Impact of Inheritance on Metrics for Size, Coupling, and Cohesion in Object-Oriented Systems

Dirk Beyer, Claus Lewerentz, and Frank Simon

Software Systems Engineering Research Group
Technical University Cottbus, Germany
(db|cl|simon)@informatik.tu-cottbus.de

Abstract. In today's engineering of object oriented systems many different metrics are used to get feedback about design quality and to automatically identify design weaknesses. While the concept of inheritance is covered by special inheritance metrics its impact on other classical metrics (like size, coupling or cohesion metrics) is not considered; this can yield misleading measurement values and false interpretations. In this paper we present an approach to work the concept of inheritance into classical metrics (and with it the related concepts of overriding, overloading and polymorphism). This is done by some language dependent *flattening* functions that modify the data on which the measurement will be done. These functions are implemented within our metrics tool *Crocodile* and are applied for a case study: the comparison of the measurement values of the original data with the measurement values of the flattened data yields interesting results and improves the power of classical measurements for interpretation.

1 Introduction

The measurement of object oriented systems seems to be a powerful tool for the qualitative assessment (cf. [5]). The availability of about 200 object oriented metrics [14] – i.e. metrics which are defined on object oriented systems -- and many books that consider the measurement of object oriented software, confirm this assumption. In general, these metrics can be classified into coupling, cohesion and size metrics. Inheritance is covered as a separate concept with its own metrics (e.g. *depth of inheritance, number of children, number of parents,* cf. [9]). However, most metrics for size, coupling and cohesion within the object oriented area ignore inheritance.

This paper shows another view: we demonstrate the impact of inheritance on other classical metrics, like size, coupling or cohesion metrics. The basic idea is the following: imagine a class A with 20 public methods, and a class B that inherits from class A and adds 15 additional public methods. Ignoring inheritance might yield the interpretation that class A is greater or even more complex than class B (as suggested by the measurement values for *number of public methods*: 20 for class A and 15 for class B). This is false when considering the functional size of a class as the set of

R. Dumke and A. Abran (Eds.): IWSM 2000, LNCS 2006, pp. 1-17, 2001.
© Springer-Verlag Berlin Heidelberg 2001

methods that a client class can use. This set remains unchanged for class A but increases to 35 methods for class B.

To examine this phenomenon in more detail we have to consider in depth the concept of inheritance and all concepts that might be touched by it, i.e. overriding, overloading and polymorphism. The idea of representing a class as it really is, i.e. considering all inherited attributes and operations, was introduced by Meyer [10]: his function *flat* constructs the flat representation of a class. In the field of object oriented measurement this concept is called *inheritance context* [3]: it allows for a selection of superclasses to be flattened into a subclass. With respect to measurement this includes flattening the associations between classes: Particularly in polymorph structures client classes are often coupled only with the interface class. Subclasses of the interface class are not coupled with the client class. Upon using the flattening process every client class with the possibility of calling operations of any subclass, is now also coupled with it.

Although it is very straight forward to consider flattening in measurement it had not been yet examined in detail, but nearly mentioned as a theoretic possibility (cf. [1], Section 4.2.4) or treated by various rare used metrics without a substantial knowledge of their behaviour or implementation details (e.g. *number of methods overridden by a subclass, number of methods inherited by a subclass, or number of methods added by a subclass*, cf. [9]). One point might be the necessity of adjusting the concept of flattening to each of the special programming languages with respect to the inheritance related concepts of overriding, overloading and polymorphism. This task is not easy, especially for C++.

This paper gives a brief overview of how the flattening has to work for C++ projects. For more details of how multiple inheritance or inheritance chains is treated see [13].

By applying flattening to large C++ systems we want to examine two points:

1. How are measurement values of classical size, coupling and cohesion metrics changed; and,
2. How can the quantitative analysis of the flattening itself be interpreted (e.g. how many methods, attributes or associations have to be copied into a subclass).

This examination is done by calculating five classical metrics for a case study. Firstly we use the usual technique, i.e. we ignore the inheritance structure; and secondly we flatten all classes before measuring them. We show that there are clear differences between the two approaches and that the quantitative analysis of the flattening itself gives many important hints for understanding the system.

This paper is structured as follows: in Section 2, the basic concepts of overloading, overriding and polymorphism are explained with respect to our flattening functions. We give an overview of these concepts for the language C++ and introduce some corresponding flattening functions. Section 3 shows the impact of flattening on measurement values of five classical metrics, and Section 4 explains how the quantitative analysis of the flattening itself can be interpreted to get a better understanding of the system and some worth suggestions for restructuring. The paper closes with a summary.

2 Overloading, Overriding, and Polymorphism in C++

Extending the simple addition of functionality into subclasses (cf. Section 1), we have to consider the concepts of overloading, overriding and polymorphism, because they might not only add but also modify functionality. These concepts only occur, if two method names (without considering parameters) are identical. Attributes can not be overloaded but they can be overridden in some cases (see below). Such situation where two members (either attributes or methods) with the same name (either locally declared or inherited) are visible in one class we call *name conflict*.

In the following, we discuss overloading, overriding and polymorphism separately for attributes and methods, and we distinguish the cases where name conflicts occur and where the name is unique.

For each situation we define a special flattening function which copies some members, i.e. methods or attributes, into a subclass with respect to some given rules. All flattening functions have three parameters: a superclass, the class where to copy members into and which is a direct subclass of the superclass, and the type of inheritance.

2.1 Attributes without Conflicting Names

Without name conflicts for attributes, the technique of flattening classes within an inheritance structure is simple: the attributes of a superclass are copied into all subclasses, if they are still visible there, which depends on the attribute's visibility.

