

ACCA7000_sem1_Assignment2_ Part1_Written_Agnes_Beviz_FINAL

by Agnes Beviz

Submission date: 16-Jan-2023 10:59AM (UTC+0000)

Submission ID: 195157415

File name:

42220_Agnes_Beviz_ACCA7000_sem1_Assignment2_Part1_Written_Agnes_Beviz_FINAL_476351_1053837151.docx
(2.99M)

Word count: 2209

Character count: 11769



Prifysgol Cymru
Y Drindod Dewi Sant
University of Wales
Trinity Saint David

University of Wales Trinity Saint David

School of Computing

15/01/2023/ACCA7000

Individual Assignment Part I

Software Design Principles for Clean and Maintainable Code

By Agnes Beviz

Table of Contents

Table of Contents.....	2
List of Tables.....	2
List of Figures	2
1. Abstract.....	4
2. High Cohesion	5
3. Loose Coupling.....	7
4. Information Hiding and Abstraction.....	9
5. Data Hiding and Encapsulation	10
6. Conclusion	12
References.....	13

List of Tables

Tab. 1 Table defining different categories of coupling [8].	7
---	---

List of Figures

Fig. 1 Illustration showing a 'clean code' implementation with low coupling and high cohesion [3].	4
Fig. 2 Illustration showing high cohesion and high coupling [3].	5
Fig. 3 Example of high cohesion in a class, taken from [2, p. 140].	6
Fig. 4 Illustration representing a software system with low cohesion and low coupling [3].	7
Fig. 5 Example of information hiding and encapsulation for databases with Views, own work from [9].	8
Fig. 6 Example of information hiding and abstraction with a method to determine if a Connect 4 game is a draw, own work from [4].	9
Fig. 7 Illustration of the concept of encapsulation (image taken from [15]).	10
Fig. 8 Example of data hiding and encapsulation using private and public access modifiers, own work from [4].	11

1. Abstract

This report introduces the four key design principles of high cohesion, low coupling, information hiding, and data hiding. These design principles are followed with the goal of creating maintainable, testable, and readable code that can be easily changed. They can be applied to any software engineering project and are core practices for Object Oriented Programming (OOP) [1].

With the design principles outlined in this report, the aim is to produce clean code [2] with high cohesion within modules and low coupling between modules, as shown in Fig. 1.

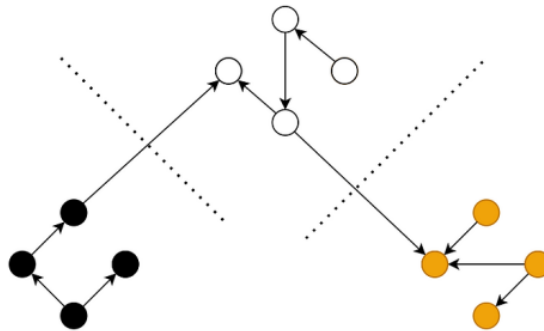


Fig. 1 Illustration showing a 'clean code' implementation with low coupling and high cohesion [3].

Several of the examples are taken from part II of this assignment [4], which is available at:
<https://github.com/mxkoder/Sem1-Assign2-Part2-Connect4-AB> .

2. High Cohesion

High cohesion involves structuring a program so that the sections of code that relate to each other are kept close together [5].

A measure of cohesion is to assess the call relationships between methods in a class [6], and their use of class variables. In a maximally cohesive class, the instance variables and methods in a class are co-dependent and form a logical unit [2, p. 140].

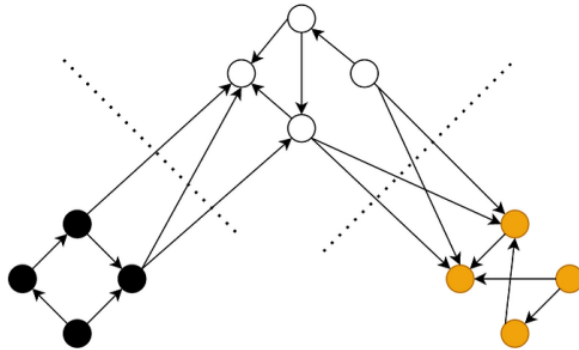


Fig. 2 Illustration showing high cohesion and high coupling [3].

In a well-designed software engineering project, the aim is usually to have maximally high cohesion (see Fig. 2), which makes the codebase easier to navigate and more maintainable [7]. In Fig. 2 coupling is also high, and the ideal system design to work towards is shown in Fig. 1 with low coupling and high cohesion. Organising code cohesively also leads to reduced repetition and helps identify which subsets of methods share variables. This process can then be used to identify subsets of methods to refactor out into smaller classes, making the code more modular [2, p. 141].

