Assessing Inheritance for the Multiple Descendant Redefinition Problem in OO Systems

Philippe Li-Thiao-Té, Jessie Kennedy and John Owens
Department of Computer Studies, Napier University, Canal Court,
42 Craiglockhart Avenue, Edinburgh EH14 1LT, Scotland, UK
e-mail: {p.li, j.kennedy, j.owens}@dcs.napier.ac.uk
http://www.dcs.napier.ac.uk/osg
Tel: +44 (0)131-455 5340 Fax: +44 (0)131-455 5394

Abstract

Current use of inheritance has illustrated that the introduction of conceptual inconsistencies is possible in a class hierarchy. This paper discusses the reasons why complete method redefinition infringes the essence of inheritance. A redefinition metric set is proposed and practical experiments demonstrate that the results obtained permit the detection of inheritance design problems. Appropriate design decisions are suggested.

Keywords: inheritance, object-oriented metrics, object-oriented design, method redefinition, class hierarchy, Smalltalk

1. Introduction

"*Systems are not born into an empty world*" stated Meyer [26]. The inheritance mechanism is one of the key points for the extendibility and reusability aspects of object-oriented (OO) systems [3, 8, 12, 13, 17, 26, 27, 28, 31]. Due to the inherent incremental development of a class hierarchy, it is important to consider the future additions of new classes [18, 29, 30] as they will influence the shape and structure of the hierarchy. Recently, a variety of models of inheritance have been well described by Taivalsaari [31]. Although they offer a vast extent of expressiveness, each of these mechanisms are still subject to conceptual design inconsistencies [2, 8, 12, 29]. In order to reuse the potential of classes, designers face the problem of property (attribute and method) reuse and method redefinition. The latter is a powerful mechanism which permits behavioural flexibility in a class hierarchy but can also affect the correctness of a class if wrongly used [18, 26, 28, 29]. This paper shows how the complete method redefinition mechanism in a superclass-subclass relationship pinpoints potential design problems in ancestors classes.

The increased interest in metrics for OO systems has been significant in the last five years [1, 9, 10, 15, 16, 20, 22, 23, 24, 25] following the pioneering work of Chidamder and Kemerer [9] with their OO metrics suite. We show how the use of measurement techniques for assessing the mechanism of method redefinition provides insights into the overall behaviour of a class hierarchy. Pragmatic experiments using our redefinition metric set were carried out on both commercial

Published in OOIS '97 conference proceedings, Brisbane, Australia, 8-12 Nov. 1997.

libraries and on small, medium-size information systems. Specifically, the contributions of this paper are:

- an identification of design inconsistencies resulting from the multiple method redefinition problem in a class hierarchy,
- the proposition of a method redefinition metric set for assessing inheritance from a behavioural viewpoint,
- empirical validation of the metric set, results obtained from the Smalltalk class library are presented.

In section 2, we will explore the formal definition of inheritance from a property inheritance viewpoint. A description of the method redefinition variants and associated problems is given in section 3 followed by our proposed redefinition metric set in section 4. Derivation of the metrics, results and analysis are explained in section 5. Finally, we discuss related work on metrics for assessing inheritance in OO sytems and consider further work.

2. Properties Inheritance Scheme

Inheritance is the main mechanism which supports the realisation of criteria such as reusability and flexibility [17, 26]. An addition of a class to an existing class hierarchy specialises a branch of the tree, thereby extending it. By inheriting features from ancestor classes, reusability is also achieved. However, there exists many models of inheritance and the correct application of any model is debatable [2, 26]. The formal definition of inheritance is characterised as follows [4, 31]:

$$\boxed{C = P \oplus \Delta C} \quad (1)$$

where a new class C is shown as a combination (⊕) of a set of properties inherited from an existing class P and the new properties (Δ) which make C a specialised version of P. In this equation, the relation superclass/subclass is assumed to be transitive, therefore P includes all cumulated properties from its own parents (C is also transitive). However, the inheritance scheme of properties from parent class to child class is open to many interpretations. Taivalsaari [31] explained that P represents the properties inherited from an existing object or class where, in fact, C is able to inherit from many classes either in the same descendant branch or multiple branches if in a multiple-inheritance situation. It is generally accepted that the deeper a class is in a hierarchy, the more difficult the control of inheritance becomes. Therefore, leaf classes are more subject to bad design than their parents.

If a subclass is to inherit its parents' properties, then the set of properties of a subclass SubCls of a class Cls becomes :

$$\boxed{\text{SubCls} = \text{Properties (Cls)} \oplus \text{Properties (SubCls)}} \quad (2)$$

where

SubCls < Cls i.e. SubCls *is_a* subclass of Cls,

Properties (class) = { inst | inst ∈ <Attributes>, mth | mth ∈ <Methods>}

Properties (class) is the set of attributes and methods of a class i.e. <Attributes>

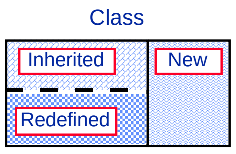
and <Methods> respectively refers to the set of possible instance variables and the list of methods in the class.

