源代码就是设计

作者：Jack Reeves

至今，我仍能记起当我顿悟并最终产生下面文章时所在的地方。那是
1986年的夏天，
我在加利福尼亚中国湖海军武器中心担任临时顾问。在这期间，我有幸参加了一个关于
Ada
的研讨会。讨论当中，有一位听众提出了一个具有代表性的问题，
“软件开发者是工程师
吗？”我不记得当时的回答，但是我却记得当时并没有真正解答这个问题。于是，我就退
出讨论，开始思考我会怎样回答这样一个问题。现在，我无法肯定当时我为什么会记起几
乎10年前曾经在
Datamation杂志上阅读过的一篇论文，不过促使我记起的应该是后续讨论
中的某些东西。这篇论文阐述了工程师为什么必须是好的作家（我记得该论文谈论就是这
个问题——好久没有看了），但是我从该论文中得到的关键一点是：作者认为工程过程的最
终结果是文档。换句话说，工程师生产的是文档，不是实物。其他人根据这些文档去制造
实物。于是，我就在困惑中提出了一个问题，
“除了软件项目正常产生的所有文档以外，还
有可以被认为是真正的工程文档的东西吗？
”我给出的回答是，
“是的，有这样的文档存在，
并且只有一份
——源代码。
”
把源代码看作是一份工程文档
——设计——完全颠覆了我对自己所选择的职业的看
法。它改变了我看待一切事情的方式。此外，我对它思考的越多，我就越觉得它阐明了软
件项目常常遇到的众多问题。更确切地说，我觉得大多数人不理解这个不同的看法，或者
有意拒绝它这样一个事实，就足以说明很多问题。几年后，我终于有机会把我的观点公开
发表。C++ Journal中的一篇有关软件设计的论文促使我给编辑写了一封关于这个主题的信。
经过几封书信交换后，编辑
Livleen Singh同意把我关于这个主题的想法发表为一篇论文。
下面就是这篇文章。

什么是软件设计？

面向对象技术，特别是
C++，似乎给软件界带来了不小的震动。出现了大量的论文和
书籍去描述如何应用这项新技术。总的来说，那些关于面向对象技术是否只是一个骗局的
问题已经被那些关于如何付出最小的努力即可获得收益的问题所替代。面向对象技术出现
已经有一段时间了，但是这种爆炸式的流行却似乎有点不寻常。人们为何会突然关注它呢？
对于这个问题，人们给出了各种各样的解释。事实上，很可能就没有单一的原因。也许，
把多种因素的结合起来才能最终取得突破，并且这项工作正在进展之中。尽管如此，在软
件革命的这个最新阶段中，
C++本身看起来似乎成为了一个主要因素。同样，对于这个问
题，很可能也存在很多种理由，不过我想从一个稍微不同的视角给出一个答案：
C++之所
以变得流行，是因为它使软件设计变得更容易的同时，也使编程变得更容易。

虽然这个解释好像有点奇特，但是它却是深思熟虑的结果。在这篇论文中，我就是想要关注一下编程和程序设计之间的关系。近10年来，我一直觉得整个软件行业都没有觉察到做出一个软件设计和什么是真正的软件设计之间的一个微妙的不同点。只要看到了这一点，我认为我们就可以从C++增长的流行趋势中，学到关于如何才能成为更好的软件工程师的意义深远的知识。这个知识就是，编程不是构建软件，而是设计软件。

几年前，我参见了一个讨论会，其中讨论到软件开发是否是一门工程学科的问题。虽然我不记得了讨论结果，但是我却记得它是如何促使我认识到：软件业已经做出了一些错误的和硬件工程的比较，而忽视了一些绝对正确的对比。其实，我认为我们不是软件工程师，因为我们没有认识到什么才是真正的软件设计。现在，我对这一点更是确信无疑。

任何工程活动的最终目标都是某些类型的文档。当设计工作完成时，设计文档就被转交给制造团队。该团队是一个和设计团队完全不同的群体，并且其技能也和设计团队完全不同。如果设计文档正确地描绘了一个完整的设计，那么制造团队就可以着手构建产品。事实上，他们可以着手构建该产品的许多实物，完全无需设计者的任何进一步的介入。在按照我的理解方式审查了软件开发的生命周期后，我得出一个结论：实际上满足工程设计标准的惟一软件文档，就是源代码清单。

