Metrics for Evaluating Concern Separation and Composition

A. Di Stefano, M. Fargetta
Dipart. di Ing. Informatica e Telecomunicazioni
Università di Catania
{adistefa,mfargetta}@dit.unict.it

G. Pappalardo, E. Tramontana
Dipart. di Matematica e Informatica
Università di Catania
{pappalardo,tramontana}@dmi.unict.it

ABSTRACT
This paper discusses an approach to evaluating the separation of concerns for an object-oriented software system. For assessing this separation, the developer is asked to specify the nature of classes through annotations. Automatic identification of some structural characteristics (e.g., inheritance, libraries, synchronisation) is used to appraise the composition and intertwining of concerns inside a class.

Categories and Subject Descriptors
D.2.8 [Software Engineering]: Metrics

General Terms
Design, Measurement

Keywords
Software Engineering, Computational Reflection

1. INTRODUCTION
Current metrics, whether static [2] or dynamic [1], only consider interactions between classes/objects. To the best of our knowledge, none of them deals with concerns, or the dependence of code on resources (possibly external to applications).

In this paper we propose some metrics for object-oriented systems, useful to shed light on the separation of concerns, and their relation with classes, within an application. We maintain that these characteristics cannot be captured exclusively by static measures, but must also be appraised dynamically, to cater for phenomena that only manifest themselves at run-time, such as the dynamic binding typically brought about by recourse to polymorphism and inheritance. In our approach, dynamic measures are based on the observation of object behaviour at run-time, performed by means of a suitable reflective architecture.

2. METRIC FOR CONCERN SEPARATION
According to the principle of separation of concerns, an application should be developed by isolating “as much as possible”, within it, separate portions providing independent solutions to specific problems or concerns. Clear separation is achieved when the design of a portion is totally unaware of the existence of the others.

Admittedly, in general, the precise identification of a concern is a difficult task, both on a conceptual and a linguistic level. Conceptually, the boundaries of a concern can be refined as the development, or the analysis, of a system goes on. From a linguistic point of view, in object-oriented systems a concern is of course typically expressed by the actions performed by a set of classes; however, this relationship varies widely with the design style and model and is therefore hard to identify on the basis of general, a priori criteria. We therefore rely on designers to specify, by annotations, the main concern each class implements.

The separation between two concerns can be measured by relating it with the overall coupling between their constituting classes. For this purpose, we elaborate on the “Coupling Between Objects” (CBO) metric of [2], and introduce $CBC(c, c')$, the “Coupling between Concerns” $c$ and $c'$. This is defined as the number of different methods, invoked by all actions performed by a set of classes; however, this relationship varies widely with the design style and model and is therefore hard to identify on the basis of general, a priori criteria. We therefore rely on designers to specify, by annotations, the main concern each class implements.

For example, consider Figure 1, which shows a class diagram where $c$ and $c'$ consist of classA and classB, and classC, classD and classE, respectively. Thus, we have $CBC(c, c') = 3$ and $CBC(c', c) = 1$.

![Figure 1: Concerns and classes.](image)

3. A CATEGORY BASED METRIC
We characterise classes by classifying each interaction they initiate into a relevant category. A category is intended to set apart interactions that share some common, distinctive features. Such a characterisation can help revealing the aim of a class and whether different concerns are intertwined inside it. Indeed, if the interactions started by a class be-
long to many different categories, this is an indication that its code is likely to address, and intertwine, many different concerns, which may cause it to be of a poorer quality. Besides highlighting concern separation, this information helps identifying the dependence of classes and applications on external resources, hence the characteristics of the execution environment they need.

For a class, we consider the following interaction categories: (a) hierarchy, interactions performed with its ancestor class; (b) peer, interactions with other (non-ancestral) application classes; (c) external, interactions with devices, such as I/O, network, etc.; (d) sync, interactions meant to synchronise access to some shared resource; (e) lib, interactions with libraries other than those involved in the two preceding categories.

