Chapter 8
Paying Attention to Principles and Wisdom

Another value that a profession provides to its members is to define the overall principles that can guide them, with a relatively high degree of reliability, to success.

A "quality" (see the previous chapter) is something in the nature of the code itself, and shows up in the choices we make when creating classes, methods, and relationships. A "principle" is a bit of wisdom about *all* designs, about those things that our profession tells us will tend to make them more successful overall.

A quality tells you what to pay attention to. A principle tells you what to shoot for, which way to lean, where your best bets are.

I am not going to claim to have found "the" set of principles that define the professional practice of software development, but rather I am going to offer up a few that I have found useful[1]. These, along with the "Practices" that follow in the next chapter, are a starting point, one that I hope we will all build upon together.

The principles I will focus on here are: Separating Use from Creation, the Open-Closed Principle, and the Dependency Inversion Principle. Also, I will look at the general wisdom offered by the "Gang of Four" to help us see how we can more reliably achieve designs that follow these principles.

Separating Use from Creation

A well-crafted system tends to separate concerns from one another. Focusing on cohesion does this, as it tells us to make sure that each entity has a single concern (or responsibility in the system).

Other concerns can be separated as well. Let's begin with one that I find is fairly often missed; the separation of the instantiation of entities ("creation") from their actual use.

---

[1] Nor do I claim to have invented all this. Much of it comes from Bob Martin and his company Object Mentor (www.objectmentor.com), with the possible exception of separating use from creation, which is something I have been working on for a while now.

Fowler's Perspectives

In his wonderful book *UML Distilled, 2nd Ed*[2]., Martin Fowler codified three "levels" of perspective from which one could consider an object-oriented design: Conceptual, Specification, and Implementation.

Considered conceptually, objects are entities with responsibilities, usually realized as Abstract Classes and Interfaces (in Java or C#), and which relate to each other in various ways to accomplish the goals of the application.

If I were an object, then the Conceptual Perspective would be concerned with "what I am responsible for."

At the Specification level, objects are entities that fulfill contracts that are specified in their public methods -- they promise services they can deliver in a specified way.

If I were an object, then the specification perspective would be concerned with "how I am used by others."

The Implementation Perspective is the "code level" or the actual programmatic solutions that objects use to fulfill these aforementioned contracts, and that therefore allows them to satisfy the responsibilities for which they were designed.

If I were an object, then the Implementation Perspective would be concerned with "how I accomplish my responsibilities."

Limiting the level of perspective at which any entity[3] in your system functions to one of these three has several advantages.

Similarly, limiting yourself to one of these perspectives during the mental process of designing any entity of your system is also advantageous.

Advantages:

1. It tends to reduce **coupling** in the system. If relationships between objects are kept at the abstract level, then the actual implementing subclasses are less likely to be coupled to one another. This is part and parcel of the advice given to us by the "Gang of Four" (The authors of the original Design Patterns[4] book), which states that we should "design to interfaces".

2. It tends to promote **cohesion** and clarity in the system, because we allow the details of the coded solutions to flow from the responsibilities that objects are intended to fulfill, and not the other way around. An object with a clearly-defined, limited responsibility is not likely to contain lots of extraneous methods and state that have nothing to do with the issue at hand.

3. It tends to give us opportunities to eliminate **redundancy** in systems, because the conceptual level entities we create (often abstract base classes) also give us a place to "push up" behavior and state that would otherwise be duplicated in implementation entities.

---

[2] *UML Distilled: A Brief Guide to the Standard Object Modeling Language, Second Edition*, by Martin Fowler, Addison-Wesley Pub Co, ISBN: 0321193687

[3] This could be a class, a method; some languages have other idioms such as delegates, etc.

[4] *Design Patterns* by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley Pub Co, ISBN: 0201633612

4.   It encourages **clarity** in our cognitive processes in general -- most people have a hard time keeping things straight when they attempt to think on multiple levels at the same time, and about the same issues.

Another Kind of Perspective

My purpose here, however, is to suggest another, similar distinction we can use in design to help us achieve the kind of flexibility and robustness that we are seeking in object-oriented solutions: the perspective of Creation vs. the perspective of Use.

Consider the following bit of code:

```java
public class SignalProcessor {

    private ByteFilter myFilter;

    public SignalProcessor() {
        myFilter = new HiPassFilter();
    }

    public byte[] process(byte[] signal) {
        // Do preparatory steps

        myFilter.filter(signal);

        // Do other steps

        return signal;
    }
}
```

Here a `SignalProcessor` instance is designed to use a `ByteFilter` implementation (`HiPassFilter`) to do a portion of its work.

This delegation is generally a good idea -- to promote good object cohesion, each class should be about one thing, and collaborate with other classes to accomplish subordinate tasks. Also, this will accommodate different kinds of `ByteFilter` implementations without altering the `SignalProcessor`'s design. This is a "pluggable" design, and allows for an easier path to extension[5]. Not surprisingly, it also creates an opportunity for cleaner and more useful testing.

Conceptually, `SignalProcessor` is responsible for processing the signal contained in a byte array. In terms of specification, `SignalProcessor` presents a `process()` method that takes and returns the byte array.

The way `SignalProcessor` is implemented is another matter, however, and there we see the delegation to the `ByteFilter` instance when the "filtering stuff" is needed. In designing `ByteFilter`, we need only consider its specification (the `filter()` method), and we can hold off considering its implementation until we are through here.

Good, clean, clear.

The problem, however, is that the relationship between `SignalProcessor` and `ByteFilter` operates at two different perspectives. `SignalProcessor` is "in charge" of

---

[5] This is an example of following the Open-Closed Principle, which we will look at shortly. For more information, see
http://www.objectmentor.com/resources/articles/ocp.pdf

creating the needed instance of `HiPassFilter`, and is *also* the entity that then **uses** the instance to do work.

