Bertrand Meyer 谈契约式设计

By Bill Venners   December 8, 2003

Summary 提要

Bertrand Meyer talks with Bill Venners about Design by Contract and the limits of formal languages for expressing contracts.

Bertrand Meyer和Bill Venners讨论契约式设计和形式语言在契约表述方面的局限性。

Bertrand Meyer is a software pioneer whose activities have spanned both the academic and business worlds. He is currently the Chair of Software Engineering at ETH, the Swiss Institute of Technology. He is the author of numerous papers and many books, including the classic Object-Oriented Software Construction (Prentice Hall, 1994, 2000). In 1985, he founded Interactive Software Engineering, Inc., now called Eiffel Software, Inc., a company which offers Eiffel-based software tools, training, and consulting.

Bertrand Meyer是同时活跃于学术和商业二界的软件先驱。他目前担任瑞士理工学院的软件工程协会主席。他撰写了数量浩繁的论文和书籍，如经典的《面向对象软件构造》（Prentice出版社，1994，2000）。1985年，他创立了交互软件工程公司。公司目前已经更名为Eiffel软件公司，提供基于Eiffel语言的软件工具、培训和咨询业务。

On September 28, 2003, Bill Venners conducted a phone interview with Bertrand Meyer. In this interview, which will be published in multiple installments on Artima.com, Meyer gives insights into many software-related topics, including quality, complexity, design by contract, and test-driven development.

2003年9月28日，Bill Venners对Bertrand Meyer做了电话访谈。在这次访谈（内容将在Artima.com分多次公布）中，Meyer对许多有关软件的问题作了鞭辟入里的论述，如软件质量、软件复杂性、契约式设计以及测试驱动型开发等。

Designing Components that Fit Together

设计能协同工作的组件

Bill Venners: In your book, Object-Oriented Software Construction, you write, "For a system of any significant size, the individual quality of the various elements involved is not enough. What will count most is the guarantee that for every interaction between two elements there is an explicit roster of mutual obligation and benefits, the contract." What is the contract? Why is it important?

你的《面向对象软件构造》一书中讲到：“对于一个大型系统来说，光保证它的各组成

部分的质量是不够的。而最有价值的是确保在任何两个组成部分的交接处设计明晰的彼此义务和权利规范，即所谓契约。”什么是契约？它的重要性在哪里？

Bertrand Meyer: There are two quite separate points in that extract. The first point is that when you're building a library, it's not enough to just accumulate good components. Take a data structure library as an example. You might have excellent classes for lists, stacks, files, and btrees, but taken together they don't make an excellent library if they are inconsistent. If they use different conventions, they aren't part of a single design. For example, when you're putting an element into an array, you might have an insert operation that takes x and i, where x is the element and i is the index. For the hash table class you might have an insert operation that takes key and x, where key is the key and x is the element. The order of arguments is reversed. The order of arguments might make perfect sense within each class, but when you start approaching the library as a whole, you're in new territory each time you look at a new class. You don't get a feeling of consistency. Instead you get a feeling of a mess—something that is a collection of pieces rather than a real engineering design. What we found many years ago when we started focusing seriously on libraries is that just as much attention has to be devoted to the construction of the library as a whole as to the construction of the individual elements. That's one point.

书里从两个彼此独立的方面讨论了这个问题。第一，当你构建一个库时，仅仅设计好每个单独的组件还是不够的。拿一个数据结构库来说。你设计了很好的列表、栈、文件和B树类，但如果它们彼此不够协调，那么当把它们放在一起时，你就不能得到一个好的库。你可能在具有共性的地方，让每个类使用了不同的规定。比如，将一个元素放入数组时，你需要带有参数x和i的插入操作，其中x是元素本身，i是插入位置；而对于哈希表类，一个插入操作则需要参数key和x，其中key是关键字，x是元素。它们的参数顺序是相反的。不同的参数顺序对于不同的类来说，的确可以非常贴切地表达操作的含义，但当你将这个库作为一个整体使用时，每次面对一个不同的类，你都像进入了一个陌生的领域。你不仅得不到一致的、连贯的感觉，反而感到混乱——那是许多碎片的堆砌而不是工程化的设计。许多年以前，我们对库作认真分析时就发现，更多的注意力应该定位到整个库的构架上，而不是某个组成部分的设计。这是我要说的第一点。

The second point pertains to how a library needs to be consistent, how its various elements can fit together like the pieces of a puzzle. It's not just a matter of defining consistent interfaces, order of arguments, things like that. It's also a matter of defining precisely how the various elements are going to communicate with each other and making sure that the conditions of this communication is very precisely defined. This is, of course, where the techniques of Design by Contract come in.

