Design Principles and Design Patterns

Robert C. Martin
www.objectmentor.com

What is software architecture? The answer is multitiered. At the highest level, there are the architecture patterns that define the overall shape and structure of software applications[1]. Down a level is the architecture that is specifically related to the purpose of the software application. Yet another level down resides the architecture of the modules and their interconnections. This is the domain of design patterns[2], packakges, components, and classes. It is this level that we will concern ourselves with in this chapter.

Our scope in this chapter is quite limitted. There is much more to be said about the principles and patterns that are exposed here. Interested readers are referred to [Martin99].

Architecture and Dependencies

What goes wrong with software? The design of many software applications begins as a vital image in the minds of its designers. At this stage it is clean, elegant, and compelling. It has a simple beauty that makes the designers and implementers itch to see it working. Some of these applications manage to maintain this purity of design through the initial development and into the first release.

But then something begins to happen. The software starts to rot. At first it isn't so bad. An ugly wart here, a clumsy hack there, but the beauty of the design still shows through. Yet, over time as the rotting continues, the ugly festering sores and boils accumulate until they dominate the design of the application. The program becomes a festering mass of code that the developers find increasingly hard to maintain. Eventu-

1. [Shaw96]
2. [GOF96]

ally the sheer effort required to make even the simplest of changes to the application becomes so high that the engineers and front line managers cry for a redesign project.

Such redesigns rarely succeed. Though the designers start out with good intentions, they find that they are shooting at a moving target. The old system continues to evolve and change, and the new design must keep up. The warts and ulcers accumulate in the new design before it ever makes it to its first release. On that fateful day, usually much later than planned, the morass of problems in the new design may be so bad that the designers are already crying for another redesign.

Symptoms of Rotting Design

There are four primary symptoms that tell us that our designs are rotting. They are not orthogonal, but are related to each other in ways that will become obvious. they are: rigidity, fragility, immobility, and viscosity.

Rigidity. Rigidity is the tendency for software to be difficult to change, even in simple ways. Every change causes a cascade of subsequent changes in dependent modules. What begins as a simple two day change to one module grows into a multi-week marathon of change in module after module as the engineers chase the thread of the change through the application.

When software behaves this way, managers fear to allow engineers to fix non-critical problems. This reluctance derives from the fact that they don't know, with any reliability, when the engineers will be finished. If the managers turn the engineers loose on such problems, they may disappear for long periods of time. The software design begins to take on some characteristics of a roach motel -- engineers check in, but they don't check out.

When the manager's fears become so acute that they refuse to allow changes to software, official rigidity sets in. Thus, what starts as a design deficiency, winds up being adverse management policy.

Fragility. Closely related to rigidity is fragility. Fragility is the tendency of the software to break in many places every time it is changed. Often the breakage occurs in areas that have no conceptual relationship with the area that was changed. Such errors fill the hearts of managers with foreboding. Every time they authorize a fix, they fear that the software will break in some unexpected way.

As the fragility becomes worse, the probability of breakage increases with time, asymptotically approaching 1. Such software is impossible to maintain. Every fix makes it worse, introducing more problems than are solved.

Such software causes managers and customers to suspect that the developers have lost control of their software. Distrust reigns, and credibility is lost.

Immobility. Immobility is the inability to reuse software from other projects or from parts of the same project. It often happens that one engineer will discover that he needs a module that is similar to one that another engineer wrote. However, it also often happens that the module in question has too much baggage that it depends upon. After much work, the engineers discover that the work and risk required to separate the desirable parts of the software from the undesirable parts are too great to tolerate. And so the software is simply rewritten instead of reused.

Viscosity. Viscosity comes in two forms: viscosity of the design, and viscosity of the environment. When faced with a change, engineers usually find more than one way to make the change. Some of the ways preserve the design, others do not (i.e. they are hacks.) When the design preserving methods are harder to employ than the hacks, then the viscosity of the design is high. It is easy to do the wrong thing, but hard to do the right thing.

Viscosity of environment comes about when the development environment is slow and inefficient. For example, if compile times are very long, engineers will be tempted to make changes that don't force large recompiles, even though those changes are not optiimal from a design point of view. If the source code control system requires hours to check in just a few files, then engineers will be tempted to make changes that require as few check-ins as possible, regardless of whether the design is preserved.

These four symptoms are the tell-tale signs of poor architecture. Any application that exhibits them is suffering from a design that is rotting from the inside out. But what causes that rot to take place?

Changing Requirements

The immediate cause of the degradation of the design is well understood. The requirements have been changing in ways that the initial design did not anticipate. Often these changes need to be made quickly, and may be made by engineers who are not familiar with the original design philosophy. So, though the change to the design works, it somehow violates the original design. Bit by bit, as the changes continue to pour in, these violations accumulate until malignancy sets in.

However, we cannot blame the drifting of the requirements for the degradation of the design. We, as software engineers, know full well that requirements change. Indeed, most of us realize that the requirements document is the most volatile document in the

project. If our designs are failing due to the constant rain of changing requirements, it is our designs that are at fault. We must somehow find a way to make our designs resilient to such changes and protect them from rotting.

Dependency Management

What kind of changes cause designs to rot? Changes that introduce new and unplanned for dependencies. Each of the four symptoms mentioned above is either directly, or indirectly caused by improper dependencies between the modules of the software. It is the dependency architecture that is degrading, and with it the ability of the software to be maintained.

In order to forestall the degradation of the dependency architecture, the dependencies between modules in an application must be managed. This management consists of the creation of dependency firewalls. Accross such firewalls, dependencies do not propogate.

Object Oriented Design is replete with principles and techniques for building such firewalls, and for managing module dependencies. It is these principles and techniques that will be discussed in the remainder of this chapter. First we will examine the principles, and then the techniques, or design patterns, that help maintain the dependency architecture of an application.

Principles of Object Oriented Class Design

The Open Closed Principle (OCP)[1]

A module should be open for extension but closed for modification.

Of all the principles of object oriented design, this is the most important. It originated from the work of Bertrand Meyer[2]. It means simply this: We should write our modules so that they can be extended, without requiring them to be modified. In other words, we want to be able to change what the modules do, without changing the source code of the modules.

1. [OCP97]
2. [OOSC98]

This may sound contradictory, but there are several techniques for achieving the OCP on a large scale. All of these techniques are based upon abstraction. Indeed, *abstraction is the key to the OCP*. Several of these techniques are described below.

Dynamic Polymorphism. Consider Listing 2-1. the `LogOn` function must be changed every time a new kind of modem is added to the software. Worse, since each different type of modem depends upon the `Modem::Type` enumeration, each modem must be recompiled every time a new kind of modem is added.

