

ULRR

Fine-grained software metrics in practice

Item Type	Meetings and Proceedings
Authors	English, Michael;Buckley, Jim;Cahill, Tony
Citation	First international symposium on empirical software engineering and measurement;pp. 295-304
Publisher	IEEE Computer Society
Download date	2026-05-20 13:39:27
Item License	https://creativecommons.org/licenses/by-nc-sa/1.0/
Link to Item	https://hdl.handle.net/10344/1185

Fine-Grained Software Metrics in Practice

Michael English, Jim Buckley and Tony Cahill
CSIS Department, University of Limerick, Ireland.
{Michael.English, Jim.Buckley, Anthony.Cahill}@ul.ie

Abstract

Modularity is one of the key features of the Object-Oriented (OO) paradigm. Low coupling and high cohesion help to achieve good modularity. Inheritance is one of the core concepts of the OO paradigm which facilitates modularity. Previous research has shown that the use of the friend construct as a coupling mechanism in C++ software is extensive. However, measures of the friend construct are scarce in comparison with measures of inheritance. In addition, these existing measures are coarse-grained, in spite of the widespread use of the friend mechanism.

In this paper, a set of software metrics are proposed that measure the actual use of the friend construct, inheritance and other forms of coupling. These metrics are based on the interactions for which each coupling mechanism is necessary and sufficient. Previous work only considered the declaration of a relationship between classes.

The software metrics introduced are empirically assessed using the LEDA software system. Our results indicate that the friend mechanism is used to a very limited extent to access hidden methods in classes. However, access to hidden attributes is more common.

1. Object-Oriented Software Development

The Object-Oriented, (OO), approach to software construction became popular due to its ability to model real-world entities naturally and to provide suitable units for building large and complex systems, [31]. It is now considered the “dominant software technology”, [14].

The object-oriented approach to software development promotes software modularity. “Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules”, [3]. Coupling is a measure of the inter-connectedness of modules and cohesion is a measure of intra-module connectedness. Modularity is considered one of the core requirements of software quality as illustrated by its inclusion in many software quality models. For example, the three software quality models,

[23, 20, 15], discussed by Kitchenham and Pfleeger, [21], all include aspects of modularity. Due to the importance of modularity, measuring coupling and cohesion in software systems is an important research area.

Inheritance is a structural OO design mechanism and its use results in the creation of coupling between classes. It is one of the distinguishing features of the OO paradigm. As such there has been much discussion and disagreement as to the appropriate uses of inheritance, e.g. [27, 26, 30, 35, 24]. These discussions focus mainly on whether inheritance should be used for code reuse or whether an underlying semantic relationship should exist. Many empirical studies have also been undertaken to examine the use of inheritance in software systems, e.g. [2, 8]. These studies tend to highlight the limited use of inheritance in practice.

The friend construct is a C++ mechanism which grants a class or function access to the internal parts of other classes. Thus, the use of the friend construct also results in the creation of coupling between classes. Since the use of the friend construct does not restrict coupling to occur through a class’s interface, it is often considered poor programming practice, [11, 26]. Indeed the use of the friend mechanism has been associated with defects in software, [6, 29]. Others suggest that it is useful under certain conditions, [25, 28]. However, overall the consensus in the literature seems to be that the use of the friend mechanism is poor programming practice, results in poor design and poor quality and thus its use should be minimised.

In contrast to inheritance, little empirical analysis of the use of the friend mechanism in existing software systems has been undertaken and thus, there is limited empirical evidence to support the consensus view expressed above. This is surprising given the extensive use of the friend mechanism, [18, 12, 13]. In fact Counsell *et al.* stated that library-based systems “showed a distinct lack of any form of coupling (including inheritance)” between classes “other than through the C++ friend facility”, [13]. This type of analysis would be useful to support or reject the many claims that have been made in the literature about the use of the friend construct in the development of object-oriented software systems, or the impact it may have on other internal at-

tributes of software systems. Existing work in this area has focused on the interplay between inheritance and the friend mechanism, [12], and on the impact the use of the friend construct may have on external attributes of software, [6].

Existing measures of the friend mechanism, [22, 6] are coarse-grained. That is, these measures focus on friend declarations as opposed to actual friend usage. In previous work metrics to measure the redundancy in the usage of the friend mechanism were proposed and evaluated, [19]. In this paper, more accurate measures of the friend mechanism, inheritance and other forms of coupling are proposed. The added utility and additional insight these metrics provide into the constitution of software systems is exposed.

2. Metrics for Coupling Mechanisms

In proposing a software quality model for software, Dromey states that a bottom up approach should be applied in building a software quality model. Such a model should take language specific features into account, [16]. Churcher and Shepperd also argued that the usefulness of Chidamber and Kemerer metrics, [9], would be greatly enhanced if clearer guidance concerning their application to specific languages were provided, [10]. Programming languages provide various ways to achieve coupling. If measures of coupling are specific to programming languages then they should measure the various ways to achieve coupling within these languages.

Briand *et al.* have explicitly highlighted the need for such metrics, [4]. These metrics provide insights into the modular structure of software systems. Briand *et al.* partly addressed this issue by defining metrics that focused on three different aspects of coupling, [6]. In doing so they defined metrics based on inheritance and the friend construct. An empirical study, utilising these metrics led the authors to conclude that: “there seems to be evidence that the use of ‘friend’ classes in C++ increases fault-proneness of classes even more than other types of coupling”.

