

# 2222124-ACCA7000-assignment

*by* Callan Birch-Williams

---

**Submission date:** 16-Jan-2023 08:06PM (UTC+0000)

**Submission ID:** 195260210

**File name:** 42481\_Callan\_Birch-Williams\_2222124-ACCA7000-assignment\_476351\_566303575.pdf (206.33K)

**Word count:** 1638

**Character count:** 11186

UWTSD

DVLA: MSc Software Engineering (Web Development)

ACCA7000

Programming, Testing and Development

**Callan Birch-Williams, 2222124**

## Contents

|                                                                               |   |
|-------------------------------------------------------------------------------|---|
| Introduction .....                                                            | 3 |
| High Cohesion and Loose Coupling .....                                        | 3 |
| Cohesion .....                                                                | 3 |
| Coupling .....                                                                | 4 |
| Relationship .....                                                            | 4 |
| Benefits .....                                                                | 5 |
| Information Hiding .....                                                      | 5 |
| Data Hiding .....                                                             | 5 |
| Example .....                                                                 | 6 |
| Bibliography .....                                                            | 8 |
| <br>                                                                          |   |
| Figure 1: Example class with no Data Hiding. ....                             | 6 |
| Figure 2: Example class with Information Hiding .....                         | 6 |
| Figure 3: Example class demonstrating the benefit of Information Hiding. .... | 7 |

## Introduction

The purpose of this document is to outline and evaluate the following software design principles and their importance in software development:

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

## High Cohesion and Loose Coupling

The concepts of cohesiveness and coupling in software development were first notably defined in an article titled “Structured Design” published by Larry Constantine, Glenford Myers and Wayne Stevens, in consideration of designing systems such that a component can be considered, implemented and changed easily with minimal consideration for impact on the other components of a system [1]. Major ideas outlined in the article are accredited to Larry Constantine and he is widely considered in software development the father of these concepts.

While a “component” within the Object-oriented programming paradigm is commonly interpreted as a class, it should be noted that the concept of cohesion and coupling is applicable at all levels of software development.

## Cohesion

Cohesion as a concept within software development represents the degree to which the elements of some component are related and belong together [2]. It can also be considered the number of relationships among elements within the same component. High cohesion suggests that the elements within a component are closely related and belong together to fulfil a well-defined purpose. Logically, low cohesion would indicate that the elements within a component are loosely related and don’t necessarily contribute to the same purpose.

It should be noted that the cohesiveness of a software component is not binary, but rather follows an ordinal scale. There are different levels of cohesion distinguishable by different associative principles [2].

Within the Object-oriented programming paradigm, the Single Responsibility Principle (A component of SOLID principles of Object-oriented class design) is commonly used as a stricter catalyst for achieving high cohesion [3].

## Coupling

Coupling as a concept within software development represents the degree to which some component depends on other components. It can also be considered the number of relationships a component has with other components. Tight coupling suggests that a component's functionality depends on many other components, whereas loose coupling suggests that a component depends on few.

Like cohesion, the degree of coupling of some software component can both be treated as a broad classification, or be evaluated against an ordinal scale, which differs depending on the programming paradigm being used. Unlike cohesion however, some concrete metrics have been established to measure the degree of coupling where applicable [4].

It is widely accepted that loose coupling is an indication of good software design. However, achieving a perfectly decoupled system is unrealistic and often undesirable in practice [5] [6].

## Relationship

Cohesion and coupling are interrelated concepts in software development. The higher the cohesion of some component in a software system, the lower the coupling between components will be [2].

A software component that adheres to the concepts of high cohesion and low coupling, when used appropriately, increases the likelihood that a component can be reused by itself and that a system can be understood, modified, and scaled more easily [7].

A highly cohesive component that also has a large degree of coupling is usually an indication of bad software design. In Object-oriented programming this is often considered an anti-pattern known as a god object, a component that depends on many other components, has unrelated functionality or some combination of the two [8].

A tightly coupled component that has low degree of cohesion is often associated with improperly selected component boundaries. That is, components are grouped by some consideration other than their defined purpose. This is often observable at the project structure level, where components are grouped by their type rather than their relation. In Object-oriented programming, components that are tightly coupled but low in cohesion will often violate the Single Responsibility Principle [3].

A software component that both lacks cohesiveness and is loosely coupled could be an indication of "destructive decoupling" [5]. An idea that is derived from a programmer over-focusing on decoupling a software component when adhering to the concept of high cohesion and loose coupling. The resulting component is decoupled to the degree where it has no clear well-defined purpose.

## Benefits

In summary, some benefits of a software system that adheres to the concept of high cohesion and loose coupling can be defined as follows:

A component that is highly cohesive and loosely coupled is easier to make changes to without affecting other components. This improves the components flexibility, maintainability, and scalability.

A highly cohesive and loosely coupled component is easier for a programmer to understand as it fulfils a well-defined purpose and removes the need for the programmer to understand other components while evaluating it.

High cohesion and loose coupling allow for improved testability as a component that fulfils a well-defined purpose and has minimal external dependencies on other components is easier to evaluate as a single unit.

