metrics?

2) Based on the conceptual similarities of Object-Oriented and Service-Oriented approaches, how applicable are the OO metrics to SOA software systems?

3) How do Object-Oriented and Service-Orientated approaches affect internal software quality attributes of a software system?

Given above questions, a case study was developed using the Object-Oriented and Service-Oriented approaches. Then the effects of these approaches on the internal software attributes of size, complexity, coupling, and cohesion were quantitatively measured and compared using a set of eleven mature and well-established software engineering metrics. The results of the quantitative study were mapped against three informal hypotheses that were formulated prior to the measurement process to allow for rich and objective discussion. The first section of this paper introduces the SOA and the differences compared to the Object-Oriented Architecture (OOA). The second section addresses the case study by describing the internal software quality attributes and metrics used as well as the preparation of the implementation and data collection methods. In the third section, an analysis and evaluation of the measurements collected is provided. The paper concludes with a conclusion and remarks.

2. Service-Oriented Architecture and the Object-Oriented Architecture

Service-Orientated Architecture is an architectural paradigm for developing software systems based on software services. It describes how services should be defined, constructed, and orchestrated. A conceptual model of SOA consists of three main components [1,2]: a service provider component (representing the component responsible for implementing the service functionality and publishing the service description for discovery in a service registry); a service consumer component (representing the client that requests and discovers the service from the service registry and binds and invokes the service); and a service registry component, also known as Service Broker [3] (representing the component which maintains service descriptions published by service providers for discovery by service consumers).

In SOA, a collection of interacting services exists either internally or externally in a complete autonomy. A service, as depicted in **Figure 1**, encapsulates its business functionality independently of other services within the architecture and thus provides a high level of separation of concerns among services. This leads to the development of loosely coupled software systems [1,4].

For the purpose of this study, Service-Oriented Architecture is defined as an *architectural paradigm for developing distributed software systems based on autonomous, reusable, interoperable, loosely coupled web services that encapsulate business logic independently and communicate via messages through standard communication protocols.*

The object-oriented paradigm realizes a software system as a set of classes and object instances that share common structure and behavior and cooperate with each other to achieve a specific function or behavior. The OO paradigm has proven to be a very powerful approach to address and conquer complex software systems through

