Analyzing Software Systems by Using Combinations of Metrics

Markus Bauer, FZI Karlsruhe*

March 29, 1999

Abstract

Reengineering an object oriented software system requires some analysis of the existing system. This paper describes, how a combination of complexity and coupling metrics can be used to make such an analysis more focused and thus more effective: the metrics are used to identify the *key classes* of a system, i.e. classes that structure the system and implement its most important concepts — understanding them and their collaborations is a good basis for every reengineering task. The paper is intended to be particularly helpful for practitioners, therefore it focuses on practical experiences that have been made with various case studies.

Keywords: object orientation, reengineering, software metrics, model capture

1 Introduction

The ability to reengineer object oriented legacy systems and transform them into more flexible and extensible systems has become a matter of vital interest in today's software industry. To allow such a transformation of a legacy system into an improved flexible system two tasks are particularly important [CDD$^+$99]:

- *model capture*: document and understand the existing system, its structure and its inner workings
- *problem detection*: identify possible quality or flexibility related problems of the system

Object oriented software metrics are expected to support these tasks, but in practice they have only been applied with limited success. Though a vast amount of metrics have been defined in the past (see for example [CK94], [LH93] or [BK95]) there is little experience in how to actually use metrics for model capture or problem detection.

To improve this situation, this paper describes a practical approach to use metrics in order to help understanding the basic structure and workings of legacy system and some experiences that were made while applying that approach to real world C++ applications.

*Forschungszentrum Informatik, Haid–und–Neu–Str. 10–14, 76131 Karlsruhe, Germany; Internet: `bauer@fzi.de`; `http://www.fzi.de/bauer.html`

Understanding the structure and the inner workings of a legacy system is an essential task during any reengineering project, since any improvements to the system rely on such knowledge. As these legacy system are often large and insufficiently documented, their source code has to be examined. This is time consuming and expensive (or even impossible).

It is therefore crucial to know which parts of the system implement its key concepts because the analysis of the system can then be focused on these important parts, thus making the model capture process much more efficient.

2 Our Approach

In the following, we describe a technique that helps us to discover the important parts of a system.

First, we observe that these important parts of the system are implemented by very few *key classes*. If we are able to identify these key classes and their collaborations, we have good starting point to learn more about the working of a system.

To identify these key classes, we use two properties that are symptomatic for them:

- Key classes either *manage* a rather large amount of other classes or *use* them in order to implement their functionality. Therefore, these key classes are tightly *coupled* with other parts of the system.
- A second property of these classes is that they tend to be rather *complex*, since they implement vast parts of the system's functionality.

For both properties, coupling and complexity, a variety of metrics have been defined in the past. Thus, to identify key classes it is rather natural to make use of these metrics: We compute both, coupling metrics and complexity metrics for the system's classes and *combine* the results.

Combining the results can be done in various ways, either formal, for example by defining a combined metric (as a single formula) or rather informal, for example by using a bar chart that shows both metrics at the same time (Figure 1).

For our case studies we have used an approach that incorporates a graphical evaluation as well as mathematical calculations, but summarizes the results some more. This is especially important for the analysis of large systems.

We use a diagram that places the classes in a coordinate system, showing the complexity metric on the first axis, and the coupling metric in the second axis (see figure 2). This way, we can easily identify the key classes we are interested in: we just have to pick those classes that are in upper right area of the diagram (see the grey shaded area in figure 2).

Mathematically, we combine the two metrics by computing the *distance* d of a class from the origin of the coordinate system: if x denotes the complexity value of a class, and y its coupling value, we compute the combined value d as:

$$d = \sqrt{x^2 + y^2}$$

In some cases, if the metrics use a very different scale, e.g. the complexity metric produces significantly higher values than the coupling metric, some normalization might be required. We could then use the formula

