[JSP, 2005] JavaServer Pages. http://java.sun.com/j2ee/JSP/

[Taglibs, 2005] Tag libraries. http://java.sun.com/products/JSP/taglibraries/

[Struts, 2005] Struts. http://jakarta.apache.org/struts/

[JSF, 2005] JavaServer Faces. http://java.sun.com/j2ee/javaserverfaces/

Authors' Information

Micael Gallego-Carrillo – ESCET, Universidad Rey Juan Carlos, C/Tulipan s/n, 28933 – Móstoles (Madrid), Spain ; e-mail : micael.gallego@urjc.es

Iván García-Alcaide – LPSI, Universidad Politécnica de Madrid, Campus Sur, Carretera de Valencia Km. 7, 28031 Madrid, Spain; e-mail : igarcia@eui.upm.es

Soto Montalvo-Herranz – ESCET, Universidad Rey Juan Carlos, C/Tulipan s/n, 28933 – Móstoles (Madrid), Spain ; e-mail : soto.montalvo@urjc.es

A SENSITIVE METRIC OF CLASS COHESION

Luis Fernández, Rosalía Peña

Abstract: Metrics estimate the quality of different aspects of software. In particular, cohesion indicates how well the parts of a system hold together. A metric to evaluate class cohesion is important in object-oriented programming because it gives an indication of a good design of classes.

There are several proposals of metrics for class cohesion but they have several problems (for instance, low discrimination). In this paper, a new metric to evaluate class cohesion is proposed, called SCOM, which has several relevant features. It has an intuitive and analytical formulation, what is necessary to apply it to large-size software systems. It is normalized to produce values in the range [0..1], thus yielding meaningful values. It is also more sensitive than those previously reported in the literature. The attributes and methods used to evaluate SCOM are unambiguously stated. SCOM has an analytical threshold, which is a very useful but rare feature in software metrics. We assess the metric with several sample cases, showing that it gives more sensitive values than other well know cohesion metrics.

Keywords: Object-Oriented Programming, Metrics/Measurement, Quality analysis and Evaluation.

ACM Classification Keywords: D.1.5 Object-oriented Programming; D.2.8 Metrics

Introduction

The capacity to measure a process facilitates its improvement. Software metrics have become essential in software engineering for quality assessment and improvement. Class cohesion is a measure of consistency in the functionality of object-oriented programs. High cohesion implies separation of responsibilities, components' independence and less complexity. Therefore, it augments understandability, effectiveness and adaptability. Actually, these are major factors of the great interest in using object-oriented programming in software engineering.

"Current cohesion metrics are too coarse criteria that should be complemented with finer-grained factors (..). Then it will be easier to assess the trade off involved in any design activity, which would make it possible to see whether a system is evolving in the right direction [Mens, 2002].

One software metric must be simple, precise, general and computable in order to be applicable to large-size software systems. Automated metric producing sensitive values of class cohesion can be of great value to most designers, developers, managers, and of course to beginners, in identifying the class cohesion in an object-oriented design.

This paper presents, as follows, a metric of class cohesion that has several advantages over previous cohesion metric proposals. The next section presents a brief summary of the state or art about proposed metrics of class cohesion. In the third section, we formulate the new metric. The existence of thresholds is studied in fourth section, where two representative values are analytically determined. This section also provides guidelines to improve a class definition when its cohesion value is below the threshold. In fifth section, the method and attributes to be considered on the evaluation of a class are discussed. The sixth section assesses qualitative and quantitatively the desirable features of our cohesion metrics. Particularly interesting is a comparative analysis with previous cohesion metrics, based on a sample of classes with different cohesion features. Finally, we summarize our conclusions.

Related Work

In a cohesive class, all the methods collaborate to provide the class services and to manage the object state; in others words methods work intensively with the attributes of the class. A metric for class cohesion was first proposed by Chidamber and Kemerer in 1991 and then revised in 1994 [Chidamber, 1994]. Then, it has been repeatedly reinterpreted and improved [Li, 1993][Li, 1995][Hitz, 1996]. Properly speaking, these proposals do not evaluate cohesion but lack of cohesion (hence it name, LCOM) in terms of the number of pairs of the class methods that use disjoints sets of attributes.

