An Analysis of the Use of Friends in C++ Software Systems

Michael English,[1] Jim Buckley,[1] Tony Cahill,[1] and Kristian Lynch[2]
[1]Department of Computer Science and Information Systems, University of Limerick, Ireland.
[2]Dept of Endocrinology, Lund University, Sweden.
{Michael.English, Jim.Buckley, Anthony.Cahill}@ul.ie
Kristian.Lynch@endo.mas.lu.se

Abstract

A commonly held belief is that a friend is a violation of encapsulation and as a consequence its use would result in poorly designed and difficult to understand software. However, little empirical evidence exists to support this claim.

This paper presents a study which assesses the design implications of including friendship in a system. A number of hypotheses are investigated based on previous work in this area by Counsell and Newson, [9]. Our initial findings suggest that classes declared as friends are coupling hotspots, that the more friends a class has the more protected and private members it will contain and that friendship is not used to access inherited protected members.

1 Introduction

Ellis and Stroustrup, [14], define a friend of a class as a function "that is not a member of the class but is permitted to use the private and protected member names from the class". However, a class B can also be defined as a friend of class A, allowing all the members of class B to access the private and protected members of class A. The friendship relationship is not symmetric. In other words, B can be a friend of A without A being a friend of B.

Different perspectives on the appropriateness of the friend mechanism exist in the literature. Coplien, [8], claims that the encapsulation provided by the protected and private mechanisms is violated by friendship. Page-Jones, [21], describes the use of friend functions to access the variables of a class directly as a "design sin", something which is usually done to improve efficiency. Similar perspectives can be found in [11], [19] and [15].

In contrast Stroustrup, [23], defends the friendship mechanism, describing it as "one protection domain granting a read-write capability to another". He argues that the declaration of a friend is part of the class declaration and rejects the argument that friendship violates encapsulation as "a combination of ignorance and confusion with non-C++ terminology". Meyers, [20], also holds the view that friends can be considered part of the class's interface, but recommends that friends should be avoided whenever possible. His advice is that if a method can be implemented in terms of the class's public interface, then it should not be made a friend. Booch, [2], in commenting on object modeling, suggests that a module's interface should be as narrow as possible and thus that friends should be chosen wisely, since friendship implies a certain trust between classes.

However, limited empirical analysis of actual software systems has been undertaken to support or reject the many claims that have been made in the literature about the friendship mechanism. Indeed, little evidence exists about its usage in the development of object-oriented software systems, its impact on other language features, or its impact on external quality characteristics such as program understanding and maintenance. One exception is Briand *et al.*, [4], who studied the relationship between various design measures and the probability of fault detection in classes during testing. They conclude that coupling based on friendship is likely to increase the likelihood of a fault.

One way to evaluate the impact of the friendship mechanism on high-level external attributes of software is to employ automated analysis to assess the effect friendship has on lower-level attributes. Then, these lower-level attributes may be used as predictors of high-level external attributes, [1]. One such study was undertaken by Counsell and Newson, [9]. They examined a number of hypotheses in relation to the use of the friend mechanism with respect to lower-level attributes in the course of their work and suggested that friendship was used as an alternative to inheritance. However, this study did not take a number of syntactic restrictions inherent in C++ into account in evaluating their hypotheses. In addition they disregarded the asymmetry of the friendship mechanism in their analysis of coupling. These were important factors in the formulation and evaluation of their hypotheses. This paper discusses several refinements to the Counsell and Newson experiment.

Section 2 looks at operator overloading and coupling in C++. In Section 3 the Counsell and Newson study, [9], is described in detail. Our empirical study is discussed in Section 4, along with the rationale which underpin the hypotheses under study and the differences between these hypotheses and those of the previous empirical study, [9]. An overview of the software systems analysed and the different metrics for which data has been collected, to facilitate evaluation of the hypotheses is also provided. Section 5 describes the statistical analysis performed and evaluates the hypotheses based on the statistical data and discusses the results. Section 6 discusses related work and section 7 concludes the paper with a summary of results and discussion of future work.

2 Friends in C++ Software

In evaluating the friend mechanism in C++ one must be cognizant of a number of syntactic restrictions in the language and of the asymmetry of the friend relationship. Section 2.1 discusses the syntactic restrictions with respect to achieving operator overloading in C++, illustrating that friendship is a valid approach in this instance. Section 2.2 discusses the effect the asymmetric relationship has on measuring coupling.