对于这个观点，人们进行了很多的争论，无论是赞成的还是反对的都足以写成无数的论文。本文假定最终的源代码就是真正的软件设计，然后仔细研究了该假定带来的一些结果。我可能无法证明这个观点是正确的，但是我希望证明：它确实解释了软件行业中一些已经观察到的事实，包括C++的流行。

在把代码看作是软件设计所带来的结果中，有一个结果完全盖过了所有其他的结果。它非常重要并且非常明显，也正因为如此，对于大多数软件机构来说，它完全是一个盲点。这个结果就是：软件的构建是廉价的。它根本就不具有昂贵的资格；它非常的廉价，几乎就是免费的。如果源代码是软件设计，那么实际的软件构建就是由编译器和连接器完成的。我们常常把编译和连接一个完整的软件系统的过程称为"进行一次构建"。在软件构建设备上所进行的主要投资是很少的——实际需要的只有一台计算机、一个编辑器、一个编译器以及一个连接器。一旦具有了一个构建环境，那么实际的软件构建只需花费少许的时间。编译50 000行的C++程序也许会花费很长的时间，但是构建一个具有和50 000行C++程序同样设计复杂性的硬件系统要花费多长的时间呢？

把源代码看作是软件设计的另外一个结果是，软件设计相对易于创作，至少在机械意义上如此。通常，编写（也就是设计）一个具有代表性的软件模块（50至100行代码）只需花费几天的时间（对它进行完全的调试是另外一个议题，稍后会对它进行更多的讨论）。我很想问一下，是否还有任何其他的学科可以在如此短的时间内，产生出和软件具有同样复杂性的设计来，不过，首先我们必须要弄清出如何来度量和比较复杂性。然而，有一点是明显的，那就是软件设计可以极为迅速地变得非常庞大。

假设软件设计相对易于创作，并且在本质上构建起来也没有什么代价，一个不令人吃惊的发现是，软件设计往往是难以置信的庞大和复杂。这看起来似乎很明显，但是问题的重要性却常常被忽视。学校中的项目通常具有数千行的代码。具有10 000行代码（设计）的软件产品被它们的设计者丢弃的情况也是有的。我们早就不再关注于简单的软件。典型的商业软件的设计都是由数十万行代码组成的。许多软件设计达到了上百万行代码。另外，软件设计几乎总是在不断地演化。虽然当前的设计可能只有几千行代码，但是在产品的生命期中，实际上可能要编写许多倍的代码。

尽管确实存在一些硬件设计，它们看起来似乎和软件设计一样复杂，但是请注意两个有关现代硬件的事实。第一，复杂的硬件工程成果未必总是没有错误的，在这一点上，它不存在像软件那样让我们相信的评判标准。多数的微处理器在发售时都具有一些逻辑错误：

桥梁坍塌，大坝破裂，飞机失事以及数以千计的汽车和其他消费品被召回——所有的这些我们都记忆犹新，所有的这些都是设计错误的结果。第二，复杂的硬件设计具有与之对应的复杂、昂贵的构建阶段。结果，制造这种系统所需的能力限制了真正能够生产复杂硬件

