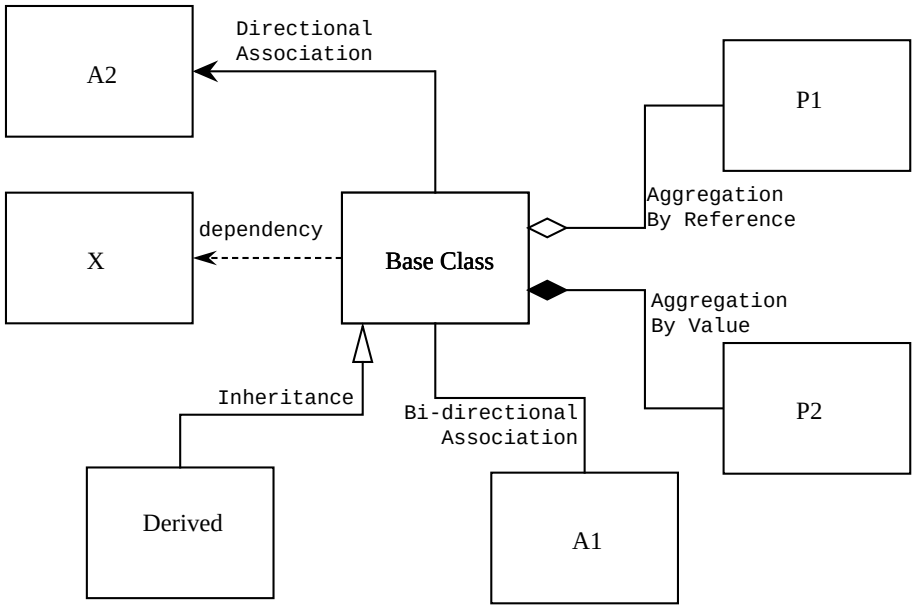
Stability

This is the sixth of my *Engineering Notebook* columns for *The C++ Report*. The articles that appear in this column focus on the use of C++ and OOD, and address issues of software engineering. I strive for articles that are pragmatic and directly useful to the software engineer in the trenches. In these articles I make use of the new *Unified Modeling Language* (UML Version 0.91) for documenting object oriented designs. The sidebar provides a very brief lexicon of this notation.

Sidebar
UML 0.9 Lexicon



Introduction

This article continues the discussion of principles that govern the structure of large C++ aplications. These principles are most appropriate for applications that exceed 50,000 lines of C++ and require a team of engineers to write

This article describes a set of principles and metrics that can be used to measure the quality of an object-oriented design in terms of the interdependence between the packages

of that design. Designs which are highly interdependent tend to be rigid, unreusable and hard to maintain. Yet interdependence is necessary if the packages of the design are to collaborate. Thus, some forms of dependency must be desirable, and other forms must be undesirable. This article proposes a design pattern in which all the dependencies are of the desirable form. Finally, this article describes a set of metrics that measure the conformance of a design to the desirable pattern.

The principles and metrics discussed in this article have to do with *stability.* In significant ways, stability is at the very heart of all software design. When we design software, we strive to make it stable in the presence of change. Indeed, this is the goal of the very first of the principles we discussed: The Open Closed Principle (OCP). In this article we will see how the concept of stability impacts the relationships between packages in the large scale structure of an application.

Reprise

But before we begin to unravel the thread of stability, a brief reprise of what has gone before in this column is in order. Over the last year, we have discussed 8 principles of object-oriented design. Here they are, in order.

1. *The Open Closed Principle.* (OPC) January, 1996. This article discussed the notion that a software module that is designed to be reusable, maintainable and robust must be extensible without requiring change. Such modules can be created in C++ by using abstract classes. The algorithms embedded in those classes make use of pure virtual functions and can therefore be extended by deriving concrete classes that implement those pure virtual function in different ways. The net result is a set of functions written in abstract classes that can be reused in different detailed contexts and are not affected by changes to those contexts.
2. *The Liskov Substitution Principle*. (LSP) March, 1996. Sometimes known as "Design by Contract". This principle describes a system of constraints for the use of public inheritance in C++. The principle says that any function which uses a base class must not be confused when a derived class is substituted for the base class. This article showed how *difficult* this principle is to conform to, and described some of the subtle traps that the software designer can get into that affect reusability and maintainability.
3. *The Dependency Inversion Principle*. (DIP) June, 1996. This principle describes the overall structure of a well designed object-oriented application. The principle states that the modules that implement high level policy should not depend upon the modules that implement low level details. Rather both high level policy and low level details should depend upon abstractions. When this principle is adhered to, both the high level policy modules, and the low level detail modules will be reusable and maintainable.

