

Assignment

by Kieran Ayres

Submission date: 15-Jan-2023 11:41PM (UTC+0000)

Submission ID: 195098288

File name: 2962_Kieran_Ayres_Assignment_476351_1350133266.docx (1.01M)

Word count: 2478

Character count: 13170



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

Java Programming, Testing and Development

By Kieran John Ayres

Table of Contents

Table of Contents

Table of Contents..... 2

Table of Figures 3

Introduction 4

1. Important Concepts in Software Development 5

 1.1. High Cohesion 6

 1.2. Loose Coupling..... 8

 1.3. Information and Data Hiding..... 9

Conclusion 11

References..... 13

Appendix 16

Table of Figures

Figure 1 - Levels of Software Cohesion..... 6

Figure 2 - Attempting High Cohesion and Loose Coupling..... 7

Figure 3 - Public Classes in Java..... 9

Figure 4 - Private Class in Java 9

Figure 5 - Explanation of the Class being Private 10

Introduction

The focal point of this report is to research and discuss four important concepts within Software Development. The concepts themselves will be in the field of software design and architecture. This expands on the reasons particular design methods are applied.

1. Important Concepts in Software Development

Software development compiles of several concepts and practices which are applied when software is built. Coding principles are of high importance as they enforce a development team to write clean and robust code [1]. This report will discuss the following metrics to assess quality [2] within software development:

- High Cohesion
- Loose Coupling
- Data Hiding
- Information Hiding

The following concepts are applied to a system to overall improve the architecture and integrity. These are commonly applied in the popular construct; Object-Oriented Programming (OOP). A way of grouping data and functions into classes, which can be classified as blueprints of objects as they contain variables and functions that can be used in a software program [3]. When an instance of the class is declared, it can be considered as an object [3].

1.1. High Cohesion

The term 'cohesion' implies a group of objects, that work well together in close proximity [4]. This is an important concept to build a strong Software system [5] [6]. High cohesion refers to the concept that classes that support similar functional purposes are in the same area of the codebase's structure. For example, if a developer was to build a calculator, a singular operation in that calculator could be a class, such as a class to apply the addition of numbers, if it was highly cohesive, there would also be a similar class in that same group of classes for other mathematical operations, such as subtraction or multiplication. High cohesion also aims to improve the overall structure of the codebase [7] [8]. It was also found to improve reusability of code [1], reusability in software development refers to code that can be used for similar functionalities across a codebase by parsing in different values to the functions. There are several levels of cohesion and can be seen in the below figure.

Rank	Cohesion type	Quality
6	Functional cohesion	Strongest (good)
5	Sequential cohesion	
4	Communication cohesion	
3	Procedural cohesion	
2	Temporal cohesion	
1	Logical cohesion	
0	Coincidental cohesion	Weakest (bad)

Figure 1 - Levels of Software Cohesion

Coincidental cohesion, the lowest form, is usually not designed but are grouped together which can cause confusion as it hasn't been planned [5] [6]. Whereas a mid-level of cohesion refers to procedural cohesion, this refers to classes being grouped together as they perform a particular set of tasks in a program [5] [6]. The highest level of cohesion, Functional cohesion aims to assist in building groups of classes which provide singular functionality, these can largely be task-based functionality [5] [7]. A class can be assessed against this, and if it has near single functionality, it is classified as highly cohesive [7]. Having high cohesion within a system, can support scalability as the codebase would be easier to edit when the packages of classes are grouped to similar functionalities. One method to ensure good practices such as high cohesion are met, is by writing highly readable code, this reduces the chances of re-writing functionality when revisiting the codebase after sometime, due to understanding what poorly written code is doing [9]. This further enforces the coding principle *Don't Repeat Yourself*, or DRY [10]. Code which doesn't repeat improves reusability across a codebase.

This also refers to a coding principle, known as the Single Responsibility Principle, commonly abbreviated to SRP [11]. Furthermore, SRP dictates that if a class does have more than a single responsibility in a software project, then it overall contributes to high coupling [12], this leads into the section 1.2 – *Loose Coupling*. Implementing SRP will build high cohesion amongst classes [5], moreover, this will improve maintainability which will further reduce the faults that will occur in the system [6]. The aim of a project is to have loose coupling and high cohesion, as research showed that this was contributed to a more performant system [2]. These should be considered at the application design stage.

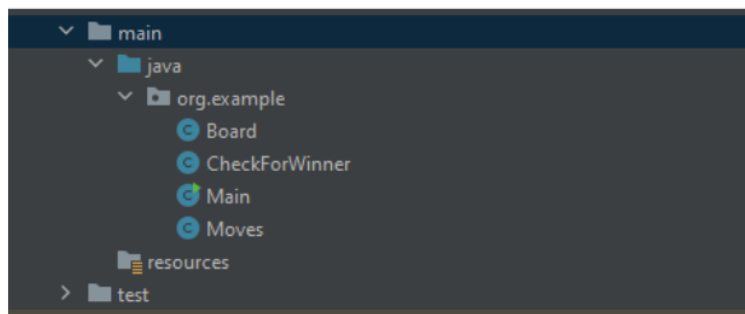


Figure 2 - Attempting High Cohesion and Loose Coupling

The above screenshot is showing the attempt at keeping responsibility separate in the codebase.

