

# Measuring the Strength of Indirect Coupling

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

## Abstract

*It is widely accepted that coupling plays an important role in software quality, particularly in the areas of software maintenance, so effort should be made to keep coupling levels to a minimum in order to reduce the complexity of the system. We have previously introduced the concept of “indirect” coupling – coupling formed by relationships/dependencies that are not directly evident – with the belief that high levels of indirect coupling will constitute greater costs to maintenance as it is harder to detect. In this paper we extend our previous studies by proposing metrics that will advance our understanding of the exact relationship between indirect coupling and maintainability. In particular, the metrics focus on the reflection of “strength” as it is a fundamental component of coupling. We present our observations on the results of applying the metrics to existing Java applications.*

## 1 Introduction

Change becomes a root of many concerns of software engineering and the issue is evident through surveys that account typically around 70% of the software budget being spent on maintenance (e.g. [13]), where change plays an intrinsic role. One notion of *quality* in software design is how easy or difficult it is for the design to cope with change. A concept that is widely associated with quality is *coupling*, which is a “measure of the *strength* of association established by a connection from one module to another” [16]. There is a natural relationship between coupling and change, as stronger interconnections between modules cause them to be more vulnerable to changes made to one another. An important design principle is to keep coupling to a minimum, as more tightly coupled modules constitute increased complexity of the system, making it more difficult to understand and easier for changes and errors to propagate through the connections.

We previously [20] argued that indirect coupling — cou-

pling where the connections are not directly visible at the source level — did not gain as much attention as the more common direct coupling measures. We found that while there are mentions of indirect forms of coupling in the literature, such forms do not address the type of connections that we think are important, especially those of which concern transfer of values via data flow. Through this work we have established the viability of the concept and thus believe that indirect coupling (IC) tells us more about quality, particularly in that it reveals connections that are otherwise not captured.

Ultimately what we want is a corroborated relationship between IC and useful external quality attributes, especially understandability and modifiability. While we have a general theory regarding such relationship, we first need the ability to make *accurate* and *repeatable* observations on how IC manifests in software systems, just as we would need for any scientific method for empirical software engineering. Acquiring this ability is in fact non-trivial, as we need to know what the observations are and establish *measures* for the relevant attributes under observation. For example, making a useful observation on the orbital patterns of planets not only requires the elevation and azimuth, but also the location of the observer and the time of the observation. We face the same problem with respect to IC, and this is what we must currently address. Once we have decided on what measures best describe the observations we can start formulating and validating precise models and hypotheses on the relationships between the internal attributes reflected by these measures and the external quality attributes.

This paper focuses on establishing a useful definition and measurement of indirect coupling to support the observations. As coupling deals with not only the establishment of interconnections between modules but also the strength therein, we need to go further than simply detecting the presence of indirect coupling. Thus the two main research questions this paper answers are:

1. What measures might be used to capture the strength

of indirect coupling, and how well do they reflect our intuitions?

2. What is the relationship between the different measures: do they present the same or different information?

Section 2 summarises coupling, indirect coupling and relevant work. Section 3 recapitulates the precise concept of indirect coupling from our previous paper and furthermore examines the various issues with defining and measuring IC. A set of metrics that aim to reflect the notion of strength are proposed. Section 4 describes the process in which the metrics are collected from existing Java applications. The observations on the metric values exhibited by these applications are presented in section 5. Section 6 discusses possible limitations of the study and future directions. Finally conclusions are drawn in section 7.

## 2 Background

### 2.1 Coupling

Coupling was introduced by Stevens et. al [16] in the context of structured design. Their idea was that systems are easier to manage when the connections between modules are as “tenuous as possible”, thereby minimising the likelihood of changes and errors propagating to other parts of the system. The concept was later on transferred to the field of object-oriented software, with classes replacing modules and the object-oriented features introducing new ways in which classes can be connected, such as through inheritance [2, 6].

Although, as Yourdon and Constantine put it, “there are no standard measures for coupling”, what remains universal across the various views of coupling is that coupling deals with types of interconnections and their strengths therein [5]. There is however no absolute indication as to what exactly strength means or what are the kinds of connections that constitute coupling. Briand et. al developed a framework that provides a consistent view across the existing coupling measures [5] by capturing a set of attributes that characterise the measures — in particular what the kinds of connections are and how strength is measured.

### 2.2 Indirect coupling

Briand et. al’s definitions of couplings, as indicated in the framework, deal with connections that are of “direct” nature. In other words, coupling is formed mostly by method invocations or in the form of a “uses” relationship. In our previous paper [20] we presented indirect coupling, where the connections are not obvious. More precisely the

connection is actually through data flow: two classes are coupled if one depends on a value that is generated by the other. We have identified class interaction patterns where two classes are not directly coupled yet there is a data flow from one to the other, hence signifying the importance of being able to capture this kind of coupling. In that paper we have presented a means to *detect* the “presence” of coupling. While detection is indeed a form of measurement, we need more comprehensive measures in order to help our observations.