2.1 Friends and Operator Overloading in C++

As mentioned above, syntactic restrictions within the C++ language influence the usage of the friendship mechanism in C++ software. Consider, for example, scalar multiplication of a vector by a double. The binary '*' operator needs to be defined for the vector class. If this operator function is implemented as a member function, then the function takes just one argument, the double, which becomes the right-hand operand of the binary '*' operator. The left-hand operand must be an object of the vector class itself, upon which the operator is called. A call to this function would look as follows:

```
vector a;
double x;
...
a*x;
```

This call to the operator is interpreted as:

```
a.operator*(x);
```

To facilitate the scalar multiplication of vectors and doubles by making the double the left operand of the '*' operator, the operator function must be defined as a non-member function as follows:

```
vector operator*(double x, vector& a)
  { ...}
```

If, in addition to this the function needs direct access to the private or protected members in the vector class, then the function operator* must be made a friend of the class.

2.2 Friends and Coupling in C++

Due to the asymmetry of friendship, we believe that use of the friendship mechanism will only impact the coupling for a class which is declared as a friend. This is a contentious assertion but can be supported, based on the variety of definitions of coupling. Briand *et al.*, [4], have looked at coupling issues and specifically have defined metrics based on friendship, taking into account the one-way nature of the friend mechanism. They say that "class A is coupled to class B if A uses B's member functions and/or instances variables".

It should be noted that there seems to be some variation in the literature as to the definition of Coupling Between Object Classes, (CBO). Chidamber and Kemerer, [6], define CBO, stating that "any evidence of one class using methods or instance variables of another constitutes coupling". A class which is declared a friend of another class, has the potential to use more members directly, (the protected and private members of the class in which it is declared). Hence the non-friend coupling of the class declared a friend might be effected by the friendship, since this relationship is providing a form of coupling to the class.

It is acknowledged that other definitions of coupling do exist. For example, Briand *et al.*, [3], define coupling formally as: $CBO(c) = |\ \{d \in C - \{c\}\ |\ uses(c,d) \vee uses(d,c)\}\ |$ and this can be interpreted as follows. The coupling of class c equals the number of classes, d, in the system C, except class c itself where c uses d or vice versa. A class c uses a class d "if a method implemented in class c references a method or an attribute implemented in class d". This formal definition measures coupling as a two-way relationship. Another alternative definition is that of Harrison *et al.*, [15], where they distinguish between CBO and another metric, the Number of Associations, (NAS). They state that "if a class uses a method of another class more than once, then the CBO metric will count each usage as a separate occurrence of coupling".

3 Counsell and Newson Study

Counsell and Newson, [9], in their study of the impact of friends within C++ systems examined the following five hypotheses:
H1: The more friends in a class, the less (non-friend) coupling found in that class.

The rationale underlying this hypothesis, according to Counsell and Newson is that "the use of friends in a class satisfies the coupling needs of the class". The coupling of a class was measured by the Number of Associations, (NAS), which is the number of lines emanating from a class on an OMT design diagram, [22]. This metric includes direct inheritance in its definition.

H2: The more friends in a class, the more methods found in that class.

Counsell and Newson expected that friends would be used in classes which have a large number of methods. As a result the friends would be able to take advantage of the functionality provided by the methods in these classes. The Number of Methods per Class, (NMC), metric was used to count the number of methods in each class and this was compared with the number of friends in the class.

H3: Classes declared as friends of other classes have less inheritance than other system classes.

This hypothesis was based on the rationale that a class declared as a friend of another class might have less inheritance, since it would have taken advantage of the inheritance in the class which contains it as a friend. This hypothesis was evaluated by examining the percentage of classes declared as friends which engage in inheritance and comparing this with the percentage of inheritance-based classes in the system as a whole.

H4: Classes containing friends that engage in inheritance have fewer descendants than other system classes.

The rationale underpinning this hypothesis is that the lower a class is in the inheritance tree, the more functionality this class will have inherited. As a result it might be more advantageous to declare a friend of a class lower in the hierarchy, since the friend, in turn, will have access to more functionality. The Number of Descendants, (NOD), metric was used to evaluate this hypothesis. The NOD is a measure of the number of classes below a class in a hierarchy. The mean NOD for classes containing friends and engaged in inheritance was compared with the mean NOD for all system classes.

H5: Classes that do not engage in any inheritance have more friends than classes which do engage in inheritance.

The rationale behind this hypothesis was that classes which do not engage in inheritance use friends as an alternative to inheritance. Thus stand-alone classes would contain more friends. This hypothesis was studied by comparing the proportion of stand-alone classes containing at least one friend with the proportion of classes which are engaged in inheritance and which contain at least one friend.

These hypotheses were tested on four software systems: LEDA, a library of efficient data types and algorithms, the GNU C++ Class Library, Edge, which is a graph editor, and Rocket, which is a compiler. No empirical evidence was found to support the first three hypotheses, while the final two were accepted.

We have identified a number of worthwhile refinements to the Counsell and Newson study:

- Counsell and Newson recognise the widespread use of the friendship mechanism to facilitate operator overloading. However, they neglected to consider it in their measurement. All our measures have excluded operator overloading in counting friend relationships.

