ELSEVIER

The Journal of Systems and Software 58 (2001) 261–269

**The Journal of
Systems and
Software**

www.elsevier.com/locate/jss

The effects of design pattern application on metric scores

Brian Huston *

Systems Engineering Faculty, Southampton Institute, East Park Terrace, Southampton, UK

Received 15 April 2000; received in revised form 29 August 2000; accepted 29 September 2000

Abstract

One method suggested for improving software quality has been that of collecting metric scores for a given design, and refactoring in response to what are deemed to be unsatisfactory metric values. More recently, the usage of design patterns has been recommended to promote adaptable designs, so reducing maintenance effort. These two approaches are therefore observed to effectively have the same general aim. The question then arises as to whether design metrics and design patterns are always compatible, and where this is not found to be the case whether the metric, the pattern or both are anomalous. Methods of analysis are presented which demonstrate the effects of applying various patterns on certain metric scores, the initial conclusion being that the two approaches are indeed mainly congruent. © 2001 Elsevier Science Inc. All rights reserved.

Keywords: Software quality; Object-orientation; Design metrics; Design patterns; Refactoring

1. Introduction

The aim of any given suite of design metrics is to provide objective feedback as to the quality of the design being measured, suggesting aspects of object-orientation which may not have been applied appropriately. There may perhaps be a general lack of subclassing or polymorphism, a high level of unnecessary coupling, or an uneven spread of behaviour amongst the classes. This highlighting of the ‘weak’ areas within a given design then implies a course of remedial action (generally a refactoring), so enabling software quality to be improved (in terms of characteristics such as reduced maintenance time or increased class reusability). The earlier in the lifecycle that a metric is available, the more useful it will be in terms of avoiding backtracking. Conversely, an early metric will be based on low information and it is therefore more ambitious to say that it is predictive of some finished system property. In addition, the granularity of metric available is clearly a function of lifecycle stage and tradeoffs may be required between reducing design and reducing implementation complexity (Tegarden et al., 1995).

This goal of producing a definitive limited set of perhaps a dozen orthogonal measures, which are gen-

erally predictive in all domains and available at a suitably early stage of the project lifecycle, has prompted much in the way of suggested measures. This is evident from the considerable number of metrics relevant to object-oriented design which have been proposed in the literature (Chidamber and Kemerer, 1994; Henderson-Sellers, 1996; Lorenz and Kidd, 1994). Such measures are often initially conceived simply on the basis of intuitive appeal. A process of refinement is then applied which may include adjustment to a ratio scale (usually the interval [0,1]), and checks for compliance with a chosen set of measurement axioms (Hitz and Montazeri, 1996; Weyuker, 1988; Shepperd and Ince, 1993). These actions serve as an initial filter on the proposed metric, prompting adjustments in order to remove some of its more obviously undesirable properties. However, proof of its actual predictive power has been dependent on correlation with finished system characteristics such as defect rate or percentage of classes reused. Given the profusion of possible systems, and the use of different implementation languages, empirical studies must be seen to represent limited sampling and taken as a whole seem to produce an excess of (sometimes conflicting) correlations (Basili et al., 1996; Binkley and Schach, 1996; Henry et al., 1995).

Other factors also confound metric application. The ambiguous definition of a metric will cause problems as its value will vary according to interpretation (Churcher and Shepperd, 1995; Shepperd and Ince, 1994).

* Present address: 21 Foxley Drive, Portsmouth PO3 5TG, UK.
E-mail address: brian.huston@talk21.com (B. Huston).

Furthermore, when presented with a set of class metric scores $M(i)$ for a set of classes $C(i)$, it is often unclear as to whether the average, minimum or maximum or indeed the standard deviation of $M(i)$ should be limited (rule of application). Finally, concepts such as lack of cohesion have a number of competing metrics which could be used to quantify them (Bansiya et al., 1999; Briand et al., 1997; Li, 1998). The aim is therefore to demonstrate an approach which appears promising in terms of producing a more direct assessment of metric performance, and recommendations for metric selection and rule of application. The method adopted is based upon an analysis of design metric interaction with well-established design pattern structures.