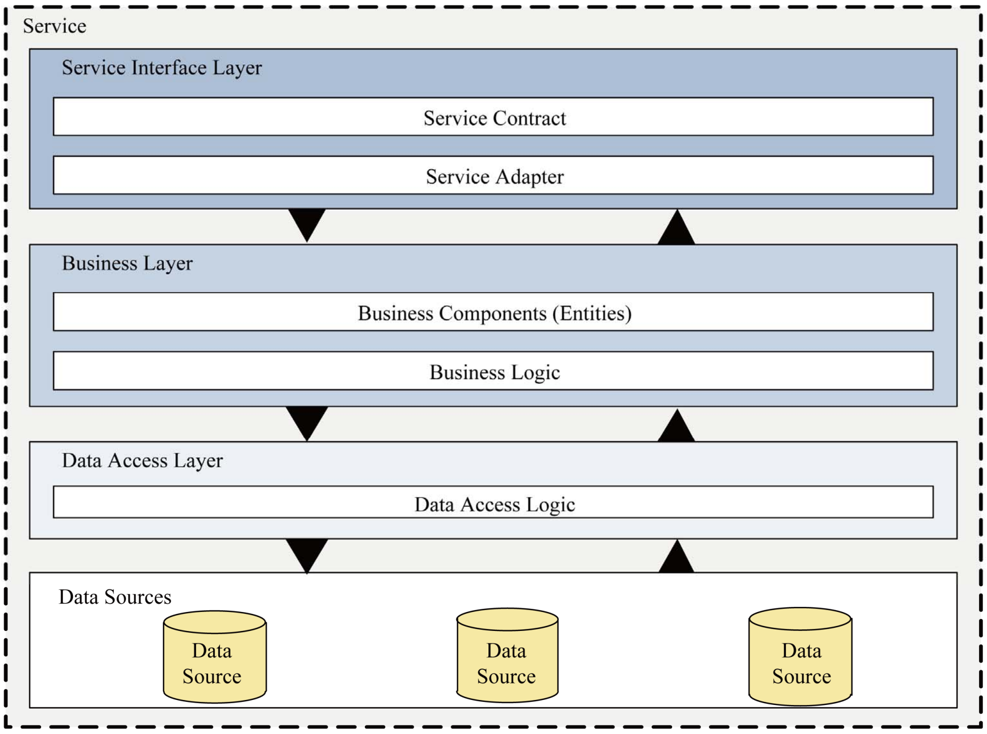

Figure 1. High-level architecture of a service within Service-Oriented architecture.

built-in mechanisms and techniques such as encapsulation, modularization, and separation of concerns. Since the service-oriented architecture reinforces the use of such mechanisms, applying OO mechanisms and techniques is a valid start to any SOA effort [5].

In an interview held by Info World in 2004, Grady Booch stated that "you start with web services and you start with good solid object-oriented architectures". He justified this by stating that "the fundamentals of engineering like good abstractions, good separation of concerns never go out of style", but "there are real opportunities to raise the level of abstraction again" [6].

Thus, the level of abstraction of services should be raised up to the business level for which the services have been developed for and not only focusing at the object's interface that describes the behavior of that object (class level). In fact, web services represent the next step in object-oriented programming, rather than developing software from a small number of class libraries provided at one location, programmers can access web service class libraries distributed worldwide [7]. **Table 1** shows the main concepts shared between the two paradigms and their key similarities and differences.

3. Related Work

The development process of any software system is driven by the architecture adopted for developing that system. However, the final goal of the software development process is to produce high quality software systems no matter which software architecture is adopted.

Barbacci [8] has examined several quality attributes and sub-factors (including efficiency, functionality, maintainability, portability, reliability, dependability, performance, and usability) and potential tradeoffs and fitness in the context of the adopted software architecture. He concluded that achieving such quality attributes does not only depend on the quality of the implementation (code-level) of the software system, but also depends on the adopted software architecture.

Likewise, Losavio, Chirinos, Levy and Ramadane-Cherif [9] have emphasized the importance of selecting the appropriate software architecture to fulfill quality requirements and attributes. The authors presented a comparison between the repository and publisher/subscriber (client/server) architectures. Their results have showed that publisher/subscriber is better than the repository approach with respect to security and efficiency in time behavior. However, the repository approach is superior for maturity and for efficiency in time.

Many research studies have investigated the effects of adopting the Object-Oriented approach on software quality and showed that object-oriented paradigm has a profound impact on software quality attributes.

For instance, Abreu and Melo [10] conducted an evaluation on the impact of object-oriented design on software quality characteristics by means of experimental validation. The authors used the MOOD (Metrics for Object Oriented Design) metrics suite to measure the OO mechanisms (including inheritance, coupling, information hiding, and polymorphism) and to assess how these mechanisms affect OO system's reliability (defect density) and maintainability (normalized rework).

The results obtained by their experiment showed that increased method encapsulation decreases defect density and the effort spent to fix defects, while attribute encapsulation did not show any effects on the quality. Also, increased method inheritance decreases defect densityand effort, whereas attribute inheritance has weak correlation

Table 1. Key similarities and difference between Service-Oriented and Object-Oriented paradigms.