设计公司的数目。对于软件来说，没有这种限制。目前，已经有数以百计的软件机构和数以千计的非常复杂的软件系统存在，并且数量以及复杂性每天都在增长。这意味着软件行业不可能通过仿效硬件开发者找到针对自身问题的解决办法。倘若一定要说出有什么相同之处的话，那就是，当
CAD和CAM可以做到帮助硬件设计者创建越来越复杂的设计时，硬
件工程才会变得和软件开发越来越像。
设计软件是一种管理复杂性的活动。复杂性存在于软件设计本身之中，存在于公司的软件机构之中，也存在于整个软件行业之中。软件设计和系统设计非常相似。它可以跨越多种技术并且常常涉及多个学科分支。软件的规格说明往往不固定、经常快速变化，这种变化常常在正进行软件设计时发生。同样，软件开发团队也往往不固定，常常在设计过程的中间发生变化。在许多方面，软件都要比硬件更像复杂的社会或者有机系统。所有这些都使得软件设计成为了一个困难的并且易出错的过程。虽然所有这些都不是创造性的想法，但是在软件工程革命开始将近
30年后的今天，和其他工程行业相比，软件开发看起来仍然
像是一种未受过训练（
undisciplined）的技艺。
一般的看法认为，当真正的工程师完成了一个设计，不管该设计有多么复杂，他们都非常确信该设计是可以工作的。他们也非常确信该设计可以使用公认的技术建造出来。为了做到这一点，硬件工程师花费了大量的时间去验证和改进他们的设计。例如，请考虑一个桥梁设计。在这样一个设计实际建造之前，工程师会进行结构分析——他们建立计算机模型并进行仿真，他们建立比例模型并在风洞中或者用其他一些方法进行测试。简而言之，在建造前，设计者会使用他们能够想到的一切方法来证实设计是正确的。对于一架新型客机的设计来说，情况甚至更加严重；必须要构建出和原物同尺寸的原型，并且必须要进行飞行测试来验证设计中的种种预计。
对于大多数人来说，软件中明显不存在和硬件设计同样严格的工程。然而，如果我们把源代码看做是设计，那么就会发现软件工程师实际上对他们的设计做了大量的验证和改进。软件工程师不把这称为工程，而称它为测试和调试。大多数人不把测试和调试看作是真正的"工程"——在软件行业中肯定没有被看作是。造成这种看法的原因，更多的是因为软件行业拒绝把代码看作设计，而不是任何实际的工程差别。事实上，试验模型、原型以及电路试验板已经成为其他工程学科公认的组成部分。软件设计者之所以不具有或者没有使用更多的正规方法来验证他们的设计，是因为软件构建周期的简单经济规律。
第一个启示：仅仅构建设计并测试它比做任何其他事情要廉价一些，也简单一些。我们不关心做了多少次构建——这些构建在时间方面的代价几乎为零，并且如果我们丢弃了构建，那么它所使用的资源完全可以重新利用。请注意，测试并非仅仅是让当前的设计正确，它也是改进设计的过程的一部分。复杂系统的硬件工程师常常建立模型（或者，至少他们把设计用计算机图形直观地表现出来）。这就使得他们获得了对于设计的一种"感觉"，而仅仅去检查设计是不可能获得这种感觉的。对于软件来说，构建这样一个模型既不可能也无必要。我们仅仅构建产品本身。即使正规的软件验证可以和编译器一样自动进行，我们还是会去进行构建
/测试循环。因此，正规的验证对于软件行业来说从来没有太多的实际
意义。
这就是现今软件开发过程的现实。数量不断增长的人和机构正在创建着更加复杂的软件设计。这些设计会被先用某些编程语言编写出来，然后通过构建
/测试循环进行验证和改
进。过程易于出错，并且不是特别的严格。相当多的软件开发人员并不想相信这就是过程的运作方式，也正因为这一点，使问题变得更加复杂。

当前大多数的软件过程都试图把软件设计的不同阶段分离到不同的类别中。必须要在顶层的设计完成并且冻结后，才能开始编码。测试和调试只对清除建造错误是必要的。程序员处在中间位置，他们是软件行业的建造工人。许多人认为，如果我们可以让程序员不

再进行"随意的编码（hacking）"并且按照交给他们的设计去进行构建（还要在过程中，犯更少的错误），那么软件开发就可以变得成熟，从而成为一门真正的工程学科。但是，只要过程忽视了工程和经济学事实，这就不可能发生。

例如，任何一个现代行业都无法忍受在其制造过程中出现超过100%的返工率。如果一个建造工人常常不能在第一次就构建正确，那么不久他就会失业。但是在软件业中，即使最小的一块代码，在测试和调试期间，也很可能会被修正或者完全重写。在一个创造性的过程中（比如：设计），我们认可这种改进不是制造过程的一部分。没有人会期望工程师第一次就创建出完美的设计。即使她做到了，仍然必须让它经受改进过程，目的就是为了证明它是完美的。