2. Design patterns and metrics

It is seen that the application of both design metrics and design patterns is geared to a common purpose, namely the elimination of bad design practices. Metrics have generally operated at a higher level, inasmuch as they are usually calculated for an entire system or subsystem in order to provide general recommendations for improvement. A metric score for depth in inheritance tree, for example, may serve to highlight that there has been an excessive abstraction of classes. Design patterns operate at a more local level, and do not merely imply what should change but also how. They indicate subsets of the design which are candidates for restructuring, and how this restructuring should be carried out.

The application of design patterns in the construction of software is most notably described by Gamma et al. (1995). Design patterns represent ‘descriptions of communicating objects and classes that are customised to solve a general design problem in a particular context’. As such, they are deemed to provide a suitable test population for evaluating (syntactic) metric behaviour. If established design patterns are accepted as being effective building blocks for design, then their usage should be compatible with the metric suite being applied. There is clearly a problem if refactorings to reduce metric scores destroy pattern structures, and vice versa. In this case the metric, the pattern, or indeed both may prove to be suspect (Ververs and Pronk, 1995).

The patterns discussed by Gamma are grouped into three categories, these being creational, structural and behavioural. Design metrics are largely concerned with static class structures rather than dynamic objects, and so the structural and behavioural categories are expected to be the most fruitful in terms of providing patterns for analysis. It is noted that the availability of a design pattern usually implies the existence of a corresponding ‘non-pattern’ solution, in other words the typical approach taken were the pattern not to be adopted (as opposed to an ‘antipattern’, Brown et al., 1998). These

non-patterns (or ‘default’ structures) will also be of use, in providing a comparison of the metric scores returned for them against those calculated for the pattern. Such an approach mirrors that adopted by Henderson-Sellers et al. (1996) in evaluating their proposed lack of cohesion in methods (LCOM) metric, in that the metric’s scores are judged for aptness against a number of structural alternatives.

3. Metrics for class coupling

The specific category of metrics to be examined initially are those related to class coupling. The term coupling has been used to denote both peer-to-peer and inheritance-based coupling (Schach, 1996); the phrase ‘internal coupling’ has also been used to describe the intra module method-attribute accessing more commonly referred to as cohesion. The form of coupling to be considered here is peer-to-peer, and may be defined to exist where a given class requires knowledge of another (in terms of making use of its services or accessing its state).

It is clear that there must be some communication between objects in order for the required overall behaviour of a system or framework to be implemented. Coupling reduction is not necessarily intended to eliminate module interaction altogether, but rather to reduce the level of ‘surplus’ connections. The evils of high levels of unnecessary coupling are well known. Classes become harder to understand and test in isolation (Cant et al., 1994), there is an increased likelihood of bug propagation on modification, and there is reduced reusability as atomic units cannot easily be extracted from a system and used elsewhere. It is to the purpose of preventing such problems that coupling metrics have been formulated and applied. As with other aspects of object-oriented design, a number of competing metrics are available. Their form is seen to reflect the varying stages of design at which they are intended to be used.

In the early stages of a design the interaction between classes is likely to be described simply in terms of a binary relationship, whereas at a more detailed stage such connections may be expanded into a number of message sends with method argument lists. Examples of early stage coupling metrics are the coupling between objects (CBO) metric (Chidamber and Kemerer, 1994) and the coupling factor (COF) metric (Abreu et al., 1995). Later stage metrics include the message passing coupling (MPC) metric (Li and Henry, 1993) and the parameters per method (PPM) metric (Lorenz and Kidd, 1994). The detail level at which design patterns are described implies that the binary relationship level will be more appropriate, therefore CBO and COF are selected as possible measures for analysis.

The CBO metric is a class level measure, and is simply defined as the number of other classes which a given class references. This metric could be calculated as a maximum or average value – however it is trivial to observe that design pattern structures such as Mediator and Observer imply that the maximum CBO must rise (perhaps significantly). It is therefore considered more fruitful to analyse an averaged form of CBO. Following on from this, the COF metric of Abreu is defined thus:

$$\text{COF} = \frac{\sum_{i=1}^{\text{TC}} \left[\sum_{j=1}^{\text{TC}} \text{is\_client}(C_i, C_j) \right]}{\text{TC}^2 - \text{TC}}, \quad i \neq j, \quad (1)$$

where TC is the total number of classes

$$\text{is\_client}(C_i, C_j) = \begin{cases} 1 & \text{iff } C_i \Rightarrow C_j \wedge C_i \neq C_j, \\ 0 & \text{otherwise.} \end{cases}$$