Listing 2-1
Logon, must be modified to be extended.

```
struct Modem
{
  enum Type {hayes, courrier, ernie) type;
};

struct Hayes
{
  Modem::Type type;
  // Hayes related stuff
};

struct Courrier
{
  Modem::Type type;
  // Courrier related stuff
};

struct Ernie
{
  Modem::Type type;
  // Ernie related stuff
};

void LogOn(Modem& m,
           string& pno, string& user, string& pw)
{
  if (m.type == Modem::hayes)
    DialHayes((Hayes&)m, pno);
  else if (m.type == Modem::courrier)
    DialCourrier((Courrier&)m, pno);
  else if (m.type == Modem::ernie)
    DialErnie((Ernie&)m, pno)
  // ...you get the idea
}
```


Of course this is not the worst attribute of this kind of design. Programs that are designed this way tend to be littered with similar if/else or switch statement. Every time anything needs to be done to the modem, a switch statement if/else chain will need to select the proper functions to use. When new modems are added, or modem policy changes, the code must be scanned for all these selection statements, and each must be appropriately modified.

Worse, programmers may use local optimizations that hide the structure of the selection statements. For example, it might be that the function is exactly the same for Hayes and Courrier modems. Thus we might see code like this:

```
if (modem.type == Modem::ernie)
   SendErnie((Ernie&)modem, c);
else
   SendHayes((Hayes&)modem, c);
```

Clearly, such structures make the system much harder to maintain, and are very prone to error.

As an example of the OCP, consider Figure 2-13. Here the LogOn function depends only upon the Modem interface. Additional modems will not cause the LogOn function to change. Thus, we have created a module that can be extended, with new modems, without requiring modification. See Listing 2-2.

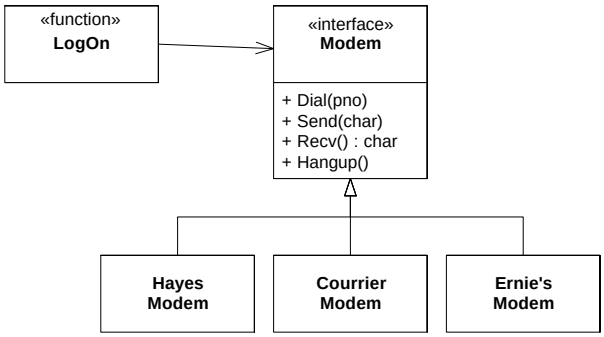

Figure 2-13

Listing 2-2
LogOn has been closed for modification

```
class Modem
{
public:
```


Listing 2-2
LogOn has been closed for modification

```
   virtual void Dial(const string& pno) = 0;
   virtual void Send(char) = 0;
   virtual char Recv() = 0;
   virtual void Hangup() = 0;
};

void LogOn(Modem& m,
           string& pno, string& user, string& pw)
{
   m.Dial(pno);
   // you get the idea.
}
```

Static Polymorphism. Another technique for conforming to the OCP is through the use of templates or generics. Listing 2-3 shows how this is done. The LogOn function can be extended with many different types of modems without requiring modification.

Listing 2-3
Logon is closed for modification through static polymorphism

```
template <typename MODEM>
void LogOn(MODEM& m,
           string& pno, string& user, string& pw)
{
   m.Dial(pno);
   // you get the idea.
}
```

Architectural Goals of the OCP. By using these techniques to conform to the OCP, we can create modules that are extensible, without being changed. This means that, with a little forethought, we can add new features to existing code, without changing the existing code and by only adding new code. This is an ideal that can be difficult to achieve, but you will see it achieved, several times, in the case studies later on in this book.

Even if the OCP cannot be fully achieved, even partial OCP compliance can make dramatic improvements in the structure of an application. It is always better if changes do not propogate into existing code that already works. If you don't have to change working code, you aren't likely to break it.

The Liskov Substitution Principle (LSP)[1]

Subclasses should be substitutable for their base classes.

This principle was coined by Barbar Liskov[2] in her work regarding data abstraction and type theory. It also derives from the concept of Design by Contract (DBC) by Bertrand Meyer[3].

The concept, as stated above, is depicted in Figure 2-14. Derived classes should be substitutable for their base classes. That is, a user of a base class should continue to function properly if a derivative of that base class is passed to it.

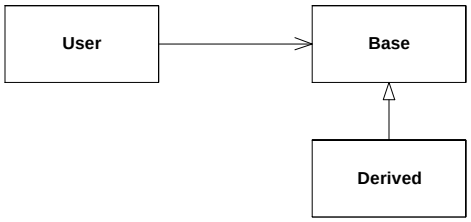

Figure 2-14
LSP schema.

In other words, if some function `User` takes an argument ot type `Base`, then as shown in Listing 2-4, it should be legal to pass in an instance of `Derived` to that function.

Listing 2-4
User, Based, Derived, example.

```
void User(Base& b);

Derived d;
User(d);
```

This may seem obvious, but there are subtleties that need to be considered. The canonical example is the Circle/Ellipse dilemma.

1. [LSP97]
2. [Liksov88]
3. [OOSC98]

The Circle/Ellipse Dilemma. Most of us learn, in high school math, that a circle is just a degenerate form of an ellipse. All circles are ellipses with coincident foci. This is-a relationship tempts us to model circles and ellipses using inheritance as shown in Figure 2-15.

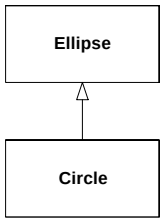

Figure 2-15
Circle / Ellipse Dilemma

While this satisfies our conceptual model, there are certain difficulties. A closer look at the declaration of `Ellipse` in Figure 2-16 begins to expose them. Notice that `Ellipse` has three data elements. The first two are the foci, and the last is the length of the major axis. If `Circle` inherits from `Ellipse`, then it will inherit these data variables. This is unfortunate since `Circle` really only needs two data elements, a center point and a radius.

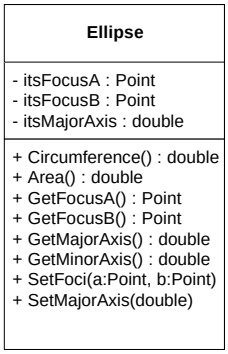

Figure 2-16
Declaration of Ellipse

Still, if we ignore the slight overhead in space, we can make `Circle` behave properly by overriding its `SetFoci` method to ensure that both foci are kept at the same value. See Listing 2-5. Thus, either focus will act as the center of the circle, and the major axis will be its diameter.

Listing 2-5
Keeping the Circle Foci coincident.

```
void Circle::SetFoci(const Point& a, const Point& b)
{
   itsFocusA = a;
   itsFocusB = a;
}
```

Clients Ruin Everything. Certainly the model we have created is self consistent. An instance of `Circle` will obeys all the rules of a circle. There is nothing you can do to it to make it violate those rules. So too for `Ellipse`. The two classes form a nicely consistent model, even if `Circle` has one too many data elements.

However, `Circle` and `Ellipse` do not live alone in a universe by themselves. They cohabit that universe with many other entities, and provide their public interfaces to those entities. Those interfaces imply a contract. The contract may not be explicitly stated, but it is there nonetheless. For example, users of `Ellipse` have the right to expect the following code fragment to succeed:

```
void f(Ellipse& e)
{
   Point a(-1,0);
   Point b(1,0);
   e.SetFoci(a,b);
   e.SetMajorAxis(3);
   assert(e.GetFocusA() == a);
   assert(e.GetFocusB() == b);
   assert(e.GetMajorAxis() == 3);
}
```

In this case the function expects to be working with an `Ellipse`. As such, it expects to be able to set the foci, and major axis, and then verify that they have been properly set. If we pass an instance of `Ellipse` into this function, it will be quite happy. However, if we pass an instance of `Circle` into the function, it will fail rather badly.

