UWL REPOSITORY

repository.uwl.ac.uk

A comparison and evaluation of variants in the coupling between objects metric

Child, Mike, Rosner, Pete and Counsell, Steve (2019) A comparison and evaluation of variants in the coupling between objects metric. Journal of Systems and Software, 151. pp. 120-132. ISSN 0164-1212

http://dx.doi.org/10.1016/j.jss.2019.02.020

This is the Accepted Version of the final output.

UWL repository link: https://repository.uwl.ac.uk/id/eprint/5824/

Alternative formats: If you require this document in an alternative format, please contact: open.research@uwl.ac.uk

Copyright: Creative Commons: Attribution-Noncommercial-No Derivative Works 4.0

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy: If you believe that this document breaches copyright, please contact us at open.research@uwl.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Rights Retention Statement:

A Comparison and Evaluation of Variants in the Coupling Between Objects Metric

Mike Child,
Department of Informatics
South Bank University, London
mike.child@lsbu.ac.uk

Pete Rosner,
School of Computing and Engineering,
University of West London, London
peter.rosner@uwl.ac.uk

Steve Counsell,
Department of Computer Science
Brunel University, London
steve.counsell@brunel.ac.uk

Abstract

The *Coupling Between Objects* metric (CBO) is a widely-used metric but, in practice, ambiguities in its correct implementation have led to different values being computed by different metric tools and studies. CBO has often been shown to correlate with defect occurrence in software systems, but the use of different calculations is commonly overlooked. This paper investigates the varying interpretations of CBO used by those metrics tools and researchers and defines a set of metrics representing the different computational approaches used. These metrics are calculated for a large-scale Java system and logistic regression used to correlate them with defect data obtained by analysing the system's version tracking records. The different variations of CBO are shown to have significantly different correlations to defects. Regarding results, a clear binary divide was found between CBO values which, on the one hand, predicted a defect and, on the other, those that did not. The results, therefore, show that a clarification or unambiguous re-definition of CBO is both desirable and essential for a general consensus on its use. Moreover, applications of the metric must pay close attention to the actual method of calculation being used and, conclusions and comparisons made as a result.

1. Introduction

The concept of coupling in software has been around since the mid-70's [Stevens1974], and attempts to measure it have been part of most metric suites since. The Coupling Between Objects metric (CBO) is a member of the long-established and widely used Chidamber and Kemerer (C&K) object-oriented metric suite [Chidamber1991; Chidamber1994; Chidamber1998]. It measures the extent of coupling between two classes. Numerous studies have collected and used the metric and many have included CBO in multivariate models designed to identify defect-prone code [Sinh2009; Singh2010; Chhillar2011; English2009; Shatnawi2008; Zhou2006; Gyimothy2005; Ping2002; Briand2001; Briand2000; Chidamber1998; Basili1996]. Many of these studies have highlighted the importance of CBO as means of capturing the predictive ability of the same metric. Basili et al., [Basili1996] empirically validated the C&K suite of metrics in terms of their ability to identify defect-prone classes; seven C++ systems were examined and the CBO, as well as four of the remaining five C&K metrics, were found to correlate significantly with defects. Chidamber et al., [Chidamber1998] found associations between a high CBO metric value and lower productivity, more rework, and greater

design effort. The study by Gyimothy et al., [Gyimothy2005] collected the number of defects found and corrected in each class of the Bugzilla database and attempted to find associations with the C&K metrics. Many of the C&K metrics were found to be positively correlated with defects, but the CBO metric was the best at predicting the defect-proneness of classes. Wilkie et al., [Wilkie2000] identified CBO to be an accurate predictor of changes to classes through a study of commercial C++ application over a period of two and a half years.

Despite these and many other studies, CBO's definition still requires interpretation in to implement it practically, and this has resulted in different metrics tools producing different values in its calculation. We assess some of these tools in Section 3.1. While in themselves, most of these studies have contributed to a strong body of knowledge about coupling and systems in general, the implication of using different interpretations of the metric is that the results from those studies cannot simply be combined into a single body of evidence for a specific purpose. The study by Briand et al., [Briand1999] provided a detailed breakdown of the different ways in which coupling could be measured and interpreted (CBO featuring highly). The paper highlighted the need for consistency in empirical studies where CBO was used. Multiple variations in coupling metrics exist. From an academic stance, therefore, if inclusion or exclusion of certain program traits leads to different end values of the CBO, then we can no longer generalise our results concerning the link between CBO and software quality with total confidence. Furthermore, from a software developer perspective, metric tools to inspect the systems under development may each produce different results; the implication of this is that it is not feasible to make totally informed decisions on refactoring and re-engineering, for example. This paper seeks to investigate the effects of this ambiguity in practice and, particularly, the impact it might have on CBO's capacity for identifying defect-prone code. We use a set of collected metrics representing the different computational definitions of CBO in the literature as a basis of our study. These metrics were calculated for a single, large-scale Java system using a bespoke tool [Rosner2014] and logistic regression then used to correlate them with defect data obtained from the system's version tracking records. Regarding results, the different variations of CBO were shown to have significantly different correlations to defects. In particular, the metrics can be divided into two groups: those that had a noticeably higher odds ratio and goodness of fit and those that had low values; put another way, there are those that exhibit a predictive property and those that do not.

The study presented seeks to highlight the variations between the versions of CBO; further studies are needed to firstly, validate previous empirical studies using the CBO and secondly, assess the extent of the threat to external validity posed as a result. That the study of one system showed such variations in the CBO values returned is worrying, and, it is this concern that is the real message of the work. We also accept that the definition of CBO counts both forward and backward links as one value, thus making no distinction between links. The view of Kitchenham [Kitchenham2010] is that the CBO metric is flawed as a result because it treats backwards links in the same way as forward links; a backward link may be more harmful than a forward link or *vice versa* regarding its propensity for defects. We do not seek to explore that issue in this paper. We are merely trying to show that the CBO, as we understand it, gives different values depending on the definition used whatever view of those definitions you have. A large number of studies have used CBO in the past twenty-five years (too many to be included in the reference list of this paper). We therefore see our study as worthwhile, even if you do not agree with the definition of CBO as being valid theoretically. Finally, a systematic mapping study of coupling and cohesion metrics was recently provided by Tiwari et al., [Tiwari2018]. The aim of the work was to map current state-of-the-art in OO coupling and cohesion metrics. Results from 137 identified studies revealed an interesting trend in keeping with our work. Most studies of coupling and cohesion were found to be evaluations of existing metrics or proposals of new ones; very few were based on experience of using metrics or proper validation of the metrics proposed. The work highlighted a key issue in the empirical world: a proliferation of definitions which need proper evaluation and assessment. A quote taken from Tiwari et al., regarding the current state in cohesion and coupling metrics highlights this point: "*.....some issues such as the lack of availability of information about the contextual usages of these metrics and their multiple interpretations by different researchers need to be resolved to establish the practical use of these metrics*".

In this paper, we explore variants in the CBO metric, but the same arguments could apply to all metrics. In fact, Landman et al. [Landman2014] explored the Cyclomatic Complexity metric [McCabe1976] and its perceived confounding relationship with lines of code; in a similar way, Arcelli et al., [Arcelli2011] undertook the same analysis for code smell detection tools and found that: "*comparing the tools is very difficult, and in some cases also using them is not very easy and immediate*". Finally, Jongeling et al. [Jongeling2017] carried out a study of sentiment analysis tools and found variations depending on the tool used. These studies demonstrate that we cannot make assumptions about two tools performing the same task providing consistent results; the question then arises "why would or should results be any different for CBO?" Of course, there is no reason why they should. Of the six C&K metrics, it is not unreasonable to suggest that CBO is more intuitively understood from a definition viewpoint than many of the other C&K metrics (e.g., the lack of cohesion in the methods of a class metric, (LCOM)); as such, it is at least therefore worthy of deeper study. The same could be said of many other OO metrics or indeed many other non-OO metrics.

We see the contents of this paper as following two phases and addressing two research questions.

Firstly, what are the different variants of CBO found in currently available tools and the literature - this is addressed in Section 3. The research question posed in this section essentially asks: '*How do the different tools that collect CBO define the metric* (Section 3.1) *and how do prominent studies of the CBO report its measurement* (Section 3.2)?' This inevitably and through necessity leads to a statement of the different forms of coupling (Section 3.3) and, for the next stage of our analysis, a statement of the metric variations (Section 3.4) that we empirically evaluate (in Section 4). The means by which we answer the research question is from a thorough literature survey of the CBO of the tools used and studies reporting CBO results; it is the knowledge and understanding that this survey gives us which provides for the work described in Sections 3.3 and 3.4.

Secondly, what is the impact of the identified CBO-variants on the previously reported relation between CBO and defects - this is addressed in Section 4). The research question posed in this section essentially asks: '*How do the definitions of CBO established in Sections 3.3 and 3.4 perform in terms of their predictive capability when we collect those variations as an independent variable and defects as the dependent variable across an open-source system?*' Section 4 therefore comprises three essential sections: the systems under study (Section 4.1), the methodology adopted (Section 4.2) and the results generated (Section 4.3). This analysis allows us to draw conclusions and provide insights into these differences and the implications for differences in both the tools that collect CBO and the different ways of measuring CBO. The remainder of the paper is arranged as follows. We first detail the research context for the paper (Section 2). In Section 3, we describe aspects of CBO's definition that require interpretation, a survey of the definitions of CBO given in the documentation of Java metric tools and literature and a definition of the CBO variations we use. Section 4 provides study details including the system studied, the methodology adopted and the results of the study based on the Eclipse project and the defect prediction data. We raise a number of discussion points and threats to validity in Section 5, before concluding and pointing to future work (Section 6).