The *is_client* function returns 1 if class C_i contains at least one non-inheritance reference to class C_j , where a reference may be an argument type, returned value or call to a server class's methods. The COF metric is therefore seen to be calculated at the system level, and is defined as the number of (ordered) class pair combinations coupled to each other as a fraction of all class pair combinations. As such it produces a ratio scale measure, limited to the interval [0,1]. It is observed that COF is effectively equivalent to a normalised average CBO, and so this is the coupling metric which is to be considered.

3.1. An efferent coupling matrix

The current structure of a design, in terms of its classes and the relationships between them, may be denoted by the term 'design state' (Whitmire, 1997). It is this which metrics aim to measure, and to which patterns may be applied to alter the state. One convenient notation for describing the components of a design state is that of matrix algebra, where the presence of a class–class relationship may be signified by a value of 1 in the appropriate row and column. It is then possible to apply basic matrix operations to such a design state matrix to produce resultant matrices containing metric scores. In addition, the transformation from a non-pattern to pattern form can be described in terms of a rearrangement of 1s and 0s within the design state matrix.

For example, an efferent (fan-out) coupling matrix can be defined as follows:

Define the efferent coupling matrix, E , to denote the binary coupling relationship between class pairs within a design. Then

$$E_{ij} = 1 \Rightarrow \text{class } (i) \text{ coupled to class } (j), \\ E_{ii} = 0 \quad \forall i.$$

An example of such a matrix is presented below for a sample structure containing 4 classes where, for in-

stance, class 1 is denoted as having efferent couples to both classes 2 and 4.

$$E = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

The COF metric discussed earlier (1) can then be derived from matrix E , using standard matrix operations, as

$$\text{COF} = \frac{\text{Trace}(E \cdot E^T)}{\text{rows}(E)^2 - \text{rows}(E)}. \quad (2)$$

The denominator term reflects the fact that the number of rows in E is equivalent to the number of classes. The numerator term returns the total number of pair-wise couplings – this is a result of each diagonal element of $E \cdot E^T$ returning the total coupling (i.e., the matrix row sum) for each individual class, as demonstrated below

Set

$$e = E \cdot E^T,$$

then

$$e_{ii} = \sum_j (E_{ij})^2 \Rightarrow e_{ii} = \sum_j E_{ij} \quad (\text{as } E_{ij} \in [0, 1] \quad \forall i, j).$$

3.2. The Mediator pattern

The Mediator pattern is intended to promote loose coupling, in a situation where interactions between groups of classes would otherwise be highly complex. It centralises the responsibility for communication, so avoiding the need for objects to refer to each other explicitly. This enables a single class to be reused in isolation, a more flexible situation than having to always reuse a complete group of classes.

An example of such a situation is presented below in Fig. 1, where the Mediator has been introduced as an extra class. The Mediator may in fact be an abstract class, so that the inter-object communication can be redefined in concrete Mediator subclasses.

Fig. 1. Applying the Mediator pattern.

In terms of the efferent coupling matrix, E , the conversion from the non-pattern form of the Mediator (where the coupling would be direct) to its corresponding pattern form can be described in terms of a simple algorithm. Denote the non-pattern form matrix as $E(P-)$ and the pattern form matrix as $E(P+)$. Then to produce $E(P+)$ from $E(P-)$ requires the following steps

```

Set  $E(P+) = E(P-)$ 
Add row and column to  $E(P+)$ , index  $k$ 

for  $i = 1$  to  $k-1$ 
  for  $j = 1$  to  $k-1$ 
    if  $E(P-)_{ij} = 1$  then
       $E(P+)_{kj} = 1$  and
       $E(P+)_{ik} = 1$ 
       $E(P+)_{ij} = 0$ 
   $E(P+)_{kk} = 0$ 

```

Using this conversion algorithm, a plot of COF for pattern versus non-pattern may be produced by generating random arrangements of the $E(P-)$ matrix and its corresponding $E(P+)$ matrix (in this case for number of classes between 3 and 16). The COF matrix formula (2) may then be applied to create a population of metric score pairs, $COF(P-)$ and $COF(P+)$. Such a monte carlo style approach leads to the following graph (Fig. 2).