If we were to make the contract of `Ellipse` explicit, we would see a postcondition on the `SetFoci` that guaranteed that the input values got copied to the member variables, and that the major axis variable was left unchanged. Clearly `Circle` violates this guarantee because it ignores the second input variable of `SetFoci`.

Design by Contract. Restating the LSP, we can say that, in order to be substitutable, the contract of the base class must be honored by the derived class. Since

`Circle` does not honor the implied contract of `Ellipse`, it is not substitutable and violates the LSP.

Making the contract explicit is an avenue of research followed by Bertrand Meyer. He has invented a language named Eiffel in which contracts are explicitly stated for each method, and explicitly checked at each invocation. Those of us who are not using Eiffel, have to make do with simple assertions and comments.

To state the contract of a method, we declare what must be true before the method is called. This is called the precondition. If the precondition fails, the results of the method are undefined, and the method ought not be called. We also declare what the method guarantees will be true once it has completed. This is called the postcondition. A method that fails its postcondition should not return.

Restating the LSP once again, this time in terms of the contracts, a derived class is substitutable for its base class if:

1. Its preconditions are no stronger than the base class method.
2. Its postconditions are no weaker than the base class method.

Or, in other words, derived methods should *expect no more and provide no less*.

Repercussions of LSP Violation. Unfortunately, LSP violations are difficult to detect until it is too late. In the Circle/Ellipse case, everything worked fine until some client came along and discovered that the implicit contract had been violated.

If the design is heavily used, the cost of repairing the LSP violation may be too great to bear. It might not be economical to go back and change the design, and then rebuild and retest all the existing clients. Therefore the solution will likely be to put into an if/else statement in the client that discovered the violation. This if/else statement checks to be sure that the `Ellipse` is actually an `Ellipse` and not a `Circle`. See Listing 2-6.

Listing 2-6
Ugly fix for LSP violation

```
void f(Ellipse& e)
{
   if (typeid(e) == typeid(Ellipse))
   {
      Point a(-1,0);
      Point b(1,0);
      e.SetFoci(a,b);
      e.SetMajorAxis(3);
      assert(e.GetFocusA() == a);
      assert(e.GetFocusB() == b);
```


Listing 2-6
Ugly fix for LSP violation

```
      assert(e.GetMajorAxis() == 3);
   }
   else
      throw NotAnEllipse(e);
}
```

Careful examination of Listing 2-6 will show it to be a violation of the OCP. Now, whenever some new derivative of `Ellipse` is created, this function will have to be checked to see if it should be allowed to operate upon it. *Thus, violations of LSP are latent violations of OCP*.

The Dependency Inversion Principle (DIP)[1]

Depend upon Abstractions. Do not depend upon concretions.

If the OCP states the goal of OO architecture, the DIP states the primary mechanism. Dependency Inversion is the strategy of depending upon interfaces or abstract functions and classes, rather than upon concrete functions and classes. This principle is the enabling force behind component design, COM, CORBA, EJB, etc.

Procedural designs exhibit a particular kind of dependency structure. As Figure 2-17 shows, this structure starts at the top and points down towards details. High level modules depend upon lower level modules, which depend upon yet lower level modules, etc..

A little thought should expose this dependency structure as intrinsically weak. The high level modules deal with the high level policies of the application. These policies generally care little about the details that implement them. Why then, must these high level modules directly depend upon those implementation modules?

An object oriented architecture shows a very different dependency structure, one in which the majority of dependencies point towards abstractions. Morevoer, the modules that contain detailed implementation are no longer depended upon, rather they *depend themselves* upon abstractions. Thus the dependency upon them has been *inverted*. See Figure 2-18.

Depending upon Abstractions. The implication of this principle is quite simple. Every dependency in the design should target an interface, or an abstract class. No dependency should target a concrete class.

1. [DIP97]

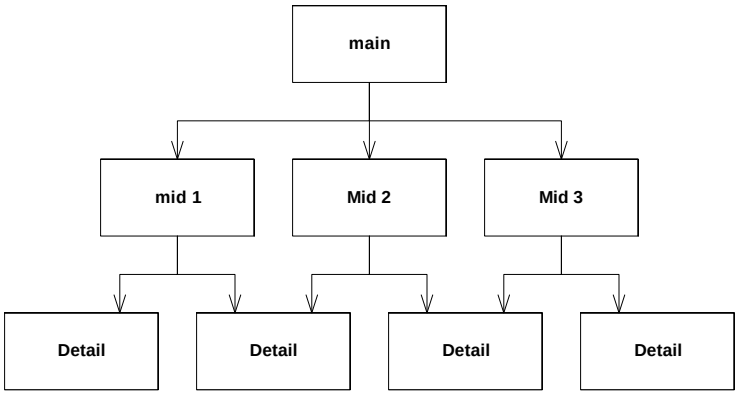

Figure 2-17
Dependency Structure of a Procedural Architecture

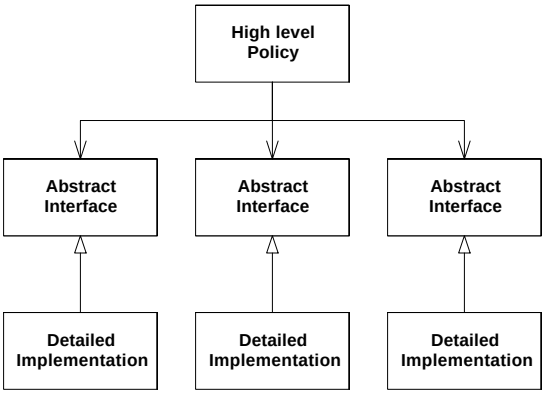

Figure 2-18
Dependency Structure of an Object Oriented Architecture

Clearly such a restriction is draconian, and there are mitigating circumstatnces that we will explore momentarily. But, as much as is feasible, the principle should be followed. The reason is simple, concrete things change alot, abstract things change much less frequently. Morevoer, abstractions are “hinge points”, they represent the places where the design can bend or be extended, without themselves being modified (OCP).

Substrates such as COM enforce this principle, at least between components. The only visible part of a COM component is its abstract interface. Thus, in COM, there is little escape from the DIP.

Mitigating Forces. One motivation behind the DIP is to prevent you from depending upon volatile modules. The DIP makes the assumption that anything concrete is volatile. While this is frequently so, especially in early development, there are exceptions. For example, the `string.h` standard C library is very concrete, but is not at all volatile. Depending upon it in an ANSI string environment is not harmful. Likewise, if you have tried and true modules that are concrete, but not volatile, depending upon them is not so bad. Since they are not likely to change, they are not likely to inject volatility into your design.

Take care however. A dependency upon `string.h` could turn very ugly when the requirements for the project forced you to change to UNICODE characters. Non-volatility is not a replacement for the substitutability of an abstract interface.

Object Creation. One of the most common places that designs depend upon concrete classes is when those designs create instances. By definition, you cannot create instances of abstract classes. Thus, to create an instance, you must depend upon a concrete class.