第二点是说，如何设计一个库才能让它协调一致，它的众多像一个难题所包含的各方面因素的可变元素怎样才能协同工作。这不是说定义兼容的接口、参数顺序等等就可以了，还必须准确定义各组成部分的相互通讯方式，以及通讯需要的条件。当然，这就是契约式设计技术所包含的内容了。

Design by Contract

契约式设计

Bill Venners: Could you give an overview of Design by Contract?

你可以概括描述一下契约式设计吗？

Bertrand Meyer: People who have heard a little bit about contracts often think they are simply a way to put the equivalent of assertions in a program: that is to say, to put in a few checks here and there for conditions that are expected to hold at specific points of execution. Such assertions are basically debugging statements. The assertion code is executed during debugging, and if one of the conditions is found not to hold, execution usually stops with a message. That's one of the things you can do with contracts, but it is only a small subset of the purpose of contracts.

对契约有所耳闻的人通常认为契约不过就是讲如何将类似断言的东西放到程序中。也就是说，在一些地方进行条件检查，从而控制程序的执行。这些断言类似物基本上可以看作调试语句。断言代码在调试阶段执行，如果发现某个条件没有得到满足，通常会给出相应信息并停止执行。这的确是你使用契约可以办到的事情，但这只是契约式设计目的中的一个很小子集。

The main purpose of contracts is to help us build better software by organizing the communication between software elements through specifying, as precisely as possible, the mutual obligations and benefits that are involved in those communications. The specifications are called contracts. The underlying observation, which is not particularly deep, is that a software system is made of a number of elements that cooperate with each other. In an object-oriented architecture, these elements might be classes and methods, which I prefer to call routines. The elements might be something else in a different programming model, but let's just assume we are working in an object-oriented context. So we have a software system made of a number of classes, and these classes themselves contain among other things routines that are going to be executed and call each other. So both the architecture of the system and its execution rely on a large set of possible communication channels between the software elements. The metaphor of contracts is used to guarantee that these communications occur not on the basis of vague expectations of services rendered, but on the basis of precise specifications of what these services are going to be.

契约的主要目的是：尽可能准确地规定软件元素彼此通讯时的彼此义务和权利，从而有效组织通讯，进而帮助我们构造出更好的软件。其中的“规定”就是我们所说的“契约”。通过深入观察——不是指局部的深入——我们认为软件系统是由大量彼此协同的元素构成的。在面向对象构架中，这些元素可以是类和方法（我更喜欢称为例程）。在其他的程序设计模式中，这些元素可以是其他的东西，不过在这里，我们假设是工作在面向对象环境。这样，我们的软件系统就是由大量类组成的；这些类又包含很多将独立执行又彼此调用的例程。所以，系统架构和系统运行都要依附于大量的存在于软件元素之间的通讯渠道。契约的言下之意就是保证这些通讯在待提供服务的准确而不是含糊不清的要求基础上进行。

Typically in such a communication between program elements—for example, when one routine calls another routine—it's like a client/supplier relationship. The calling routine, which we will call the client, is performing some computation, probably for the need of its own client. In order to perform that computation, that is to say, to perform its own service, it needs services from some other software element, typically some other routine. So there is a client/supplier relationship, where the client needs a certain service and the supplier provides that service. And in the basic scheme both the client and the supplier are routines.

可以举一个程序元素间通讯的典型例子。当一个例程呼叫另一个例程时，它们构成客户/服务关系。呼叫例程（我们也叫它客户例程），可能正在为了满足它自己的客户的需求进行计算。为了完成计算（或者说为了完成自己要提供的服务），它需要从别的软件元素获得服务。那么这里就存在一个客户/服务关系——客户端需要某个服务，服务端提供这个服务。在这个很基本的模型里，客户端和服务端都是例程。

For the software and software designer to be able to guarantee any kind of correctness and robustness properties, they must know the precise constraints over such communications. This is where the contract metaphor from business applies to software. Say I'm contracting from you to have a certain business operation performed on my behalf. For example, perhaps you have a part that I can use for a product that I'm manufacturing. So I'm your client, and you're my supplier. In business, we're going to organize our collaboration on the basis of a contract: a precise statement of the mutual obligations and benefits that we'll expect. In software, we're going to do exactly the same thing when we write client and supplier routines.

