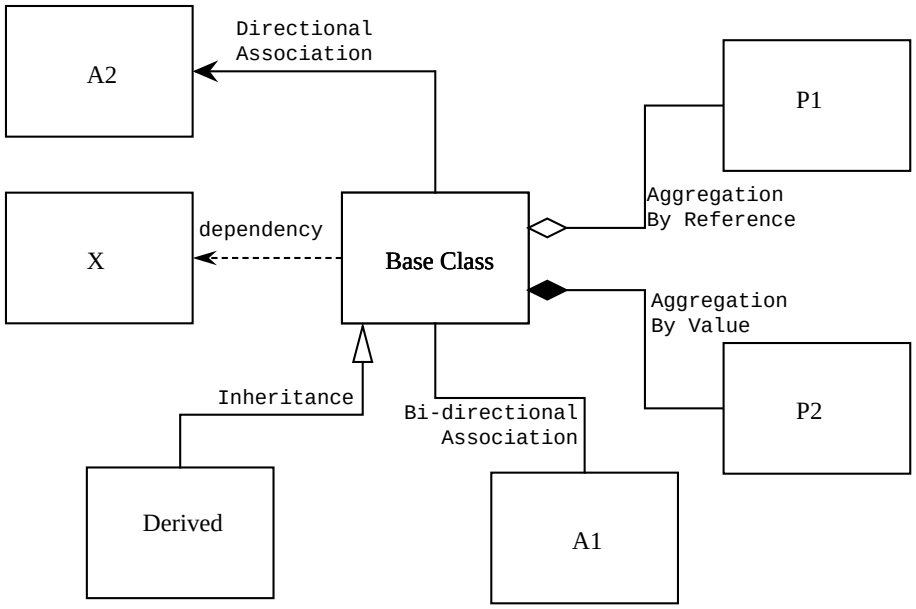
Granularity

This is the fifth 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 0.9) for documenting object oriented designs. The sidebar provides a very brief lexicon of this notation.

Sidebar
UML 0.9 Lexicon



Introduction

This article marks an important transition in this column. The first four articles which appeared in the January, March, May, and August issues of this magazine, described principles that govern the micro-structure of object-oriented software applications. This article is the first of several that will describe principles that govern the *macro* structure of *large* object-oriented applications. I emphasize the word *large*, because these principles are

most appropriate for applications that exceed 50,000 lines of C++ and require a team of engineers to write. This is a timely topic. John Lakos has recently published a book: ***<title>*** and several articles in the C++ Report ***<refs>*** that deal with the development of large C++ systems.

This article discusses *granularity.* This is a topic that is addressed by all of the major methodologists in slightly different ways, using very different vocabularies. We will examine these differences in an attempt to understand the common thread that binds them all together. Such a thread *does* exist, and it will lead us through some of the most important principles of software engineering.

Reprise

But before we begin to unravel the thread of granularity, a brief reprise of what has gone before in this column is 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) May, 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.

Granularity

As software applications grow in size and complexity they require some kind of high level organization. The class, while a very convenient unit for organizing small applications, is too finely grained to be used as an organizational unit for large applications. Something "larger" than a class is needed to help organize large applications.

Several major methodologists have identified the need for a larger granule of organization. Booch[1], uses the term "class category" to describe such a granule, Bertrand Meyer[2] refers to "clusters", Peter Coad[3] talks about "subject areas", and Sally Shlaer and Steve Mellor[4] talk about "Domains". In this article we will use the UML 0.9 terminology, and refer to these higher order granules as "packages".

The term "package" is common in Ada and Java circles. In those languages a package is used to represent a logical grouping of declarations that can be imported into other programs. In Java, for example, one can write several classes and incoporate them into the same package. Then other Java programs can 'import' that package to gain access to those classes.

Designing with Packages.

In the UML, packages can be used as containers for a group of classes. By grouping classes into packages we can reason about the design at a higher level of abstraction. The goal is to partition the classes in your application according to some criteria, and then allocate those partitions to packages. The relationships between those packages expresses the high level organization of the application. But this begs a large number of questions.

