OO Design Quality Metrics

An Analysis of Dependencies

By Robert Martin
October 28,1994

2080 Cranbrook Road
Green Oaks, IL 60048
Phone: 708.918.1004
Fax: 708.918.1023
Email: rmartin@oma.com

Abstract

This paper describes a set of metrics that can be used to measure the quality of an object-oriented design in terms of the interdependence between the subsystems of that design. Designs which are highly interdependent tend to be rigid, unreusable and hard to maintain. Yet interdependence is necessary if the subsystems of the design are to collaborate. Thus, some forms of dependency must be desirable, and other forms must be undesirable. This paper proposes a design pattern in which all the dependencies are of the desirable form. Finally, this paper describes a set of metrics that measure the conformance of a design to the desirable pattern.

Introduction

What is it about OO that makes designs more robust, more maintainable, and more reusable? This is a poignant question since there have been many recent examples where applications designed with so-called OO methods have turned out not to fulfill those claims. Are these qualities of robustness, maintainability and reusability intrinsic to OOD? If so, why don't all applications designed with OOD have them? If not, then what other characteristics does a object-oriented design require in order to have these desirable qualities?

This paper presents the case that simply using objects to model an application is insufficient to gain robust, maintainable and reusable designs. That there are other attributes of a design that are required. That these attributes are based upon a pattern of interdependencies between the subsystems of the design that support communications within the design, isolate reusable elements from non-reusable elements, and block the propagation of change due to maintenance.

Moreover, this paper presents a set of metrics that can be easily applied to a design, and that measures the conformance of that design to the desired pattern of dependencies. These metrics are "Design Quality" metrics. They provide information to the designers regarding the ability of their design to survive change, or to be reused.

Dependency

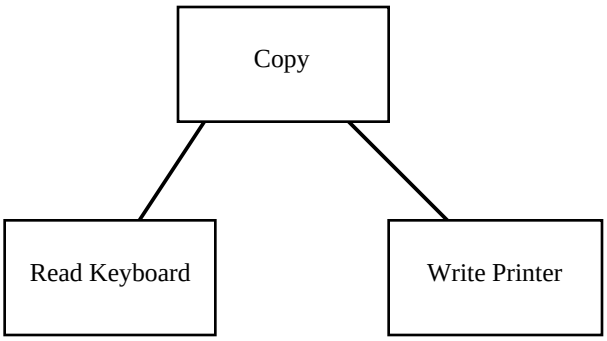
What is it that makes a design rigid, fragile and difficult to reuse. It is the interdependence of the subsystems within that design. A design is rigid if it cannot be easily changed. Such rigidity is due to the fact that a single change to heavily interdependent software begins a cascade of changes in dependent modules. When the extent of that cascade of change cannot be predicted by the designers or maintainers the impact of the change cannot be estimated. This makes the cost of the change impossible to estimate. Managers, faced with such unpredictability, become reluctant to authorize changes. Thus the design becomes rigid.

Fragility is the tendency of a program to break in many places when a single change is made. Often the new problems are in areas that have no conceptual relationship with the area that was changed. Such fragility greatly decreases the credibility of the design and maintenance organization. Users and managers are unable to predict the quality of their product. Simple changes to one part of the application lead to failures in other parts that appear to be completely unrelated. Fixing those problems leads to even more problems, and the maintenance process begins to resemble a dog chasing its tail.

A design is difficult to reuse when the desirable parts of the design are highly dependent upon other details which are not desired. Designers tasked with investigating the design to see if it can be reused in a different application may be impressed with how well the design would do in the new application. However if the design is highly interdependent, then those designers will also be daunted by the amount of work necessary to separate the desirable portion of the design from the other portions of the design that are undesirable. In most cases, such designs are not reused because the cost of the separation is deemed to be higher than the cost of redevelopment of the design.

Example: the "Copy" program.

A simple example may help to make this point. Consider a simple program which is charged with the task of copying characters typed on a keyboard to a printer. Assume, furthermore, that the implementation platform does not have an operating system that supports device independence. then we might conceive of a structure for this program that looks like this:



There are three modules. The "Copy" module calls the other two. One can easily imagine a loop within the "Copy" module. The body of that loop calls the "Read Keyboard" module to fetch a character from the keyboard, it then sends that character to the "Write Printer" module which prints the character.

The two low level modules are nicely reusable. They can be used in many other programs to gain access to the keyboard and the printer. This is the same kind of reusability that we gain from

subroutine libraries.

However the "Copy" module is not reusable in any context which does not involve a keyboard or a printer. This is a shame since the *intelligence* of the system is maintained in this module. It is the "Copy" module that encapsulates a very interesting policy that we would like to reuse.