对于软件和软件设计者来说，为了保证各方面的正确性和健壮性，他们就必须懂得通讯的准确约束规范。在这个地方，我们就将商业中的契约概念应用到软件中。为了我的利益，我必须对你实施某项业务时作出约束。比如，你可能有某个在我正开发的产品中可以使用的部分。那么我就是你的客户，你是我的服务提供者。商业中，我们会通过契约——对我们期望的彼此义务和权利的准确表述——的方式彼此协调。对于软件来说，我们撰写客户和服务例程时，都必须严格遵循同样的契约表述。

So we're going, for example, to impose on the client certain obligations as to the kind of original program state that is permissible when the client calls the supplier or the kind of arguments that the client routine passes to the supplier. These are pre-conditions, and they're obligations for the client. In the other direction, we are going to express the conditions that the supplier routine must guarantee to the client, on completion of the supplier's task. That's the post-condition of the contract, specifically, the post-condition of that particular routine. And of course, the post-condition is an obligation for the supplier.

比如，什么原始程序状态是客户端呼叫服务端的必要条件，客户例程传递给服务例程的参数应该是什么类型，都是我们要在客户端作出的约束。这些就是前置条件，是客户端的义务。在另一方面，为了完成服务端的工作，我们也要阐明服务端必须对客户端作出保证的条件。这是契约中的后置条件，很多情况下，也就是某个例程的后置条件。当然，后置条件是服务端的义务。

Pre- and post-conditions are symmetric. The pre-condition of the routine is an obligation for the client, because the pre-condition says that before calling a particular routine, the client must satisfy a particular property. For the supplier, the pre-condition is a benefit, because it facilitates the supplier's job by restricting the set of cases that the supplier has to handle. The obligation of the client is a benefit for the supplier. The post-condition is of course an obligation for the supplier, because it describes the job that the supplier has to guarantee to perform. But the post-condition is clearly a benefit for the client, because it describes the result that the client is entitled to expect from the execution of the routine for the benefit of its own problem set.

前置和后置条件应该是对称的。例程的前置条件是客户端的义务，因为前置条件规定客户端在调用一个特定的例程前，必须满足特定的要求。对于服务端来说，前置条件就成了权利，因为它通过对服务端必须处理的一系列工作作出约束，从而使服务端的工作更易完成。后置条件当然就是服务端的义务，因为它描述服务端必须确保处理的工作。但后置条件很显然又是客户端的权利，因为它描述了客户端有资格期望的该例程执行后的返回结果，这对客户端解决自己的问题集是有好处的。

Before moving on I can describe a very simple example of pre- and post-conditions: a routine that computes the square root of a real number. The pre-condition would say that the real number has to be non-negative, because otherwise the result would be undefined. The post-condition may say that the result returned by the routine is the approximation, within a specified precision, of the exact mathematical square root of the routine.

在结束此话题之前，我可以举一个有关前置和后置条件的简单例子：一个例程要计算一个实数的平方根。那么前置条件要求实数必须是非负数，因为负数的平方根没有定义。后置条件则可以这样表述：例程返回在数学上指定的精度内最接近的平方根。

So pre- and post conditions are two fundamental elements of contracts. The third fundamental element—which is really useful mostly in an object-oriented context, where we have not only routines but at the higher level, classes—is invariants. A class invariant is a condition that applies to an entire class. It describes a consistency property that every instance of the class must satisfy whenever it's observable from the outside. That means that this property, the class invariant, must be satisfied whenever an instance of the class is created. And every exported routine, routines that can be called from the outside of the class, must preserve it. That is to say, assuming the class invariant was satisfied before the routine was called, the routine must ensure that class invariant is again going to be satisfied on exit. It's as if the class invariant is added to the pre- and post-condition of every single exported routine of the class. But more fundamentally, a class invariant is a way to characterize the fundamental consistency and integrity properties of the class and its instances.