Concept	OO Paradigm	SO Paradigm
Building Block	Objects are the key building blocks	Services are the key building blocks
Abstraction	A class hides the state and behavior of objects and provides an interface that separates the object's behavior from its implementation	A service hides the underlying implementation and logic through service interface (contract). This interface describes the business needs that the service is going to satisfy
Statelessness	Objects encapsulate data and behavior. The state of an object may be altered by the behavior of other objects	Services encapsulate business logic. Services are stateless; they do not depend on the state or context of other services
Loose Coupling	Although objects can be loosely coupled, the use of inheritance in the OO paradigm tends to create more tightly coupled objects	The use of service interfaces tends to create a loosely coupled environment which decouples services from consumers
Reusability	Objects modularity through the use of abstraction and encapsulation allows wide reusability of objects and classes	Separation of concerns between service interfaces and business logic allows modifying interfaces or business logic without affecting the service
Granularity Level	Within a distributed environment, objects have many dependencies with other objects resulting in fine grained objects	Services tend to be coarser-grained. Generally a service encapsulates a single business process independently that can be invoked through a service interface

with defect density and effort. The authors claimed that increased polymorphism would reduce defect density and rework; however, they noted that high values of Polymorphism Factor (POF) reduce these benefits. Finally, as coupling increases, defect density and rework also increase because coupling increases complexity, encapsulation, understandability, and maintainability.

A research conducted by IBM [11] to assess the direct and indirect effects of the object-oriented approach on software quality. The study involved three object-oriented projects developed by IBM. The study was based on measuring the quality attributes (code quality, correctness, usability, adaptive maintainability, perfective maintainability, and performance) of the three projects in the context of several object-oriented aspects (namely: O.O. user interface, O.O. design, O.O. programming, and iterative development) versus their corresponding traditional aspects (namely: action-oriented user interface, structured design, procedural programming, and waterfall development). The study showed that object-oriented technology produces immediate benefits in many aspects of software quality and productivity.

Briand, Wust and Lounis [12] provide an empirical study on the relationship of object-oriented design measures (cohesion, coupling, inheritance, and size in terms of methods and method parameters) and the fault-proneness of OO systems. The results obtained showed that classes with higher coupling, cohesion, and size are more likely to be fault-prone and classes that are located deeper in the inheritance hierarchy are less fault-prone.

Special attention has been given to the development of SOA to ensure the maturity of the field. Also, several standards have emerged (e.g.: WS-Security, WS-Reliable Messaging) to ensure the quality and interoperability of SOA among different vendors and organizations. However, little work has been done so far on how SOA impacts the quality of software and is still an open issue that needs to be empirically evaluated.

For instance, Haines [13] conducted a set if interviews with software developers and IT managers from different organizations to address the factors that affect the software development process by adopting SOA. The findings of this study points out that developing information systems based on web services and SOA is different from how information systems were developed in several areas. As a consequence, the development process of SOA requires changes in order to meet the requirements of web services and SOA. Interestingly, the author pointed that a key issue is that maintaining services once they have been published as well as the design of service interfaces in terms of granularity and reuse are significantly important for SOA software systems.

Perepletchikov *et al.* [14] provided a comparative study on the impact of object orientation and service orientation on the structural attributes of size (in terms of LOC), complexity (in terms of cyclomatic complexity (CC), weighted method per class (WMC), and number of children (NOC)), coupling between objects (CBO), response for a class (RFC)), and cohesion (in terms of lack of cohesion of methods (LCOM)) of software. The authors developed a system with two approaches. The first approach was built using coarse-grained services developed using the object-oriented principles, and the second approach consisted of fine-grained services developed using Business Process Execution Language (BPEL) scripts. Both systems were developed using Java. The results indicate that the SOA approach exhibits lower coupling and allows easier propagation of changes than the OO approach. On the other hand, the OO approach exhibits lower complexity than the SOA.

Another study by Perepletchikov [15] investigated the impact of several development strategies; top-down, bottom-up and meet-in-the middle of SOA on the project (Capital Cost and Development Effort) and structural software attributes involving Complexity, Coupling, and Cohesion. The authors provide guidelines for each development strategy when building SOA software systems.

The authors stated that services built from scratch (top-down development) should be coarser-grained so that the developed services would be highly reusable. When services are built using bottom-up strategy, the focus is on service granularity where services should be fine-grained in order to increase cohesion and decrease complexity and coupling. Finally, the authors pointed out several conflicting factors, most importantly is the service granularity. They stated that developing coarse-grained service introduces increased coupling and decreased cohesion resulting in lower system quality in terms of maintainability, reliability, and efficiency.