- Counsell and Newson argued in hypothesis H1, that the more friends in a class the less non-friend coupling in that class, based on the rationale that the use of friends in a class satisfies the coupling needs of the class. We believe that the use of friends in a class may satisfy the coupling needs of the classes declared as friends not of those which contain friends.

  Since the class declared as a friend, gains direct access to more functionality, this facilitates the coupling of the declared friend class. Due to the asymmetry of friendship, the friend declaration should have no effect on the coupling of the class containing the friend declaration.

- For hypothesis H2, since any class has direct access to another class's public interface, friendship is not necessary to access these members. Friendship grants access to the private and protected members of a class which otherwise would not be accessible directly. Thus it is more likely that as the number of friends in a class increases so will the number of protected and private members. Also, given that friendship grants access equally to methods *and* attributes we have included all members in this evaluation.

- The Depth in the Inheritance Tree, (DIT), of a node is defined as "the maximal path from the node to the root of the tree", [6]. This metric seems a more accurate measure of "less inheritance" in hypothesis H3, than just categorising classes in terms of whether they are inheritance-based classes or not.

- Based on the rationale given for hypothesis H4, the NOD metric does not seem an appropriate measure of the amount of extra functionality obtained through inheritance. Two classes which have the same set of ancestors, while having inherited the same amount of functionality, (assuming the same type of inheritance, i.e. public, private, etc..), thus have the same DIT, but may have totally different NOD values, depending on how many descendants each class has in the inheritance hierarchy. Therefore, DIT seems a more appropriate metric to measure the amount of functionality inherited. In addition, the examination of the *mean* NOD is not a suitable statistic to use in comparing two groups of unknown distribution.

- The measurement of hypothesis H5 only considered whether or not a class contained a friend, whereas the hypothesis states that stand alone classes would contain more friends. Therefore the evaluation of this hypothesis should compare the distribution of the number of friends in stand-alone classes with the distribution of the number of friends in inheritance based classes.

4 Empirical Study

The hypotheses which are outlined below are based on the hypotheses presented by Counsell and Newson, [9]. The differences between our new hypotheses and those previously presented, and the measures which we use in order to test the hypotheses are discussed here.

For all the hypotheses, the friend count will be based on the friends of a class, excluding the methods which are overloaded operators. Incorporating such usage of friendship in metric collection is not desirable and therefore is not included in this context.

It has been suggested that friendship can be used as an alternative to inheritance, [9] and that it is used as a form of coupling, [9, 4]. Therefore, this empirical study attempts to distinguish between the use of friends within an inheritance scenario and also the use of friendship by stand alone classes. It also attempts to discover a relationship between friendship and coupling.

The data required for this analysis has been extracted from four software systems with the aid of the source code analysis tool, Understand for C++, [17]. The definition of the measures used in evaluating the hypotheses are presented below.

4.1 Hypotheses Definition and Discussion

It is important in this discussion to distinguish between a class which has, or contains friends and a class which is declared as, or made a friend of another class. Consider the following code segment:

```
class A {
  friend class B;
  ...
};
```

In this code segment class A contains class B as a friend and class B is declared as a friend of class A. Class B is granted access to the non-public members of A. Therefore class B has direct access to more functionality as a result of the friendship relationship. In this context the following are the hypotheses under investigation:

- **H1**: Classes declared as friends,(B), of other classes, (A), have less non-friend coupling than classes which are not declared as friends.

  The rationale underlying this hypothesis is as follows. Since friendship is considered a form of coupling, [9, 4], and since a class declared as a friend gains direct access to extra functionality, it might be expected that such a class needs other forms of coupling to a lesser extent and thus will have less non-friend coupling.

  This hypothesis is different from hypothesis H1 presented by Counsell and Newson. Our focus is to compare the coupling of classes which are declared as friends and the coupling of classes which are not declared as friends.

  In order to evaluate H1 the classes in a system are split into two groups, those which are declared as friends and those which are not. For each class, the CBO for the class is calculated. The CBO measure which is used here does not include coupling due to inheritance, nor does it include coupling due to friend classes or functions. This measure is also a one-way measure, i.e. if a class c uses methods or attributes of a class d, then this coupling counts towards the coupling of class c but not class d. This CBO measure is defined by Understand, [17], as: "The number of classes this class is coupled to. Coupling means using a type, data, or member from that class. Any number of couplings to a given class counts as 1 towards the metric total."

  Counsell and Newson, [9], base their measure of coupling on the NAS.

- **H2**: The more friends of a class, (A), the more private and protected members in that class, (A).

  This hypotheses differs from H2 of Counsell in that a count is taken of only the private and protected members and not all members of a class, since any class, including a friend class B, already has access to the public members of class A. The friendship is only needed to access either protected or private members of A. If the class, A, is also a subclass in an inheritance hierarchy, then class B will also have direct access to all protected members inherited by A from its ancestors. These members are also considered in this hypothesis.