Clearly, class groups which exhibit low levels of coupling are not really intended for the application of a Mediator pattern, and this is reflected in the low score region for $COF(P-)$. At this point, $COF(P+)$ scores are in some cases superior (in terms of being lower) than the corresponding $COF(P-)$ values. It is only at the point where $COF(P-)$ values start to exceed approximately 0.35 that the pattern forms consistently produce superior scores. This implies that a value between 0.3 and 0.4

Fig. 2. COF for Mediator pattern versus non-pattern.

may in fact be desirable when setting an upper alarm value for COF.

4. The Bridge pattern and inheritance metrics

The Bridge pattern is used to decouple an abstraction from its implementation. Gamma et al. (1995) quotes the example of a *Windows* class hierarchy, where each subclass of *Window* also requires further subclasses to provide platform specific implementations. In the worst case, an implementation specific subclass could be required for every abstracted class within the hierarchy.

The Bridge pattern is intended to separate an abstraction class hierarchy from its implementation hierarchy. Its general form is illustrated in Fig. 3 below.

The classes marked with an 'i' prefix are the implementation subclasses, with up to N of them for each abstraction class (prefix 'A'). The application of the pattern avoids this continuous subclassing, by converting the design to two coupled but separate hierarchies.

Any analysis of the Bridge pattern in terms of its effect on metric scores is then seen to be dependent on two dimensional variables:

$b1$ = no of abstraction classes,

$b2$ = no of implementation classes.

It is observed that the total number of classes (TC) within the non-pattern and pattern structures are then respectively

Bridge non-pattern

Bridge pattern

Fig. 3. Applying the Bridge pattern.

$$TC(P-) = b1 + b1 \cdot b2, \quad (3)$$

$$TC(P+) = b1 + b2 + 1. \quad (4)$$

The initial analysis to be presented is specifically limited to the case of single inheritance hierarchies. Such a restriction is not considered to be invalid, given that a number of programming languages support only single inheritance. The aim is to initially avoid the extra complexities introduced by multiple and repeated inheritance (Chung and Lee, 1992).

4.1. Depth in tree

The first inheritance metric to be considered is depth in tree (DIT), a measure proposed by Chidamber and Kemerer (1994). The deeper that a class is in the hierarchy, the more behaviour it is likely to inherit and so the more complex it becomes. Conversely, very shallow trees can imply insufficient levels of abstraction. This implies that the average value for DIT is likely to be band pass i.e., $LL \leq DIT_{ave} \leq UL$.

Suppose that the sum of DIT over all the abstraction classes is now given by the new variable $b3$

$$b3 = \sum DIT = d_1 + d_2 + \dots + d_{b1}.$$

Then, for the non-pattern form (which includes all the implementation classes) this sum becomes

$$\begin{aligned} \sum DIT(P-) &= d_1 + d_2 + \dots + d_{b1} + b2 \cdot (d_1 + 1) \\ &\quad + b2 \cdot (d_2 + 1) + \dots + b2 \cdot (d_{b1} + 1) \end{aligned}$$

as each implementation class is at a depth one more than its parent abstraction class. This simplifies to the following expression

$$\sum DIT(P-) = b3 + b2 \cdot (b3 + b1).$$

The sum of DIT for the pattern form can also be easily derived as the sum of DIT for the two separated hierarchies, where all classes within the implementation hierarchy are at depth = 1 except the abstract root class.

$$\sum DIT(P+) = b3 + b2.$$

To calculate the average values for DIT, simply divide the above summations by the total number of classes calculated earlier in (3) and (4).

$$DIT_{ave}(P-) = \frac{b3 + b2 \cdot (b3 + b1)}{b1 + b1 \cdot b2}, \quad (5)$$

$$DIT_{ave}(P+) = \frac{b3 + b2}{b1 + b2 + 1}. \quad (6)$$

Given the above equations, it is then possible to formulate a minimisation/maximisation problem subject to constraints. The quantity to be minimised and maximised is $DIT_{ave}(P+) - DIT_{ave}(P-)$. The aim in working with this quantity is to establish the range of metric score differences caused by the transition from the non-

pattern to the pattern. The constraints applied are reasonable limits upon the variables $b1, b2$. Constraints for $b3$ are seen to be a function of the constraints placed on $b1$.

$$b1 \geq 4, \quad b1 \leq 16$$

(between 4 and 16 classes in the abstraction hierarchy),

$$b1 \geq 1, \quad b2 \leq 8$$

(between 1 and 8 implementation classes).