4. Design of the Empirical Study

4.1. Internal Software Quality Attributes and Metrics

An attribute is a "*measurable physical or abstract property of an entity*" [16]. A software quality attribute of a software system is a characteristic, feature or property that describes part of the software system. Internal software quality attributes reflect structural properties of a software system (e.g.: software size in terms of Lines of Code).

Such attributes can be quantitatively measured and directly applied to object-oriented systems. The attributes used in this study refer to the internal software attributes of size, complexity, coupling, and cohesion, which are explained bellow.

After a thorough study and analysis of the metrics mentioned in the literature a set of eleven metrics was chosen based on their importance and applicability to the object-oriented approach. The metrics used are mapped to the internal software attributes to be measured are discussed in below sections.

1) Size: is defined as the size of the software system in terms of Lines of Code (LOC) [17] that constitute the system. LOC is treated as follows:

- In the OO approach, all C# (pronounced c sharp) source files (classes) excluding comments and whites spaces were counted as code;
- In the SO approach, C# source files associated with each layer (Service Interface Layer, Business Layer, and Data Access Layer) were counted as code.

2) Complexity: is defined as the internal logic carried out in a software module or program unit. The measures used in this regard are as follows.

a) Traditional Cyclomatic Complexity (CC) and Extended Cyclomatic Complexity (ECC):

- In the OO approach, all conditional statements and loops within method bodies were counted in order to derive CC according to [18], in addition, compound conditional statements were counted to derive ECC according to [17].
- In the SO approach, all conditional statements and loops of each service and within the hosting application including the references of each service were counted in order to derive CC again according to [18] as well as compound conditional statements were counted to derive ECC again according to [17].

b) Halstead's Complexity (HC):

- In the OO approach, all operators and operands of each C# source file were counted and HC formulas were applied to derive HC according to [17,19].
- In the SO approach, all operators and operands of the source files of each service and the hosting application including the references of each service were counted to derive HC again according to [19, 20].

c) Maintainability Index (MI): the MI was computed for the entire system of both approaches since it is not computed on the method or class level [17]. The MI was derived according to [19, 20] for both systems.

d) Weighted Method per Class (WMC): the WMC can be measured by either counting the number of methods within a class or computing the total CC of the methods [14,17]. Since the CC was already computed, the total number of methods was calculated to indicate WMC in the OO approach, and the total number of operations in each service indicates WMC in the SO approach.

e) Depth of Inheritance Tree (DIT) and Number of Children (NOC):

- In the OO approach, the values of DIT and NOC were computed according to [21].
- In the SO approach, the concept of inheritance does not exist; that is, a service does not inherit from another service.

3) Coupling: is defined as a measure or indication of the strength of interdependencies and interconnections among software modules. The measures used in this regard are as follows.

a) Coupling between Objects (CBO):

- In the OO approach, CBO was measured according to [21].
- In the SO approach, CBO was measured as the coupling between services, where a service is said to be coupled to another service if one of them sends a message to the other.

b) Response for Class (RFC):

- In the OO approach, all local methods within a class and methods directly invoked within that class were counted to derive RFC according to [21]; in other words, all accessible methods within the class hierarchy were measured;
- In the SO approach, all methods within service layers as well as methods within the hosting application from which messages are sent to services were counted.

4) Cohesion: is defined as the extent to which each operation of a software module implements and performs a single task or functionality. In this context, cohesion has been measured in terms of the lack of cohesion of methods (LCOM) as follows.

Lack of Cohesion of Methods (LCOM):

- In the OO approach, LCOM was measured by counting the number of disjoint sets produced from the intersection of the sets of attributes used by the methods implemented [21].
- In the SO approach, LCOM was measured by counting the number of disjoint sets produced from the intersection of the sets of attributes used by the operations implemented within each service and among service layers, again according to [21].