Among the interactions initiated by a class cl, we define the amount pertaining to category k, denoted ITCk(cl), as the number of different methods of category k invoked by cl. A class can be characterised by a tuple of values, one for each of the categories identified above. Furthermore, we express the category diversity among the interactions performed by class cl by an index fragm(cl). This is intended to unveil a potentially cumbersome concern heterogeneity, or fragmentation, within the code of class cl, and is defined:

$$\text{fragm}(cl) = \frac{1}{\sqrt{K} - 1} \left( \frac{\sum_{k=1}^{K} \text{ITC}_k(cl)}{\sqrt{\sum_{k=1}^{K} (\text{ITC}_k(cl))^2}} - 1 \right)$$

where K is the number of categories. It is not difficult to prove that the right-hand expression takes its maximum, namely 1, if ITCk(cl) is constant with k, and its minimum, namely 0, if all categories but one score 0.

4. MEASUREMENTS ON AN APPLICATION

Let us consider an address book application consisting of the classes: Person, handling contact information; AddrBook, handling Person objects; Store, managing Person objects on disk; Dialog, ListDialog and InfoDialog handling windows that show data and messages to the user.

The concerns implemented by the address book are: graphical, consisting of Dialog, ListDialog and InfoDialog, which provide the user front-end; domain, consisting of Person, AddrBook, which provide the core functionalities; and i/o, implementing access to disks by means of Store. The separation between concerns is illustrated in Table 1 in terms of the “coupling” metric CBC, and concern heterogeneity within classes is shown in Table 2 by the “categories” metrics ITC and fragm.

<table>
<thead>
<tr>
<th>graphical</th>
<th>domain</th>
<th>i/o</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 1: Measures of CBC.

CBC values show that graphical and i/o concerns are not coupled, whereas, for every other concern pair, dependence is in both directions (e.g. domain is coupled with graphical and conversely).

As an indication of the effort needed for understanding classes, the values of fragm in Table 2 are in good agreement with what visual code inspection suggests.

<table>
<thead>
<tr>
<th>Person</th>
<th>AddrBook</th>
<th>Store</th>
<th>Dialog</th>
<th>ListDialog</th>
<th>InfoDialog</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.00</td>
<td>0.58</td>
<td>0.33</td>
<td>0.17</td>
<td>0.12</td>
</tr>
</tbody>
</table>

Table 2: Measures of interaction categories.

The address book has been enhanced by introducing a checkpointing functionality, through a reflective metalevel. A first version has a checkpointing metalevel that is strictly separated from the baselevel. This approach is costly, since at run-time full inspection of the application classes needs to be performed, to detect the types and values of changing state variables. A second, lighter version of the metalevel exploits some knowledge about the application classes (i.e. fields names and types), so that run-time inspection is no longer needed. However, this version of the metalevel is not fully separated from the baselevel. As we can expect, CBC figures show that baselevel concerns are not coupled upwards, with either version of the metalevel. Downwards, the first version of metalevel concern checkpoint is not coupled with any baselevel concern, whereas the second version is coupled with domain.

5. DYNAMIC MEASURES

In the presence of polymorphism and inheritance, static code exploration cannot detect coupling. The proposed metrics are thus modified to consider the actual classes invoked during execution. The new measures are calculated by counting at run-time the number of different methods invoked by class instances. Dynamic CBC considers calls from instances of classes belonging to a concern, to instances of classes belonging to another concern. Dynamic ITC considers calls from a class instance to an interaction category.

To perform measures on systems at run-time, it is necessary to connect them to a measurement tool. We have employed a reflective architecture that measures a running application by associating metaobject classes with all the application classes. This is a typical example where clear separation between systems and measurement tools is of great importance. Thanks to the reflective solution adopted, application objects are strictly unaware that measures are being performed on them.

6. CONCLUSIONS

This work has proposed an elaboration of the standard metric that evaluates coupling between objects, as a way to estimate the separation of concerns. We have also suggested that useful information can be derived from a categorisation of the interactions started by classes.

7. REFERENCES