Before defining the flattening function for attributes with unique names we have to think about ***wrapped attributes,*** which are attributes that are not directly accessible, i.e. they have the visibility private, but they are accessible by visible get and set methods. The encapsulation principle of object oriented design advises against the direct access to any attribute. The recommended construction is to access these attributes by special get methods and set methods. Our consideration of attributes above is blind to attributes used in this way, although ignoring these attributes within the subclass does not reflect the real situation: imagine a class A with 10 private attributes and each attribute has one get method and one set method. If then a class B inherits from this class and does not add, modify or delete anything there would be no reason why class A and class B should be considered as different. However, if measured separately, this would happen if no flattening or even if our previous flattening function would be applied. Therefore, also these wrapped attributes have to be considered for the flattening process. This kind of visibility we call *invisible* (i.e. not visible but indirectly accessible) in contrast to *inaccessible* (i.e. not visible and not accessible).

Now we are able to define the flattening function for attributes that do not have name conflicts:

flatten_unique_attribute_names (*superclass, subclass, inheritance type*): This flattening version copies all attributes (visible and invisible ones) of the *superclass* into the *subclass* if an attribute with the same name does not exist in the *subclass*. The attribute's visibility in the target class depends on the *inheritance type* and the *attribute's visibility*.

2.2 Attributes with Conflicting Names

In some cases overriding of attributes exists: If a subclass defines an attribute with the same name as a visible attribute of a superclass, the subclass' attribute overrides the superclass' attribute, i.e. it is no longer directly accessible but only by the use of the scope operator. For the later effect on measurement, it has to be decided, whether also attributes that are accessible only by explicitly using the scope operator have to be considered or not. For the purpose of this paper, which is to show the impact of inherited members on measurement values, we do not consider the use of members via the scope operator because of the following reasons:

1. For static members it is applicable to all classes independently of any inheritance relation between them.
2. The re-definition of an attribute within a subclass is a deliberate decision. Using the scope operator bypasses this decision. We can not really imagine examples in which the use of both attributes, the inherited via scope operator and the re-defined one, might make much sense.
3. In real-life applications we analysed (cf. [7]) the scope operator is not used very frequently.
4. Using the scope operator increases the dependencies between the particular classes. The reuse of a class is much more difficult because it is not sufficient to reproduce all use-relations and inheritance relations but it is necessary to adapt all code using the scope operator to the new environment.

Because of this we do not need to define a further flattening function for attributes that have the same name, because if two such attributes exist, the one of the superclass can be ignored.

2.3 Methods without Conflicting Names

The flattening function for methods with unique names is similar to the flattening function for attributes with unique names: the methods of a superclass are copied into the subclass if they are still visible there.

As done for attributes we have to think about *wrapped methods*, which are methods that are not directly accessible, i.e. they have the visibility private, but are accessible through other visible methods. This concept of private auxiliary methods is often used to extract some specific functionality. For our purpose all private methods that are used by a non-private method have to be considered for the flattening process. Their visibility within the target class is changed to *invisible*. Now we can define the flattening function for methods that do not have name conflicts:

flatten_unique_method_names (*superclass, subclass, inheritance type*): This flattening version copies all methods (visible and invisible ones) of the *superclass* into the *subclass* if no method with the same name exists in the *subclass*. The method's visibility in the target class depends on the *inheritance type* and the *method's visibility*.

2.4 Methods with Conflicting Names

Methods within a subclass can overload (for the definition of similar operations in different ways for different data types or numbers) or override (for a redefinition of a method) a method of a superclass. It overrides if the *signatures* [11] are identical and it overloads if the names are equal but at least one parameter is different (for details cf. [13]).

With respect to inheritance there is a very important point to remember if dealing with overriding and overloading in C++: a method in a subclass will override all methods with the same name from the superclass, but not overload them! So, whenever two methods from two classes in an inheritance relation have the same name, the flattening function must not copy the method from the superclass into the subclass and, therefore, there is no need for an additional flattening function.

2.5 Polymorphic Use-Relations

In this section we deal with *polymorphism*, in particular *run-time polymorphism* (in contrast to *compile-time polymorphism* that can be achieved by overloading and that can be solved by the compiler) (cf. [12]). The technique to implement this kind of polymorphism is called *late binding*. This means that the determination which version of a function is called when a message with appropriate name occurs is made at run time and is based on the caller's object type. In C++ this type of late binding is only possible if an object is accessed via pointer (or via reference, which is an implicit pointer, [12], pp. 341) and if the called method is declared `virtual`.

If considering polymorphic structures it is not always possible to decide at compile-time which function will be executed at run-time. Due to this, the static analysis of polymorphic use-relations between classes is often reduced to use-relations between superclasses [8].

Because the static analysis of source code always considers only potential of use and not the actual frequency of use, there should be a function *flatten_polymorphic_use_relations* that copies the use-relations into all methods of the subclass that could be executed by calling a method in a superclass. This flattening function depends not only on the superclass and subclass but also on the class that uses methods of the superclass [13].

The corresponding flattening function is defined as follows:

flatten_polymorphic_use_relations (*superclass, subclass, inheritance's type*): This flattening version adds use-relations that cover calls of the client class to public methods of the subclass. This is done in the following way:
A use-relation between a method m_{usingc} of the client class and a public method m_{subc} of the subclass is added if
the inheritance's type is public,

- m_{usingc} uses the method $m_{superc,}$ declared in the superclass,
- m_{superc} is defined as virtual,
- m_{superc} is overridden or implemented by m_{subc} within the subclass.