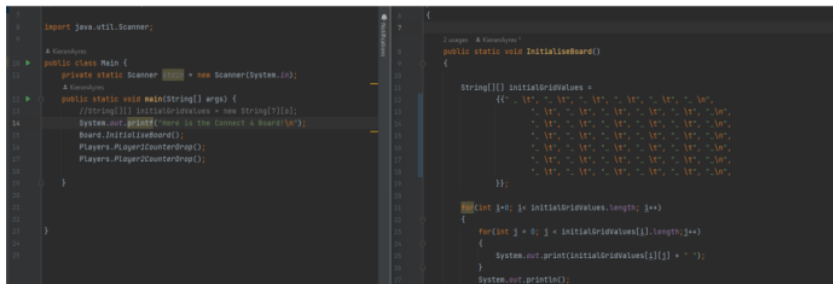
1.2. Loose Coupling

Coupling is a design metric which refers to the level of dependence that a class has on other classes within a codebase [1] [13] [14]. Loose coupling specifically refers to the concept that how software classes connect to one another and that they should have a minimal level of dependency on one another [2]. By doing this, if a change was made to one class, it wouldn't affect many others. If the minimal dependency level was enforced, loose coupling helps reduce the 'ripple effect', this refers to one change causing a chain of problems in the application [15] [16]. This improves the maintainability of the application and therefore improves the system's architecture [5], thus, having high cohesion and loose coupling. Loose coupling also improves the ability to Unit test, due to the lack of dependencies that the class relies on [17]. Which can also build confidence in a system. Loose coupling also refers to another coding principle known as the *Open Close Principle*, this means that classes should be closed for modification but open for extension, this minimises the chance of change happening as it is encouraged to add new code for purpose of extension. This promotes loose coupling as it is closed for modification as in the case of a poorly designed system, a single change can result in changes being needed across several classes [12]. Research alludes that it is difficult to implement the coupling method. Metrics such as coupling are often assessed against factors such as reusability of code in the system [14]. Whereas a system with high coupling would be the opposite, which means it would have high level of connectivity between classes with classes having high dependency on one another [18].

The overarching idea of loose coupling is scalability and maintainability. An example of loosely coupled based architecture is a Microservice architecture, which consists of small services which are independently deployable [19] but would operate as one service through communicating with one another through a layer of the architecture [19]. Their smooth integration implies high cohesion, making them more scalable

1.3. Information and Data Hiding

Research referred to Data Hiding and Information Hiding as highly similar concepts, which is the idea of hiding data members to block access from other classes in a program, leading to a greater level of security [3]. Access levels would be granted through access modifiers, which means whether a class is private, or public [20] [21] [22] [23]. Which would further enforce the DRY principle [10]. These are done through a class constructor and are called through *getters and setters*. These can also be accessed through the method of inheritance, a method in which a new class can inherit all the parent class's functionality [24], further enforcing code that's 'DRY' [3]. However, this would create tight coupling, as a change to the parent could result in the child of the class requiring a change. Implementation of private classes shields access from other classes, thus, promoting loose coupling, as by blocking access to methods across the code files, there is less code dependent on one another [20]. This is achieved through encapsulation [3] [20]. As implied, encapsulation can block access to other classes. This is done through setting variables or functions as private, if they are set as public, they become accessible [3] [25], research shows that this improves performance [25], while it's also shown that encapsulation hides complexity [3]. This data is then accessed through a technique such as developing a function to call them. An example of a function, calling a class which has public access can be seen below. Main.Java is calling a public class from Board.Java.



```

1 import java.util.Scanner;
2
3 public class Main {
4     private static Scanner scanner = new Scanner(System.in);
5
6     public static void main(String[] args) {
7         //String[][] initialGridValues = new String[10][10];
8         System.out.println("Here is the Connect 4 Board");
9         Board.initialiseBoard();
10        Players.Player2CounterUp();
11        Players.Player2CounterUp();
12    }
13 }