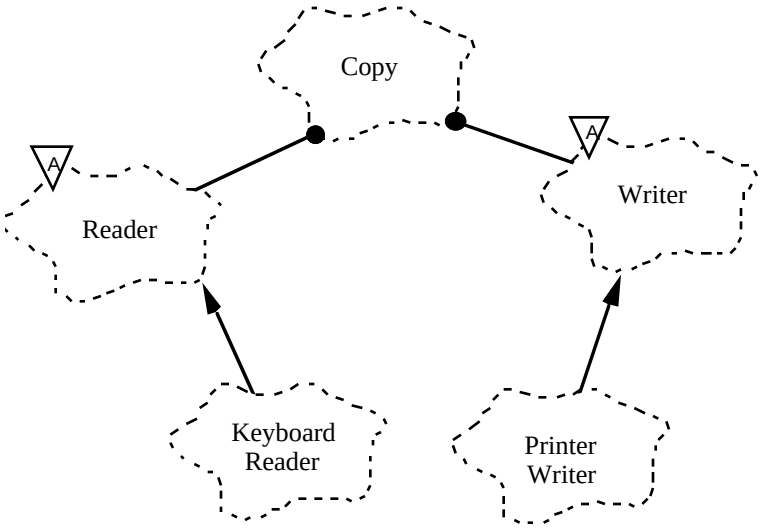
For example, consider a new program that copies keyboard characters to a disk file. Certainly we would like to reuse the "Copy" module since it encapsulates the high level policy that we need. i.e. it knows how to copy characters from a source to a sink. Unfortunately, the "Copy" module is dependent upon the "Write Printer" module, and so cannot be reused in the new context.

We could certainly modify the "Copy" module to give it the new desired functionality. We could add an 'if' statement to its policy and have it select between the "Write Printer" module and the "Write Disk" module depending upon some kind of flag. However this adds new interdependencies to the system. As time goes on, and more and more devices must participate in the copy program, the "Copy" module will be littered with if/else statements and will be dependent upon many lower level modules. It will eventually become rigid and fragile.

Inverting Dependencies with OOD

One way to characterize the problem above is to notice that the module that contains the high level policy, i.e. the "Copy" module, is dependent upon its details. If we could find a way to make this module independent of the details that it controls, then we could reuse it freely. We could produce other programs which used this module to copy characters from any input device to any output device. OOD gives us a mechanisms for performing this dependency inversion.

Consider the following simple class diagram:



Here we have a "Copy" class which contains an abstract "Reader" class and an abstract "Writer" class. One can easily imagine a loop within the "Copy" class which gets characters from its "Reader" and sends them to its "Writer". Yet this "Copy" class does not depend upon the "Keyboard Reader" nor the "Printer Writer" at all. Thus the dependencies have been inverted. Now the "Copy" class depends upon abstractions, and the detailed readers and writers depend

upon the same abstractions.

Now we can reuse the "Copy" class, independently of the "Keyboard Reader" and the "Printer Writer". We can invent new kinds of "Reader" and "Writer" derivatives which we can supply to the "Copy" class. Moreover, no matter how many kinds of "Readers" and "Writers" are created, "Copy" will depend upon none of them. There will be no interdependencies to make the program fragile or rigid.

Good Dependencies

What makes the OO version of the copy program robust, maintainable and reusable? It is its lack of interdependencies. Yet it does have *some* dependencies; and those dependencies do not interfere with those desirable qualities. Why not? Because the targets of those dependencies are extremely stable; i.e. they are unlikely to change.

Consider the nature of the "Reader" and "Writer" classes. In C++ they could be represented as follows:

```
class Writer {public: virtual void Write(char) = 0;};
class Reader {public: virtual char Read() = 0;};
```

These two classes are very unlikely to change. What forces exist that would cause them to change? Certainly we could imagine some if we stretched our thinking a bit. But in the normal course of events, these classes are extremely stable.

Thus, there are very few forces that could cause "Copy" to be changed. "Copy" is an example of the "Open/Closed" principle at work. "Copy" is open to be extended since we can create new versions of "Readers" and "Writers" for it to drive. Yet "Copy" is closed for modification since we do not have to modify it to achieve those extensions.

Thus, we can say that a "Good Dependency" is a dependency upon something that is very stable. The more stable the target of the dependency, the more "Good" the dependency is. By the same token a "Bad Dependency" is a dependency upon something that is instable. The more instable the target of the dependency is, the more "Bad" the dependency is.

Stability

How does one achieve stability? Why, for example, are "Reader" and "Writer" so stable? Consider again the forces that could make them change. They depend upon nothing at all, so a change from a dependee cannot ripple up to them and cause them to change. I call this characteristic "Independence". Independent classes are classes which do not depend upon anything else.