Introducing the origin of properties in (2) gives:

(3) $\boxed{\text{SubCls} = \text{Properties}_{\text{inherited}}\ (\text{SubCls}) \oplus \text{Properties}\ (\text{SubCls})}$

where $\text{Properties}_{\text{inherited}}\ (\text{SubCls}) = \{\ x \mid x \in \text{Properties}\ (\text{Cls}),\ x \text{ is publicly available to SubCls}\}$,

From (2) and (3), a subclass SubCls is a combination of its inherited properties and its currently defined ones. (3) introduces properties overlapping in the definition when reuse of properties are achieved.



$$\text{Properties}_{\text{redefined}}\ (\text{SubCls}) \subseteq \text{Properties}_{\text{inherited}}\ (\text{SubCls})$$

$$\text{Properties}_{\text{redefined}}\ (\text{SubCls}) = \{\ x \mid x \in \text{Properties}_{\text{inherited}}\ (\text{SubCls}),\ x \text{ is replaced, extended or realised}\ \}$$

Fig. 1: Class properties

$\text{Properties}_{\text{redefined}}(\text{SubCls})$ are the (inherited) redefined properties as opposed to $\text{Properties}_{\text{inherited}}\ (\text{SubCls})$ which is a superset including the ones accessible and used without modification. Because of the variety of possible modifications to a property, for instance, complete redefinition, extension or realisation, this is a possible source of incompatibility between a class and its subclass. As stated by Taivalsaari, inheritance use does not guarantee a conceptual specialisation intention. The mechanism of redefinition has been criticised [2, 12, 18, 26, 29] for not bearing any kind of semantic relationship with its initial implementation, especially when the method is completely overridden. Unfortunately, the *inheritance "scoping" control*[1] facility does not prevent this conceptually inconsistent situation. Indeed, a non-strict *is_a* policy is more likely to introduce unsubstitutable classes and is used either for convenience reasons or because it *uses_a* parent class property. The next section describes the different types of method redefinition and their associated characteristics.

3. Method Redefinition: Uses and Abuses

"Redefinition is an important semantic mechanism for providing the object-oriented brand of polymorphism" -- Meyer [26]. The basic principle of method redefinition is simple: it is a syntactic programming language facility which allows a class C to replace inherited implementations by keeping the same signatures for the new methods. Conceptually, one of the main reasons for using redefinition is to provide the flexibility of defining a different algorithm when the semantics of the method remain the same. Thereby the ability for a method to hold many forms in many subclasses, i.e. achieving polymorphism. At run-time, the correct behaviour will

[1] The process of declaring appropriate modifiers to a class, an attribute or a method will be referred to as the *inheritance scoping control* facility.

then be dynamically bound to the object which receives the message. In the following sections, a description of how incremental development leads to side-effects with the method redefinition problem is given.

3.1. The Method Redefinition Variants

Despite its very important role in a class hierarchy design process, the term redefinition, also known as overriding, is actually used in a confused way. Sometimes, it is referred to in the sense of method extension and other times in the sense of method replacement. Although, in both cases, the method is effectively redefined, their aims diverge completely. Method extension permits the reuse of the inherited property whereas method replacement stops the heritage of a parent property by not using it and replacing completely the inherited implementation with a new one. Method replacement seems intuitively unnatural unless in the case of a polymorphic method. For example, consider the following Smalltalk Collection branch:

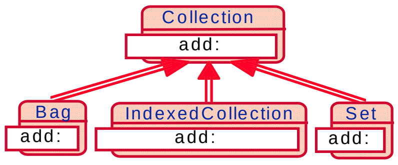


Fig. 2: Part of the Smalltalk Collection branch

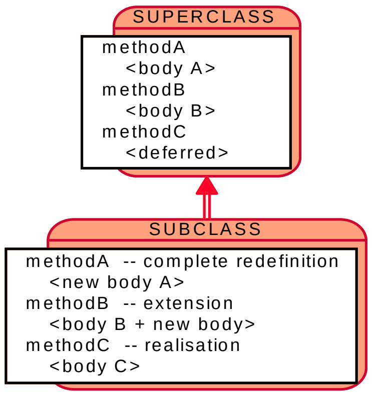


Fig. 3: Method redefinition variants

The add: method of the class Collection is considered abstract (virtual in C++, deferred in Eiffel) indicating that any subclasses must provide the implementation of the method, therefore polymorphic. Firesmith described a set of inheritance guidelines which gives practical advice concerning a class hierarchy design [12]. However, in practice there are no guarantees that a given case of method redefinition is correct. A system can actually work without satisfying the guidelines or essence of inheritance.