1. What are the best partitioning criteria?
2. What are the relationships that exist between packages, and what design princi-

---

1. *Object Oriented Analysis and Design with Applications*, Grady Booch, Benjamin Cummings, 1994
2. *Object Success*, Bertrand Meyer, Prentice Hall, 1995
3. *OOA*, Coad, et. al., Yourdon Press, 1990
4. *Object Lifecycles Modeling the World in States*, Sally Shlaer & Stephen Mellor, Yourdon Press, 1992

ples govern their use?

3. Should packages be designed before classes (Top down)? Or should classes be designed before packages (Bottom up)?
4. How are packages physically represented? In C++? In the development environment?
5. Once created, to what purpose will we put these packages?

To answer these questions, I have put together several design principles which govern the creation, interrelationship, and use of packages.

The Reuse/Release Equivalence Principle (REP).

> ***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 THE PACKAGE.***

Reusability is one of the most oft claimed goals of OOD. But what is reuse? Is it reuse if I snatch a bunch of code from one program and textually insert it into another? It is reuse if I steal a module from someone else and link it into my own libraries? I don't think so.

The above are examples of code copying; and it comes with a serious disadvantage: you own the code you copy! If it doesn't work in your environment, *you* have to change it. If there are bugs in the code, *you* have to fix them. If the original author finds some bugs in the code and fixes them, *you* have to find this out, and *you* have to figure out how to make the changes in your own copy. Eventually the code you copied diverges so much from the original that it can hardly be recognized. The code is *yours*. While code copying can make it easier to do some initial development; it does not help very much with the most expensive phase of the software lifecycle, maintenance.

I prefer to define reuse as follows. I reuse code if, and only if, I never need to look at the source code (other than the public portions of header files). I need only link with static libraries or include dynamic libraries. Whenever these libraries are fixed or enhanced, I receive a new version which I can then integrate into my system when opportunity allows.

That is, I expect the code I am reusing to be treated like a product. It is not maintained by me. It is not distributed by me. I am the customer, and the author, or some other entity, is responsible for maintaining it.

When the libraries that I am reusing are changed by the author, I need to be notified. Moreover, I may decide to use the old version of the library for a time. Such a decision will be based upon whether the changes made are important to me, and when I can fit the integration into my schedule. Therefore, I will need the author to make regular releases of the library. I will also need the author to be able to identify these releases with release numbers or names of some sort.

Thus, I can reuse nothing that is not also released. Moreover, when I reuse something in a released library, I am in effect a client of the entire library. Whether the changes affect me or not, I will have to integrate with each new version of the library when it comes out, so that I can take advantage of later enhancements and fixes.

And so, the REP states that the granule of reuse can be no smaller than the granule of release. Anything that we reuse must also be released. Clearly, packages are a candidate for a releasable entity. It might be possible to release and track classes, but there are so many classes in a typical application that this would almost certainly overwhelm the release tracking system. We need some larger scale entity to act as the granule of release; and the package seems to fit this need rather well.

The Common Reuse Principle (CRP)

> ***THE CLASSES IN A PACKAGE ARE REUSED TOGETHER. IF YOU REUSE ONE OF THE CLASSES IN A PACKAGE, YOU REUSE THEM ALL.***

This principle helps us to decide which classes should be placed into a package. It states that classes that tend to be reused together belong in the same package.

Classes are seldom reused in isolation. Generally reusable classes collaborate with other classes that are part of the reusable abstraction. The CRP states that these classes belong together in the same package.

A simple example might be a container class and its associated iterators. These classes are reused together because they are tightly coupled to each other. Thus they ought to be in the same package.

The reason that they belong together is that when an engineer decides to use a package a dependency is created upon the whole package. From then on, whether the engineer is using all the classes in the package or not, every time that package is released, the applications that use it must be revalidated and rereleased. If a package is being released because of changes to a class that I don't care about, then I will not be very happy about having to revalidate my application.

Moreover, it is common for packages to have physical representations as shared libraries or DLLs. If a DLL is released because of a change to a class that I don't care about, I still have to redistribute that new DLL and revalidate that the application works with it.