3 Impact of Flattening on Size, Coupling, and Cohesion Metrics

The flattening functions are implemented within the metrics tool *Crocodile* ([8], [13]) that allows to measure large object oriented systems and to apply the flattening functions. This section demonstrates the impact of the flattening process on some typical object oriented metrics. At first we describe a case study in detail and give a quantitative overview how much information was added to the flattened version. Afterwards we explain very briefly some typical metrics (two size, one coupling and two cohesion metrics) that we used for our exploration. Some diagrams show the differences between the measurement values of the normal and the flattened source code data. Afterwards we analyse these value changes in detail and give some specific interpretations, which show, that the values obtained from the flattened version reflect the intuition more appropriately than the traditional values.

3.1 Description of the Case Study

As case study, we used the source code of the Crocodile metrics tool itself. We know the sources in detail and so we are able to evaluate the results; it is written completely in C++; it uses inheritance -- which is not true for all C++ projects, especially for those originally written in C -- ; it is based on a GUI framework; and it uses a library for data structures. This integration of external components is typical for today's software products and demonstrates the necessity of a selection mechanism for the flattening process because not all superclasses have to be flattened in all subclasses [13].

The analysed system consists of 113 files and 57 classes. It has a maximum inheritance depth of 4, has 18 inheritance relations, and does not use multiple inheritance.

These numbers showed to be invariant against the flattening functions because no classes, files, or inheritances are added. The table below gives a quantitative overview about the classes: the first column describes the kind of considered objects, the second counts its occurrences in the original data, the third counts its occurrences in the flattened data and the last shows the percentage increase of the occurrences in the flattened data.

Table 1. Quantitative overview on original and flattened sources

	Original sources	Flattened sources	% Increase
Methods	589	706	+20%
Attributes	226	314	+39%
Use_Method_Relations	666	692	+4%

To demonstrate the impact of the flattening process on the measurement values for classical metrics we chose five typical object oriented metrics, i.e. two size metrics, one use-based coupling metric and two use-based cohesion metrics. In Sections 3.2. to 3.4. we show how the values for these classical metrics change by our flattening functions. In Sections 4.2. to 4.4. we interpret these changes in detail and give some specific interpretations.

3.2 Impact of Flattening on Size Metrics

Two widespread-used size metrics for object oriented source code are the *number of methods* (*NoM*, cf. *NOM* in [4], *number of instance methods* in [9]) as indicator for functional size and the *number of attributes* (*NoA*, cf. *NOA* in [4], *number of instance variables* in [9]) as indicator for size of encapsulated data.

The impact of flattening on the metric *NoM* is shown by the distribution of the proportional changes within the system (cf. Figure 1). The x-axis is divided into intervals of percentage of the measurement value changes before and after flattening; the y-axis shows the count of classes having a measurement value change within the interval. As displayed the measurement values remain unchanged for 39 classes (0% changing of the values). These are exactly the classes that have not any superclass. On the other side the range of change is between 101 and 200 percent for 12 classes and between 1401 and 1500 percent for one other class !

The same kind of diagram for the metric *NoA* is shown in Figure 2; to be able to display proportional changes also for classes having no attributes before flattening, we set these proportional changes to the value: *number of added attributes* * 100.

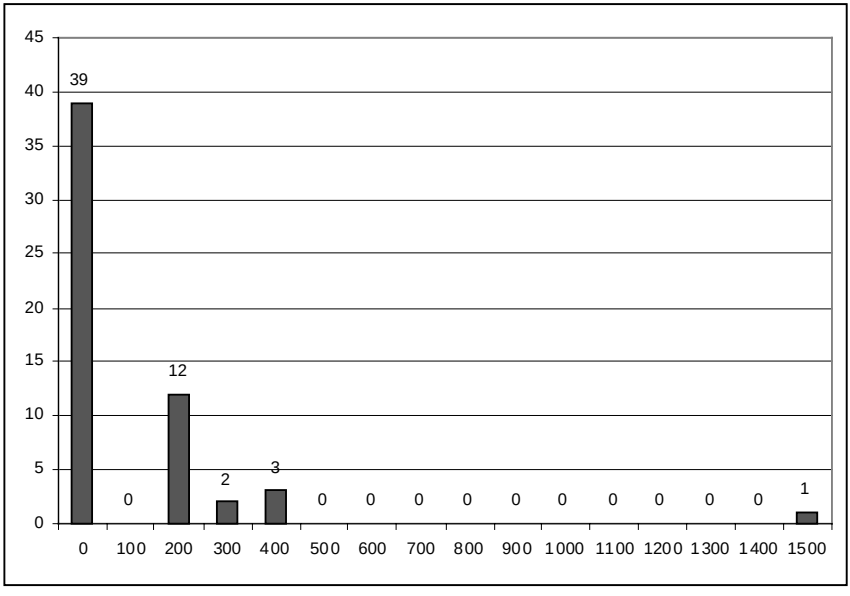

Fig. 1. Distribution of proportional measurement value changes for NoM

As in the previous Figure the values did not change for 39 classes because they have not any superclass. The other values are changed a lot again.

As demonstrated for both metrics, *number of methods (NoM)* and *number of attributes (NoA)*, it is clear that size metrics are very sensitive to our flattening process: Over 31 % of the measured values changed, one up to 2.900 % ! Thus, ignoring inheritance for size metrics might yield misleading numbers which in turn might yield false interpretations. Some classes which seem to be very small (and thus easy to understand and to maintain) are in fact very large (and thus difficult to

understand and to maintain) because they get (and need) a lot of functionality and data from their superclasses.