In order to assess the "goodness" of a class hierarchy in terms of criteria such as coupling, cohesion, reuse or inheritance, it is important to understand and define what characteristics are to be measured. Our hypothesis is that a high level of redefinition or its variants suggest a possible conceptual design problem in the hierarchy e.g. a class which was wrongly-subclassed. The redefinition of a method will be assessed regarding its main variants [21] described in Fig. 3. The SUPERCLASS's methods are assumed to be publicly inherited. In SUBCLASS, the first case of the redefinition variants depicts an arguable case of inheritance where a complete redefinition of a method is done. Whereas the last two cases: extension and realisation, represent the recommended use of property inheritance. Cancellation of methods is an example of complete redefinition which restricts or stops the inheritance scheme. An extension to the implementation of methodB permits the reuse of inherited code and the addition of extra code which makes the subclass a specialised version. When a method is declared deferred in a parent class, the subclass must provide its implementation, i.e. the method is realised. It should be noted that all cases of inheritance fall under one of the different types of method redefinition mentioned. A method m of class C is redefined if and only if:

- m is an inherited method,
- m(C) signature is the same as in its original definition,
- m(C) implementation is either, replaced, extended or provided.

3.2. The Method Redefinition Problem

Why redefine if inherited? A major criticism of redefinition lies in the essence of inheritance itself. The two notions of property redefinition and property heritage are paradoxical. Surprisingly enough, method redefinition, including correct and incorrect use, happens more often than expected in a class hierarchy. For example, the redefinition metric results for the Smalltalk class library (Fig. 4) show that the amount of redefinition reaches 57.07% at DIT=4 (depth of inheritance metric [9]) in the hierarchy. On the first three levels of the hierarchy, the results obtained more than double from one level to another, denoting high "redefinition activity". One possible reason for such a redefinition profile is due to the incremental development of software. A closer look at the implementation of the same method redefined many times along a branch of the hierarchy revealed that common code had not been factorised. This phenomenon seems typical of the case of many developers working on the same part of a system without modifying the others' code (class dependency problem). Chidamber and Kemerer’s *coupling between objects* (CBO) metric [10] permits the detection of weak and strong coupling. The CBO is recommended to be as low as possible. However, with new design techniques such as design patterns [13], the dependency between classes present in a pattern is high as they are strongly dependent (the purpose of a pattern).

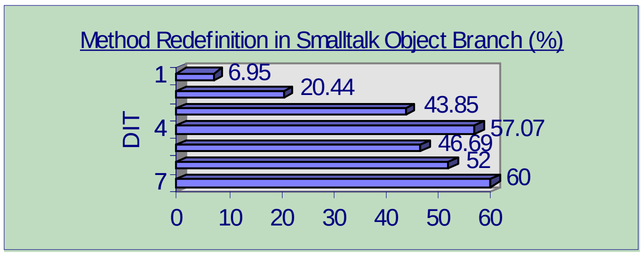


Fig. 4: Smalltalk hierarchy redefinition profile

3.2.1. Multiple Descendant Redefinition (MDR) Problem

The principle of inheritance involves an ownership transfer of features from the parent class to its subclasses. When a class inherits a method which has been publicly defined, the subclass has the right to change the property inheritance scheme for future heirs.

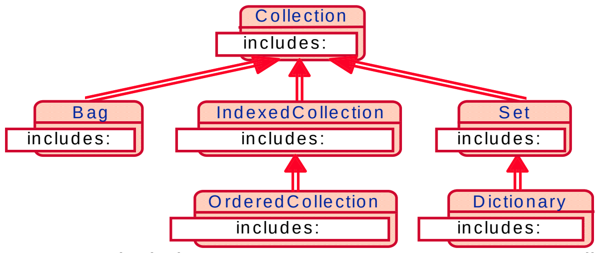


Fig. 5: Life history of the includes: redefined method in the Smalltalk Collection branch

In Fig. 5, the includes: method is used to test if an element is present in a collection. At first sight, a representation of the life history of the <u>completely</u> redefined includes: method casts doubt on the correctness of the design. Although, all IndexedCollection are Collection, they do not test the inclusion of elements in the same manner as IndexedCollection introduces a key for access. The solution is thus to redefine the includes: method to cancel the inherited implementation from the class Collection. Similarly, for OrderedCollection, the same method is completely redefined again. Clearly, the property inheritance scheme is broken and nothing is inherited from the parent class. Furthermore, the includes: method has not been originally declared as deferred and all its subclasses hold completely different forms, an incorrect case of polymorphism by definition. This situation will be referred to as the *multiple descendant redefinition* problem. It should be noted that such classification, although conceptually incorrect can be implemented in any programming language. Further complex method redefinition situations may also arise when a combination of many super calls exists in the same method. However, it is always possible to find an alternative construction to avoid complete method redefinition. For instance, mixin classes [4] are now well established and are a good candidate for solving the problem of redefinition.