4. *The Interface Segregation Principle*. (ISP) Aug, 1996. This principle deals with the disadvantages of “fat” interfaces. Classes that have “fat” interfaces are classes whose interfaces are not cohesive. In other words, the interfaces of the class can be broken up into groups of member functions. Each group serves a different set of clients. Thus some clients use one group of member functions, and other clients use the other groups.

   The ISP acknowledges that there are objects that require non-cohesive interfaces; however it suggests that clients should not know about them as a single class. Instead, clients should know about abstract base classes that have cohesive interfaces; and which are multiply inherited into the concrete class that describes the non-cohesive object.

5. *The Reuse/Release Equivalency Principle* (REP) Nov/Dec, 1996 The granule of reuse is the granule of release. Only components that are released through a tracking system can be effectively reused. This granule is what UML 9.1 refers to as a package.

6. *The Common Reuse Principle* (CRP) Nov/Dec, 1996. The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all.

7. *The Common Closure Principle* (CCP) Nov/Dec, 1996. The classes in a package should be closed together against the same kinds of changes. A change that affects a package affects all the classes in that package.

8. *The Acyclic Dependencies Principle* (ADP) Nov/Dec, 1996.The dependency structure between packages must be a Directed Acyclic Graph (DAG). That is, there must be no cycles in the dependency structure.

Stability and Depencency

The June, 1996 issue of the C++ report discussed the Dependency Inversion Principle (DIP). As a preface to that discussion we talked about what it is that makes a design rigid, fragile and difficult to reuse. We said it was 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 packages. 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

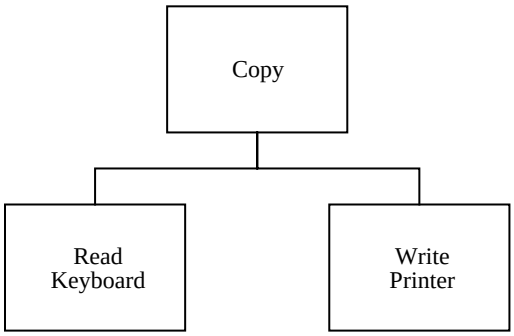
maintenance organization. Users and managers are unable to predict the quality of their product. Simple changes to one part of the application lead to failures in other parts that appear to be completely unrelated. Fixing those problems leads to even more problems, and the maintenance process begins 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.

The “Copy” program...once again.

To illustrate this point, lets once again look at the “Copy” program from the DIP article.

Recall that this simple program is charged with the task of copying characters typed on a keyboard to a printer, and that the implementation platform does not have an operating system that supports device independence. Recall also that we designed this program using “Structured Design”. Here is what we came up with.



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.

The “Copy” module looks like this.

```
void Copy()
{
  int c;
  while ((c = ReadKeyboard()) != EOF)
    WritePrinter(c);
}
```

Note that this 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.

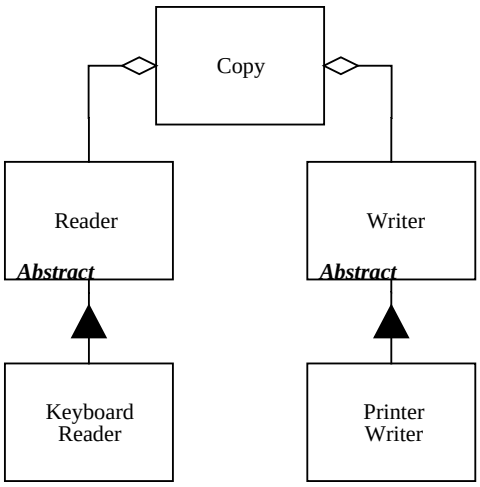
For example, consider a new program that copies keyboard characters to a disk file. Certainly we would like to reuse the "Copy" module since it encapsulates the high level policy that we need. i.e. it knows how to copy characters from a source to a sink. Unfortunately, the "Copy" module is dependent 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”.

```
class Reader
{
  public:
    virtual int Read() = 0;
};

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

void Copy(Reader& r, Writer& w)
{
  int c;
  while((c=r.Read()) != EOF)
    w.Write(c);
}
```

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. This is the essense of the DIP.

