

Indirect Coupling As a Criteria for Modularity

Hong Yul Yang, Ewan Tempero
Department of Computer Science
University of Auckland
Auckland, New Zealand
{hongyul|ewan}@cs.auckland.ac.nz

1 Introduction

Assessment of a modularisation technique should involve some form of measurement as to how modular the code becomes as a result of applying the technique. One problem we face is that there is a lack of a precise definition of *modularity*. The IEEE glossary of software engineering terminology defines modularity as “the degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components” [1]. While this is not an operational definition, especially as the notion of discreteness, or *independence*, needs to be further defined, it provides a good starting point for discussion. To measure according to this definition we need to measure “degree of independence”. We are interested in a particular form of dependence, namely *indirect* dependence and believe this is an important criteria for assessing modularity.

2 Direct and indirect dependence

We define dependence between components as being “direct” if the existence of the dependence is determinable by just by looking at the source (or design description) of the dependent component. For instance, in the program from figure 1, we can see that class A is directly dependent on classes C and D. In the simplest case, a renaming change to C or D affects A, as it would then require recompilation of A. Now *indirect* dependence is the complement of direct dependence. While indirect dependence may be thought of as just the transitive closure of direct dependences we find that this is not sufficient [4].

By some definition of ‘dependent’, C and D are independent. For example, removing C from the program would not cause compilation of D to fail. However a closer inspection reveals that a different kind of dependence exists between C and D. At its current state, executing the program (through A’s main method) results in a null pointer exception thrown

```
class A {
    void main() {
        C c = new C();
        D d = new D();
        d.bar(c.foo());
    }
}

class B {
    String str;
    void set(String str) {
        this.str = str;
    }
    String get() {
        return str;
    }
}

class C {
    B foo() {
        B b = new B();
        //b.set("hi");
        return b;
    }
}

class D {
    void bar(B b) {
        System.out.println
            (b.get().length());
    }
}
```

Figure 1. Illustration of indirect dependence

by D because it tries to dereference `b.get()`, which evaluates to null. This is caused by C failing to initialise the value of the field (`str`) of the B instance object. Now, by uncommenting the statement `b.set("hi")` in C we can avert the null pointer exception in D. In other words there is a change to C that affects D, which signifies dependence, and furthermore an indirect one. This is what might be called *data flow* dependence (as indicated by the arrows in figure 1), which is our current focus.

We believe there is additional cost associated with indirect coupling as it is not only the presence of dependence but also the task of identifying the dependence at first place that requires effort. For instance almost any developer who had faced a null pointer exception (or any failure of this nature) would have had to spend time tracking down the source of the null value.

3 Toward modularisation

It is important to be aware of indirect coupling (IC) due to its “hidden” nature. We are not claiming that IC is completely undesirable. Just as some instances of “direct” cou-

```

class A {
  void main() {
    C c = new C();
    D d = new D();
    d.bar(c);
  }
}

class D {
  void bar(C c) {
    B b = c.foo();
    System.out.println
      (b.get().length());
  }
}

```

Figure 2. Removing the indirect dependence in figure 1

pling are needed to hold the program together, some data flow dependence is also inevitable. What we need to focus on instead is determining which forms of indirect coupling are *avoidable*. We argue that a system with high levels of avoidable indirect coupling is “unmodular”.

We can modify classes A and D from the above program to what is shown in figure 2, where the dependence between C and D is now turned into a direct one. We argue this is a better design at least in the sense that the connections are made more *explicit*, which contributes to reduced effort. Of course whether by making this change we end up introducing greater effort in some other dimension is not clear. We need empirical evidence to be able to make this judgment, which is the subject of our current research.

One of the first steps to take in our study was in determining the prevalence of indirect coupling in actual software applications [3], which led us to develop metrics to represent “how much” of it is going on. The metrics are based on data flow paths, which we call “chains” — for example, the data flow path indicated by the arrows in figure 1 constitutes one such chain. The metrics are based on the idea that the longer the chains, the further apart the dependents are, which signifies being more “hidden”. Also the more the number of chains associated with a component the more dependences it creates.

Our study in [3] aimed to discover any noticeable patterns among the measures and see whether the measures are concordant to our intuition on the inherent quality of the applications under study (based on our experiences with the code). The patterns were generally heterogeneous, but the results have shown that the metrics do help in pointing out some peculiarities in the design. For example figure 3 shows a histogram of a particular metric that measures how many classes are involved with chains originating from a given class. What this signifies is that there are nearly 100 classes that potentially “affect” more than 100 classes according to this metric. Of course no conclusions can be drawn as to whether this signifies bad design without more information, but the result shows promise as it coincides with other quality views on the same application: for instance it is reported to involve some strong cyclic depen-

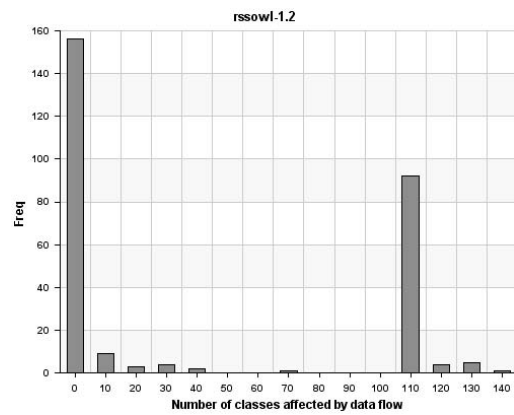


Figure 3. Histogram of indirect coupling measures for application rssowl

dencies [2].

While there is some promise that the above metrics are telling us something useful, we still need to determine the exact extent to which they affect quality. Thus the next important step is to empirically validate these metrics against the relevant quality attributes, including modifiability and understandability, and we are setting up a controlled experiment for this.

4 Conclusion

In this paper we discussed a criteria for modularity, namely indirect dependence, or coupling. We have explored the potential impact of indirect coupling on quality and studied its effect on existing applications through initial empirical studies. We strongly believe in the need to devise modularity techniques to focus explicitly on the control of indirect coupling, and to do this we first have to determine the exact effect of indirect dependence on external quality attributes.

References

- [1] IEEE standard glossary of software engineering terminology, 1990.
- [2] H. Melton and E. Tempero. An empirical study of cycles among classes in java. *To Appear In Empirical Software Engineering: An International Journal*.
- [3] H. Y. Yang and E. Tempero. Measuring the strength of indirect coupling. In *Australian Software Engineering Conference*, 2007.
- [4] H. Y. Yang, E. Tempero, and R. Berrigan. Detecting indirect coupling. In *Australian Software Engineering Conference*, 2005.