The variable $b3$ represents ΣDIT for the abstraction classes. The limits for this variable can be derived as a function of $b1$.

$$b3 \geq b1 - 1, \quad b3 \leq (1/2) \cdot (b1 - 1) \cdot (b1 - 2).$$

The first constraint here reflects the fact that minimum ΣDIT occurs where all the abstraction classes are the children of one parent class i.e., $b1 - 1$ classes are at depth = 1. The second constraint is a result of maximum ΣDIT occurring when the inheritance hierarchy is completely linear i.e., the classes are at depth $(0, 1, 2, \dots, b1 - 1)$.

Solving the above non-linear optimisation problem yields a metric score difference range for the pattern and non-pattern.

Minimum difference occurs at

$$DIT_{ave}(P+) = 4.52,$$

$$DIT_{ave}(P-) = 7.45,$$

$$\text{when } b1, b2, b3 = (16, 8, 105),$$

Maximum difference occurs at

$$DIT_{ave}(P+) = 0.89,$$

$$DIT_{ave}(P-) = 1.44,$$

$$\text{when } b1, b2, b3 = (16, 1, 15).$$

The decreased metric scores for the pattern illustrate the fact that its application will reduce average DIT in the circumstances covered by the constraint conditions. It will reduce an excessive value for average DIT, and this is consistent with the metric's aim of decreasing accumulated complexity in the leaf nodes. However, by plotting some of the intermediate points between the two extrema given above, it can be seen that the level of reduction continues for lower scores of average DIT (Fig. 4). This may be undesirable, as these scores could be pushed below lower tolerance limits.

4.2. Number of children

The number of children (NOC) metric is again proposed by Chidamber and Kemerer (1994). The viewpoint adopted for this measure is that the greater the NOC the greater the level of reuse since inheritance is a

Fig. 4. Average DIT for Bridge pattern versus non-pattern.

form of reuse. Against this, a class with a large NOC has a greater potential influence on the design, and may require more testing. This again implies a band-pass metric.

Suppose now that the sum for NOC over all the Bridge abstraction classes is given simply as a function of the number of abstraction classes, namely the value $b1 - 1$. This is merely a reflection of the fact that each class in the abstraction hierarchy is taken to share a common descendent root node.

Then, the sum of the total NOC for the non-pattern form of Bridge is given by

$$\sum \text{NOC}(P-) = b1 - 1 + b1 \cdot b2$$

as each of the $b1$ abstraction classes has an additional $b2$ implementation subclasses.

The corresponding summation formula for the pattern form is given by

$$\sum \text{NOC}(P+) = b1 - 1 + b2$$

as the NOC in the implementation hierarchy is simply the number of implementation classes.

To calculate the average values for NOC, simply divide the above summations by the number of classes calculated earlier for DIT in (3) and (4). This is taking the initial assumption that leaf nodes (i.e., classes with no children) are to be included in the average NOC calculation.

$$\begin{aligned} \text{NOC}_{\text{ave}}(P-) &= \frac{b1 - 1 + b1 \cdot b2}{b1 + b1 \cdot b2} \\ &= 1 - \frac{1}{b1 \cdot (1 + b2)}, \end{aligned} \quad (7)$$

$$\text{NOC}_{\text{ave}}(P+) = \frac{b1 - 1 + b2}{b1 + b2 + 1}. \quad (8)$$

Fig. 5. Average NOC for Bridge pattern versus non-pattern.

The constraints on $b1$ and $b2$ are taken to be the same as those for the average DIT calculations.

Again, solving the non-linear optimisation problem yields a metric score difference range for the pattern and the non-pattern.