However, the metrics defined by Briand *et al.*, based on the friend construct do not accurately measure the contribution of the friend relationship to the overall coupling of the associated classes. In fact these metrics simply count the overall coupling of the associated classes and not the coupling that the friend relationship is responsible for. Measures of language-specific features should consider the nature of these features. This suggests that measures of language-specific features need to measure the interactions for which a feature is necessary and sufficient.

3. Coupling Measurement Frameworks

In the context of coupling metrics, Wilkie and Kitchenham, [33], have highlighted the different levels of granu-

larity that can be associated with the coupling measure, i.e. some metrics count linkages at a class level, whereas others count connections between member functions and consider the types of the parameters of these functions. In [34], they propose a framework for the refinement of Coupling Between Objects (CBO), a coupling measure defined by Chidamber and Kemerer, [9]. Coupling metrics based on the first two stages of this refinement model were proposed by [32] and [34] respectively. These metrics do not take the nature of the interactions between classes into account.

Briand *et al.*, [6], extended the underlying concepts of the metrics from [34] to address this issue. They defined coupling metrics which consider the *locus* or direction of coupling through the notion of import and export coupling. In addition, they consider two other dimensions of coupling measurement: *relationship* and *type*. ‘Friend’, ‘inheritance’ and ‘other’ are the three forms of relationship. The type of relationship categorises how classes are connected, class-attribute, class-method and method-method are the three types of coupling considered.

The definition of Briand *et al.*’s metrics based on method-method interactions are given below. In these definitions $MM(d, c)$ stands for the method-method interactions from class d to class c . $Friends(c)$ is the set of classes that are declared as friends of class c . $Friends^{-1}(c)$ is the set of classes that declare c as a friend class. $Others(c)$ are all other classes in the system except $Friends(c)$, $Friends^{-1}(c)$, $Ancestors(c)$, $Descendants(c)$ and c itself. Import Coupling and export coupling are denoted by IC and EC respectively. These metrics were defined formally in [5].

- $FMMEC(c) = \sum_{d \in Friends(c)} MM(d, c)$.
- $DMMEC(c) = \sum_{d \in Descendants(c)} MM(d, c)$.
- $OMMEC(c) = \sum_{d \in Others(c) \cup Friends^{-1}(c)} MM(d, c)$.
- $IFMMIC(c) = \sum_{d \in Friends^{-1}(c)} MM(c, d)$.
- $AMMIC(c) = \sum_{d \in Ancestors(c)} MM(c, d)$.
- $OMMIC(c) = \sum_{d \in Others(c) \cup Friends(c)} MM(c, d)$.

Briand *et al.* and Wilkie and Kitchenham have examined the usefulness of their metrics as predictors of external quality attributes. Briand *et al.* have shown empirically that design constructs like friendship influence the fault-proneness of classes, [6]. However, the metrics utilised are coarse-grained as illustrated in the following section. More sophisticated metrics which associate interactions between classes with the design construct which enables the interaction (e.g., the friend construct is needed to access the private members (that is, methods or attributes) of a class) can be used to refine the measurements which have been used in

the past and to show the coupling mechanisms more likely to be responsible for fault-proneness. Some refined metrics for measuring friendship accurately are defined in this paper and will be used to assess any links between friendship and external attributes of systems in future work.

4. Coarse-grained nature of Existing Metrics

The following example illustrates the limitations (in terms of coarseness) of the Briand *et al.* metrics. With respect to this framework the focus here is on the ‘method-method’ type dimension. ‘Method-Attribute’ interactions were not considered by Briand *et al.*, [6], but are in the refined metrics defined in section 5 below. The other type dimensions, (class-attribute and class-method) are not considered here since one of the main aims of this work is to examine the use of the friend mechanism. This mechanism is not utilised until an interaction between a method in one class and a member of another class occurs.

Consider the diagram in figure 1. In this figure class *c* is declared as a friend of class *a* and class *a* declares class *c* as a friend. Class *c* also inherits from class *a*, so class *c* is a derived class of *a* and therefore *c* is also a descendant of class *a*.

Based on the definitions above, the method-method interaction coupling metrics of Briand *et al.* for class *c* in figure 1 are presented in table 1. The notation used in figure 1 is based on the Buhr diagrams from Ada, [7].

This example highlights a couple of interesting facts in relation to these metrics.

- Intuitively one might expect that a complete measure of export coupling based on method-method interactions could be achieved by adding the 3 export coupling metrics (FMMEC, DMMEC and OMMEC). However, if a class is both a descendant and a friend of a class then the interactions between these classes will contribute to two metrics. In the example above since class *d* is both a descendant and a friend of class *c*, the interactions from class *d* to class *c*, (*E3*, *E4*), contribute to the two metrics, (FMMEC(*c*) and DMMEC(*c*)). If the three export coupling metrics were combined into one measure of export coupling for class *c*, then the sum of the three metrics (8) described here would provide a result greater than that logically expected. The total number of export coupling method-method interactions for class *c* is 6. This inconsistency is inappropriate from a measurement theory perspective.
- If a method is defined and implemented as a private member of a class, then a descendant class needs the friend relationship to access this private member directly, that is, there are some interactions between the

descendant class *d* and class *c* for which inheritance is not sufficient. In the example in figure 1 the interaction *E4* is in this category and is included in the DMMEC(*c*) metric. While this interaction is occurring between classes in an inheritance hierarchy, it seems inappropriate to include it in DMMEC(*c*) since the inheritance relationship isn’t sufficient to allow *E4*.