### 2.3 Related work

The idea of non-evident relationships in software was also previously discussed in [21] as “hidden dependencies”. Their focus is on “detection”, an algorithm for which is presented but does not seem to have been implemented. Another relevant concept is “ripple effect” [4], which also deals with change propagation through data flow paths. Coupling measures in terms of program slices was proposed recently in [11]. Program slicing is a technique for finding all the parts of a program that can influence a value at a given point in the system [18]. Slice-based coupling suggests that A and B are coupled when a *forward slice* of a point in A (i.e. parts that are influenced by A) intersects with a backwards slice of B (i.e. parts that influence B). Our work can be distinguished by the fact that we are explicitly dealing with notions of “strength” by incorporating aspects like “distance” and “area”, and we have performed empirical studies to demonstrate these notions.

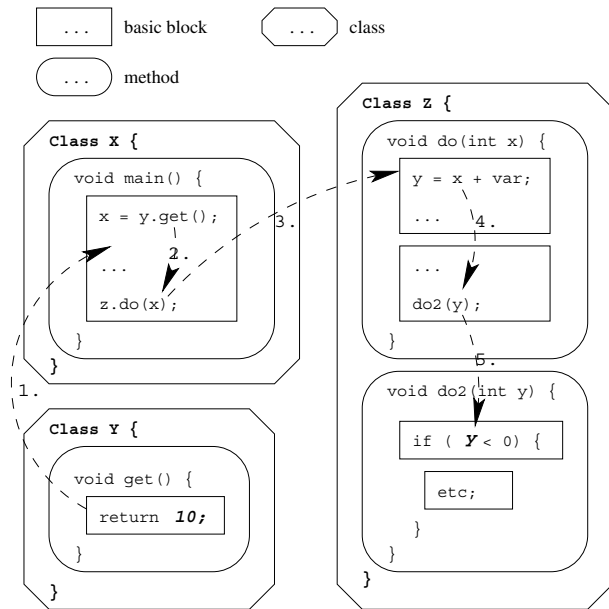
## 3 Indirect coupling

### 3.1 Definition

The particular form of indirect coupling we focus on is defined such that *a given class C is indirectly coupled via data flow to another class D iff there exists a value used in C that is defined in D* [20]. The premise of this indirect coupling is that the behaviour of C is potentially dependent on the value generated by D. Thus a *change* in D, i.e. modifying the value being defined, is likely to *affect* C, which as a result may produce an erroneous behaviour due to the value being wrong.

Figure 1 illustrates a contextual view of an example of a data flow that causes an indirect coupling between class Z and Y. Class Z uses the value of y in its method `do2`, which is eventually defined in the `get` method of class Y. We further label this particular usage site as an *initial* usage site, to distinguish it from usage sites that in turn act as definitions for other usage sites. Similarly, we label the definition (def) site as a *terminal* def site, and they are in the form of a literal (integer, string, etc.) or a constructor call (in the case

of the defined value being an object). An initial usage and terminal definition effectively form the two ends of a *chain* of alternating usages and definitions.



**Figure 1. Indirect coupling: chain crossing different levels of boundaries**

### 3.2 Chains

We use chain as a central attribute when it comes to reasoning about indirect coupling. A chain can be expressed in terms of graph vocabulary. Each statement in this example would correspond to a node, while each immediate data flow from a definition site to a usage site corresponds to edges (shown as dashed arrows in Figure 1). Note the various boundaries within which nodes can be co-located. In the obvious case, there are class and method boundaries (shown in their respective shapes). We can also group nodes into basic blocks, where each block represents a series of non-branching statements. Although basic block is a notion usually used in the context of compilers, it can easily be mapped to source code as well. The justification for counting basic blocks is mainly technical. As described later on, the program analysis for computing chains is performed on Jimple code, which is a simplified, finer-grained representation of Java, and thus the mapping from Java statements to Jimple statements are not necessarily one-to-one. Basic blocks serve as a common denominator between the two abstractions that allow us to reason about understandability still at the source code level.

It is useful to observe the *length* of a chain, i.e. how many boundaries it crosses, thereby capturing some notion

of “distance”. Intuitively, we can map distance to maintenance effort, since the longer the chain of the flow of values across the system, the more work will be required to trace this flow, potentially having to switch between different methods and different classes, etc. This shares a similar line of reasoning to concepts like spatial complexity [9] where distance is reflected by the number of lines of code “traveled” between points in a program. The actual unit of length (i.e. the *granularity* of the chain) will be determined by the level of boundary being considered, whether it is in terms of classes, methods or blocks. In this example, the length of the chain in terms of classes is 2 (X to Y, Y to Z), 3 in terms of methods (get() → main() → do() → do2()), and 4 in terms of basic blocks.

