

Elanwy Evans 1300155 160123

by Elanwy Evans

Submission date: 16-Jan-2023 04:41PM (UTC+0000)

Submission ID: 195237385

File name: 42119_Elanwy_Evans_Elanwy_Evans_1300155_160123_476351_1937968316.docx (416.45K)

Word count: 1839

Character count: 10546



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

University of Wales Trinity Saint David
School of Computing

16/01/2023/ACCA7000
Assignment 1

Programming and Software Testing

By Elanwy Evans (Student ID 1300155)

Table of Contents

High Cohesion & Loose/Low Coupling..... 3

Data & Information Hiding..... 5

Bibliography..... 8

High Cohesion & Loose/Low Coupling

Many studies have demonstrated that structural metrics relating to complexity, coupling and cohesion (CCC) are good at gauging software architecture quality at various stages in the development process, and this architecture is a crucial part of the initial design that determines the quality of the finished software system. Software complexity measurements are usually based on the interaction between a programmer and the program they are developing, which is often influenced by the size and control structure of the program. The higher the complexity, the more difficult the program is for developers to understand [1].

Coupling refers to the measure of external dependency between elements of two or more different software entities (e.g. between methods from two different classes) [2]. High/tight coupling means that different entities rely heavily on each other, making it difficult to understand each entity in isolation. Because of their dependence on each other, they become difficult to re-use, as changes made to one entity must also be reflected in the other [1].

Cohesion is a measure of internal dependency between elements within an entity (e.g. methods within a class) [2] and indicates how much functionality a single entity contributes to the whole software system. Cohesion represents the degree to which a section of a code base forms a logically single, atomic entity. For example, a class with high cohesion will contain highly related methods that, together, contribute towards a section of a software system [1]. A class with low cohesion will include methods that have no logical relationship with each other, which is a result of poorly chosen boundaries (see figure 1) [3].

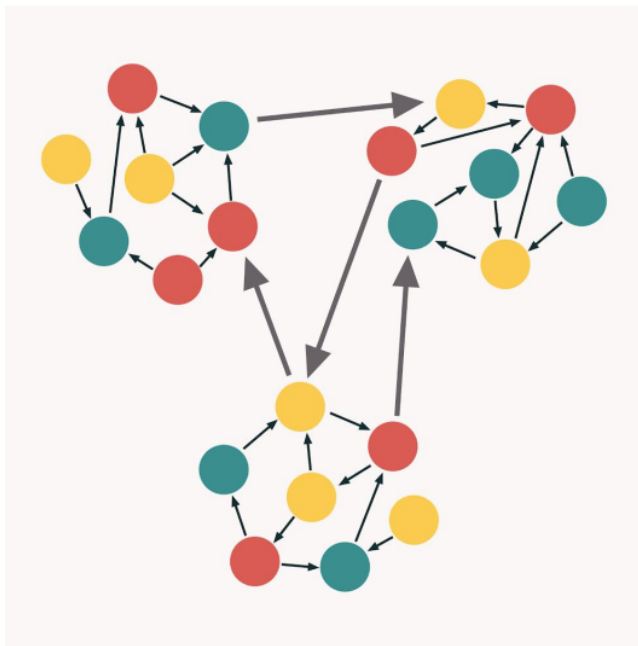


Figure 1: Visual Representation of Poorly Chosen Boundaries (circles of the same colour denote units that are related to each other) [3]

In object-oriented systems, it is desirable to create a software system with high cohesion and low coupling (see figure 2) to try and preserve the quality and maintainability of the system for as long as possible. If these characteristics aren't prioritised early in the design stage, the quality of the system will deteriorate rapidly, leaving the system vulnerable to problems and the source code will become difficult, expensive and time-consuming to maintain. As a result of trying to maintain this complex system, it becomes more likely that developers will miss sections of code that need updating after changes have been made to their highly coupled entities. Bugs are also likely to be introduced unintentionally as the code becomes more difficult to read, which will gradually erode the system's integrity [4]. These vulnerabilities become perfect targets for malicious users, who could access confidential information and/or cause severe damage to the software system [1].

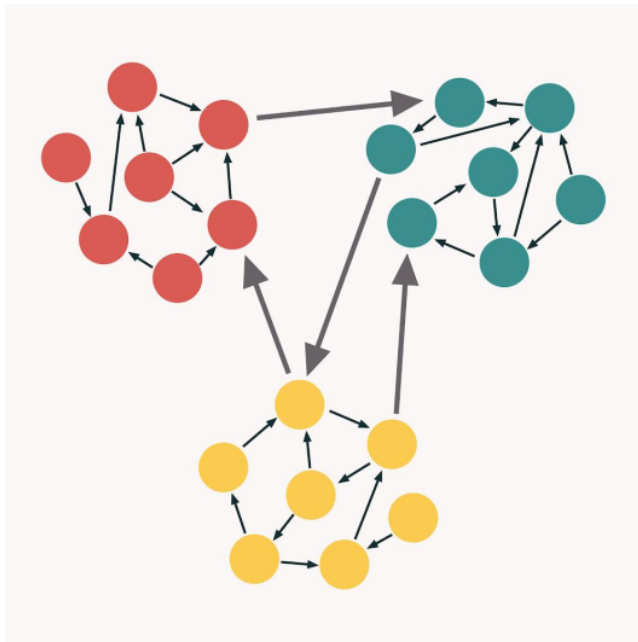


Figure 2: Visual Representation of an Ideal Software System with High Cohesion and Low Coupling [3]