Creation of instances can happen all through the architecture of the design. Thus, it might seem that there is no escape and that the entire architecture will be littered with dependencies upon concrete classes. However, there is an elegant solution to this problem named ABSTRACTFACTORY[1] -- a design pattern that we'll be examining in more detail towards the end of this chapter.

The Interface Segregation Principle (ISP)[2]

Many client specific interfaces are better than one general purpose interface

The ISP is another one of the enabling technologies supporting component substrates such as COM. Without it, components and classes would be much less useful and portable.

The essence of the principle is quite simple. If you have a class that has several clients, rather than loading the class with all the methods that the clients need, create specific interfaces for each client and multiply inherit them into the class.

1. [GOF96] p??
2. [ISP97]

Figure 2-19 shows a class with many clients, and one large interface to serve them all. Note that whenever a change is made to one of the methods that `ClientA` calls, `ClientB` and `ClientC` may be affected. It may be necessary to recompile and redeploy them. This is unfortunate.

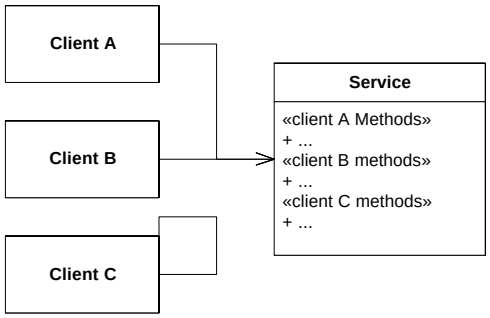

Figure 2-19
Fat Service with Integrated Interfaces

A better technique is shown in Figure 2-20. The methods needed by each client are placed in special interfaces that are specific to that client. Those interfaces are multiply inherited by the `Service` class, and implemented there.

If the interface for `ClientA` needs to change, `ClientB` and `ClientC` will remain unaffected. They will not have to be recompiled or redeployed.

What does Client Specific Mean? The ISP does not recommend that every class that uses a service have its own special interface class that the service must inherit from. If that were the case, the service would depend upon each and every client in a bizarre and unhealthy way. Rather, clients should be categorized by their type, and interfaces for each type of client should be created.

If two or more different client types need the same method, the method should be added to both of their interfaces. This is neither harmful nor confusing to the client.

Changing Interfaces. When object oriented applications are maintained, the interfaces to existing classes and components often change. There are times when these changes have a huge impact and force the recompilation and redeployment of a very large part of the design. This impact can be mitigated by adding new interfaces to existing objects, rather than changing the existing interface. Clients of the old inter-

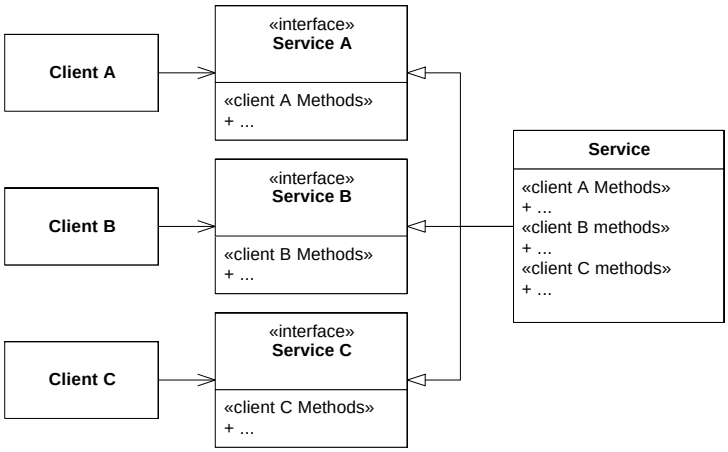

Figure 2-20
Segregated Interfaces

face that wish to access methods of the new interface, can query the object for that interface as shown in the following code.

```
void Client(Service* s)
{
   if (NewService* ns = dynamic_cast<NewService*>(s))
   {
     // use the new service interface
   }
}
```

As with all principles, care must be taken not to overdo it. The specter of a class with hundreds of different interfaces, some segregated by client and other segregated by version, would be frightening indeed.

Principles of Package Architecture

Classes are a necessary, but insufficient, means of organizing a design. The larger granularity of packages are needed to help bring order. But how do we choose which classes belong in which packages. Below are three principles known as the *Package Cohesion Principles*, that attempt to help the software architect.

The Release Reuse Equivalency Principle (REP)[1]

The granule of reuse is the granule of release.

A reusable element, be it a component, a class, or a cluster of classes, cannot be reused unless it is managed by a release system of some kind. Users will be unwilling to use the element if they are forced to upgrade every time the author changes it. Thus. even though the author has released a new version of his reusable element, he must be willing to support and maintain older versions while his customers go about the slow business of getting ready to upgrade. Thus, clients will refuse to reuse an element unless the author promises to keep track of version numbers, and maintain old versions for awhile.

Therefore, one criterion for grouping classes into packages is reuse. Since packages are the unit of release, they are also the unit of reuse. Therefore architects would do well to group reusable classes together into packages.

The Common Closure Principle (CCP)[2]

Classes that change together, belong together.

A large development project is subdivided into a large network of interelated packages. The work to manage, test, and release those packages is non-trivial. The more packages that change in any given release, the greater the work to rebuild, test, and deploy the release. Therefore we would like to minimze the number of packages that are changed in any given release cycle of the product.

To achieve this, we group together classes that we think will change together. This requires a certain amount of precience since we must anticipate the kinds of changes that are likely. Still, when we group classes that change together into the same packages, then the package impact from release to release will be minimzed.

The Common Reuse Principle (CRP)[3]

Classes that aren't reused together should not be grouped together.

A dependency upon a package is a dependency upon everything within the package. When a package changes, and its release number is bumped, all clients of that pack-

1. [Granularity97]
2. [Granularity97]
3. [Granularity97]

age must verify that they work with the new package -- even if nothing they used within the package actually changed.

We frequently experience this when our OS vendor releases a new operating system. We have to upgrade sooner or later, because the vendor will not support the old version forever. So even though nothing of interest to us changed in the new release, we must go through the effort of upgrading and revalidating.

The same can happen with packages if classes that are not used together are grouped together. Changes to a class that I don't care about will still force a new release of the package, and still cause me to go through the effort of upgrading and revalidating.

Tension between the Package Cohesion Principles

These three principles are mutually exclusive. They cannot simultaneously be satisfied. That is because each principle benefits a different group of people. The REP and CRP makes life easy for reusers, whereas the CCP makes life easier for maintainers. The CCP strives to make packages as large as possible (after all, if all the classes live in just one package, then only one package will ever change). The CRP, however, tries to make packages very small.

Fortunately, packages are not fixed in stone. Indeed, it is the nature of packages to shift and jitter during the course of development. Early in a project, architects may set up the package structure such that CCP dominates and develoipment and maintenance is aided. Later, as the architecture stabilizes, the architects may refactor the package structure to maximize REP and CRP for the external reusers.

The Package Coupling Principles.

The next three packages govern the interlationships between packages. Applications tend to be large networks of interlated packages. The rules that govern these interrelationship are some of the most important rules in object oriented architecture.

