Communications in Mathematics and Applications
Vol. 13, No. 2, pp. 775–782, 2022
ISSN 0975-8607 (online); 0976-5905 (print)
Published by RGN Publications
DOI: 10.26713/cma.v13i2.2033

http://www.rgnpublications.com

Review Article

Facilitating Software Reuse Through Design Characteristics in Object-Oriented Paradigm

Rihab Al-Mutawa and Wajdi Aljedebi*

Department of Computer Science, Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah, Saudi Arabia
***Corresponding author:** waljedaibi@kau.edu.sa

Received: January 5, 2022 **Accepted:** February 24, 2022

Abstract. Software are used to increases quality as well as productivity that could be associated with lower costs. Implementing software with capability for reuse is difficult without sufficient support. This paper includes tutorial represents four important design characteristics in object-oriented paradigm for facilitating the production of more reusable software. It reviews the concepts of modularity, cohesion, coupling and information hiding and provides a discussion about their effect on software reuse. Object-oriented approach is having wide attention both in research environments and in industry. This tutorial is prepared for newbie developers of reusable software assets and for everyone else interested in the subject.

Keywords. Software reuse, Object-oriented, Design characteristics, Modularity, Cohesion, Coupling, Information hiding

Mathematics Subject Classification (2020). 68N19

1. Introduction

The demand for new software applications is nowadays increasing at exponential rate accordingly the cost of developing them is increasing. A major problem is the number of qualified and experienced professionals required for this growing demand is not increasing enough [4]. Effective software products reuse can help in increasing productivity and quality, saving time, and decreasing the cost of software development [7, 22]. There are two approaches for code reuse: development of reusable code from scratch or extraction and identification of reusable

code from already developed code [4]. This paper is concerned with development of reusable code from scratch in object-orientation. One of the most outstanding goals of object-orientation is to increase the reusability of software [24]. The paper provides a guide explains the four important design principles in object-oriented software engineering: modularity, cohesion, coupling, and information hiding, and it provides a discussion about their influence on software reuse. It helps newbie object-oriented developers of software for reuse who need to know the important design characteristics in object-oriented and their impact on reuse [26]. Measuring of these characteristics will give a chance to predict the change needed to make the module reusable or to predict its reusability [29]. This topic of measuring them is out of the scoop for this work.

The paper is organized as follows. Section 2 gives background information about software reuse, object-oriented paradigm. Section 3 surveys the important design characteristics in object-oriented paradigm and explain show they are related to software reuse. Section 4 discusses the effect of these design characteristics on software reuse. Finally, Section 5 summarizes the contributions of this work.

2. Background

Since programming began before 1940s, software reuse has been practiced ever since. Reuse as a distinct area of study in software engineering is often traced to Doug Mcilroy's paper, back from 1968, which proposed basing the software industry on reusable components [10, 15, 25]. Software reusability is considered to be important for good software [29].

Object-orientation dates are back to the 1950's but it is only got popular in 1989 [23]. Object-oriented programming is a special type of modularization prepared to ensure programs quality [14]. A typical task in object-oriented approach is reusing software assets [6]. It makes programming easier and faster also with higher quality because of the reuse of existing classes from a library and those classes are previously tested. These promises of reuse and quality are meant to separate object-oriented languages from the more traditional languages among other promises such as easier maintenance [23]. Object-oriented programming is intended to be both a superior development tool for programs and a partial effective solution to the software reuse problems that include lack of tool support [8, 31], and it succeeded in achieving its goal and became superior to other sorts of software [23].

3. Design Characteristics

In literature, there are many guidelines found to support the designer of object-oriented paradigm. The important classic software engineering guidelines developed especially for structured design are also important in object-oriented design. These software engineering guidelines or characteristics are: modularity, low module coupling and high module cohesion, data hiding [19].

3.1 Modularity

Modularity is the degree to which a computer program or system is composed of discrete components where a change to one component has minimal effect on other components [28]. It is a conclusion of separation of concerns and module is a device to implement a concern [30]. The primary mechanism for reuse is module [23]. One of the purposes of a module is acting as a unit of reuse and reusability is one of the most important benefits of modularization. This

reusability means that the module can be copied from the project it was originally developed for and used in other projects. Copying, reusing and adapting modules(such as functions, classes, aspect or components) are a frequent activity in all development of software projects. Typically, the need to adapt the module is minimal. Ideal techniques of modularization is support the reuse of modules by giving clearly defined boundaries for the modules in the source file, or requiring that each module be put into a file of its own [12].