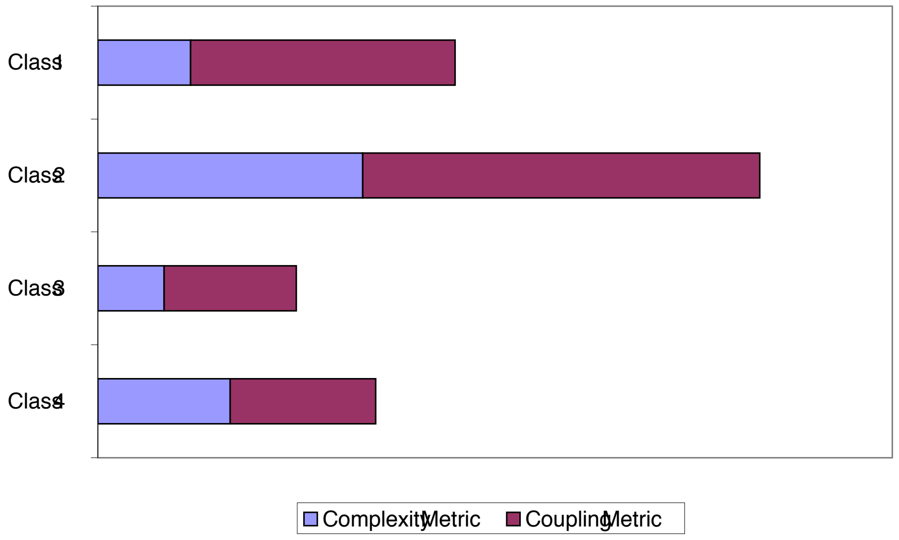

Figure 1: Combining metrics by using a stacked bar chart.

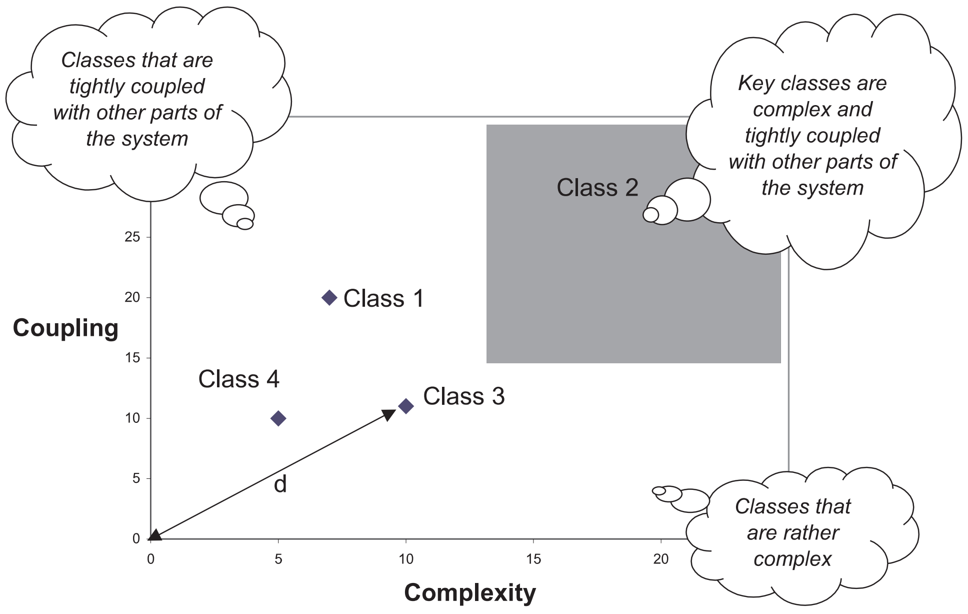

Figure 2: Graphical analysis of a system.

$$d = \sqrt{(\frac{x}{x_{\max}})^2 + (\frac{y}{y_{\max}})^2}$$

This combined value allows us to compare classes – classes with higher values for d are better candidates to key classes of the system than classes with lower values for d.

As we will see in the next section, coordinate systems and tables with combined metric values d provide a good means to identify the key classes of the system: classes that represent the important parts of a system.

3 Practical Experiences

We have applied our approach to various case studies written in C++ from different backgrounds, including academic projects as well as real world industrial applications. In the following, we describe some experiences we made with our approach and these case studies.

One of our case studies, the *ET++ Application Framework* [WG94] is covered in more detail since it is well documented and publically available, so our results can be easily reproduced. Two other case studies, code named *River* and *CLEA* have been used to confirm our results.

- *ET++* is a framework to facilitate the construction of applications that provide a graphical user interface.
- *River* is an application that visualizes water flow data for rivers. Its implementation is based on Microsoft's MFC-Framework.
- *CLEA (Class Library for Engineering Applications)* is both a class library and a framework that is used to implement a family of applications that support mechanical engineers. *CLEA* also relies on Microsoft's MFC.