The barrier to achieving high cohesion and low coupling is known as a 'code smell' (which results in low cohesion and high coupling) and can only be corrected by code refactoring [4]. Several tools and techniques can be used to detect code smells, such as the Jaccard distance coefficient used by JDeodorant, the move method used by JMove and Methodbook, HIST, inCode etc. Most of these approaches use complex calculations to measure the similarities between entities and methods, which helps developers group similar source code components together, and reduce coupling by placing dissimilar components into separate modules [4].

During the refactoring process, it is often the case that developers focus too much on low coupling as it is conceptually easier to understand, neglecting cohesion as a result. Entities

become so segregated from each other that they become incomprehensible and incohesive (see figure 3) [3].

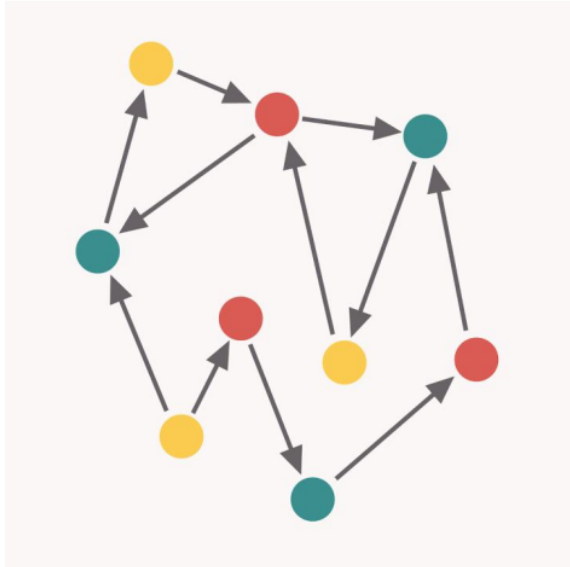


Figure 3: Visual Representation of a Software System after Excessive Decoupling [3]

Data & Information Hiding

The sources referenced in this report use the terms 'data hiding' and 'information hiding' interchangeably as no reputable sources could be found to evidence any differences between the terms. The term 'element' will be also used to describe a module, method, function, variable or a segment of code.

'Information hiding' is an object-oriented term [5] first described by David Parnas in 1972, which describes the principle of segregating sections of a software system most likely to change, to protect the rest of the code base from requiring modifications if a change did occur [6]. Parnas explained that this can be achieved through modularization of a software system, meaning that a new module can be created with little knowledge of the code contained within other modules (thus achieving loose coupling). Therefore, if changes are made to the code inside one module, it protects the rest of the system from also having to be changed [6].

Data hiding is commonly associated with the terms abstraction and encapsulation. Encapsulation describes a segment of code that has boundaries around it, which can be used to manage access to different elements of a code base [7].

Abstraction describes the separation of a code element's interface from its implementation details, which means a developer only needs to understand the interface and which parameters to pass in (if there are any) [8][9]. Object-oriented languages such as Java and C++ utilise abstraction and encapsulation techniques to create user-defined types known as classes [7][9].

A class is defined as a group of objects that behave similarly or have similar characteristics [5]. There are 3 types of access specifier that can be assigned to a class and the members within it: private, public and protected [10]. Private elements can only be accessed by other elements defined within the same class and are hidden from other classes [11][12].

For example, a private variable within class A will be visible to other elements within class A, but invisible to class B.

Public elements can be accessed from anywhere [10][11]. It is generally best practice to keep data private [9], however if data within an element needs to be accessed and/or modified by another class, it needs to be declared public [5][10].

The example below (figure 4) demonstrates that if you have a private method (createBoard()) inside a public method (playGame()), any section of the program (Main class) can call the public method which will execute the private method, but the private method will not be directly accessible by the Main class.

```
//Public Game class
public class Game {

    //public method that calls the private 'createBoard()' method
    public static void playGame() {
        createBoard();
    }

    //private method for creating a board
    private static void createBoard() {
        //code to create a board
    }
}

//Main class which calls the 'playGame()' method within the Game class
public class Main {
    public static void main(String[] args) {

        Game.playGame();

    }
}
```

Figure 4: Example Java Code Snippet Demonstrating the Use of Public and Private Access Specifiers