Minimum difference occurs at

$$\begin{aligned} \text{NOC}_{\text{ave}}(P+) &= 0.67, \\ \text{NOC}_{\text{ave}}(P-) &= 0.88, \\ \text{when } b1, b2 &= (4, 1), \end{aligned}$$

Maximum difference occurs at

$$\begin{aligned} \text{NOC}_{\text{ave}}(P+) &= 0.92, \\ \text{NOC}_{\text{ave}}(P-) &= 0.99, \\ \text{when } b1, b2 &= (16, 8). \end{aligned}$$

The pattern is observed to have a distinct reducing effect on average NOC, and show a strong correlation (see Fig. 5), when including leaf classes in the metric's calculation. The possible problem here is that lower scores for average NOC may then be reduced below lower tolerance limits.

4.2.1. Number of children (excluding leaf classes)

It has been noted that there exists an alternative interpretation of the NOC metric, namely one which excludes the inheritance hierarchy leaf node classes from the calculation. This will fundamentally alter the distribution of scores for the metric, as average NOC on this basis is no longer $\in[0,1]$ but rather $\in[0,\infty]$.

The actual total NOC counts (the numerator values in (7) and (8)) will not change, however the denominator values for number of classes (given in (3) and (4)) will no longer be valid. Leaf nodes must be removed from these total class counts, and to this end an additional variable,

$b4$, must be introduced. This is intended to denote the number of leaf nodes in the abstraction hierarchy. The limits on $b4$ are then 1 (a completely linear hierarchy) and $b1 - 1$ (all classes share one common parent).

This leads to the following expressions for non-pattern and pattern average NOC

$$\text{NOC}_{\text{ave}}(P-) = \frac{b1 - 1 + b1 \cdot b2}{b1}, \quad (9)$$

$$\text{NOC}_{\text{ave}}(P+) = \frac{b1 - 1 + b2}{b1 - b4 + 1}. \quad (10)$$

The denominator in (9) reflects the fact that the only non-leaf classes in the non-pattern are the abstraction classes (these all have at least one implementation subclass). Where the hierarchies have been separated for (10), only non-leaf abstraction classes plus the new abstract implementation root class remain for the class count.

The constraints on $b1$ and $b2$ are taken to be the same as those for the average DIT calculations.

Again, solving the non-linear optimisation problem yields a metric score difference range for the pattern and the non-pattern.

Minimum difference occurs at

$$\text{NOC}_{\text{ave}}(P+) = 1.44,$$

$$\text{NOC}_{\text{ave}}(P-) = 8.94,$$

$$\text{when } b1, b2, b4 = (16, 8, 1),$$

Maximum difference occurs at

$$\text{NOC}_{\text{ave}}(P+) = 8,$$

$$\text{NOC}_{\text{ave}}(P-) = 1.94,$$

$$\text{when } b1, b2, b4 = (16, 1, 15).$$

The pattern is observed to have a non-determinate effect on average NOC, and show little correlation when the leaf nodes are excluded (see Fig. 6). The Bridge pattern therefore implies that leaf nodes should be included in order to produce a more discriminatory measure.

5. The Visitor pattern and method counts

The Visitor pattern allows for new operations to be performed upon all of the classes in an inheritance hierarchy structure, without actually having to add the operation to each hierarchy class. It is recommended for use in the circumstances where the structure of the class hierarchy being visited is not likely to change, but where new operations may often be required. The basic effect of its application is the removal of a subset of operations from one class hierarchy (the Node hierarchy), and their placement within a new Visitor hierarchy (Fig. 7). This

Fig. 6. Average NOC for Bridge pattern versus non-pattern (–leaf classes).

implies changes to metrics relating to method counts, inheritance structure and polymorphism.

Define

$v1$ = no of nodes to visit,

$v2$ = no of operations.

Then the total number of classes for the non-pattern and pattern structures are:

$$\text{TC}(P-) = 1 + v1, \quad (11)$$

i.e., all nodes + abstract root node

$$\text{TC}(P+) = (1 + v1) + (1 + v2), \quad (12)$$

i.e., $\text{TC}(P-) + \text{new visitor hierarchy}$.

Now consider the number of methods metric, NOM (Lorenz and Kidd, 1994). This is equivalent to the

Fig. 7. Applying the Visitor pattern.

