Talk
on
"Design Metrics for Object-Oriented Software Systems"

by Fernando Brito e Abreu, INESC/ISEG

ABSTRACT

This presentation introduces some advances towards the quantitative evaluation of design attributes of object-oriented software systems. We believe that these attributes can express the quality of internal structure, thus being strongly correlated with quality characteristics like analyzability, changeability, stability and testability, which are important to software developers and main tainers. An OO design metrics set is reviewed, along with its rationale. Several suppositions regarding the design are evaluated. Results described elsewhere show that some design heuristics can be derived and used to help guide the design process.

INTRODUCTION

The backbone of any software system is its design. It is the skeleton where the flesh (code) will be supported. A defective skeleton will not allow harmonious growth and will not easily accommodate change without amputations or cumbersome prothesis with all kinds of side effects. Because requirements analysis is most times incomplete, we must be able to build software designs which are easily understandable, alterable, testable and preferably stable (with small propagation of modifications). The Object-Oriented (OO) paradigm includes a set of mechanisms such as inheritance, encapsulation, polymorphism and message-passing that are believed to allow the construction of designs where those features are enforced. However, a designer must be able to use those mechanisms in a "convenient" way. Long before the OO languages became widespread, it was possible to build software with an OO "flavor", using conventional 3rd generation languages. Conversely, by simply using an OO language that supports those mechanisms we are not automatically favored with an increase in software quality and development productivity, because its effective use relies on the designer's ability. Being a "creative" activity, where multiple alternatives are often available for the same partition of the system being modeled, design would greatly benefit if some heuristics could help choose the way. Design metrics are being used for this purpose.

Maintenance is (and will surely continue to be) the major resource waster in the whole software life-cycle. Maintainability can be evaluated through several quality sub-characteristics like analyzability, changeability, stability and testability. Object- orientation is believed to reduce the referred effort waste, if its basic mechanisms are used conveniently. We believe and expect to prove that, at the system level, there are patterns for the extent of use of encapsulation, inheritance, polymorphism or cooperation among classes which are closely correlated with those quality characteristics. By finding those patterns, through thorough experimental validation, we do not expect to identify a good design when we see one, but rather to say that a certain design is more maintainable than another. This is particularly useful for inexperienced designers, often faced with a combinatorial explosion of arbitrary design decisions.

THE MOOD METRICS SET

The MOOD (Metrics for Object Oriented Design) set includes the following metrics:

Method Hiding Factor (MHF)

Attribute Hiding Factor (AHF)
Method Inheritance Factor (MIF)
Attribute Inheritance Factor (AIF)
Polymorphism Factor (PF)
Coupling Factor (CF)

Each of those metrics refers to a basic structural mechanism of the object-oriented paradigm as encapsulation (MHF and AHF), inheritance (MIF and AIF), polymorphism (PF) and message-passing (CF) and are expressed as quotients. The numerator represents the actual use of one of those mechanisms for a given design. The denominator, acting as a normalizer, represents the hypothetical maximum achievable use for the same mechanism on the same design (i.e. considering the same number of classes and inheritance relations). As a consequence, these metrics:

1. are expressed as percentages, ranging from 0% (no use) to 100% (maximum use);
2. are dimensionless, which avoids the often misleading, subjective or "artificial" units that pervaded the metrics literature with its often "esoteric" flavor.

Being formally defined, the MOOD metrics avoid subjectivity of measurement and thus allow replicability. In other words, different people at different times or places can yield the same values when measuring the same systems. These metrics are also expected to be system size independent. A partial demonstration of this assertion is included below. Size independence allows inter-project comparison, thus fostering cumulative knowledge. The MOOD metrics definitions make no reference to specific language constructs. However, since each language has its own constructs that allow for implementation of OO mechanisms in more or less detail, a binding for relevant languages is needed. Bindings and validation experiments with C++ and Eiffel are currently under way. This ex pected, but yet to be proved, language independence will broaden the applicability of this metric set by allowing comparison of hetero geneous system implementations.

The MOOD metric set enables expression of some recommendations for designers. An Electronic Engineering analogy for representing our design heuristics will be used. Let it be called "the filters metaphor".