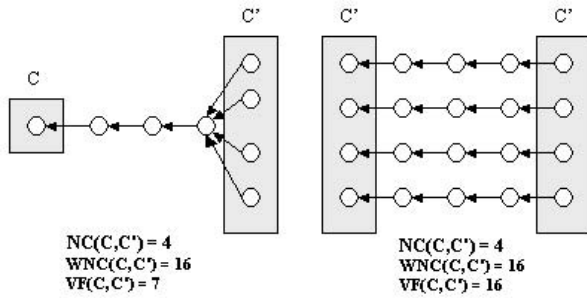
The choice of granularity of chains depends on which of them (or whether all of them) best quantifies effort, which is not clear-cut and leads to a trade-off. For example, reasoning about distance in terms of classes would likely be the simplest, as often with coupling we are talking about the interactions between classes. However we might often find that long chains take place within classes as a result of, say, self-calls, which may cause an additional effort as compared to just a single method within a class, in which case it may be more appropriate to reason about at method level boundaries. Hence for the time being we should consider all of them to be useful.

### 3.3 Measurement

Measuring coupling can be viewed from two perspectives. One is the relationship between a given pair of classes (*inter-class*), as coupling is a binary relationship, where each measure per a given pair of classes would represent the strength of connection. The other is on the aggregation of coupling relationships with respect to a single class with the intent of seeing how much influence a given class has over the system (*per-class*).

A useful guide to determining how to gauge strength is to refer back to Yourdon and Constantine’s question “How much of one module (class) must be known in order to understand another module (class)?” We could employ the notions of chains and lengths discussed above. As far as inter-class measures are concerned, higher numbers of chains between classes is likely to indicate that more aspects of one class must be known in order to understand the other class. Also the longer the chains, the harder it would be to actually determine the very fact that they are coupled. With per-class measures, the question above could be rephrased as “How much of the class’s surroundings (other connections and classes) must be known to understand it?”

One compelling scenario is when making a change to class C, one would need to know what lengths one must go to in order to make sure the change does not propagate a



**Figure 2. Illustration of inter-class metrics and overlapping chains**

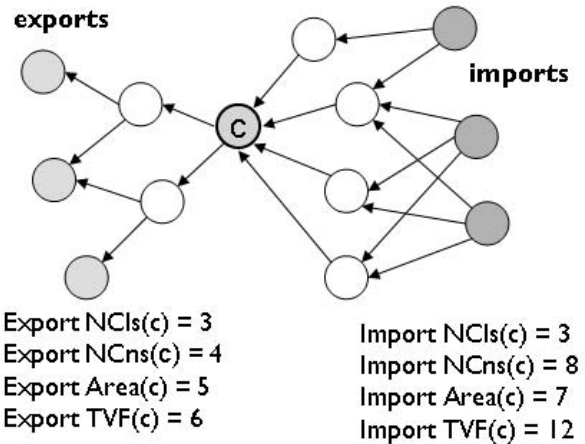
negative effect. The reasoning can be applied in the other direction as well: if something goes wrong in this class, one would want to know how much work is needed to determine possible places that would have caused it. Here, again, the more chains it has going to other classes, the more things one would need to know. And likewise, the longer the chains, the greater the lengths one would have to go to understand. With this discussion in mind, the following are the proposed metrics.

### 3.3.1 Inter-class metrics

**No. chains** —  $NC(c, c')$  — is a count of how many chains flow from  $c$  to  $c'$ . It reflects the intuition that the more the number of chains going from A to B, the stronger the connection as discussed above.

**Weighted No. Chains** —  $WNC(c, c')$  — is an extension to NC where each chain is weighted in terms of its length (depending on granularity). For the purpose of this study, the metric is simply the sum of the lengths of all chains from  $c$  to  $c'$ . Through observations, however, we could find an appropriate scale to normalise the measure. This is however not of direct importance as we are primarily concerned about the relationships between magnitudes, not in themselves. For example, given classes A, B, C and D such that A has 5 short chains (length 2) to B and C has 3 very long chains (length 8) to D, the second case suggests a stronger measure than the first:  $8 \times 3 = 24$  and  $5 \times 2 = 10$  respectively.

**Volume of flow between classes** —  $VF(c, c')$  — is a distinct count of the edges within all of the chains between  $c$  and  $c'$ . It is conceptually equivalent to WNC where the weight is reduced for chains that share some data flow paths. Figure 2 depicts an extreme case of two pairs of classes sharing the same WNC of 16 ( $4 \times 4$ ). One could form a line of reasoning such that



**Figure 3. Illustration of per-class measures**

the latter case should yield a greater strength value as, the more “spread out” the chains are the greater the “area” that needs to be inspected, which in turns leads to greater difficulty in understanding the whole network of dependencies.

### 3.3.2 Per-class metrics

From a single class’s perspective, distinction should be made between classes it is coupled to and those that are coupled to it. This distinction is also referred to as import and export [5], respectively. With indirect coupling, export coupling corresponds to all classes that depend on a value generated in the class in question, and the other direction applies for import. For the purpose of terminology, we will say that a class that has a high degree of export coupling has a potentially high *impact* and conversely, if it has lots of import couplings, it is potentially *vulnerable*.