2. Research context

The CBO metric was proposed and defined in two revisions by C&K in 1991 and 1994 [Chidamber1991; Chidamber1994], respectively. Essentially, it is a count of the number of unique classes that any given class is coupled to by at least one member access (e.g., a method call or a field access). Although the C&K definitions state that classes are coupled only if they "affect the history" of one another, this is clarified as meaning the "usage history" and does not require that internal state is modified as a result of the access involved. The 1991 version of CBO, which, henceforward, we will refer to as CK1, excluded coupling within an inheritance hierarchy (i.e., couples to ancestor and descendent classes were excluded) while the 1994 revision (henceforward CK2) removed this rule.

Although the later paper is clearly intended to supersede the earlier, the original rule on inheritance has continued to be quoted in the literature [Zhou2006] and potentially to have been implemented in practical applications of the metric or metric tools [Borland2008]. According to C&K's definition, CBO should be calculated for a given class as the union of incoming and outgoing couples, but a significant number of both studies and metric tool implementations exclude incoming couples. The reasons for this are not clear, but the existence of similar coupling metrics measuring *afferent* and *efferent coupling* which measure incoming and outgoing coupling (in the same way as fan-in and fan-out [Martin1994]) might have been an influence in these decisions.

In the same vein, a valid research line is to determine how the resolution of the destination of a member access affects the calculation of CBO. This relates to the location of the implementation of an accessed member *vis-a-vis* the location of its declaration and the reference type used to access it. The relevant three circumstances which need to be considered are as follows. Firstly, a member accessed on the reference type is not implemented in that type, but is inherited from an ancestor class (this applies to both member fields and methods). Secondly, a method invoked on the reference type is not implemented in that type, but is only an abstract stereotype and implemented in one or more descendent classes of the reference type (no equivalent circumstance for member fields exists). Thirdly, a method invoked on the reference type *is* implemented in that type but is also overridden by implementations in one or more descendent classes of the reference type (no equivalent circumstance for member fields exists). In each of these cases, both CK1 and CK2 definitions are ambiguous about which pairs of classes should actually be considered to be coupled. That is, whether the source of the access should be considered to be coupled to the reference type it uses or the class of the actual implementation of the member it is located in (which in practice might only be determined at runtime, due to late-binding). These ambiguities are investigated by Briand et al., [Briand1999].

A further ambiguity concerns the invocation of constructors. C&K do not mention constructors in either of their definitions. In Java, constructors are not generally considered to be methods. Methods are offered by objects, while constructors are used to *create* objects. The Java Language Specification states that: "Constructors are not members" [Gosling2015]. In spite of this, Java metric tools do appear to count calls to constructors as couples. Part of the reason for this is that many tools appear to count any reference to a class as a couple to that class, even if no access to a member is ever made. The reasons for this are not clear, but may have been influenced by similar metrics based upon of compile-time dependencies [Martin1994]. In this paper, to define the different variants of CBO and make clear their differences, coupling cases are formally described and named. Variant metrics are then defined in terms of the treatment given to each coupling case.

The terminology used to describe the different types of coupling in this paper is as follows. In each case, there is a *source* class that invokes some method on a *target* class type. The target class is the aforementioned reference type. The invoked method is declared in the target class or an ancestor of it, while implementations of it can be present in any of the classes in the same hierarchy as the target. Some of these coupling cases apply to variable (or field) access as well as method invocation, while some of them do not. This is noted in each description and, where field access is applicable, it is to be understood to be included in the counting of the coupling type.

3. CBO variants

3.1 Tool support

We begin by surveying the tools that collect CBO and the variations in those metrics that they provide. The starting point for this survey consisted of a search for the terms 'CBO and Coupling Between Objects' in the IEEE and ACM digital libraries. A basic snowballing technique was then used to recursively follow both the citations and references given for, all the relevant papers identified [Wohlin2014; Brereton2007]. This process carried the survey beyond the initial repositories. Because

we were concerned specifically with identifying the tools and calculation methods for CBO, it was necessary to limit papers in the final set to those that explicitly made use of CBO to generate data.

Different metrics tools have been shown to produce inconsistent results when calculating CBO. The study by Lincke et al., [Lincke2008] presents the results of a number of different tools to calculate metrics. Five of these tools calculate the CBO metric, but there are significant differences between them. Several give similar results, but none appear to be identical. The tools they compared concerning CBO were as follows: `Analyst4j` [CodeSWAT2013], `CKJM` [Spinellis2012], `CCCC` [Littlefair2006], `Understand for Java` [Scientific2013] and `VizzAnalyzer` [Arisa2005]. Whether the differences in the values are due to the interpretation of the metric the developers employed is unclear. All that we can surmise is that tools that produce different CBO values for the same system must be calculating it differently. Ultimately, the only way to know precisely how a given tool calculates any given metric would be to examine its source code, but this is impractical from a time perspective. Moreover, the documentation of metric tools often simply quotes a standard definition of each metric, with scant detail about how any ambiguities might have been handled in their implementations. We could also have carried out black box testing by providing the tool with example programs and observing the results similar to that used by Bowring [2004], but again this would have been a relatively time-consuming activity.

The authors also analyzed the documentation of metrics tools that collect CBO. Six tools were examined: `Analyst4j` [CodeSWAT2013], `Borland Together` [Borland2008], `CKJM` [Spinellis2012], `JHawk` [Virtual2013], `Understand for Java` [Scientific2013] and `Structure Analysis for Java (STAN)` [Odysseus2011]. Four of these tools calculated CBO as import couples only and considered references to classes to constitute couples. This implies two things: four of the tools count only the import set of couples, not the union of import and export couples (as the C&K definition stipulates); and that these four tools also consider any reference to another class as being a couple even if there are no invocations of methods or accesses to fields of that class (e.g., declaring a variable as being of some class C is counted as a couple to C even if it is never used). Of the remaining two, one had no documented details of its implementation available. Only a single tool (Analyst4j) appears to define CBO in the same way as C&K's 1994 work. In surveying published research that uses CBO values, we found that several other tools were sometimes cited, when, according to their documentation, they do not actually calculate CBO; they calculate other coupling metrics. The tools concerned are C and C++ Code Counter (`CCCC`) [Littlefair2006, `Eclipse Metric Plug-in` [Sauer2011] and `VizzAnalyzer` [Arisa2005]. The study by [Cruz2009] gives no definition of CBO, while [Tang1999] does not indicate the tool used. Clearly, there is inconsistency in the way that tools have been implemented to calculate CBO as well as in the documentation of how the CBO was calculated.

3.2 Empirical and theoretical study use

We next explore the different variations of CBO used by key theoretical and empirical work. Regarding counting CBO as the import set or the union of the import and export set of couples, several authors specify that they count import coupling [Basili1996; Wilkie2000; Yu2002; Gyimothy2005; Zhou2006] while others do not specify this but use a tool which counts import coupling [Shatnawi2008; Raed2008; Cruz2009]. Some studies state that they used union counting, but the tool they used counted import coupling [English2009; Kalpani2012]. Others claim that they count coupling by union [Tang1999; Briand1999; ElEmam2001; English2009; Singh2010; Johari2012], but use a tool which does not [English2009; Johari2012]. With regard to the inclusion or exclusion of the inheritance hierarchy, most authors either state that they include it or can be assumed to on the basis that they cite the 1994 C&K paper and say no different; a few either state that they exclude it or use a tool which states that it excludes it [Ping2002; Yuming2006; Shatnawi2008]. With regard to counting references to classes as couples, the picture is also rather unclear. Some studies state that they do count references as couples [Wilkie2000; El Emam2001; Johari2012] while others state that they do not, but use a tool which says it does [Shatnawi2008; Cruz2009; English2009].

Published research has clearly used a variety of versions of CBO. In many cases, the studies do not appear to have been aware that the tools they were using were not measuring what they assumed they were. Coupled with the inconsistency of tool definitions, this lack of consistency makes the comparison of research results difficult and represents a general threat to internal, construct and external validity of conclusions across empirical studies.

3.3 Coupling types

To inform our analysis of the CBO and its variations, it is convenient to describe the different types of coupling and illustrate each, where appropriate, with a class diagram, as follows.

3.3.1 Local Inheritance coupling

Local Inheritance coupling (see Figure 1) differs from direct coupling in only one respect: the target class is an ancestor of the source class. This type of coupling *is* described by C&K and forms the basis of the difference between their two definitions of CBO.

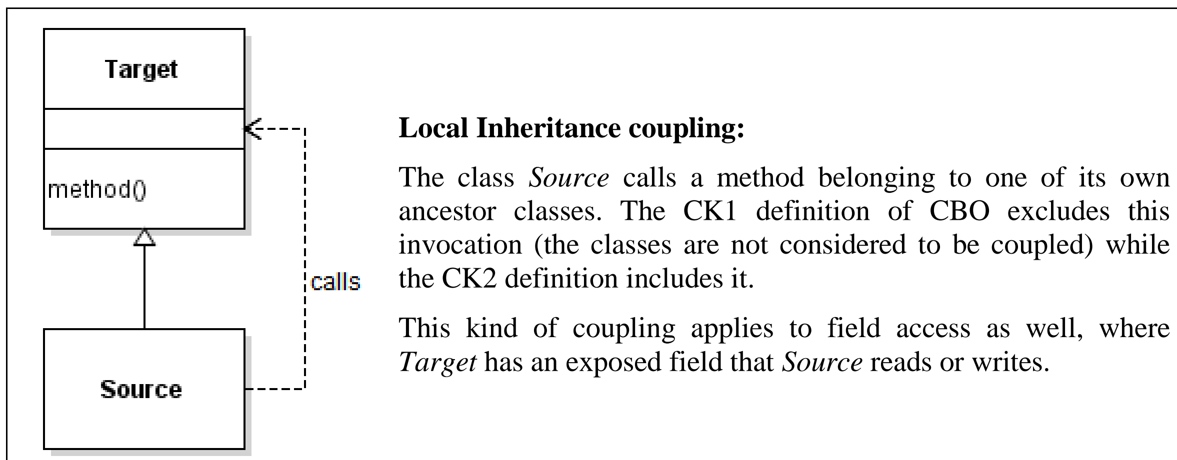

Figure 1. Local Inheritance coupling

Most authors make no mention of this CK1/CK2 issue with regard to the CBO statistics they present, but [Yuming2006] and [Kumari2011] state that they do not use inheritance when computing CBO. Some authors, such as [Kanman2004] and [Briand1999] explicitly use two versions of CBO, once including inheritance and once not, because of the ambiguity between the two papers.