Thus, I want to make sure that when I depend upon a package, I depend upon every class in that package. Otherwise I will be revalidating and redistributing more than is necessary, and wasting lots of effort.

The Common Closure Principle (CCP)

> ***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.***

More important than reusability, is maintainability.

If the code in an application must change, where would you like those changes to occur: all in one package, or distributed through many packages? It seems clear that we would rather see the changes focused into a single package rather than have to dig through a whole bunch of packages and change them all. That way we need only release the one changed package. Other packages that don't depend upon the changed package do not need to be revalidated or rereleased.

The CCP is an attempt to gather together in one place all the classes that are likely to change for the same reasons. If two classes are so tightly bound, either physically or conceptually, such that they almost always change together; then they belong in the same package. This minimizes the workload related to releasing, revalidating, and redistributing the software.

This principles is closely associated with the Open Closed Principle (OCP). For it is "closure" in the OCP sense of the word that this principle is dealing with. The OCP states that classes should be closed for modification but open for extension. As we learned in the article that described the OCP, 100% closure is not attainable. Closure must be strategic. We design our systems such that the are closed to the most likely kinds of changes that we foresee.

The CCP amplifies this by grouping together classes which cannot be closed against certain types of changes into the same packages. Thus, when a change in requirements comes along; that change has a good chance of being restricted to a minimal number of packages.

The Acyclic Dependencies Principle (ADP)

> ***THE DEPENDENCY STRUCTURE BETWEEN PACKAGES MUST BE A DIRECTED ACYCLIC GRAPH (DAG). THAT IS, THERE MUST BE NO CYCLES IN THE DEPENDENCY STRUCTURE.***

Have you ever worked all day, gotten some stuff working and then gone home; only to arrive the next morning at to find that your stuff no longer works? Why doesn't it work? Because somebody stayed later than you! I call this: "the morning after syndrome".

The "morning after syndrome" occurs in development environments where many developers are modifying the same source files. In relatively small projects with just a few

developers, it isn't too big a problem. But as the size of the project and the development team grows, the mornings after can get pretty nightmarish. It is not uncommon for weeks to go by without being able to build a stable version of the project. Instead, everyone keeps on changing and changing their code trying to make it work with the last changes that someone else made.

The solution to this problem is to partition the development environment into releasable packages. The packages become units of work which are the responsibility of an engineer, or a team of engineers. When the responsible engineers get a package working, they release it for use by the other teams. They give it a release number and move it into a directory for other teams to use. They then continue to modify their package in their own private areas. Everyone else uses the released version.

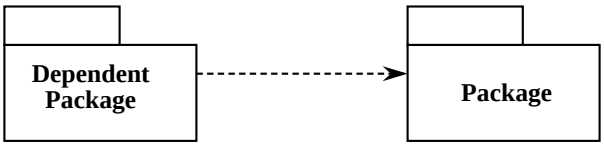
As new releases of a package are made, other teams can decide whether or not to immediately adopt the new release. If they decide not to, they simply continue using the old release. Once they decide that they are ready, they begin to use the new release.

Thus, none of the teams are at the mercy of the others. Changes made to one package do not need to have an immediate affect on other teams. Each team can decide for itself when to adapt its packages to new releases of the packages they use.

This is a very simple and rational process. And it is widely used. However, to make it work you must *manage* the dependency structure of the packages. There can be no cycles. If there are cycles in the dependency structure then the "morning after syndrome" cannot be avoided. I'll explain this further, but first I need to present the graphical tools that the UML 0.9 uses to depict the dependency structures of packages.

Packages depend upon one another. Specifically, a class in one package may `#include` the header file of a class in a different package. This can be depicted on a class diagram as a dependency relationship between packages (See Figure 1).

Figure 1
Dependencies between Packages.