3.2.2. Descendant Heritage Extent with and without MDR anomaly

Suppose that a branch of a hierarchy collapses. Instead of having many classes in the branch, an equivalent behavioural construction would be to regroup all the methods from all classes in the branch into a single larger class. This process is known as *flattening* [16]. In the flat class, all methods are unique and for the ones redefined within the branch, only the latest version appears. This method is sometimes convenient for assessing behavioural characteristics of the hierarchy. In Fig. 6 the extent of the expected descendant heritage is modelled for the Child class. When a class inherits properties from its parents, all of them are virtually present in the class plus the delta parts: x and y. In an *is_a* relationship, part of the inherited properties is reused without modification and another part is redefined. The right-hand side of Fig. 6 shows how a subclass' properties may recover the ones from its parents. The recovery part includes all the methods which are redefined in the Child class.

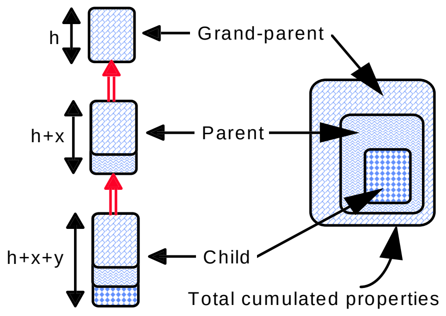


Fig. 6: Expected descendant heritage extent

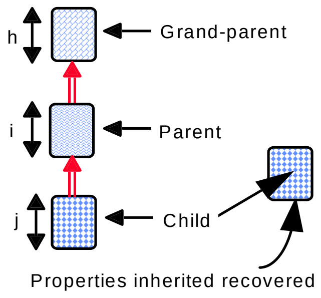


Fig. 7: as Fig. 6 with MDR anomaly

In an extreme situation, suppose that the Parent class completely redefines all the Grand-parent's methods, and the Child class redefines all the Parent's methods, the extent of inherited properties is now completely recovered by the Child class (Fig. 7), therefore no features come from its ancestors although it is a subclass.

An MDR guideline can be formulated as:

> Providing the hypothesis that the multiple descendant redefinition problem breaks the properties inheritance scheme in a class hierarchy, a method m from a class C should not be completely redefined more than twice down a given branch.

In order to detect and thus assess such potential design problems in a class hierarchy, a set of method redefinition metrics is proposed and tested in the following section.

4. A Candidate Method Redefinition Metric Set

The approach taken to define our product metrics was based on the GQM/MEDEA (Goal Question Metric/MEtricDEfinition Approach [5]) approach which provide practical guidelines for building metric sets. Here, we will summarise the steps involved in applying the method.

Step 1: Experimental goal(s)

Object of study: method redefinition mechanism in a class hierarchy
Purpose: detection of MDR anomaly
Quality focus: conceptual design consistency for property heritage
Viewpoint: designer

Step 2: Assumptions

Assumption 1: the deeper a class is in a hierarchy, the more complex it is.
Assumption 2: the deeper a class is in a hierarchy, the more likely the MDR problem arises
Assumption 3: see the MDR guideline formulated in section 3.2.2.

Step 3 and 4: Relevant measurement concept and product abstraction

(see section 6 on "Abstract properties of metrics"). Further work is required for these two steps to formalise the redefinition metric set. However, since the rationale behind our redefinition metrics set is fairly straightforward, emphasis was placed on the fundamental steps 1 and 6.

Step 5: Define metrics (see section 4.1)

Step 6: Experimental validation of the metrics (see section 5)
The proposed set of redefinition metrics are :

- Percentage of redefined methods in a class (PRMC and PRMC').
- Percentage of redefined methods per level within a hierarchy (PRMH) which is decomposed into:
  - ∗ Percentage of completely redefined methods in a class (PCRM).
  - ∗ Percentage of extended methods in a class (PEM).

4.1. Percentage of Redefined Methods per Level Within a Hierarchy (PRMH)

Current metrics assessing inheritance are system or class-level metrics whereas our approach evaluates the amount of redefinition level by level. Providing that a class hierarchy is ideally designed, abstract classes should appear closer to the root of the hierarchy and specialised (or concrete) classes should be situated nearer to the bottom. Our redefinition metric is aimed at depicting such a profile. For instance, $PRMH_1$ metric (Fig. 8: branch A at level 1) measures the shaded classes. The PRMH metric can also be applied at the system level as classes are not necessarily organised in a class hierarchy. For simplicity, we will keep the numbering level absolute in comparison with the root (class Object) level 0. The notation $C_{m,n}$ gives the location of a class C, at rank n, for a given level m in the branch, e.g. class B at level 2 of branch A, is named $B_{2,1}$. The rank is arbitrarily numbered from 0 to n, n $\in$ N, from left to right at the considered level.