Separation of concerns is a related software engineering principle to cohesion and involves breaking up complex problems into simpler sub sections that deal with each aspect of the problem separately [6, p. 139]. Applying the principles of information hiding, abstraction and separation of concerns will help implement high cohesion [6, p. 124]. A disadvantage of working towards high cohesion is that this usually increases the number of lines of code in a project [2], but this is offset by improved readability and maintainability.

```
public class Stack {  
    private int topOfStack = 0;  
    List<Integer> elements = new LinkedList<>();  
  
    public int size() {  
        return topOfStack;  
    }  
  
    public void push (int element) {  
        topOfStack++;  
        elements.add(element);  
    }  
  
    public int pop() throws PoppedWhenEmpty {  
        if (topOfStack == 0) {  
            throw new PoppedWhenEmpty();  
        }  
  
        int element = elements.get(--topOfStack);  
        elements.remove(topOfStack);  
        return element;  
    }  
}
```

Fig. 3 Example of high cohesion in a class, taken from [2, p. 140].

An example of high cohesion is shown above in Fig. 3, with the methods *push* and *pop* using both instance variables in the class.

3. Loose Coupling

The concept of coupling is related to cohesion, with coupling describing the strength of relationships between members of different classes, and cohesion being a measure of the strength of relationships between members within the same class [7]. Stable systems have high cohesion and low coupling [8], as shown in Fig. 1. A representation of a system with low coupling and high cohesion is shown below in Fig. 4, where there are few connections between separate parts of the program. There are different types of coupling, with examples and definitions outlined in Tab. 1.

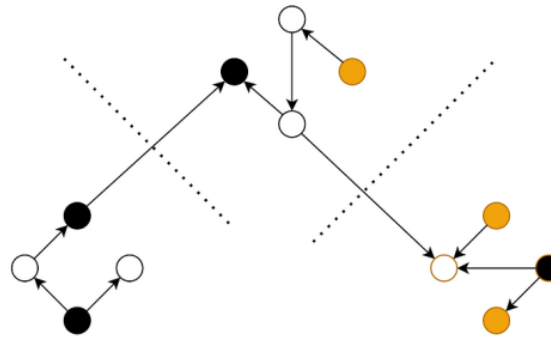


Fig. 4 Illustration representing a software system with low cohesion and low coupling [3].

Systems with low coupling are easier to test, as elements can be isolated and tested independently [2, p. 150]. If modules are less dependent on each other, it also makes code easier to understand, debug, and maintain, as changes can be made without generating a cascade of alterations needing to be made in other parts of the program [5].

Type of coupling	Definition
Implementation Coupling	If the implementation of one module depends on the implementation of another module [8, p. 18].
Temporal Coupling	When there are dependencies on synchronous calls, for example HTTP requests. Can be reduced by caching information, or asynchronous programming [8, p. 22].
Deployment Coupling	When large systems need to be redeployed even when small changes are made to internal modules. Can be reduced by decomposing a system into smaller independently deployable services [8, p. 23].
Domain Coupling	The dependencies between services in the domain. For example, if a delivery needs to be made, the service will first need to know what the customer ordered and where to deliver it to [8, p. 24].

Tab. 1 Table defining different categories of coupling [8].

In the shift towards organising functionality into microservices rather than monoliths, having loosely coupled code also brings the advantage of being able to deploy services independently, reducing costs and increasing system stability [8].

The use of views in SQL (see Fig. 5) is an example of low implementation coupling. If for example the way the organisation of the database is structured is altered, the changes can be localised in the database. The `printAllPickUpDropOffLocations` method in the Java program may not need to be altered and can call the same view.

```
private static void printAllPickUpDropOffLocations() {  
    try {  
        ResultSet results =  
Database.databaseQueryCreateStatement("SELECT \n" +  
            "\t*\n" +  
            "FROM\n" +  
            "\tview_locations;");  
  
        String[] stringTitles = tableTitlesToStringArray("id, street,  
city, country");  
        while (results.next()) {  
            System.out.printf("Location ID: %d, Street: %s, City: %s,  
Country: %s\n",  
                results.getInt(stringTitles[0]),  
                results.getString(stringTitles[1]),  
                results.getString(stringTitles[2]),  
                results.getString(stringTitles[3]));  
        }  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
}
```

Fig. 5 Example of information hiding and encapsulation for databases with Views, own work from [9].