There are two problems with Chidamber and Kemerer's expression [Chidamber, 1994]:

i. the metric has no upper limit, so it is not easy to grasp the meaning of the computed value, and
ii. there are a large number of dissimilar examples, all of them giving the same value (LCOM=0) but not necessarily indicating the same cohesion.

In spite of its pitfalls, LCOM still is the most widely used metric for cohesion evaluation [Bansiya, 1999]. Li et al. [Li, 1995] estimate the number of non-cohesive classes that should be redesigned by computing the number of subsets of methods that use disjoint subsets of attributes. We call this value the number of clusters of the class. Etzkorn et al. [Etzkorn, 1998] study which formulation provides the best interpretation of lack of class cohesion.

Henderson-Sellers [Henderson-Sellers, 1996] proposed a completely new expression for cohesion evaluation, called LCOM*:

$$LCOM^{*} = \frac{\left[\frac{1}{a}\sum_{j=1}^{a}\mu(A_j)\right] - m}{1-m} \tag{1}$$

where a and m are the number of attributes and methods of the class, respectively, and $\mu(A_j)$ is the number of methods that access the datum A_j ($1 \le j \le a$). Notice that this expression is normalized to the range [0..1]. He claims that this metric has the advantage of being easier to compute than previous ones.

A problem with LCOM* is that it is not clear how it can give any account of class cohesion or inter-method cohesion (relation between methods). In effect, the summatory in the numerator just counts how many times all the attributes are accessed, independently of which method accesses each datum. This problem is illustrated in Table 1, with a comparison of several metrics, including ours. The table includes a representation of classes in

the second column. As well, classes are represented by another table in which rows can be found the attributes and in the columns the methods accessed by them. So, a filled $cell_{i,j}$ means that the i-th method uses the j-th attribute. For example, class C has 6 attributes and 6 methods, and its method M_4 uses attributes A_1, A_2, A_3 and A_4.

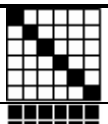
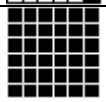
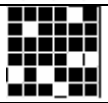
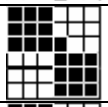
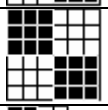
In particular, rows F, G and H show that whereas class F seems to be a good candidate to be split into two classes, it is not clear whether a more cohesive design can be found for classes G or H. However, the three classes are evaluated by LCOM* with the same value, it seems to be clear that it does not give a good account of class cohesion at least with these two examples.

Metrics measuring lack of cohesion, as proposed in [Chidamber, 1994][Li, 1993][Li, 1995][Hitz, 1996], evaluate the number of disjoint pairs of methods. From the opposite point of view, cohesion is related to the overlapping of these sets. It would be desirable to measure not only whether two methods are disjoint, but also, how big the intersection set is. Let us consider that two methods are more cohesive if they co-use more attributes. Consequently, cohesion is directly proportional to the cardinality of intersection for all the possible pairs of methods. [Park, 1998] suggested an evaluation of Class Cohesion (CC) as the sum of the connection intensity between all the possible pairs of methods. Let us denote I_k for the subset of attributes used by a given method M_k. The connection intensity of a pair of methods is defined as:

$$C_{i,j} = \frac{Card(I_i \bigcap I_j)}{a} \tag{2}$$

where *a* is the total number of instance attributes in the class. CC is more intuitive than LCOM*, but it also has deficiencies, as is shown in Table 1, where it also computes the same value for F, G, H classes.

Table 1. Values of different cohesion metrics on 8 sample classes.

CLASS		*LACK of COHESION*			*COHESION*	
Name	*Methods/Attribute Table*	*Chidamber (1994)*	*Li (1995)*	*Henderson (1996)*	*Park (1998)*	*SCOM*
A		15	6	1	0	0
B		0	1	0	1	1
C		0	1	0.23	0.62	0.82
D		0	1	0.57	0.23	0.29
E		3	2	0.60	0.20	0.20
F		2	2	0.67	0.17	0.17
G		0	1	0.67	0.17	0.25
H		0	1	0.67	0.17	0.36

Sensitive Class Cohesion Metric (SCOM)

In this section, we introduce a new cohesion metric that we call SCOM (standing for Sensitive Class Cohesion Metric). At this stage, let $\{A_1,\ldots,A_a\}$ and $\{M_1\ldots M_m\}$ be the set of attributes and methods of the class, respectively, to be considered.