## Information Hiding

The concept of information hiding was first defined by David Parnas in the interest of modular programming with the intention of reducing the interconnectedness of a system (Which relates to the concept of coupling which would be formally defined later) [9].

Following the definition of the software design concept encapsulation, Parnas himself described information hiding and encapsulation as synonymous terms [10]. However, distinctions between the two concepts are often drawn. Encapsulation may be considered the same as information hiding but is often interpreted as the practical implementation of the information hiding concept, particularly within the Object-oriented programming paradigm [11].

Information hiding as a concept within software development is limiting access to the details of implementation of a component. By exposing only the necessary information of a component to other components, they are less likely to be impacted by changes made within the component.

## Data Hiding

Data hiding is a domain-specific technique used to protect the elements of a component. In Object-oriented programming, this is commonly observable at a class-level and is part of the implementation of information hiding (i.e., It is necessary when trying to achieve information hiding at a class-level). The elements within a component are protected from external components by making the elements hidden using some language-specific modifier. Each element is then only allowed to be modified through methods or functions. This allows a component to control how its elements are modified or accessed.

## Example

The following example classes were constructed in the Java programming language and added to this document using SlidesCodeHighlighter [12] to demonstrate the importance of information hiding.

Consider the Java class shown in Figure 1 as an example of the issues that arise from no information hiding. The details of implementation are not hidden from other components and therefore a future change (For example, changing the storage of monies from pounds to pence) would require changes in other components. This closely relates the previously discussed concept of tight coupling. Additionally, the variable “pounds” is specified with the public access modifier. This means that the class has no control over the access and modification of pounds (No data hiding).

```
public class BankAccount {  
    public int pounds;  
}
```

Figure 1: Example class with no Data Hiding.

By using the private access modifier on “pounds” (Shown in Figure 2 below) and creating a public method to interface with it, “pounds” is now hidden from other components and the class now has control over how it is modified, achieving both data hiding and information hiding in this case.

```
public class BankAccount {  
    private int pounds;  
  
    public void deposit(int pounds) {  
        if (pounds < 0) throw new IllegalArgumentException();  
        this.pounds += pounds;  
    }  
}
```

Figure 2: Example class with Information Hiding.

Figure 3 demonstrates how by hiding the data and implementation details of a component, internal changes can be made without breaking the existing functionality of components that interface with it. External components do not know that monies are now stored in pence, nor do they need to. New functionality can be added without breaking components dependant on existing behaviour (In this example, method overloading is added to support depositing pence, without changing the original pound deposit behaviour).

```
public class BankAccount {  
    private int pence;  
  
    public void deposit(int pounds) {  
        if (pounds < 0) throw new IllegalArgumentException();  
        this.pence += pounds * 100;  
    }  
  
    public void deposit(int pounds, int pence) {  
        if (pounds < 0 || pence < 0) throw new IllegalArgumentException();  
        this.pence += pounds * 100 + pence;  
    }  
}
```

*Figure 3: Example class demonstrating the benefit of Information Hiding.*

## Bibliography

- [1] L. Constantine, G. J. Myers and W. P. Stevens, "Structured Design," *IBM Systems Journal*, pp. 115-139, 1974.
- [2] E. Yourdon and L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, New Jersey, U.S: Prentice Hall, Inc, 1979.
- [3] C. M. Robert, *Agile Software Development, Principles, Patterns, and Practices*, 2002.
- [4] R. Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill Education / Asia, 2008.
- [5] V. Korikov, "Cohesion and Coupling: the difference," *Enterprise Craftsmanship*, 2 September 2015. [Online]. Available: <https://enterprisecraftsmanship.com/posts/cohesion-coupling-difference/>. [Accessed January 2023].
- [6] C. Taube-Schock, R. J. Walker and I. H. Witten, "Can We Avoid High Coupling?," in *ECOOP 2011 - Object-Oriented Programming*, 2011.
- [7] E. Gamma, R. Helm, R. Johnson, J. Vlissides and G. Booch, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
- [8] A. J. Riel, *Object-Oriented Design Heuristics*, Addison-Wesley Professional, 1996.
- [9] D. L. Parnas, *On the Criteria To Be Used in Decomposing Systems into Modules*, 1972.
- [10] D. L. Parnas, "Designing Software for Ease of Extension," In *Proceedings of the 3rd international conference on Software engineering (ICSE '78)*. IEEE Press, 264–277., 1978.
- [11] A. T. Cohen, *Data abstraction, data encapsulation and object-oriented programming*, vol. 19, New York: Association for Computing Machinery, 1984.
- [12] "SlidesCodeHighlighter," [Online]. Available: <https://romannurik.github.io/SlidesCodeHighlighter/>.

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

-----

An excellent piece of work, well-formed and sourced. Well done, Callan.

A very strong implementation, excellent use of OO (even if it was beyond the scope of the module), well-written and structured code, and good use of unit tests. Rudimentary "AI" opponent implemented, with headroom for a better opponent to be added in future.

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