- For classes not related through inheritance, the friend relationship is not required for all interactions. In the example in figure 1, the interactions *E1* and *E2* both contribute to FMMEC(*c*). However, while *E1* is an interaction between classes related by the friend mechanism, the declaration of class *f* as a friend of class *c* is only required for the interaction *E2* which accesses a private method of class *c*.

A similar situation to this arises for import coupling where the interactions considered in evaluating the import coupling of class *c* are in the opposite direction (from class *c* to other classes in the system). For example, in the case of class *c* in figure 1, its import coupling interactions are with its ancestors (class *a*), classes where *c* is declared as a friend, (class *a* and class *g*), and the classes in *Others*(*c*), (class *h*). The limitations described here are addressed by the refined metrics presented below.

5. Definition of Fine-Grained Metrics based on the Briand *et al.* framework

The metrics defined in this section are refinements of the metrics based on method-method interactions defined by Briand *et al.*, [6]. The naming convention used here is adapted from the Briand *et al.* approach. For each metric defined by Briand *et al.* there are three metrics defined here and each of their names is preceded by ‘O_’ to emphasise that the refined metrics *only* calculate interactions based on the associated relationship: inheritance, friend, other. In the following definitions only the ‘MP’ (method-procedure), (that is, method-method) interactions are defined. The ‘MA’ (method-attribute) interactions can be similarly defined and the ‘MM’ (method-member) interactions are the sum of the ‘MA’ and ‘MP’ interactions. The definition of ‘MM’ interactions along with ‘MP’ and ‘MA’ interactions facilitates the use of this composition measure whenever appropriate.

In the following definitions *HMPI*(*c*, *d*) stands for the set of **hidden method-method (procedure) interactions** from class *c* to class *d*. *VMPI*(*c*, *d*) stands for the **visible method-method (procedure) interactions** from class *c* to class *d*. These sets of interactions are dependent on the relationship(s) between the classes, (that is, is one class a friend of the other, or is one class a derived or descendant class of the other). They are also dependent on the

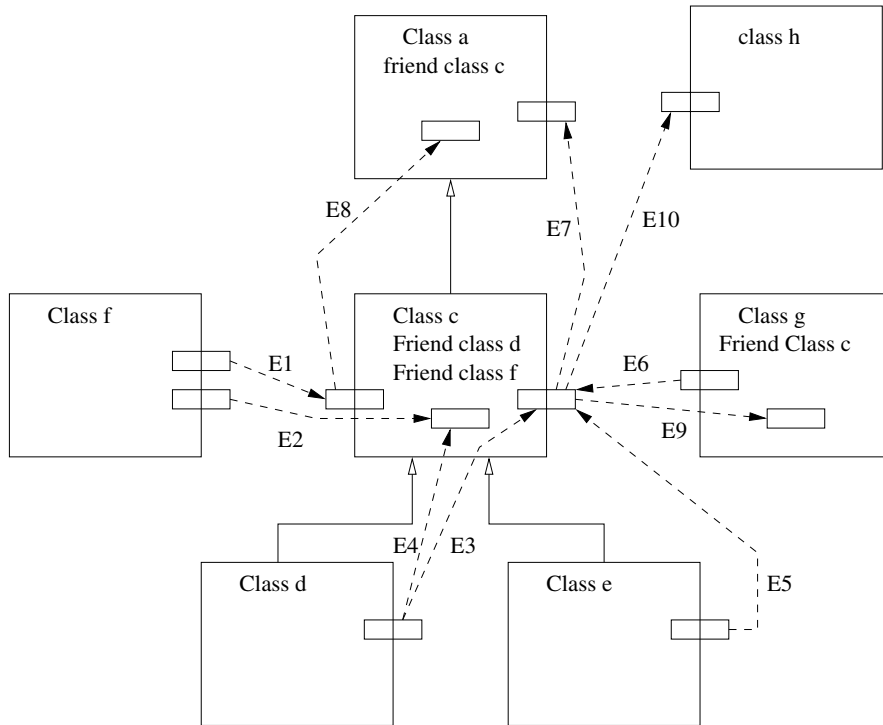


Figure 1. Illustration of Existing and Refined Metrics

Metric	Value	M-M Interactions
FMMEC	4	E1,E2,E3,E4
DMMEC	3	E3,E4,E5
OMMEC	1	E6
IFMMIC	3	E7,E8,E9
AMMIC	2	E7, E8
OMMIC	1	E10

Table 1. Metric Values for class c in figure 1

accessibility, (whether private, protected or public), of the called method within the server class d . A method is hidden within the server class if a friend relationship is necessary to facilitate direct access to this method. $HMPI(c, d)$ and $VMPI(c, d)$ are defined formally in [17].