即使我们从日本的管理方法中没有学到任何东西，我们也应该知道由于在过程中犯错误而去责备工人是无益于提高生产率的。我们不应该不断地强迫软件开发去符合不正确的过程模型，相反，我们需要去改进过程，使之有助于而不是阻碍产生更好的软件。这就是"软件工程"的石蕊测试。工程是关于你如何实施过程的，而不是关于是否需要一个CAD系统来产生最终的设计文档。

关于软件开发有一个压倒性的问题，那就是一切都是设计过程的一部分。编码是设计，测试和调试是设计的一部分，并且我们通常认为的设计仍然是设计的一部分。虽然软件构建起来很廉价，但是设计起来却是难以置信的昂贵。软件非常的复杂，具有众多不同方面的设计内容以及它们所导致的设计考虑。问题在于，所有不同方面的内容是相互关连的（就像硬件工程中的一样）。我们希望顶层设计者可以忽视模块算法设计的细节。同样，我们希望程序员在设计模块内部算法时不必考虑顶层设计问题。糟糕的是，一个设计层面中的问题侵入到了其他层面之中。对于整个软件系统的成功来说，为一个特定模块选择算法可能和任何一个更高层次的设计问题同样重要。在软件设计的不同方面内容中，不存在重要性的等级。最低层模块中的一个不正确设计可能和最高层中的错误一样致命。软件设计必须在所有的方面都是完整和正确的，否则，构建于该设计基础之上的所有软件都会是错误的。

为了管理复杂性，软件被分层设计。当程序员在考虑一个模块的详细设计时，可能还有数以百计的其他模块以及数以千计的细节，他不可能同时顾及。例如，在软件设计中，有一些重要方面的内容不是完全属于数据结构和算法的范畴。在理想情况下，程序员不应该在设计代码时还得去考虑设计的这些其他方面的内容。

但是，设计并不是以这种方式工作的，并且原因也开始变得明朗。软件设计只有在其被编写和测试后才算完成。测试是设计验证和改进过程的基础部分。高层结构的设计不是完整的软件设计；它只是细节设计的一个结构框架。在严格地验证高层设计方面，我们的能力是非常有限的。详细设计最终会对高层设计造成的影响至少和其他的因素一样多（或者应该允许这种影响）。对设计的各个方面进行改进，是一个应该贯穿整个设计周期的过程。如果设计的任何一个方面内容被冻结在改进过程之外，那么对于最终设计将会是糟糕的或者甚至无法工作这一点，就不会觉得奇怪了。

如果高层的软件设计可以成为一个更加严格的工程过程，那该有多好呀，但是软件系统的真实情况不是严格的。软件非常的复杂，它依赖于太多的其他东西。或许，某些硬件没有按照设计者认为的那样工作，或者一个库例程具有一个文档中没有说明的限制。每一个软件项目迟早都会遇到这些种类的问题。这些种类的问题会在测试期间被发现（如果我们的测试工作做得好的话），之所以如此是因为没有办法在早期就发现它们。当它们被发现时，就迫使对设计进行更改。如果我们幸运，那么对设计的更改是局部的。时常，更改会波及到整个软件设计中的一些重要部分（莫非定律）。当受到影响的设计的一部分由于某种

