How to encapsulate software (Part 1).

Edmund Kirwan

www.EdmundKirwan.com [*]

Abstract

This paper describes how to encapsulate software using encapsulation theory.

Keywords

Encapsulation theory, encapsulation, potential coupling.

1. Introduction

This paper is not going to turn you into a super-programmer.

It is not going bring you overnight riches and success.

It is not going to whiten your teeth.

This paper is just an idea about encapsulation.

There are a lot of ideas about encapsulation out there. The idea developed here in no way strives to be better than any competitor and instead attempts to establish the goal of encapsulation based on historical experience and establish the rules of encapsulation based on logical reasoning.

If your idea of encapsulation differs from that presented here, fine: black-clad, automatic-weaponed militia will not crash through your bedroom door at 2am. If all this paper does is satisfy you that your encapsulation is better than this one then this paper will have achieved something.

Java is used where actual programming is referenced, but any language that supports basic encapsulation is applicable.

For reasons which may never become clear, an odd-sounding caveat must be issued in advance: this paper considers sets of absolute information hiding only. With that out of the way, let us begin by examining what software actually is.

2. Initial distinctions

Every software product can be viewed as two, intertwined elements: a static specification (the source code) and a run-time behaviour (the program doing, hopefully, whatever it was specified to do).

Encapsulation concerns only the former, not the latter. Encapsulation is for people, not processors. Encapsulation and run-time behaviour are, to use that nerdiest of words, "Orthogonal." From the perspective of encapsulation, a program may be a dilapidated, creaking, wooden shack scheduled for dismantlement yet may run with the precision and roaring swiftness of a heat-seeking missile; alternately, a program may be

[*] © Edmund Kirwan 2009. Revision 1.4 November 20th 2009. (Revision 1.0 written August 17th 2009.) All entities may republish, but not for profit, all or part of this material provided reference is made to the author and the title of this paper. The latest version of this paper is available at [1].

beautifully, robustly encapsulated yet crash in the lightest summer breeze.[1]

Encapsulation itself, furthermore, cracks along the lines of the two great forces that animate it: one being qualitative, the other quantitative.

The qualitative derives its influence from the natural language semantics of the problem that the program is trying to solve. Basically, if a program is composed of two packages, a *gui* package and a *calculation* package, and you are tasked with adding a new pop-up window to the program, then, without any other information, you would be right to assume that your new functionality would go into the *gui* package. This just makes sense: a pop-up window has more in common with GUI functionality than with calculation functionality.

This mapping, however, from natural language semantics to source code is not automatic. Machines cannot do it. It is subjective in nature. It is here, if anywhere, that art makes its last stand against the panzers of computer science.

Quantitative encapsulation, however, is one that can be assessed through objective measurement. By feeding a program into an analyser, for example, the analyser can identify whether the program is quantitatively well- or poorly-encapsulated. There is no room for subjectivity here. Here be science.

To achieve a quantitative encapsulation requires a mathematical theory of encapsulation.

This paper presents such a theory of encapsulation.

3. So what is encapsulation?

We need a definition.

It would be nice if there were an international, standard definition of encapsulation upon which we could build our mathematical model, but for this to be the case would require an international organisation for standardization, and this international organisation for standardization would have had to have gone to the bother of defining, "Encapsulation," in a meaningful way.

Well, fortunately, there is an international organisation for standardization: it's called the International Organisation for Standardization, and these good people have officially defined encapsulation as being (see [5]):

Encapsulation: *(drum roll) the property that the information contained in an object is accessible only through interactions at the interfaces supported by the object (cymbal crash!).* [Sound effects added by the author.]

You might think that the good people at ISO (as they are known, from a Greek term, apparently) would only bother to issue such a definition if they believed it to be of some merit; and you'd be right. Other great minds have reached the same conclusion.