‘weighted methods per class’ metric (Chidamber and Kemerer, 1994), with each method weight set to unity. The summations for this metric are then given by

$$\sum \text{NOM}(P-) = (1 + v1) \cdot v2,$$

$$\sum \text{NOM}(P+) = 1 \cdot (1 + v1) + (1 + v2) \cdot v1.$$

(as there is 1 *accept* operation remaining per node).

Average values for the above are then calculated by dividing by the number of classes given in (11) and (12).

$$\text{NOM}_{\text{ave}}(P-) = \frac{(1 + v1) \cdot v2}{1 + v1} = v2, \quad (13)$$

$$\text{NOM}_{\text{ave}}(P+) = \frac{1 + 2 \cdot v1 + v1 \cdot v2}{2 + v1 + v2}. \quad (14)$$

Constraints are simply defined as the number of nodes and operations which may be considered to be present.

$$v1 \geq 4, \quad v1 \leq 16$$

(between 4 and 16 nodes in the hierarchy),

$$v2 \geq 2, \quad v2 \leq 16$$

(between 2 and 16 operations).

Again, solving the above non-linear optimisation problem yields a metric score difference range for the pattern and the non-pattern.

Minimum difference occurs at

$$\text{NOM}_{\text{ave}}(P+) = 3.32, \text{NOM}_{\text{ave}}(P-) = 16, \\ \text{when } v1, v2 = (4, 16),$$

Maximum difference occurs at

$$\text{NOM}_{\text{ave}}(P+) = 3.25, \text{NOM}_{\text{ave}}(P-) = 2, \\ \text{when } v1, v2 = (16, 2).$$

The pattern is seen to have a significant reducing effect where classes would otherwise be over-subscribed with

overrides of virtual methods. It has a lesser effect where the number of Visitor operations is more limited, and in this case can actually increase average NOM slightly. A plot of average NOM for pattern versus non-pattern (Fig. 8) reveals that applying the pattern will generally be in accordance with attempts to use average NOM as a band-pass metric.

6. Conclusions and future work

The purpose of considering design patterns in conjunction with design metrics has been to determine their compatibility, in terms of their application in the improvement of design quality. The usage of a metric must be questioned if it discourages the integration of an established design pattern into a software structure. Conversely, there may be circumstances where the application of a given pattern may not be appropriate, and a metric violation highlights this.

The analysis undertaken has considered the three patterns Mediator, Bridge and Visitor, and compared them with variations of their non-pattern forms. This comparison has relied on formulating metric scores in terms of the dimensional parameters for the pattern. An example of such dimensional parameters are the number of nodes and the number of operations for the Visitor pattern.

The results observed display little in the way of pattern-metric conflicts, and it appears that the usage of the patterns examined can be relied upon to reduce high metric scores which might otherwise cause alarm. The caveat to this is that certain of the patterns examined could cause lower limit violations for band-pass measures, one particular case being the Bridge pattern and the average NOC metric (including leaf nodes).

A number of issues have been identified which will need addressing in order to produce a set of more complete and convincing conclusions. Firstly, the number of patterns and metrics presented within this paper has been limited (although it is observed that only certain metric scores will be influenced by a given pattern). Further patterns will need to be examined in order to evaluate metrics relating to concepts such as polymorphism, cohesion and information hiding (Abreu et al., 1995; Rising and Calliss, 1994). Examples of such patterns are the Strategy, State and Memento patterns. The Bridge pattern can also be further studied to evaluate metrics such as the Reuse Ratio and Specialisation Ratio measures (Henderson-Sellers, 1996).

Scalability is a concern given that the pattern under consideration may represent only a very minor part of an overall design. In this case, any effect on metric scores may be diluted by the structure of the classes external to the pattern. This implies that the conclusions so far reached would only be valid when the metrics are cal-

Fig. 8. Average NOM for Visitor pattern versus non-pattern.

culated at the individual subsystem (or framework) level. One solution to this is to include the external values as additional variables in the optimisation equations, subject to their own constraints.

There is also a requirement to look at combinations of patterns. This is due to the possibility that a given metric may be consistent with one pattern form, but in conflict with another. In this case it must be investigated whether the metric should be rejected outright, whether a reworking of the metric form will resolve the situation, or whether the patterns themselves could in some sense be in opposition.