因此，前置和后置条件是契约理论的两个基本要素。第三个要素（它实际上在面向对象领域非常有用，因此我们不仅仅在例程，而通常是更高层次——类中使用）是不变式。类不变式是应用到整个类的。类不变式描述一种属性，类的每个实例在该属性对外部可见的任何时候都必须确保它是一致的。这就意味着无论什么时候创建一个类实例，该属性（或者说类不变式）的一致性都必须得到确保。任何一个暴露的例程（从类外部可以调用的例程）都必须保护它；也就是说，假设类不变式在该例程调用前被满足了，那么例程退出时必须再次确保类不变式得到满足。如此看来，类不变式好像是被附加到了类暴露例程的前置和后置条件上。但从根本上讲，类不变式是一种刻画类和类实例的基本的一致性和完整性的方法。

For example, if you have a bank account class with a field that represents the current balance and a field that contains a list of all deposits and withdrawals since the opening of the account, then you would have a class invariant that states that the value of the balance field is equal to the total of all the deposit values so far minus the total of all the withdrawal values so far. It's typical that class invariants define properties that indicate how the various constituents of the class, such as balance and deposit/withdrawal lists, maintain consistency with each other.

例如，你有一个银行帐户类，它包含一个表示当前余额的字段和一个包含帐户自开通以来所有储蓄和支取清单的字段，那么你也应该有一个类不变式表明余额字段的值与累计储蓄和累计支取之差相一致。这个典型例子表明：类不变式定义属性，该属性指明类的可变要素（如余额和存储/支取清单）如何维护彼此一致性。

Pre- and post-conditions and class invariants are not the only elements of contracts. There are also loop invariants and a few others, but pre- and post-conditions and class invariants are really the basic fabric of contracts. In my experience, relying on these notions—that is to say, making sure when you write software that don't just write the implementation, but also write the more abstract properties underlying the implementation in the form of contracts—provides a greatly added software development experience in several respects. It helps ensure correctness in the first place, helps debugging, helps testing, helps ensure inheritance is properly handled, helps managers, provides a quite effective form of documentation, and a few others.

契约的要素不光有前置、后置条件和类不变式。还有循环不变式等等，但前置、后置条件和类不变式是基本要素。依据我的经历来说，依靠这些概念——就是说，你开发软件时，不要仅仅注意实现的具体化，也应该按照契约规定的形式提供一些更加抽象的性能——可以从几个方面大大增加你的软件开发经验。它帮助你保证软件正确性，让软件便于调试、测试，利于确保继承的正确处理，有助于管理人员，并提供了十分有效的文档建立方式，等等。

The Limits of Formal Languages

形式语言的局限性

Bill Venners: Are there parts of contracts that are difficult or impossible to express in a formal language that you can only express in a human language?

在形式语言里，契约的某些部分是否难以或者无法表达，而只能通过人类语言来表述？

Bertrand Meyer: The simple answer is yes. There are really three reasons why it sometimes feels hard to express a contract in terms of a programming language. As a background, the way contracts are expressed in Eiffel is very simple: they are just Boolean expressions. Boolean expressions are intended precisely to express runtime true or false properties—properties that at any point in the execution may hold or not hold. That's exactly how Boolean expressions are used, for example, in an if-then-else construction. That's also how they are used to express contracts in Eiffel, with one major extension: the old keyword. The old keyword makes it possible to express in a post-condition that a certain property at the completion of a routine is relative to a certain property on entry to the routine. If you have, for example, a routine that adds a certain amount of money to a bank account, obviously the post-condition of this routine is going to have to express that the new balance is related to the old balance. The new balance must be the balance before the execution of the routine plus some amount. The only way to express this properly is to have some notation to refer to the original value of the balance. So with this major, but single, extension, the language of contracts in Eiffel is just the language of Boolean expressions.

