NET OBJECTIVES EZINE
VOLUME 1, ISSUE 5 MAY 2004

The Perspectives of Use vs. Creation in Object-Oriented Design

Scott L. Bain, Senior Consultant, Net Objectives, slbain@netobjectives.com

Articles and events of interest to the software development community

In This Issue:

Address:
275 118th Avenue SE
Suite 115
Bellevue, WA 98005
Telephone:
(425) 688-1011
Email:
info@netobjectives.com

Contents:

Background

This article came out of the work I'm doing at Net Objectives, teaching people how to write effective object-oriented code. In it I introduce a useful and perhaps different way of looking at solving certain design problems that face us every day. Rather than paying attention to what objects do, functionally, I have focused on a key distinction: to differentiate objects that use other objects from those that instantiate/manage them. I have found that this greatly simplifies and improves the code, especially in terms of future maintenance.

Fowler's Perspectives

In his wonderful book UML Distilled, 3rd Ed., 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."

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

Volume 1, Issue 5
May 2004

NET OBJECTIVES

Address:
275 118th Avenue SE
Suite 115
Bellevue, WA 98005
Telephone:
(425) 688-1011
Email:
info@netobjectives.com

The **Implementation** perspective is the "code level" or the actual programmatic solutions that objects use to fulfill those very 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[2] 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[3] 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 encourages a **cleaner** cognitive process 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.

One of the benefits that comes with experience (in any domain) is greater clarity regarding what to pay attention to, and what you can put off until later. These *distinctions* are critical to the successful understanding and practice of any complex activity.

In software development, **Cohesion** is just such a distinction. Cohesion refers to the internal consistency of any entity (method, class, package, etc…) in a system. We say that something has "strong" cohesion if everything it contains belongs to the same function (if it's a method), responsibility (if it's a class), or domain (if it's a package). Conversely, we say it has "weak" cohesion if it's a mixed bag of unrelated issues. A system consisting of one class, which was responsible for everything, containing one big method, that did everything, would be an example of extremely weak cohesion.

Coupling is another key distinction. It refers to the degree and nature dependencies *between* entities in the system. The more dependencies there are, and the more intricate they are, the more difficult the system will be to maintain.

[2] This could be a class, a method; some languages have other idioms such as delegates, etc…
[3] Design Patterns by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley Pub Co, ISBN: 0201633612

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're seeking in an object-oriented solution: the perspective of Creation vs. the perspective of Use.