This would seem trivial, and is in fact quite commonplace in routine designs. But let's consider these two responsibilities, *using* objects vs. *making* objects, as separate cohesive concerns, and examine them in terms of the coupling they create. As with Fowler's perspectives, we'll find that keeping these roles separate in systems will help us keep quality high, and ease the way for the evolutionary path that the system will follow.

The Perspective of Use

In order for one object to *use* another, it must have access to the public methods it exports. If the second object was held simply as "object", then only the methods common to all objects will be available to the using object, `toString()`, `Hashcode()`, and so forth. So, to make any meaningful use of the object-being-used, the using object must usually know one of three things:

- The actual type of the object-being-used, or
- An interface the object-being-used implements, or
- A base class the object-being-used is derived from.

To keep things as decoupled as possible, we would prefer one of the second two options, so that the actual object-being-used could be changed in the future (so long as it implemented the same interface or was derived from the same base class) without changing the code in the using object.

The using object (client), in other words, should ideally be coupled only to an abstraction, not to the actual concrete classes that exist, so that we are free to add more such classes in the future without having to maintain the client object. This is especially important if there are many different types of clients that use this same service, but it is always, generally, a good idea.

Put another way, we'd prefer that clients have abstract coupling to services.

A Separate Perspective: Creation

Naturally, if we exempt the client from "knowing" about the actual `ByteFilters` implementations that exist, and how they are constructed, that implies that something, somewhere will have to know these things.

I am suggesting that this is another distinction of *perspective*: **Use** vs. **Creation**. In the same way that the users of an instance should not be involved with its construction, similarly the builders of an instance should not be involved with its use. Therefore we typically call such a "constructing object" a *factory*[6]. Figure 8.1 describes a design along these lines.

---

[6] In the terminology I am promoting here, a "factory" is anything that produces an instance. It may build the instance on demand, or give out an instance built beforehand, or hand out an instance obtained from another source… from the perspective of the object using the factory, the details are irrelevant, and are therefore not coupled to.
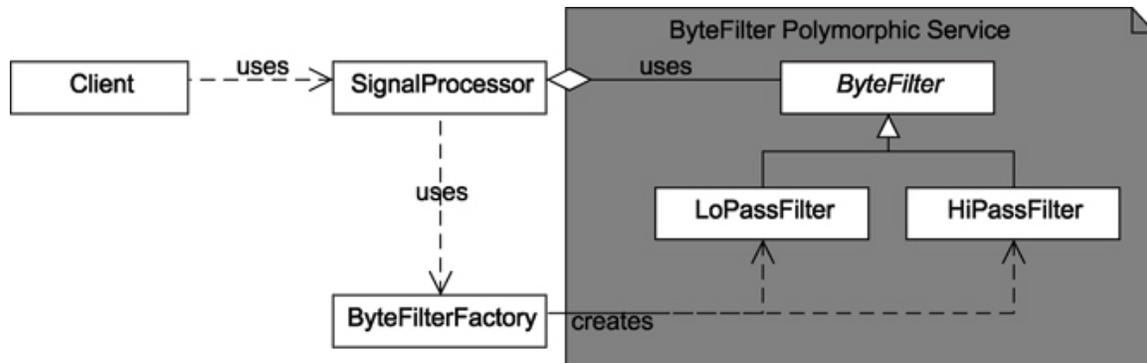
Figure 8.1
Use vs. Creation

What is critical to consider is the nature of the coupling from the two perspectives of **Use** and **Creation**, and therefore what will have to be maintained when different things change.

If you consider the ByteFilter abstraction and its two concrete implementations to be a "polymorphic service" (that is, that ByteFilter is a service with two versions, and this variation is handled through polymorphism), then SignalProcessor relates to this service from the use perspective while ByteFilterFactory relates to it from the creation perspective.

The coupling from SignalProcessor to the ByteFilter polymorphic service is to the identity of the abstract type ByteFilter (simply that this abstraction exists at all) and the public methods in its interface. There is no coupling of any kind between SignalProcessor and the implementing subclasses HiPassFilter and LoPassFilter, assuming we have been good object-oriented programmers and not added any methods to the interfaces of these subclasses.

The coupling from ByteFilterFactory to the ByteFilter polymorphic service is quite different. The factory is coupled to the subclasses, since it must build instances of them with the "new" keyword. It also, therefore, is coupled to the nature of their constructors. It is also coupled to the ByteFilter type (it casts all references it builds to that type before returning them to the SignalProcessor), but not the public methods in the interface – *if we have limited the factory to the construction perspective only*. The factory never calls methods on the objects it builds.[7]

The upshot of all this is to limit the maintenance we must endure when something changes to *either* the users of this service *or* the creator of the specific instances.

If the subclasses change – if we add or remove different implementations of ByteFilter, or if the rules regarding when one implementation should be used vs. another happen to change – then ByteFilterFactory will have to be maintained, but *not* SignalProcessor.

If the interface of ByteFilter changes – if we add, remove, or change the public methods in the interface – then SignalProcessor will have to be maintained, but *not* ByteFilterFactory.

It is interesting to note that there is one element of the design that both users and creators are vulnerable to: the abstraction ByteFilter itself. Not its interface, but its existence. This realization points up the fact, long understood by high-level designers, that finding the right abstractions is among the most crucial issues in object-oriented design. Even if we get the interfaces wrong, it is not as bad as missing an entire abstract concept.

---

[7] For our purposes here, we do not consider the constructor to be part of the interface of an object.

5

If we consider systems as evolving, emerging entities, as opposed to fixed and planned entities, then we know we'll want to limit the impact of change, overall. This principle helps us, because it tends to place the concrete coupling in a single entity: an object factory.

The meaning of clean separation

Clean separation means that the relation between any entity A and any other entity B in a system should be limited so that A makes B or A uses B, but never both.

Considering Construction Details Last

The notion that entities in a design have perspectives at which they operate implies a kind of cohesion. Cohesion, as I have said, is considered a virtue because strongly cohesive entities tend to be easier to understand, less-tightly coupled to other entities, and allow for more fine-grained testing of the system.