Firstly, let us consider connection intensity of a pair of methods (see formula 2 above). It would be more accurate if it were divided by the maximum possible value, rather than by the total number of attributes of the class, as proposed by Park in formula 2. As the largest possible cardinality of the intersection of two sets is the minimum of their cardinalities, the connection intensity formula becomes:

$$C_{i,j} = \begin{cases} 0, & if\ I_i \bigcap I_j = \phi \\ \dfrac{card(I_i \bigcap I_j)}{\min(card(I_i), card(I_j))} & otherwise \end{cases}$$

Connection intensity of a pair of methods must be given more weight when such a pair involves more attributes. Of course, the largest contribution is found when every attribute in the class is used by any of the two methods. The weight factor is:

$$\alpha_{i,j} = \frac{card(I_i \bigcup I_j)}{a}$$

Finally, the formula of our cohesion metric results:

$$SCOM = \frac{2}{m(m-1)} \sum_{i=1}^{m-1} \sum_{j=i+1}^{m} C_{i,j} * \alpha_{i,j} \qquad (3)$$

where the coefficient of the summatory is the inverse of the total number of method pairs: $\binom{m}{2} = \frac{m(m-1)}{2}$.

As a consequence, the metric is normalized to the range [0..1], where the two extreme values are:

i. Value zero: there is no cohesion at all, i.e. every method deals with an independent set of attributes.

ii. Value one: full cohesion, i.e. every method uses all the attributes of the class.

Thresholds

The threshold or "alarm value" minimum (respectively, maximum) of a metric is a value that indicates a design problem for a software entity whose evaluation lies below (or over) it. The threshold calls for the developer's attention to focus on a particular module or chunk for further evaluation [Lorenz, 1994].

As we mentioned in the previous section, the SCOM cohesion metric is ranged between zero (representing no cohesion at all) and one (representing full cohesion). Although an empirical threshold must be determined with real projects, we consider that a theoretical study about singular points gives information about how to interpret the values the metric delivers. In the next two subsections, we present the analytical expressions for the minimal value of a class that has one cluster and the maximum value of a class that has at least two clusters. The third subsection deals with the influence of these values on the SCOM threshold and its applicability for class evaluation.

Minimal cohesion value for one cluster.

Let us consider a class with *m* methods and *a* attributes defined so that it is impossible to find two clusters.

We start with *m=2* and then we include one by one new methods to the class satisfying the former condition. The lower contribution to cohesion is given by a method with just one attribute. The connection intensity $C_{i,j}$ of a pair of methods with only one attribute has the denominator equal to one (see formula 3). Counting the number of non-null terms, we induce the total number of pairs of the class with *m* methods and *a* attributes.

Table 2 shows the minimum number of pairs of non-null methods for the values of *a=3* and *a=4* and consecutive values of *m* starting from *m=2*. These minimums are expressed in terms of *a*'s and *m*'s. From this table, the general expression establishing the number of terms that contributes to cohesion in a significant manner can be deduced by induction.

Table 2. Number of pairs non-null methods.

	a=3		a=4	
m	S	S(a,m)	S	S(a,m)
2	1	a·0+1	1	a(0)+1
3	2	a·0+2	2	a(0)+2
4	3	a·(0+1)	3	a(0)+3
5	5	a(0+1)+2	4	a(0+1)
6	7	a(0+1)+4	6	a(0+1)+2
7	9	a(0+1+2)	8	a(0+1)+4
8	12	a(0+1+2)+3	10	a(0+1)+6
9	15	a(0+1+2)+6	12	a(0+1+2)
10	18	a(0+1+2+3)	15	a(0+1+2)+3
…	…	…	…	…

From these data, it can be induced that the number of terms that contributes in a significant manner to cohesion, named *S*, has a linear behavior. Hence, *S* can be expressed by $S = a \cdot t + r$, where:

$$t = \frac{1}{2} INT(\frac{m-1}{a}) \cdot \left[INT(\frac{m-1}{a}) + 1 \right]$$

and

$$r = \left[1 + INT(\frac{m-1}{a}) \right] \cdot Mod(\frac{m-1}{a})$$

hence,

$$S = \frac{a}{2} INT(\frac{m-1}{a}) \cdot \left[INT(\frac{m-1}{a}) + 1 \right] + \left[1 + INT(\frac{m-1}{a}) \right] \cdot Mod(\frac{m-1}{a})$$

and extracting common factors:

$$S(m,a) = \frac{1}{2} \left[1 + \text{int}(\frac{m-1}{a}) \right] \left[\text{mod}(\frac{m-1}{a}) + m - 1 \right]$$