5.1 Import Coupling Class Metrics

The term import refers to the direction of the interactions between classes. Import coupling interactions can exist between a class and the classes where it is declared a friend and between a class and its ancestors. The refined import coupling metrics are defined in this section.

5.1.1 O_IFMPIC

O_IFMPIC for a class c counts method-method interactions from c to all classes which declare c as a friend *and for which the friend declaration is necessary*. The emphasised text here distinguishes this metric from the corresponding metric IFMMIC defined by Briand *et al.*. It is expected that O_IFMPIC will be much smaller than IFMMIC because we believe that only a small proportion of all interactions between classes involve accessing hidden members of a class.

In the example in figure 1 O_IFMPIC(c)=2. The edges $E8$ and $E9$ contribute to this metric. Note that IFMMIC(c)=3 (see table 1).

O_IFMPIC is formally defined as:

$$O_IFMPIC(c) = \sum_{d \in Friends^{-1}(c)} | HMPI(c, d) |$$

5.1.2 O_AMPIC

O_AMPIC counts method-method interactions between a class and its ancestors for which an inheritance relationship is sufficient. This metric considers interactions with ancestor classes. A class c can access some of the members of its ancestor classes directly. O_AMPIC counts these interactions. Those members which are declared private or inherited privately in the ancestor class are not included in this metric since the friend mechanism is needed to gain access to these members.

O_AMPIC(c)=1 ($E7$) in the example in figure 1. This metric will probably be quite similar to the metric AMMIC since a difference only occurs if a class is a friend as well as being a descendant of another class.

O_AMPIC is formally defined as:

$$O_AMPIC(c) = \sum_{d \in Ancestors(c)} | VMPI(c, d) |$$

5.1.3 O_OMPIC

O_OMPIC counts method-method interactions between a class and classes (other than ancestors of the class) for which a friend declaration is not necessary.

O_OMPIC(c)=1 ($E10$) in the example in figure 1. It is expected that O_OMPIC will be larger and both O_AMPIC and O_IFMPIC should, in general, be smaller than the corresponding Briand *et al.* metrics. The greatest change will be evident in the O_OMPIC and O_IFMPIC metrics.

O_OMPIC is formally defined as:

$$O_OMPIC(c) = \sum_{d \in \{C - \{c \cup Ancestors(c)\}\}} | VMPI(c, d) |$$

5.2 Export Coupling Class Metrics

In this section the export coupling metrics which are a refinement of the Briand *et al.* export coupling metrics are defined. The calculation of these metrics is illustrated for class c in figure 1. As with the import coupling metrics only metrics based on method-method(MP) interactions are defined here. The method-attribute(MA) based metrics are defined similarly and the method-member(MM) metrics are the sum of the corresponding MP and MA metrics.

5.2.1 O_FMPEC

O_FMPEC for a class c should count method-method interactions for which the friend declaration is necessary between c and the classes that declare c as a friend.

O_FMPEC(c)=2 ($E2$ and $E4$). It is anticipated that this metric will return small values in general since it is based on interactions with hidden methods only. It is expected that only a small number of the methods in these classes will interact with hidden methods. This metric will generally be smaller than the corresponding metric FMMEC since only a subset of the interactions considered for FMMEC are considered for O_FMPEC.

O_FMPEC is formally defined as follows:

$$O_FMPEC(c) = \sum_{d \in Friends(c)} | HMPI(d, c) |$$

5.2.2 O_DMPEC

O_DMPEC should count method-method interactions between the descendants of a class and the class itself for which an inheritance relationship is sufficient.

O_DMPEC(c)=2 ($E3$ and $E5$). It is likely that this metric will not be significantly different from DMMEC. In fact there will only be a difference between these metrics if some descendant class(es) are also declared as friends of the class itself, (c in this case).

O_DMPEC(c) is formally defined as follows:

$$O_DMPEC(c) = \sum_{d \in Descendants(c)} |VMPI(d, c)|$$

5.2.3 O_OMPEC

O_OMPEC should count method-method interactions for which a friend declaration is not necessary between classes (except descendants) and the specific class.

O_OMPEC(c)=2 (E1 and E6). In general the value of this metric will be larger than OMMEC since some interactions from classes declared as friends of the class might also be included.

O_OMPEC(c) is formally defined as follows:

$$O_OMPEC(c) = \sum_{d \in \{C - \{c \cup Descendants(c)\}\}} |VMPI(d, c)|$$

Because of the preciseness of these metrics, they can be combined into complete measures of import and export coupling by adding the appropriate metrics. If these import and export coupling metrics are called **Method-Member Import Coupling (MMIC)** and **Method-Member Export Coupling (MMEC)** respectively, then:

MMEC(c)=6, (E2, E4, E3, E5, E1, E6) and MMIC(c)=4, (E8, E9, E7, E10) for the example in figure 1.

6. Illustration of the fine-grained nature of the refined metrics

In this section Briand's metrics are compared with the more refined metrics that have been derived from this metrics set and are defined in section 5 using the LEDA software system, [1]. In each graph in this study the horizontal axis of the graph represents either the classes declared as friends (for import coupling metrics) or the classes declaring friends (for export coupling). A number on this axis represents a class but the number associated with each class has no importance (classes were ordered alphabetically according to their names and assigned a number in this way). Classes maintain the same number in all graphs for either import or export coupling so metrics can be compared across these graphs also. The vertical axis represents the count of the associated metrics.