Another reason that "Reader" and "Writer" are stable is that they are depended upon by many other classes. "Copy", "KeyboardReader" and "KeyboardWriter" among them. In fact, the more varieties of "Reader" and "Writer" exist, the more dependents these classes have. The more dependents they have, the harder it is to make changes to them. If we were to change "Reader" or "Writer" we would have to change all the other classes that depended upon them. Thus, there is a great deal of force *preventing* us from changing these classes, and enhancing their stability.

I call classes that are heavily depended upon, "Responsible". Responsible classes tend to be stable because any change has a large impact.

The most stable classes of all, are classes that are both Independent and Responsible. Such classes have no reason to change, and lots of reasons not to change.

Class Categories: the granule of Reuse and Release

It is seldom that a class can be reused in isolation. "Copy" provides a good example. It must be reused with the abstract "Reader" and "Writer" classes. It is generally true that a class has a set of collaborating classes from which it cannot eassily be separated. In order to reuse such classes, one must reuse the entire group. Such a group of classes is highly cohesive, and Booch calls them a "Class Category".

A Class Category (hereinafter referred to as simply a category) is a group of highly cohesive classes that obey the following three rules:

1. The classes within a category are closed together against any force of change. This means that if one class must change, all of the classes within the category are likely to change. If any of the classes are open to a certain kind of change, they are all open to that kind of change.

2. The classes within a category are reused together. They are strongly interdependent and cannot be separated from each other. Thus if any attempt is made to reuse one class within the category, all the other classes must be reused with it.

3. The classes within a category share some common function or achieve some common goal.

These three rules are listed in order of their importance. Rule 3 can be sacrificed for rule 2 which can, in turn, be sacrificed for rule 1.

If categories are to be reused, they must also be released and given release numbers. If this were not the case, reusers would not be able to rely upon the stability of the reused categories since the authors might change it at any time. Thus the authors must provide releases of their categories and identify them with release numbers so that reusers can be assured that they can have access to versions of the category that will not be changed.

The dependencies between categories are the ones we want to manage.

Since categories are both the granule of release and reuse, it stands to reason that the dependencies that we wish to manage are the dependencies *between* categories rather than the dependencies within categories. After all, within a category, classes are expected to be highly interdependent. Since all the classes within a category are reused at the same time, and since all classes in a category are closed against the same kind of changes, the interdependence between them cannot do much harm.

Thus, we can move our discussion of dependency up a level, and discuss the "Independence", "Responsibility" and "Stability" of categories instead of classes. The categories with the highest stability are categories which are both independent and highly responsible. And dependencies upon stable categories are "good" dependencies.

Dependency Metrics

The responsibility, independence and stability of a category can be measured by counting the

dependencies that interact with that category. Three metrics have been identified:

Ca : Afferent Couplings : The number of classes outside this category that depend upon classes within this category.

Ce : Efferent Couplings : The number of classes inside this category that depend upon classes outside this categories.

I : Instability : (Ce ÷ (Ca+Ce)) : This metric has the range [0,1]. I=0 indicates a maximally stable category. I=1 indicates a maximally instable category.

Not all categories should be stable

If all the categories in a system were maximally stable, the system would be unchangeable. In fact, we want portions of the design to be flexible enough to withstand significant amount of change. How can a category which is maximally stable (I=0) be flexible enough to withstand change? The answer is to be found in the "Open/Closed" principle. This principle tells us that it is possible and desirable to create classes that are flexible enough to be extended without requiring modification. What kind of classes conform to this principle? Abstract classes.

Consider the "Copy" program again. The "Reader" and "Writer" classes are abstract classes. They are highly stable since they depend upon nothing and are depended upon by "Copy" and all their derivatives. Yet, "Reader" and "Writer" can be extended, without modification, to deal with many different kinds of I/O devices.

Thus, if a category is to be stable, it should also consist of abstract classes so that it can be extended. Stable categories that are extensible are flexible and do not constrain the design.

If stable categories should be highly abstract, one might infer that instable categories should be highly concrete. In fact, this stands to reason. An abstract category *must* have dependents since there must be classes, outside the abstract category, that inherit from it and implement the missing pure interfaces. However, we do not want to encourage dependencies upon instable categories. Thus, instable categories should not be abstract, they should be concrete.

We can define a metric which measures the "abstractness" of a category as follows:

A : Abstractness : (# abstract classes in category ÷ total # of classes in category). This metric range is [0,1]. 0 means concrete and 1 means completely abstract.

