敏捷建模

AM（敏捷建模）是一种态度，而不是一个说明性的过程。AM 是敏捷建模者们坚持的价值观、敏捷建模者们相信的原则、敏捷建模者们应用的实践组成的集合。 AM 描述了一种建模的风格。当它应用于敏捷的环境中时，能够提高开发的质量和速度，同时能够避免过度简化和不切实际的期望。 AM 可不是开发的“食谱”，如果你寻觅的是一些细节的指导，如建立 UML 顺序图或是画出用户界面流图，你可以看看在建模 Artifacts 中列出的许多建模书籍，我特别推荐我的书 The Object Primer 2/e(尽管这有失公允)。

AM 是对已有方法的补充，而不是一个完整的方法论。 AM 的主要焦点是在建模上，其次是文档。也就是说，AM 技术在你的团队采用敏捷方法（例如 eXtreme Programming，Dynamic Systems Development Method （DSDM），Crystal Clear）的基础上能够提高建模的效果 。AM 同样也可以用于那些传统过程（例如 Unified Process），尽管这种过程较低的敏捷性会使得 AM 不会那么成功。

AM 是一种有效的共同工作的方法，能够满足 Project Stakeholder 的需要。敏捷开发者们和 Project Stakeholder 进行团队协作，他们轮流在系统开发中扮演着直接、主动的角色。在“敏捷”的字典中没有“我”这个单词。

AM 是有效的，而且也已开始有效。当你学习到更多的 AM 知识时，有件事对你来说可能不好接受，AM 近乎无情的注重有效性。AM 告诉你：要使你的 Project Stakeholder 的投资最大化；当有清晰的目的以及需要解了受众的需要时要建立模型或文档；运用合适的工件来记录手头的情形；不论何时都尽可能创建简单的模型。

AM 是来自于实践中，而不是象牙塔般的理论。AM 的目标就是以一种有效的态度描述系统建模的技术，它有效率，足够胜任你手头的工作。我和我在 Ronin International (http://www.ronin-intl.com) 的同事将大量的 AM 技术应用于实践已经很多年了，我们琢磨的这些技术应用于非常广泛的客户，他们遍布各个不同的工业领域。而且，从 2001 年 2 月开始，就有数百位的建模专家通过“敏捷建模邮件列表”(http://www.agilemodeling.com/feedback.htm) 对这些技术进行过充分的检查和讨论。

AM 不是灵丹妙药。敏捷建模是改进众多专家软件开发成果的有效技术，充其量也就是这样了。它并不是什么了不得的灵丹妙药，能够解决你开发中的所有问题。如果你努力的工作；如果你专注其上；如果打心眼儿里接受它的价值观、它的原则、它的实践；你就可以改进你做为一个开发人员的效果。

AM 是面向一般的开发人员的，但并不是要排斥有能力的人。AM 的价值观、原则和实践都简单易懂，其中的很多内容，可能你都已经采用或期待多年了。应用 AM 技术并不是要你去练水上飘，但你需要有一些基本的软件开发技能。AM 最难的就是它逼着你去学习更广泛的建

模技术，这是个长期的、持续性的活动。学习建模在一开始可能很难，但你可以试着一次学习一样技术来完成你的学习。

AM 并不是要反对文档。文档的创建和维护都会增大项目涉众的投资。敏捷文档尽可能的简单，尽可能的小，目的只集中在和目前开发的系统有直接关系的事情上，充分了解受众的需要。

AM 也不是要反对 CASE 工具。敏捷建模者使用那些能够帮助开发人员提高效果，提升价值的工具。而且，他们还尽力使用那些能够胜任工作的最简单的工具。

AM 并不适合每一个人。

一、敏捷建模的价值观

AM 的价值观包括了 XP 的四个价值观：沟通、简单、反馈、勇气，此外，还扩展了第五个价值观：谦逊。

沟通. 建模不但能够促进你团队内部的开发人员之间沟通、还能够促进你的团队和你的 project stakeholder 之间的沟通。

简单. 画一两张图表来代替几十甚至几百行的代码，通过这种方法，建模成为简化软件和软件（开发）过程的关键。这一点对开发人员而言非常重要－它简单，容易发现出新的想法，随着你（对软件）的理解的加深，也能够很容易的改进。