  Hypothesis H2 is evaluated by counting the number of friends and the number of private and protected members in each class, (A). Included in this measure are all inherited protected members of the ancestors of a class, assuming the type of inheritance is either public or protected, since these members can also be considered protected in the current class. Private members of an ancestor class are not accessible to a subclass and therefore are not accessible to a friend class of the subclass.

- **H3**: Classes declared as friends,(B), of other classes, (A), have less inheritance than classes which are not declared as friends.

  This hypothesis is the same as hypothesis H3 of Counsell but we measure the amount of inheritance in a different way.

  The evaluation of hypothesis H3 requires a measure of the amount of inheritance a class has. This is determined by the DIT metric. The DIT is defined as "the maximum length from the node to the root of the tree", [6]. A class lower in the inheritance hierarchy is likely to have inherited more members and will also have used the inheritance mechanism more than a class close to the base of the hierarchy. If this hypothesis is accepted, then it may indicate that friendship is used as an alternative to multiple inheritance for classes declared as friends and which are contained in inheritance hierarchies, but are not base classes.

- **H4**: Classes, (A), which contain friends and which engage in inheritance appear lower in the inheritance hierarchy than other classes engaged in inheritance.

  The idea in this hypothesis is that the lower a class is in the inheritance tree, the more functionality this class will have inherited and thus the more favourable it might be to declare a friend of the class, since the friend, in turn, will have access to more functionality.

  Counsell and Newson formed a different hypothesis based on this rationale, stating that classes containing friends that engage in inheritance have less descendants than other classes engaged in inheritance.

  We believe that the Depth of Inheritance Tree, (DIT), is a more accurate measure of both the amount of inheritance in a class and the extra functionality obtained through inheritance.

  To evaluate H4, the DIT is determined for classes which are in hierarchies and which have friends and those in hierarchies which have no friends. The measurement of this hypothesis differs from the measurement of hypothesis H4 of Counsell and Newson, since DIT is taken as the measure of the amount of extra functionality obtained through this inheritance, as opposed to NOD, which was used in [9].

- **H5**: Classes, (A), that do not engage in any inheritance have more friends than classes which do engage in inheritance.

  This hypothesis is the same as that introduced by Counsell and Newson, except that the count of friends excludes overloaded operators, as with all other hypotheses defined here. The rationale behind the hypothesis is based on the fact that "classes not engaging in inheritance tend to use friends as an alternative to inheritance". If this hypothesis is accepted, then this would indicate that friendship is used as an alternative to inheritance.

  Hypothesis H5 is evaluated by counting the number of friends of stand-alone classes and of classes in inheritance hierarchies.

4.2 Overview of the Software Systems

Four software systems were analysed as part of this empirical study. These systems came from various application domains. Two of the systems under study, LEDA and Libg++, were also part of the analysis undertaken by Counsell and Newson, [9]. However, based on the summary statistics available, different versions of these systems were examined here. The four systems were:

1. LEDA,(V4.1), a library of efficient data types and algorithms, particularly useful in the construction, analysis and layout of graphs. This system consisted of approximately 219KLOC and 521 classes.

2. Libg++(V2.6.2), the GNU C++ class library, which provides all of the standard C++ functions and classes. It consists of 51KLOC and 151 classes.

3. Darwin2k(V0.90), is a system which provides simulation and synthesis capabilities for robotics. It consists of 107KLOC and 319 classes.

4. Bayonne(V1.2.13), is the telephony server for the GNU project. It consists of 200KLOC and 174 classes.

Table 1 contains summary metrics for each of the four systems. Each column of the table represents a system as numbered above. LEDA has the highest proportion of stand-alone classes, (38%) and also the highest number of inheritance trees. The average number of classes per inheritance tree is approximately 6 and with a maximum DIT of 5. This might indicate that the trees are quite narrow, (just one or two classes inheriting from a parent class). The width of the inheritance tree influences the calculation of the NOD and thus could impact the outcome of hypothesis H4, of Counsell and Newson. Libg++ has approximately 33% stand-alone classes, but only 7 inheritance hierarchies. Therefore the average number of classes per hierarchy is 14 and with the same DIT as LEDA, these hierarchies must be wider than those of LEDA, (having a lot of subclasses inheriting from another class). Darwin2k and Bayonne have 23% and 7% proportion of stand-alone classes respectively, leaving approximately 41 and 32 classes respectively per

	1	2	3	4
Classes	521	150	319	174
Stand-alone Classes	198	51	74	12
Inheritance Trees	53	7	6	5
Maximum DIT	5	5	6	3
Friends	863	301	129	138
Friend Classes	215	35	71	107
Friend Functions	648	261	58	31
Friend Op Overloads	296	146	27	0
Classes Declared Friends	137	30	34	66
Classes Containing Friends	152	25	31	54

Table 1. Summary Metrics for the Four Systems

hierarchy and considering the maximum DIT in each case, the hierarchies in each of these systems must be relatively wide in comparison with LEDA or Libg++.