The essence of the above definition is that something is separated from something else: in this case, the object's information is separated from those wishing to access it, the object's clients. This is a reiteration of a foundational computer concept, the separation of concerns, first penned in 1974 by that great, Dutch computer scientist, Edsger W. Dijkstra, who, while reflecting on the exasperating limitations of the human mind, described it thus (see [6]):

Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency, all

1 Note that the source code can have several stakeholders. The programmer is a stakeholder who cares about encapsulation. The unit tester is a stakeholder who cares about ensuring that behaviour does not change unintentionally; the unit tester does not care for encapsulation, and encapsulation may even impede the unit tester's work. For this reason, unit testers often copy the source code - removing all accessors and thus removing all encapsulation - and unit test the copy, rather than the original.

the time knowing that one is occupying oneself only with one of the aspects. We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. In another mood we may ask ourselves whether, and if so: why, the program is desirable. But nothing is gained --on the contrary!-- by tackling these various aspects simultaneously. It is what I sometimes have called "the separation of concerns", which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of.

Encapsulation allows clients of the objects to concentrate on the object's interfaces, rather than on the information directly, so that if the interfaces can be defined to be somehow less burdensome than the information itself, then the clients have fewer problems to tackle.

Two years before Dijkstra, in 1972, another intellectual tower of computing wrote along similar lines. Canadian David Parnas described how software should be designed using the technique of, "Information hiding," (see [7]):

We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others.

The point being that if functionality that might change is hidden, then changing this functionality will have a less burdensome impact than would be the case had that functionality been accessible to all other modules.

Reading the encapsulation definition above, we see that if an object's information is accessible only via the object's interfaces, then the object itself must manifest two fundamentally different phenomena: that which is accessible to clients (the interfaces) and that which is not accessible to clients (the contained information). It is trivial - indeed obvious - that the ISO and Parnas definitions can be married by the proposal that the information which is not accessible to clients is information-hidden in the Parnasian sense. We make precisely this proposal here.

The nature of the information that Parnas suggests to hide is also noteworthy. This is information is not mere data but, "Design decisions," which may be embodied in not only data but also in functional behaviour. Methods that might change should be hidden just as much as data that might change. Entire classes should be hidden just as much as methods. Today, people talk about, "Data-encapsulation," yet this must only be a subcategory of encapsulation as defined here (which would perhaps more fully and more awkwardly be described as, "Behaviour and data encapsulation"); the rules we derive should apply to encapsulation and therefore to data-encapsulation as a matter of logical extension.

History, nonetheless, has served us a piping hot helping of obvious benefit from this recipe of encapsulation, but we do not yet have enough to begin our mathematical foray.

One ingredient is still missing.

4. Potential coupling

For our encapsulation to be quantitative, we need something to measure. For this, we must once more peer into the past.

Back in 1974 again (was it the flares or the glam rock that made people so intelligent back then?), Messrs Stevens, Myers and Constantine produced a paper in which were written two of the most important, consecutive sentences in the history of computing literature (see [8]):

The fewer and simpler the connections between modules, the easier it is to understand each module without reference to other modules. Minimizing connections between modules also minimises the paths along which changes and errors can propagate into other parts of the system, thus eliminating disastrous, "Ripple effects," where changes in one part causes errors in another, necessitating additional changes elsewhere, giving rise to new errors, etc.

To describe these connections, the authors invented two terms: coupling and cohesion.

Coupling is the measure of the strength of association established by a connection from one module to another. Cohesion (though illuminated less clearly) is the measure of the strength of association established by a connections within a module. Coupling is inter-module, cohesion intra-module. The paper proceeds to describe how both coupling and cohesion depend on the type of connection and the contexts in which the connections are made. The complexity introduced by defining the various types of connection (and hence coupling) and the various types of context (and hence cohesion) have been of enormous utility to programmers ever since.

Two points, however, must be raised.

Firstly, in our search for a theory, we need simplification. If we are to develop a useful, quantitative theory of encapsulation, we must minimise the number of phenomena to consider. This paper, thus, will abandon the difference between intra-module and inter-module connection, between coupling and cohesion. A new, superseding concept must consider them equivalent, at least in terms of how they are counted. A static software dependency between two classes in a package will count as a single dependency just as will a software dependency between two classes in two different packages.

Secondly, let us return to a key phrase in Parnas's writing.

Parnas writes of design decisions, " … that are likely to change." This statement involves projection. Parnas considers software not just as it is but as it might evolve into the future.