6.1 Illustration of Import Coupling Metrics

The metric IFMMIC, [6], is an import coupling metric which focuses on method-method interactions between a

class which is declared as a friend and the classes which declare it as a friend. All method-method interactions between these classes are considered as part of this metric. However, not all of these interactions are based on friendship, i.e. the friend relationship is not needed for many of them. In addition, interactions with attributes of classes are not considered in this metric. On the other hand the refined metric O_IFMMIC rectifies these two issues, first only interactions for which the friend relationship is necessary are considered and second, interactions with members (methods and attributes) of classes, (other than methods) are also considered. Figure 2 illustrates the distribution of these metrics for the LEDA software system. LEDA has 137 classes which are declared as friends and these are represented by numbers on the horizontal axis of figure 2. Therefore the increasing values of the x-axis have no significance.

- It is clear from figure 2 that the refined metric O_IFMMIC is bigger for many classes than the metric IFMMIC. On this basis, the contribution to the metric O_IFMMIC of accesses to hidden attributes must outweigh the contribution of accesses to visible methods in IFMMIC.

The actual extent to which access to attributes and methods contribute to the refined metric is illustrated by the refined metrics O_IFMAIC and O_IFMPIC. It might be anticipated given figure 2 that the proportion of hidden methods in classes would be quite small in comparison with attributes, and thus resulting in small values for O_IFMPIC. The weak correlations in table 2 for the import coupling metrics also illustrate that these metrics measure different aspects of coupling.

- As can be seen from figure 2 the metrics O_IFMMIC and O_IFMAIC are very similar for many classes. In fact for most classes no difference exists. This highlights the clear and important difference between the fine-grained metrics defined in the previous section, (c.f. section 5) and the metrics of Briand *et al.*, [6]. For some classes Briand's IFMMIC metric returns values in excess of 100, (c.f. figure 2). The corresponding metric O_IFMPIC returns values which are either zero or close to zero, (Recall that O_IFMMIC = O_IFMAIC + O_IFMPIC, (c.f. section 5)). The refined metric O_IFMPIC only considers method-method interactions for which friendship is necessary, whereas IFMMIC counts all method-method interactions from a class declared as a friend.

7. Illustration of Export Coupling Metrics

Given the analysis based on import coupling metrics presented above, it is expected that the analysis of classes

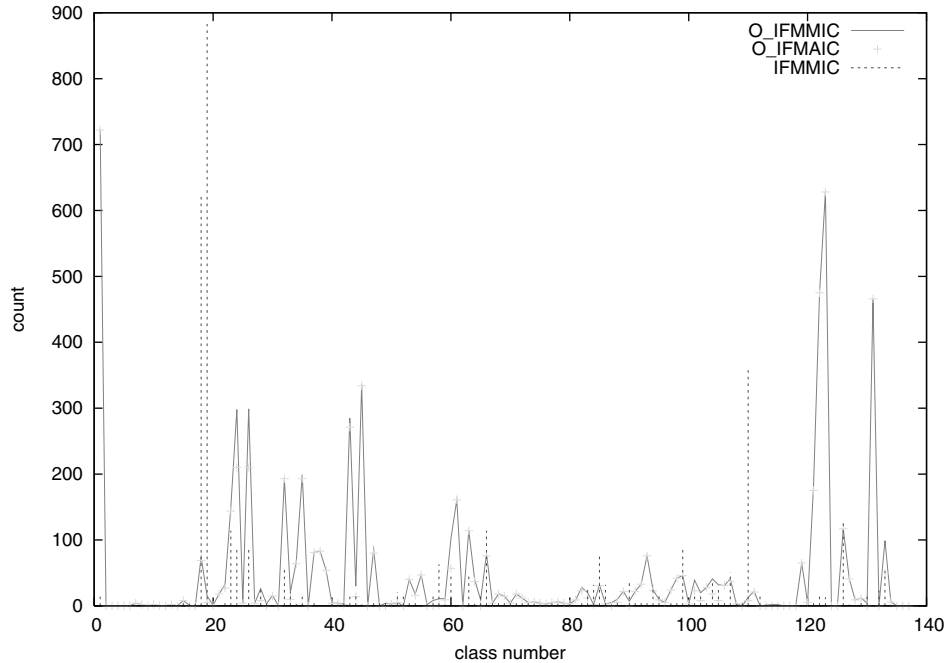


Figure 2. The distribution of O_IFMMIC, O_IFMAIC and IFMMIC for LEDA

Metric1	Metric2	<i>r</i>
<i>O_IFMMIC</i>	<i>O_IFMPIC</i>	.422**
<i>O_IFMPIC</i>	<i>O_IFMAIC</i>	.232**
<i>O_IFMAIC</i>	<i>IFMMIC</i>	.421**
<i>O_OMMIC</i>	<i>O_OMAIC</i>	.259**
<i>O_OMPIC</i>	<i>O_OMAIC</i>	.180
<i>O_OMAIC</i>	<i>OMMIC</i>	.262**
<i>O_FMMEC</i>	<i>O_FMPEC</i>	.325**
<i>O_FMPEC</i>	<i>O_FMAEC</i>	.121
<i>O_FMAEC</i>	<i>FMMEC</i>	.330**
<i>O_OMMEC</i>	<i>O_OMAEC</i>	.259**
<i>O_OMPEC</i>	<i>O_OMAEC</i>	.083
<i>O_OMAIC</i>	<i>OMMEC</i>	.262

