Redundant Coupling

Peter Kos
Rochester Institute of Technology
plk5062@rit.edu

Abstract

In the field of software engineering, researchers have coined the terms "cohesion" and "coupling" to describe the structure and interaction between classes in an object-oriented environment [6]. The traditional wisdom is to use low coupling and high cohesion to reduce side-effects while making isolated changes in code. In this paper, we present an alternative methodology: *redundant coupling*. In redundant coupling, coupling between classes is maximized in order to create the strongest possible foundation in the architecture. This has been shown to provide a 45% increase in unit test stability (in flaky environments), 71% increase in developer confidence, and a 14% increase in my personal happiness since my second wife left me.

1 Introduction

Coupling and cohesion are core ideas in software architecture. They are a little ambiguous too, as modern computer science education does not usually address the issues that can (allegedly) arise from high coupling and low cohesion.

2 Theory

Coupling is introduced as a way to "[minimize] the paths along which changes and errors can propagate into other parts of the system, thus eliminating disastrous 'ripple' effects" [6, p. 233]. However, this operates on two false assumptions:

1. Individual change's effects on the whole system need to be minimized
2. Errors can propagate into other parts of the system

Changing a piece of code involves a deep insight into its function. (One can imagine careless changes, but these are part of the natural software development lifecycle).

...

Errors, too, are by definition, a symptom of the system. The solution for an error may be in an individual class, or across multiple classes, yet the codebase as a whole suffers.

2.1 Definitions

Definition 1. *Equivalent access is a reciprocal access between two classes A and B, such that the following holds:*

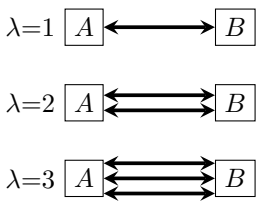
- *Let α be a field of A,*
- *Let β be a field of B,*
- *Let $\mathbb{A}(\alpha)$ be an access of field α,*
- *Let $\mathbb{A}(\beta)$ be an access of field β*

An access is equivalent if and only if

$$\Theta(\mathbb{A}(\alpha)) = \Theta(\mathbb{A}(\beta))$$

- For the fields $\alpha =$ `String` and $\beta =$ `String`, these would be an equivalent access, as the access/set complexity of a string [1] is $\mathcal{O}(1)$.
- For the fields $\alpha =$ `HashMap<T>` and $\beta =$ `HashMap<T>`, the same is true, as here, the complexity of any hash map operation is $\mathcal{O}(n)$.
- For the fields $\alpha =$ `String` and $\beta =$ `HashMap<T>`, $\mathbb{A}(\alpha) = \mathcal{O}(1)$, and $\mathbb{A}(\beta) = \mathcal{O}(n)$. (A HashMap, with a sufficiently large load factor, may need to iterate through all of its elements to get a specified ith element.) Therefore, in this case, class A and B would **not** have equivalent access between fields α and β.

Definition 2. *The coupling factor λ is defined to be the number of equivalent accesses between two classes A and B.*



[1] In most programming languages.

2.2 λ coupling factor

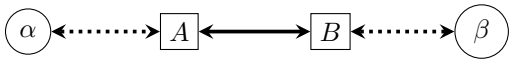
We define two classes A and B to have *one-string coupling* if they are bidirectionally coupled in one mechanism, whether that be:

- Reciprocal method-calling, or
- Equivalent access.

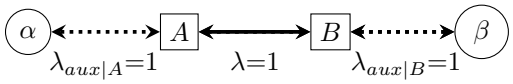
To see how redundant coupling increases the foundational stability of a class, we take to the example of auxiliary classes.

Auxiliary classes are defined as classes that we do not anlayze at the present moment, that are **not** coupled to any other class. The load factor between two classes A and B, e.g., λ_{AB}, is only defined with respect to A and B. (Later on, we define a process called promotion that addresses how we can move between classes of focus).

Say that we have an auxiliary class α that is coupled to A, and the same with β for B.



We can see that the load factor between each class is as follows:



The auxiliary coupling factor, $_{aux}$, is equal to the coupling factor of our two classes of interest (λ). We define this to be a point of **Fowler Equlibrium**.