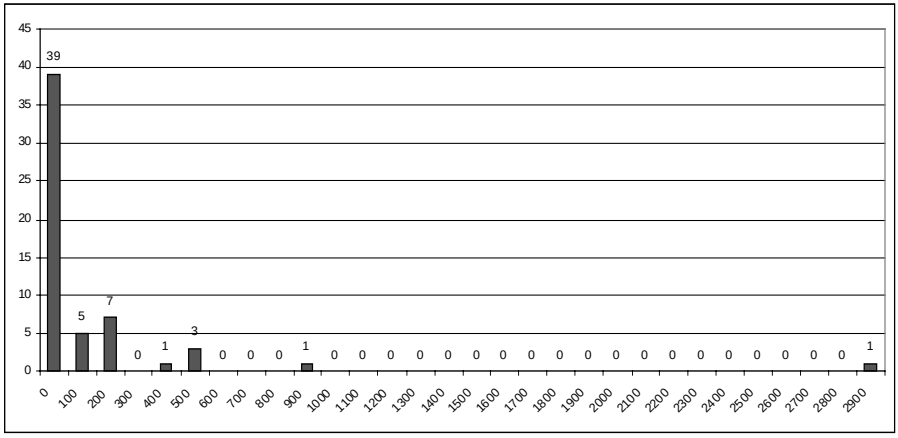

Fig. 2. Distribution of proportional measurement value changes for NoA

3.3 Impact of Flattening on Coupling Metrics

The coupling between entities – in our case classes – covers their degree of relation with respect to a special system abstraction. Very often the entities plus their use-relation (e.g. *RFC* or *fan-out* in [4]) give this abstraction.

As a typical example for this class of metrics we chose the metric *number of externally used methods*, i.e. the count of distinct external methods called by methods within a class (*efferent coupling to methods*, *effCM*). The distribution of the proportional changes of the original and flattened version is shown in Figure 3; the classes that have a measurement value of 0 before flattening are treated like in the visualisation of the *NoA* values.

For this metric only the measurement value of one class that has a superclass did not change.

As shown before, also in this category of metrics, many low measurement values increase: Nearly 30 % of the measurement values changed, three up to 400 % ! Only upon using the flattening functions the usual reduction of coupling to superclasses within an inheritance relation is extended to coupling to the subclasses. Because use-based coupling is an important indicator for class understanding and class clustering (e.g. into subsystems, cf. [7]) these changes of measurement values have a strong impact. Thus, ignoring inheritance for coupling metrics might give a misleading picture of a system which again might suggest false restructuring actions.

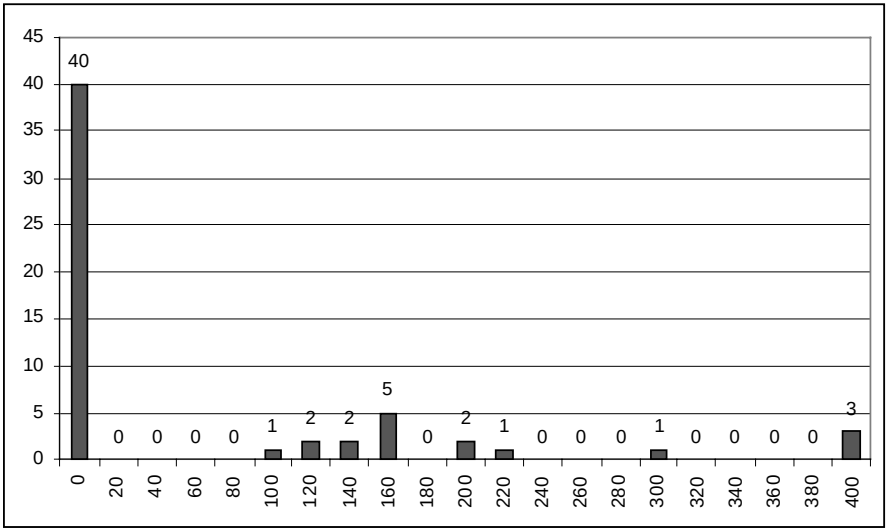

Fig. 3. Distribution of proportional measurement value changes for effCM

3.4 Impact of Flattening on Cohesion Metrics

The third important category of metrics for object oriented systems is the area of cohesion, i.e. to which degree the members of an entity belong together. We examine two different types of cohesion for two different abstraction levels: at first we use the well-known inner class cohesion measure *LCOM* [2], which assumes cohesion within a class to be based only on attribute use-relations. Secondly we use a generic distance based cohesion concept developed at our metrics research team [6]. Here we instantiated it on the class level in such a way that cohesion between classes is based on method-to-method use-relations plus method-to-attribute use-relations: The more two classes use from each other the smaller is their distance.

For the *LCOM* measure, we got the distribution diagram of the measurement value changes as displayed in Figure 4. Again, the increase of the measurement values is obvious, which suggests a weaker class cohesion. However, the validation of this result within the system can not be made: The problem is the very specific point of view to cohesion for *LCOM*, i.e. cohesion is based only on attribute use-relations: Due to flattening many methods are copied into the subclass. With them, the used private attributes are copied as wrapped attributes. Of course no newly implemented method within the subclass has direct access to this attribute, i.e. the *LCOM* value increases. The solution to consider only non-wrapped attributes within subclasses also increases the *LCOM* values because then some methods do not use any attribute of the subclass.