简单来说是这样的。实际中有三个原因造成了人们有时候感觉以编程语言的方式难以表述契约。首先交待一个背景，在Eiffel语言中，表述契约的方法是很简单的——就是使用布尔表达式。布尔表达式的设计意图就是表达运行时值为真或假的属性，在某个时刻判断此属性的值就可以决定执行是否继续。这也就是布尔表达式的使用方法，比如在一个if-then-else块中。在Eiffel中，再加上一个主要的扩展——引入“old”关键字——就可以通过这种方法来表述契约。old关键字使在后置条件中表述例程完成后的属性和例程开始前的属性的二者关系成为可能。例如，你用某个例程将一定数额的存款累加到一个银行帐户上，那么很明显，这个例程的后置条件必须表述当前余额和过去余额的关系。当前余额必须是例程执行前的余额与增加量之和。正确表述这种关系的唯一办法就是使用某个符号引用余额的原始值。因此，有了这个主要的但单纯的扩展后，Eiffel的契约设计语言实际就可看作布尔表达式语言。

There are three occasions in which this approach of using Boolean expressions may appear restrictive. One occasion that is justified—where you will have to resort to human language descriptions instead of formal ones—is when you have externally visible properties. For example, if you have in a graphical library an operation that changes the color of a certain pixel on the screen to red, you'd like to describe a post-condition that says the resulting color is red. Maybe to some extent you can do that, but what you really want to express is that if a person is sitting at a terminal, and the person is not color blind, he or she will accept that the new color is red. Clearly this kind of assertion cannot be expressed in a formal language, because it refers to properties outside of the realm of the software program. In practice it may not be such a big deal, because what you usually want in practice is not necessarily the guarantee that someone accepts the color as red, but the guarantee that the RGB value of the pixel being displayed is within a certain range. And of course that can be expressed as a purely Boolean property.

在三种情况下，使用布尔表达式去表述契约可能显得有些力不从心。第一种情况是你用属性表述物理可见性时，你不得不借助人类语言来作评判，形式语言无能为力。例如，你的图形库中有一个操作是将屏幕上某点的颜色设置为红色，这时你一般会用一个后置条件表述该点的最终颜色为红色。在一定程度上说，你这样做就够了，但实际上你真正想表述的是：有一个人正坐在电脑前面，此人不是色盲，他/她认可该点是红色。很显然，这样的断言无法用形式语言作出表述，因为它描述的是超出软件语言范围的属性。当然在实际中，这未免小题大做，因为通常情况下，我们不是要保证某人认可该点的颜色是红色，而是要保证该像素显示颜色的RGB值落在了指定范围。当然，这就可以用一个纯布尔属性来表达了。

The second occasion where Boolean expressions may seem restrictive is one that scares most people who have looked at the issue from a theoretical perspective: the language of Boolean expressions is relatively limited and doesn't have first-order predicate calculus. This has led some people who design the specification mechanisms for programming languages, or in the case of UML, for modeling languages, to include facilities from first-order predicate calculus as a language extension. I think that is largely a mistake. At least we don't need to do this in Eiffel, because we have a high-level mechanism known as agents to describe essentially high-level functional operations on objects. So anything that you may want to express on a complex object that would seem to require first-order predicate calculus can be expressed actually quite nicely within the confines of the programming language.

很多人从理论角度考察时，又惊惧地发现布尔表达式形似柔弱的第二种情况：使用布尔表达式的语言有局限性，因为不具备一阶谓词演算特性。这就促使一些人在编程语言的描述

机制和建模语言（如统一建模语言，UML）的处理机制中加进了部分一阶谓词演算的东西，作为一种功能扩展。我认为这是一个很大的误解。至少在Eiffel中我们不需要这样做，因为我们有一种叫做代理的高水平机制来描述在对象上的高水平的功能性操作。所以对于一个复杂对象，任何看起来需要一阶谓词演算才能描述的东西，实际都可以在编程语言范畴内实现十分精细地表述。

What is quite easy to express with Boolean expressions, for example, is a property that if you're adding an element at the end of a list, the last element of the list is now the one that you just added. That's quite easy to express, and it's definitely expressed in the corresponding Eiffel concept. What is less easy to express is the property that all the other elements of the list, the ones that you haven't touched, are still there, in the same order, and are equal to their previous values. This kind of assertion seems to require special language extensions and has led people to suggest introducing first-order predicate calculus. But what we've found is that apart from the agents mechanisms, which has been in Eiffel now for a number of years, there's really no need for introducing first-order predicate calculus, or there exists operators. We can basically express those assertions in the realm of the programming language.