The Chidamber and Kemerer metrics suite, known by CK metrics suite, was chosen since it is well-established and intensively discussed in the literature [21-23]. It addresses the following metrics: Weighted Method per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling between Objects (CBO), Response for Class (RFC), and Lack of Cohesion of Methods (LCOM)

4.2. Building the Testing Case

For the purpose of this study, a bank Automated Teller Machine (ATM) was selected as a case study [7]. The decision to consider the ATM as a case study was based on the fact that it presents a typical distributed system since it communicates with services from other banks and branches, and thus, suitable for SOA deployment. The ATM system has three main functions:

1) Balance inquiry which tells how much balance is available in an account, 2) Withdraw money for withdrawing money from a balance by subtracting the withdrawal amount from the available balance and 3) Deposit money for depositing the entered amount of money to the balance.

The system validates the user by card and PIN numbers. When the user withdraws an amount of money, the system checks if the available balance satisfies the amount to be withdrawn and if not, the transaction is canceled.

Although the ATM is considered to be a small case study, its design makes appropriate usage of the object-oriented structures and mechanisms such as association, inheritance, and aggregation. The case study was developed using C#.NET. In order to reflect the actual requirements of the ATM, the system was modified to retrieve and store accounts information from a database.

The development of the service-oriented version of the ATM system was carried out in two phases. In the first phase the ATM case study was thoroughly analyzed in order to identify the top-level use-cases and generate the Use-Case Priority Matrix [24,25]. In the second phase, five use-cases were considered as candidate services for the service-oriented system based on the use-case priority matrix.

The identified service candidates are: Authenticate User, Inquiry Balance, Withdraw Money, Deposit Money, and Invalid Account Number/PIN. The service-oriented system was developed as a set of fine-grained services (in which each service embeds its own business rules and logics, *i.e.*: withdraw service will contain the business rules and logic for withdrawing money from a customer account, etc,) as follows: the Authenticate User and Invalid Account Number/PIN use-cases (service candidate) were designed and implemented first based on the use-case priority matrix rankings, and thus conforming to the Rational Unified Process (RUP) methodology [26].

The remaining use-cases (Inquiry Balance, Withdraw Money, and Deposit Money) were designed, implemented separately and then integrated into the existing system. After that, a set of software metrics (*i.e.* those described in the previous subsection) was applied to the final resulting system implemented in both approaches (OO and SO) to evaluate their impacts on the internal software attributes of size, complexity, coupling, and cohesion.

The SO version of the ATM case study was developed using the C#.NET programming language and the Web Service Software Factory, also known as the Service Factory [27] (which includes best practices for developing service-oriented applications).

4.3. Software Quality Metrics Usage

Figure 2 illustrates the usage of the metrics used in this study and is described as follows:

1) The LOC metric was used to compare the size;

2) The CC, EC, HC, MI, WMC, DIT, and NOC metrics were used to compare the degree of complexity;

3) The CBO, and RFC metrics were used to compare the level of coupling between classes in the OO approach, and the coupling between service components in the SOA approach;

4) The LCOM metric was used to compare cohesion of classes in the OO approach, and the degree of cohesion between service components in the SOA approach.

4.4. Data Collection and Measurement Tools

The process of collecting metrics data from the source code of both systems was conducted as follows:

1) Automated Data Collection: this process involved using software metrics tools for collecting metrics data from C# code programs. Due to the limitations of the tools available for collecting metrics data of C# only the following tools were used:

- C# Code Metrics for calculating the LOC, CC, ECC, WMC, CBO, and LCOM.
- Reflector Code Metrics and NDepend for calculating DIT and NOC.
- C# Analyze for calculating HC.

The decision to use these tools was based on the fact that these tools provide most of the relevant metrics to this study.

2) Manual Data Collection: due to the lack and limitations of automated software tools, the data for RFC was collected manually.