The Acyclic Dependencies Principle (ADP)[1]

The dependencies betwen packages must not form cycles.

Since packages are the granule of release, they also tend to focus manpower. Engineers will typically work inside a single package rather than working on dozens. This tedency is amplified by the package cohesion principles, since they tend to group

1. [Granularity97]

together those classes that are related. Thus, engineers will find that their changes are directed into just a few package. Once those changes are made, they can release those packages to the rest of the project.

Before they can do this release, however, they must test that the package works. To do that, they must compile and build it with all the packages that it depends upon. Hopefully this number is small.

Consider Figure 2-21. Astute readers will recognize that there are a number of flaws in the architecture. The DIP seems to have been abandoned, and along with it the OCP. The GUI depends directly upon the communications package, and apparently is reponsible for transporting data to the analysis package. Yuk.

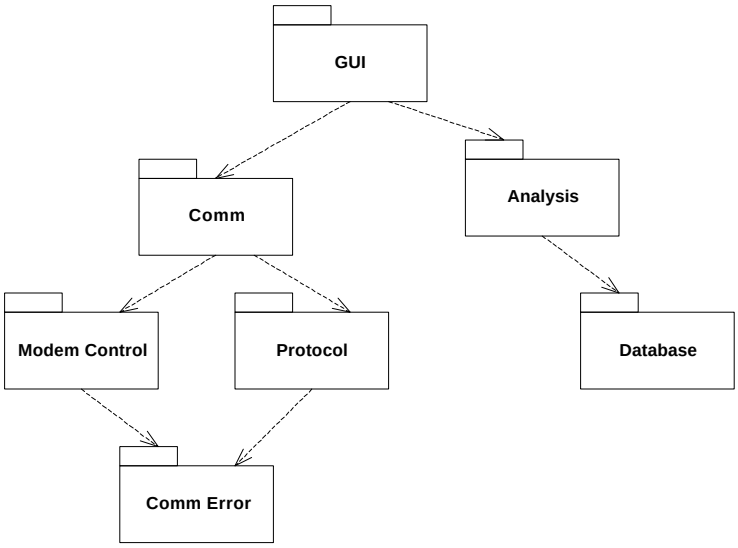

Figure 2-21
Acyclic Package Network

Still, lets use this rather ugly structure for some examples. Consider what would be required to release the `Protocol` package. The engineers would have to build it with the latest release of the `CommError` package, and run their tests. `Protocol` has no other dependencies, so no other package is needed. This is nice. We can test and release with a minimal amount of work.

A Cycle Creeps In. But now lets say that I am an engineer working on the `CommError` package. I have decided that I need to display a message on the screen. Since the screen is controlled by the `GUI`, I send a message to one of the `GUI` objects to get my message up on the screen. This means that I have made `CommError` dependent upon `GUI`. See Figure 2-22.

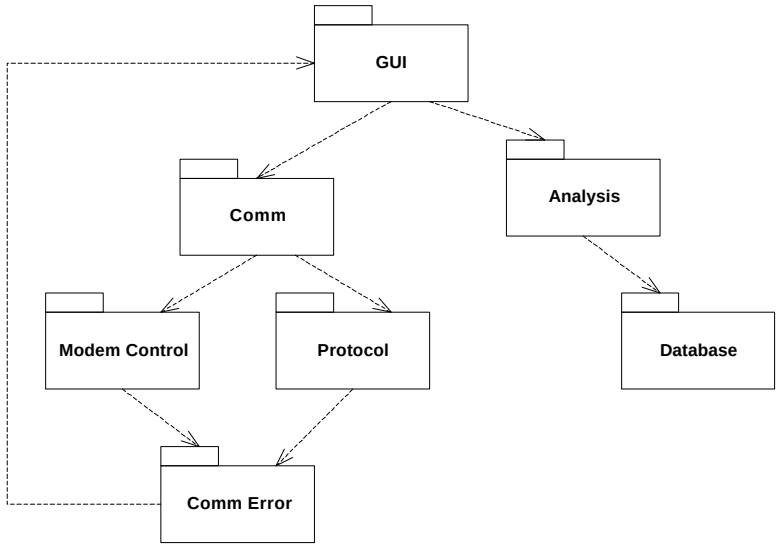

Figure 2-22
A cycle has been added.

Now what happens when the guys who are working on `Protocol` want to release their package. They have to build their test suite with `CommError`, `GUI`, `Comm`, `ModemControl`, `Analysis`, and `Database`! This is clearly disastrous. The workload of the engineers has been increased by an abhorent amount, due to one single little dependency that got out of control.

This means that someone needs to be watching the package dependency structure with regularity, and breaking cycles wherever they appear. Otherwise the transitive dependencies between modules will cause every module to depend upon every other module.

Breaking a Cycle. Cycles can be broken in two ways. The first involves creating a new package, and the second makes use of the DIP and ISP.

Figure 2-23 shows how to break the cycle by adding a new package. The classes that `CommError` needed are pulled out of `GUI` and placed in a new package named `MessageManager`. Both `GUI` and `CommError` are made to depend upon this new package.

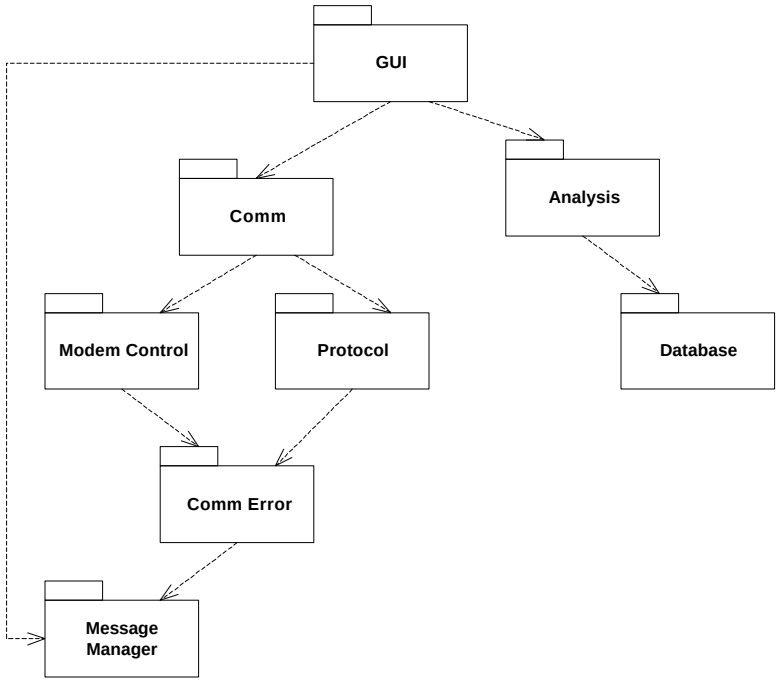

Figure 2-23

This is an example of how the package structure tends to jitter and shift during development. New package come into existence, and classes move from old package to new packages, to help break cycles.

Figure 2-24 shows a before and after picture of the other technique for breaking cycles. Here we see two packages that are bound by a cycle. Class A depends upon class X, and class Y depends upon class B. We break the cycle by inverting the dependency between Y and B. This is done by adding a new interface, BY, to B. This interface has all the methods that Y needs. Y uses this interface and B implements it.