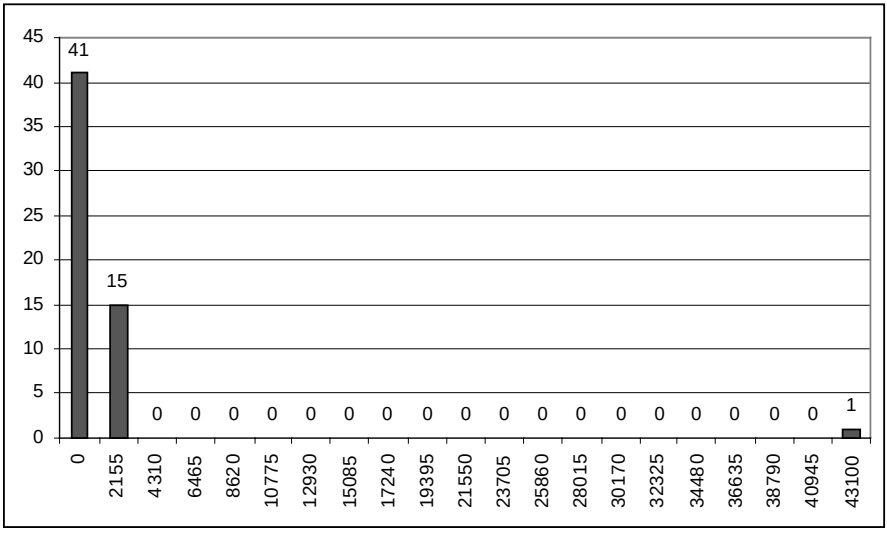

Fig. 4. Distribution of proportional measurement value changes for LCOM

The main result of these measurement changes is that attribute-based cohesion decreases heavily for classes having at least one superclass.

The opposite -- which is an increase of cohesion -- is the result of the second view on cohesion: The changes of the measurement values can not be displayed as above because the cohesion measurement values are not measured for classes but for pairs of classes, which for example can be used as geometric distances within a visualisation (cf. Section 4.4). The flattening of one class X might change all measurement values of class pairs where X is part of. For the given 57 classes there exist 57^2 values. The frequency of relative changes for the two versions is displayed within Figure 5.

Most distances remain unchanged (2995) or increase only minimal (122; cumulated within the interval]0.05..0] to 3117). The most obvious changes are the decreases of distances (~4% of the distances), i.e. the cohesion increases. This comes from the added coupling to subclasses. Ignoring this fact might yield the false interpretation that a class X is only cohesive to a used superclass Y but not to Y's subclasses, which in turn might suggest the false subsystem creation, i.e. to separate Y's subclasses from class X.

On the other side, the flattening can also decrease cohesion: If a class X heavily interacts with the whole functionality of a class Y that has a superclass, one might assume a high degree of cohesion. The flattening process, however, reveals that class X uses only a little part of the functionality of class Y because the latter one inherits a lot of functionality from its superclass. Thus, cohesion decreases.

As shown in this section, also cohesion metrics are sensible to the flattening process. Thus, ignoring inheritance for cohesion metrics might give a misleading impression about the cohesion of the system.

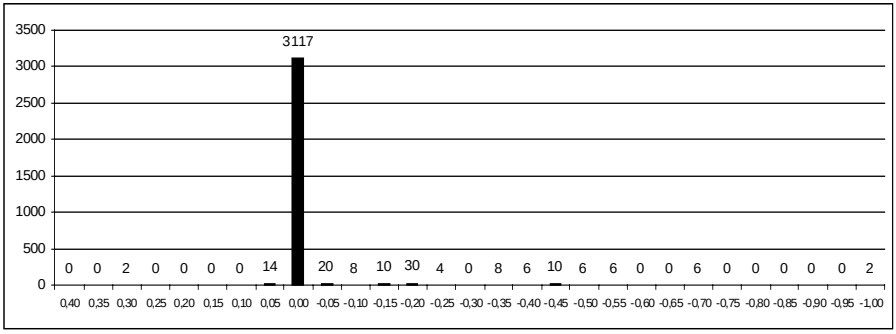

Fig. 5. Frequency of relative changes for class cohesion metric within both versions

4 Quantitative Analysis of the Flattening Process Itself

The last section demonstrates how heavily the flattening process changes several kinds of software measures and how misleading the separate consideration of inheritance for the other metrics might be. In this section we do not examine only the value changes over the whole system but try to interpret them; additionally, we try to interpret value changes for single classes that might give additional information. Thus, the quantitative analysis of the flattening functions itself -- i.e. how many members gets a subclass -- can be seen as a new kind of metrics that gives a new kind of information.

At first we try to interpret the value changes for the whole system. Afterwards for every metric we show a ranking list of the classes for which the flattening process changed the values most, and give an interpretation.

4.1 Quantitative Analysis of the Overall Flattening Results

While flattening the following observations were conspicuous: None of the added methods is a wrapped method and only 13 of the added attributes are wrapped attributes. This has the following reasons:

1. In many cases inheritance is used to create type hierarchies, i.e. the superclass is used as abstract class or only even as interface for subclasses.
2. The partial implementations within the superclasses use some private attributes but never private methods: the functionality of methods that are flattened into subclasses is low and does not need any private auxiliary method but only some private attributes.

Thus, it is possible to characterise the pre-dominant use of inheritance within a system by only analysing these numbers. This kind of analysis is not possible with any

typical inheritance metric like *depth of inheritance*, *number of children* or *number of parents* (cf. [9]).