References

- Abreu, F.B., Esteves, R., Goulao, M., 1995. The design of Eiffel programs: quantitative evaluation using the MOOD metrics. Technical Report, INESC, Lisbon, Portugal.
- Bansiya, J., Etzkorn, L., Davis, C., Li, W., 1999. A class cohesion metric for object-oriented designs. *Journal of Object-Oriented Programming* 11 (8), 47–52.
- Basili, V.R., Briand, L.C., Melo, W.L., 1996. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions On Software Engineering* 22 (10), 751–760.
- Binkley, A.B., Schach, S.R., 1996. A comparison of sixteen quality metrics for object-oriented design. *Information Processing Letters* 58, 271–275.
- Briand, L.C., Daly, J.W., Wust, J., 1997. A unified framework for cohesion measurement in object-oriented systems. IESE-Report No 040.97/E, Fraunhofer IESE.
- Brown, W.J., Malveau, R.C., McCormick, H.W., Mowbray, T.J., 1998. *AntiPatterns*. Wiley, New York.
- Cant, S.N., Henderson-Sellers, B., Jeffery, D.R., 1994. Application of cognitive complexity metrics to object-oriented programs. *Journal of Object-Oriented Programming* 7 (4), 52–63.
- Chidamber, S., Kemerer, C., 1994. A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering* 20 (6), 476–493.
- Chung, C.-M., Lee, M.-C., 1992. Inheritance-based metric for complexity analysis in object-oriented design. *Journal of Information Science and Engineering* 8 (3), 431–437.
- Churcher, N.I., Shepperd, M.J., 1995. Comments on ‘A metrics suite for object-oriented design’. *IEEE Transactions on Software Engineering* 21 (3), 263–265.
- Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. *Design Patterns: Elements of Reusable Design*. Addison-Wesley, Reading, MA.
- Henderson-Sellers, B., 1996. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, NJ.
- Henderson-Sellers, B., Constantine, L.L., Graham, I.M., 1996. Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design). *Object-Oriented Systems* 3, 143–158.
- Henry, S., Kafura, D., Schulman, R., 1995. Measuring object-oriented design. *Journal of Object-Oriented Programming* 8 (4), 48–55.
- Hitz, M., Montazeri, B., 1996. Chidamber and Kemerer’s metrics suite: a measurement theory perspective. *IEEE Transactions on Software Engineering* 22 (4), 267–271.
- Li, W., 1998. Another metric suite for object-oriented programming. *Journal of Systems and Software* 44, 155–162.
- Li, W., Henry, S., 1993. Object-oriented metrics that predict maintainability. *Journal of Systems and Software* 23, 111–122.
- Lorenz, M., Kidd, J., 1994. *Object-Oriented Software Metrics*. Prentice-Hall, New Jersey.
- Rising, L.S., Calliss, F.W., 1994. An information hiding metric. *Journal of Systems and Software* 26, 211–219.
- Schach, S.R., 1996. The cohesion and coupling of objects. *Journal of Object-Oriented Programming* 9, 48–50.
- Shepperd, M., Ince, D., 1993. *Derivation and Validation of Software Metrics*. Oxford University Press, Oxford.
- Shepperd, M., Ince, D., 1994. A critique of three metrics. *Journal of Systems and Software* 26 (3), 197–210.
- Tegarden, D.P., Sheetz, S.D., Monarchi, D.E., 1995. A software complexity model of object-oriented systems. *Decision Support Systems* 13, 241–262.
- Ververs, F., Pronk, C., 1995. On the interaction between metrics and patterns. In *Proceedings of OOIS ’95 Dublin*, pp. 303–314.
- Weyuker, E., 1988. Evaluating software complexity measures. *IEEE Transactions on Software Engineering* 14 (9), 1357–1365.
- Whitmire, S., 1997. *Object-Oriented Design Measurement*. Wiley, New York.

Brian Huston is a Principal Software Engineer at BAE Systems, where he has worked for the last ten years. He received his B.Sc. degree in Mathematics and Statistics from the University of Sussex in 1989, and M.Sc (with distinction) in Information Systems from the University of Portsmouth in 1996. He is currently approaching completion of a Ph.D. in Software Engineering, which he has been pursuing at Southampton Institute. His main research activity lies in the area of object-oriented design, with a particular emphasis on design metrics and patterns. Other areas of interest include real time simulation, distributed systems and software reuse.