In relation to the friend metrics: number of friends, number of friend classes, friend functions and number of operator overloads which are friends, each specific and appropriate friend relationship contributes 1 towards these metrics. Almost 50% of the friend relationships in Libg++ are a result of operator overloading, with 34% of LEDA's friend relationships also resulting from it. Operator overloading also contributes 20% to Darwin2k's friend count but Bayonne doesn't have any overloaded operators. The overall high usage of friendship to facilitate operator overloading in 3 of the systems illustrates the influence that this specific use of the friend mechanism could have on the evaluation of the hypotheses if it were included in the statistical analysis.

Also listed in table 1 are the number of classes which are declared as friends of other classes and the number of classes containing at least one friend class, in each of the 4 systems. The proportion of classes declared as friends varies from 11% for Darwin2k to 37% for Bayonne. The proportion of classes containing friends follows a similar distribution, with 10% of classes in Darwin2k, up to 31% of classes in Bayonne containing at least one friend class.

5 Statistical Analysis and Interpretation

Section 5.1 provides an overview of the statistical tests used in this analysis. Sections 5.2 to 5.6 look at each of the hypotheses separately, and provide statistical results and interpretation of results in each case.

5.1 Statistical Techniques

Two test statistics are used in this empirical study. The Pearson Chi-Square Test of Association was used to determine whether or not there was an association between two groups. The benefits of the Chi-Square test are:

- It makes no assumption about the shape of a distribution. Therefore, it allows us to make inferences about properties of software systems in general.
- If the Chi-Square statistic and p-value produced as part of the test indicate, that a difference exists between two populations, which is not down to chance, then this test facilitates the identification of where distributions differ.

The Chi-Square statistic is based on frequency-categorical data. This means that the data must be categorised and such categories must be mutually exclusive and exhaustive. The frequency of occurrence of data in different categories is noted, not the actual data itself. For example, for hypothesis H1, we categorise the CBO distribution of values into four ranges. These groups are determined by the 25, 50 and 75 percentiles, (the 50 percentile is the median), of the CBO distribution and are illustrated by the columns of table 2. This table illustrates the cross-tabulation results for the LEDA system for hypothesis H1. The rows in this table represent whether a class is declared a friend or not.

For Hypotheses H3, H4 and H5 we construct similar categorisations. The resulting cross-tabulation produces frequency data and also proportional data. Table 3 contains the p-value returned by the Chi-Square test for these hypotheses.

The Spearman correlation coefficient is a measure of the linear association between two variables. This is a non-parametric method, i.e. it makes no assumption about the distribution of the data. The correlation coefficient ranges from -1 to +1. A value of zero for the correlation coefficient indicates that no linear relationship exists. A value of +1 indicates a perfect positive linear relationship, i.e. as one variable increases the other increases and a correlation coefficient of -1 indicates a perfect negative relationship, i.e. as one variable increases the other decreases. Hypothesis H2 was tested using the Spearman test and the results of this test are shown in table 4.

5.2 H1: Classes Declared as Friends have less Coupling

Table 3 displays the p-values for this hypothesis. For three of the systems the p-value is < 0.001 and therefore we can conclude that the two populations under study are different for these three systems. Table 2 displays the cross tabulation results for the LEDA system, where DFR, i.e. 'class declared as friend' is a boolean. From this table we can determine that 32.6% of classes which are not declared as friends appear in the lower quartile, (CBO range = 0-1),

CBO Ranges					
DFR		0-1	2	3-4	$\geq$5
0	Ct	125	79	98	82
	%	32.6%	20.6%	25.5%	25.5%
1	Ct.	23	23	29	62
	%	16.8%	16.8%	21.2%	45.3%

Table 2. H1: Chi-Square Output for LEDA

	H1	H3	H4	H5
LEDA	.000	.002	.000	.316
libg++	.000	.033	.034	.000
Darwin2k	.038	.000	.001	.000
Bayonne	.000	.001	.477	.028

Table 3. P-values for the Chi-Square Statistic for Four Hypotheses

while only 16.8% of classes which are declared as friends appear in this quartile. For classes declared as friends, the proportion of classes in each quartile increases as you move to the upper quartile. The opposite occurs for classes not declared as friends. Therefore we can conclude that classes in LEDA which are declared as friends have considerably higher coupling than classes which are not declared as friends. A similar trend also occurs for Libg++ and Bayonne. In fact, it is even more pronounced for these systems. For example for Libg++ 50% of classes declared as friends have a coupling $\geq$3, whereas only 5% of classes which are not declared as friends have a CBO value in this range.