Object-oriented programming is the best paradigm that divides up work. It is scalable and makes large systems easier to build and maintain because subsystems can be developed and tested autonomously [23]. When a system is divided into modules, it requires high cohesion and low coupling [27]. The goal is to create modules with internal integrity i.e., having a high cohesion and small, direct, visible, and flexible relations to other components i.e., having low coupling [20]. Cohesion and coupling are attributes of quality and they are generally recognized as being among the most probable quantifiable indicators for software maintainability [3].

3.2 Cohesion

Cohesion is the manner and degree to which the tasks performed by a single module in software are related to one another. It is an indication to module strength and contrasts with coupling [28]. It falls into one of the following categories: functional, sequential, communicational, procedural, temporal, logical and coincidental. These categories are ordered from highest (best) cohesion type to lowest (worst) cohesion type [9]. A module must have well-defined responsibilities, i.e. it has high cohesiveness. Cohesiveness of module means that it is carrying out only one task. When the module is highly cohesive, i.e. all elements in the module directly deal with the one basic task or a group of similar tasks the module carries out [17]. A high degree of cohesion is an indicator of good design for module [13]. High cohesion leads to code that is easier to understand. Understanding is predominantly cited as the most time consuming component of the maintenance activity. Developers of object-oriented programming normally assign responsibilities to classes to keep the level of cohesion high as the main objective [1], and achieving a high degree of cohesion between the methods in a class can be yield by encapsulating several methods in the class [23]. This increases the probability of reuse and creates modules with keeping its complexity manageable. If the responsibilities of a class are unrelated then, the level of cohesion decreases, i.e., they do a wide range of distinct actions or operate on various types of data [1].

In the guideline literature, class cohesion has two different meanings. The first meaning refers to the degree in which operations and values within a class are related to each other. The second meaning indicates the number of major functions performed by a class. Class that has more than one main function is not cohesive and should split into multiple classes. In object oriented design, both meanings can be subsumed into a central idea that good design of object requires steps of first identifying major responsibilities, second, dividing the responsibilities among objects, and finally identifying the internal data needed by the object so that it performs the services it is responsible for [13].

Documenting software architecture is an indication of cohesion. It facilitates communication between several stakeholders, recordsearly decisions about high-level design, and allows reuse of design components between projects. As stated above, cohesion is also represented through modules. By breaking the project down into pieces, the isolation of the problems and fixing

them become easier and these pieces could be reused. And to improve cohesion, reliability, and maintainability throughout the software lifecycle; refactoring is used. Refactoring is a technique for restructuring a code, altering its internal body structure without changing its external behavior. It enhances code design and code quality. Also,it increases code reuse and developer productivity. An example for increasing code reuse through refactoring is if multiple functions use similar piece of code i.e. duplicated code, then, the common code can be refactored into new function that the multiple functions can call [5].

High cohesion facilitates reuse because of well-defined modules, simplifies modification because all relevant code in one place, and lowers coupling to other modules because it raises coupling within the module.

3.3 Coupling

Coupling is the manner and degree of interdependence between modules in software [28]. It is the complement of cohesion [20]. It falls into one of the following categories: no direct coupling, data, stamp, control, external, common and content. These categories are ordered from lowest (best) coupling type to highest (worst) coupling type [9]. Two classes are coupled, when a function in one class makes calls to a function in another class, or accesses attributes of the other class, and when two objects send messages to each other then they are coupled. Low degrees of coupling between classes can lead to more comprehensible code, less testing required, more reusable code, more modular code, and more maintainable code. If classes are highly coupled then the effect of change or error in one class may propagate to a large number of other classes [13]. More faults might be introduced due to activities of inter-class. Also, too much coupling points to a weakness in class encapsulation and may prevent reuse [2]. Degree of interdependence between classes should be lowered to a certain degree. There is a necessary certain amount of coupling so that the software is useful. Classes that are related through inheritance are of necessity highly related to each other [13]. Code reuse supported in object-oriented approach through inheritance, polymorphism and aggregation [18]. But, inheritance breaks encapsulation as it is often said because inheritance reveals a subclass to details of its parent's implementation [11].