Protected elements are only available to classes defined within the same package (and outside the package but only via inheritance) [10]. In the example below (figure 5), the Board class is defined with the 'org.board' package and the Game class is defined within the 'org.example' package. A package acts as a logical separation of concerns [13] as independent elements of the application are separated into distinct sections. The Board package isolates code relating to the creation of a board from the logic around the Game class, which would include methods such as tracking winning conditions etc.

```
// Board class is defined within the 'org.board' package
package org.board;
public class Board {
    // Variables are protected so can't be accessed from a class outside
    'org.board' package
    protected static int MAX_COLUMN = 5;
    protected static int MAX_ROWS = 6;

    public static void printBoard() {
        String[][] grid = new String[MAX_COLUMN][MAX_ROWS];
        createBoard(grid);
    }

    protected static String[][] createBoard(String[][] board) {
        //code to create a board
        return board;
    }
}

// Game class is defined within the 'org.example' package
package org.example;
// Game class imports Board class from the 'org.board' package
import org.board.Board;

//Public Game class
public class Game {
    //Can access as the 'public' access modifier is used
    public static void main(String[] args) {
        Board.printBoard();
    }
}
```

Figure 5: Example Java Code Snippet Demonstrating the use of a Protected Access Specifier

Information hiding is an important part of OOP as:

- It can be used to hide confidential information from users and is essential for data privacy [5]
- It protects information from accidental or deliberate alterations by controlling access to elements within a class [5] [11]
- It abstracts complicated code away from developers, making the code base more readable and easier to maintain [14]
- It increases cohesion by using encapsulation to create boundaries between unrelated segments of code, whilst grouping similar sections of code together [9]
- It reduces coupling by limiting interdependencies and encourages atomicity [5]

Together, low coupling, high cohesion and information/data hiding are important traits of successful object-oriented software systems and, together, provide high quality, maintainability and security to the final product.

Word count: 1474

Bibliography

- [1] I. Chowdhury and M. Zulkernine, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities," *Journal of Systems Architecture*, vol. 57, no. 3, pp. 294–313, Mar. 2011, doi: 10.1016/j.sysarc.2010.06.003.
- [2] M. Alzahrani, "Extract Class Refactoring Based on Cohesion and Coupling: A Greedy Approach," *Computers*, vol. 11, no. 8, p. 123, Aug. 2022, doi: 10.3390/computers11080123.
- [3] V. Khorikov, (2015), *Cohesion and Coupling: the difference*. [Online]. Available: <https://enterprisecraftsmanship.com/posts/cohesion-coupling-difference/>
- [4] Md. M. Rahman, R. R. Riyadh, S. M. Khaled, A. Satter, and Md. R. Rahman, "MMRUC3: A recommendation approach of *move method* refactoring using coupling, cohesion, and contextual similarity to enhance software design," *Software: Practice and Experience*, vol. 48, no. 9, pp. 1560–1587, May 2018, doi: 10.1002/spe.2591.
- [5] J. Terra, (2022), *An Introduction to Data Hiding in C++*. [Online]. Available: <https://www.simplilearn.com/tutorials/cpp-tutorial/data-hiding-in-cpp>
- [6] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972, doi: 10.1145/361598.361623.
- [7] Tutorialspoint, *Data Encapsulation in C++*. [Online]. Available: https://www.tutorialspoint.com/cplusplus/cpp_data_encapsulation.htm
- [8] Study.com, (2017), *Data Abstraction: Definition & Example*. [Online]. Available: <https://study.com/academy/lesson/data-abstraction-definition-example.html>
- [9] T. Y. Han, S. J. Hong, S. H. Hwang, B. C. Lee, and C. S. Byun, "Object-Oriented Programming in the Development of Containment Analysis Code," in *Proceedings of the KNS spring meeting*, KNS: Republic of Korea, 2009, pp. 721–722.
- [10] M. Mohammad, "'Study of Java Access Control Mechanism,'" *IOSR Journal of Computer Engineering*, vol. 15, no. 1, pp. 59–66, 2013, doi: 10.9790/0661-1515966.
- [11] S. Stevenson, (1997), *Modularity and Information Hiding*. [Online]. Available: <https://people.computing.clemson.edu/~steve/CW/241/241hints/node16.html>
- [12] S. M. Biju, "Difficulties in Understanding Object Oriented Programming Concepts," *Lecture Notes in Electrical Engineering*, pp. 319–326, Aug. 2012, doi: 10.1007/978-1-4614-3535-8_27.
- [13] N. Natesan, (2019), *How to implement Design Pattern – Separation of concerns*. [Online]. Available: <https://www.castsoftware.com/blog/how-to-implement-design-pattern-separation-of-concerns>

[14] P. Asagba and E. Ogheneovo, "A Comparative Analysis of Structured and Object-Oriented Programming Methods," *Journal of Applied Sciences and Environmental Management*, vol. 12, no. 4, Jun. 2010, doi: 10.4314/jasem.v12i4.55217.

FINAL GRADE

82/100

GENERAL COMMENTS

Instructor

Engagement with Literature Skills (15): 10

Knowledge and Understanding Skills (15): 13.5

Cognitive and Intellectual Skills (15): 13.5

Practical Application Skills (50): 41

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

A good number of sources used, but trending a little heavily towards non-academic web sources. I like the diagrammatic, graph-based approach to explaining the concepts of coupling and of cohesion. A very good report overall.

A very good implementation, well-concieved and created. An appropriate number of unit tests, and a good coverage of the brief overall. Well done, Elanwy.

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6