Coupling and cohesion appear to describe software as-is, yet we surely would prefer our theory of encapsulation to contain this predictive aspect so that it might present programmers with more than a description of the program in its current state (of which the programmer will likely be aware) and instead offer a guide through a program's future development.

This paper will thus define the concept of potential coupling in an attempt to unite the two views.

Potential coupling is the maximum possible number of source code dependencies within a program. These dependencies may not yet be realised, but they are allowed given the current information-hidden and non-information-hidden elements of a program. Potential coupling therefore describes the landscape into which a program may evolve in terms of the possible dependencies it may develop.

It is upon potential coupling that we shall base our mathematical treatment; indeed, encapsulation theory is defined as the study of the potential coupling of sets.

Yet now that we have this potential coupling - this maximum possible number of dependencies - of our program, what of it? Are they any hints from this array of great minds concerning what we should do with our potential coupling?

Yes, there are.

If we follow ISO's advice, making information inaccessible within objects will reduce the potential coupling of a program.

If we follow Dijkstra's advice, separating concerns into that which is accessible and that which is not will reduce the potential coupling of a program.

If we follow Parnas's advice, hiding design decisions within a module from all other modules will reduce the potential coupling of a program.

If we follow Messrs Stevens's, Myers's and Constantine's advice, minimizing the connections within and between modules will reduce the potential coupling of a program.

It seems our goal has been set for us: we wish to use encapsulation theory to reduce the potential coupling of a program.

Great.

All we need now is a way to count potential coupling.

And a way to reduce it.

5. The model: a brief overview

Encapsulation theory describes the building of a model of a program and the analysis of that model to conclude various changes that can be fed back into the program to reduce its potential coupling.

The model consists of the following elements and their equivalent program language constructs:

• Elements. An element is a model of a class.

• Ordered pairs. An ordered pair is a model of a static, source code dependency, the pair being of the form *(a, b)* where *a* and *b* are different classes.

• Disjoint primary sets. A disjoint primary set is a model of a package.

• Sets. A set is a model of an entire program containing packages represented by disjoint primary sets.

And that's it.

Every model we build of a program will contain only those four elements.

an element, however, can be one of two types:

• Information-hidden. This is a class that is declared with the default accessor (also informally called, "Package-private").

• Violational. This is a class that is declared public in a package, and so is accessible outside that package. (It is, "Violational," because it violates information-hiding.)

A model, by definition, captures the essential aspects of a system while disregarding the inessential. A model of the above elements discards a huge amount of information about the original program, notably its run-time behaviour. As we have seen, however, encapsulation is not concerned with run-time behaviour.

Other omissions also deserve note. There is no place in the above model for selective access to classes from outside package: as the Heidi principle mercilessly states, "You're either in or you're out." The `protected` accessor, therefore, cannot be modelled. Usually, there are so few protected classes that this is not a concern, but where a program uses this accessor significantly, encapsulation theory will be a poor choice of analysis tool.

Java's reflection is also not modelled; reflection can be used to work around information-hiding. Again, a program using reflection extensively will not be modelled accurately.

These points notwithstanding, the process of modelling a program is then:

1. Map all packages to disjoint primary sets.

2. Map all public classes to violational elements within those disjoint primary sets.

3. Map all package-private classes to information-hidden elements within those disjoint primary sets.

4. Form ordered pairs between all elements within each disjoint primary set.

5. Form ordered pairs from each element in each disjoint primary set to every violational element in every other disjoint primary set.

6. Count the total number of ordered pairs.

That point *6* gives us what we need: a way to objectively count potential coupling.

Not, of course, that you should do this mapping yourself, manually, on an architect's desk, with a pencil and paper, and an eraser, and an ocean of coffee. The whole point of a quantitative encapsulation is that it is machine verifiable: let a program struggle with the heavy lifting. The very first such program is, "Fractality," and it's free to download at [2]. Just point it at your Java code and it will do the rest.

Now that we have counted our potential coupling, we just need to know the rules for reducing it.