```

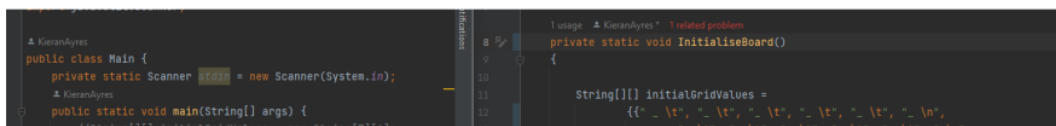
```

1 //Connect 4 Board
2 public static void initialiseBoard()
3 {
4     String[][] initialGridValues =
5     {
6         {"", "", "", "", "", "", ""},
7         {"", "", "", "", "", "", ""},
8         {"", "", "", "", "", "", ""},
9         {"", "", "", "", "", "", ""},
10        {"", "", "", "", "", "", ""},
11        {"", "", "", "", "", "", ""},
12        {"", "", "", "", "", "", ""},
13    };
14
15    for (int i = 0; i < initialGridValues.length; i++)
16    {
17        for (int j = 0; j < initialGridValues[i].length; j++)
18        {
19            System.out.print(initialGridValues[i][j] + " ");
20        }
21        System.out.println();
22    }
23 }

```

Figure 3 - Public Classes in Java

The advantage to encapsulation is that it protects the classes so they can't be modified. For example, in an application, user's data wouldn't be made available to be called or modified. As can be seen, if the above function to private, it blocks it from being called in other files.



```

1 //Connect 4 Board
2 private static void initialiseBoard()
3 {
4     String[][] initialGridValues =
5     {
6         {"", "", "", "", "", "", ""},
7         {"", "", "", "", "", "", ""},
8         {"", "", "", "", "", "", ""},
9         {"", "", "", "", "", "", ""},
10        {"", "", "", "", "", "", ""},
11        {"", "", "", "", "", "", ""},
12        {"", "", "", "", "", "", ""},
13    };
14
15    for (int i = 0; i < initialGridValues.length; i++)
16    {
17        for (int j = 0; j < initialGridValues[i].length; j++)
18        {
19            System.out.print(initialGridValues[i][j] + " ");
20        }
21        System.out.println();
22    }
23 }

```

Figure 4 - Private Class in Java

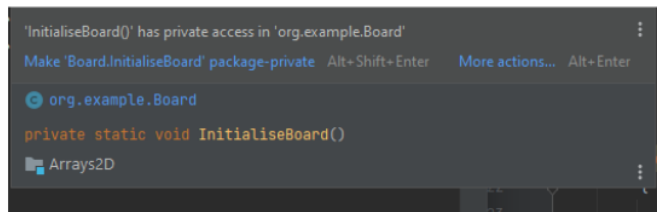


Figure 5 - Explanation of the Class being Private

Another way in which private methods may be accessed is through different access levels that are integrated into a system [20]. A user with a higher level would have a higher level of access. One of the advantages of implementing information hiding is that it improves the structure of the system's security. Information hiding is important as it enforces the architecture of a system to be designed with thought, while actively being implemented to protect classes against other classes that they "don't directly use" [26].

Conclusion

In conclusion, These are all important metrics in an application as many codebases are active for many years, thus, being understandable is fundamental to how the system can survive the ever-changing nature of the Software industry. Ultimately, implementing these software design principles in the system can result in performance improving to an optimal level, while improving security while making the code reusable, and maintainable [5]. The last conclusion that could be drawn from the research is that the metrics all link to the popular design choice, to separate the concerns in an application. A system which implements with the separation of concerns will mean that the program will be divided into distinct areas [27] . It is linked to information hiding as a good, structured codebase could have encapsulated code [23]. Interestingly, this could also be considered to have high cohesion due to it being separated accordingly, while being loosely coupled as the encapsulation will assist in minimize classes being dependent to one another.

References

- [1] R. K. Mandeep Kaur, "Improving the Design of Cohesion and Coupling Metrics for Aspect Oriented Software Development," *International Journal of Computer Science and Mobile Computing*, vol. 4, no. 5, p. 8, 2015.
- [2] P. K. B. Parul Ghandi, "Optimization of Object-Oriented Design using Coupling Metrics," *International Journal of Computer Applications*, vol. 27, no. 10, p. 4, 2011.
- [3] M. R. S. RAUT, "Research Paper on Object-Oriented Programming (OOP)," *International Research Journal of Engineering and Technology (IRJET)*, vol. 7, no. 10, p. 5, 2020.
- [4] C. Dictionary, *Cohesion*, N/A: N/A, Cambridge Dictionary.
- [5] M. H. Habib Izadkhah, "Class Cohesion Metrics for Software Engineering: A Critical Review," *Computer Science Journal of Moldova*, vol. 25, no. 1, p. 32, 2017.
- [6] P. N. P. S. R. Annushri Sethi, "Analysis of Cohesion and Coupling Metrics for Object Oriented System," *International Journal of Scientific Research in Science, Engineering and Technology*, vol. 2, no. 2, p. 6, 2016.
- [7] J. K. C. Amit Rathee, "Improving Cohesion of a Software System by Performing Usage Pattern Based Clustering," in *6th International Conference on Smart Computing and Communications*, Kurukshetra, India, 2017.
- [8] M. H. Y. Z. Y. Y. Matheus Paixao, "An Empirical Study of Cohesion and Coupling: Balancing Optimisation and Disruption," in *IEEE Transactions In Evolutionary Computation*, N/A, 2017.
- [9] T. Sedano, "Code Readability Testing, an Empirical Study," in *International Conference on Software Engineering Education and Training*, Dallas, TX, USA, 2016.
- [10] R. S. P. C. E. W. Ivan Cabezas, "Using Software Engineering Design Principles as Tools for Freshman Students Learning," in *IEEE Frontiers in Education Conference*, Uppsala, 2020.