The "Main Sequence"

We are now in a position to define the relationship between stability (I) and abstractness (A). We can create a graph with A on the vertical axis and I on the horizontal axis. If we plot the two "good" kinds of categories on this graph, we will find the categories that are maximally stable and abstract at the upper left at (0,1). The categories that are maximally instable and concrete are at the lower right at (1,0).

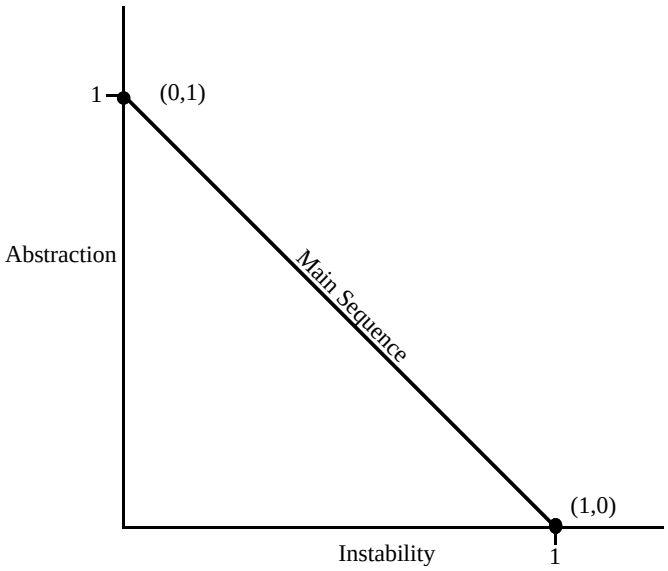
But not all categories can fall into one of these two positions. Categories have *degrees* of abstraction and stability. For example, it is very common that one abstract class derives from another abstract class. The derivative is an abstraction that has a dependency. Thus, though it is maximally abstract, it will not be maximally stable. Its dependency will decrease its stability.

Consider a category with A=0 and I=0. This is a highly stable and concrete category. Such a category is not desirable because it is rigid. It cannot be extended because it is not abstract. And it is very difficult to change because of its stability.

Consider a category with A=1 and I=1. This category is also undesirable (perhaps impossible) because it is maximally abstract and yet has no dependents. It, too, is rigid because the abstractions are impossible to extend.

But what about a category with A=.5 and I=.5? This category is *partially* extensible because it is partially abstract. Moreover, it is partially stable so that the extensions are not subject to maximal instability. Such a category seems "balanced". Its stability is in balance with its abstractness.

Consider again the A-I graph (below). We can draw a line from (0,1) to (1,0). This line represents categories whose abstractness is "balanced" with stability. Because of its similarity to a graph used in astronomy, I call this line the "Main Sequence".



A category that sits on the main sequence is not "too abstract" for its stability, nor is "too instable" for its abstractness. It has the "right" number of concrete and abstract classes in proportion to its efferent and afferent dependencies. Clearly, the most desirable positions for a category to hold are at one of the two endpoints of the main sequence. However, in my experience only about half the categories in a project can have such ideal characteristics. Those other categories have the best characteristics if they are on or close to the main sequence.

Distance from the Main Sequence

This leads us to our last metric. If it is desirable for categories to be on or close to the main sequence, we can create a metric which measures how far away a category is from this ideal.

D : Distance : $|(A+I-1)\div 2|$: The perpendicular distance of a category from the main sequence. This metric ranges from [0,~0.707]. (One can normalize this metric to range between [0,1] by using the simpler form $|(A+I-1)|$. I call this metric Dn).

Given this metric, a design can be analyzed for its overall conformance to the main sequence. The D metric for each category can be calculated. Any category that has a D value that is not near

zero can be reexamined and restructured. In fact, this kind of analysis have been a great aid to the author in helping to define categories that are more reusable, and less sensitive to change.

Statistical analysis of a design is also possible. One can calculate the mean and variance of all the D metrics within a design. One would expect a conformant design to have a mean and variance which were close to zero. The variance can be used to establish "control limits" which can identify categories that are "exceptional" in comparison to all the others.

Conclusion and Caveat

The metrics described in this paper measure the conformance of a design to a pattern of dependency and abstraction which the author feels is a "good" pattern. Experience has shown that certain dependencies are good and others are bad. This pattern reflects that experience. However, a metric is not a god; it is merely a measurement against an arbitrary standard. It is certainly possible that the standard chosen in this paper is appropriate only for certain applications and is not appropriate for others. It may also be that there are far better metrics that can be used to measure the quality of a design.

Thus, I would deeply regret it if anybody suddenly decided that all their designs must unconditionally be conformant to "The Martin Metrics". I hope that designers will experiment with them, find out what is good and what is bad about them, and then communicate their findings to the rest of us.