** denotes a significant correlation at the 1% level

Table 2. Spearman Correlation Coefficient (*r*) for metrics

declaring friends using export coupling metrics should reinforce these findings. The refined metrics should highlight that it is indeed access to hidden attributes in these classes which is the most significant contributor to the friend based coupling. However, it is still unclear how the metric values are composed. Figure 3 illustrates the proportion of the refined friend based export coupling metrics, based on method-member interactions and method-attribute interactions. As with import coupling the number of method-method interactions is the difference between the number of method-member and method-attribute interactions. Briand *et al.*'s metric FMMEC is also shown in figure 3.

- The metric O_FMMEC is bigger than the metric FMMEC for many classes. Again this results from the fact that interactions with hidden attributes are more common in classes than all method-method interactions. However there are also a significant number of classes where both metrics are zero, indicating redundancy in friend class declarations. There are also some classes where FMMEC is bigger than O_FMMEC. In these classes many non-hidden methods are accessed. However, this analysis shows that for classes declaring friends, interactions based on friendship are typically more prevalent than all the method-method interactions between friend-related classes. The lack of a correlation between the pairs of export coupling metrics further highlights that these metrics measure dif-

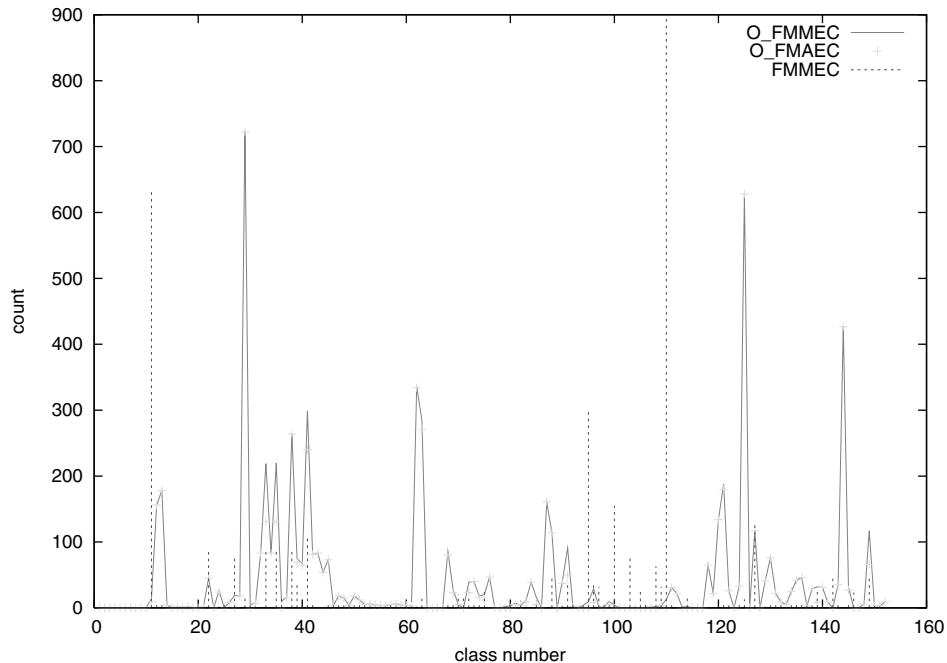


Figure 3. Distribution of O_FMMEC, O_FMAEC and FMMEC for LEDA

ferent aspects of coupling.

- The metric O_FMMEC is zero for most classes, indicating that no hidden methods are accessed. There are a few exceptions to this rule, for example there are approximately 100 interactions with hidden methods of two classes (the classes with numbers 34 and 36 on the x-axis of figure 3). In general, this suggests that, for classes declaring friends, interactions with hidden methods of these classes are not a significant contributor to the friend-based coupling of these classes.
- At this point it is also worth noting that the friend mechanism is used to access internal classes in the LEDA system¹. Internal classes are those which generally have no public interface. In these circumstances such classes would declare friend class relationships. As a result friend based interactions with these classes would be recorded by the refined metric O_FMMEC. If these interactions involve method-method interactions then they would also be recorded by Briand *et al.*'s metric FMMEC. However, FMMEC will include interactions with visible methods of a class. On the other hand O_FMMEC only includes interactions with hidden members and O_OMMEC includes interactions with visible members. Therefore the refined metrics can be used to characterise internal classes since

¹suggested via personal communication with the developers of LEDA

O_OMMEC=0 and O_FMMEC will be greater than zero.

- The FMMEC metric records all method-method interactions with classes declaring friends. As can be seen in figure 3, the metric FMMEC is close to zero for many classes. This suggests that only a small number of interactions access methods in the classes declaring the friend. Some of these classes may be internal classes. A re-examination of the source-code confirms this. For example O_FMMEC for class number 29 is 724 and this class has no public members. This is also the case for classes numbered 144, 63 and 62 where O_FMMEC is 427, 285 and 336 respectively.