3.3.2 Foreign Inheritance coupling

Foreign Inheritance coupling (see Figure 2) refers to the situation in which the target class does not implement the method being invoked, but inherits it from an ancestor class. In contrast to local inheritance coupling, there is no relationship between the source class and the target class. The question now arises as to whether a couple should be considered to exist between *Source* and *Target* or between *Source* and *Ancestor*.

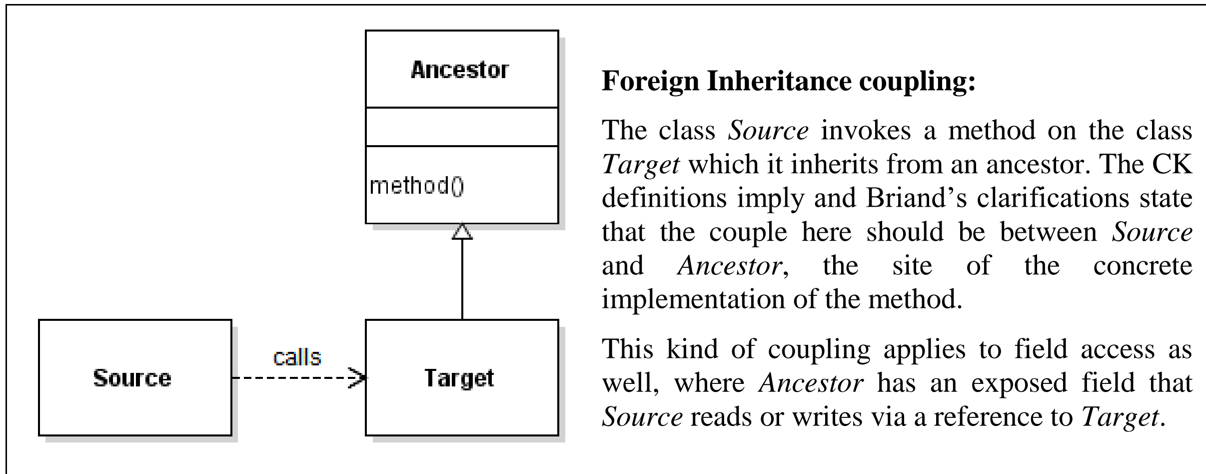

Figure 2. Foreign Inheritance coupling

As with local inheritance, this kind of coupling applies to field access as well, where *Ancestor* has an exposed field that *Source* reads or writes via a reference to *Target*.

3.3.3 Abstract or Interface coupling

Abstract coupling (see Figure 3) refers to the situation in which the target class only has (or inherits from) an abstract declaration of the method being invoked. An abstract method has no executable body and when called at runtime the object the target refers to must be an instance of some subclass of the target type which has a concrete implementation of the method. Any class can declare abstract methods. A class which *only* declares abstract methods is sometimes described as an interface and in some languages (such as Java) "interface" is a separate named language construct distinct from class. Thus, *Interface* coupling is essentially another name for *Abstract* coupling.

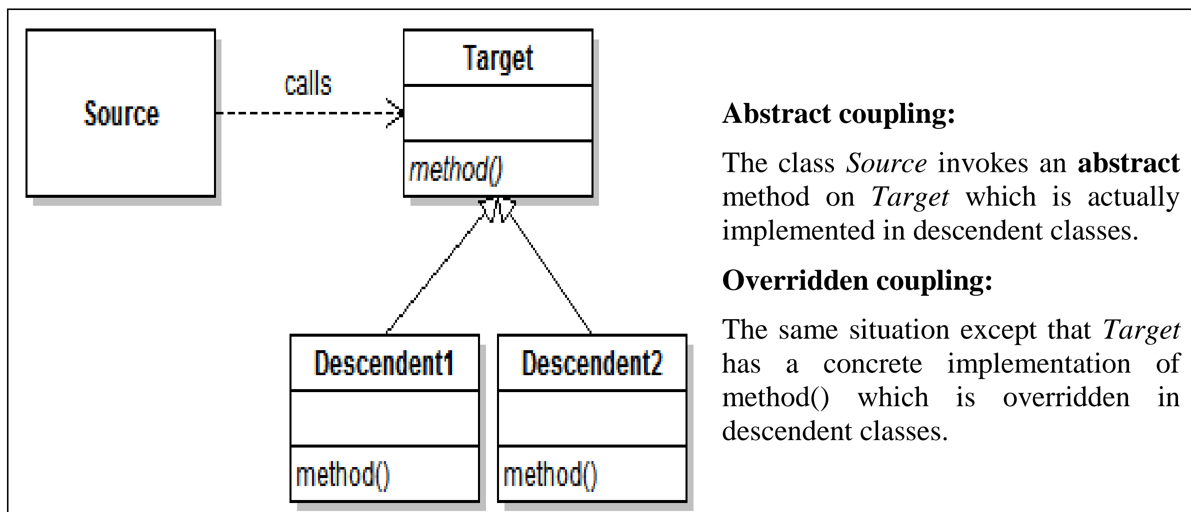

Figure 3. Abstract coupling

Again, there is an ambiguity as to which classes should be coupled: *Source* and *Target* or *Source* and one or both of the descendent classes. Because fields cannot be declared to be abstract, this kind of coupling is only applicable to method invocations.

3.3.4 Overridden coupling

Overridden coupling is similar to *Abstract* coupling in that subclasses may be the ultimate destination of the method invocation. The difference in this case is that the target class does provide a concrete implementation of the method being invoked, but it is also overridden in one or more subclasses of the target type. At runtime, the actual class of the object referred to by target will determine which of

the method implementations is actually invoked. Because fields cannot be overridden, this kind of coupling is only applicable to method invocations.

3.3.5 Constructor coupling

Constructor coupling (see Figure 4) refers to the invocation of a constructor rather than a method. Whether constructor calls should be considered to constitute a couple or not is ambiguous and depends on whether a constructor should be considered to be a special kind of method.

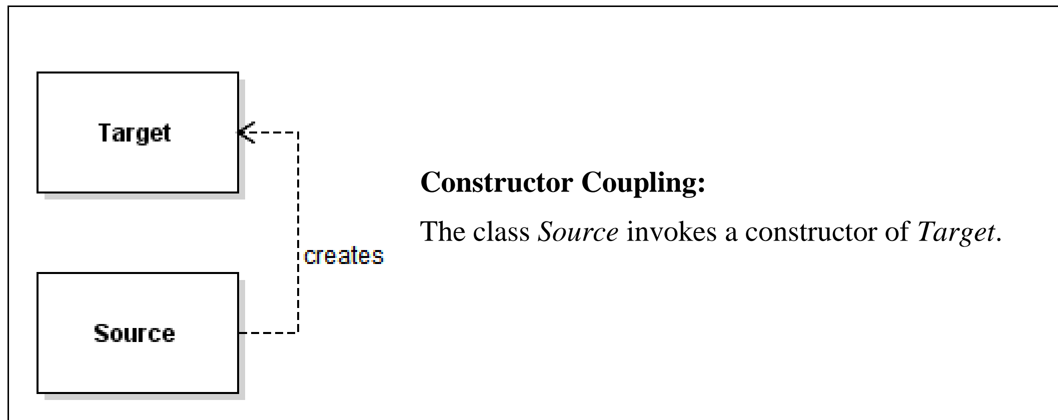

Figure 4. Constructor coupling

3.4 CBO subsets

To assess the potential impact of the variations of CBO used and the potential impact that using different variations of the metric causes, we felt that a more in-depth analysis was required. Studying all possible combinations of CBO metrics is impractical because of the range of combinations generated, so a subset of specific variants was selected and a rationale defined for this to be achieved. We decided that a pragmatic selection of what we thought would be the most valuable and interesting combinations of parameters would be the best way to proceed based on our understanding of the CBO.

As we have indicated, there are two sources for variations in CBO: those described by researchers and those implemented in tools. The originators of the CBO metric gave two slightly different definitions, and these were clarified and formalised in a wide-ranging review of metrics by Briand et al., [Briand99]. The formal definitions presented in that paper provide a suitable basis from which to classify other variations. Combining C&K's 1991 and 1994 definitions (CK1 and CK2, respectively), Briand termed these *CBO* for the final 1994 definition and *CBO'* for the earlier 1991 definition. These are given names and described in the subsections below. They are constructed using the word 'static' to indicate that the metric treats abstract and interface coupling according to Briand's static approach and 'poly' to indicate that Briand's polymorphic approach is used. Finally, CK1 or CK2 is used to indicate whether the first or second revision of C&K's definition is used with regard to couples within the inheritance hierarchy. The terms in the definitions used in the CBO variants that deviate from these categories (employing a 'mixed and match approach') are labelled accordingly. For example:

a. Couples within the same inheritance hierarchy are not counted (Local inheritance coupling as *per* Figure 1).
b. Couples to a method not implemented by the target class but inherited from one of its ancestors are counted as couples to the ancestor (Foreign inheritance coupling as *per* Figure 2).
c. Couples to abstract methods are not counted, and any classes implementing the methods are not counted (Abstract coupling as *per* Figure 3).
d. Couples to classes overriding an invoked method are not counted (Overridden coupling as *per* Section 3.3.4).

e. Couples to constructors are not counted (Constructor coupling as *per* Figure 4).