4.2 Quantitative Analysis of the Results of Flattening on Size Metrics

To get an overview on the effects on some specific measurement value changes a trend diagram is shown: there only those measurement values are displayed which changed by applying the flattening process. For each changed class the value before and after flattening are shown in grey or black respectively. The measurement pairs are ordered by their relative increase. We renamed all classes in all diagrams from `class1` to `class57.`

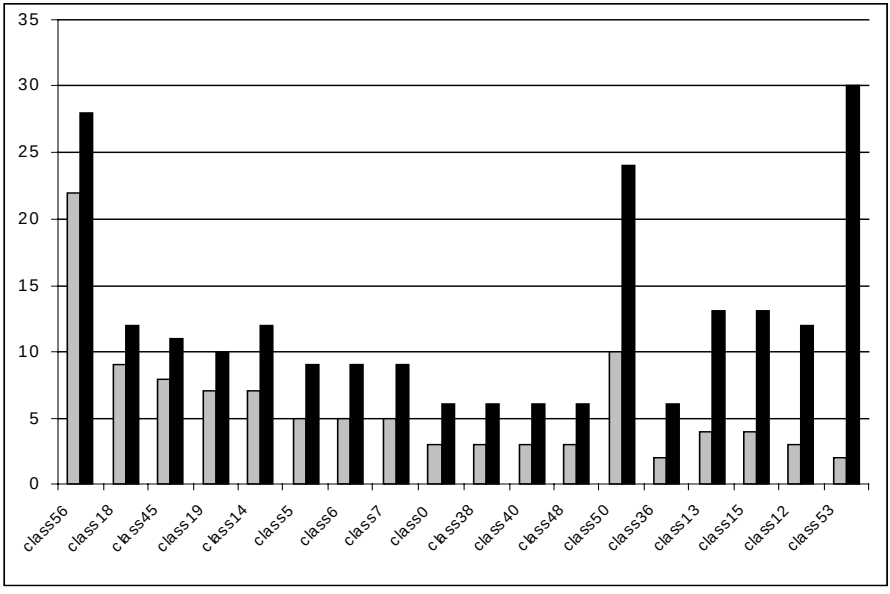

Fig. 6. Measurement values of NoM before and after the flattening process

In Figure 6 we contrast the measurement values of the non-flattened classes with the flattened classes for the metric *NoM*. We only discuss the results for the classes with the most extreme values. The highest relative increase is 15: `class53` has 2 methods before the flattening process and has 30 methods after the flattening process. This class is a subclass of the parser class *FlexLexer* that comes with the *flex* library package and defines the interface of the class for the lexical analyser. In fact, our subclass does not add or modify anything but defines its own constructor and destructor. In this case, inheritance is used only to change the given class name. Thus, the flattened version corresponds to our knowledge that `class53` is a very large class that offers a lot of functionality.

The same measurement visualisation is used for the metric *NoA* in Figure 7:

as for *NoM*, the impact of flattening is clear: 12 classes of the non-flattened version that seem to be anomalous --because they have no attributes (classes `0, 36, 38,`

`40, 48, 5, 6, 7, 13, 14, 15, 53`) -- are changed in a way that they have at least one attribute, i.e. they seem to be more than only a container of functions. On the other side, there are two classes with 29 attributes (classes `53` and `56`): Both classes again belong to the parser functionality of the analysed system: A thorough analysis of this situation revealed, that the inheritance used is very debatable and should be renovated.

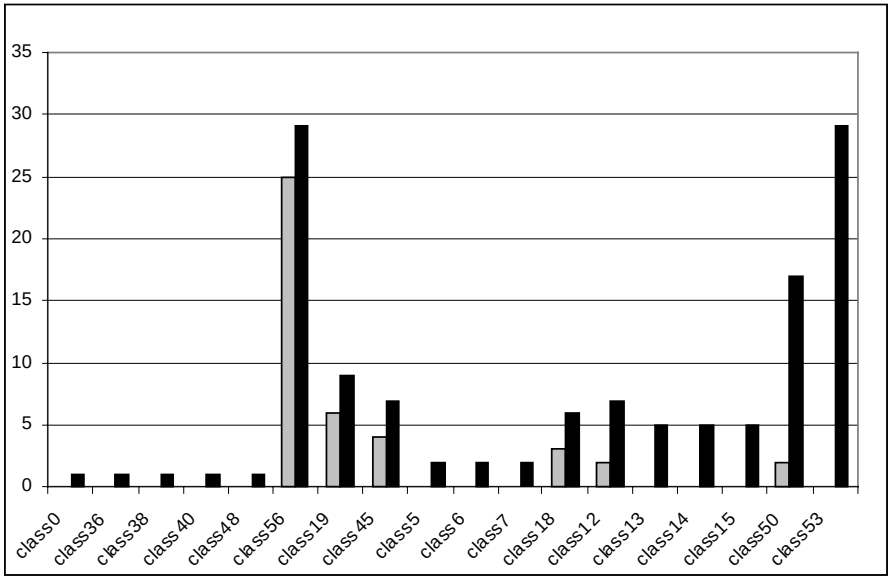

Fig. 7. Measurement values of NoA before and after the flattening process

4.3 Quantitative Analysis of the Flattening Process on Coupling Metrics

As above we contrast the value changes of the non-flattened classes (grey bars) to the flattened classes (black bars) for the metric *effCM* (cf. Figure 8).

It is very interesting that the measurement value of `class5` increases to 49 (before 34): This class is used for the calculation of a special cohesion concept that is integrated into the analysed system. It calculates the cohesion values by using a lot of interfaces. There are many different kinds of cohesion, which are all compatible to one cohesion concept, implemented by different subclasses. Thus, after flattening, the coupling is extended from the interface to all its implementations.

The highest increase occurs for classes `0, 38` and `50`: All three classes are subclasses of an abstract superclass that is partially implemented. Because this implementation is inherited by all subclasses, the subclasses get additional coupling.

Another interesting point is the increase of four classes that have no coupling at all within the non-flattened system (classes `14, 15, 18` and `45`). All four classes are abstract superclasses that have an implementation based on pure virtual methods of the same class, i.e. they have only couplings to methods of the same class. But by resolving the polymorphic structure (by adding potential use-coupling) the

implemented methods of the superclass have couplings to subclasses implementing the pure virtual methods. Only the "flattened" measurement reflects the coupling to all inheritance sub-trees. Neglecting these additional couplings could yield the false clustering to separate some concrete implementations from the interface and also from a client class. This view on a system based on potential use-coupling changes measurement values in a very interesting way because it shows coupling that is not necessary for compiling (all four superclasses of the last example are compilable without including any other class of the analysed system) but for runtime (otherwise there would be linker errors).