5. Results and Discussion

In order to allow for an objective comparison and discussion, three informal hypotheses were formulated prior to the process of collecting metrics data based on intensive analysis of the related literature [2-4,6,13,28] concerning service orientation. These hypotheses were evaluated against the metrics data collected which include:

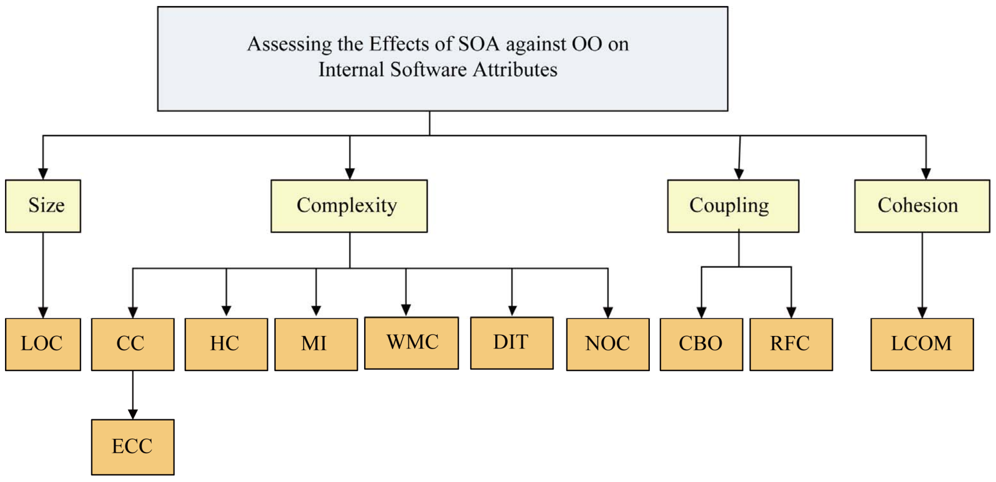
Assessing the Effects of SOA against OO on Internal Software Attributes
Size
Complexity
Coupling
Cohesion
LOC
CC
HC
MI
WMC
DIT
NOC
CBO
RFC
LCOM
ECC

Figure 2. Taxonomy of metrics usage.

- HYPOTHESIS 1: Implementations developed using the SO approach exhibit higher cohesion within methods compared to those of the OO approach since the business rules and logic are encapsulated within business components which are mapped against service operations rather than embedding the logic within the application code.
- HYPOTHESIS 2: Implementations developed using the SO approach exhibit lower coupling than those developed using the OO approach since services are autonomous and each operating on its own business components.
- HYPOTHESIS 3: Implementations developed using the SO approach exhibit higher size and complexity compared to those developed using the OO approach due to the structures and mechanisms required to implement and consume services as well as the fact that the OO approach is more mature and have been extensively experienced.

The measurement values collected from the OO and SO versions of the ATM system are shown in **Table 2**. **Figures 3** and **4** depict these values for the two versions.

6. Conclusions

The purpose of this study was to investigate the applicability of Object-Oriented software metrics to the Service-Oriented approach and how service orientation affects internal software attributes of size, complexity, coupling, and cohesion using a case study developed with the two contrasted approaches. The resulting implementations were measured using a set of eleven well-established and mature software engineering measures.

The quantitative comparison resulted in the following suggestions: 1) The SO approach tends to promote higher reusability of modules compared to those of the OO ap-

Table 2. Measurement values as collected from the Object-Oriented and Service-Oriented of the ATM testing case.

Attributes	Metrics	OO	SO
Size	LOC	432	601.
Complexity	CC	64	135
	ECC	65	141.0
	HC	14427.394	26867.799
	MI	108.63	63.01
	WMC	36	26.0
	DIT	15	1
	NOC	3	1
Coupling	CBO	27	10
	RFC	72	40
Cohesion	LCOM	5.65	4.96