The following types of impact/vulnerability measures are proposed, and figure 3 illustrates the differences between the metrics based on the central class C and its associated classes.

**Number of export/import classes** —  $NCl(c)$  — is a count of how many classes  $c$  exports/imports.

**Number of export/import chains** —  $NCns(c)$  — is a count of all chains between  $c$  and its export/import classes. This is equivalent to the sum of  $NC(c, c')$  for all  $c'$  to which chains exist from  $c$ .

**Export/import area** —  $Area(c)$  — in addition to counting the export/import classes of  $c$ , counts all the “nodes” (which depends on the granularity level) involved within the chains between them, thus reflecting a kind of “area” measure. In figure 3, we see that C has

the same value of  $NCl_s$  for both import and export sides but greater value for  $Area$  on the imports side, which highlights that there are overall more classes both actively and passively involved in the chains than on the exports side.

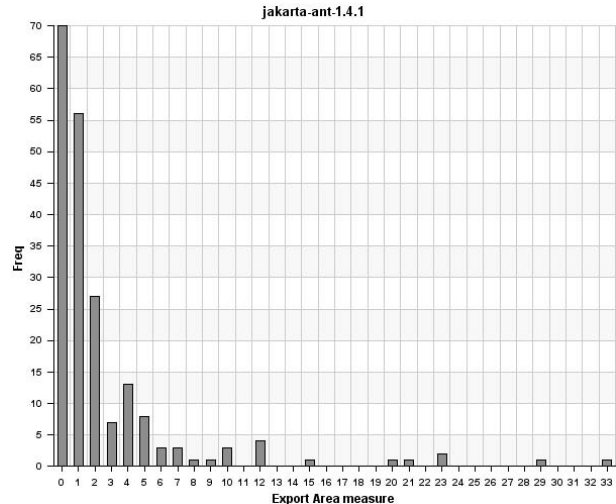
**Export/import total volume of flow** —  $TVF(c)$  — counts the distinct “edges” of export/import chains from  $c$ . This is equivalent to the aggregation of  $VF(c, c')$  for all  $c'$  to which chains exist from  $c$ . This will be able to distinguish between cases where there are same number of nodes but different magnitude of flow among them.

## 4 Methodology

The metrics are gathered by use of our tool, ICD, initially mentioned in [20]. Several modifications were made to the tool since then. It is now made standalone in order to facilitate batch-style analyses, although the prospect of IDE integration in the future still remains. Also it analyses Java bytecode, or more specifically, Jimple from the Soot framework [17], which is an SSA (static single assignment)-form intermediate representation derived from bytecode. It also utilises static analysis features of the Indus project [10], which is based on Soot.

The first step of the measurement is to analyse the source/Jimple for use-def information and generate a graph data-structure comprising the usage/definition sites and data flows among them. The algorithm for computing this employs standard inter-procedural data flow analysis techniques such as those found in the slicing literature (e.g. [1]), and in particular the data- and object-flow analyses that Indus provides, as described in [15]. The precision of the analysis is thus dependent on the underlying techniques, which deal with well-known issues such as polymorphism and array aliasing to a certain extent.

This data-flow graph is used as a basis of all the remaining analyses. Each node of the graph corresponds to a single “value” being either used or defined — the most atomic level of granularity, and a directional edge exists from node A to node B if A defines B. Chains are enumerated by starting from *initial* uses, and traversing paths until they reach *terminal* defs and backtracking so that eventually all possible paths are covered. There are indeed issues when there is a possible cycle (for instance in recursive methods). Currently only acyclic paths are considered as there are infinite ways in which cyclic paths can be enumerated, but this will have to be addressed in the future. The various granularities for a single chain are obtained by “grouping” adjacent nodes having the same granularities. For example, with a chain of  $(X.a, X.b, Y.c, Z.d, Z.e)$  where a to e are the atomic nodes and X to Z are the methods they respectively belong



**Figure 4. Distribution of export Area measures for ant**

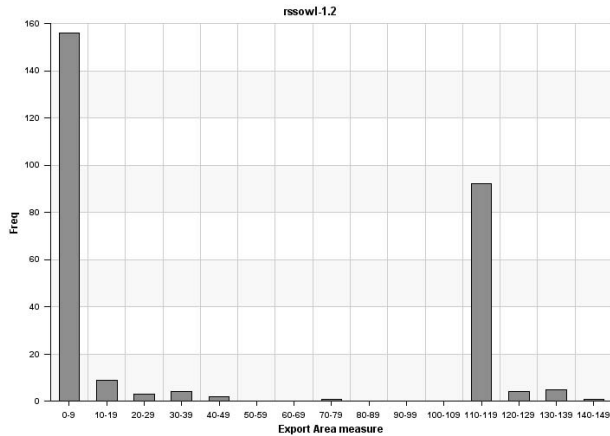
to, then the method level chain would be  $(X, Y, Z)$ . This provides the basic mechanism for computing chain-based metrics ( $NC$ ,  $WNC$  and  $NCns$ ).