In software reuse procedure, the most serious problem with the component extraction step is an overly high degree of dependency i.e., coupling throughout the whole software product. It is therefore hard to reuse required packages or subsystems without making numerous changes. If one package is picked up for reusing, then almost the whole system is gotten because of dependencies. An indication to the effect of dependency between the different subsystems is the recompilation of large parts of the system for only minor modifications in any of its subsystems. Subsequently, it is difficult to extract a single subsystem as a reusable component out of the whole system [29].

Object-oriented design and programming reduces coupling [23]. Data and its functionality are incorporated into objects thereby reducing the coupling between objects [29], and to improve coupling, software reuse and maintainability; refactoring is used [3].

3.4 Information Hiding

Information hiding is a development technique in software such that each module's interfaces uncover as little as possible about the module's inner workings and using information about the module that is not in the module's interface specification by other modules is prevented [28].

The best approach in protecting data is the object-oriented [23]. It hides data from the public by encapsulation that is the technique used to accomplish the goal of information hiding [16] Encapsulation in object-oriented languages gave rise by information hiding [21]. It is a development technique in software that consists of isolating a function or a set of data and operations in system, on those data within a module and providing precise specifications for the module [28].

Information hiding is a key concept in object-oriented programming. Data that is probably is going to change should be separated and hidden from the public interface. The internal data structure and implementation of the public interface should not be exposed in the interface, so that these internals can be modified without affecting applications using the interface,then the result is interface that is simpler to understand, and so that reuse of the class takes less effort [13].

4. Discussion

The important design characteristics which are modularity, strong cohesion, weak coupling and information hiding in object-oriented paradigm are all an indication for good design because they positively effects on understanding, facilitating software reuse and maintenance of software products. Accordingly lack of these characteristics when they are suited to be done would lead to opposite results including obstructing software reusability. Developing software with highly reusability in object-oriented software engineering requires implementing the four concepts mentioned earlier, and combining object-oriented software engineering's concepts with reuse-based software engineering's development for reuse process can create a reusable software assets (such as patterns, components, frameworks, application systems, etc) with highest reusability.

Reusability of software considered effective on software quality. There is a direct relationship between software reusability and software quality. Software quality grows as reuse of software modules increases because of the reuse of existing modules that are previously tested, and quality of software cannot be known and improved unless if it can be measured. There are papers such as [2] present approaches to measure the software reusability for attributes of software quality in object-orientation that are modules, encapsulation, cohesion and coupling. When the characteristics get measured and enhanced as possible -such by refactoring for module's cohesion - then they will have the best positive influence on software reuse. Measuring these characteristics will give a chance to evaluate them. i.e., predict module's reusability and accordingly, predict the required change to make to the module to make it reusable for when needed.

The mentioned design concepts do not exist only on object-oriented paradigm but what makes them special in software reuse in this paradigm are as following. First, Object-oriented programming is the best approach implements module because of its feature of scalability and it makes large systems easier in building and maintenance because its subsystems can be developed and tested independently. Second, assigning responsibilities to classes is keeping the level of cohesion high and it is normally is a main objective by developers. Third, data and its functionality are integrated into objects thereby reducing the coupling between objects. Finally, the best approach in protecting data is the object-oriented because it hides data from the public by encapsulation which used in achieving the information hiding.

High cohesion of modules leads to low coupling between modules and low cohesion of modules leads to high coupling between modules. The divided system into modules is requiring high cohesion and low coupling and achieving a high degree of cohesion between the methods in a class and low coupling between classes can be yield by encapsulating several methods in the class. This raises the likelihood of reuse and creates modules with keeping its complexity under control. This can explain how all the four design characteristics are related to each other in the subject of software reuse.

5. Conclusion

The field of software reuse has got much attention over the last two decades and continued until now, particularly with the common use of object technology. The increasing high requests for new software application is reinforced and encouraged software reuse especially that the increasing demand is associated with increasing cost in developing them and the number of qualified and experienced professionals required for this growing demand is not growing enough.