Notice the placement of BY. It is placed in the package with the class that uses it. This is a pattern that you will see repeated throughout the case studies that deal with pack-

ages. Interfaces are very often included in the package that uses them, rather than in the package that implements them.

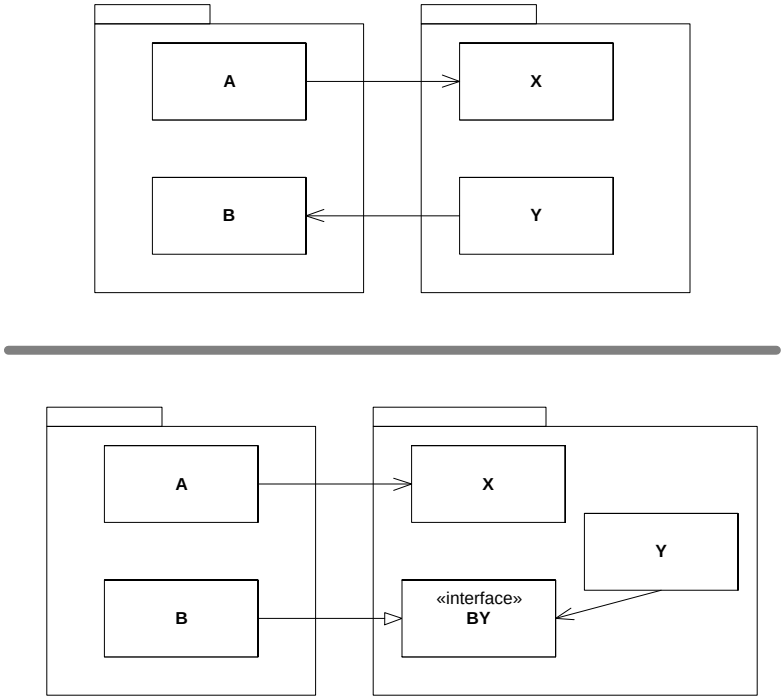

Figure 2-24

The Stable Dependencies Principle (SDP)[1]

Depend in the direction of stability.

Though this seems to be an obvious principle, there is quite a bit we can say about it. Stability is not always well understood.

Stability. What is meant by stability? Stand a penny on its side. Is it stable in that position? Likely you'd say not. However, unless disturbed, it will remain in that posi-

1. [Stability97]

tion for a very very long time. Thus stability has nothing direclty to do with frequency of change. The penny is not changing, but it is hard to think of it as stable.

Stability is related to the amount of work required to make a change. The penny is not stable because it requires very little work to topple it. On the other hand, a table is very stable because it takes a considerable amount of effort to turn it over.

How does this relate to software? There are many factors that make a software package hard to change. Its size, complexity, clarity, etc. We are going to ignore all those factors and focus upon something different. One sure way to make a software package difficult to change, is to make lots of other software packages depend upon it. A package with lots of incomming dependencies is very stable because it requires a great deal of work to reconcile any changes with all the dependent packages.

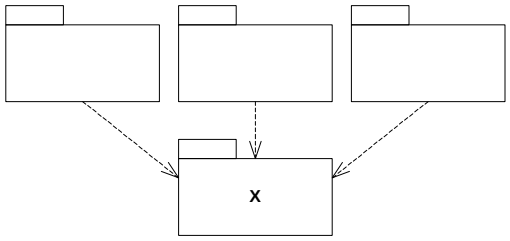

Figure 2-25
X is a stable package

Figure 2-25 shows X: a stable package. This package has three packages depending upon it, and therefore it has three good reasons not to change. We say that it is *responsible* to those three packages. On the other hand, X depends upon nothing, so it has no external influence to make it change. We say it is *independent.*

Figure 2-26, on the other hand, shows a very instable package. Y has no other packages depending upon it; we say that it is irresponsible. Y also has three packages that it depends upon, so changes may come from three external sources. We say that Y is dependent.

Stability Metrics. We can calculate the stability of a package using a trio of simple metrics.

Ca Afferent Coupling. The number of classes outside the package that depend upon classes inside the package. (i.e. incomming dependencies)

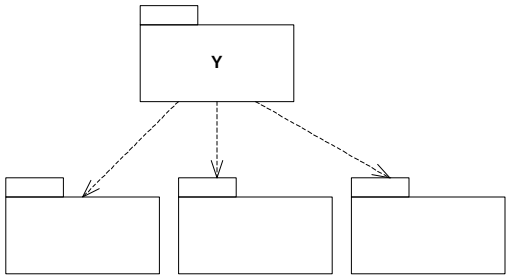

Figure 2-26
Y is instable.

Ce Efferent Coupling. The number of classes outside the package that classes inside the package depend upon. (i.e. outgoing dependencies)

I Instability. $I = \frac{Ce}{Ca + Ce}$. This is a metric that has the range: [0,1].

If there are no outgoing dependencies, then I will be zero and the package is stable. If there are no incomming dependencies then I will be one and the package is instable.

Now we can rephrase the SDP as follows: “Depend upon packages whose I metric is lower than yours.”

Rationale. Should all software be stable? One of the most important attributes of well designed software is ease of change. Software that is flexible in the presence of changing requirements is thought well of. Yet that software is instable by our definition. Indeed, we greatly desire that portions of our software be instable. We want certain modules to be easy to change so that when requirements drift, the design can respond with ease.

Figure 2-27 shows how the SDP can be violated. `Flexible` is a package that we intend to be easy to change. We want `Flexible` to be instable. However, some engineer, working in the package named `Stable`, hung a dependency upon `Flexible`. This violates the SDP since the I metric for `Stable` is much lower than the I metric for `Flexible`. As a result, `Flexible` will no longer be easy to change. A change to `Flexible` will force us to deal with `Stable` and all its dependents.

The Stable Abstractions Principle (SAP)[1]

Stable packages should be abstract packages.

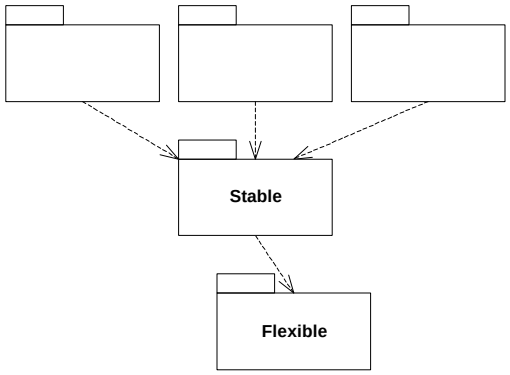

Figure 2-27
Violation of SDP.

We can envision the packages structure of our application as a set of interconnected packages with instable packages at the top, and stable packages on the bottom. In this view, all dependencies point downwards.

Those packages at the top are instable and flexible. But those at the bottom are very difficult to change. And this leads us to a dilemma: Do we want packages in our design that are hard to change?

Clearly, the more packages that are hard to change, the less flexible our overall design will be. However, there is a loophole we can crawl through. The highly stable packages at the bottom of the dependency network may be very difficult to change, but according to the OCP they do not have to be difficult to extend!