In Structured Design, L. Constantine defines the degree of coupling, assessing the complexity and type of connections between modules [5]. For example, if a module references the interface of another module this is considered lower coupling than connecting directly to the complexity within the module. For accessing common data, there is also a sliding scale in strength of coupling:

- Modules can access each other's private data (high coupling),
- Modules use a global data structure (weaker coupling, but still strong as changes to global variables may affect all classes),
- When data is shared through class parameters in methods, this is a weaker form of coupling as design changes cause more localised changes in the system [6, p. 121].

Coupling is directly related to information hiding and abstraction [8], as these design principles apply to hiding complexity within a class behind an interface other classes can interact with.

4. Information Hiding and Abstraction

Information hiding is the principle of compartmentalising the implementation details of a module and hiding complexity from the rest of the system [10, p. 10].

A key aim of information hiding is to modularise sections of code which are likely to change frequently and together, localising changes to be made [11, p. 298]. The term 'Information Hiding' was coined by David Parnas, who also proposed a shift away from organising modules according to steps in the domain process. He instead put forward the idea of structuring module decomposition around design decisions [12], which today is a core OOP approach. If the parts of the system that need to be altered frequently are separated from more static modules, the changes can be made more safely in a localised manner, and the code is easier to maintain [8]. Designing a system following information hiding principles also improves code reusability, driving towards creating generic modules that have applications to other systems [10, p. 10].

Abstraction is a concept related to information hiding and is the practice of separating the interface of an entity from the internal complexity of implementing features [11, p. 298]. Combined with low coupling, classes should then depend only on the interface of another class, not on the details of how the methods are implemented [2, p. 150].

```
public static boolean gameIsADraw (int [][] gameGrid) {  
    if ( !GameWin.haveWinner(gameGrid) && gridIsFilled(gameGrid) ) {  
        return true;  
    }  
    return false;  
}
```

Fig. 6 Example of information hiding and abstraction with a method to determine if a Connect 4 game is a draw, own work from [4].

The *gameIsADraw* method shown in Fig. 6 is an example of information hiding and refers to the *haveWinner* method at the interface of the *GameWin* class. If the rules or the implementation change for how a game win is determined, it does not impact the implementation of the *gameIsADraw* method above, and alterations can be localised to the methods directly affected in the separate *GameWin* class.

5. Data Hiding and Encapsulation

Data hiding is a core OOP principle and focuses on preventing unintended changes or intrusions to data held in a class [13, p. 133]. This is distinct from information hiding, where the key aim is to hide implementation complexity and localise changes to be made for design decisions [12].

Encapsulation is a mechanism to achieve data hiding and control visibility of the inner workings and data in a class or module [11, p. 298]. In OOP, classes contain variables and their associated methods, and the visibility of these can be controlled at the module boundary [8] (see Fig. 7), using for example using access modifiers [14, p. 273] such as 'public' and 'private' in Java.

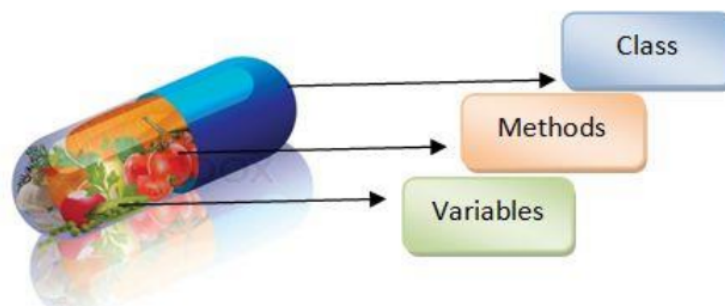


Fig. 7 Illustration of the concept of encapsulation (image taken from [15]).

In Java, variables within a class should be set to private access where possible, so they cannot be inherited or altered outside the class, preventing unintended changes [10, p. 13], while public members can be accessed both inside and outside a class [14, p. 273]. The value of a private variable can then only be modified outside the class using public methods, which helps keep object integrity and prevents accidental changes [16]. Where classes do need to be extended, protected or package access gives another level of controlling data visibility [10, p. 13].