If we strive to limit the perspective at which any entity operates, we improve its cohesion, and the benefits are similar to those gained by cohesion of state, function, and responsibility.

The perspective of use vs. the perspective of creation is one powerful way to separate entities for stronger cohesion. It also implies that construction will be handled by a cohesive entity, a factory of some kind, and that we should not have to worry about how that will be accomplished while we are determining the use-relationships in our design.

In fact, leaning on this principle means that we can allow the *nature* of the use relationships in our design to determine the *sorts* of factories that will be the best choice to build the instances concerned (there are many well-defined creational patterns).

In their book *Design Patterns Explained: A New Perspective on Object-Oriented Design*, Alan Shalloway and James Trott illustrated the notion of "design by context", wherein they showed that some aspects of a design can provide the context by which one can determine/understand other aspects. The notion is a big one, implying much about the role of patterns in design, analysis, and implementation, but a key part of their thesis was this:

> "During one of my projects, I was reflecting on my design approaches. I noticed something that I did consistently, almost unconsciously: I never worried about how I was going to instantiate my objects until I knew what I wanted my objects to be. My chief concern was with relationships between objects as if they already existed. I assumed that I would be able to construct the objects that fit in these relationships when the time comes to do so. "[8]

> They then synthesized a powerful, universal context for proper design, as follows:

> "Rule: Consider what you need to have in your system before you concern yourself with how to create it."

I had a hard time with this, at first. Before I was an object-oriented developer, I was a procedural programmer, and in those days one's program tended to load all at once, and then run; a very straightforward process. The idea that parts of my program, called "objects" would be loading and running at various times throughout the runtime session seemed like a huge problem, and the first issue to resolve. The idea that it was actually the *last* issue to resolve took quite a

---

[8] *Design Patterns Explained: A New Perspective on Object-Oriented Design*, by Alan Shalloway and James R. Trott, Addison-Wesley Pub Co, ISBN: 0201715945

leap of faith on my part, but I must say it has always proven to be the right path. This is an example of how a profession can provide you with wisdom that is hidden from the uninitiated, or even counter-intuitive from the relative neophyte's point of view.

At any rate, the separation of use and construction empowers you to follow this rule, and therefore to derive the benefits that flow from following it. Your system will have stronger cohesion, will be more extensible and flexible, and the task of maintenance will be significantly simplified.

The Real World

The implication, of course, is that every class should have a factory that instantiates it. Am I really suggesting this? In fact, if you follow the logical implications of this book it might lead to believe that for every object in our design we should have the following:

- An interface or abstract class that hides it
- A factory that builds it
- A unit test that explains it
- A mock object or set of mock objects that decouple it

For *every class*? No. What I want is the *value* of having done this, without doing it until it is actually necessary. I am greedy, you see… I want all the benefits of this approach, but I do not want to pay anything to get them, or until I actually need them. I want the juice without the orange, I want to have my cake and eat it too. Don't you?

To satisfy my greed, I will want to lean on some key "practices", which is the subject of the next chapter.

The Open-Closed Principle

We draw upon the wisdom that comes from the traditions of our profession. Some of this is relatively new, some is not.

Many years ago, Ivar Jacobson said: "All systems change during their life cycles. This must be borne in mind when developing systems expected to last longer than the first version."

I think I can feel fairly safe in saying that the modern economy expects pretty much all software to last longer than its first version. Hm. Well, maybe if our software is burned onto a chip, and sent to Jupiter on a probe, then we can assume we will not have to maintain it. But for everything else…

Somewhat later, Bertrand Meyer put it another way: "Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."

This has come to be known as the Open-Closed Principle.

The overall idea is this: as much as possible, we would like to set things up in our architecture in such a way that when things change, as we know they will, we will be able to accommodate each change by writing new code and cleanly plugging it into the existing system, rather than having to go back and modify the code that already exists.

Again, in the courses I teach, a question I often ask is (by a show of hands) how many of the students would rather work on older code, in maintenance mode, as opposed to starting a brand new project. I almost never get a single hand raised.

Why is this? I think we can learn a little about the nature of software by paying attention to our own natural reactions. We would rather work on new software rather than changing old code for a variety of reasons, including:

- Writing new stuff tends to be more interesting. We like to be creative, and when we design something new it feels more stimulating in this way.

- It is a chance to get things right. Before we gave up hope, we worked under the belief that maybe *this time* we will get everything right and never have to change it. That hope, as misguided as it is, feels good.

- Maintenance is a chance to fail. It is a chance to be the person who broke the system, and naturally is the one expected to fix it.

I think this third reason is a very strong motivating force in software development.

Open-closed is a quality of your design. A design can be "a little open-closed", "somewhat open-closed", or "very open-closed". I bear no illusions that any code can be *completely* open-closed, but it is a good goal to try and get as much of it as you can.

Put another way, if you have two ways of solving a problem, all other things being equal you should choose the one that is relatively more open-closed. It will make your life easier in the future. One major value of studying patterns is that they all, whatever else it true about them, tend to be more open-closed than the alternatives.

At the time that Meyer coined the term, he almost certainly was referring to the use of inheritance to extend classes[9], rather than changing them. As we'll see when we investigate what the Gang of Four suggested as an alternative, we'll see that this principle can be adhered to in many ways, and doe not necessarily lead to the overuse of inheritance.