7.1 Illustration of Refined Measurement of Friends and Inheritance

The refined metrics provide a deeper insight into the use of C++ language constructs, especially the friend and inheritance mechanisms as the examples in this section illustrate.

Stroustrup has suggested that the friend mechanism might be misused to give a derived class access to private members in the parent class which should have been assigned protected access, [28]. Differences between the refined metrics and those defined by Briand highlight where there is potential for this situation to arise as the following example illustrates.

- As pointed out in sections 5.1.2 and 5.2.2, the refined metrics based on method-method interactions between a class and its ancestors (O_AMPIC) and descendants (O_DMPEC) will generally be very similar to the corresponding Briand *et al.* metrics (AMMIC and DMMEC respectively). This claim holds true for the LEDA system as illustrated in figure 4 which illustrates the export coupling metrics.
- Classes that are related through both inheritance and the friend mechanism will return different results. For example, for class number 58 in figure 4, AMMIC=63 and IFMMIC=63. This class is both a friend and a derived class of its only parent class. However, O_AMPIC=53 and O_IFMPIC=10. Of the 54 method-member interactions counted in O_AMMIC, 53 are method-method interactions which are also counted in AMMIC. For this class O_IFMPIC=10, which means that there are 10 method-method interactions between these classes, where the called method is hidden. Therefore, there are a total of 63 method-method interactions between these classes, corresponding to Briand's metric AMMIC.

Another difference which arises here is that all these method-method interactions also contribute to the metric IFMMIC, which means that Briand *et al.*'s metric IFMMIC=63 also. The corresponding refined metric O_IFMMIC=11 and, as stated above, there are 10 method-method interactions in this category, (O_IFMPIC=10).

- The refined and existing metrics that consider interactions classified as 'other' are quite similar as expected, with one or two exceptions. For example, for one class, (class number 18 in figure 2), O_OMMIC=2479 and OMMIC=1606. The difference here is that 883 of the methods accessed by this class are in a friend class but the friend declaration is not needed to access them, that is, they are not hidden. However, they are still counted by Briand's IFMMIC metric, (see figure 2). The refined metric O_OMMIC which correctly classifies them as interactions for which the friend mechanism is not required, includes them.

8 Conclusion

The need for metrics which measure different variations of coupling has been highlighted in this paper. In previous research the widespread use of the friend construct has been highlighted. In addition, the bluntness of existing measures of the friend mechanism has been highlighted.

A set of metrics which are refinements of the research of Briand *et al.*, [6], have been presented in this paper. These

metrics associate each interaction between classes with its appropriate coupling mechanism. The metrics defined here measure method-member and method-attribute interactions along with method-method interactions.

Our analysis indicates that metrics based on 'friend' and 'other' types of coupling are significantly different to existing metrics due to the precise measurement of these mechanisms. Thus, these metrics may be useful in prediction models. As illustrated in this paper these metrics are also useful to guide a deeper analysis of the structure of software systems.

Our future work will involve utilising these refined metrics in prediction models of external quality attributes of software to evaluate the practical value in terms of improved prediction of external quality attributes. These metrics may also be useful for identifying a cause-effect relationship between internal and external attributes of software. Such an analysis of the qualitative and quantitative approaches to empirical software engineering.

References

- [1] Algorithmic Solutions Software Gmbh. LEDA: Library of Efficient Data Types and Algorithms. <http://www.algorithmic-solutions.com/enleda.htm>, 2006.
- [2] J. Bieman and J. Zhao. Reuse Through Inheritance: a Quantitative Study of C++ Software. In *SSR '95: Proceedings of the 1995 Symposium on Software Reusability*, pages 47–52, New York, NY, USA, 1995. ACM Press.
- [3] G. Booch. *Object Oriented Design with Applications*. The Benjamin/Cummings Publishing Company Inc., 1991.
- [4] L. Briand, J. Daly, V. Porter, and J. Wüst. A Comprehensive Empirical Validation of Design Measures for Object-Oriented Systems. In *Fifth International Symposium on Software Metrics*, pages 246–257, November 1998.
- [5] L. Briand, J. Daly, and J. Wüst. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, January/February 1999.
- [6] L. Briand, P. Devanbu, and W. Melo. An Investigation into Coupling Measures for C++. In *International Conference on Software Engineering*, pages 412–421, 1997.
- [7] R. Buhr. *System design with Ada*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984.
- [8] M. Cartwright. An empirical view of inheritance. *Information and Software Technology*, 40(14):795–799, 1998.
- [9] S. Chidamber and C. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Trans. Software Eng.*, 20(6):476–493, 1994.
- [10] N. Churcher and M. Shepperd. Comments on A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 21(3):263–265, March 1995.
- [11] J. O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley Longman Publishing Co., Inc., 1992.
- [12] S. Counsell and P. Newson. Use of Friends in C++ Software: An Empirical Investigation. *Journal of Systems and Software*, 53(1):15–21, 2000.