Fig. 8 shows an example of data hiding using encapsulation, where the method of the *SpecialMove* class is public and its variables are private. The *useUp1Move* method is at the interface of the class, and if the system design changes the implementation can be changed locally in the *SpecialMove* class with no or minimal changes in the rest of the program.

```
public class SpecialMove {  
    private int numberInGrid;  
    private Character symbolInGrid;  
    private int numberOfMoves;  
    private String moveCommand;  
  
    public SpecialMove (int numberInGrid, Character symbolInGrid, int  
numberOfMoves, String moveCommand){  
        this.numberInGrid = numberInGrid;  
        this.symbolInGrid = symbolInGrid;  
        this.numberOfMoves = numberOfMoves;  
        this.moveCommand = moveCommand;  
    }  
  
    public boolean useUp1Move () {  
        if(this.numberOfMoves < 1) {  
            return false;  
        }  
  
        this.numberOfMoves -= 1;  
        return true;  
    }  
}
```

Fig. 8 Example of data hiding and encapsulation using private and public access modifiers, own work from [4].

6. Conclusion

The design principles of low coupling, high cohesion, information hiding, and data hiding are interdependent. When applied in a coordinated way, they lead to well structured, modular, and maintainable code.

The trade-offs for implementing these principles are usually increased number lines of code [2], and higher use of computing capacity, which has become a negligible issue with the developments in processing power. These trade-offs are counterbalanced by the time and cost savings with improved debugging, maintainability, testability, and system stability [8] when best practice software design principles are applied.

References

- [1] B. Stroustrup, "What is Object Oriented Programming?". *IEEE Software*, vol. 88, no. 5, pp. 740-7459, 1988.
- [2] R. C. Martin. *Clean Code: A Handbook Of Agile Software Craftmanship*. Boston: Pearson Education Inc., 2008.
- [3] P. Efthymiou. (20 05 2022). "Understanding Software Cohesion & Coupling" [Online]. Available: <https://www.petrosefthymiou.com/post/understanding-software-cohesion-coupling>.
- [4] A. Beviz, "Asignement 2 Part2: Connect 4 Console Game using Java", 2023.
- [5] W. P. Stevens, G. J. Myers and L. L. Constantine, "Structured Design". *IBM Systems Journal*, vol. 13, no. 2, pp. 231 - 256, 1999 [REPRINT].
- [6] O. Vogel, I. Arnold, A. Chughtai and T. Kehrler. *Software Architecture: A Comprehensive Framework and Guide for Practitioners*. New York: Springer, 2009.
- [7] A. Shalloway and J. E. Trott. *Design Patterns Explained: A New Perspective on Object Oriented Design*. Boston: Addison-Wesley Professional, 2004.
- [8] S. Newman. *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. Sebastopol, CA: O'Reilly Media, Inc., 2020.
- [9] A. Burke, A. Beviz, E. Evans and K. Pandya, "Car Rental Company combined assessment for Semester 1" MSc Software Engineering, UWTSD, 2023.
- [10] N. Dale, D. T. Joyce and C. Weems. *Object-Oriented Data Structures Using Java*, 3rd ed, Sudbury, MA: Jones and Barlett Publishers, 2012.
- [11] J. W. McCormick and P. C. Chapin. *Building High Intergity Applications with Spark*. New York: Cambridge University Press, 2015.
- [12] D. L. Parnas, "On The Criteria To Be Used in Decomposing Systems into Modules". *Programming Techniques*, vol. 15, no. 12, pp. 1053 - 1058, 1972.
- [13] D. A. Chowdury. *Object Oriented Programming with C++*. New Dehli: Educreation Publishing, 2018.

- [14] R. Khurana. *Data Structures and Object-Oriented Programming with C++*. New Dehli: Vikas Publishing House PVT Ltd, 2010.
- [15] N. Zaman. (18 11 2022) "Learn Object Oriented Programming Using C#: Part 5" [Online]. Available: <https://www.c-sharpcorner.com/UploadFile/e881fb/learn-object-oriented-programming-using-C-Sharp-part-5/>.
- [16] J. Terra. (30 09 2022). "An Introduction to Data Hiding in C++" [Online]. Available: <https://www.simplilearn.com/tutorials/cpp-tutorial/data-hiding-in-cpp#:~:text=Data%20hiding%20is%20an%20object,or%20unintended%20changes%20and%20intrusions.>

FINAL GRADE

93/100

GENERAL COMMENTS

Instructor

Engagement with Literature Skills (15): 13

Knowledge and Understanding Skills (15): 14

Cognitive and Intellectual Skills (15): 14

Practical Application Skills (50): 48

Transferable Skills for Life and Professional Practice (5): 4

A very well-written, presented and sourced report. A strong covering of the key concepts and a large number of academically credible sources used.

A really good implementation. While it was beyond the scope of this module, you've made a good job of the OO side of things too! Nicely written, well-structured, extremely detailed tests. Great work!

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14