Open-Closed at the Class Level

If you look back at the previous section in this chapter, the ideal design for polymorphic service called "`ByteFilter`" was extremely open-closed, as shown in Figure 8.2.
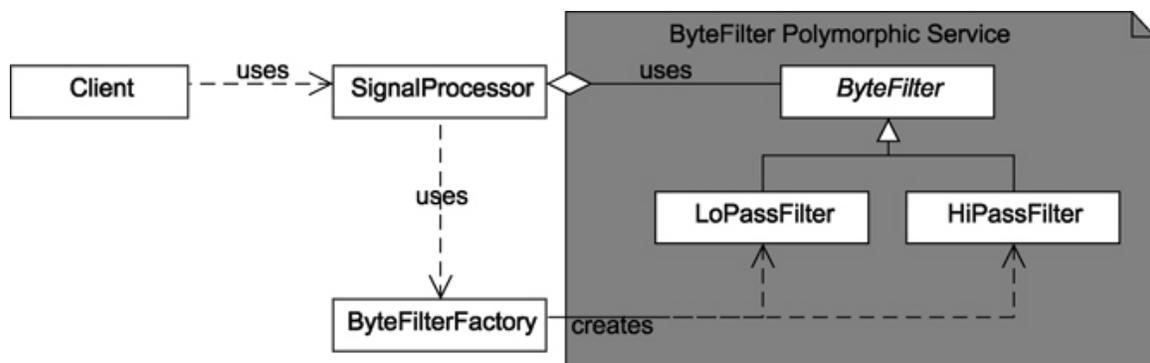
Figure 8.2
The `ByteFilter` Polymorphic Service

If we get a new requirement that says "we have to accommodate a square wave filter now", we can make the needed change by writing a *new* derived class of the `ByteFilter` abstraction, and the only code that will have to change is in the `ByteFilterFactory` class. The

---

[9] http://en.wikipedia.org/wiki/Bertrand_Meyer

`SignalProcessor` class is only coupled to the `ByteFilter` abstraction, so the code within it will not have to be touched at all. The class would not even have to be recompiled. This is also true of the Client, and any future users of this `ByteFilter` polymorphic server.

This is an example of using delegation, rather than inheritance, to extend a system's flexibility. You'll note that `SignalProcesser` does not change.

My experience in designs that use factories (pretty much all of my designs use factories) is that while there will tend to be *many* consumers of a service[10], that there will usually be only a *single* factory to build the proper instances. So, while I may fail to "be open-closed" where the factory is concerned, at least I am "very open-closed" everywhere else in the system, at least as regards this service.

Factories, therefore, promote the Open-Closed Principle. They help us to focus whatever code changes that will be needed in a single place.

Why bother? When we can achieve this kind of class-level adherence to the Open-Closed Principle, we lean very heavily on encapsulation. The new thing we will have to create to accommodate the "square wave" requirement will be its own class, encapsulated away from all other entities in the system. The fact that we do not touch any of the other classes (except for the factory) means that they are protected also.

Remember that one of the main negative forces in traditional development has always been the unexpected side-effects that can come from change. Here, what we have done is change what change is, from altering code to adding code.

We have changed the rules of the game such that we do what we like to do (write new code) more often than we do what we would rather not do (maintain old code).

The Kobayashi Maru

I'm a bit of a nerd, I'll admit. I've seen all things Star Trek, and I can recite lines from most of the episodes and films.

In "Star Trek II: The Wrath of Khan", the filmmakers began with a trick played on the audience… the Enterprise encounters an ambush, and is destroyed, killing everyone. But it turns out that this is a simulation, the final exam for starship captains at Starfleet, called the Kobayashi Maru, and is an un-winnable scenario by design.

Why they'd want to send a captain off on a failure was puzzling to me, but so be it. We learn, however, later in the film, that one graduate did, in fact, win the un-winnable scenario. Captain Kirk, of course.

How did he win? He's reluctant to say, but finally admits that he won by cheating. He re-programmed the simulation so it could be won.

I think of the open-closed principle as sort of the Kobayashi Maru if OO design. I'd rather write new software than monkey around with old software, so I set myself up to win by changing the game. When something is open-closed, "changing" it *is* writing new software.

---

[10] In fact, one could argue that the more clients a service develops over time, the more successful and valuable the service has proven to be. We *hope* we'll get a lot of use out of any given service, which will tend to mean many clients as the service persists.

Open-Closed at the Method Level

If we follow some of the practices in the next chapter (especially Encapsulating Construction), we will find that this class-level of the Open-Closed Principle will be available to us more often than it would otherwise be.

However, we do not work in a perfect world, and certainly changes will come along that require us to alter the code of existing classes.

That said, if we pay attention to method cohesion then the granularity of our methods will increase the probability that code changes within classes can be limited to adding methods, or if a method does have to be change, to changing the code in a single method that is, at least in terms of its temporary method variables, relatively encapsulated from the rest of the class.

What I am trying to avoid is changing code that is entwined with other code; I want to get away, as much as possible, from "wading in" to a mass of spaghetti that I only partly understand.

I am going to go over this in detail in the next chapter when I discuss Programming by Intention, so I will leave this for now… except to say that this is another example of how following professional practices can be a kind of "safety net" that protects you from having to make dangerous change.

The Dependency Inversion Principle

Another key decision that we have to make in design has to do with the public interfaces of our classes, and the signatures of our methods.

When you design the public interface (public methods) of a class, or the public interface (signature) of a method, there are two possible ways to make this decision:

- Code up the functionality in question, and then decide what sort of interface to provide to the rest of the system so that the functionality can be used, or
- Determine how the rest of the system would "prefer" to use this functionality (an ideal interface from the "outside perspective"), and then determine how to satisfy this need by implementing accordingly.

In other words, you can decide how a thing will work, then how it will be used, or you can determine how a thing will be used, and then how it works.

Traditionally, we have started by writing the implementing code.

This seems logical. Once we know how the class or method works, then we can decide how to expose its services to others. It turns out, however, that the opposite view is the stronger one.

We call this "Dependency Inversion"[11] because of this relationship, shown in Figure 8.3.