6. Rules of encapsulation

A theory is as useful as its application is trivial. Encapsulation theory, fortunately, gives rise to just four rules describing how to reduce potential coupling. These are:

1. Scope minimally.
2. Group isoledensally.
3. Hide uniformly.
4. Violate non-uniformly.

Let us take a quick look at each rule and see how potential coupling is reduced.

Rule 1: Scope minimally

The granddaddy of all encapsulation rules.

The, "Scope," here refers to accessibility of a class. This rule says, basically, that if there are no dependencies on a class from outside its package, then that class should be declared package-private.

From the point of view of potential coupling, actual dependencies are unimportant: all that matters are potential dependencies. As soon as a class is declared public, potential dependencies form towards it from every other class in the system. They may not actually form today, maybe not tomorrow, but soon, and for the rest of your life.

On a pragmatic level, the probability of an inter-package dependency forming on a class is higher when that class is public; and if a class is not intended to receive inter-package dependencies then there is no benefit in making it public.

A public class on which there are no inter-package dependencies is like a window left open in a bank at night: it doesn't mean that the bank will be burgled, but it doesn't help.

Rule 2: Group isoledensally

Don't let the scary word put you off: this one's oddly satisfying once you get the hang of it.

This rule tells you how many packages your program should have.

Take any program. To find out how many packages it should have, count the number classes in it; call this *n*. Count the number of packages in it; call this *r*. Count the number of public classes in it; call this *p*.

Then make a two-step calculation.

Firstly, calculate the specific violational density, *d*. This is just the number of public classes per package, or:

$$d=\frac{p}{r}$$

Secondly, calculate the square root of the number of classes divided by the specific violational density, this is the number of packages that the system should have, denoted by r_{ild} where:

$$r_{ild}=\sqrt{\frac{n}{d}}$$

Why should anyone bother doing all this? Because this gives the number of packages that will minimise the potential coupling of a uniformly-distributed set given these particular parameters. Even if your program is not uniformly-distributed, this provides a fine target to aim for.

Remember, this is quantitative encapsulation, and it must always been be balanced by semantic encapsulation: if you have *10* packages, all of which make perfect sense qualitatively, but the above equation tells you that you should have *8* packages, then it's probably safe to obey your semantic instincts. The goal here is not to reach r_{ild} at all costs, but to approach within qualitative constraints.

("Isoledensal," incidentally, just means that the minimization conserves a violation density, in this case the specific violation density. The choice is arbitrary; there are all sorts of other parameters that we might have conserved but few match the potential-coupling-reducing prowess of this choice.)

Rule 3: Hide uniformly

This rule states that package-private classes should be distributed evenly throughout all packages. If you have *50* package-private classes and *10* packages then you should have *5* package-private classes per package.

Such uniformity has been found to reduce potential coupling in most cases, and it's not difficult to see why. The number of potential dependencies within a package rises approximately with the square of the number of classes within the package; so if the number of classes in a package are doubled, the potential coupling rises by *4*.

Again, this is a goal to be approached rather than ruthlessly pursued. More pragmatically it takes the form of, "Avoid mixing small and gigantic packages." Few programmers would instinctively divide a system of 1000 classes into two packages, one containing *7* classes, the other *993*. Rule *3* merely supports this instinct.

Rule 4: Violate non-uniformly

This rule only barely makes it into the short-list as, purely quantitatively, it has relatively little impact.

It states that public classes, unlike the uniformly-distributed package-private classes of Rule *3*, should be clustered together as tightly as possible, ideally with all the free public classes put into one package. (As each package must contain one public class - or else the package is not contactable by the rest of the system - then one public class per package is not free to move, whereas the rest are.)

This rule is based on a conjecture which presents the form of a set with the lowest possible potential coupling (lower even than an entirely uniformly-distributed set), and clearly such dogged pursuit of minimisation would not suit most Java software written today. So this rule is really only for hard-core, encapsulation purists and they are few. Given the state of encapsulation today, this rule is rather like recommending watching the Super Bowl through an electron microscope.

Except in one sense.

Apart from that ideal case, this rule says that potential coupling is lowered when public classes are bunched together even if there is more than one such package: this maps well to the concept of interface repositories used in technologies such as CORBA and OSGI. Interface repositories have uses far beyond encapsulation, so let us conclude here by saying that this rule recommends the use of interface repositories, but the recommendation is light.