Some general purpose data about these case studies is given in the upper part in table 2.

All case studies have been analyzed with a tool set developed within the FAMOOS project[1], which analyses C++ code, extracts class declarations, method declarations, variable declarations, method invocations and variable accesses from the code and stores them into databases. With an SQL-like query language different kinds of software metrics can be programmed and computed using these databases. (For a detailed description of the tool set and the metrics supported, see [Mar97]).

To identify the key classes in these case studies using our approach, we have mainly employed the *WMC* metric [CK94] as complexity metric and the *DAC* metric [LH93] as coupling metric, because we achieved best results with them. The WMC metric measures the complexity of a class by summing up the *McCabe cyclomatic complexities* of its methods. The DAC metric measures the coupling of a class by counting the number of other classes that are used as attributes in the class. Possible alternative metrics would include NOM (complexity) and RFC (coupling), see [CK94].

For each case study, we computed WMC and DAC metrics and created the coordinate system — WMC metric vs. DAC metric — as described in the previous section. For *ET++*, this coordinate system is shown in figure 3. We observe that only a few classes have high values for both metrics.

[1] See `http://dis.sema.es/projects/FAMOOS`

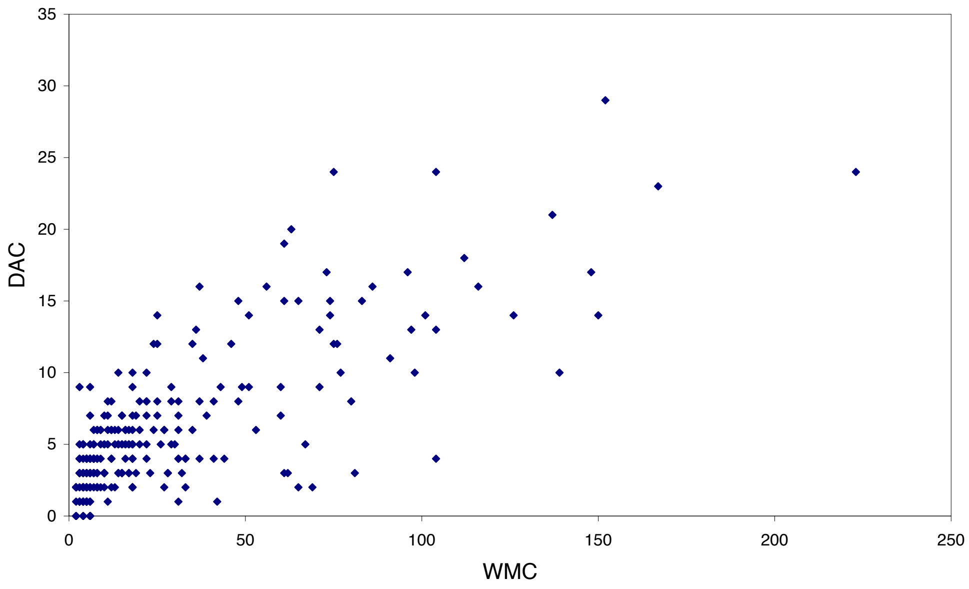

Figure 3: Coordinate system showing WMC and DAC metrics for *ET++*.

Using the coordinate system, we identified the key classes of the systems by performing the following steps:

1. *Pick candidates* for the key classes. Such candidates are classes in the upper right corner of the coordinate system. (We defined upper right corner as the rectangle bordered by the third quartiles of both metrics.)[2]

2. *Structure the candidates* into categories. This should be done to get a better overview on the list of candidates and to prepare for the next step. We structured the candidates by using naming conventions and inheritance relationships.

3. *Eliminate classes* that are not really key classes and that are not that important to understand how the system works. We did this by browsing through the source code.

In table 1 we present some examples for key classes we found by applying this approach to the *ET++* case study.

Table 2 shows some of our measurements for all three case studies. At the bottom, we present the number of classes that represent candidates for key classes, and the number of classes that we finally identified as key classes after step 3 above.

Some observations:

- For each case study, only about 15 percent of all classes have been identified as candidates to be key classes. For some basic model capture purposes it is usually sufficient, to examine only those classes, that have been picked candidates for key classes. Thus, we have drastically reduced the effort necessary to understand the basic workings of the system.