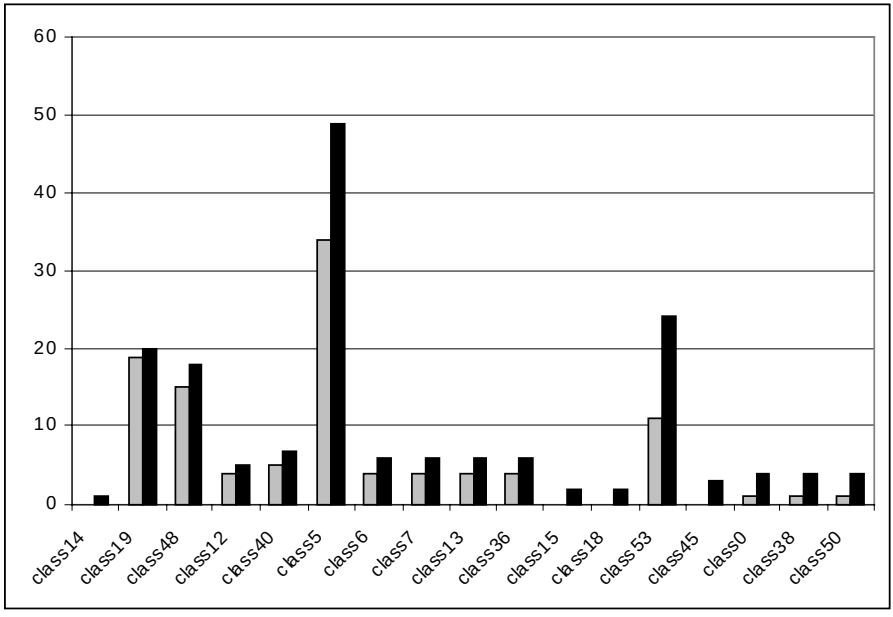

Fig. 8. Measurement values of effCM before and after the flattening process

4.4 Quantitative Analysis of the Flattening Process on Cohesion Metrics

In Figure 9 the measurement values of *LCOM* are displayed for usual and for the flattened version. As explained in Section 3.4., these value changes are not very informative because of the conceptual weakness of *LCOM*. Thus, this diagram we present only for completeness.

Much more information is available by examination of flattening for our second kind of cohesion. A 3D-visualisation (cf. [6]) based on the values calculated from the flattened version reflects better our knowledge about the measured system. Because of the difficulties to reproduce the 3D-visualisation in black/white printing, in the following we use a 2D-representation of the same data. Two classes having a shorter distance are more cohesive than two classes having a longer distance. The absolute position of a class is not interpretable but its distances from other classes. In Figure 10

this visualisation is shown for the usual system; in Figure 11 the flattened version is presented.