Other measures —  $VF$ ,  $Area$  and  $TVF$  — instead of dealing with individual chains, act on a sub-graph of the data-flow graph on which the chains are based. We can still apply the above grouping method for chains to these sub-graphs. For instance, if we want to achieve method granularity from the atomic nodes, we can identify each set of nodes  $N = n_1, n_2, ..n_x$ , such that all nodes share the same method *and* each node has at least one edge to or from any other node of  $N$ . Then we could group each them into a single “meta-node” representing a method. We redefine edges such that there is a single edge from a method to another if there is an edge between its subparts. Once boundaries and edges are determined, getting the metrics is a simple process a of counting the constituent nodes or edges (for  $VF$ ,  $Area$  and  $TVF$ ) or doing a standard traversal starting from use sites and ending in def sites or vice versa (for  $NCl_s$ ).

## 5 Results

### 5.1 Metrics overview

Having defined the metrics, one of the first observations to make is what characteristics are reflected in existing applications. We applied them to Java applications from a corpus that we have compiled [14]. The size of the applications that could be handled were limited by the complexity of the analysis provided by the Soot and Indus frameworks. The analysed applications range from sizes of 29 classes to 775.



**Figure 5. Distribution of export Area measures for `rsslowl`**

Due to space limitations we cannot present the results in full, but instead we show a few representative examples. It was found that the typical shape of distribution of the measures is that of figure 4, which plots a histogram of the export *Area* measures (in class granularity) for `ant`. The shape of the graph suggests a power-law-like curve, although the exact relationship cannot be determined. Some notable variants include the application `rsslowl` (figure 5), where we can see a marked increase in the classes exhibiting values of 110 or more, which effectively suggests that those classes are potentially affecting such many other classes. While we cannot yet determine whether this is good or bad, we still benefit from the measures being able to reveal these areas.

Figure 6 plots the maximum export *TVF* over the applications, ordered by their size in terms of number of classes in over different granularity levels: class, method and block. Note that the values at method- and class-level granularities have been scaled in order to improve readability for the purpose of discussion in the next subsection (the actual values do not matter as we are only concerned with relationships). Certain “spikes” in the measures are visible, especially with `jparse`, considering its relatively small size (65 classes), `axion` and `hsqldb`. Again, the extreme values by themselves do not suggest any implication on quality, nevertheless they do serve to highlight potential peculiarities in the design. For example with `jparse`, the distinct gap between the maximum impact in terms of classes and that of blocks is apparently attributed to one of its parser classes, where there are numerous calls made primarily between methods and interactions within methods.

## 5.2 Relationship between metrics

The other main analysis was in obtaining a correlation between the different measures and granularities, which is aimed to help us better understand the relationship between the different metrics. For each application, one metric value for each class was plotted against another metric value for the same class. Or, with inter-class metrics, the same was done for each class pair. The Pearson’s correlation coefficient was then obtained. Since there are so many possible combinations of metrics and granularities that we cannot present them in full, we instead present some highlights.

### 5.2.1 Inter-class metrics

While the individual correlations between the inter-class metrics for each application varied, there were certain strong patterns. Figure 7 plots the correlation coefficients for selected applications (for readability) and metric, measured in the class granularity level. *NC* and *WNC* are found to have a consistently strong correlation where applications such `ant`, `jeppers` and `jparse` stand out. What this could mean is that the lengths remain uniform across the chains, which hence do not provide as much additional information as originally speculated, at least according to these applications. On the other hand *VF* had a consistently weak correlation with *NC*. An explanation of this could be along the lines of the example of `ant` as discussed above. The correlation between *WNC* and *VF* were slightly on the stronger side, in fact more prevalent than *NC* and *WNC* in the case of `drawswf`. The correlation pattern remains fairly consistent across granularity levels.

### 5.2.2 Per-class metrics

Figure 8 depicts a similar correlation information for per-class metrics at the class granularity level. The most notable correlation lies between export *Area* metric and export *TVF* metric. The applications displayed a consistently strong correlations above 0.93. Also the export *NCl*s metric displays strong correlations to both export *Area* and export *TVF*. The chain-based metric (*NCns*) on the other hand displays varying correlations to the other three metrics. Although not shown here, the correlations lessen at the lower granularity levels (i.e. methods and blocks). Also the export metrics are generally more correlated than with its import counterparts. Still the relationship patterns among the different metrics remain consistent.

## 6 Discussion and future work

There is a wealth of further combinations and configurations of metrics to investigate, which we intend to explore.