The Darwin2k system returned a p-value of .038, indicating a 3.8% chance that this result is random. However, a pronounced difference between the distributions occurs in the lower and upper quartiles, with little difference in between. The trend seems the same as in the other three systems but not as pronounced. Given that the Chi-Square test does not report a significantly small p-value needs further investigation, but may be because the proportion of the classes declared as friends in the system is very small, (see table 1).

Our initial hypothesis was based on the premise that friendship might be used as an alternative to other forms of coupling and thus in classes which have a lower coupling value. However the opposite seems to be the case: classes which are declared friends have a high degree of coupling. This seems to indicate that classes which are declared friends are coupling hot-spots in software systems.

Interestingly the literature provides suggested uses of friends which concur with this analysis. Cline, [7], suggests that a class which might need to be split into two separate classes, without increasing the number of members, could use the friend mechanism to facilitate communication between these classes.

Lippman, [18], also discusses the use of friendship where there is tight coupling between classes. Suppose a method should be made a member of two different classes. One suggestion would make the method a member of one class, with the other class being declared a friend of the class which contains the method. The view that friend classes occur where a class is tightly coupled to other classes is also expressed in [12].

5.3 H2: The More Friends, the More Private and Protected Members

Table 4 shows the statistical results for hypothesis H2. We tried to determine if a linear correlation existed between the number of friends in a class and the number of private and protected members in the class. For three of the four systems a linear correlation was reported by the Spearman statistic. In each case the p-value was less than .01. Such a correlation was not uncovered by Counsell and Newson in an analysis that counted all methods in a class but did not include inherited methods. We have only included those methods for which friendship is necessary to provide direct access. Therefore we included non-public methods in the class itself and all inherited protected methods, if the inheritance type was either public or protected.

The Spearman values are not close to 1. This indicates that in each case the linear correlation isn't very strong. This may be because another variable is influencing the correlation. For example, a much stronger linear correlation might be identified if stand-alone classes and inheritance based classes were studied separately. This needs further investigation. For the libg++ system a linear correlation doesn't exist according to the Spearman statistic.

Since friendship grants access to the private and protected members and the inherited protected members, it seems appropriate that as the number of friends, which need access to these members increases, the number of these particular members should also increase. Given that this hypothesis holds for three of the four systems it may be worth evaluating this hypothesis on systems from the same application domain to establish if a relationship exists within each domain.

5.4 H3: Classes declared as Friends have less Inheritance

Table 3 contains the statistical results for hypothesis H3. The results indicate that the distributions of the DIT metric for classes which are declared as friends and for classes not declared as friends are significantly different for two of the four systems. This may also be the case for LEDA or at

		N	p	Spearman
H2	LEDA	521	.000	.191
	libg++	150	.197	.106
	Darwin2k	319	.004	.161
	Bayonne	174	.000	.468
H2.1	LEDA	521	.000	.297
	libg++	150	.005	-.283
	Darwin2k	319	.252	-.073
	Bayonne	174	.084	-.136

Table 4. Pearson Statistics for H2 and H2.1

least for parts of the distributions for the LEDA system.

We need to examine each of the four systems in more detail to establish if any relationship between the DIT and the two groups, (classes which are, and are not declared as friends). For the LEDA system the 25, 50 and 75 percentiles are the same for each group. There is little or no variation between the proportion of each group in the upper and lower quartiles. We conclude that any difference in the distributions for the LEDA system could be down to chance. For Darwin2k, it is clear from an examination of the cross-tabulation that the DIT value is smaller across the distribution for classes declared as friends, thus supporting the hypothesis. However, the Bayonne system demonstrates the opposite effect, i.e. the DIT is smaller for classes not declared as friends.

Therefore, we must reject H3 on the basis of the analysis above. We cannot determine any relationship between DIT and classes which are declared as friends. This may be because we are examining both inheritance based classes and stand-alone classes together and, given that all stand-alone classes have a DIT of zero this may be skewing the results. A refinement of this hypothesis might examine stand-alone and inheritance-based classes separately in relation to this hypothesis. It is also possible, in line with Counsell and Newson's findings, that the declaration of a class as a friend is independent of its placement in the inheritance hierarchy.

5.5 H4: Classes which contain friends and engage in Inheritance have a higher DIT

For the purposes of the discussion of this hypothesis we refer to classes which engage in inheritance and contain friends as 'group 1' and classes which engage in inheritance and do not contain friends as 'group 2'.

The p-values presented in table 3 indicate that the distributions differ for group 1 and group 2 for the LEDA and Darwin2k systems. However, for LEDA, 31.5% of group 2 classes have a DIT ≥ 2 whereas only 4.2% of group 1 classes fall into this category.We do not support the hypothesis here since it seems that classes engaged in inheritance and which do not contain friends have a higher DIT. The same conclusion can be drawn for the Darwin2k system.