The minimum weight factor αi,j occurs when both methods deal with the same attribute being 1/a, but some coefficients will actually be higher. Therefore:

$$SCOM\min > \frac{2}{m(m-1)a} S(m,a) = SCOM\min K \quad (4)$$

We call $SCOM_{minK}$ to the minimal known value. A class with SCOM < $SCOM_{minK}$ does not satisfy the induction premise. Consequently, we can claim that it has at least two clusters and it must be subdivided into smaller, more cohesive classes.

Maximum cohesion value for two clusters.

From the opposite point of view, it is possible to evaluate the maximum value of SCOM in presence of two clusters. In this situation there are at least *m-1* null terms in (3); in other words, there are at most *(m-1)·(m-2)/2* non-null terms. The biggest $C_{i,j}$ is valued one. The biggest weight factor occurs when one cluster has *a-1* attributes and the other has just one attribute and all the methods in each cluster involve all the attributes in such a cluster. In this situation, $\alpha_{i,j}=(a-1)/a$.

Finally, the maximum value of SCOM for a class with m methods arranged in two clusters is:

$$SCOM2\max = \frac{(a-1)(m-2)}{ma} \tag{5}$$

Threshold applicability.

As we explained above, the equation (4) provides the minimum value below which certainly there are two subsets using non-overlapping attributes. This class can be automatically split into two (or more) smaller and simpler classes.

The metrics for class size by Lorenz and Kidd [Lorenz, 1994] suggest a threshold of 20 instance methods and 4 class methods, being 12 the number average of methods per class in a project. They also proposed 3 as a threshold both for instance attributes and class attributes in a class, and 0, 1 the class attributes average in a project. With these particular values (*a=3* and *m=12*), the minimal cohesion is $SCOM_{minK}$=0.13.

Moreover, SCOM2max provides the analytical value that guarantees the existence of one cluster. For *a=3* and *m=12*, $SCOM_{2max}$=0.56. Figure 1 places both values on the cohesion range.

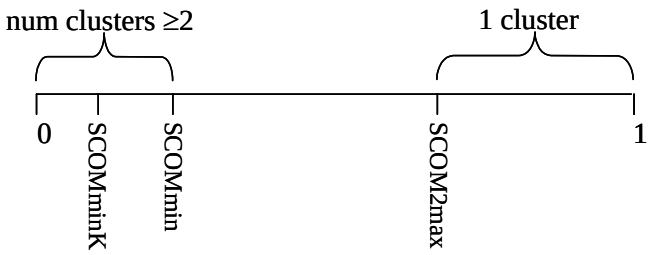


Figure 1. Significant values on cohesion range.

The presence of two clusters clearly indicates that the class may be split. Not all the classes with more than one cluster can be detected by $SCOM_{minK}$. Li formulation could help but, even in presence of just one cluster, the class design can be low cohesive and SCOM is giving account of such a situation.

The threshold considered in software evaluation should be between $SCOM_{minK}$ and 1. It looks feasible to demand guarantee of just one cluster, then SCOM will actually be between $SCOM_{2max}$ and 1. The adequate value, inside these limits, should be established by studying real projects.

Quoting Lorenz and Kidd [Lorenz], "metrics are guidelines of the quality of the design and not absolute laws of nature". Thus, when the alarm is raised, the corresponding software entity is suspicious to have design problems, so it must be re-analyzed. However, thresholds are affected by many factors (as pointed out in the next section), so the warning may sometimes be disregarded.