原因不能更改时，那么为了能够适应影响，设计的其他部分就必须得遭到破坏。这通常导致的结果就是管理者所认为的"随意编码"，但是这就是软件开发的现实。

例如，在我最近工作的一个项目中，发现了模块
A的内部结构和另一个模块
B之间的一
个时序依赖关系。糟糕的是，模块
A的内部结构隐藏在一个抽象体的后面，而该抽象体不
允许以任何方法把对模块
B的调用合入到它的正确调用序列中。当问题被发现时，当然已
经错过了更改
A的抽象体的时机。正如所料，所发生的就是把一个日益增长的复杂的"修
正"集应用到
A的内部设计上。在我们还没有安装完版本
1时，就普遍感觉到设计正在衰退。
每一个新的修正很可能都会破坏一些老的修正。这是一个正规的软件开发项目。最后，我和我的同事决定对设计进行更改，但是为了得到管理层的同意，我们不得不自愿无偿加班。
在任何一般规模的软件项目中，肯定会出现像这样的问题，尽管人们使用了各种方法来防止它的出现，但是仍然会忽视一些重要的细节。这就是工艺和工程之间的区别。如果经验可以把我们引向正确的方向，这就是工艺。如果经验只会把我们带入未知的领域，然后我们必须使用一开始所使用的方法并通过一个受控的改进过程把它变得更好，这就是工程。
我们来看一下只是作为其中很小一点的内容，所有的程序员都知道，在编码之后而不是之前编写软件设计文档会产生更加准确的文档。现在，原因是显而易见的。用代码来表现的最终设计是惟一一个在构建
/测试循环期间被改进的东西。在这个循环期间，初始设计
保持不变的可能性和模块的数量以及项目中程序员的数量成反比。它很快就会变得毫无价值。
在软件工程中，我们非常需要在各个层次都优秀的设计。我们特别需要优秀的顶层设计。初期的设计越好，详细设计就会越容易。设计者应该使用任何可以提供帮助的东西。结构图表、Booch 图、状态表、
PDL等等——如果它能够提供帮助，就去使用它。但是，
我们必须记住，这些工具和符号都不是软件设计。最后，我们必须创建真正的软件设计，并且是使用某种编程语言完成的。因此，当我们得出设计时，我们不应该害怕对它们进行编码。在必要时，我们必须应该乐于去改进它们。
至今，还没有任何设计符号可以同时适用于顶层设计和详细设计。设计最终会表现为以某种编程语言编写的代码。这意味着在详细设计可以开始前，顶层设计符号必须被转换成目标编程语言。这个转换步骤耗费时间并且会引入错误。程序员常常是对需求进行回顾并且重新进行顶层设计，然后根据它们的实际去进行编码，而不是从一个可能没有和所选择的编程语言完全映射的符号进行转换。这同样也是软件开发的部分现实情况。
也许，如果让设计者本人来编写初始代码，而不是后来让其他人去转换语言无关的设计，就会更好一些。我们所需要的是一个适用于各个层次设计的统一符号。换句话说，我们需要一种编程语言，它同样也适用于捕获高层的设计概念。
C＋＋正好可以满足这个要
求。C++是一门适用于真实项目的编程语言，同时它也是一个非常具有表达力的软件设计语言。
C++允许我们直接表达关于设计组件的高层信息。这样，就可以更容易地进行设计，并且以后可以更容易地改进设计。由于它具有更强大的类型检查机制，所以也有助于检测到设计中的错误。这就产生了一个更加健壮的设计，实际上也是一个更好的工程化设计。

最后，软件设计必须要用某种编程语言表现出来，然后通过一个构建
/测试循环对其进
行验证和改进。除此之外的任何其他主张都完全没有用。请考虑一下都有哪些软件开发工具和技术得以流行。结构化编程在它的时代被认为是创造性的技术。
Pascal使之变得流行，
从而自己也变得流行。面向对象设计是新的流行技术，而
C++是它的核心。现在，请考虑
一下那些没有成效的东西。
CASE工具，流行吗？是的；通用吗？不是。结构图表怎么样？
情况也一样。同样地，还有
Warner-Orr图、Booch图、对象图以及你能想起的一切。每一个
都有自己的强项，以及惟一的一个根本弱点——它不是真正的软件设计。事实上，惟一一个可以被普遍认可的软件设计符号是
PDL，而它看起来像什么呢？
这表明，在软件业的共同潜意识中本能地知道，编程技术，特别是实际开发所使用的