If the stable packages at the bottom are also highly abstract, then they can be easily extended. This means that it is possible to compose our application from instable packages that are easy to change, and stable packages that are easy to extend. This is a good thing.

Thus, the SAP is just a restatement of the DIP. It states the packages that are the most depended upon (i.e. stable) should also be the most abstract. But how do we meaure abstractness?

The Abstractness Metrics. We can derive another trio of metrics to help us calculate abstractness.

1. [Stability97]

Nc Number of classes in the package.

Na Number of abstract classes in the package. Remember, an abstract class is a class with at least one pure interface, and cannot be instantiated.

A Abstractness. $A = \frac{Na}{Nc}$

The A metric has a range of [0,1], just like the I metric. A value of zero means that the package contains no abstract classes. A value of one means that the package contains nothing but abstract classes.

The I vs A graph. The SAP can now be restated in terms of the I and A metrics: I should increase as A decreases. That is, concrete packages should be instable while abstract packages should be stable. We can plot this graphically on the A vs I graph. See Figure 2-28.

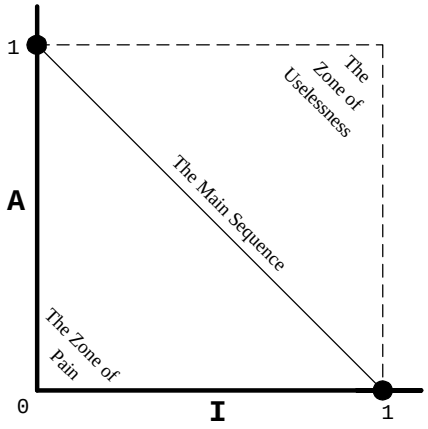

Figure 2-28
The A vs I graph.

It seems clear that packages should appear at either of the two black dots on Figure 2-28. Those at the upper left are completely abstract and very stable. Those at the lower right are completely concrete and very instable. This is just the way we like it. However what about package X in Figure 2-29? Where should it go?

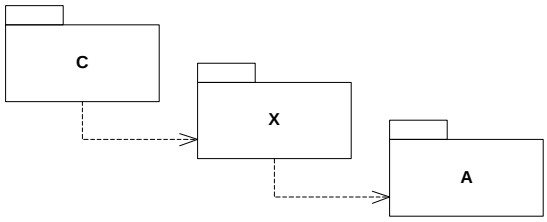

Figure 2-29
What do we put X on the A vs I Graph?

We can determine where we want Package X, by looking at where we don't want it to go. The upper right corner of the AI graph represents packages that are highly abstract and that nobody depends upon. This is the zone of uselessness. Certainly we don't want X to live there. On the other hand, the the lower left point of the AI graph represents packages that are concrete and have lots of incomming dependencies. This point represents the worst case for a package. Since the elements there are concrete, they cannot be extended the way abstract entities can; and since they have lots of incomming dependencies, the change will be very painful. This is the zone of pain, and we certainly don't want our package to live there.

Maximizing the distance between these two zones gives us a line called the *main sequence*. We'd like our packages to sit on this line if at all possible. A position on this line means that the package is abstract in proportion to its incomming dependencies and is concrete in proportion to its outgoing dependencies. In other words, the classes in such a package are conforming to the DIP.

Distance Metrics. This leaves us one more set of metrics to examine. Given the A and I values of any package, we'd like to know how far that package is from the main sequence.

D Distance. $D = \frac{|A + I - 1|}{\sqrt{2}}$. This ranges from [0,~0.707].

D' Normalized Distance. $D' = |A + I - 1|$. This metric is much more convenient than D since it ranges from [0,1]. Zero indicates that the package is directly on the main sequence. One indicates that the package is as far away as possible from the main sequence.

These metrics measure object oriented architecture. They are imperfect, and reliance upon them as the sole indicator of a sturdy architecture would be foolhardy. However, they can be, and have been, used to help measure the dependency structure of an application.

Patterns of Object Oriented Architecture

When following the principles described above to create object oriented architectures, , one finds that one repeats the same structures over and over again. These repeating structures of design and architecture are known as design patterns[1].

The essential definition of a design pattern is a well worn and known good solution to a common problem. Design patterns are definitively not new. Rather they are old techniques that have shown their usefulness over a period of many years.

Some common design patterns are described below. These are the patterns that you will come accross while reading through the case studies later in the book.

It should be noted that the topic of Design Patterns cannot be adequately covered in a single chapter of a single book. Interested readers are strongly encouraged to read [GOF96].

Abstract Server

When a client depends directly on a server, the DIP is violated. Changes to the server will propogate to the client, and the client will be unable to easily use similar servers. This can be rectified by inserting an anbstract interface between the client and the server as shown in Figure 2-30.

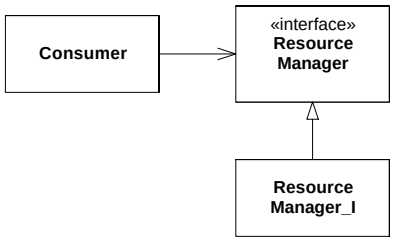

Figure 2-30
Abstract Server

The abstract interface becomes a “hinge point” upon which the design can flex. Different implementations of the server can be bound to an unsuspecting client.

1. [GOF96]

Adapter

When inserting an abstract interface is infeasible because the server is third party software, or is so heavily depended upon that it cannot easily be changed, an ADAPTER can be used to bind the abstract interface to the server. See Figure 2-31.

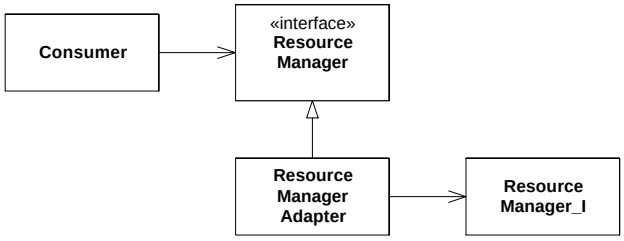

Figure 2-31
Adapter

The adapter is an object that implements the abstract interface to delegate to the server. Every method of the adpater simply translates and then delegates.

Observer

It often occurs that one element of a design needs to take some form of action when another element in the design discovers that an event has occurred. However, we frequently don't want the detector to know about the actor.

Consider the case of a meter that shows the status of a sensor. Every time the sensor changes its reading we want the meter to display the new value. However, we don't want the sensor to know anything about the meter.

We can address this situation with an OBSERVER, see Figure 2-32. The `Sensor` derives from a class named `Subject`, and `Meter` derives from an interface called `Observer`. `Subject` contains a list of `Observer`s. This list is loaded by the `Regsiter` method of `Subject`. In order to be told of events, our `Meter` must register with the `Subject` base class of the `Sensor`.