Criteria of Application

The features of object-oriented programming do not always make obvious what methods and attributes to consider for estimating cohesion. Moreover, different authors use different criteria and even they say nothing about what to do with them. This un-definition or not precise definition in the way of evaluating cohesion makes very difficult to interpret the obtained values from the metrics. We discuss here the most relevant aspects.

i. Transitive closure of attributes. Whenever M_i invokes method M_j, M_i transitively uses each attribute being used by M_j. Consequently, these attributes must be taken into account for the metric evaluation. The same reason is argued to consider objects that are exported to clients which map a subsequent client request (method invocation sending the argument "this"), as it happens in the double dispatching and visitor patterns. Attributes involved in subsequent request must be counted for the exporter method.

ii. Class and instance methods and attributes. Class cohesion involves both instance and class methods and attributes. It is common that class attributes are less involved in methods than instance attributes, which would decrease inadequately the class cohesion. These attributes could be removed to evaluate cohesion of a single class. However, the rate between class and instance attributes use to be too small in the average to affect evaluation substantially, so they can either be removed or considered.

iii. Constructor and destructor methods. Etzkorn *et al*. [Etzkorn, 1998] suggested to exclude constructor methods but to keep the destructor method for LCOM calculations. The argument was that these particular functions often access most or all of the variables of a class. As LCOM looks for methods with non-overlapping attributes, a method using all the attributes lowers its capability of discrimination. However, we think that both constructor and destructor methods must be considered. For instance, a constructor that initializes all the attributes (as probably happens in the Car class of a concessionaire) contributes to a larger cohesion value than a constructor that just initializes some attributes (as will likely happen in a Student class, where the name will be required from the very beginning, but the student final evaluation will not).

iv. Access methods. The *get* and *put* private methods make easier future maintenance. They typically cope with just one attribute. However, the resulting cohesion value should not be negatively affected by a good programming style. Therefore, we suggest excluding them.

v. Methods overload. It is common in object-oriented programming to have a given method called with a concrete value. Some programming languages (such as C++) allow parameter initialization in absence (for example, an interval class being displaced to the coordinate origin). When a language lacks this feature (for instance, in Java or Smalltalk), the method is overloaded by defining a new method without this parameter declaration, where the new method's implementation just consists in the invocation of the former with a constant value. Both methods cope with the same functionality and in this particular case, only the general one must be considered for cohesion evaluation.

vi. Inheritance in cohesion evaluation. The question of whether inherited variables must be considered for cohesion evaluation was posed by Li *et al*. [Li, 1995]. They only considered local attributes. However, in [Etzkorn, 1998] argues in favor of the consideration of inherited attributes.

The state of one instance is defined by its own and parent attributes, so both of them must be considered. Thus, we propose to consider the parent's methods and attributes in the parent class cohesion evaluation, and the parent's and its own attributes, but just its own methods being involved on subclass cohesion evaluation. In some types of inheritance, the subclass methods seldom use the inherited attributes; in such a situation, the metric value decreases.

vii. Methods dealing with no attributes. We find this situation in abstract classes and sometimes in static methods. On the one hand, abstract methods must not be considered for cohesion evaluation of the abstract class. They must be considered for the subclass where they are actually implemented. On the other hand, static methods not dealing with attributes (as happens in the factory pattern) are not considered for cohesion. It is expected that instance methods manage at least one attribute; otherwise, a different metric from cohesion must be used and it must raise the appropriate alarm. Therefore, methods, which deal with no attributes, must not be considered.

viii. Attributes not used by any method. This is an undesirable situation in absence of inheritance. Actually, some compilers (e.g. Fortran90 or Java) warn about it. A different metric will deal with it. In presence of inheritance, the parent class sometimes references an attribute to be managed by its subclasses. These attributes must not be considered for superclass evaluation, but they must be considered for subclass evaluation.

Metric Assessment

In this section, we assess our SCOM metric. Firstly, we make an informal assessment. Then, we make an empirical comparison with other metrics.

Qualitative analysis of cohesion metrics.

The desirable properties (or metametrics) of a software metric have been proposed elsewhere [Henderson-Seller, 1996][Xenos, 2000]. We assess SCOM informally with respect to such properties:

i. Simplicity (the definition of the metric is easily understood). SCOM has a simple definition.
ii. Independence (the metric does not depend on its interpretation). SCOM is independent because it has a clear definition based on syntactic information (attributes and methods).
iii. Accuracy (the metric is related to the characteristic to be measured). SCOM is accurate since it is based on cohesion features, namely dependencies between attributes and methods.
iv. Automation (effort required to automate the metric). SCOM has an analytical definition that depends on the attributes used by the methods in a class and their transitive closure. This information can be automatically extracted from the class implementation.
v. Value of implementation (the metric may be measured in early stages or it evaluates the success of the implementation). SCOM measures the class implementation.
vi. Sensitivity (the metric provides different values for non-equivalent elements). SCOM is sensitive for different cases, as it is shown in the next subsection.
vii. Monotonicity (the value computed for a unity is equal or greater/less than the sum of the values of its components). SCOM is decreasingly monotonous.
viii. Measurement scale (rate/absolute/nominal/ordinal, being more useful rate and absolute scales than nominal and ordinal ones). SCOM gives a normalized rate scale.
ix. Boundary marks (upper and/or lower bounds). SCOM has bounds on both sides of the scale since it ranges between 0 and 1.
x. Prescription (the metric describes how to improve the entity measured). SCOM establishes how to improve the class as we describe below.

There are two typical cases where some restructuring actions can be performed on a class. Firstly, when two (or more) non-overlapping clusters are detected in a class, it must be split into two (or more) classes. The methods using a subset of non-overlapping attributes must be grouped, taking care to preserve the appropriate granularity (size) of the class.

Secondly, when the threshold alarm is raised but isolated clusters are not detected, the class is a candidate to be re-analyzed. The programmer must inquire whether any attribute can be moved to another class, look for any undiscovered class, or check whether all the class responsibilities are necessary (or any can be moved to a different class). By doing this clean up, some methods may go away or may be shortened. As a consequence, the comprehensibility of the application and the quality of the class design are improved, while code length actually decreases.

Quantitative analysis of cohesion metrics.

We have applied different cohesion metrics, including SCOM, to several sample classes in order to empirically comparing their results. Table 1 illustrated such results for eight sample classes.

Notice that while Chidamber, Li-Henry and Henderson evaluate lack of cohesion, Park and SCOM evaluate cohesion, so their extreme values have the opposite interpretation.

Class A has no cohesion at all while class B has the highest cohesion, so they obtain the limit values for metrics with bounds (Henderson, Park and SCOM).

Classes B, C and D represent very different situations. Chidamber and Li-Henry metrics show low sensibility, rating the same value (zero) for these classes, whereas the other metrics give an account of their differences.
Classes E and F are very similar: they share the property of having two clusters, being obvious that they could be split into two. Chidamber's metric yields different values for these classes, showing that the meaning of non-zero values in this formulation are difficult to understand because it does not have an upper bound. Henderson's metric also exhibits low discrimination because the lack of cohesion measure obtained is not as high as we could expect.
Classes F, G and H also represent different situations. Henderson and Park cohesion metrics yield the same value, showing lower discrimination capability than expected. However, SCOM gives account of their differences, yielding the lowest value for the obviously worst design (F).
SCOM is more sensitive than Park's formulation, as can be seen comparing the cases D and E. Moreover, Park's formulation seems to be too restrictive for the cohesion's upper extreme, as it is shown in class C. This class has a good cohesive appearance, so a value higher than 0.62 is intuitively expected for a magnitude ranging between 0 and 1. SCOM behaves better, yielding the value 0.82.
Remember that our metric also allows computing the analytical values $SCOM_{minK}$ and $SCOM_{2max}$ that give information about the cohesion for a particular class, without any previous human analysis of its table structure. For a class with 4 attributes and 4 methods, $SCOM_{minK}$=0.13 and $SCOM_{2max}$=0.38. It can be seen that classes F, G and H are low cohesive; in particular, F should be split. For a class with 6 attributes and 6 methods, $SCOM_{2max}$=0.56. The classes D and E have values far below $SCOM_{2max}$ (0.29 and 0.20, respectively), indicating design problems. The value 0.82 obtained for class C suggests a good cohesive design, as expected analyzing its methods/attributes table.

Conclusion

This paper proposes a metric to evaluate class cohesion, that we have called SCOM, having has several relevant features. It has an intuitive and analytical formulation. It is normalized to produce values in the range [0..1]. It is also more sensitive than those previously reported in the literature. It has an analytical threshold, which is a very useful but rare feature in software metrics. The attributes and methods used for SCOM computation are also unambiguously stated.

The metric is simple, precise, general and amenable to be automated, which are important properties to be applicable to large-size software systems. By following the prescription provided by the metric, the understanding and design quality of the application can be improved.