3.4.1 Static-CK1

The 1991 C&K definition as clarified by Briand et al., [Briand2001] is as *per* the example in the previous section (Section 3.4) with identical Conditions a-e.

3.4.2 Static-CK2

This variation is the revised 1994 C&K definition, as clarified by Briand et al., [Briand2001] and defined in an identical way to Static-CK1, except that, concerning Condition 'a' in Section 3.4, couples within the same inheritance hierarchy *are* counted.

3.4.3 Poly-CK1

This variation is the 1991 C&K definition as clarified by Briand et al., [Briand2001], but including Briand's polymorphic couples. It is the same definition as Section 3.4 except for Conditions c-e, where:

c. Couples to abstract methods are counted as couples to all classes implementing the method.
d. Couples to all classes overriding an invoked method are counted.
e. Couples to constructors are not counted.

3.4.4 Poly-CK2

The Poly-CK2 variation is the revised 1994 C&K definition, as clarified by Briand 2001, but including Briand's polymorphic couples. This is identical to Poly-CK1, except that couples within the same inheritance hierarchy *are* counted.

All four of these variations of the CBO metric are defined to be calculated as the union of import and export coupling sets for each class. In the survey of previous work, it was found that six studies counted import-only coupling, four (notably including Briand) counted the union of import and export coupling and for the remainder it was unknown. In practice, most tools appear to calculate CBO for the import set only. Of the tools surveyed, only Analyst4j documented that it used a union implementation of one of these initial variations (Static-CK2). Because of this common practice, each of these metrics will therefore be additionally calculated for the import set only.

3.5 Tools definition

Judging from their documented behaviour, `CKJM`, `JHawk` and `Understand for Java` all appear to use the following definition, implemented for import couples only:

a. Couples within the same inheritance hierarchy are counted.
b. Couples to a method not implemented by the target class, but inherited from one of its ancestors are counted as couples to the target.
c. Couples to abstract methods are counted as couples to the target.
d. Couples to classes overriding an invoked method are not counted.
e. Couples to constructors are counted.

It is important to note that these three tools and Borland Together appear to accept any reference to a class type as a couple to that type (i.e., they use reference coupling). In fact, if a tool counts coupling using reference coupling and makes no special exceptions or provisions, Tools is the algorithm effectively being used. `Borland Together` differs from Tools only by excluding couples within the same inheritance hierarchy (as *per* C&K's first definition [Chidamber1991]) and is also implemented as import coupling. `JHawk's` classic CBO would appear to be this algorithm extended to the

union set. Despite the stated behaviour of some of these tools being mostly identical, the results they yield have been shown to differ on the same systems. The conclusion we draw from this can only be that the stated behaviour is inaccurate or lacking in detail. Nonetheless, the most important distinguishing features (besides the use of import coupling) are evident:

a. Couples to methods that have been inherited are not resolved to the concrete location of the method in an ancestor class, but instead counted against the target class.
b. Couples to abstract methods are counted as if the destination were not abstract.
c. Invocations of constructors are counted as couples.

All of these three clearly conflict with the original definitions. To investigate the individual impact of these measures, three further metrics are defined to capture each of them. These metrics are each based on Static-CK2, as the Briand metric most closely approximating Tools when implemented for import coupling only. In each case, the point which differs is italicised.

3.5.1 Tool-based 'static-CK2-plus-abstract-targets'

This definition is a modification of Static-CK2 which additionally counts calls to abstract methods as calls to the class or interface declaring them. It is the same definition as for Static-CK2 except for Condition c.

c. Couples to abstract methods are counted as couples to the target.

3.5.2 Tool-based 'static-CK2-wth-foreign-inh-targets'

This definition is a modification of Static-CK2 which counts calls to foreign inherited methods as couples to the target, not the implemented site in an ancestor.

a. Couples within the same inheritance hierarchy *are* counted.
b. Couples to a method not implemented by the target class but inherited from one of its ancestors are counted as couples to the target.
c. Couples to abstract methods are not counted, and any classes implementing the methods are not counted.
d. Couples to classes overriding an invoked method are not counted.
e. Couples to constructors are not counted.

3.5.3 Tool-based 'static-CK2-plus-constructors'

This definition is a modification of Static-CK2 which additionally counts calls to constructors. This is to quantify the impact of inclusion of constructors.

a. Couples within the same inheritance hierarchy *are* counted.
b. Couples to a method not implemented by the target class but inherited from one of its ancestors are counted as couples to the ancestor.
c. Couples to abstract methods are not counted, and any classes implementing the methods are not counted.
d. Couples to classes overriding an invoked method are not counted.
e. Couples to constructors are counted.

To summarise, the metrics selected for this study based on the descriptions just provided are listed in Table 1. See Section 3.4.4 (last paragraph) for a description of the "Applied as" column. Table 2 provides the CBO variants used by each of the considered tools.

Base metric	Applied as	Remarks
Static-CK1, Static-CK2, Poly-CK1, Poly-CK2	UNION	These are the two original CK definitions, as clarified by Briand and each applied with and without Briand's polymorphic modification.
Static-CK1, Static-CK2, Poly-CK1, Poly-CK2	IMPORT	As tools widely implement CBO by counting only import couples, the same four metrics will be used again on the import set.
Static-CK2-plus-abstract-targets Static-CK2-with-foreign-inh-targets Static-CK2-plus-constructors	IMPORT	The three main inconsistencies found in tools, each applied independently to Static-CK2 and applied as import as tools typically do.
Tools	IMPORT	Effectively the combination of all three inconsistencies, representative of the kinds of implementation apparently present in tools.

Table 1. CBO variants studied

Tool	Variant metric used (according to documentation)	Applied as	Remarks
Analyst4j	Static-CK2	UNION	
CKJM, JHawk, Understand for Java, Borland Together	Tools	IMPORT	JHawk also offers the option of "classic CBO", which is the same variant but using union coupling. Borland Together differs slightly from this variant by excluding inheritance couples within the class hierarchy.
STAN, CCCC VizzAnalyzer	Unknown	Unknown	The documentation does not give details and whether the tools support CBO is unclear in some cases.

Table 2. Variants of CBO used by the tools

4. Case-study detail

Demonstrating a relationship between CBO and defect-proneness of classes can be achieved through either linear or logistic regression on the CBO value and defect status of classes in a given system. Logistic regression is the more commonly used approach [Cox1958] and, for that reason, was selected for this study. To perform regression, data indicating which classes in a system are defective is required. The data for this study was obtained by combining the information stored in a version control system with the accompanied issue tracking system (in this case, Git and Bugzilla, respectively). Other approaches that have been used include using data provided by the professional developers of a system or using students to develop a system and then test it for defects [Basili1996; Briand1999; Briand2000; Briand2001]. A fundamental pre-requisite for this approach is that the system selected for study must have version control and issue tracking data available. It is also necessary that the data contained in these systems is of relatively high quality so that it can be used effectively. This means that the project should, ideally, have strong conventions for commenting version commits and documenting development issues.

4.1 System under study

The Eclipse Java Development Tools (JDT) Core project was selected [Eclipse2013], since it is a large, mature open-source project with all sources available in the *Git* version control system and full, online access to the Bugzilla issue tracking system. As a subproject within a larger project, defects are discovered and reported by members of other teams resulting in high-quality issue reports and proper use of the issue tracking system. We also need to consider how the time interval over which defects are identified is defined. There are various ways of doing this, such as using all defects reported between two distinct versions of the system or examining the dates at which a defect was reported and subsequently fixed with regard to the date of the release of the system being analysed. This study used the simple method of including defects from the version of the system against which the defect was reported.

A previous study by [English2009] also used the JDT system and reported finding a comparatively strong correlation between CBO and defects along with Lines of Code (LOC) and the Response For a Class metric of C&K. The work drew comparisons and found better predictive capability than the earlier work of Gyimothy et al., [Gyimothy2005] where CBO was also found to be the best metric (along with LOC) for predicting the defect-proneness of classes. English et al., did not provide a detailed description or breakdown of CBO and cited a later definition of the metric provided according C&K as a basis of their empirical study [Chidamber1994]. However, the results resonate strongly with the work in this paper and motivate the importance of scrutinising CBO. One interesting conclusion made by English et al., further illustrates the importance of studying variations in the definition of the CBO. They suggest that differences in results between studies where CBO is evaluated may be as much to do with variations in the systems under study than with the metric itself. Variations in the definition of CBO imply that differently composed systems (comprising different emphases on coupling types) will produce different results when empirically evaluated for defect-prediction effectiveness.