Figure 2-33 describes the dynamics of the collaboration. Some entity passes control to the `Sensor` who determines that its reading has changed. The `Sensor` calls `Notify` on its `Subject`. The `Subject` then cycles through all the `Observer`s

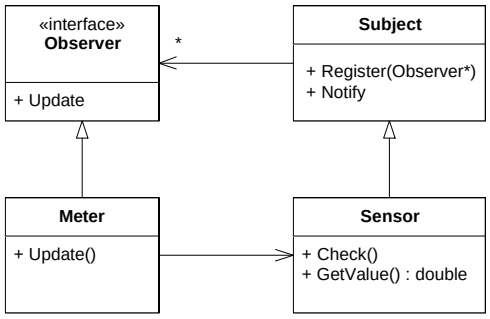

Figure 2-32
Observer Structure

that have been registered, calling `Update` on each. The `Update` message is caught by the `Meter` who uses it to read the new value of the `Sensor` and display it.

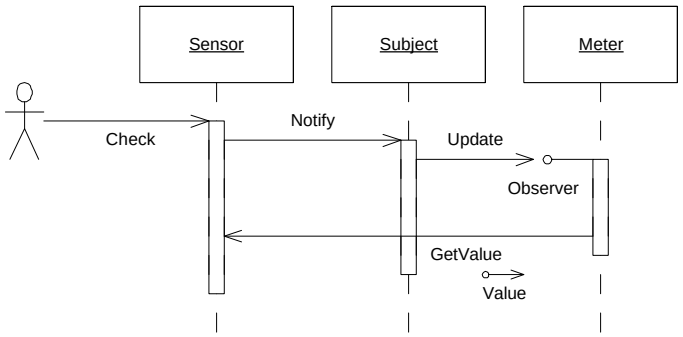

Figure 2-33

Bridge

One of the problems with implementing an abstract class with inheritance is that the derived class is so tightly coupled to the base class. This can lead to problems when other clients want to use the derived class functions without dragging along the baggage of the base hiearchy.

For example, consider a music synthesizer class. The base class translates MIDI input into a set of primitive `EmitVoice` calls that are implemented by a derived class. Note that the `EmitVoice` function of the derived class would be useful, in and of itself. Unforunately it is inextricably bound to the `MusicSynthesizer` class and the `PlayMidi` function. There is no way to get at the PlayVoice method without dragging the base class around with it. Also, there is no way to create different implementations of the `PlayMidi` function that use the same `EmitVoice` function. In short, the hiearchy is just to coupled.

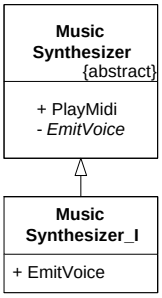

Figure 2-34
Badly coupled hierarchy

The BRIDGE pattern solves this problem by creating a strong seperation between the interface and implementation. Figure 2-35 shows how this works. The `MusicSynthesizer` class contains an abstract `PlayMidi` function which is implemented by `MusicSynthesizer_I`. It calls the `EmitVoice` function that is implemented in `MusicSynthesizer` to delegate to the `VoiceEmitter` interface. This interface is implemented by `VoiceEmitter_I` and emits the necessary sounds.

Now it is possible to implement both `EmitVoice` and `PlayMidi` separately from each other. The two functions have been decoupled. `EmitVoice` can be called without bringing along all the `MusicSynthesizer` baggage, and `PlayMidi` can be implemented any number of different ways, while still using the same `EmitVoice` function.

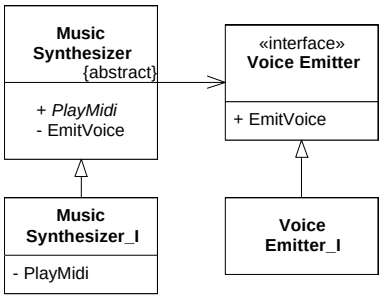

Figure 2-35
Hierarchy decoupled with Bridge

Abstract Factory

The DIP strongly recommends that modules not depend upon concrete classes. However, in order ot create an instance of a class, you must depend upon the concrete class. ABSTRACTFACTORY is a pattern that allows that dependency upon the concrete class to exist in one, and only one, place.

Figure 2-36 shows how this is accomplished for the `Modem` example. All the users who wish to create modems use an interface called `ModemFactory`. A pointer to this interface is held in a global variable named `GtheFactory`. The users call the `Make` function passing in a string that uniquely defines the particular subclass of `Modem` that they want. The `Make` function returns a pointer to a `Modem` interface.

The `ModemFactory` interface it implemented by `ModemFactory_I`. This class is created by `main`, and a pointer to it is loaded into the `GtheFactory` global. Thus, no module in the system knows about the concrete modem classes except for `ModemFactory_I`, and no module knows about `ModemFactory_I` except for `main`.

Conclusion

This chapter has introduced the concept of object oriented architecture and defined it as the structure of classes and packages that keeps the software application flexible, robust, reusable, and developable. The principles and patterns presented here support

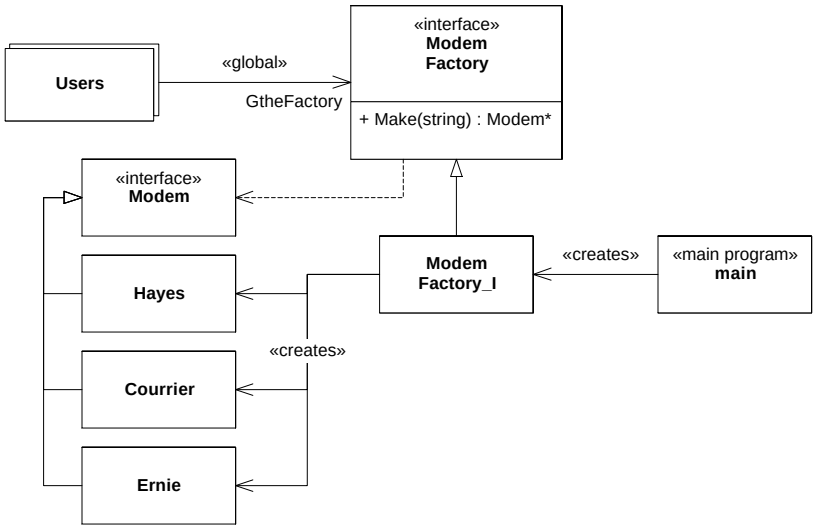

Figure 2-36
Abstract Factory

such architectures, and have been proven over time to be powerful aids in software architecture.

This has been an overview. There is much more to be said about the topic of OO architecture than can be said in the few pages of this chapter, indeed by foreshortening the topic so much, we have run the risk of doing the reader a disservice. It has been said that a little knowledge is a dangerous thing, and this chapter has provided a little knowledge. We strongly urge you to search out the books and papers in the citings of this chapter to learn more.

Bibliography

[Shaw96]: Patterns of Software Architecture (???), Garlan and Shaw, ...

[GOF96]: Design Patterns...

[OOSC98]: OOSC...

[OCP97]: The Open Closed Principle, Robert C. Martin...

[LSP97]: The Liskov Substitution Principle, Robert C. Martin

[DIP97]: The Dependency Inversion Principle, Robert C. Martin

[ISP97]: The Interface Segregation Principle, Robert C. Martin

[Granularity97]: Granularity, Robert C. Martin

[Stability97]: Stability, Robert C. Martin

[Liksov88]: Data Abstraction and Hierarchy...

[Martin99]: *Designing Object Oriented Applications using UML*, 2d. ed., Robert C. Martin, Prentice Hall, 1999.