反馈. Kent Beck 在 Extreme Programming Explained 中有句话讲得非常好：“乐观是编程的职业病，反馈则是其处方。”通过图表来交流你的想法，你可以快速获得反馈，并能够按照建议行事。

勇气. 勇气非常重要，当你的决策证明是不合适的时候，你就需要做出重大的决策，放弃或重构（refactor）你的工作，修正你的方向。

谦逊. 最优秀的开发人员都拥有谦逊的美德，他们总能认识到自己并不是无所不知的。事实上，无论是开发人员还是客户，甚至所有的 project stakeholder，都有他们自己的专业领域，都能够为项目做出贡献。一个有效的做法是假设参与项目的每一个人都有相同的价值，都应该被尊重。

二、敏捷建模的原则

敏捷建模（AM）定义了一系列的核心原则和辅助原则，它们为软件开发项目中的建模实践奠定了基石。其中一些原则是从 XP 中借鉴而来，在 Extreme Programming Explained 中有它们的详细描述。而 XP 中的一些原则又是源于众所周知的软件工程学。复用的思想随处可见！基本上，本文中对这些原则的阐述主要侧重于它们是如何影响着建模工作；这样，对于这些借鉴于 XP 的原则，我们可以从另一个角度来看待。

核心原则：

主张简单. 当从事开发工作时，你应当主张最简单的解决方案就是最好的解决方案。不要过分构建（overbuild）你的软件。用 AM 的说法就是，如果你现在并不需要这项额外功能，那就不要在模型中增加它。要有这样的勇气：你现在不必要对这个系统进行过分的建模（over-model），只要基于现有的需求进行建模，日后需求有变更时，再来重构这个系统。尽可能的保持模型的简单。

拥抱变化. 需求时刻在变，人们对于需求的理解也时刻在变。项目进行中，Project stakeholder 可能变化，会有新人加入，也会有旧人离开。Project stakeholder 的观点也可能变化，你努力的目标和成功标准也有可能发生变化。这就意味着随着项目的进行，项目环境也在不停的变化，因此你的开发方法必须要能够反映这种现实。

你的第二个目标是可持续性. 即便你的团队已经把一个能够运转的系统交付给用户，你的项目也还可能是失败的——实现 Project stakeholder 的需求，其中就包括你的系统应该要有足够的鲁棒性（robust ），能够适应日后的扩展。就像 Alistair Cockburn 常说的，当你在进行软件开发的竞赛时，你的第二个目标就是准备下一场比赛。可持续性可能指的是系统的下一个主要发布版，或是你正在构建的系统的运转和支持。要做到这一点，你不仅仅要构建高质量的软件，还要创建足够的文档和支持材料，保证下一场比赛能有效的进行。你要考虑很多的因素，包括你现有的团队是不是还能够参加下一场的比赛，下一场比赛的环境，下一场比赛对你的组织的重要程度。简单的说，你在开发的时候，你要能想象到未来。

递增的变化. 和建模相关的一个重要概念是你不用在一开始就准备好一切。实际上，你就算想这么做也不太可能。而且，你不用在模型中包容所有的细节，你只要足够的细节就够了。没有必要试图在一开始就建立一个囊括一切的模型，你只要开发一个小的模型，或是概要模型，打下一个基础，然后慢慢的改进模型，或是在不在需要的时候丢弃这个模型。这就是递增的思想。

令 Stakeholder 投资最大化. 你的 project stakeholder 为了开发出满足自己需要的软件，需要投入时间、金钱、设备等各种资源。stakeholder 应该可以选取最好的方式投资，也可以要求你的团队不浪费资源。并且，他们还有最后的发言权，决定要投入多少的资源。如果是这些资源是你自己的，你希望你的资源被误用吗。