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 unlikely to change; they are non-volatile.

Consider the nature of the "Reader" and "Writer" classes shown above. 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 have very low volatility.

Since "Copy" depends upon modules that are non-volatile, 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 with low volatility. The less volatile 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 volatile. The more volatile the target of the dependency is, the more "Bad" the dependency is.

Stability

Unfortunately, the volatility of a module is a difficult thing to understand. Volatility depends upon all kinds of factors. For example, some programs print their version numbers when asked. The module that contains that version number is completely volatile. It is changed every time the application is released. On the other hand, other modules are changed far less often.

Volatility also depends upon market pressures and the whims of the customers. Whether or not a module is likely to change depends upon whether that module happens to contain something that a customer is going to want to change. These things are hard to predict.

However, there is one factor that influences volatility that is very easy to measure. I call this factor *stability.*

The classic definition of the word stability is:"Not easily moved."[1] This is the definition that we will be using in this article. That is, stability is not a measure of the likelihood that a module will change; rather it is a measure of the difficulty in changing a module.

Clearly, modules that are more difficult to change, are going to be less volatile. The harder the module is to change, i.e. the more stable it is, the less volatile it will be.

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.

The Stable Dependencies Principle (SDP)

> ***THE DEPENDENCIES BETWEEN PACKAGES IN A DESIGN SHOULD BE IN THE DIRECTION OF THE STABILITY OF THE PACKAGES. A PACKAGE SHOULD ONLY DEPEND UPON PACKAGES THAT ARE MORE STABLE THAT IT IS.***

Designs cannot be completely static. Some volatility is necessary if the design is to be maintained. We accomplish this by conforming to the Common Closure Principle (CCP). By using this principle we create packages that are sensative to certain kinds of changes. These packages are *designed* to be volatile. We *expect* them to change.

Any package that we expect to be volatile should not be depended upon by a package that is difficult to change! Otherwise the volatile package will also be difficult to change.

By conforming to the SDP, we ensure that modules that are designed to be instable (i.e. easy to change) are not depended upon by modules that are more stable (i.e. harder to change) than they are.

---

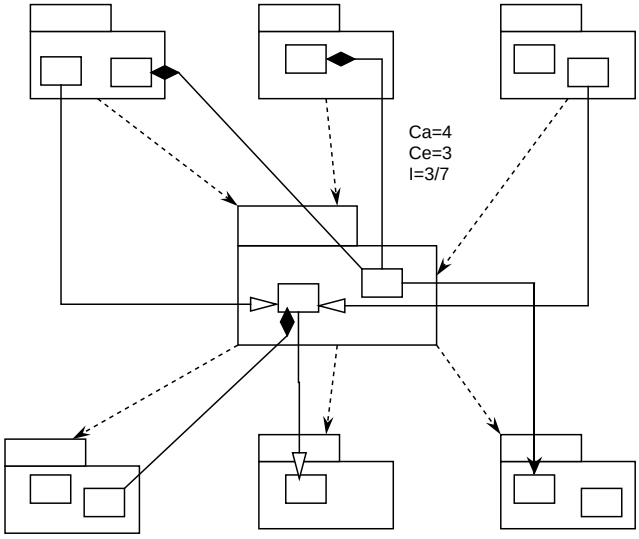
1. Websters Third New International Dictionary

Stability Metrics

How can we measure the stability of a package? One way is to count the number of dependencies that enter and leave that package. These counts will allow us to calculate the *positional* stability of the package.

- Ca : Afferent Couplings : The number of classes outside this package that depend upon classes within this package.
- Ce : Efferent Couplings : The number of classes inside this package that depend upon classes outside this package.
- I : Instability : (Ce ÷ (Ca+Ce)) : This metric has the range [0,1]. I=0 indicates a maximally stable package. I=1 indicates a maximally instable package.

The Ca and Ce metrics are calculated by counting the number of *classes* outside the package in question that have dependencies with the classes inside the package in question. Consider the following example:



The dashed arrows between the package represent package dependencies. The relationships between the classes of those packages show how those dependencies are actually implemented. There are inheritance, aggregation, and association relationships.