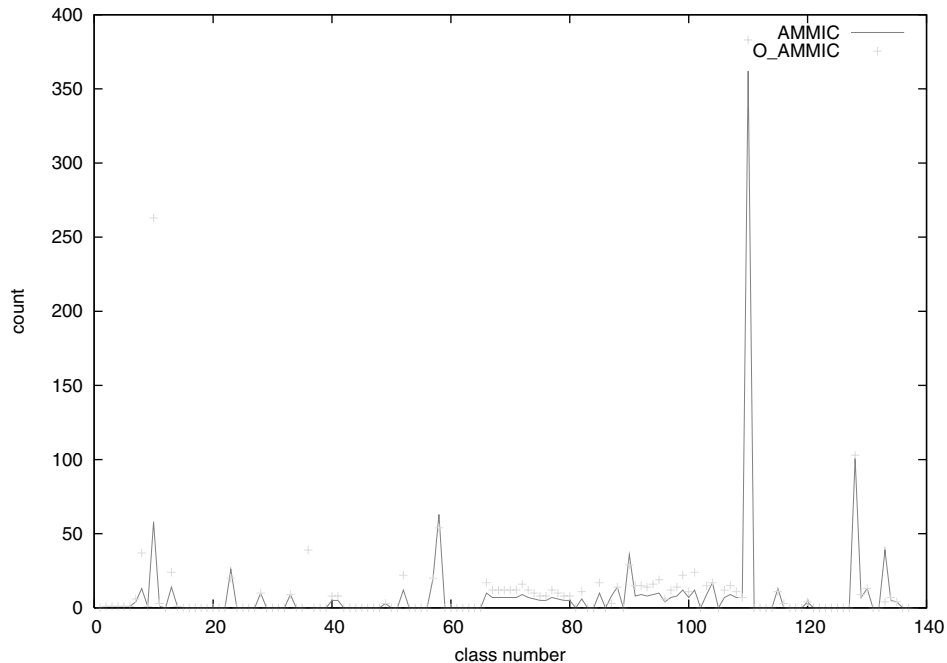


Figure 4. Comparison of distribution of AMMIC and O_AMMIC in LEDA

- [13] S. Counsell, P. Newson, and E. Mendes. Design Level Hypothesis Testing Through Reverse Engineering of Object-Oriented Software. *International Journal of Software Engineering*, 14(2):207–220, 2004.
- [14] I. Deligiannis. A Review of Experimental Investigations in Object-Oriented Technology. *Empirical Software Engineering*, (7):193–231, 2002.
- [15] R. Dromey. A Model of Software Product Quality. *IEEE Transactions on Software Engineering*, 21(2):146–162, February 1995.
- [16] R. Dromey. Cornering the Chimera. *IEEE Software*, 13(1):33–43, 1996.
- [17] M. English. *An Analysis of Coupling Mechanisms in C++ Software using Fine-Grained Software Metrics*. PhD thesis, University of Limerick, 2007.
- [18] M. English, J. Buckley, and T. Cahill. Applying Meyer’s Taxonomy to Object-Oriented Systems. In *Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 35–44, September 2003.
- [19] M. English, J. Buckley, and T. Cahill. A Friend in Need is a Friend Indeed. In *Fourth International Symposium on Empirical Software Engineering*, pages 469–478, 2005.
- [20] ISO/IEC. *ISO/IEC 9126-1 Software Engineering -Product Quality- Part 1: Quality model*, 2001.
- [21] B. Kitchenham and S. Pfleeger. Software Quality: The Elusive Target. *IEEE Software*, 13(1):13–21, 1996.
- [22] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics*. Prentice-Hall, Englewood Cliffs, N.J., 1994.
- [23] J. McCall, P. Richards, and G. Walters. Factors in Software Quality. Technical Report Ad/A-049-014/015/055, National Technological Information Service, 1977.
- [24] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1997.
- [25] S. Meyers. *Effective C++ — 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley Longman Publishing Co., Inc., second edition edition, 1998.
- [26] M. Page-Jones. *What Every Programmer Should Know About Object-Oriented Design*. Dorset House Publishing, 1995.
- [27] G. Singh. Single versus Multiple Inheritance in Object Oriented Programming. *OOPS Messenger*, 6(1):30–39, 1995.
- [28] B. Stroustrup. *The Design and Evolution of C++*. ACM Press/Addison-Wesley Publishing Co., 1994.
- [29] R. Subramanyam and M. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on Software Engineering*, 29(4):297–310, 2003.
- [30] A. Taivalsaari. On the Notion of Inheritance. *Communications of the ACM*, 1996.
- [31] D. Watt. *Programming Language Design Concepts*. John Wiley & Sons, 2004.
- [32] F. Wilkie and B. Hylands. Measuring Complexity in C++ Application Software. *Software - Practice and Experience*, 28(5):513–546, April 1998.
- [33] F. Wilkie and B. Kitchenham. Coupling Measures and Change Ripples in C++ Application Software. *The Journal of Systems and Software*, 52:157–164, 2000.
- [34] F. Wilkie and B. Kitchenham. An Investigation of Coupling, Reuse and Maintenance in a Commercial C++ Application. *Information & Software Technology*, 43(13):801–812, 2001.
- [35] C. Willis. Analysis of Inheritance and Multiple Inheritance. *IEEE Software Engineering Journal*, July 1996.