编程语言的改进和软件行业中任何其他东西相比，具有压倒性的重要性。这还表明，程序员关心的是设计。当出现更加具有表达力的编程语言时，软件开发者就会使用它们。
同样，请考虑一下软件开发过程是如何变化的。从前，我们使用瀑布式过程。现在，
我们谈论的是螺旋式开发和快速原型。虽然这种技术常常被认为可以"消除风险"以及"缩短产品的交付时间"，但是它们事实上也只是为了在软件的生命周期中更早地开始编码。这是好事。这使得构建
/测试循环可以更早地开始对设计进行验证和改进。这同样也意味着，
顶层软件设计者很有可能也会去进行详细设计。
正如上面所表明的，工程更多的是关于如何去实施过程的，而不是关于最终产品看起来的像什么。处在软件行业中的我们，已经接近工程师的标准，但是我们需要一些认知上的改变。编程和构建
/测试循环是工程软件过程的中心。我们需要以像这样的方式去管理它
们。构建
/测试循环的经济规律，再加上软件系统几乎可以表现任何东西的事实，就使得我
们完全不可能找出一种通用的方法来验证软件设计。我们可以改善这个过程，但是我们不能脱离它。
最后一点：任何工程设计项目的目标是一些文档产品。显然，实际设计的文档是最重要的，但是它们并非惟一要产生的文档。最终，会期望某些人来使用软件。同样，系统很可能也需要后续的修改和增强。这意味着，和硬件项目一样，辅助文档对于软件项目具有同样的重要性。虽然暂时忽略了用户手册、安装指南以及其他一些和设计过程没有直接联系的文档，但是仍然有两个重要的需求需要使用辅助设计文档来解决。
辅助文档的第一个用途是从问题空间中捕获重要的信息，这些信息是不能直接在设计中使用的。软件设计需要创造一些软件概念来对问题空间中的概念进行建模。这个过程需要我们得出一个对问题空间中概念的理解。通常，这个理解中会包含一些最后不会被直接建模到软件空间中的信息，但是这些信息却仍然有助于设计者确定什么是本质概念以及如何最好地对它们建模。这些信息应该被记录在某处，以防以后要去更改模型。
对辅助文档的第二个重要需要是对设计的某些方面的内容进行记录，而这些方面的内容是难以直接从设计本身中提取的。它们既可以是高层方面的内容，也可以是低层方面内容。对于这些方面内容中的许多来说，图形是最好的描述方式。这就使得它们难以作为注释包含在代码中。这并不是说要用图形化的软件设计符号代替编程语言。这和用一些文本描述来对硬件科目的图形化设计文档进行补充没有什么区别。
决不要忘记，是源代码决定了实际设计的真实样子，而不是辅助文档。在理想情况下，可以使用软件工具对源代码进行后期处理并产生出辅助文档。对于这一点，我们可能期望过高了。次一点的情况是，程序员（或者技术方面的编写者）可以使用一些工具从源代码

中提取出一些特定的信息，然后可以把这些信息以其他一些方式文档化。毫无疑问，手工对这种文档保持更新是困难的。这是另外一个支持需要更具表达力的编程语言的理由。同样，这也是一个支持使这种辅助文档保持最小并且尽可能在项目晚期才使之变成正式的理由。同样，我们可以使用一些好的工具；不然的话，我们就得求助于铅笔、纸以及黑板。
总结如下：
实际的软件运行于计算机之中。它是存储在某种磁介质中的0和1的序列。它不是使用C++语言（或者其他任何编程语言）编写的程序。
程序清单是代表软件设计的文档。实际上把软件设计构建出来的是编译器和连接器。
构建实际软件设计的廉价程度是令人难以置信的，并且它始终随着计算机速度的加快而变得更加廉价。
设计实际软件的昂贵程度是令人难以置信的，之所以如此，是因为软件的复杂性是令人难以置信的，并且软件项目的几乎所有步骤都是设计过程的一部分。
编程是一种设计活动——好的软件设计过程认可这一点，并且在编码显得有意义时，