Now, lets say we want to calculate the stability of the package in the center of the diagram. We find that there are four classes outside that package that have relationships that target the classes inside it. Thus Ca=4. Moreover, there are three classes outside the central package that are the targets of relationships involving classes inside the central pakcage. Thus, Ce=3, and I=3/7.

In C++, these dependencies are typically represented by `#include` statements. Indeed, the I metric is easiest to calculate when you have organized your source code such that there is one class in each source file.

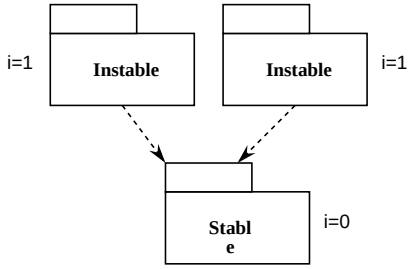
When the I metric is 1 it means that no other package depends upon this package; and this package does depend upon other packages. This is as instable as a package can get; it is *irresponsible* and *dependent*. Its lack of dependents gives it no reason *not* to change, and the packages that it depends upon may give it ample reason *to* change.

On the other hand, when the I metric is zero it means that the package is depended upon by other packages, but does not itself depend upon any other packages. It is *responsible* and *independent*. Such a package is as stable as it can get. Its dependents make it hard to change, and its has no dependencies that might force it to change.

The SDP says that the I metric of a package should be larger than the I metrics of the packages that it depends upon. i.e. I metrics should decrease in the direction of dependency.

Not all packages should be stable

If all the packages in a system were maximally stable, the system would be unchangeable. This is not a desirable situation. Indeed, we want to design our package structure so that some packages are instable and some are stable. The following figure shows the ideal configuration for a system with three packages.



The changeable packages are on top and depend upon the stable package at the bottom. Putting the instable packages at the top of the diagram is my own convention. By arraning them this way then any arrow that puts *up* is violating the SDP.

Where do we put the high level design?

Some software in the system should not change very often. This software represents the high level architecture and design decisions. We don't want these architectural decisions to be volatile. Thus, the software the encapsulates the high level design model of the system should be placed into stable packages. The instable packages should only contain the software that is likely to change.

However, if the high level design is placed into stable packages, then the source code that represents that design will be difficult to change. This could make the design inflexible. How can a package 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.

The Stable Abstractions Principle (SAP)

> ***PACKAGES THAT ARE MAXIMALLY STABLE SHOULD BE MAXIMALLY ABSTRACT. INSTABLE PACKAGES SHOULD BE CONCRETE. THE ABSTRACTION OF A PACKAGE SHOULD BE IN PROPORTION TO ITS STABILITY.***

This principle sets up a relationship between stability and abstractness. It says that a stable package should also be abstract so that its stability does not prevent it from being extended. On the other hand, it says that an instable package should be concrete since it its instability allows the concrete code within it to be easily changed.

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 package is to be stable, it should also consist of abstract classes so that it can be extended. Stable packages that are extensible are flexible and do not constrain the design.

The SAP and the SDP combined amount to the Dependency Inversion Principle for Packages. This is true because the SDP says that dependencies should run in the direction of stability, and the SAP says that stability implies abstraction. Thus, dependencies run in the direction of abstraction.

However, the DIP is a principle that deals with classes. And with classes there are no shades of grey. Either a class is abstract or it is not. The combination fo the SDP and SAP deal with packages, and allow that a packages can be partially abstract and partially stable.

Measuring Abstraction

The A metric is a measure of the abstractness of a package. Its value is simply the ratio of abstract classes[2] in a package to the total number of classes in the package.

---

2. Remember that an abstract class is simply a class that has at least one pure virtual function.

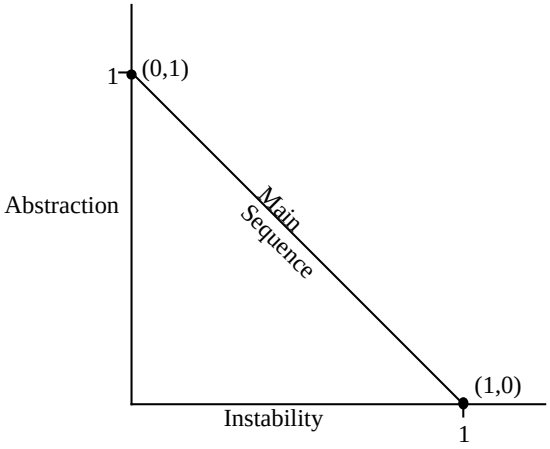
A = abstractClasses ÷ totalClasses

The A metric ranges from 0 to 1. Zero implies that the package has no abstract classes at all. A value of one implies that the package contains nothing but abstract classes.