[2]The third quartile of a data sample is defined as the smallest value of those 25 percent of the data points, that have largest values.

Category	Class	WMC	DAC	d
Base	`Object`	97	13	0.6
	`Class`	71	9	0.4
	`Collection`	96	17	0.7
Managers	`Application`	86	16	0.7
	`Document`	126	14	0.7
	`Dialog`	36	13	0.5
	`Menu`	75	15	0.6
Views	`VObject`	148	17	0.9
	`View`	74	14	0.6
	`TreeView`	35	12	0.4
	`TextView`	167	23	1.1
Ports	`Port`	150	14	0.8
	`WindowPort`	139	10	0.7
	`XWindowPort`	152	29	1.1
	`SunWindowPort`	223	24	1.3
	`PostscriptPort`	83	15	0.6
Average values		27.5	5.9	0.3

Table 1: Key classes for *ET++*.

	ET++	River	CLEA
LOC	55 000	14 200	90 000
Classes	356	52	236
avg. WMC	27.5	35.6	21.9
avg. DAC	5.9	6.6	5.9
max. WMC	223	273	222
max. DAC	29	23	35
3rd quartile WMC	32	35	27
3rd quartile DAC	8	10	8
avg. d	0.3	0.3	0.6
max. d	2.3	1.4	6.3
Candidates for key classes	40 (11%)	9 (16%)	34 (14%)
Key classes	29 (8%)	7 (13%)	30 (13%)

Table 2: Measurements and results for the case studies.

- After structuring these candidates and eliminating some some less important classes, the remaining classes for *ET++* covered most of the concepts described in [WG94], so our approach would have provided us with a good starting point to understand the system's structure by looking at the source code of these candidates.
- For *River* and *CLEA* we were able to discuss our results with the developers of the systems. They confirmed that we have indeed identified those classes that implement the key concepts of the systems.
- By just looking at class sizes, the list of candidates would have been much more inaccurate. So our approach to use a *combination* of metrics is sensible.

4 Conclusions and Further Work

Using a combination of coupling and complexity metrics proved to be a very powerful concept for identifying the key classes of a system. We have therefore shown, that metrics can be successfully applied for model capture purposes, since they allow for a focused examination of the system.

It is very probable, that the concept of combining multiple metrics is very promising in other cases as well. During problem detection, for example, we could use a combination of complexity and cohesion metrics (for example WMC and TCC) to identify classes that should be split, because they implement two or more separate concepts.

Combining metrics that measure the same property of classes in different ways can be a good idea to make the results more robust (i.e less dependent on the specific definitions of the metrics). For example a combination of two coupling metrics (like DAC and RFC) could give better results.

Furthermore we can conclude, that graphical evaluation techniques make the application of metrics for the analysis of legacy systems more intuitive and usable, because they give a concise representation of the systems characteristics. It is therefore desirable to invest some more research in this area.

References

[BK95] J. M. Bieman and B. K. Kang. Cohesion and reuse in an object-oriented system. *Proc. ACM Symposium on Software Reusability*, apr 1995.

[CDD$^+$99] O. Ciupke, S. Demeyer, S. Ducasse, R. Marinescu, R. Nebbe, T. Richner, M. Rieger, B. Schulz, S. Tichelaar, and J. Weisbrod. *The FAMOOS Handbook of Reengineering*. Software Composition Group, University of Bern, Switzerland, 1999.

[CK94] S. R. Chidamber and C. F. Kemerer. A Metric Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[LH93] W. Li and S. Henry. Maintenance metrics for the object oriented paradigm. *IEEE Proc. First International Software Metrics Symp.*, pages 52–60, may 1993.

[Mar97] Radu Marinescu. The use of software metrics in the design of object oriented s ystems. Master's thesis, University P̈olytechicaT̈imisoara, sep 1997.

[WG94] A. Weinand and E. Gamma. ET++ – a portable, homogenous class library and application framework. In W.R. Bischofberger and H.P. Frei, editors, *Computer Science Research at UBILAB, Strategy and Projects*, Proeedings of the UBILAB Conference, pages 66–92. UBILAB, Universitätsverlag Konstanz, September 1994.

[WGM88] A. Weinand, E. Gamma, and R. Marty. ET++ — an object-oriented application framework in C++. In *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications, ACM SIGPLAN Notices*, page 46, November 1988.