Definition 3. *A **Fowler Equilibrium** is present between two classes A and B such that*

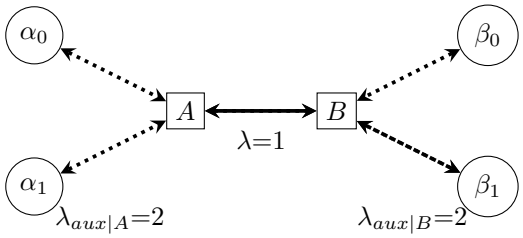
$$\lambda_{AB} = \lambda_{aux|A} = \lambda_{aux|B}$$

(where λ_{AB} is the coupling factor between A and B, and $\lambda_{aux|A}$ is the coupling factor between the auxiliary classes of A.)

The total auxiliary coupling factor is represented as such:

$$\sum_{\alpha \in A} \lambda = \lambda_{aux|A}$$

However, once we add two additional auxiliary classes on each side, we run into an issue:



Here, the auxiliary coupling factors are not at equilibrium with the interior coupling between our classes of interest. This ia a **misbalanced** system, and this is fundamental issue this paper serves to address.

3 Limitations

We acknowledge [2] that it is "unlikely, for financial and structural reasons" [5] to develop a model that can account for coupling in non-OOP paradigms.

Additionally, it is not known how this would affect projects in multiple languages. Mayer and Schroeder define an `XLL approach` [3, p. 97], where developers manually specify links between abstract concepts present in each language. In this system, one could couple the artifacts themselves, but this is more akin to coupling specific fields within a class – not the class overall.

4 Testing Methodology

Tests were originally written using a source-compiled version of JUnit 2, until the inferiority of this solution was realized through several cognitive behavioral therapy sessions.

Instead, we pivoted to use `Rust 1.58.0-nightly`. Rust [2] provides a safe, fast environment without garbage collection, which allowed us to mitigate any cross-bag insect contamination [4].

One potential issue is a common misconception that Rust does not provide an OO environment. We assert that it is possible to represent an OO-pseudostructure within Rust, without resorting to an inferior language. (See Appendix A.)

5 Results

As shown previously [1], we found a few notable improvements in codebases that used redundant coupling over the traditional low coupling / high cohesion model:

1. 45% increase in unit test stability
2. 71% increase in developer confidence
3. 38% reduction in alimony paperwork

Unit test stability occurred a natural consequence of allowing changes to propagate throughout the codebase. Intermittent tests can be caused by any number of bugs, and are usually localized to one or two classes. Through redundant coupling, the codebase is unilaterally influenced by these issues, which allow the system overall to behave more reliably.

Developer confidence was also observed to increase markedly.

[2] i.e., forced to concede

6 Conclusion

Todo.

7 Appendix A

```
// Some sample code we wrote as part of a
    demonstrative paper on Rust coupling.
// We are not sure if it compiles.

struct User {
    var first_name: &str,
    var last_name: &str,
    var age: u8
}

impl User {
    fn get_full_name(&self) -> &str {
        first_name + last_name
    }
    fn have_birthday(&mut self) {
        self.age += 1;
    }
}

struct Login {
    var cur_user: User
}

impl Login {
    fn login(&mut self) {
        todo!();
    }
    fn get_full_name() -> &str {
        cur_user.get_full_name()
    }
}
```

Listing 1: Example of an OO structure in Rust, with coupling

References

[1] Peter Kos. Redundant coupling. Association for Computational Heresy, 2022.

[2] Nicholas D. Matsakis and Felix S. Klock. The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, page 103104, New York, NY, USA, 2014. Association for Computing Machinery.

[3] Philip Mayer and Andreas Schroeder. Cross-language code analysis and refactoring. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pages 94–103, 2012.

[4] Sulochana Paudyal, George P Opit, Frank H Arthur, Georgina V Bingham, Mark E Payton, Sandipa G Gautam, and Bruce Noden. Effectiveness of the zerofly® storage bag fabric against stored-product insects. *Journal of stored products research*, 73:87–97, 2017.

[5] Michael L. Scott. Cover letter for dean of the golisano college of computing and information sciences. 2022.

[6] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 38(2):231–256, 1999. Copyright - Copyright International Business Machines Corporation 1999; Last updated - 2021-09-10; CODEN - IBMSA7.