The p-value for Bayonne indicates that it is highly unlikely that the distributions are different and this is borne out in the cross-tabulation. The cross-tabulation for the Libg++ system, portrays some variation but this seems to indicate that a higher proportion of group 2 classes have a DIT ≥ 2. This again seems to indicate the opposite to the statement of hypothesis H4 above.

We expected that classes which contain friends would appear lower in the inheritance hierarchy, thus having a higher DIT, since these classes inherit functionality from all ancestor classes. However, from this analysis we cannot accept this hypothesis, in fact as mentioned above, for some classes the opposite seems to be the case. Therefore, the fact that a class can obtain extra functionality through inheritance does not seem to encourage other classes to make use of this inherited functionality.

To investigate the relationship between the usage of friendship and inheritance further we have looked at a variation of hypothesis H2, since we believe that it could cast further light on the results reported here. Therefore we have carried out a variation on H2, (henceforth referred to as H2.1), where instead of measuring the number of private and protected members in total, we have only measured the inherited protected members. If a linear correlation does not exist between the number of friends in a class and the number of inherited protected members of the class, then this is further evidence that classes are not declared friends of other classes for the purpose of accessing this inherited functionality. This analysis concurs with the results obtained above: classes containing friends do not necessarily appear lower in the inheritance hierarchy.

As table 4 indicates, for H2.1, the p-values suggest that a correlation exists for only two systems, but the Spearman value for these systems is positive for one and negative for the other indicating that as the number of friends in a class increases the number of inherited protected members increases for one system and decreases for the other system. Therefore, we can conclude that there is no evidence to support a correlation between inherited protected members in a class and the number of friends that class has. This gives further support to the statement that friends are not declared in classes to take advantage of the extra functionality available through inheritance. Thus classes typically contain friends only to access the functionality within the class itself.

5.6 H5: Stand-alone classes have more friends than Inheritance-based classes

Table 1 shows the statistical results for hypothesis H5. There exists a difference between the distributions for stand-alone classes and inheritance based classes for Libg++, Darwin2k. An examination of the distributions provides support for the hypothesis above. For example, in Libg++ 27.5% of stand-alone classes have more than 2 friends, whereas no inheritance based class has more than 2 friends. There is no evidence to support this claim in LEDA. In fact the distributions seem very similar. For the Bayonne system, the p-value indicates that the distributions may well be the same, or at least that any differences in them may be down to chance. However an examination of the distributions leads us to conclude that at least parts of the distributions are different, lending some support to the above hypothesis. For example, for inheritance based classes, the proportion which do not contain any friends is 68.5%, compared with 33.3% for stand-alone classes. Also, the proportion of stand-alone classes which contain one friend is 50% compared with 16% for inheritance based classes. It is also the case that 84.5% of all classes in the system fall into one of these categories. The proportion of classes which contain two friends, or more than two friends remains fairly constant across the stand-alone and inheritance based classes. The lack of any difference for these two groups probably resulted in the large p-value reported. Most of the classes in the system have either zero or one friend. A lower p-value would probably be achieved by only splitting the data into 2 groups, classes which have a friend count of zero and classes which have a friend count greater than zero. It therefore seems reasonable to accept hypothesis H5 for the Bayonne system.

Given that there is a significant difference in the number of friends in stand-alone classes and inheritance based classes for 3 of the systems, the hypothesised relationship may well exist. It may also indicate that friends are used as an alternative to inheritance, or to multiple inheritance, if the classes declared as friends of the singletons already have a parent class. Further analysis is needed to investigate these issues. We cannot support this hypothesis conclusively since the hypothesis doesn't hold for LEDA.

6 Related Work

In another paper, Counsell *et al.* [10], have studied the relationship between high level features, such as association, aggregation and generalisation and low level features such as the number of methods and attributes in a class. In this study the friend facility was ignored as a form of association because the analysis tool in use did not provide this information. However, once again it was recognised that the use of friends is widespread and it has been suggested that friends are used as an alternative to inheritance and aggregation, as a form of coupling.

Many other studies exist in the literature which empirically investigate coupling issues in object-oriented software systems. Briand *et al.*, [4], attempted to find a correlation between different coupling measures and also between coupling and the fault-proneness of classes. Their initial results indicated that the use of friend classes in C++ increases the fault-proneness of classes. However, a subsequent refined study, [5], which distinguished between coupling to library and non-library classes, could not find a relationship between coupling based on friendship and fault-proneness. Hitz and Montazeri, [16], have described some refinements to the CBO metric based on how access to methods or attributes of a class is achieved.