举例来说，如果你将一个元素增加到列表末尾，该列表当前的最后一个元素就是你刚刚增加的元素。这个属性是非常容易描述的，Eiffel中的相应理论对此有明确表述。不容易描述的属性是列表中的所有别的你不曾动过的元素，它们仍在原地，顺序和值都保持不变。这种情况下的断言似乎就需要实现特定的语言扩展，于是一些人建议引入一阶谓词演算。但我们发现使用Eiffel中已经存在多年的代理机制就可以解决，完全没有必要引入一阶谓词演算和there exists操作符。在编程语言范畴内，我们基本上就可以表述这些断言。

Also, first-order predicate calculus isn't good enough anyway. A typical kind of property that you might want to express, say, in a class invariant for a graph class, is that the graph has no cycles. This is not expressible as a first-order predicate calculus property. On the other hand, it is expressible as a Boolean expression if you accept including function calls in the Boolean expression. So that particular argument for extending the assertion language or for expressing contracts in English goes away at least in the long term. In the short term, these techniques of expressing high-level ambitious properties are still under development, and you will still find for that kind of property assertions expressed informally as comments.

同时，从各方面来说，一阶谓词演算都不是足够好。比如在一个图类中，你可能想要表述的典型属性是该图没有回路。这就是一个一阶谓词演算不可表述的属性。但换个角度，如果你允许在布尔表达式中植入包含函数调用，则它就是一个布尔表达式可表述的属性。因此，有关扩展断言语言和英语中的契约表述的争论，在一个较长时期后将会沉寂下来。短时期内，表述高复杂性属性的技术仍会处于开发阶段，不过你能找到对这类属性非正式的、注释性的断言表述。

The third occasion where defining contracts using Boolean expressions raises issues is non-functional contracts. Maybe I shouldn't go into this too much, because this is more of a research issue. Nevertheless, once you have completely expressed all the functional contracts—the contracts that specify what the input may be to operations, what outputs are legitimate, and what globally consistency constraints must be maintained—you might still want to express as part of the specification things like: this operation will always execute in less than half a millisecond, or this particular component will never use more than 300 kilobytes. These are performance contracts, and it's a quite interesting research area. I think there will be usable results in this area in the next few years, but it falls beyond the realm of Design by Contract as it is generally understood in practice today.

使用布尔表达式定义契约引起争议的第三种情况是在非功能性契约上。可能我不能对此说得过多，因为这还是一个研究课题。但是，在你完成所有功能性契约——这些契约规定了操作的合法输入、输出，以及必须维护的全局一致性约束——表述后，你可能还希望表述类似如下一些内容：该操作应该在半毫秒内完成，这个组件消耗的内存应该不大于300KB。这些都属于性能契约，也是一个让人颇感兴趣的研究领域。我想，一些年后，这个领域将会出现具有实用性的成果，不过它已经超出了契约式设计的范畴，在实际中，我们目前对此一般都能理解。

Bill Venners: That was actually my next question: is performance part of the contract? It does seem that performance is important in the contract sense. People want quality of service guarantees.

这恰恰是我预备的下一个问题：契约在操作性能方面如何表述？看来，从契约角度来说，性能也是非常重要的。人们需要高质量的服务保证。

Bertrand Meyer: Right, absolutely. It's very important. I think everyone realizes this. The problem is that it is pretty difficult to ensure and test. And it also has to be defined properly. When you define, for example, a response time, is it a minimum response time? Is it a maximum response time? Is it an average response time? If so, what is the statistical distribution? I don't think these questions have no possible answers, but they make the issue quite delicate. And they have no direct counterpart with functional contracts. So performance contracts are definitely an area in which more work is needed and actually is being performed.

绝对正确。它是十分重要的。我想每个人都认识到了这一点。问题是，对此的保证和测试是非常困难的。同时它还需要被准确定义。比如，你定义了一个响应时间，那么它是最小响应时间，抑或最大响应时间，还是平均响应时间呢？然后，统计分布又是什么样子呢？对此我并不认为没有可能的答案，但这些疑问使问题变得非常棘手。因此，性能契约被划入了一个需要更多努力的专门领域，实际上，此领域的工作已在进行之中。

Trackback: http://tb.blog.csdn.net/TrackBack.aspx?PostId=92277