This is not a perfect metric. It presumes that a class with 20 concrete functions and one pure virtual function should be counted the same as a class with nothing but pure virtual functions. However, I have found no good way to characterize the abstractness of a class based upon a ratio of virtual to non-virtual functions. Also, the fact that a single pure virtual function exists is very significant. Imperfect though it is, I have had good results with this metric.

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 packages on this graph, we will find the packages that are maximally stable and abstract at the upper left at (0,1). The packages that are maximally instable and concrete are at the lower right at (1,0).



But not all packages can fall into one of these two positions. packages have degrees of abstraction and stability. For example, it is very common for one abstract class to derive 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.

Since we cannot enforce that all packages sit at either (0,1) or (1,0) we must admit that there is a locus of points on the A/I graph which defines reasonable positions for packages. We can infer what that locus is by finding the areas where packages should *not* be. i.e. zones of *exclusion*.

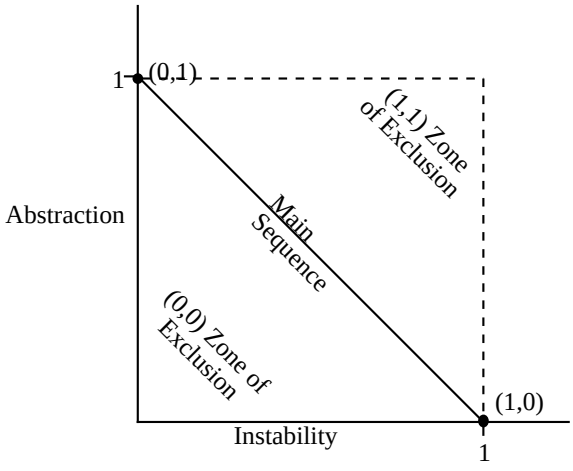
Consider a package in the area of A=0 and I=0. This is a highly stable and concrete package. Such a package is not desirable because it is rigid. It cannot be extended because it is not abstract. And it is very difficult to change because of its stability. Thus, we do not want to see well designed packages sitting near (0,0). The area around (0,0) is a zone of exclusion.

It should be noted that many packages do indeed fall within the (0,0) zone. An example would be a database schema. Database Schemas are notoriously volatile and are highly depended upon. This is one of the reasons that the interface between OO applications and databases is so difficult. Another example of a package that sits on (0,0) is a package that holds a concrete utility library. Although such a package has an I metric of 1, it may in fact be non-volatile. Consider a "string" package for example. Even though all the classes within it are concrete, it may still be non-volatile. Such packages are harmless in the (0,0) zone since they are not likely to be changed.

Consider a package with A=1 and I=1. This package 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. Packages in the (1,1) zone are pretty meaningless. Thus, this too is a zone of exclusion.

But what about a package with A=.5 and I=.5? This package 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 package seems "balanced". Its stability is in balance with its abstractness. Thus, the zone where A=I does not seem to be a zone of exclusion.

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



A package 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 package to hold are at one of the two endpoints of the main sequence. However, in my experience only about half the packages in a project can have such ideal characteristics. Those other packages 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 packages to be on or close to the main sequence, we can create a metric which measures how far away a package is from this ideal.

D : Distance : $|(A+I-1)\div\sqrt{2}|$: The perpendicular distance of a package 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 D’).

Given this metric, a design can be analyzed for its overall conformance to the main sequence. The D metric for each package can be calculated. Any package 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 packages 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 packages that are “exceptional” in comparison to all the others.

For those of you who are interested, I have written a bash script that scans a C++ directory structure and calculates all these metrics. You can get that script from the ‘freeware’ section of my website: ‘`http://www.oma.com`’

Conclusion

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.

This article is the culmination of the “Principles” series. I may, from time to time, revisit this topic as more priciples are identified.

This article is an extremely condensed version of a chapter from my new book: *Patterns and Advanced Principles of OOD*, to be published soon by Prentice Hall. In subsequent articles we will explore various design patterns, and their strengths and weaknesses with regard to implementation in C++. We will also be discussing multi-threading, relational databases, memory management and many other issues of interest to the C++ designer.