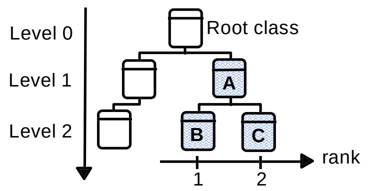


Fig. 8. Complexity metrics at hierarchy level

The redefinition metric for a class and for a given level m are defined as:

$PRMC=\frac{NRM}{NIM}*100$	$PRMC'=\frac{NRM}{NPIM}*100$	$PRMH_m=\frac{\sum_{n=1}^{NC} PRMC_{m,n}}{NC}$	(a)

where NRM is the number of redefined methods, NIM is the number of instance methods, NIM > 0, NC is the number of classes for a given level m, NC > 0,

$PRMC_{m,n}$ is the percentage of redefined methods for all classes $C_{m,n}$. In the current calculation of PRMC (first approach), the equation is a function of the NIM defined locally. However, any class C inherits methods from all its parents, making them potentially available for use (via themethod lookup mechanism). For this reason, the *cumulative redefinition approach* to the same calculation is given by the PRMC' equation (second approach) where NPIM is the number of potential instance methods, NPIM > 0. Indeed, NPIM is expected to increase from top to bottom of a hierarchy, thus, PRMH decreases when DIT increases. Experiments with this metric are detailed in [22].

The PRMH in (a) is general. A refined version includes the redefinition variants:

$$PCRM = \frac{NCRM}{NIM} * 100 \qquad PEM = \frac{NEM}{NIM} * 100 \qquad PRMH_m = \frac{\sum_{n=1}^{NC}(PCRM + PEM)}{NC} \qquad \text{(b)}$$

where NIM > 0, NC > 0, NCRM is the number of completely redefined methods and NEM is the number of extended methods.

Due to the inclusion of the DIT metric within our redefinition metric set, the depiction of *redefinition profiles* of hierarchies is possible.

5. Experiments on the Collection and Stream Branch

Our experiments were done on the Smalltalk Express[2] class library. An "OO system metric browser" tool was implemented in Smalltalk in order to test the proposed metrics. Additional facilities include a repository of metrics results stored as persistent objects and a method profiler for help in the localisation of potential suspect methods. Specification of the prototype metrics tool is described in [22].

The Collection classes in Smalltalk have been well-studied by many researchers [7, 14, 31], particularly those due to conceptual design problems occurring in leaf classes. Cook [7] proposed a complete new re-design of the Collection branch. A major problem concerns the amount of cancellation of property inheritance in leaf classes. Smalltalk's inheritance scoping control permits a class to stop the visibility and accessibility of a method to its subclasses in redefining the method with a body containing the code self shouldNotImplement. This situation is often recognised as source of bad design.

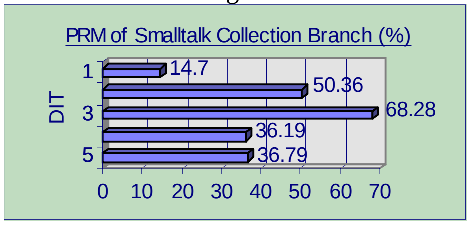


Fig. 9: Collection redefinition profile

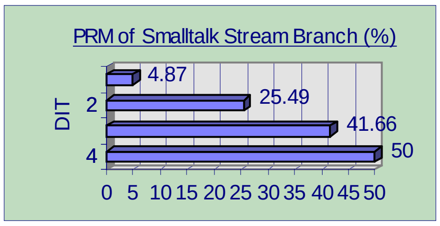
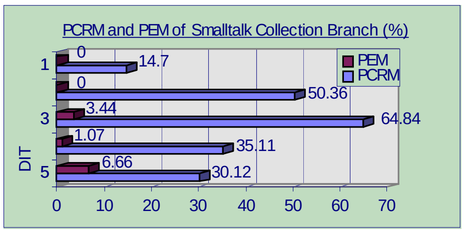


Fig. 10: Stream redefinition profile

[2]In this paper, Smalltalk Express™ designates the version based on Smalltalk/V® Win16 and WindowBuilder® Pro/V provided by ObjectShare®, a Division of ParcPlace, http://www.objectshare.com

Fig. 9 and 10 represent the PRMH for the Collection and Stream branch. At DIT=2, the rate of redefinition is already high with 50.36% (Fig. 9). A simple explanation is that all classes at level 2 have realised the abstract methods which is normal. Supposing that a threshold of 50% of method redefinition should raise an alarm to potential design defects, we would take a closer look at the peak happening at DIT=3 (Fig. 9). A simple way would be to derive the PCRM metric for each class of the concerned level. Clearly, on Fig. 13 the FixedSizeCollection class holds 100% of methods completely redefined, an unusual result in such a hierarchy. Although the percentage of deferred methods is not shown on the figure, the above-mentioned class seems to be wrongly-subclassed. With the help of a method profiler tool [22], it has been possible to study and locate precisely, particular problems in methods of the concerned class.