4.2 Study methodology

Defects reported against versions 2.1 and 3.1 of the JDT system were used and defective classes identified then correlated with the CBO variant metrics calculated from the corresponding system releases. Version control commit operations were searched to identify a specific bug-id from the issue tracking system. However, in practice, some interpretation was required. Studies are not always explicit in describing this process, despite having a potentially critical impact on their results. Two general approaches have been used: a non-discriminatory assumption that *all* classes changed in relation to a defect report are considered defective and a discriminatory assessment of which classes should be considered to be at defect. These approaches will be referred to as *assumed defect location* and *assessed defect location*. Assumed defect location is easy to accomplish since the classes are listed as changed in the relevant version control commit records, and it is unambiguous since no subjective judgement needs to be made. Nevertheless, it is also likely that it will overestimate the number of defective classes. There are two reasons why this may happen. Firstly, the commits can contain changes unrelated to the defect and which happen to have been made at the same time. In the course of this study, we observed that a refactoring operation such as a variable renaming or correcting a spelling mistake in comments in the source code may have been carried out by a developer in the course of their work. A commit may, therefore contain ten changed classes for reasons such as these, while only one of them actually contains any meaningful corrective changes. The topic of 'tangled code' in which: "all changes to all modules appear related, possibly compromising the resulting analyses through noise and bias" was studied by Herzig et al., [Herzig2015]. In the same study of five open-source projects, up to 15% of defect fixes comprised of tangled changes. Secondly, the changes required to fix a defect are not necessarily confined to the actual site of the defect. Supporting changes are often required in other classes, not at defect in or of themselves. The disadvantage of assumed defect location is therefore that it inevitably over-estimates the number of defective classes. It is also reasonable to point out that not all issues reported in Bugzilla are actual defects. Issue reporters could misclassify feature requests and refactoring suggestions as defects. This could have a significant effect on the defect prediction process. In our defence, this was dealt with as 'part and parcel' of the assessment of individual issue reports. We did not filter on the issue classification field (so we inspected reports labelled 'enhancement' as well as all levels of severity) but we *did* classify issues as enhancements and eliminated them where it was deemed appropriate. This is an additional reason for using the assessed defect location approach, because, in assessing the issue, enhancements that had not been labelled as such were eliminated (instead of relying on the labels in the database).

Some studies report inspecting the classes to identify those at defect or as input to a defect prediction model [Briand1999; Elberzhager2014]. This process is referred to as 'assessed' defect location. For our study, the individual classes changed in the commits were examined together with the developer commentary to try and identify the true location of the defect. In practice, this was not always possible, and where it was not possible to make a reasonable judgement, no classes were marked as

defective. We believe that this process more accurately identified the defective classes. Once defect data was acquired, the CBO variant metrics were calculated for every class in the target version of the system and logistic regression performed for each metric individually against the defect data. The strength of the relationship was judged through the use of the odds ratios (confirmed by goodness-of-fit) measures and the difference between the variants quantified. The methodology can therefore be summarised as follows. The Eclipse Java Development Tools (JDT) Core project was used to acquire defect data and defects identified according to the version of the system they were reported against. For each set of defects, the defective classes were identified using the assessed defect location approach. Each of the CBO variant metrics was then calculated for every class of the target version(s) of the system. Logistic regression was performed for each metric using each of the four sets of defective classes. In each case, the relative strengths of the metrics as defect predictors were compared and the differences between the results sets noted. Table 3 gives the summary data from the two releases of the JDT studied.

Statistic	version 2.1	version 3.1
Issues reported	461	1104
No. of classes in system	685	873
No. of classes *assessed* as being defective	148 (22%)	262 (30%)
No. of classes *assumed* defective	369 (54%)	529 (61%)

Table 3. Summary statistics for the two releases

It is interesting to note that the proportion of classes in the system assessed as defective are comparable with 22% and 30%. The last row in the table shows the resulting number of defective classes when no discrimination is made and all classes changed in a commit matched to a defect-fix are assumed to be defective. As we might expect, more classes are considered defective; in each case approximately twice as many classes are getting marked as defective using this method than when judgement was exercised.

4.3 Results

In the results tables which follow, the intercept, coefficient and error of the regression model are given, together with the odds ratio and the likelihood ratio pseudo-R^2 goodness-of-fit measure. We make the following observations about the values in those tables

1. The p-value has been omitted as it is less than 0.0001 in every case.
2. The odds ratio and pseudo-R^2 are the most appropriate values for comparison of the metrics, and their values are illustrated by bars to make this comparison visually easier.
    a. An odds ratio of one suggests that there is essentially no relationship between the variables. The magnitude of the difference with one indicates the strength of the influence exerted. Although a stronger influence indicates that the metric is a better predictor of defect-proneness, it must be accompanied by a reasonable model fit to be given any credibility.
    b. The pseudo-R^2 provides a measure of goodness-of-fit. According to [Louviere2000], a rule of thumb for this value is that between 0.2 and 0.4 is considered a good fit. Higher values would be even better, but are not considered to be common for this measure.
3. Both the odds ratio and pseudo-R^2 columns are scaled to be directly comparable across all the result tables.
    a. In the case of the odds ratio, the data bar ranges from 1.0 to 1.5, values chosen simply for visual clarity.

   b. In the case of the pseudo-R^2 column, the data bar ranges from 0 to 0.5. As an additional aid to comprehension, where the value fails to reach the 0.2 threshold for acceptability, the figure is shown in fainter text.
4. For a metric to be considered a valid defect predictor, it should have both a strong odds ratio and a pseudo-R^2 of at least 0.2.

4.3.1 Logistic regression

Tables 4 and 5 show the results for versions 2.1 and 3.1 of the JDT Core system, respectively. One interesting observation is that the pairs of metrics that differ only by their use of the CK1 or CK2 rules for coupling within the inheritance hierarchy exhibit near identical results. For example, IMPORT-Poly-CK1 and IMPORT-Poly-CK2 have identical values in the results for version 2.1 and almost identical values in the results for version 3.1. The same is true for IMPORT-Static-CK1 with IMPORT-Static-CK2, UNION-Poly-CK1 with UNION-Poly-CK2 and UNION-Static-CK1 with UNION-Static-CK2. This indicates that, at least in this particular system, it is very rare for a couple within the inheritance hierarchy to significantly contribute to the net coupling of any class. It might even be that in version 2.1 of the system, *there is no significant* coupling of this type, while the small differences in values in the results for Version 3.1 indicates that there cannot be much. We conclude that the difference in the first and second revisions of the C&K definitions of CBO is likely to be of little consequence in practice, unless the system under analysis here is atypical and systems with a higher proportional occurrence of coupling within the inheritance hierarchy are common.

The next observation is that the metrics can be divided into two groups: those that have noticeably higher odds ratios and goodness of fit and those that have low values. In other words, there are those that exhibit a predictive property and those that do not.

Metric	Intercept	Coeff.	Error	Odds Ratio	Pseudo R^2
IMPORT-POLY-CK1	-1.7814	0.0185	0.0028	1.0187	0.0629
IMPORT-POLY-CK2	-1.7814	0.0185	0.0028	1.0187	0.0629
IMPORT-STATIC-CK1	-2.5012	0.1637	0.0171	1.1779	0.2062
IMPORT-STATIC-CK2	-2.5012	0.1637	0.0171	1.1779	0.2062
IMPORT-STATIC-CK2-PLUS-ABSTRACT-TARGETS	-2.6968	0.1528	0.0147	1.1651	0.2354
IMPORT-STATIC-CK2-PLUS-CONSTRUCTORS	-2.5977	0.1479	0.0152	1.1594	0.2169
IMPORT-STATIC-CK2-WITH-FOREIGN-INH-TARGETS	-2.5012	0.1637	0.0171	1.1779	0.2062
IMPORT-TOOLS	-2.7181	0.1347	0.0131	1.1442	0.2365
UNION-POLY-CK1	-1.9973	0.0173	0.0025	1.0174	0.0686
UNION-POLY-CK2	-1.9973	0.0173	0.0025	1.0174	0.0686
UNION-STATIC-CK1	-2.0509	0.0562	0.0072	1.0578	0.1310
UNION-STATIC-CK2	-2.0509	0.0562	0.0072	1.0578	0.1310
					2.1 reported / assessed

Table 4. Results for issues reported against version 2.1 using the assessed defect location technique

Metric	Intercept	Coeff.	Error	Odds Ratio	Pseudo R^2
IMPORT-POLY-CK1	-1.3803	0.0229	0.0026	1.0232	0.0905
IMPORT-POLY-CK2	-1.3867	0.0227	0.0026	1.0229	0.0894
IMPORT-STATIC-CK1	-2.0710	0.1840	0.0156	1.2020	0.2393
IMPORT-STATIC-CK2	-2.1232	0.1745	0.0148	1.1906	0.2317
IMPORT-STATIC-CK2-PLUS-ABSTRACT-TARGETS	-2.1716	0.1598	0.0134	1.1733	0.2418
IMPORT-STATIC-CK2-PLUS-CONSTRUCTORS	-2.1913	0.1546	0.0133	1.1671	0.2275
IMPORT-STATIC-CK2-WITH-FOREIGN-INH-TARGETS	-2.1471	0.1821	0.0155	1.1997	0.2332
IMPORT-TOOLS	-2.2082	0.1447	0.0125	1.1557	0.2310
UNION-POLY-CK1	-1.5857	0.0182	0.0019	1.0184	0.1000
UNION-POLY-CK2	-1.5846	0.0179	0.0019	1.0181	0.0994
UNION-STATIC-CK1	-1.7502	0.0721	0.0075	1.0748	0.1649
UNION-STATIC-CK2	-1.7573	0.0669	0.0070	1.0692	0.1587
					3.1 reported / assessed

Table 5. Results for issues reported against version 3.1 using the assessed defect location technique

The metrics which include polymorphism[1] all display an odds ratio close to one and a pseudo-R^2 value which falls short of acceptable. The same is true of the metrics which use union counting[2] (two of which also use polymorphism). For all these metrics, IMPORT-Poly-CK1, IMPORT-Poly-CK2, UNION-Poly-CK1, UNION-Poly-CK2, UNION-Static-CK1 and UNION-Static-CK2, the relationship to defects is shown to be weak statistically. According to these results, the system we studied and the methodology we used, these metrics have no practical application to the prediction of defects. (Appendix B provides data and analysis on the predictive value of each variation of CBO.)

