

ACCA_7000_OJORDAN(P144558)

by Oliver Jordan

Submission date: 16-Jan-2023 11:31AM (UTC+0000)

Submission ID: 195165170

File name: 2911_Oliver_Jordan_ACCA_7000_OJORDAN_P144558__476351_674589432.docx (25.03K)

Word count: 1551

Character count: 7967

Java Programming, Testing and Development

Oliver Jordan (P144558)

ACCA7000 Programming and Software Development

MSc Software Development (Web Development)

Tim Bashford

Table of Contents

1	<i>Introduction</i>	3
2	<i>Cohesion</i>	3
3	<i>Loose Coupling</i>	5
4	<i>Data Hiding, Information Hiding and Encapsulation.....</i>	6
5	<i>Conclusion.....</i>	7
6	<i>Bibliography.....</i>	8

1 Introduction

There are many concepts surrounding the architecture of code that have been identified as areas to monitor in order to program at a higher quality. This report aims to investigate and understand some of those concepts. As a program was also required for submission alongside this report, areas where these concepts could be brought into, or where they already exist, will also be highlighted.

2 Cohesion

Programs are developed in order to tackle a problem, and larger programs may be tackling multiple problems. With solutions having the possibility of becoming large and having several interconnecting pieces of code, the architecture of the solution becomes important to properly maintain. If the code for such a solution were to be found in a single file, whoever came to maintain the code after its development would likely find their sanity halfway out the door.

This is where the concept of cohesion comes into play. Cohesion refers to how well the contents of a modular set of code relates to each other. A module with higher cohesion is more transparent, readable and easier to debug [1]. Cohesion is strived for in order to make maintaining code simpler, and for code to be found easier for reuse or to be worked upon. Legacy systems are often restructured in order to increase their cohesion so that they may continue to stay alive with less upkeep required from developers [2].

When considering how well the contents of a modular piece of code relate to each other, all functions must be considered. Function X and Function Y may be directly related to one another, but the level of cohesion is impacted if Function Z has nothing to do with them. In this case, moving Function Z to its own class or file would increase the cohesion of both resulting modules.

There are several levels of cohesion to consider [1], ordered in least cohesive to most:

- Coincidental association – These modules contain functions that only associated by chance, perhaps three lines of code have been brought out to a function simply because they are the same three lines of code as somewhere else.
- Logical association – Modules that contain functions that can be logically linked together, though the functions may do something altogether different, are logically associated. A module that contains various input functions regardless of where the input is coming from is a nice example of this.
- Temporal association – Modules that run functions that all need to occur at a similar point in time; a video game may perform a first load upon its launch and several rendering engine initialisations, and place all these functions in a single module called “startUp”.
- Procedural association – A module may contain functions that are always executed in a certain order, for example taking an input and then validating it, making it procedurally associated.
- Communicational association – Communication association occurs when functions all perform acts upon the same set of data.
- Sequential association – When functions are grouped because the output of one becomes the input of the next, they are sequentially associated.
- Functional association – When functions are grouped because they contribute to the solution to a particular problem, they are functionally associated.

It should be noted that while these are ordered in levels of cohesion, a coincidentally associated module should not be restructured for the pure sake of increasing cohesion, unless the lack of cohesion is detrimental to the project.

The code submitted alongside this report was created for a console based connect four game. All of the functions associated with the running of the game are in a single class file, connectFour.java. This module exhibits functional association, as the code within is all there for the same functional purpose.

3 Loose Coupling

Coupling refers to how many links to another modular piece of code exists in a module. If, say in java, a class has to call a second class in to execute several different functions, then it could be considered to have a tight coupling with the second class. On the flip side to this, a class may only other classes to interact with it by retrieving a final piece of data, and nothing else. This class would have a loose coupling with any class that interacts with it.

Loose coupling is useful to programmers as code that has fewer dependants and dependencies can be restructured with little impact to other code [3] [4]. It also allows code to become reusable with much less effort – if a chunk of code only has one way to interact with it, it becomes very much a plug and play scenario.

Loosely couple code is more often than not highly cohesive, as code that is loosely coupled require all their moving parts to be in one place, the code is more likely to exhibit communicational or even functional cohesion.