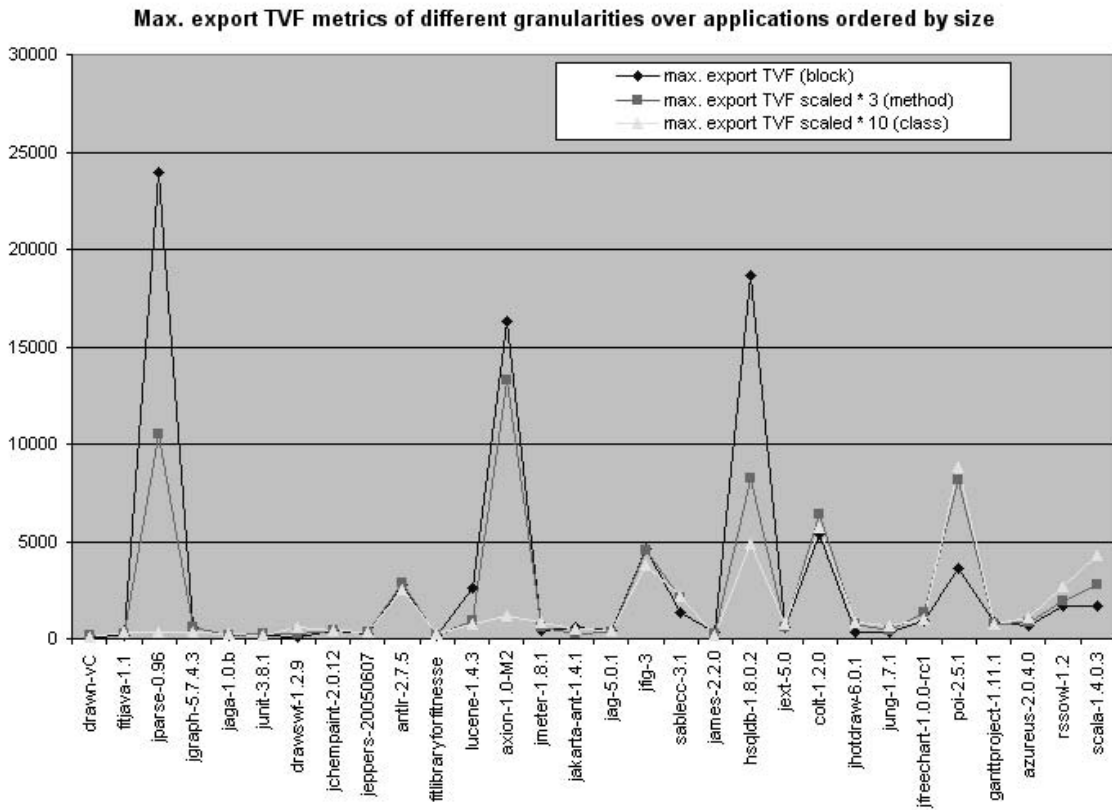


Figure 6. Varying granularities for export TVF

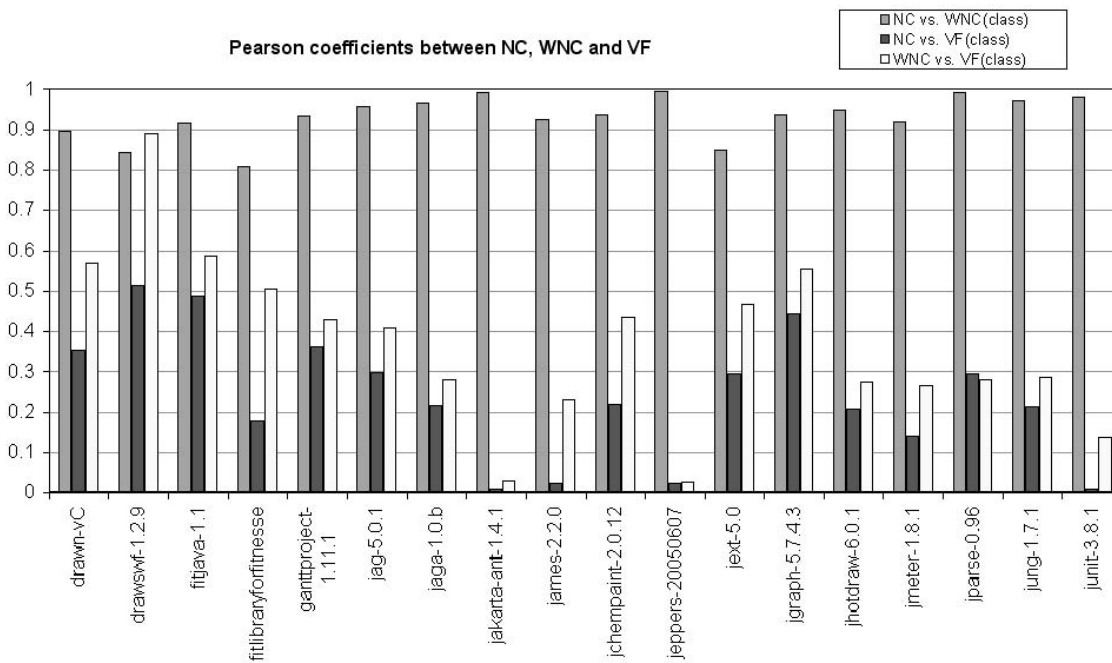


Figure 7. Comparison of inter-class metrics

### Pearson coefficients between export (impact) metrics

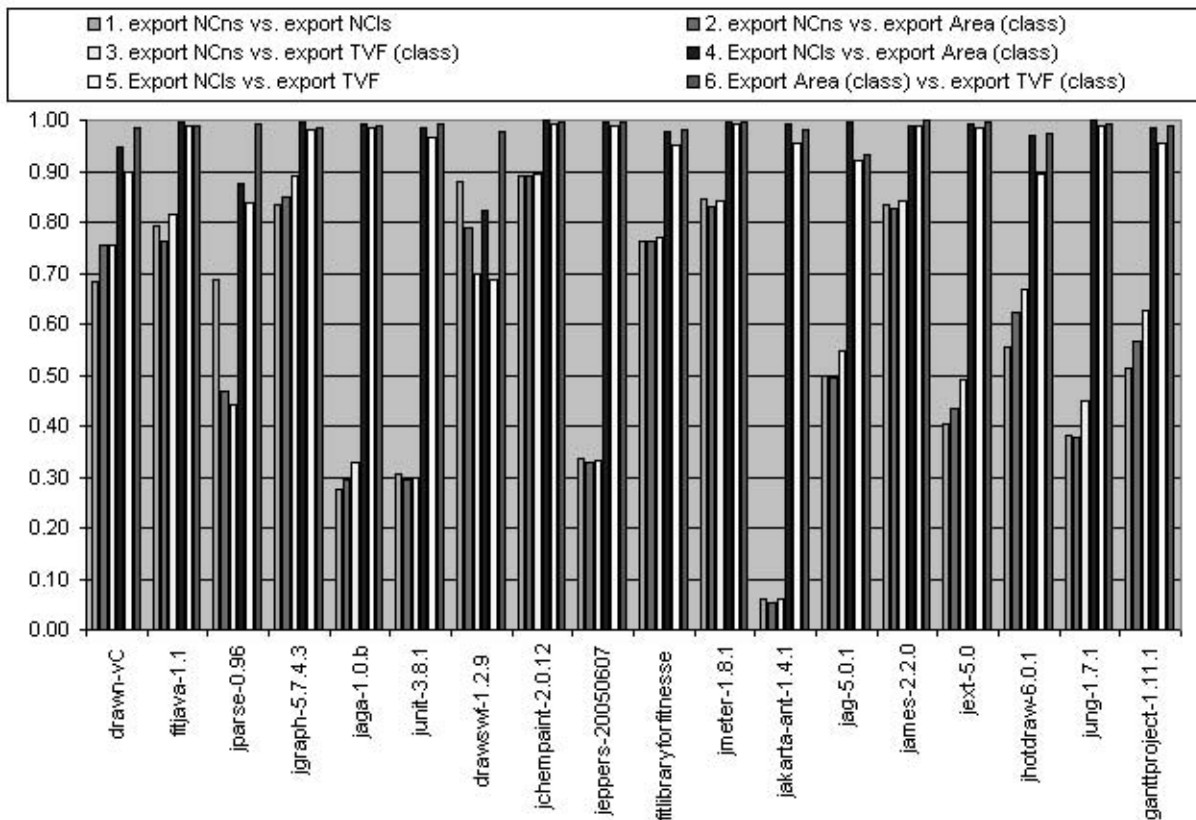


Figure 8. Comparison of per-class metrics

We have not exhaustively compared every possible combination of metrics, but what we have got so far gives a good enough picture of how these relate.

We have to keep in mind that the applications under study may not necessarily be a representative sample. Future work lies in expanding our corpus and collecting measures from more applications of greater range of domains. It is also worth exploring whether certain domains of systems inevitably exhibit stronger measure of indirect coupling, which do not necessarily contribute to the actual overall maintainability. This applies to applications such as *jparse*, *axion* and *hsqldb* where it is evident that the complex data flow involved in internally complex but largely “black-box” (parsing for example) components causes the measures to rise.

There is a possible issue with the meaning of the differing scales of magnitudes of the metrics, which vary from typically hundreds to tens of thousands. For the very purpose of this study the scale of magnitude was not important as we were more interested in their relationships. However the question of what is the threshold for determining an un-

desirable level must be answered later on. With some metrics this is more apparent than some others as, for instance, it tends to be easier to intuit with numbers of classes than the count of individual data-flow edges.

Then there is the question of “validation.” Within the software engineering community there has been a tendency to assume that validation of a metric means showing that it has some relationship to some quality attribute. However, as Bieman et al. observe, validation ought to mean demonstrating that the principles of measurement theory have been obeyed [3]. They refer to the measurement theory meaning as *internally valid* and the meaning generally used as *valid in the extended sense*. The conclusion is that validating a proposed metric in the extended sense is meaningless if the metric is not even internally valid. Our priority has been to consider only internal validation.