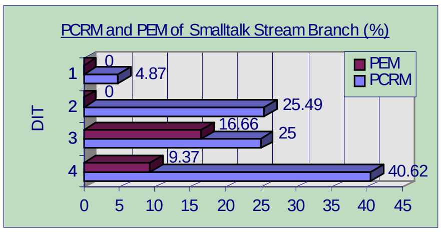
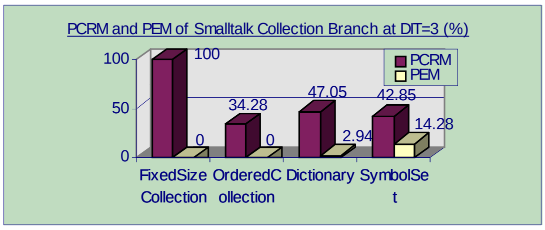


Fig. 11: Detailed Collection profiles

Fig. 12: Detailed Stream profiles

The PCRM for the Stream branch (Fig. 12, 14) is high with 40.62% at DIT=4, which represents a factor increase of 60% from the previous level. This confirms the Smalltalk Stream branch's generally recognised design defect. Due to the single inheritance scheme, the ReadWriteStream class inherits only from the WriteStream class. There is a duplication and redefinition of methods from the ReadStream to WriteStream.



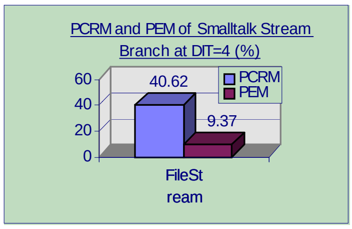


Fig. 13: Collection branch at DIT = 3

Fig. 14: FileStream redefinition profile

6. Discussion

Chidamber and Kemerer [9, 10] proposed a suite of six metrics for assessing the complexity of an OO model, 2 of which are related to the metrics described earlier. The DIT metric is based on the following assumptions:

- a class which is located deep in a hierarchy is more likely to inherit a great number of methods, hence increasing its complexity,
- a deep tree involves greater overall design complexity since the number of classes and methods are important,

- a class which is located deep in a hierarchy benefits from the potential reuse of inherited methods.

Our metrics set adopts these assumptions, however, rather than use DIT as a stand alone metric we have incorporated it into our PRMH metric to give a more meaningful metric. The WMC metric is the weighted method per class which takes into account the static complexity of methods in a class. If the complexity is equal to one, WMC becomes simply the number of methods metric. Churcher and Shepperd [11] showed that the metric was open to many interpretations when considering its use with constructors and destructors in C++. In addition, unlike our PRMH metric it makes no observations as to which methods are inherited and of those inherited, which are redefined and which are not.

Lorenz and Kidd [25] included in their metrics set, the number of methods overridden by a subclass and produced an average extracted from tests on project results. However, unlike our metrics it was done at class level only, no metrics were proposed at hierarchy level and system level. In addition, their metrics are not represented as percentages which clouds interpretation. For example, if number of overriden methods = 5, the class complexity is not the same if the class contains a total of 10 methods (50%) or if the class contains a total of 100 (5%).

The MOOD (Metrics for Object-Oriented Design) set [6] addresses the evaluation of the main keypoints of mechanisms of the OO paradigm. The six metrics are: the method hiding factor (MHF), the attribute hiding factor (AHF), the method inheritance factor (MIF), the attribute inheritance factor (AIF), the polymorphism factor (PF) and the coupling factor (CF). MHF and AHF refer to encapsulation as they detect the amount of hidden attributes and methods. Again, no differentiation is made in the nature of the methods when deriving their metrics for inheritance. Thus, because of the possible existence of completely redefined methods within a class hierarchy, their measure of MIF and PF are affected and does not assess inheritance in such cases.

Lewis [20] proposed a set of fine-grained metrics for assessing overloading, overriding and polymorphism issues. Related metrics are the overridden method references (ORMR), the degree of method overriding (DMOR), the degree of polymorphism (DP) and the degree of obscured polymorphism (DOP). ORMR is applied at method or class level and is taken in the general sense of overriding. ORMR is aimed to be used with DMOR which counts the number of existing forms of a method in the whole application. DP relates to the justified use of method overriding but DOP seems to be language-dependent as it is directed at measuring unspecified polymorphic methods. None of their proposed metrics are considered as ratios and no case studies were presented.