The remaining metrics, IMPORT-Static-CK1, IMPORT-Static-CK2 and its variations (including Tools) all have an odds ratio greater than one as well as respectable pseudo-R^2 values, indicating that the relationship could be considered sound. The odds ratio does not represent a strong relationship for any metric, but the pseudo-R^2 value supports the relationship's existence. The metrics which do show a predictive property differ from each another in only subtle ways. They are all import metrics (counting couples *to* other classes and not *from* other classes). IMPORT-Static-CK1 and IMPORT-Static-CK2 are Briand's clarifications of the two definitions published by C&K and as discussed, show minimal differences. Then there are the three modifications of the CK2 metric using abstract targets, constructors and foreign inheritance targets and, finally, the Tools metric which combines all three modifications. Given that these metrics are closely related, it is not surprising that they display similar performances as predictors. In one case in version 2.1, IMPORT-Static-CK2-With-Foreign-Inh-Targets produces identical values to IMPORT-Static-CK1 and IMPORT-Static-CK2 (already noted to be identical to each other). This may be because the classes in the system do not exhibit the kinds of coupling the metrics treat differently. Again, this suggests that the differences in calculation method are inconsequential in practice. To confirm that this is not anecdotal it would be necessary to widen the study across multiple systems of different types. The difference in the odds ratios of these metrics is not large, but a pattern is identifiable in both versions presented. The variants which count constructors and abstract targets show a slightly weaker relationship than the others; the Tools metric which counts both constructors and abstract targets is the weakest in both cases. This is interesting because Tools is the variant closest to what most metrics tools appear to do. These results suggest, very tentatively, that including accesses to constructors and abstract method sites decreases the effectiveness of coupling as a predictor of defects.

We also created a logistic model based on size alone, then created classification tables for CBO only, CBO and size and size-only models. In both releases, the combined model performed best and the CBO-only model performed slightly better than the size-only model. Clearly it is possible to predict defective classes almost as well using size as it is using CBO. But using both is better than using either one; this implies that they both must be contributing unique information to the predicted value.

5. Discussion

5.1 Issues Raised

The preceding analysis also raises a number of issues for research studies. The first issue relates to the implications of the results for both past and future empirical studies of the CBO metric. While we have only studied two versions of one system as our basis, it is clear that, even if only one study shows significant differences between CBO values, then other systems may exhibit similar or even more pronounced differences. Put another way, comparison of studies where CBO was used would be highly problematic, unless those studies used identical definitions of CBO. It is well-known that excessive coupling is positively correlated with defects; consequently, if the independent variable has multiple interpretations, then no real sense of the extent of the overall correlation can be made.

[1] These are the variants in which couples are counted to all possible destination classes for abstract and overridden methods.
[2] These variants treat couples as bi-directional, so if class A uses class B, both A and B count it as a couple, instead of only A counting it as a couple.

Secondly, the study demonstrates, in a clear way, that making the assumptions explicit about what data is being collected is crucial to the conclusions that can be drawn. In our study, the choice of tool seems to determine what variety of CBO is collected; however, a developer or researcher should be able to use *any* tool available and be confident that the same data is being collected (or at the very least that there is an explanation of any variation from a standard definition) [Lincke2008]. At the moment, this is not the case. It is evident that a set of standards needs to be agreed on a core definition of CBO that all tools abide by, even at this stage in the lifetime of the CBO. There are also issues related to how easy it would be to replicate any study of coupling; any researchers would need to use the same tool as the original study and this is not always possible. Replication is difficult enough without the additional burden of interpreting the semantics of the metrics used. Some further interesting parallels and differences with the work of English et al., [English2009] can be shown from our results and these also point to important and potential areas of study. In their paper, CBO consistently showed itself to be a better predictor of defective or non-defective classes than other metrics (it performed even better than LOC and WMC as measures); however, no metric was found to be particularly good at predicting 'highly' defect-prone classes. In our paper, we *only* considered the former i.e., the relationship between CBO and whether a class was either 'defective' or 'not defective' (a binary distinction). This limits a comparison between our work and the work of English et al., to a certain extent, since only one of their four research questions maps to our study. However, it would be interesting to explore the *extent* of class defect-proneness and the relationship that such classes have with CBO in the two systems as further study. To do this would require both CBO and defects to be tracked across multiple versions of systems over time and this is a major empirical undertaking. In the work of English et al., C&K's Response for a Class (RFC) metric, essentially a coupling metric, also showed itself to be a good predictor of defect-prone classes. In this paper, we did not consider the relationship between CBO and RFC. To do so would have required an in-depth dissection of the RFC metric and then a comparison with CBO. The RFC metric is one of the C&K metrics that remains highly utilised in empirical studies yet difficult to understand. Interpreting the values it gives is also problematic and this rendered the metric less useful than the CBO. In their concluding remarks, English et al., also allude to further work examining the 'severity' of a defect and its relationship with CBO and this is an aspect that might also inform the results or our study. The problem of course is that defect severity is subjective only. Finally, we found that a combination of predictors performed better than any single metric (i.e., LOC and CBO rather than CBO by itself). In this sense, it would be interesting and informative to replicate our study using this result with the same data used by English et al. Overall, the results of English et al., justify the extensive use of CBO in empirical studies since there can be little doubt of the link between CBO and defects. In effect, our study reflects how careful we need to be in defining and interpreting coupling-based metrics, despite acknowledging that fact.

It is important to note that class size has intuitive relationships with both measures of coupling such as CBO and the propensity for defects [McIntosh2016; Koru2009]. A larger class contains more code and thus more opportunities for both couples to other classes and defective code to be present. CBO is therefore unlikely to be independent of class size and this is commonly measured by means of Pearson or Spearman correlations (Pearson's is a parametric measure making assumptions about the distribution, while Spearman's is non-parametric). Our tool for calculating the various CBO values operates on Java byte-code rather than the source code and therefore LOC values were not readily available. Instead, we followed the practice of Briand et al., [Briand1999] and measured class size as a count of the number of methods implemented in the class. We included constructors in this count as the aim was to approximate the amount of code the class consisted of. This was then correlated to the variant CBO metrics using Spearman's rho. The results are shown in Appendix A. The correlation values range between 0.36 and 0.54 for version 2.1 and 0.45 and 0.65 for version 3.1. These results confirm that all the variants of CBO have some overlap with class size (an effect that is slightly more pronounced in version 3.1 of the system). In most cases, the values are around 0.5 and could be described as only moderately positive [Briand1999]. Whilst this suggests that class size might also display a relationship with defects, we did not model this by regression as our aim was to compare variations of CBO with each other, rather than different metrics altogether. Since multiple correlations have been computed the *p*-values were adjusted to control for the false discovery rate. We therefore checked for the adjustment using the Benjamini-Hochberg False Discovery Rate (BHFDR)

[Benjamini1995] test. The test is designed to control for the expected proportion of incorrectly rejected observations (i.e., Type I errors). We found from the analysis that all *p*-values were all at least eleven places below the point. So, this would not affect the validity of the conclusions we have drawn. Table 6 shows the computed values.

	version 2.1		version 3.1	
Metric	***p*-value**	**BH FDR**	***p*-value**	**BH FDR**
IMPORT_POLY_CK1	2.43E-11	2.43E-11	8.26E-19	9.01E-19
IMPORT_POLY_CK2	2.43E-11	2.43E-11	1.17E-18	1.17E-18
IMPORT_STATIC_CK1	1.20E-21	2.40E-21	2.89E-32	1.73E-31
IMPORT_STATIC_CK2	1.20E-21	2.40E-21	7.00E-32	2.41E-31
IMPORT_STATIC_CK2_PLUS_ABSTRACT_TARGETS	3.66E-25	4.39E-24	1.11E-32	1.33E-31
IMPORT_STATIC_CK2_PLUS_CONSTRUCTORS	2.48E-22	9.90E-22	4.23E-31	8.47E-31
IMPORT_STATIC_CK2_WITH_FOREIGN_INH_TARGETS	1.20E-21	2.40E-21	8.02E-32	2.41E-31
IMPORT_TOOLS	1.21E-24	7.27E-24	3.75E-31	8.47E-31
UNION_POLY_CK1	5.56E-12	6.67E-12	7.19E-22	1.08E-21
UNION_POLY_CK2	5.56E-12	6.67E-12	1.09E-21	1.45E-21
UNION_STATIC_CK1	4.60E-15	6.89E-15	5.32E-22	9.12E-22
UNION_STATIC_CK2	4.60E-15	6.89E-15	1.25E-21	1.50E-21

Table 6. Benjamini-Hochberg False Discovery Rate analysis

5.2 Threats to validity

We also need to be mindful of a number of threats to the validity of the study presented. Firstly, we only studied two versions of one system, severely limiting the external validity of the study. However, we did not embark on this study with the intention of providing a large number of systems as an empirical basis. The aim of the study was to demonstrate, even for just a single system how varied (or otherwise) the CBO data could be. In that sense, the paper is different from an empirical study where we would try to analyse as many systems as possible to limit the external threat posed [Easterbrook2008].

Secondly, the basis of our study is that collecting coupling data accurately is of critical importance to understanding systems and drawing conclusions. However, it may be that in industry all that is usually required is a rough estimate of how much coupling there is in a class as a guide to re-engineering and similar activities. In that case, the differences may not be as relevant or as important as we portray.