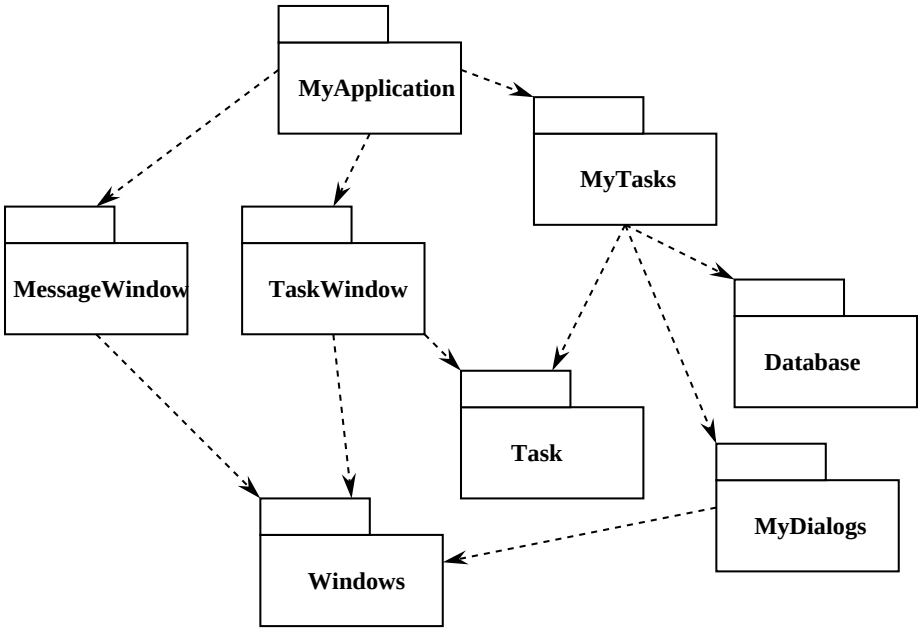


Packages, in UML 9.0 are depicted as "tabbed folders". Dependency relationships are dashed arrows. The arrows point in the direction of the dependency. That is, the arrow head is placed next to the package that is being depended upon. In C++ terms, there is a `#include` statement in a class within the dependent package that references the header file of a class in the package being depended upon.

Consider the package diagram in Figure 2. Here we see a rather typical structure of packages assembled into an application. The function of this application is unimportant for the purpose of this example. What *is* important is the dependency structure of the packages. Notice how this structure is a *graph*. The packages are the *nodes*, and the depen-

dency relationships are the *edges*. Notice also that the dependency relationships have direction. So this structure is a *directed graph*.

Figure 2
Package Diagram without Cycles



Now notice one more thing. Regardless of which package you begin at, it is impossible to follow the dependency relationships and wind up back at that package. This structures has no cycles. It is a *directed acyclic graph*. (DAG).

Now, notice what happens when the team responsible for MyDialogs makes a new release. It is easy to find out who is affected by this release; you just follow the dependency arrows backwards. Thus, MyTasks and MyApplication are both going to be affected. The teams responsible for those packages will have to decide when they should integrate with the new release of MyDialogs.

Notice also that when MyDialogs is released it has utterly no affect upon many of the other packages in the system. They don't know about MyDialogs; and they don't care when it changes. This is nice. It means that the impact of releasing MyDialogs is relatively small.