---

[11] "We" being Alan, David, Amir, Rob, Rod, and I. Bob Martin, who coined the term, may have a different idea of what it means. Visit http://www.objectmentor.com/resources/articles/dip.pdf for his take on the DIP.
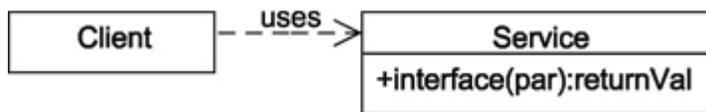
Figure 8.3
Dependency Inversion

We can, quite accurately, state that the Client here has "a dependency" to Service. It uses Service to get some work done, and so it "depends" on Service (and if you change Service's interface, you will have to change the way Client uses it, certainly).

However, we are stating that the way we determine what the interface of service should be *in the first place* is by determining what would be most convenient for Client, before we even create Service's actual implementing code.

So, Client *depends* on Service, but the interface of Service *depends* on what Client wants it to be.

Why do it this way?

Because the way a particular piece of functionality is accomplished is very likely to change, but the needs of the client objects will tend to be more stable. Therefore, paying attention to the needs of the client object first will tend to produce more stable interfaces, which leads to less/simpler maintenance.

There is a practice called Programming by Intention, which I will cover in the next chapter, which helps to produce this inversion pretty much automatically, at the method level. Similarly, Test-Driven Development, a discipline I will cover a few chapters on, tends to force you to do this at the class level.

Practices and disciplines do these things for us, which is why I am focusing on them in this book. However, understanding the principles behind a better design is also important; no set of rote practices will, by themselves, make you a professional.

Advice from the Gang of Four

The first two chapters of the Gang of Four's book is actually a treatise on object-oriented design. The wisdom they shared there is fundamental to the patterns; that is, the various patterns are contextual applications of these key concepts.

In a way, the actual patterns themselves are not really what is important about patterns. They are useful to know because they give us a leg-up on good design, help us to communicate at a higher level with others in our profession, and they give us a way to capture what we know about particular, repeated situations in development.

However, what makes them important is that each pattern is an example of following the general advice that was presented in the initial chapters of the book, which itself is really a particularly clear-eyed and wise view of what good object-orientation is in the first place.

Here is what they said, and what I think it means:

GoF: "Design to Interfaces"

My first reaction to this was "Sure. What choice do I have?"

In my view at the time, the interface of an object is the collection of public methods that represent it to the rest of the system, and in a language like Java or C#, the public methods are all I have access to. If by "design to" they meant "create relationships between", then the interface was all I could possibly choose.

Obviously, this is not what they meant.

Design is more than this. It is the structure of the classes, where the responsibilities are, how information is held and flows from one place to another, how behavior will be varied under different circumstances, etc…

So, by "Design to Interfaces", the Gang of Four was really talking about two things:

- The assignment of responsibilities between client and service objects, and
- The use of abstractions to hide specifics "implementing" classes.

Designing to the Interface of a Method

For an example of this first idea, I got clear on this when my friend Jeff McKenna (of AgileAction) conducted a very interesting exercise in a class we were co-teaching.

He went to a student in the front row and said "tell me your driver's license number".

The student, after blinking for a moment or two, reached into his back pocket, pulled out his wallet, opened it, removed his driver's license, and read the number to Jeff.

Jeff asked the next student the same question. He did precisely the same thing.

Jeff then picked another student, a young lady, who was seated nearby. He said to her "tell me your driver's license number."

She picked up her purse from the floor, opened it, took out a snap-purse from inside, opened that, took out her wallet, opened that, and read the license number from the license without removing it from it's glassine envelope inside the wallet.

He went from student to student, asking the same question in the same way. One of the students actually could recite his driver's license number from memory (perhaps he got pulled over a lot by the police). And then there was the guy who refused to tell Jeff his driver's license number – apparently, he threw an exception.

At any rate Jeff's point was that he took *no different action* with each student in order to obtain what he wanted, even though the actual steps each student went through to satisfy his request were different from student to student.

Jeff's "design," his plan of action, what he said and what he expected as a return, was to the interface of the student.

As a counter-example, imagine if Jeff had gone to the first student and said "reach into your back pocket, pull out your wallet, open it, remove your driver's license, and read the number to me." This would work with the majority of the male students in the class, but as soon as he got to the young lady with the purse, or the guy who had the number memorized, this request would no longer work.

Jeff, therefore, would either have to limit himself to wallet-on-the-hip students, or would have to know a lot of different ways of asking for a license number, and know when to use each one. He would also, in each case, have to know which sort of student he was dealing with, and would have to add more complexity to his "design" when new students arrived who stored their drivers license number in yet different ways.

If he had been implementation-specific in his design, Jeff would have severely limited his flexibility, because the responsibility of *how* to satisfy his request would be placed in the wrong spot. It would also:

- Weaken Jeff's *cohesion*. He would have to know non-Jeff stuff (how each student stored their driver's license number) as well as his own issues (of which Jeff, believe me, has many!).

12

- Create accidental *coupling* between Jeff and this particular group of students. If the first student started carrying his license in his shirt pocket, later, then Jeff would have to change the way he interacted with him.

By placing the specifics of implementation in each student, and keeping the interface generic, this means Jeff stays simple (well, simpl*er*). It also means that we can, in the future, have a student come along that stores her driver's license in some way we never thought of (in her shoe, or tucked in her hat band), and Jeff could take the same action as before.

There is a practice in the next chapter that helps us get this right most of the time, without even trying. It is called "Programming by Intention". I will leave that for now, except to say that this practice helps us to follow the "Design to Interfaces" advice at the *method level*.