Thirdly, the use in the study of two subsequent versions of the same system (JDT Core 2.1 and 3.1) suggests that there might be defects reported only against version 3.1 (but already present in version 2.1) or reported against both versions. In defence of this threat, version 2.1 (March 2003) was followed by three maintenance releases and a major release 3.0 (June 2004) with two subsequent maintenance releases of its own. Version 3.1 came after that in June 2005. We feel that the presence of a major release between the two studied systems to a certain extent minimises the carry-over of issues. Moreover, while version 2.1 was receiving maintenance releases, version 3.0 development was taking place on a separate branch. Finally, we have only studied certain combinations of CBO variants in this paper. Other variants might have illustrated or uncovered other interesting traits. However, we made a conscious decision to include what we understood to be the key combinations and those most likely to illustrate the point we were trying to make.

6. Conclusions and further work

In this paper, we used a set of collected metrics representing the different computational definitions of CBO as a basis of a study of its variations. The metrics were calculated for a single, large-scale Java system using a bespoke tool and logistic regression then used to correlate them with defect data obtained from the system's version tracking records. In terms of results, the different variations of CBO showed significantly different correlations to defects. In particular, we found that the metrics could be divided into two groups: those that exhibit a predictive property and those that do not. It is clear from the study that clarification or unambiguous re-definition of CBO is both desirable and essential for a general consensus on its use and application. Otherwise, it will become impossible to compare the results of empirical studies where CBO is used. Researchers need to be aware of the different applications of the metric and the actual method of calculation being used if conclusions and comparisons can be made as a result.

In terms of further work, we would like to run the collection tool on more systems and from different application domains. This might inform the results presented in this paper. It would also be useful to re-visit some past empirical studies to see whether different results would have been reported had different calculations of the CBO been used. Of course, the problem with this sort of study is that it relies on having the original data available and the problems that inevitably arise when conducting a replication. The tool and data used in this paper are available freely for other researchers to use; the data especially is available on request of the lead author. Finally, it would be interesting to undertake a similar study but using incoming and outgoing coupling metrics (i.e., by decomposing CBO into its two constituent parts).

Acknowledgements

This work was partly funded by a grant from the UK's Engineering and Physical Sciences Research Council (EPSRC) under grant numbers: EP/L011751/1 and EP/M024083/1.

References

[Arcelli2011] Arcelli Fontana, F., Mariani, E., Morniroli, A., Sormani, R., Tonello, A., An Experience Report on Using Code Smells Detection Tools, International Conference on Software Testing, Workshops 2011: pages 450-457.

[Arisa2005] VizzAnalyzer, available from: arisa.se/vizz_analyzer.php.

[Basili1996] Basili, V. R., Briand, L. C. and Melo, W. L. (1996) A validation of object-oriented design metrics as quality indicators, IEEE Transactions on Software Engineering, 22 (10), pp. 751-761.

[Benjamini1995] Benjamini, Y., Hochberg, Y., Controlling the false discovery rate: a practical and powerful approach to multiple testing, Journal of the Royal Statistical Society, Series B, 57 (1): 289–300, 1995.

[Borland2008] Borland (2008) Together, 12.5, Available from: http://www.borland.com/products/together/ [Accessed 08 July 2013].

[Bowring2004] Bowring, J.F., Rehg, J.M., Harrold, M.J., Active learning for automatic classification of software behavior. ACM/SIGSOFT International Symposium on Software Testing and Analysis, Boston, Massachusetts, 195-205, 2004.

[Brereton2007] Brereton, P., Kitchenham, B.A., Budgen, D., Turner, M., Khalil, M., Lessons from applying the systematic literature review process within the software engineering domain. Journal of Systems and Software 80(4): 571-583 (2007).

[Briand1998] Briand, L., Daly, J., Porter, V., and Wust, J., Predicting fault-prone classes with design measures in object-oriented systems, Proceedings International Symposium on Software Reliability Engineering, Paderborn, Germany, 1998, pp. 334-343.

[Briand1999] Briand, L. C., Wüst, J. K., Ikonomovski, S. V. and Lounis, H. (1999) Investigating quality factors in object-oriented designs: an industrial case study, Proceedings of the International Conference on Software Engineering, pp. 345-354, 1999.

[Briand1999] Briand, L. C., Daly, J. W. and Wüst, J. K. (1999) A unified framework for coupling measurement in object-oriented systems, IEEE Transactions on Software Engineering, 25 (1), pp. 91-121.

[Briand2000] Briand, L. C., Wüst, J. K., Daly, J. W. and Porter, D. V. (2000) Exploring the Relationships between Design Measures and Software Quality in Object-Oriented Systems, Journal of Systems and Software, 51 (3), pp. 245-273.

[Briand2001] Briand, L. C., Wüst, J. K. and Lounis, H. (2001) Replicated Case Studies for Investigating Quality Factors in Object-Oriented Designs, Empirical Software Engineering, 6 (1), pp. 11-58.

[Chhillar2011] Chhillar, R. S. and Nisha (2011) Empirical analysis of object-oriented design metrics for predicting high, medium and low severity faults using Mallows Cp, SIGSOFT Softw. Eng.Notes, 36 (6), pp. 1-9.

[Chidamber1998] Chidamber, S. R., Darcy, D. P. and Kemerer, C. F., Managerial use of metrics for object-oriented software: an exploratory analysis, IEEE Transactions on Software Engineering, 24 (8), pp. 629-639, 1998.

[Chidamber1994] Chidamber, S. R. and Kemerer, C. F. (1994) A metrics suite for object oriented design, IEEE Transactions on Software Engineering, 20 (6), pp. 476-493.

[Chidamber1991] Chidamber, S. R. and Kemerer, C. F. (1991) Towards a metrics suite for object oriented design, SIGPLAN Not., 26 (11), pp. 197-211.

[CodeSWAT2013] CodeSWAT.COM (2013) Analyst4j, 1.5, Available from: http://www.codeswat.com/

[Cox1958] Cox, D., The regression analysis of binary sequences (with discussion), Journal of the Royal Statistical Society B. 20: 215–242, 1958.

[Cruz2009] Cruz, A. E. C. and Ochimizu, K. (2009) Towards logistic regression models for predicting fault-prone code across software projects, 2009. 3rd International Symposium on Empirical Software Engineering and Measurement, pp. 460-463.

[Easterbrook2008] Easterbrook S., Singer J., Storey M.A., Damian D., Selecting Empirical Methods for Software Engineering Research. In: Shull F., Singer J., Sjøberg D.I.K. (eds.) Guide to Advanced Empirical Software Engineering. Springer, London, 2008.

[Eclipse2013] Eclipse Foundation (2013) Eclipse, Available from: http://www.eclipse.org.

[Elberzhager2014] Elberzhager, F., Münch, J., Rombach, D., Freimut, B., Integrating inspection and test processes based on context-specific assumptions. Journal of Software: Evolution and Process 26(4): 371-385 (2014)

[ElEmam2001] El Emam, K., Benlarbi, S., Goel, N. and Rai, S. N. (2001) The confounding effect of class size on the validity of object-oriented metrics, IEEE Transactions on Software Engineering, 27 (7), pp. 630-650.

[English2009] English, M., Exton, C., Rigon, I. and Cleary, B. (2009) Fault detection and prediction in an open-source software project, Proceedings of the 5th International Conference on Predictor Models in Software Engineering, Vancouver, British Columbia, Canada, New York, NY, USA: ACM, pp. 17:1-17:11.

[Gosling2015] Gosling, J., Joy, B., Steele, G., Bracha, G. and Buckley, A. (2015) The Java Language Specification, Java SE 8 Edition. Oracle Corporation.

[Gyimothy2005] Gyimothy, T., Ferenc, R. and Siket, I. (2005) Empirical validation of object-oriented metrics on open source software for fault prediction, IEEE Transactions on Software Engineering, 31 (10), pp. 897-910.

[Herzig2015] Herzig, K., Zeller, A., The impact of tangled code changes, Proceedings Working Conference on Mining Software Repositories, San Francisco, USA, 121-130, 2015.

[Herzig2013] Herzig, K., Just, S., Zeller, A., It's not a bug, it's a feature: how misclassification impacts bug prediction. International Conference on Software Engineering (ICSE), San Francisco, USA, pages 392-401, 2013.

[Johari2012] Johari, K. and Kaur, A. (2012) Validation of object oriented metrics using open source software system: an empirical study, SIGSOFT Softw.Eng.Notes, 37 (1), pp. 1-4.

[Jongeling2017] Jongeling, R., Sarkar, P., Datta, S., Serebrenik, A, On negative results when using sentiment analysis tools for software engineering research. Empirical Software Engineering, 22(5):2543-2584, 2017.

[Kitchenham2010] Kitchenham, B., What's up with software metrics? - A preliminary mapping study, Journal of Systems and Software 83(1):37-51, 2010.

[Koru2009] Güneş Koru, A., Zhang, D., El Emam, K., Liu, H., An Investigation into the Functional Form of the Size-Defect Relationship for Software Modules, IEEE Transactions on Software Engineering, 35(2): 293-304, 2009.

[Landman2014] Landman, D., Serebrenik, A., Vinju, J., Empirical Analysis of the Relationship between CC and SLOC in a Large Corpus of Java Methods, IEEE International Conference on Software Maintenance and Evolution, Victoria, Canada, 2014, pages 221-230.

[Lincke2008] Lincke, R., Lundberg, J. and Löwe, W. (2008) Comparing software metrics tools, in: Proceedings of the 2008 international symposium on Software testing and analysis, Seattle, WA, USA, New York, NY, USA: ACM, pp. 131-142.