There has also been extensive research in devising measures to quantify the amount of coupling in systems. Chidamber and Kemerer, [6], have defined six object-oriented software metrics, including the one discussed previously based on coupling. These metrics have formed the basis for the definition of many more metrics which are variations and extensions of these and which are summarised in [3]. Lorenz and Kidd, [19], have also defined a set of easy to apply project and design metrics which include a metric based on the use of friend functions.

7 Conclusion

In this paper an empirical study of four software systems was carried out to investigate the use of the friend mechanism in C++ software. Since it has been suggested that friendship breaks encapsulation and is considered poor programming practice, [8, 9], examination of actual software systems is necessary to establish how friendship is actually used and to try to associate its usage with other program constructs. Ultimately, the aim of this work is to establish if friendship impacts external quality characteristics, such as understandability, maintainability, re-usability, and reliability.

Since the internal quality of a system influences its external quality, [1], the first step in the study of the use of the friend mechanism is to study and measure its usage against other internal program mechanisms and metrics. This is what we have done in this paper. In fact we have established that classes which are declared as friends have considerably higher coupling that classes not declared as friends. This relationship has heretofore not been established. This finding indicates that classes declared as friends could highlight coupling hot-spots in systems.

We have also established a possible link between the number of friends in a class and the number of private and protected members in the class, illustrating a very practical usage of friendship, to access restricted methods and data, which otherwise would not be accessible directly. We have demonstrated these findings through a refined measurement process which ignores friend functions which are overloaded operators. We found no evidence to support the hypothesis that friends tend to be found in classes deep in the inheritance hierarchy. In fact we have shown that friendship is not used to access inherited protected members. We found some evidence to support the claim that stand-alone classes use friends more than classes involved in inheritance.

From this analysis it seems clear that a much more sophisticated set of metrics, than already exists is required for the analysis of the quality of software systems. These metrics need to be based on the rationality of expert programmers. Ejiogu, [13], has stated that effective software metrics should be "simple and computable" and "programming language independent". While such characteristics should be attributable to metrics from a practical perspective they are not applicable to metrics which are required to evaluate the quality of software design. These metrics must be program language dependent and be based on the rationale behind the use of constructs by appropriately skilled users.

References

[1] N. Bevan. Quality in use: Meeting user needs for quality. *J. Syst. Softw.*, 49(1):89–96, 1999.

[2] G. Booch. *Object Oriented Design with Applications*. The Benjamin/Cummings Publishing Company Inc., 1991.

[3] 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.

[4] L. C. Briand, P. T. Devanbu, and W. L. Melo. An Investigation into Coupling Measures for C++. In *International Conference on Software Engineering*, pages 412–421, 1997.

[5] L. C. Briand, J. Wüst, J. W. Daly, and D. V. Porter. Exploring the relationships between design measures and software quality in object-oriented systems. *The Journal of Systems and Software*, 51(3):245–273, 2000.

[6] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Trans. Software Eng.*, 20(6):476–493, 1994.

[7] M. Cline. C++ FAQ LITE - Frequently Asked Questions. web: www.parashift.com.

[8] J. O. Coplien. *"Advanced C++ Programming Styles and Idioms*. Addison-Wesley Longman Publishing Co., Inc., 1992.

[9] S. Counsell and P. Newson. Use of Friends in C++ Software: An Empirical Investigation. *Journal of Systems and Software*, 53(1):15–21, 2000.

[10] 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.

[11] H. Deitel and P. Deitel. *C++ How To Program*. Pearson Education, Inc., 2003.

[12] K. Dukes. http://www.devarticles.com/c/a/cplusplus/c-plus-tricks-of-the-trade%3a-friend-functions/3/. www, 2002.

[13] L. Ejiogu. *Software Engineering with Formal Metrics*. QED Publishing, 1991.

[14] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., 1990.

[15] R. Harrison, S.Counsell, and R. Nithi. An Overview of Object-Oriented Design Metrics. In *International Conference on Software Technology and Engineering Practice,(STEP)*, pages 230–234. IEEE Computer Society Press, July 1997.

[16] M. Hitz and B. Montazeri. Chidamber and Kemerer's Metrics Suite: A Measurement Theory Perspective. *IEEE Trans. Softw. Eng.*, 22(4):267–271, 1996.

[17] S. T. Inc. Understand for C++. web: www.scitools.com.

[18] S. B. Lippman. *C++ Primer*. Addison Wesley, 2 edition, 1991.

[19] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics*. Prentice-Hall, Englewood Cliffs, N.J., 1994.

[20] S. Meyers. *Effective C++*. Addision-Wesley, 1998.

[21] M. Page-Jones. *What Every Programmer Should Know About Object-Oriented Design*. Dorset House Publishing, 1995.

[22] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. *Object-Oriented Modelling and Design*. PHI, 1991.

[23] B. Stroustrup. *The Design and Evolution of C++*. ACM Press/Addison-Wesley Publishing Co., 1994.