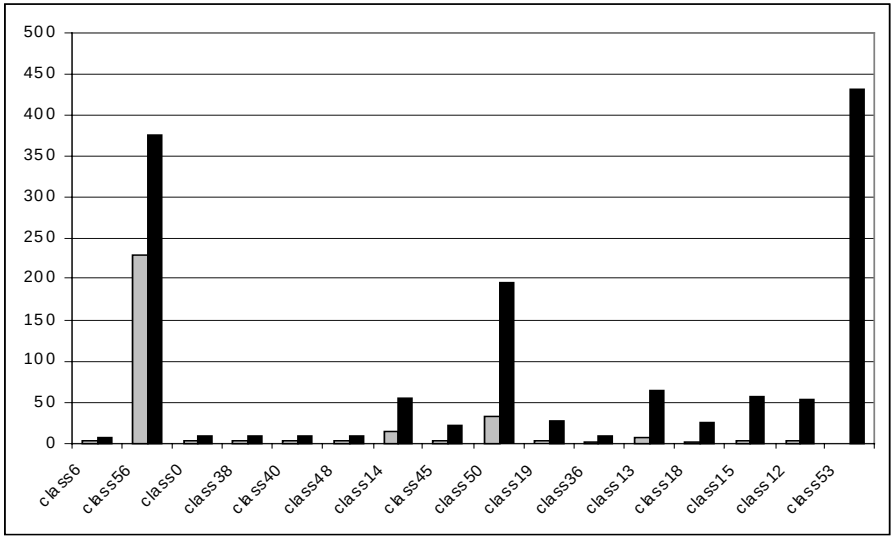

Fig. 9. Measurement values of LCOM before and after the flattening process

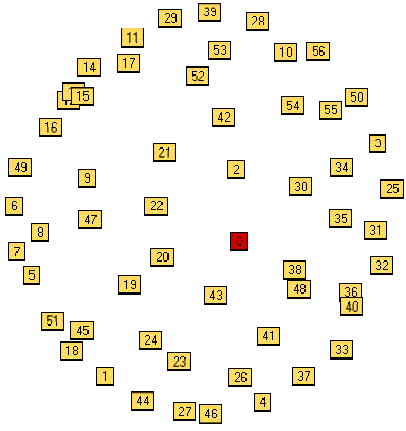

Fig. 10. Distance based cohesion before Flattened

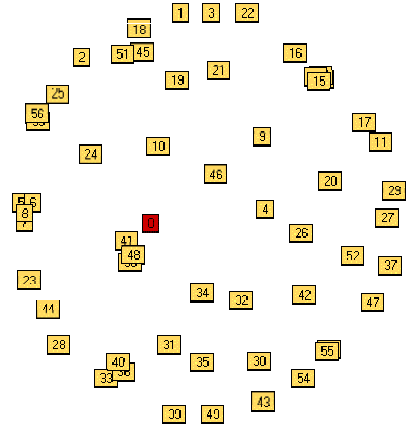

Fig. 11. Distance based cohesion after flattened

The first impression is that the flattened version has more clusters of classes like classes {41, 48, 38 (partly hidden)}, {5, 6, 7, 8}, {55,50 (hidden)} or {56, 53 (partly hidden)}. The latter cluster is the one whose cohesion values increase most, i.e. the distance decreases from 0.958 to 0.001 (cf. the interval –0.95 to –1 within Section 3.4). This are the same classes that attracted our attention in the interpretation of the *NoM* and *NoA* values (classes 53 and 56). In fact, a class that inherits from another class and adds or modifies nearly nothing is very cohesive to its superclass because they are nearly identical. They provide the same functionality, they have the same coupling, etc.

On the other side there exists one class pair whose distance increases from 0.605 to 0.939, i.e. their cohesion decreases (`class53` with `class54` having no superclass, i.e. that is not mentioned in the other considerations). In this case the client `class54` has an object of the subclass `class53` as member and uses some functionality from it. However, `class53` inherits a lot of functionality from its superclass that is not used by `class54`, i.e. the cohesion between `class53` and `class54` decreases. Unfortunately, the superclass of `class53` is a reused class adapted by inheritance. A good restructuring would be to reduce at least the visibility of methods of the superclass that are not used within `class53` or to reuse the class by aggregation.

5 Summary

Measuring object oriented systems without considering the possibility to distribute functionality and data over several classes within an inheritance structure is only one view. We have motivated the view in which every class is changed to its flattened presentation, i.e. inherited members are also considered, because it gives another interesting view on a system. Afterwards we have presented a concept for the programming language C++ that explains, how this flattening process should work in detail. There we concentrated on how to handle inherited attributes, inherited methods, how to resolve polymorphism, and how corresponding flattening functions have to be defined.

These concepts are implemented in the metrics engine Crocodile allowing us to get experience with our new flattening concept. Within our case study we validated our hypothesis that this new view on a system might be helpful by the following three points:

1. Within the non-flattened version, we detected many classes whose interpretation of the measurement values would be misleading. If the view on a system that is presented by the measurement values might suggest false interpretations, a further view seems to be useful.
2. With the flattened version we detected many classes that indeed showed some anomalies and that indeed would be good candidates for restructuring. Nevertheless, there also exist constraints in this version that made us marking some outliers as difficult and not necessary to change, e.g. using standard classes like *FlexLexer*.
3. Considering the differences between the measurement values of the flattened and non-flattened version produces another interesting view on a system. This information gives new insights into the kind of inheritance, e.g. if inheritance is

used only for source code sharing, for defining an interface or for creating type hierarchies.

Because of the very interesting results of this work we will investigate in porting our concepts to other object oriented languages like Java. First experiences of the application to large Java projects like JWAM (cf. http://www.jwam.de) looks very promising, especially for the visualisation within our generic cohesion concept.

References

1. Briand, L.C., Daly, J.W., Wüst, J.: A unified framework for cohesion measurement in object-oriented systems. Fraunhofer Institute for Experimental Software Engineering, Technischer Bericht ISERN-97-05 (1997)
2. Chidamber, S., Kemerer, C.: A metrics suite for object oriented design. IEEE Transactions on Software Engineering, 20(1994)6, 476-493
3. Erni, K., Lewerentz, C.: Applying design-metrics to object oriented frameworks. In: Software metrics symposium, IEEE Computer Society Press (1996) 64-74
4. Henderson-Sellers, B.: Object-oriented metrics – measures of complexity. Prentice Hall, New Jersey (1996)
5. Köhler, G., Rust, H., Simon, F.: Understanding object oriented software systems without source code inspection. Published in Proceedings of Experiences in reengineering workshop, Edited by Oliver Ciupke, Karlsruhe (1999)
6. Lewerentz, C., Löffler, S., Simon, F.: Distance based cohesion measuring. In: Proceedings of the 2nd European Software Measurement Conference (FESMA) 99, Technologisch Instituut Amsterdam (1999)
7. Lewerentz, C., Rust, H., Simon, F.: Quality - Metrics - Numbers - Consequences: Lessons learned. In: Reiner Dumke, Franz Lehner (Hrsg.): Software-Metriken: Entwicklungen, Werkzeuge und Anwendungsverfahren. Gabler Verlag, Wiesbaden (2000) pp. 51-70
8. Lewerentz, C., Simon, F.: A product metrics tool integrated into a software development environment. In: Proceedings of object-oriented product metrics for software quality assessment workshop (at 12th ECOOP), CRIM Montreal (1998)
9. Lorenz, M. Kidd, J.: Object-Oriented Software Metrics – A practical guide. Prentice Hall, New Jersey (1994)
10. Meyer, B.: Object-oriented Software construction. Prentice Hall, London (1988)
11. Meyer, B.: Object-oriented software construction. 2nded., Prentice Hall, London (1997)
12. Schildt, H.: C++: The Complete Reference. 3rd edition, McGraw-Hill, Berkeley (1998)
13. Simon, F., Beyer, D.: Considering Inheritance, Overriding, Overloading and Polymorphism for Measuring C++ Sources. Technical Report 04/00, Computer Science Reports, Technical University Cottbus, May (2000)
14. Zuse, H.: A framework of software measurement. de Gruyter, Berlin (1998)