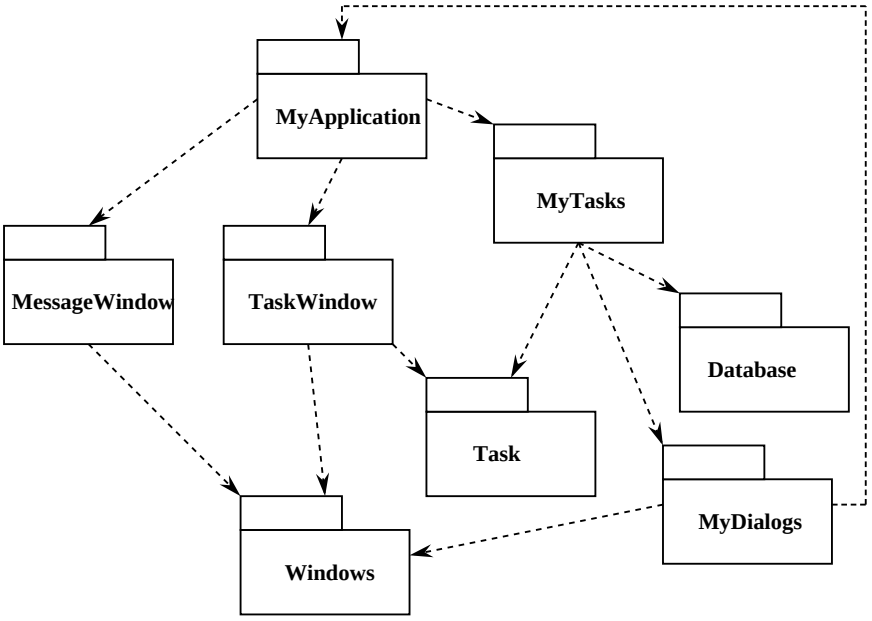
When the engineers responsible for the MyDialogs package would like to run a unit test of their package, all they need do is compile and link their version of MyDialogs with the version of the Windows package that they are currently using. None of the other packages in the system need be involved. This is nice, it means that the engineers responsible for MyDialogs have relatively little work to do to set up a unit test; and that there are relatively few variables for them to consider.

When it is time to release the whole system; it is done from the bottom up. First the Windows package is compiled, tested, and released. Then MessageWindow and Mydialogs. These are followed by Task, and then TaskWindow and Database. MyTasks is next; and finally MyApplication. This process is very clear and easy to deal with. We know how to build the system because we understand the dependencies between its parts.

The Effect of a Cycle in the Package Dependency Graph.

Let us say that the a new requirement forces us to change one of the classes in MyDialogs such that it #includes one of the class headers in MyApplication. This creates a dependency cycle as shown in Figure 3.

Figure 3
Package Diagram *with* Cycles



This cycle creates some immediate problems. For example, the engineers responsible for the MyTasks package know that in order to release, they must be compatible with Task, MyDialogs, Database, and Windows. However, with the cycle in place, they must now also be compatible with MyApplication, TaskWindow and MessageWindow. That is, MyTasks now depends upon *every other package in the system.* This makes MyTasks very difficult to release. MyDialogs suffers the same fate. In fact, the cycle has had the effect that MyApplication, MyTasks, and MyDialogs must always be released at the same time. They have, in effect, become one large package. And all the engineers who are working in any of those packages will experience “the morning after syndrome” once again. They will

be stepping all over one another since they must all be using exactly the same release of each other.

But this is just the tip of the trouble. Consider what happens when we want to unit test the Mydialogs package. We find that we must link in every other package in the system; including the Database package. This means that we have to do a *complete build* just to unit test MyDialogs. This is intolerable.

If you have ever wondered why you have to link in so many different libraries, and so much of everybody else's stuff, just to run a simple unit test of one of your classes, it is probably because there are cycles in the dependency graph. Such cycles make it very difficult to isolate modules. Unit testing and releasing become very difficult and error prone. And compile times grow geometrically with the number of modules.

Breaking the Cycle.

It is always possible to break a cycle of packages and reinstate the dependency graph as a DAG. There are two primary mechanisms.

1. Apply the Dependency Inversion Principle (DIP). In the case of Figure 3, we could create an abstract base class that has the interface that MyDialogs needs. We could then put that abstract base into MyDialogs and inherit it into MyApplication. This inverts the dependency between MyDialogs and MyApplication thus breaking the cycle. See Figure 4.
2. Create a new package that both MyDialogs and MyApplication depend upon. Move the class(es) that they both depend upon into that new package.