[Littlefair2006] Littlefair, T. (2006) CCCC, 3.1.4. Available from: http://cccc.sourceforge.net/.

[Louviere2000] Louviere, J. J., Hensher, D. A. and Swait, J. D. (2000) Stated Choice Methods: Analysis and Applications. Cambridge: Cambridge University Press.

[McCabe1976] McCabe, T., A Complexity Measure, IEEE Transactions on Software Engineering, 2(4):308–320, 1976.

[McIntosh2016] McIntosh, S., Kamei, Y., Adams, B., Hassan, A., An Empirical Study of the Impact of Modern Code Review Practices on Software Quality, Empirical Software Engineering 21(5): 2146-2189, 2016.

[Martin1994] Martin, R. C. (1994) Object Oriented Design Quality Metrics: An analysis, Clean Coder, Uncle Bob Consulting LLC.

[Odysseus2011] Odysseus Software GmbH (2011) STAN, Available from: http://stan4j.com [Accessed 2011].

[Ping2002] Ping, Y., Systa, T. and Muller, H. (2002) Predicting fault-proneness using OO metrics. An industrial case study, 2002. Proceedings of the Sixth European Conference on Software Maintenance and Reengineering, pp. 99-107.

[Rosner2014] Rosner, P., Child, M. and Counsell, S. (2014) Visualising Java Coupling and Fault Proneness, 5th International Conference on Information Visualization and Applications, Lisbon, Portugal, 5-8 January 2014.

[Sauer2011] Sauer, F. (2011) Eclipse Metrics plug-in, Available from: http://metrics.sourceforge.net.

[Scientific2013] Scientific Toolworks, I. (2013) Understand, Available from: http://www.scitools.com.

[Shatnawi2008] Shatnawi, R. and Li, W. (2008) The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process, Journal of Systems and Software, 81 (11), pp. 1868-1882.

[Singh2010] Singh, Y., Kaur, A. and Malhotra, R. (2010) Empirical validation of object-oriented metrics for predicting fault proneness models, Software Quality Control, 18 (1), pp. 3-35.

[Singh2009] Singh, Y., Kaur, A. and Malhotra, R. (2009) Software Fault Proneness Prediction Using Support Vector Machines, in: Proceedings of The World Congress on Engineering 2009, London, U.K., July 1-3, International Association of Engineers, pp. 240-245.

[Spinellis2012] Spinellis, D. (2012) CKJM, 1.9. Available from: http://www.spinellis.gr/sw/ckjm/

[Stevens1974] Stevens, W. P., Myers, G. J., Constantine, L. L., Structured design, IBM Systems Journal, 13(2): 115–139, 1974.

[Tang1999] Mei-Huei Tang, Ming-Hung Kao and Mei-Hwa Chen (1999) An empirical study on object-oriented metrics, Proceedings. Sixth International Software Metrics Symposium, pp. 242-249.

[Tiwari2018] Tiwari, S., Singh Rathore, S., Coupling and Cohesion Metrics for Object-Oriented Software: A Systematic Mapping Study, Proceedings of the 11th Innovations in Software Engineering Conference (ISEC '18). ACM, Article 8, 11 pages.

[Virtual2013] Virtual Machinery (2013) JHawk, Available from: http://www.virtualmachinery.com/jhawkprod.htm

[Wilkie2000] Wilkie, F. G. and Kitchenham, B., Coupling measures and change ripples in C++ application software, Journal of Systems and Software, 52 (2–3), pp. 157-164, 2000.

[Yuming2006] Yuming, Z. and Leung, H. (2006) Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults, Software Engineering, IEEE Transactions on Software Engineering, 32 (10), pp. 771-789.

[Wohlin2014] Wohlin, C (2014) Guidelines for snowballing in systematic literature studies and a replication in software engineering. Empirical Assessment in Software Engineering, (EASE 2014). London, UK, 38:1-38:10

Appendix A – Size Correlations (v2.1 and v3.1 – Spearman's rho)

Metric (v2.1)	**NUM_METHODS**
IMPORT POLY CK1	0.36
IMPORT POLY CK2	0.36
IMPORT STATIC CK1	0.42
IMPORT STATIC CK2	0.42
IMPORT STATIC CK2 PLUS ABSTRACT TARGETS	0.44
IMPORT STATIC CK2 PLUS CONSTRUCTORS	0.44
IMPORT STATIC CK2 WITH FOREIGN INH TARGETS	0.42
IMPORT TOOLS	0.45
UNION POLY CK1	0.38
UNION POLY CK2	0.38
UNION STATIC CK1	0.54
UNION STATIC CK2	0.54

Metric (v3.1)	**NUM_METHODS**
IMPORT POLY CK1	0.49
IMPORT POLY CK2	0.47
IMPORT STATIC CK1	0.54
IMPORT STATIC CK2	0.49
IMPORT STATIC CK2 PLUS ABSTRACT TARGETS	0.50
IMPORT STATIC CK2 PLUS CONSTRUCTORS	0.53
IMPORT STATIC CK2 WITH FOREIGN INH TARGETS	0.50
IMPORT TOOLS	0.54
UNION POLY CK1	0.46
UNION POLY CK2	0.45
UNION STATIC CK1	0.65
UNION STATIC CK2	0.63

Appendix B – CBO as a predictor (v2.1 and v3.1)

To assess the effectiveness of the CBO metrics at defect prediction the regression model was used to build classification tables comparing the classes predicted as being defective to those recorded as being defective. Where the regression model gave a probability of 0.5 or more it was interpreted as a prediction of a defective class; Tables B.1 and B.2 show the results for versions 2.1 and 3.1 of the system. It can be observed that the various metrics display differing predictive properties. Those that achieve at least a 25% improvement over the results expected for proportional by-chance predictions are indicated as significant. It can be noted that even the best of these metrics incorrectly identify defective classes as non-defective more often than they correctly identify defective classes, however, given that metrics such as CBO are usually used along with other metrics in multivariate models and that our aim is to compare versions of CBO rather than establish it as a defect predictor *per se*, these flaws have little impact on this study.

Cases: 685 of which 148 true (t) and 537 false (f). Proportional By-Chance-Accuracy Rate: 66.12%					
Metric	**Correct**	**Incorrect**	**% Correct**	**% Improved**	**Sig?**
IMPORT POLY CK1	543 (20t, 523f)	142 (128t, 14f)	79.27%	19.88%	FALSE
IMPORT POLY CK2	543 (20t, 523f)	142 (128t, 14f)	79.27%	19.88%	FALSE
IMPORT STATIC CK1	575 (52t, 523f)	110 (96t, 14f)	83.94%	26.94%	TRUE
IMPORT STATIC CK2	575 (52t, 523f)	110 (96t, 14f)	83.94%	26.94%	TRUE
IMPORT STATIC CK2 PLUS ABSTRACT TARGETS	574 (58t, 516f)	111 (90t, 21f)	83.80%	26.72%	TRUE
IMPORT STATIC CK2 PLUS CONSTRUCTORS	573 (53t, 520f)	112 (95t, 17f)	83.65%	26.50%	TRUE
IMPORT STATIC CK2 WITH FOREIGN INH TARGETS	575 (52t, 523f)	110 (96t, 14f)	83.94%	26.94%	TRUE
IMPORT TOOLS1	578 (58t, 520f)	107 (90t, 17f)	84.38%	27.61%	TRUE
UNION POLY CK1	545 (20t, 525f)	140 (128t, 12f)	79.56%	20.32%	FALSE
UNION POLY CK2	545 (20t, 525f)	140 (128t, 12f)	79.56%	20.32%	FALSE
UNION STATIC CK1	548 (28t, 520f)	137 (120t, 17f)	80.00%	20.98%	FALSE
UNION STATIC CK2	548 (28t, 520f)	137 (120t, 17f)	80.00%	20.98%	FALSE

Table B.1 Defect prediction performance in version 2.1

Cases: 873 of which 262 true (t) and 611 false (f). Proportional By-Chance-Accuracy Rate: 57.99%					
Metric	**Correct**	**Incorrect**	**% Correct**	**% Improved**	**Sig?**
IMPORT POLY CK1	617 (52t, 565f)	256 (210t, 46f)	70.68%	21.87%	FALSE
IMPORT POLY CK2	617 (52t, 565f)	256 (210t, 46f)	70.68%	21.87%	FALSE
IMPORT STATIC CK1	709 (125t, 584f)	164 (137t, 27f)	81.21%	40.05%	TRUE
IMPORT STATIC CK2	706 (120t, 586f)	167 (142t, 25f)	80.87%	39.45%	TRUE
IMPORT STATIC CK2 PLUS ABSTRACT TARGETS	711 (129t, 582f)	162 (133t, 29f)	81.44%	40.44%	TRUE
IMPORT STATIC CK2 PLUS CONSTRUCTORS	706 (120t, 586f)	167 (142t, 25f)	80.87%	39.45%	TRUE
IMPORT STATIC CK2 WITH FOREIGN INH TARGETS	707 (126t, 581f)	166 (136t, 30f)	80.99%	39.65%	TRUE
IMPORT TOOLS1	706 (121t, 585f)	167 (141t, 26f)	80.87%	39.45%	TRUE
UNION POLY CK1	648 (78t, 570f)	225 (184t, 41f)	74.23%	28.00%	TRUE
UNION POLY CK2	647 (76t, 571f)	226 (186t, 40f)	74.11%	27.80%	TRUE
UNION STATIC CK1	683 (94t, 589f)	190 (168t, 22f)	78.24%	34.91%	TRUE
UNION STATIC CK2	683 (94t, 589f)	190 (168t, 22f)	78.24%	34.91%	TRUE

Table B.2 Defect prediction performance in version 3.1