Designing to the Interface of a Class

Another way to consider this piece of advice has to do with class relationships. The Gang of Four was suggesting that when you have abstractions in a design, as shown in Figure 8.4.
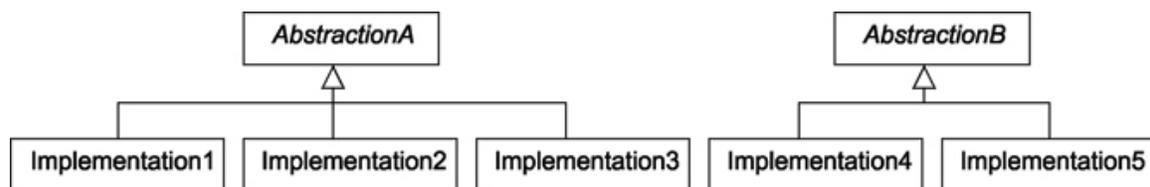
Figure 8.4
Abstractions in a design

…you should, whenever possible, try to create the relationship between these abstractions at the highest level possible, as shown in Figure 8.4, between AbstractionA and AbstractionB.
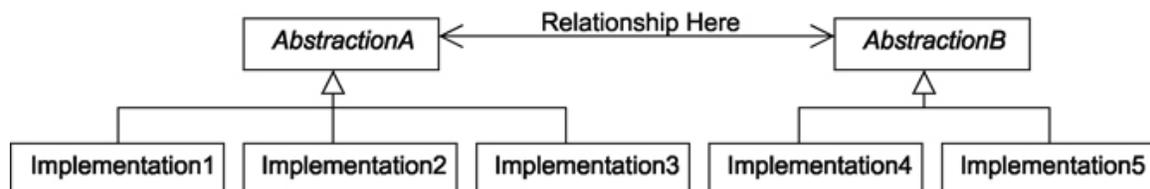
Figure 8.5
Relationship between abstractions

I sometimes paraphrase the Gang of Four here and call this "Design to Abstractions". There are several advantages to this:

- There will be, quite simply, fewer relationships. If I created relationships among the implementation classes, above, I would potentially end up with 6 of them. With the relationship "high up" at the abstract level, there will be one-to-one, or perhaps one-to-many, rather than n-to-m.
- The derived, implementing classes will not be coupled to one another. If both Jeff and I talked to the students in the same way, then the students would not be coupled to which teacher was asking for the license, any more than the teacher would be coupled to the particular student being asked. The relationship would be from "Teacher" to "Student", which are the abstractions in this example.

- It is open-closed. We can add a derivation to either/both sides of the relationship without requiring a change in the code.

Designing to interfaces is a kind of limitation we impose on ourselves, and the benefits it yields are many.   It is an example of how a profession can "learn things" that benefit all the members of the profession, rather than expecting that each member will learn hard lessons on his own.

## GoF: "Favor Object Aggregation[12] Over Class Inheritance"

The Gang of Four also said that we should "favor" an aggregation approach over the use of inheritance to vary behavior. The first time I read this I said "okay, I get that. Delegate, do not inherit." Then I proceeded to read the individual design patterns in the book, and inheritance was used everywhere.

I thought "if I am not supposed to use inheritance, how come they are doing it every time I turn around?" Obviously, again, I was misunderstanding what they meant.

First, they said "Favor" aggregation over inheritance. They did not say "never use inheritance, always delegate".

Secondly, what they really meant was that we should use inheritance for something other than what I would traditionally have used it for, which was to specialize existing concrete objects with new behavior. This was, in fact, the original meaning of "open-closed". Now the Gang of Four was suggesting we use inheritance in a different, more powerful way.

Going back to the signal processor example, an older (perhaps more recognizable) approach to varying its filter mechanism would have been to do the approach shown in Figure 8.6.
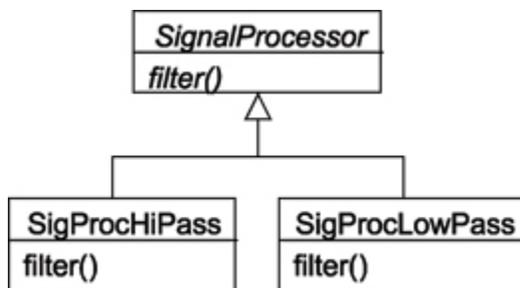
Figure 8.6
*A different signal processor approach*

The base class, `SignalProcessor`, implements all the behaviors, except (perhaps) `filter()`. The `filter()` method would either be abstract in the base class, or the base class would implement it with default behavior.

Either way, the derived classes would override `filter()` with the specific filtering behavior desired.

---

[12] The GoF actually used the term "Composition". In the days of unmanaged code, Composition and Aggregation implied different styles of memory management and cleanup when objects were no longer needed. I think this is a distraction for us here, and when you consider that the OMT and UML use these terms with precisely opposite meanings, I think it is best to stick with the more general notion of "Aggregation" or, if you prefer, "Delegation". In C++, this may actually be Composition, depending on which entity is responsible for cleaning up the memory of the service object(s).

This works, of course, and is a very commonplace way of accomplishing and containing a variation in traditional object-orientation. What the Gang of Four was saying was… do not do this, unless you have some very compelling reason to do so. A better way, which they are suggesting you should "favor", is shown in Figure 8.7.
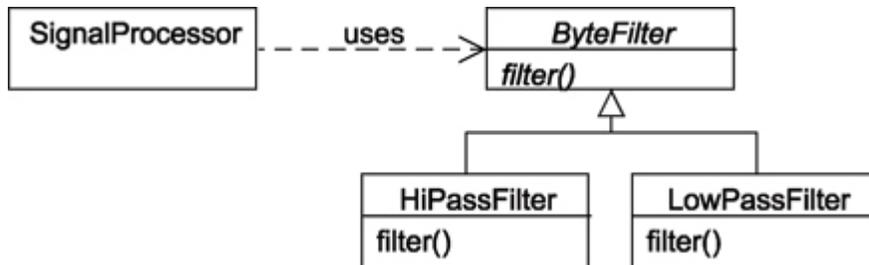
Figure 8.7
A better way is to use inheritance to categorize

The first design uses inheritance to "specialize" a real thing, SignalProcessor. The second design uses inheritance to "categorize" the HiPassFilter and LowPassFilter as *conceptually the same thing* and therefore the base class is a concept, ByteFilter, not a concrete thing.