Current research on OO metrics has not yet addressed the multiple descendant redefinition problem. Our proposed metric set was aimed at the assessment of a class hierarchy from a behavioural viewpoint and the detection of abuses of the method redefinition mechanism. The results shown in the experiments revealed that such abuses exist in the current Smalltalk Express hierarchy, however they are theoretically possible in any language. As suggested earlier this may be simply due to the inherent incremental development of a class hierarchy, especially when

different people are involved in the development. It should be emphasised that a system can be in a perfect working state even when containing MDR anomalies. The MDR problem increases the code re-engineering difficulty and affects the natural extension of the inheritance tree which becomes degenerated in presence of MDR.
A limitation of our metrics was that it required support from additional tools in order to precisely pinpoint defects in methods. Our method profiler realised that task by providing a life history of each redefined method of each class along a particular branch of the hierarchy. The analysis of suspect classes were facilitated.

An important area of measurement theory is the interpretation and analysis of metrics results. Most of the current metrics propose thresholds or averages as alarmers for raising potential design flaws in a system. Design decisions can only be suggested in this paper but the data interpretation technique from [23] was used. The MDR problem happens for at least two reasons:

- ➢ a class is wrongly-subclassing its parent class i.e. the class does not satisfy the *is_a* relationship,
- ➢ a bad design of interfaces of parent classes for example, lack of abstraction.

A possible solution for the first reason is to move the suspected class higher in the hierarchy so the class would inherit from early implementation of the method, thereby minimising the chance for the MDR problem. In return, the concerned class will have to resolve all super calls to the original parent. This can be handled by the introduction of the original parent class as an aggregate which is instantiated in a constructor method. The great benefit of this solution is that it can be executed automatically. As opposed to the first solution, the second reason will probably require manual intervention of the designer.
The experimental validation of the metrics confirmed that the metrics measured the desired characteristics. However, concerning some abstract properties of good metrics mentioned by Kolewe [19], further work is ongoing into the development of the necessary theoretical foundations needed. However, we shall briefly comment on the cited characteristics for our redefinition metric set:

- ✓ *noncoarseness*: we considered many different programs and were able to find different metrics results.
- ✓ *nonuniqueness*: if we consider two classes A and B derived from the same parent class where the same modifications on inherited methods are done and no added operations are made, we could be in the case where the PRMC is the same for both classes.
- ✓ *importance of implementation*: we assess a class's internal complexity by looking at its methods redefinition. The metric depends on the implementation.
- ✗ *monotonicity*: not applicable for our metric as its purpose is not to have a general value for the whole system. However, we could compute for two classes A and B their respective PRMC. Assuming that a class C contains all the methods from A and B with no name space conflicts, $PRMH^C = PRMH^A + PRMH^B$. For this characteristic, our redefinition metric can be extended in order to calculate a mean value of redefined methods for a whole system.
- ✗ *nonequivalence of interaction*: same comment as previous characteristic.
- ✓ *interaction increases complexity*: as inheritance is a strong form of coupling and

interaction is implemented via methods in a class, inheriting or adding new methods to a class increases its complexity, therefore the PRMH vary accordingly. Further verification requires to be done.

- *nonequivalence of permutation*: not applicable.

Inheritance in current OO systems is still hazardous. A conceptual gap exists between OO modelling constructs and their mapping onto a language. The implementation of an inheritance relationship between classes using any OO programming language is actually a real source of design problems. In particular, this paper described the problem of multiple descendant redefinition with a refinement of the definition of inheritance. The MDR problem is recognised as a conceptual design inconsistency happening early in the design of the hierarchy. The derivation of our proposed redefinition metric set demonstrated that the knowledge of redefinition profiles of an OO class hierarchy gave us insights into the behavioural aspect. Precise detection of such anomalies have been possible. Similarly, the redefnition metrics can be derived on an OO system not necessarily organised as a hierarchy. We believe that the redefinition metrics and its variants are a strong and simple candidate for detecting complex design problems occurring within a class hierarchy. Further tests and development of its foundations is still necessary together with appropriate guidance for design decisions. Work is continuing in the areas of design transformation rules, (semi) automatic re-organisation of the class hierarchy and the design-evaluation cycle.

Acknowledgment: We would like to thank Mike Jackson (Wolverhampton University) for his comments on early versions of the paper.