Bibliography

[Mens, 2002] T.Mens and S.Demeyer. Future trends in software evolution metrics. In: Proc. IWPSE2001, ACM, 2002, pp. 83-86.

[Chidamber, 1994] S.R.Chidamber, and C.F.Kemerer. A Metrics Suite for Object Oriented Design. In: IEEE Transactions on Software Engineering, 1994, 20(6), pp. 476-493.

[Li, 1993] W.Li, and S.M.Henry. Maintenance metrics for the object oriented paradigm. In: Proc. 1st International Software Metrics Symposium, Baltimore, MD: IEEE Computer Society, 1993. pp. 52-60.

[Li, 1995] W.Li, S.M.Henry et al. Measuring object-oriented design. In: Journal of Object-Oriented Programming, July/August, 1995, pp.48-55.

[Hitz, 1996] M.Hitz, and B.Montazeri. Chidamber & Kemerer's metrics suite: a measurement theory perspective. In: IEEE Transactions on Software Engineering, 1996, 22(4), pp. 267-271.

[Bansiya, 1999] J.Bansiya, L.Etzkorn, C.Davis, and W.Li. A Class Cohesion Metric For Object-Oriented Designs. In: JOOP, 1999, 11(8), pp. 47-52.

[Etzkorn, 1998] L.Etzkorn, C.Davis and W.Li. A practical look at the lack of cohesion in methods metric. In: JOOP, 1998, 11(5), pp. 27-34.

[Henderson-Sellers, 1996] B.Henderson-Sellers. Object-Oriented Metrics: Measures of Complexity. In: New Jersey, Prentice-Hall, 1996, pp. 142-147.

[Park, 1998] S.Park, E.S.Hong et al. Metrics measuring cohesion and coupling in object-oriented programs. In: Journal of KISS, 1998, 25(12), pp. 1779-87.

[Lorenz, 1994] M.Lorenz, and J.Kidd. Object-Oriented Software Metrics: A Practical Guide. In: New Jersey, Prentice Hall, 1994, p. 73.

[Xenos, 2000] M.Xenos, D.Stavrinoudis, K.Zikouli, and D.Christodoulakis. Object-Oriented Metrics: A Survey. In: Proc. FESMA, Madrid, 2000, pp. 1-10.

Authors' Information

Luis Fernández – UPM, Universidad Politécnica de Madrid; Ctra Valencia km 7, Madrid-28071, España; e-mail: setillo@eui.upm.es

Rosalía Peña – UA, Universidad de Alcalá; Ctra Madrid/Barcelona km 33, Alcalá-28871, España; e-mail: rpr@uah.es

HANDLING THE SUBCLASSING ANOMALY WITH OBJECT TEAMS

Jeff Furlong, Atanas Radenski

Abstract: Java software or libraries can evolve via subclassing. Unfortunately, subclassing may not properly support code adaptation when there are dependencies between classes. More precisely, subclassing in collections of related classes may require reimplementation of otherwise valid classes. This problem is defined as the subclassing anomaly, which is an issue when software evolution or code reuse is a goal of the programmer who is using existing classes. Object Teams offers an implicit fix to this problem and is largely compatible with the existing JVMs. In this paper, we evaluate how well Object Teams succeeds in providing a solution for a complex, real world project. Our results indicate that while Object Teams is a suitable solution for simple examples, it does not meet the requirements for large scale projects. The reasons why Object Teams fails in certain usages may prove useful to those who create linguistic modifications in languages or those who seek new methods for code adaptation.

Keywords: Languages; Code reuse; Subclassing

ACM Classification Keywords: D.3.2 [Programming Languages]: Language Classifications – Extensible languages, object-oriented languages; D.3.3 [Programming Languages]: Language Constructs and Features – Classes and objects, data types and structures, inheritance, polymorphism

1 Introduction

As the requirements for an object-oriented application evolve, so do the applications themselves. For example, an application or even a programming language may need to be enhanced with new functionalities or new linguistic features. Consequently, the set of Java classes that define the application or the set of classes that define a compiler may need to be adequately adapted for such changes [10].

Subclassing is the principle object-oriented programming language feature that provides code adaptation. Patterns may provide an alternative solution, but we are interested in a direct linguistic primitive. Subclassing