There has also been a trend to require that metrics meet some set of properties, such as those proposed by Weyuker[19]. However there is no agreement as to what are the right set of properties, and those that have been proposed have met with criticism [7]. Due to this uncertainty,

we have chosen not to follow this trend.

Thus the next step in the research will involve validation of these metrics in the extended sense. What we eventually want to find out is whether these measures actually correspond in some way to the actual maintainability of the system. The challenge with this is in finding out how exactly maintenance effort is to be determined.

A number of existing empirical studies on maintainability have used change-proneness [8, 12, 9] as a guide to maintenance effort. Usually it is computed in terms of amount of code changed between different versions. Although this is not a direct indicator of maintenance effort, it still will be useful since one of the utilities of applying indirect coupling is in finding out “hot-spots” and verifying the relationship between high indirect coupling and high change-proneness will contribute to this.

However it would be best to gain a more direct handle on maintenance effort than change-proneness, which would be to set up a controlled experiment that simulates a maintenance process and measure the corresponding efforts, especially for comprehension and modification. This will involve human participants as “maintainers” and the apparatus will consist of a small software system with the different IC metrics as variables. The experiment will be run by giving the participants a series of tasks relating to comprehending and correctly modifying code and recording their efforts by some means, for instance time taken.

## 7 Conclusion

We have proposed several metrics for capturing the strength of indirect coupling (IC) based on a preliminary theory that the longer the distance and the greater the area the more work is required to trace the connections and understand them. It is worth restating that the role of this study is in creating accurate *observations* of IC and showing that this is a non-trivial process by laying out the various aspects of IC that are relevant to quality and how they could be represented as measurements. By applying the metrics to existing Java applications we were able to find out how they can shape the observations. Among the findings, there are a select few “hot spots” that exhibit large values and many instances of small values. Also, we have found that there were certain groups of measures that were highly correlated, which leads to a possibility that they are essentially reflecting the same information. Finally it was interesting to see that the correlation between the same pairs of metrics vary according to granularity. These observations will help us formulate detailed models of the relationship between internal indirect coupling attributes with external quality attributes, which would then be corroborated using empirical methods.

## References

- [1] M. Allen and S. Horwitz. Slicing java programs that throw and catch exceptions. In *PEPM '03: Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 44–54, New York, NY, USA, 2003. ACM Press.
- [2] E. V. Berard. *Essays on Object-Oriented Software Engineering*, volume 1, chapter 7. Prentice Hall, Englewood Cliffs, New Jersey, 1993.
- [3] J. M. Bieman, N. Fenton, D. A. Gustafson, A. Melton, and L. M. Ott. *Software Measurement*, chapter Fundamental Issues in Software Measurement, pages 39–52. International Thomson Computer Press, 1999.
- [4] S. Black. Computing ripple effect for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13:263–279, 2001.
- [5] L. C. Briand, J. W. Daly, and J. K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, January/February 1999.
- [6] S. R. Chidamber and C. F. Kemerer. Towards a metrics suite for object oriented design. In *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 197–211, New York, NY, USA, 1991. ACM Press.
- [7] N. Fenton. Software measurement: A necessary scientific basis. *IEEE Transactions on Software Engineering*, 20(3):199–206, March 1994.
- [8] A. Foyen. Dynamic coupling measurement for object-oriented software. *IEEE Trans. Softw. Eng.*, 30(8):491–506, 2004. Member-Erik Arisholm and Member-Lionel C. Briand.
- [9] N. E. Gold, A. Mohan, and P. J. Layzell. Spatial complexity metrics: An investigation of utility. *IEEE Trans. Software Eng.*, 31(3):203–212, 2005.
- [10] Indus project site. <http://indus.projects.cis.ksu.edu/>.
- [11] B. Li, Y. Zhou, J. Mo, and Y. Wang. Analyzing the conditions of coupling existence based on program slicing and some abstract information-flow. In *snpd-sawn*, pages 96–101, Los Alamitos, CA, USA, 2005. IEEE Computer Society.
- [12] W. Li and S. Henry. Object-oriented metrics that predict maintainability. *J. Syst. Softw.*, 23(2):111–122, 1993.
- [13] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Commun. ACM*, 21(6):466–471, 1978.
- [14] H. Melton and E. Tempero. An empirical study of cycles among classes in Java. In *UoA-SE-2006-1*. Department of Computer Science, University of Auckland, 2006.
- [15] V. P. Ranganath. Object-flow analysis for optimizing finite-state models of java software. Master’s thesis, Kansas State University, 2002.
- [16] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.
- [17] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

- [18] M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.
- [19] E. J. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, September 1988.
- [20] H. Y. Yang, E. Tempero, and R. Berrigan. Detecting indirect coupling. In *Australian Software Engineering Conference*, 2005.
- [21] Z. Yu and V. Rajlich. Hidden dependencies in program comprehension and change propagation. In *IWPC '01: Proceedings of the 9th International Workshop on Program Comprehension*, page 293, Washington, DC, USA, 2001. IEEE Computer Society.