Consider the following bit of code:

```
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 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[4].

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. 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 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 practice. 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.

[4] This is an example of following the Open-closed Principle. For more information, see http://www.objectmentor.com/resourcese/articles/ocp.pdf

Volume 1, Issue 5
May 2004

NET OBJECTIVES

Address:
275 118th Avenue SE
Suite 115
Bellevue, WA 98005
Telephone:
(425) 688-1011
Email:
info@netobjectives.com

Volume 1, Issue 5
May 2004

NET OBJECTIVES

Address:
275 118th Avenue SE
Suite 115
Bellevue, WA 98005
Telephone:
(425) 688-1011
Email:
info@netobjectives.com

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()` and so forth. So, to make any real use of the object-being-used, the using object must know one of three things:

- The actual type of the object-being-used, or
- An Interface the object-being-used implements, or
- The Superclass the object-being-used is derived from.

To keep things as decoupled as possible, we'd prefer one of the second two options, so 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.

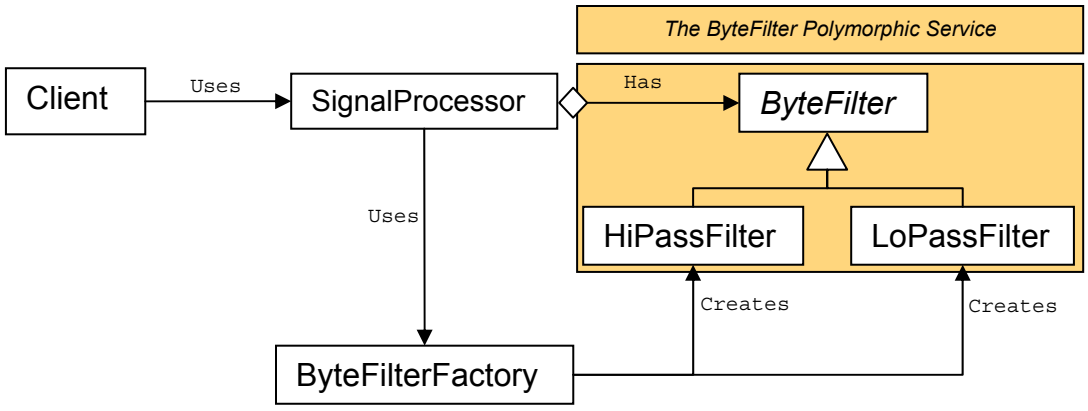
The using object (client), in other words, should ideally be coupled only to an abstraction, not to the actual concrete classes that exist, so that we are free to add more such classes in the future without having to maintain the client object. This is especially important if there are many different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

A Separate Perspective: Construction

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'm suggesting that this is another kind of *perspective* distinction. **Use** vs. **Construction**. In the same way that the users of an object should not be involved with its construction, similarly the constructor of an object should not be involved with its use. Therefore we typically call such a "constructing object" a *factory*.

This implies a design along these lines:



"For every complex problem there is a simple solution. And it is wrong."
-- H.L. Mencken

What's 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've been good

Volume 1, Issue 5
May 2004

NET OBJECTIVES

Address:
275 118th Avenue SE
Suite 115
Bellevue, WA 98005
Telephone:
(425) 688-1011
Email:
info@netobjectives.com

Volume 1, Issue 5
May 2004

NET OBJECTIVES

Address:
275 118th Avenue SE
Suite 115
Bellevue, WA 98005
Telephone:
(425) 688-1011
Email:
info@netobjectives.com

OO programmers and not added any methods to the interfaces of these subclasses.[5]

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's 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've limited the factory to the construction perspective only. The factory never calls methods on the objects it builds.[6]

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's 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 the most crucial issue in OO design. Even if we get the interfaces wrong, it's not as bad as missing an entire abstract concept.

So, the implication of this clean separation would be:

*The relationship between any entity A and any other entity B in a system should be limited such that A **makes** B or A **uses** B, and **never both**.*

Separating Perspectives in the Real World

Does this mean that for every class in your design there should be another class, termed "the factory," which other classes must use to instantiate it? Even when there is no variation, just a simple, single class of an ordinary sort? That does seem like overkill.

The problem is, we never know when something is going to vary in the future. Our abilities to predict change along these lines have traditionally been dramatically poor. Luckily, there is a middle ground, which is to encapsulate the constructor in single classes.

To do this, one simply makes the constructor of the object private (or protected), and then

About the author – Scott Bain

Scott Bain is a 27-year veteran in computer technology, with a background in development, engineering, and design. He has also designed, delivered, and managed training programs for certification and end-user skills, both in traditional classrooms and via distance learning. Scott teaches courses and consults on Design Patterns, XML, Refactoring and Unit Testing, and Test-Driven Development. Scott is a frequent speaker at developer conferences such as JavaOne and SDWest. He is the co-author (with Alan Shalloway) of "An Introduction to XML and its Family of Technologies" (ISBN: 0971363005; August 10, 2001) and is currently co-authoring Emergent Design: Refactoring and Design Patterns for Agile Development, also with Alan Shalloway.

[5] See http://www.objectmentor.com/resources/articles/lsp.pdf for more information on the Liskov Substitution Principle which, among other things, implies this very important lesson.
[6] For our purposes here, we do not consider the constructor to be part of the interface of an object.

adds a static method to the class that uses the constructor to return an instance. Here's a code snippet to illustrate the procedure:

```
public class Sender {

	private Sender() {
		// do any constructor behavior here
	}

	public static Sender getInstance() {
		return new Sender();
	}

	// the rest of the class follows

}

public class Client {
	private Sender mySender;

	public Client() {
		mySender = Sender.getInstance();
	}

}
```

The key difference between this "Encapsulated Construction"[7] and a more familiar approach is the fact that Client *must* build its `Sender` instance this way: `mySender = Sender.getInstance()`, rather than the more traditional `mySender = new Sender();` due to the private constructor on `Sender`.

At first this might seem rather pointless. After all, we've really accomplished nothing special that we could not have done with the more traditional form. However, what we have done is taken control of "new". The "new" keyword in modern languages like Java and C# cannot be overloaded, and thus we cannot control what it returns; it always returns, literally, the class named directly after the keyword. A method like `getInstance()` however, can return *anything that qualifies* as the type indicated. The value of this is clear when `Sender` changes, later, from a simple class to a polymorphic service:

```
public abstract class Sender {
	public static Sender getInstance() {
		if (someDecisionLogic()) {
			return new SenderImpl1();
		} else {
			return new SenderImpl2();
		}
	}
}

public class SenderImpl1 extends Sender {
	// one version of the Sender service
}

public class SenderImpl2 extends Sender {
	// another version of the Sender service
}

public class Client {
	private Sender mySender;

	public Client() {
		mySender = Sender.getInstance();
	}
}
```

[7] I cannot say who first suggested this, but I learned it from Joshua Bloch, who wrote about it in his book Effective Java Programming Language Guide, Addison-Wesley Pub Co, ISBN: 0201310058. I recommend this book highly, for this and many other reasons.

Volume 1, Issue 5
May 2004

Address:
275 118th Avenue SE
Suite 115
Bellevue, WA 98005
Telephone:
(425) 688-1011
Email:
info@netobjectives.com

Volume 1, Issue 5
May 2004

NET OBJECTIVES

Address:
275 118th Avenue SE
Suite 115
Bellevue, WA 98005
Telephone:
(425) 688-1011
Email:
info@netobjectives.com

The main point here is that Client does not need to change when this variation comes about. It's common for there to be many clients for such a service, and so the limited change here could decidedly be non-trivial in terms of maintenance.

Aren't we violating the overall concept of limiting perspectives? After all, `Sender` is now both a *conceptual* object (an abstract class) and also an implemented class (in that it's now implementing factory behavior). Yes, we are, in a limited way, because we must sometimes bow to pragmatism in order to allow for unanticipated change, like this one, without putting in anticipatory code for every *possible* change.

In this example, `getInstance()` in `Sender` is now making a simple decision about which subclass to build, and so long as that decision stays simple, we can probably live with it. If it becomes complex at all, however, we will want a separate factory to build the `Sender` subclasses. Does that meant the client objects will have to change, to switch from calling a static method on `Sender` to calling some other method on a separate factory? Not at all. We'll just delegate:

```
Public abstract class Sender {
	private static SenderFactory myFactory = SenderFactory.getInstance();

	public static Sender getInstance() {
		return myFactory.getSender();
	}
}

public class Client {
	private Sender mySender;