- [11] A. A. e. al, "Applying the Single Responsibility Principle in Industry: Modularity Benefits and Trade-offs," in *Proceedings of the Evaluation and Assessment on Software Engineering*, N/A, 2019.
- [12] S. I. H. Harmeet Singh, "Effect of SOLID Design Principles on Quality of Software: An Empirical Assessment," *International Journal of Scientific & Engineering Research*, vol. 6, no. 4, p. 4, 2015.
- [13] A.-C. J. E. R. T.-B. C. Ruiz Cenicerros JA, "The external and data loose coupling for the integration of software units: a systematic mapping study," *PeerJ Comput Sci.*, vol. N/A, no. N/A, p. 33, 2021.
- [14] S. S. R. Saurabh Tiwari, "Coupling and Cohesion Metrics for Object-Oriented Software: A Systematic Mapping Study," in *Innovations in Software Engineering Conference*, N/A, 2018.
- [15] C. Taube-Schock, "Patterns of Change: Can modifiable software have high coupling," The University of Waikato, Hamilton, 2012.
- [16] J. B. Kevin P. Scheibe, "Systematic Risk and the Ripple Effect in the Supply Chain," in *Handbook of Ripple Effects in the Supply Chain*, N/A, Springer, 2019, p. 15.
- [17] D. S. M. Biju, "Dependency Injection for loose coupling of Objects," *International Journal of Scientific & Engineering Research*, vol. 3, no. 5, p. 5, 2012.
- [18] R. J. W. I. W. Craig Taube-Schock, "Can We Avoid Coupling," in *ECOOP 2011 - Object-Oriented Programming*, Lancaster, 2011.
- [19] V. L. C. P. Davide Taibi, "Architectural Patterns for Microservices: a Systematic Study," in *8th International Conference on Cloud Computin and Services Science*, Madeira, 2018.
- [20] D. D. Ross Harmes, "Encapsulation and Information Hiding," in *Pro JavaScript Design Patterns*, N/A, Apress, 2008, p. 16.
- [21] P. G. G. C. K. & T. R. Klaus Ostermann, "Revisiting Information Hiding: Reflections on Classical and Nonclassical Modularity," in *European Conference on Object-Oriented Programming*, N/A, 2011.

- [22] A. T. Friedrich Steimann, "From Public to Private to Absent: Refactoring JAVA Programs under Constrained Accessibility," in *European Conference on Object-Oriented Programming*, N/A, 2009.
- [23] H. Y. J. C. R. Peter W. O'Hearn, "Seperation and Information Hiding," in *31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming*, N/A, 2004.
- [24] A. K. G. Sujeet Kumar, "A Brief Study on Inheritance," *International Journal of Research*, vol. 1, no. 10, p. 4, 2014.
- [25] F. B. & M. B.-T. G. Omar Pizarro-Vasquez, "Encapsulation Component and its Incidence into Scientific Software Performance," in *Conference Proceedings in Communication, Smart Technologies and Innovation for Society*, N/A, 2021.
- [26] R. C. Martin, "Information Hiding," in *Clean Architecture: A Craftsman's Guide to Software Structure and Design: A Craftsman's Guide to Software Structure and Design*, N/A, Addison-Wesley, 2017, p. 1.
- [27] H. M. R. M. Hamid Mcheick, "A Case Study of Distributed and Evolving Applications Using Seperation Of Concerns," in *International Conference on Software and Data Technologies*, N/A, 2007.
- [28] G. C. D.-L. M. Z. O.-M. Istvan Gergely Czibula, "An aggregated coupling measure for the analysis of object-oriented software systems," *The Journal of Systems and Software*, vol. N/A, no. N/A, p. 20, 2019.

Appendix

Assignment

GRADEMARK REPORT

FINAL GRADE

85 / 100

GENERAL COMMENTS

Instructor

Engagement with Literature Skills (15): 15

Knowledge and Understanding Skills (15): 12

Cognitive and Intellectual Skills (15): 14

Practical Application Skills (50): 40

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

A huge number of sources used in a well-written and structured report. Well done, Kieran.

A good effort with the implementation - terse code written. I think you could've taken this further, but certainly meeting the brief.

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

PAGE 15

PAGE 16