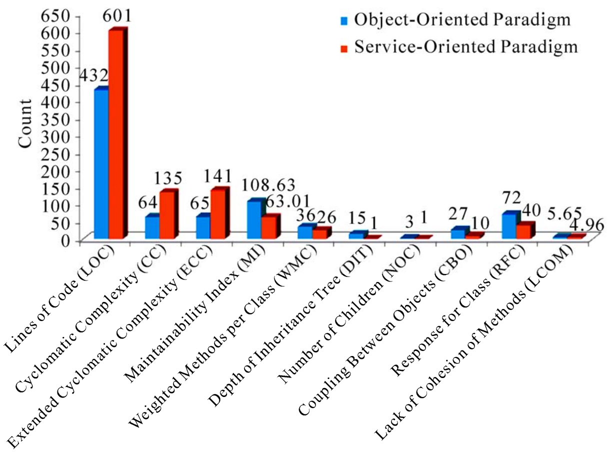

Figure 3. Software quality metrics results of both Object-Oriented and Service-Oriented implementations.

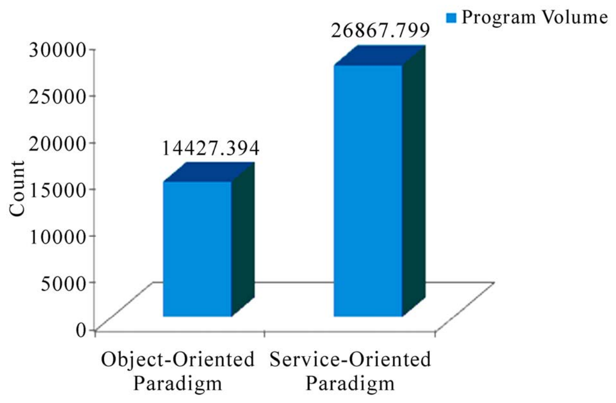

Figure 4. Total program volume (Halstead Complexity) of Object-Oriented and Service-Oriented implementations.

proach, 2) The SO approach provides a lower degree of coupling between modules than those of the OO approach, 3) The OO approach exhibits lower complexity than the SO approach, and 4) Not all OO metrics are applicable to the SO approach.

From this we conclude that there should be a compromise among internal software attributes in order to maintain a high degree of reusability while keeping the degree of complexity and coupling as low as possible. Another conclusion that can be made is that there is a need for developing a set of software metrics specifically toward SO approaches in order to measure the internal software attributes of SOA software systems since not all OO metrics are applicable to the SO approach.

There are a number of limitations associated with this study. Firstly, there are many different ways of designing and implementing the ATM case study using both OO and SO approaches. As a consequence, different designs and implementations might not exhibit the values presented in this study. Secondly, non-functional requirements, such as performance and security were not taken into account in the original OO case study. As a result, the effects of non-functional requirements on the internal software attributes are not clear.

REFERENCES

[1] T. Erl, "Services-Oriented Architecture: Concepts, Technology, and Design," Prentice Hall, Upper Saddle River, 2005.

[2] Z. Stojanovic and A. Dahanayake, "Service-Oriented Software System Engineering: Challenges and Practices," Idea Group Publishing, Hershey, 2005.

[3] A. Arsanjani, "Service-Oriented Modeling and Architecture: How to Identify, Specify, and Realize your SOA," Whitepaper, IBM Corporation, November 2004. http://www.ibm.com/developerworks/libraary/ws-soa-design1/

[4] T. Erl, "Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services," Prentice Hall, Upper Saddle River, 2004.

[5] O. Zimmermann, P. Krogdahl and C. Gee, "Elements of Service-Oriented Analysis and Design," Developer Works, IBM Corporation, 2004.

[6] G. Booch, "IBM's Grady Booch on Solving Software Complexity," InfoWorld Interview, 2004. http://www.infoworld.com/article/04/02/02/HNboochint_1.html

[7] Deitel & Deitel, "Visual C# 2005: How to Program," 2nd Edition, Deitel & Associates, Pearson Education, Upper Saddle River, 2006.

[8] M. Barbacci, "Software Quality Attributes and Architecture Tradeoffs," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, 2003.