The advantages are several and, of course, tie back to the qualities I emphasized in the last chapter:

- Both SignalProcessor and the various filters will be more *cohesive* because each will be about fewer issues internally.
- The filters can be tested outside the context of the specific usage of SignalProcessor, and therefore are more *testable*.
- The coupling between SignalProcessor and the various filters is kept to the abstract level, and therefore the *coupling* is kept to a minimum, and only where it is most logical.
- There will tend to be fewer *redundancies*, because anything common to all filters can be placed in the filter base class.

Also, consider the runtime flexibility. In the first design, imagine that we had instantiated the SignalProcessorHP class, and it was up and running. Now imagine that, in the same runtime, session, we had to switch to the lo-pass filter… the only way to do this in the first design is to make a new instance of SignalProcessorLP, somehow transfer the state from the previous instance to the new one, and then kill SignalProcessorHP.

The real problem with this, apart from potential performance problems, is the "somehow transfer the state" step. Either we would have to:

- Break encapsulation on the state of the SignalProcessor, or
- Make each version of SignalProcessor able to clone itself into the other versions, which would, of course, couple them to each other.

Neither option seems very good.

The design using delegation, of course renders this issue moot, because we can simply provide a mechanism (a setFilter() method) on the SignalProcessor which will allow us to give it a different ByteFilter subclass any time we want to switch the filtering algorithm,

without breaking encapsulation on `SignalProcessor`'s state, or creating any unneeded and undesirable coupling between the various versions of the filter.

Even if we start out without this `setFilter()` method, perhaps because this dynamic aspect is unneeded or even undesirable, we can later make it dynamic by adding the method, without needed to touch any of the existing code in the service. Adding a new method, and leaving existing methods alone, is also open-closed.

Also, what if something else starts to vary? This seems to be so commonplace as to be expected: something else, unforeseen in the original design, now starts to vary. Imagine in the `SignalProcessor` there is a `prepare()` step which was initially fixed, and now has two versions: `LossyPreparation` and `PrecisePreparation`.

In the inheritance-heavy approach, we would simply inherit again (see Figure 8.8).
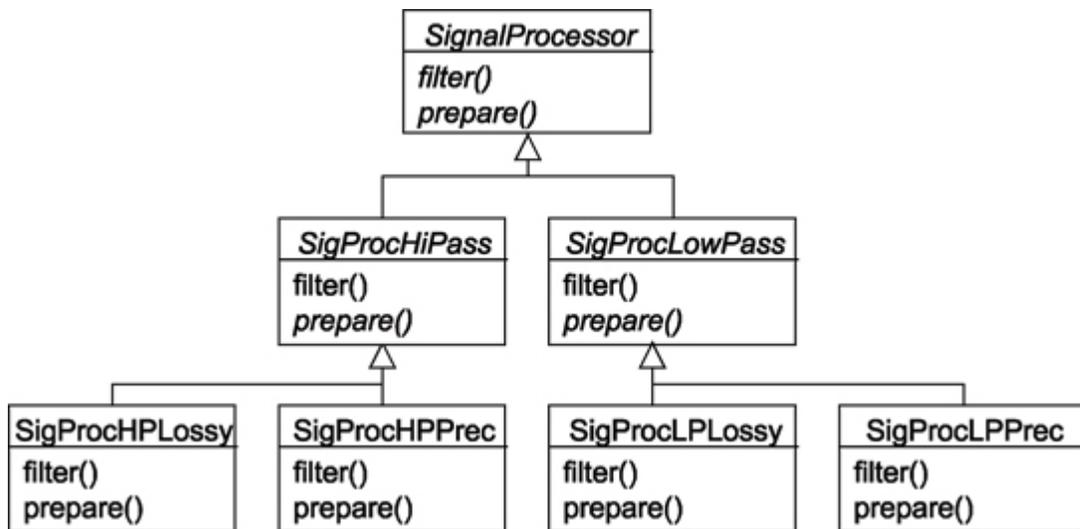
Figure 8.8
The inheritance-heavy approach

Note that the cohesion, testability, and dynamism problems are getting worse and worse here. It is also getting harder and harder to name these classes in such a way that they will fit nicely in my pages here.

On the other hand, using delegation, as shown in Figure 8.9 scales more readily.
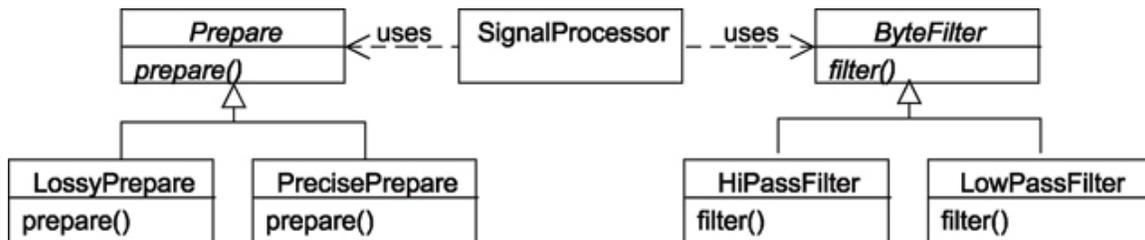
Figure 8.9
Using delegation

16

Not only is the cohesion of `SignalProcessor` not getting any worse, it is actually getting better each time we "pull an issue out", and more cohesion leads to easier and more useful tests.

And notice: we are focusing on larger issues than simply "does it work". Both approaches will "work."

As professionals, we are asking additional questions:

"How well will each of these stand up over time?"

"What kind of return on investment will each provide for my customer?"

"Which represents my desire to build things that will last, and continue to be valuable?"

GoF: Consider what should be variable in your design and encapsulate the concept that varie*s*

I must admit that this statement really blew my mind when I first read it. It literally stopped me cold, and made me wonder if I should even keep reading the book.

"Encapsulate the concept?" I thought. "How do you encapsulate a concept?" I knew how to encapsulate *data*, which was the point of objects in the first place (I thought), and I know how to encapsulate *function*, which was the advantage of methods and functions over old-style subroutines – you supply the inputs and receive an output, but you have no other coupling to a method or function. But… encapsulate… a *concept*?