The "Jitters"

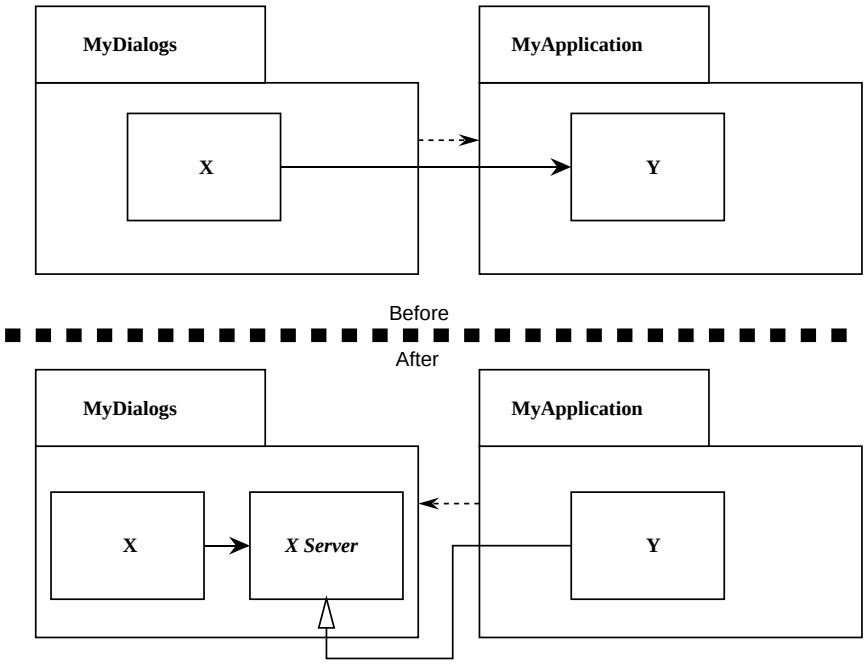
The second solution implies that the package structure is not stable in the presence of changing requirements. Indeed, as the application grows, the package dependency structure jitters and grows. Thus the dependency structure must always be monitored for cycles. When cycles occur they must be broken somehow. Sometimes this will mean creating new packages, making the dependency structure grow.

Top Down Design

The issues we have discussed so far lead to an inescapable conclusion. The package structure cannot be designed from the top down. This means that is it not one of the first things about the system that is designed. Indeed, it seems that it gets designed after many of the classes in the system have been designed, and thereafter remains in a constant state of flux.

Many should find this to be counterintuitive. We have come to expect that large grained decompositions are also high level functional decompositions. When we see a

Figure 4
Breaking the Cycle with Dependency Inversion



large grained grouping like a package dependency structure, we feel that it ought to somehow represent the function of the system. Yet this does not seem to be an attribute of package dependency diagrams.

In fact, package dependency diagrams have very little do to with describing the function of the application. Instead, they are a map of how to *build* the application. This is why they aren't designed at the start of the project. There is no software to build, and so there is no need for a build map. But as more and more classes accumulate in the early stages of implementation and design, there is a growing need to map out the dependencies so that the project can be developed without the "morning after syndrome". Moreover, we want to keep changes as localized as possible, so we start paying attention to the common closure principle and collocate classes that are likely to change together.

As the application continues to grow, we start becoming concerned about creating reusable elements. Thus the Common Reuse Principle begins to dictate the composition of the packages. Finally, as cycles appear the package dependency graph jitters and grows.

If we were to try to design the package dependency structure before we had designed any classes, we would likely fail rather badly. We would not know much about common closure, we would be unaware of any reusable elements, and we would almost certainly create packages that produced dependency cycles. Thus the package dependency structure grows and evolves with the logical design of the system.

Conclusion

Managing a complex project using packages and their interdependencies is one of the most powerful tools of OOD. By creating packages that conform to the three principles described in this paper, we set the stage for robust, maintainable, and reusable software. Packages are the units that focus change, that enable reuse, and that provide the unit of release that prevents developers from interfering with each other.

My next article will be the last in the "Principles" series. It will describe the last two principles. These principles provide governance over the relationships between packages, rather than the composition of the packages. They also introduce some design quality metrics that measure conformance to the Dependency Inversion Principle (DIP).

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 many of the other principles of object oriented design. We will also study various design patterns, and their strengths and weaknesses with regard to implementation in C++. We will study the role of Booch's class categories in C++, and their applicability as C++ namespaces. We will define what "cohesion" and "coupling" mean in an object oriented design, and we will develop metrics for measuring the quality of an object oriented design. And after that, we will discuss many other interesting topics.