There, however, we have it: the four rules of encapsulation.

8. Isoledensal configuration efficiency

Just counting the potential coupling is fine, but when we write programs, it is nice to have a fixed target. If our program has a potential coupling of 10,000 on Monday and a potential coupling of 20,000 on Friday, have we degraded our program's encapsulation or have we introduced sufficient functionality to merit this rise in potential coupling? To answer this question it would be nice to have a simple percentage which could compare how much our program is encapsulated to an idealised, perfect program of the same size. Thus, if our program is 60% on Monday and 80% on Friday, then we know we have increased our program's level of encapsulation and that our program has edged closer to the (unattainable) goal of 100% perfect encapsulation.

Encapsulation theory gives us precisely such a metric: the isoledensal configuration efficiency.

This configuration efficiency is a number between *0* and *1*, *0* being a completely unencapsulated program (in which every class is public) and *1* being a perfectly encapsulated program. If we multiply it by *100*, then the configuration efficiency takes the form of a percentage and we can use this percentage to track our program's encapsulation irrespective of its size.

It goes without saying, of course, that for a given number of classes when we reduce potential coupling we increase the configuration efficiency.

9. The set stack

Although we have only discussed classes encapsulated within packages, the same rules apply to all such things that can be mapped as elements in disjoint primary sets; specifically, methods can be encapsulated in classes with precisely the same rules as those that govern the encapsulation of classes within packages.

10. And now for the bad news ...

If you've read this far, you may have wasted your time.

Or so it seems, at least.

A survey analysed over sixty open-source, Java projects with the aim of recording the configuration efficiency to see how well encapsulated the programs were. The results speak for themselves.

The vast majority of the programs exhibited a configuration efficiency of between 20% and 30%. None of the programs reached above a 50% efficiency.

For some reason, most of the programs were written with 75% public classes. In other words, for every information-hidden class written, three public classes were written. Rule *1*, "Scope minimally," went A.W.O.L., with an enormous number of public classes receiving no dependencies from outside their packages.

There are only two possible reasons for this state of affairs:

1. Encapsulation theory is horribly wrong.
2. Or something else.

The zeroth rule of all advice lists is relevant here: think for yourself.

If you think four rules of encapsulation sound reasonable, if you can see how they might confer advantage upon your programs, then by all means use them.

If you think they sound ridiculous, if you don't get out of bed in the morning without maximising something's scope, if no one is going to tell you how many packages mathematically minimise anything thank you very much, then by all means ignore them.

But have a reason for whatever you do. Know why you're doing it. Defend it when challenged.

Thank you for taking the time to read this paper.

11. Further reading

If you would like to know more about encapsulation theory, you can find the original series of papers which introduced the concept freely available on the author's website (see title above). Each paper follows the same format: the first half is a simple description of a particular idea, the second half is string of theorems from which the particular idea was deduced. Those not mathematically-minded are more than welcome to ignore the second halves entirely.

The first paper is at [3] and describes potential coupling with numerous examples. No other paper will make much sense unless you read this one first.

8. References

[1] "How to encapsulate software (Part 1)," Ed Kirwan, www.EdmundKirwan.com/pub/paper7.pdf
[2] Fractality, Ed Kirwan, www.EdmundKirwan.com/frac.jar
[3] "Encapsulation theory fundamentals," Ed Kirwan, www.EdmundKirwan.com/pub/paper1.pdf
[4] "Encapsulation theory: L3 potential coupling," Ed Kirwan, www.EdmundKirwan.com/pub/paper6.pdf
[5] "Information technology - Open Distributed Processing," ISO/IEC 10746, 1998.

[6] "On the role of scientific thought," Dijkstra, Edsger W (1982), in *Selected writings on Computing: A Personal Perspective*, New York, NY, USA: Springer-Verlag New York, Inc., pp. 60–66.

[7] "On the Criteria To Be Used in Decomposing Systems into Modules," D. L. Parnas, 1972.
[8] "Structured design," W. P. Stevens, G. J. Myers, and L. L. Constantine SD 13-2 p. 115ff, 1974.