[9] F. Losavio, L. Chirinos, N. Levy and A. Ramadane-Cherif, "Quality Characteristics for Software Architecture," *Journal of Object Technology*, Vol. 2, No. 2, March-April 2003, pp. 133-150. doi:10.5381/jot.2003.2.2.a2

[10] F. B. Abreu and W. Melo, "Evaluating the Impact of Object-Oriented Design on Software Quality," *Proceedings of the* 3*rd International Software Metrics Symposium*, Berlin, March 1996, pp. 90-99. doi:10.1109/METRIC.1996.492446

[11] N. P. Capper, R. J. Colgate, J. C. Hunter and M. F. James, "The Impact of Object-Oriented Technology on Software Quality: Three Case Histories," *IBM Systems Journal*, IBM, Vol. 33, No. 1, 1996, pp. 131-157.

[12] L. Briand, J. Wust and H. Lounis, "Investigating Quality Factors in Object-Oriented Designs: An Industrial Case Study," *ICSE'*99 *Proceedings of the* 21*st international conference on Software engineering*, New York, 1999.

[13] M. Haines, "The Impact of Service-Oriented Application Development on Software Development Methodology," *Proceeding of the* 40*th Hawaii International Conference on System Sciences*, Hawaii, January 2007, pp.172b.

[14] M. Perepletchikov, C. Ryan and K. Frampton, "Comparing the Impact of Service-Oriented and Object-Oriented Paradigms on the Structural Properties of Software," *Second International Workshop on Modeling Inter-Organizational Systems*, Cyprus, Vol. 3762, 2005, pp. 431-441.

[15] M. Perepletchikov, C. Ryan and Z. Tari, "The Impact of Software Development Strategies on Project and Structural Software Attributes in SOA," *Proceedings of the* 2*nd INTEROP Network of Excellence Dissemination Workshop*, Cyprus, 31 October -4 November 2005.

[16] C. Kaner and W. Pond, "Software Engineering Metrics: What Do They Measure and How Do We Know?" 10*th International Software Metrics Symposium METRICS*, Chicago, 11-17 September 2004.

[17] L. Laird and C. Brennan, "Software Measurement and Estimation: A Practical Approach," *IEEE Computer Society*, John Wiley & Sons, Hoboken, 2006.

[18] T. McCabe and A. Watson, "Software Complexity,"

Journal of Defense Software Engineering, 1994. http://www.stsc.hill.af.mil/crosstalk/1994/12/xt94d12b.asp

[19] M. Halstead, "Elements of Software Science," Elsevier North-Holland, Inc., New York, 1977.

[20] K. Welker and P. Oman, "Software Maintainability Metrics Models in Practice," *Journal of Defense Software Engineering*, 1995. http://www.stsc.hill.af.mil/crosstalk/1995/11/maintain.asp

[21] S. Chidamber and C. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, June 1994.

[22] V. Basili, L. Briand and W. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators," *IEEE Transactions on Software Engineering*, Vol. 22, No. 10, October 1996.

[23] L. Briand, J. Daly, V. Porter and J. Wust, "A Comprehensive Empirical Validation of Product Measures for Object-Oriented Systems," Fraunhofer Institute of Experimental Software Engineering, Kaiserslautern, 1998.

[24] Ariadne Training, "UML Applied-Object Oriented Analysis and Design," 2nd Edition, Ariadne Training Limited, UK, 2005.

[25] W. Bentley, "System Analysis & Design Methods," 7th Edition, McGraw-Hill, New York, 2007.

[26] P. Kruchten, "The Rational Unified Process: An Introduction," 3rd Edition, Addison-Wesley, Boston, 2003.

[27] Microsoft, "Introducing the Web Service Software Factory," Microsoft Corporation, 2006. http://www.msdn2.microsoft.com/en-us/library/aa480534.aspx

[28] M. Endrei, J. Ang, A. Arsanjani, S. Chua, P. Comte, P. Krogdahl, M. Luo and T. Newling, "Patterns: Service-Oriented Architecture and Web Services," IBM Redbooks, IBM Corporation, 2004.