就会毫不犹豫的去编码。
编码要比我们所认为的更频繁地显现出它的意义。通常，在代码中表现设计的过程会揭示出一些疏漏以及额外的设计需要。这发生的越早，设计就会越好。
因为软件构建起来非常廉价，所以正规的工程验证方法在实际的软件开发中没有多大用处。仅仅建造设计并测试它要比试图去证明它更简单、更廉价。
测试和调试是设计活动——对于软件来说，它们就相当于其他工程学科中的设计验证和改进过程。好的软件设计过程认可这一点，并且不会试图去减少这些步骤。
还有一些其他的设计活动——称它们为高层设计、模块设计、结构设计、构架设计或者诸如此类的东西。好的软件设计过程认可这一点，并且慎重地包含这些步骤。
所有的设计活动都是相互影响的。好的软件设计过程认可这一点，并且当不同的设计步骤显示出有必要时，它会允许设计改变，有时甚至是根本上的改变，
许多不同的软件设计符号可能是有用的——它们可以作为辅助文档以及工具来帮助简化设计过程。它们不是软件设计。
软件开发仍然还是一门工艺，而不是一个工程学科。主要是因为缺乏验证和改善设计的关键过程中所需的严格性。
最后，软件开发的真正进步依赖于编程技术的进步，而这又意味着编程语言的进步。C++就是这样的一个进步。它已经取得了爆炸式的流行，因为它是一门直接支持更好的软件设计的主流编程语言。
z
C++在正确的方向上迈出了一步，但是还需要更大的进步。
后记
当我回顾几乎10年前所写的东西时，有几点让我印象深刻。第一点（也是和本书最有关的）是，现今，我甚至比那时更加确信我试图去阐述的要点在本质上的正确性。随后的一些年中，许多流行的软件开发方法增强了其中的许多观点，这支持了我的信念。最明显的（或许也是最不重要的）是面向对象编程语言的流行。现在，除了C++外，出现了许多其他的面向对象编程语言。另外，还有一些面向对象设计符号，比如：UML。我关于面向对象语言之所以得到流行是因为它们允许在代码中直接表现出更具表达力的设计的论点，现在看来有点过时了。
重构的概念——重新组织代码基础，使之更加健壮和可重用——同样也和我的关于设计的所有方面的内容都应该是灵活的并且在验证设计时允许改变的论点相似。重构只是提供了一个过程以及一组如何去改善已经被证实具有缺陷的设计的准则。

最后，文中有一个敏捷开发的总的概念。虽然极限编程是这些新方法中最知名的一个，但是它们都具有一个共同点：它们都承认源代码是软件开发工作中的最重要的产品。另一方面，有一些观点——其中的一些我在论文中略微谈到过——在随后的一些年中，对我来说变得更加重要。第一个是构架，或者顶层设计的重要性。在论文中，我认为构架只是设计的一部分内容，并且在构建/测试循环对设计进行验证的过程中，构架需要保持可变。这在本质上是正确的，但是回想起来，我认为我的想法有点不成熟。虽然构建/测试循环可能揭示出构架中的问题，但是更多的问题是常常由于改变需求而表现出来的。一般来说，设计软件是困难的，并且新的编程语言，比如：Java或者C++，以及图形化的符号，比如：UML，对于不知道如何有效地使用它的人来说，都没有多大的帮助。此外，一旦一个项目基于一个构架构建了大量的代码，那么对该构架进行基础性的更改，常常相当于丢弃掉该项目并重新开始一个，这就意味着该项目没有出现过。即使项目和机构在根本上接受了重构的概念，但是他们通常仍然不愿意去做一些看起来就像是完全重写的事情。这意味着第一次就把它作对（或者至少是接近对）是重要的，并且项目变得越大，就越要如此。幸运的是，软件设计模式有助于解决这方面问题。

还有其他一些方面的内容，我认为需要更多地强调一下，其中之一就是辅助文档，尤其是构架方面的文档。虽然源代码就是设计，但是试图从源代码中得出构架，可能是一个令人畏惧的体验。在论文中，我希望能够出现一些软件工具来帮助软件开发者自动地维护来自源代码的辅助文档。我几乎已经放弃了这个希望。一个好的面向对象构架通常可以使用几幅图以及少许的十几页文本描述出来。不过，这些图（和文本）必须集中于设计中的关键类和关系。糟糕的是，对于软件设计工具可能会变得足够聪明，以至于可以从源代码的大量细节中提取出这些重要方面的内容这一点，我没有看到任何真正的希望。这意味着还得必须由人来编写和维护这种文档。我仍然认为，在源代码完成后，或者至少是在编写源代码的同时去编文档，要比在编写源代码之前去编写文档更好一些。

最后，我在论文的最后谈到了C++是编程——并且因此是软件设计——艺术的一个进步，但是还需要更大的进步。就算我完全没有看到语言中出现任何真正的编程进步来挑战C++的流行，那么在今天，我会认为这一点甚至要比我首次编写它时更加正确。