Theoretically, a high-pass filter is not expected to affect signal frequencies above a certain value (the cutoff frequency). Below that value, the filter acts as a hindrance for frequency. By analogy, a high-pass heuristic is the one that suggests that there is a lower limit for a given metric. Going below that limit is a hindrance to resulting software quality. For those who do not like thresholds, we may say that the analogy is even more perfect, if we realize that "real" filters do not have them. Indeed their shape is not a step but a curve with a bigger slope at the cutoff zone. Resulting software quality characteristics are also expected to be strongly attenuated (or increased, depending on the direction) as we approach the cutoff values. The reasoning for a band-pass heuristic is similar, except that we have two cutoff zones (a lower and an higher one).

AHF and MHF are a measure of the use of the information hiding concept that is supported by the encapsulation mechanism. Information hiding allows, among other things, to: (i) cope with complexity by looking at complex components such as "black boxes", (ii) reduce "side-effects" provoked by implementation refinement, (iii) support a top-down approach, (iv) test and integrate systems incrementally. For attributes (AHF) we want this mechanism to be used as much as possible. Ideally all attributes would be hidden, thus being only accessed by the corresponding class methods. Very low values for AHF should trigger the designers' attention. The corresponding design heuristic shape is that of a high-pass filter. The number of visible methods is a measure of the class functionality. Increasing the overall functionality will then reduce MHF. However, for implementing that functionality we must adopt a top- down approach, where the abstract interface (visible methods) should only be the tip of the iceberg. In other words, the implementation of the classes interface should be a stepwise decomposition process, where more and more details are added. This decomposition will use hidden methods, thus getting the above mentioned information-hiding benefits and favoring a MHF increase. This apparent contradiction is reconciled if we consider MHF to have values within an interval. A very low MHF would then indicate an insufficiently abstracted imple mentation. Conversely, a high MHF would indicate very little functionality. The design heuristic shape for MHF is then the one of a band-pass filter.

MIF and AIF are measures of inheritance. This is a mechanism for expressing similarity among classes that allows the portrayal of generalization and specialization relations and a simplification of the definition of inheriting classes, by means of reuse. At first sight we might be tempted to think that inheritance should be used extensively. However, the composition of several inheritance relations builds a directed acyclic graph (inheritance hierarchy tree), whose depth and width make understandability and testability quickly fade away. A band-pass filter shape seems appropriate for the corresponding heuristics.

The COF metric is a measure of coupling between classes. Coupling can be due to message-passing among class instances (dynamic coupling) or to semantic association links (static coupling). It has been noted that it is desirable that classes communicate with as few others as possible and even then, that they exchange as little information as possible. Coupling relations increase complexity, reduce encapsulation and potential reuse, and limit understandability and maintainability. Thus, it seems that we should avoid it as much as possible. Very high values of COF should be avoided by designers. However, for a given application, classes must cooperate somehow to deliver some kind of functionality. Therefore, COF is expected to be lower bounded. Accordingly, the design heuristic shape will be the one of a band-pass filter.

Resulting polymorphism potential is measured through the PF metric. Polymorphism arises from inheritance and its use has pros and cons. Allowing binding (usually at run time) of a common message call to one of several classes (in the same hierarchy) is supposed to reduce complexity and to allow refinement of the class taxonomy without side-effects. On the other hand, if we need to debug such a taxonomy, by tracing the control flow, this same polymorphism will make the job harder. This is particularly true if we compare this situation with the procedural counterpart, where for a similar functionality we usually have a series of decision statements for triggering the required operation. We can then state that polymorphism ought to be bounded within a certain range. Naturally, a band-pass filter is the corresponding shape for the respective design heuristic.

FINAL REMARKS

The adoption of the Object-Oriented paradigm is expected to help produce better and cheaper software. The main structural mechanisms of this paradigm, namely, inheritance, encapsulation, information hiding or polymorphism, are the keys to foster reuse and achieve easier maintainability. However, the use of language constructs that support those mechanisms can be more or less intensive, depending mostly on the designer ability. We can then expect rather different quality products to emerge, as well as different productivity gains. Advances in quality and productivity need to be correlated with the use of those constructs. We then need to evaluate this use quantitatively to guide OO design. This set allows comparison of dif ferent systems or different implementations of the same system. Some design heuristics based on a filter metaphor can be derived. It is hoped that those heuristics will be of some help to novice designers. Object-orientation is not "the" silver bullet, but it seems to be the best bullet available today to face the pervasive software crisis. Keeping on the evolution track means we must be able to quantify our improvements. Metrics will help us to achieve this goal.