With regards to the connect four code submitted for this module, the main connect four module is only intended to be interacted with by calling the class's startGame function, which kicks off the whole connect four board. This can be considered loose coupling, as if the entire inner workings of the connect four game were to change, the code that actually sets it going in other code will not need to change.

4 Data Hiding, Information Hiding and Encapsulation

These three concepts are so closely linked, it can be difficult to pick them apart. Booch [5] describes information hiding as the process of ensuring the inner workings of a piece of code are never seen by any other pieces of code or users. For example, a piece of code that calculates an output based on a singular input and several internal factors would never expose said internal factors to anything other than the associated code.

Data hiding specifically refers to the act of ensuring individual pieces of data within a piece of code are only visible within said code. This ensures that the data cannot be accessed and modified or manipulated where it should not be able to and helps to keep coupling low. Data encapsulation, on the other hand is the act of ensuring methods are also hidden where appropriate.

Information hiding, conversely, hides the design of the module itself [5]. This is to ensure that any design choices that may affect the program as a whole is contained within a module. As an example, a point in 3D space may be described in one piece of code as an object containing three numbers (X,Y,Z). Another piece of code describes a similar point of space as an array of these numbers instead. Information hiding would ensure that even though these design choices should clash, an interface exists that allows them to work together, and that any design changes that occur within a module affect primarily that module.

Ensuring that code is only interacted with for the final output that the code is designed to deliver can help keep modules loosely coupled and highly cohesive – hidden data cannot be easily coupled to other modules, and if all functions are enforcing hiding correctly, then each module will likely contain the methods and data it needs within itself, creating a nice level of cohesion. Creating reusable code through these methods also allow programmers to build solutions that require a

common problem solved more efficiently, and may even encourage library sharing, as why write code again when a perfectly serviceable solution already exists?

Within the submitted code, several pieces of data were left as public as they were being used elsewhere in the project – for example, certain object data was being manipulated to set up tests. Following the research done in this section, the code has been modified to enforce data hiding, with the creation of functions designed to interface with the class to set and retrieve data where needed.

5 Conclusion

This report has briefly explored some programming concepts that influence code design with the intention of creating modular, easily to restructure code. All of these concepts appear to be prevalent in object-oriented programming, a programming style which most large projects and programming languages follow.

Using the research carried out for this report, some code written for submission alongside this report was highlighted where it followed one of these concepts, and improvements made where applicable.

Overall, these concepts describe the importance of highly cohesive and modular code, as they can become the building blocks of many solutions when they are easily restructured and contained.

6 Bibliography

- [1] E. Yourdon and L. L. Constantine, Structured Design, New York: Yourdon Press, 1978.
- [2] A. Lakhotia and J.-C. Deprez, "Restructuring functions with low cohesion," in *Sixth Working Conference on Reverse Engineering*, Atlanta, 1999.
- [3] E. Gamma and E. Al., Design Patterns: Elements of Reusable Object-Oriented Software, Boston: Addison-Wesley, 1995.
- [4] W. B. McNatt and J. M. Bieman, "Coupling of Design Patterns: Common Practices and Their Benefits," in *25th Annual International Computer Software and Applications Conference. COMPSAC 2001*, Chicago, 2001.
- [5] G. Booch and E. Al., Object-Oriented Analysis and Design with Applications, Upper Saddle River: Addison-Wesley, 2007.

FINAL GRADE

70/100

GENERAL COMMENTS

Instructor

Engagement with Literature Skills (15): 10

Knowledge and Understanding Skills (15): 9

Cognitive and Intellectual Skills (15): 12

Practical Application Skills (50): 36

Transferable Skills for Life and Professional Practice (5): 3

A good effort on the report. A small selection of academically credible sources used. How you formulate your argument could use a little work, your written style is fine, but a tad informal in places. You might also consider including a few relevant diagrams/graphs/tables to break up the body of the text and swapping to single spacing.

A good effort with the code - the solution is maybe a little brute forced, however. There are terser, cleaner ways of achieving the same thing while avoiding, for example, the massive switch statement. Nonetheless, a good piece of work overall.

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8