	public Client() {
		mySender = Sender.getInstance();
	}
}
```

This is optional, of course. If the number of clients is small, or if we have a lot of time to refactor, we could change the clients to call the factory directly. It's just not necessary, and most importantly we've not painted ourselves into a corner here. You'll also note that the constructor of `SenderFactory` (not shown) is obviously encapsulated as well, since `Sender` is using a static `getInstance()` method to build one. We never know when *factories* might become polymorphic services either, so we encapsulate their constructors as well.

Additionally, it's not at all uncommon for Factories to be Singletons (a design pattern that ensures that only one instance of a class will ever exist, and provides global access to it). The refactoring steps from encapsulated construction to the Singleton pattern are trivial.

Conclusions

The notion that entities in a design have perspectives at which they operate implies a kind of cohesion. Cohesion 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:

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

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.

We also find that good practices support and are supported by other good practices.

Test-Driven Development, for example, tends to produce classes that are more testable (since testing them is an early concern), which also means they tend toward strong cohesion, finer granularity, etc… Separating use from construction makes *using entities* and *constructing entities* separately testable, and allows for easier insertion of mock objects when they are needed to eliminate dependencies that can make testing difficult.

The Design Patterns perspective on design, also, is very compatible with and supported by the separation of use from construction. The lion's share of the patterns are concerned with hiding variations behind abstractions. This enables the Open-Closed Principle, and all the benefits that fall from it.

However, when using objects do not "know" what actual objects they are using (they "know" only about the abstractions), then something, somewhere must take care of this issue. This turns out to be the creation objects, and these are quite naturally separate from the objects user the instances in question.

We are building for ourselves a true "profession" by seeking out the qualities, principles, practices, and patterns we have identified as valuable. They form the basis for our common wisdom.

Finding these concepts and integrating them with what we already know will make us all more successful, and will enable us to add value to our customers, culture, and economy. They will also help us to communicate and collaborate more effectively, and will help to create an atmosphere of creativity and innovation.

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

Volume 1, Issue 5
May 2004

NET OBJECTIVES

Address:
275 118th Avenue SE
Suite 115
Bellevue, WA 98005
Telephone:
(425) 688-1011
Email:
info@netobjectives.com

Volume 1, Issue 5
May 2004

Address:
275 118th Avenue SE
Suite 115
Bellevue, WA 98005
Telephone:
(425) 688-1011
Email:
info@netobjectives.com

Consequently, it makes what we do more fun. I tend to believe that indicates we're on the right track.

-Scott Bain-

To join a discussion about this article, please go to http://www.netobjectivesgroups.com/6/ubb.x and look under the E-zines category.

Net Objectives - What We Do

Net Objectives provides enterprises with a full selection of training, coaching and consulting services. Our Vision is "Effective software development without suffering". We facilitate software development organizations migration to more effective and efficient processes. In particular, we are specialists in agility, effective analysis, design patterns, refactoring and test-driven development.

We provide a blend of training, follow up coaching and staff supplementation that enables your team to become more effective in all areas of software development. Our engagements often begin with an assessment of where you are and detail a plan of how to become much more effective. Our trainers and consultants are experts in their fields (many of them published authors).

When you've taken a course from Net Objectives, you will see the world of software development with new clarity and new insight. Our graduates often tell us they are amazed at how we can simplify confusing and complicated subjects, and make them seem so understandable, and applicable for everyday use. Many of our students remark that it is the best training they have ever received.

The following courses are among our most often requested. This is not a complete list, though it is representative of the types of courses we offer

Agile Project Management - This 2-day course analyzes what it means to be an agile project, and provides a number of best practices that provide and/or enhance agility. Different agile practices from different processes (including RUP, XP and Scrum) are discussed.

Agile Use Case Analysis - This 3-day course provides theory and practice in deriving software requirements from use cases. This course is our recommendation for almost all organizations, and is intended for audiences that mix both business and software analysts.

Design Patterns Explained: A New Perspective on Object-Oriented Design - This 3-day course teaches several useful design patterns as a way to explain the principles and strategies that make design patterns effective. It also takes the lessons from design patterns and expands them into both a new way of doing analysis and a general way of building application architectures.

Test-Driven Development: Iterative Development, Refactoring and Unit Testing - This course teaches how to use either Junit, NUnit or CppUnit to create unit tests. Lab work is done in both Refactoring and Unit Testing together to develop very solid code by writing tests first. The course explains how the combination of Unit Testing Refactoring can result in emerging designs of high quality.

Effective Object-Oriented Analysis, Design and Programming In C++, C#, Java, or VB.net - This 5-day course goes beyond the basics of object-oriented design and presents 14 practices which will enable your developers to do object-oriented programming effectively. These practices are the culmination of the best coding practices of eXtreme Programming and of Design Patterns.

Software Dev Using an Agile (RUP, XP, SCRUM) Approach and Design Patterns - This 5-day course teaches several design patterns and the principles underneath them, the course goes further by showing how patterns can work together with agile development strategies to create robust, flexible, maintainable designs

If you are interested in any of these offerings, if your user group or company is interested in Net Objectives making a free technical presentation to them, or if you would like to be notified of upcoming Net Objectives events, please visit our website, or contact us by the email address or phone number below:

www.netobjectives.com • mike.shalloway@netobjectives.com • 404-593-8375