Efficient reuse for software products can be helpful in increasing productivity and quality, saving time,and decreasing the cost of software development. This paper as presented is concerned with development of reusable code from scratch in object-oriented paradigm. Increasing the reusability of software is a prominent goal in object-orientation. This goal is achieved through four important design concepts in object-oriented programming: modularity, cohesion- and it has to be high-, coupling-and it has to be low- and information hiding by encapsulation. All of these concepts have positive influence on software reuse as discussed above.

Competing Interests

The authors declare that they have no competing interests.

Authors' Contributions

All the authors contributed significantly in writing this article. The authors read and approved the final manuscript.

References

[1] N. Barnes, D.P. Hale and J.E. Hale, The cohesion-based requirements set model for improved information system maintainability, *Proceedings of the 2006 Southern Association for Information Systems Conference* (2006).

[2] P.K. Bhatia and R. Mann, An approach to measure software reusability of OO design, in: *Proceedings of 2nd National Conference on Challenges & Opportunities in Information Technology* (COIT-2008), RIMT-IET, Mandi Gobindgarh, March 29, 2008, pp. 26 – 30.

[3] B.D. Bois, S. Demeyer and J. Verelst, Refactoring - improving coupling and cohesion of existing code, *11th Working Conference on Reverse Engineering*, 2004, pp. 144 – 151, DOI: 10.1109/WCRE.2004.33.

[4] N. Budhija and S.P. Ahuja, Review of software reusability, in: *International Conference on Computer Science and Information Technology* (ICCSIT'2011), Pattaya, Thailand (December 2011).

[5] S.M. Chandrika, E.S. Babu and N. Srikanth, Conceptual cohesion of classes in object oriented systems, *International Journal of Computer Science and Telecommunications* **2**(4) (2011), 38 – 44, URL: https://www.ijcst.org/Volume2/Issue4/p8_2_4.pdf.

[6] B. Childs and J. Sametinger, Literate programming and documentation reuse, in: *Proceedings of Fourth IEEE International Conference on Software Reuse*, pp. 205 – 214, (1996), DOI: 10.1109/ICSR.1996.496128.

[7] J.L. Cybulski, *Introduction to Software Reuse*, Technical Report TR 96/4, The University of Melbourne, Australia (1996).

[8] D.J. Eck, *Objects and object-oriented programming*, Section 1.5, in: *Introduction to Programming Using Java*, Version 9.0, JavaFX Edition, (2022), http://math.hws.edu/javanotes/c1/s5.html.

[9] J. Eder, G. Kappel and M. Schrefl, Coupling and cohesion in object-oriented systems, *Technical Report*, University of Klagenfurt, Austria (1994).

[10] W.B. Frakes and K. Kang, Software reuse research: status and future, *IEEE Transactions on Software Engineering* **31**(7) (2005), 529 – 536, DOI: 10.1109/TSE.2005.85.

[11] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, Reading, MA, USA (1995), URL: http://www.javier8a.com/itc/bd1/articulo.pdf.

[12] A. Garcia, P. Greenwood, G. Heineman, R. Walker, Y. Cai, H.Y. Yang, E. Baniassad, C.V. Lopes, C. Schwanninger and J. Zhao, *ACM SIGSOFT Software Engineering Notes* **32**(5) (2007), 31 – 37, DOI: 10.1145/1290993.1291005.

[13] A. Goethals, *Use of 3d Program Visualization to Show Visibility, Cohesion, and Quality of Java Class Elements*, Doctoral dissertation, University of Florida, Florida, USA (2002).

[14] S. Gosch, *A Short History of Programming Languages*, YUMPU (2007), URL: https://www.yumpu.com/en/document/view/5443927/a-short-history-of-programming-languages-1-how-computers-and-.

[15] L. Hardesty, Automatic code reuse: System makes modifications necessary to transplant code from one program into another, *MIT New on Campus and Around the world*, MIT News Office (September 19, 2017), URL: https://news.mit.edu/2017/automatic-code-reuse-0920.