References

1. D H. Abbot, T D. Korson and J D. McGregor. A Proposed Design Complexity Metric for Object-Oriented Development. Department of Computer Science, Clemson University, Clemson, SC29634-1906, 1994.
2. J M. Armstrong and R J. Mitchell. Uses and abuses of inheritance. Software Engineering Journal, Jan. 1994.
3. G Booch. Object-oriented analysis and design with applications. Benjamin/Cummings, 1994.
4. G Bracha and W Cook. Mixin-Based Inheritance. OOPSLA/ECOOP '90 Conference proceedings, Canada, 1990.
5. L Briand, S Morasca and V R. Basili. Goal-Driven Definition of Product Metrics Based on Properties. Institute for Advanced Computer Studies, Dpmt. of Computer Science, Univ. of Maryland, Technical Report CS-TR-3346, Sep. 1994.
6. F Brito e Abreu, M Goulão and R Esteves. Towards the Design Quality Evaluation of Object-Oriented Software Systems. Proceedings of the 5th International Conference on Software Quality, Austin, Texas, USA, Oct. 1995.
7. W R. Cook. Interfaces and Specifications for the Smalltalk-80 Collection Classes. OOPSLA '92 Conference proceedings, Vancouver, Canada, Oct. 18-22, ACM SIGPLAN 1992; Not. 27, 10:1-15.
8. L. F. Capretz and P. A. Lee. Object-Oriented Design: Guidelines and Techniques.

Information and Software Technology, Apr. 1993; 35(4):195-206.
9. S R. Chidamber and C F. Kemerer. Towards a Metric Suite for Object-Oriented Design. OOPSLA'91 Conference proceedings, Oct. 1991; pp. 197-211.
10. S R. Chidamber and C F. Kemerer. A Metric Suite for Object Oriented Design. IEEE Transactions on Software Engineering, Jun. 1994; 20(6).
11. N I. Churcher and M J. Shepperd. Comments on A metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, March 1995; 21(3).
12. D Firesmith. Inheritance guidelines. Journal of Object-Oriented Programming, May 1995; pp. 67-72.
13. E Gamma, R Helm, R Johnson J Vlissides. Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley, ISBN 0-201-63361-2, 1995.
14. A Goldberg and D Robson. Smalltalk-80, The Language and its Implementation. Addison-Wesley, ISBN 0-201-11371-6, 1985.
15. R Harrison and R. Nithi. An Empirical Evaluation Of Object-Oriented Design Metrics. OOPSLA '96 Conference proceedings, Workshop on "OO Product Metrics", 1996.
16. B Henderson-Sellers. Object-Oriented Metrics, Measures of Complexity. Prentice Hall Object-Oriented Series, ISBN 0-13-239872-9, 1996.
17. B Henderson-Sellers and Julian Edwards. BookTwo of Object-Oriented Knowledge - The Working Object. Prentice Hall, ISBN 0-13-093980-3, 1994.
18. K Koskimies and J Vihavainen. The problem of Unexpected Subclasses. Journal of Object-Oriented Programming, Oct. 1992; pp. 53-59.
19. R Kolewe. Metrics in Object-Oriented Design and Programming. Software Development, Oct. 1993; 1:53-62.
20. J A. Lewis. Quantified Object-Oriented Development: Conflict and Resolution. 4th Software Quality Conference, University of Abertay, Dundee, Jul. 1995; 1:220-229.
21. S Lewis. The Art and Science of Smalltalk. Prentice Hall/Hewlett-Packard Professional Books, ISBN 0-13-371345-8, 1995.
22. P Li-Thiao-Té. Integrating Measurement Techniques in An Object-Oriented Design Process. Tech. Report, Object Systems Group, Napier University, Edinburgh, 1996.
23. P Li-Thiao-Té, J Kennedy and J Owens. Mechanisms for Data Interpretation of Metrics for OO Systems. To appear in TOOLS Asia ‘97 Conference proceedings, 1997.
24. W Li and S Henry. Object-Oriented Metrics Which Predict Maintainability. Journal of Software Systems, 1993; 23(2):117-122.
25. M Lorenz and J Kidd. Object-Oriented Software Metrics. Prentice Hall Object Oriented Series, Englewood Cliffs (N.J.), 1994.
26. B Meyer. Object-oriented Software Construction. Prentice Hall International, C.A.R. Hoare, Series Editor, ISBN 0-13-629049-3, 1988. http://www.eiffel.com
27. A Newman and al. Special Edition, Using Java. Que Corp., ISBN 0-7897-0604-0, 1996.
28. J Rumbaugh, M Blaha, W Premerlani, F Eddy, and W Lorensen. Object-Oriented Modeling and Design. Prentice-Hall, 1991.
29. J Rumbaugh. A Matter of Intent: How to Define Subclasses. Journal of Object-Oriented Programming, Sept. 1996; pp. 5-9, 18.
30. E Seidewitz, Controlling Inheritance. Journal of Object-Oriented Programming, Jan. 1996; pp. 36-42.
31. A Taivalsaari. On the Notion of Inheritance. ACM Computing Surveys, Sept. 1996; 28(3):439-479.