I was envisioning code like "`private Beauty`" or some such. It just did not make sense to me. What did concepts have to do with code?

Of course, the reason it did not make sense was my limited idea of what "encapsulation" is in the first place.

I thought of encapsulation as the hiding of data, or, as I gained some sophistication with object-orientation, the hiding of implementation. In truth, that is just the tip of the iceberg… Encapsulation is the hiding of anything at all.

This realization takes us in many fruitful directions, but here, in examining the general design advice of the Gang of Four, we can focus ourselves on the specific idea that *variation can be encapsulated conceptually*.

Let's go back to the delegation approach with the Signal Processor, shown in Figure 8.10.
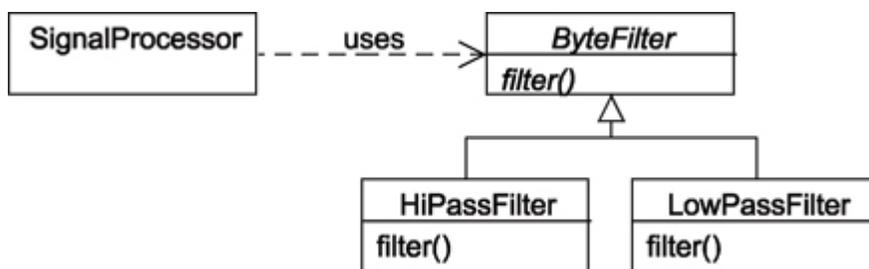
Figure 8.10
The delegation approach with Signal Processor

If you think of encapsulation as the hiding of *anything*, then you can say that the abstract class `ByteFilter` *encapsulates* the derived classes `HiPassFilter` and `LoPassFilter`,

because `SignalProcessor` cannot see them, does not know they exists, nor which one is actually being used during any given runtime session.

`ByteFilter` is, of course a conceptual class. It represents the "idea" of filtering bytes, and how another entity would interact with such a filter (the interface), without specifying any particular filtering algorithm.

This was what the GoF was suggesting… that we look for things we can encapsulate, and encapsulate them conceptually wherever we can. The benefits are many:

- We are free to change (add, remove, modify) any of the implementations of `ByteFilter` without changing `SignalProcessor` at all. We have achieved a high degree of "Open-Closed-ness".

- The coupling between `SignalProcessor` and the `ByteFilter` service is purely to the Interface.

- We get the dynamism mentioned above, in that we can change `ByteFilters` without remaking the `SignalProcessor`.

- Each class in the relationship will tend to be easier to test, individually, because it is more cohesive.

But wait… did not we get all these benefits by following those first two pieces of advice: "Design to Interfaces" and "Favor Aggregation Over Inheritance"?

Yes, and in a sense this third point is the point-of-all-points for the Gang of Four. But in the chaos of software development, our situational awareness varies greatly, and so having multiple ways to think of a problem increases the chance that we will not miss a critical issue.

Testing relates to cohesion. Cohesion relates to encapsulation. Encapsulation relates to redundancy. Redundancy relates to testing. Our goal is to find the truth, to find a design that reflects the true nature of the problem, and there are many ways to find the truth.

The main reason I have been focusing on the idea of "building a true profession" is that I believe the qualities, principles, practices, disciplines, and overall guidance we'll get from such a thing will greatly empower us to find these highly-appropriate designs, and thus increase our success tremendously.

Furthermore, when the Gang of Four said we should encapsulate "variation", they did not simply mean "varying algorithms" or "varying behavior" or even "varying state". They meant… varying *anything*.

What else can vary? Many, many things, it turns out, and this accounts for the fact that there are many different design patterns, where each one encapsulates some different varying thing.

For example, let us say we have a Client object that delegates to a single service object. Then, in a different situation, it delegates to two service objects. To handle this variation in code we'd do something like this:

```
if (someConditional) {
      Service1.doOperation();
} else {
      Service1.doOperation();
      Service2.doOperation();
}
```

…or something similar. This is not open-closed, because if we add a case where 3 services are needed, we'll have to change this code. However, the Gang of Four would also tell us that

this variation (1, 2, or more) is not encapsulated. It's an issue of "cardinality", and if possible we should encapsulate that too.

Can we? Yes, if we know the Decorator Pattern (see Appendix B)

Or, perhaps the client always delegates to both services, but sometimes it does it in one sequence, sometimes in another:

```
if (someConditional) {
      Service1.doOperation();
      Service2.doOperation();
} else {
      Service2.doOperation();
      Service1.doOperation();
}
```

Here again, this is a variation, one of order, or sequence, and it is not encapsulated. Could it be? Yes, if we know the Chain of Responsibility Pattern (and again, Appendix B will explain If you do not know it. Un-encapsulated, imagine what will happen if we get a third service, and then a fourth? How would this code look if it supported all the possible sequences then? What if both the number *and* the sequence needed to vary.

The point is, many, many things can vary, and we should encapsulate them as much as we can, because every time we do we get to win the Kobayashi Maru, the un-winnable scenario.

Software that decays sets us up to fail. Let's all take a page from Captain Kirk, and set ourselves up to win.

He always got the green-skinned alien dancing girls, after all. ☺

Summary

Principles and wisdom help us in general ways. Adhering to a principle is a good thing, but there is really no end to it. A system could always be more open or closed to more things. There is always another level of abstraction, and therefore more encapsulation to add to systems.

How far do you go?

The concept of the practice, which is where we'll go next, is very different. Practices are eminently practical, simple, and doable. You follow them, or you don't. They are also easy to share with others, to promote them generally across a team. Like principles and wisdom, they are based on experience (often, the experience of others) and so are very well-grounded.

Practices help you to follow principles; principles give you a reason to follow practices. Now that we've discussed the reasons, we'll look at the actions that promote them.