[16] R. Harmes and D. Diaz, *Pro JavaScript Design Patterns*, Apress, (2008), URL: https://pepa.holla.cz/wp-content/uploads/2016/08/Pro-JavaScript-Design-Patterns.pdf.

[17] S. Harris, *CISSP All-in-One Exam Guide*, 5th edition, McGraw-Hill, Inc., New York, USA, 1008 pages, URL: https://dl.acm.org/doi/10.5555/1594805.

[18] Y. Hassoun, *Coupling, Code Reuse and Open Implementation in Reflective Systems*, Doctoral dissertation, School of Computer Science and Information Systems, Birkbeck College, University of London, US (2005), URL: https://www.dcs.bbk.ac.uk/site/assets/files/1025/yhassoun.pdf.

[19] I.M. Holland and K.J. Lieberherr, Object-Oriented Design, *ACM Computing Surveys* **28**(1) (1996), 273 – 275, URL: https://dl.acm.org/doi/pdf/10.1145/234313.234421.

[20] T. Husted, C. Dumoulin, G. Franciscus, D. Winterfeldt and C.R. McClanahan, *Struts in Action: Building Web Applications With the Leading Java Framework*, Manning Publications, 664 pages (2003), URL: https://ur.sa1lib.org/book/461946/506b86.

[21] I. Jacobson, M. Christerson, P. Jonsson and G. Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, Wokingham, England (1992), URL: https://search.library.uq.edu.au/primo-explore/fulldisplay?vid=61UQ&tab=61uq_all&docid=61UQ_ALMA21113243750003131&lang=en_US&context=L&query=sub,exact,Image%20processing%20--%20Periodicals,AND&mode=advanced.

[22] M. Jha and L. O'Brien, A comparison of software reuse in software development communities, *2011 Malaysian Conference in Software Engineering*, 2011, pp. 313 – 318, DOI: 10.1109/MySEC.2011.6140690.

[23] P. Jorgensen, D. Fernandez, A. Fischer, M. Greco, B. Hussey, S. Kuchta, H. Li, S. Overkamp, D. Rodenberger and R. VanderWal, *Has the Object-Oriented Paradigm Kept Its Promise*?, Report, Grand Valley State University, Allendale, USA (2002), URL: http://ddi.cs.uni-potsdam.de/HyFISCH/Informieren/Programmiersprachen/OOPromisesAndReality.pdf.

[24] T. Lindner and A. Rüping, *How Formal Object-Oriented Design Supports Reuse?*, Report, Forschungszentrum Informatik (FZI), Karlsruhe, Germany (1995).

[25] K.C. Louden, *Programming Languages: Principles and Practice*, 2nd edition, Cengage Learning (2003), UR: http://www.cs.sjsu.edu/~louden/pltext/plpp_ch01.pdf.

[26] S. Parker, Building reusable software, in *Proceedings of TOOLS Europe'99: Technology of Object Oriented Languages and Systems. 29th International Conference*, Nancy, France (1999), pp. 409, DOI: 10.1109/TOOLS.1999.10002.

[27] S.L. Pfleeger and J. Atlee, *Software Engineering Theory and Practice*, Prentice Hall (2001).

[28] J. Radatz, *IEEE Standard Glossary of Software Engineering Terminology*, The Insitute of Electrical and Electronics Engineers, New York, USA (1990), URL: http://www.mit.jyu.fi/ope/kurssit/TIES462/Materiaalit/IEEE_SoftwareEngGlossary.pdf.

[29] H. Ramakrishnan, *Analysis of Complexity and Coupling Metrics of Subsystems in Large Scale Software Systems*, Doctoral dissertation, University of Central Florida, Orlando, Florida (2006), URL: https://stars.library.ucf.edu/cgi/viewcontent.cgi?article=1745&context=etd.

[30] N. Singh and N.S. Gill, Aspect-oriented requirements engineering for advanced separation of concerns: a review, *International Journal of Computer Science Issues* **8**(5) (2011), 288 – 297, URL: https://ijcsi.org/papers/IJCSI-8-5-2-288-297.pdf.

[31] I. Sommerville, *Software Reuse, Software Engineering*, 9th edition, Addison-Wesley, Boston, 773 pages (2011).