有目的的建模.对于自己的 artifact，例如模型、源代码、文档，很多开发人员不是担心它们是否够详细，就是担心它们是否太过详细，或担心它们是否足够正确。你不应该毫无意义的建模，应该先问问，为什么要建立这个 artifact，为谁建立它。和建模有关，也许你应该更多的了解软件的某个方面，也许为了保证项目的顺利进行，你需要和高级经理交流你的方法，也许你需要创建描述系统的文档，使其他人能够操作、维护、改进系统。如果你连为什么建模，为谁建模都不清楚，你又何必继续烦恼下去呢？首先，你要确定建模的目的以及模型的受众，在此基础上，再保证模型足够正确和足够详细。一旦一个模型实现了目标，你就可以

结束目前的工作，把精力转移到其它的工作上去，例如编写代码以检验模型的运作。该项原则也可适用于改变现有模型：如果你要做一些改变，也许是一个熟知的模式，你应该有做出变化的正确理由（可能是为了支持一项新的需求，或是为了重构以保证简洁）。关于该项原则的一个重要暗示是你应该要了解你的受众，即便受众是你自己也一样。例如，如果你是为维护人员建立模型，他们到底需要些什么？是厚达 500 页的详细文档才够呢，还是 10 页的工作总览就够了？你不清楚？去和他们谈谈，找出你想要的。

多种模型.开发软件需要使用多种模型，因为每种模型只能描述软件的单个方面，“要开发现今的商业应用，我们该需要什么样的模型？”考虑到现今的软件的复杂性，你的建模工具箱应该要包容大量有用的技术（关于 artifact 的清单，可以参阅 AM 的建模 artifact）。有一点很重要，你没有必要为一个系统开发所有的模型，而应该针对系统的具体情况，挑选一部分的模型。不同的系统使用不同部分的模型。比如，和家里的修理工作一样，每种工作不是要求你用遍工具箱里的每一个工具，而是一次使用某一件工具。又比如，你可能会比较喜欢某些工具，同样，你可会偏爱某一种模型。有多少的建模 artifact 可供使用呢，如果你想要了解这方面的更多细节，我在 Be Realistic About the UML 中列出了 UML 的相关部分，如果你希望做进一步的了解，可以参阅白皮书 The Object Primer -- An Introduction to Techniques for Agile Modeling。

高质量的工作.没有人喜欢烂糟糟的工作。做这项工作的人不喜欢，是因为没有成就感；日后负责重构这项工作（因为某些原因）的人不喜欢，是因为它难以理解，难以更新；最终用户不喜欢，是因为它太脆弱，容易出错，也不符合他们的期望。

快速反馈.从开始采取行动，到获得行动的反馈，二者之间的时间至关紧要。和其他人一共开发模型，你的想法可以立刻获得反馈，特别是你的工作采用了共享建模技术的时候，例如白板、CRC 卡片或即时贴之类的基本建模材料。和你的客户紧密工作，去了解他们的的需求，去分析这些需求，或是去开发满足他们需求的用户界面，这样，你就提供了快速反馈的机会。

软件是你的主要目标. 软件开发的主要目标是以有效的方式，制造出满足 project stakeholder 需要的软件，而不是制造无关的文档，无关的用于管理的 artifact，甚至无关的模型。任何一项活动（activity ），如果不符合这项原则，不能有助于目标实现的，都应该受到审核，甚至取消。

轻装前进.你建立一个 artifact，然后决定要保留它，随着时间的流逝，这些 artifact 都需要维护。如果你决定保留 7 个模型，不论何时，一旦有变化发生（新需求的提出，原需求的更新，团队接受了一种新方法，采纳了一项新技术...），你就需要考虑变化对这 7 个模型产生的影响并采取相应的措施。而如果你想要保留的仅是 3 个模型，很明显，你实现同样的改变要花费的功夫就少多了，你的灵活性就增强了，因为你是在轻装前进。类似的，你的模型越复杂，越详细，发生的改变极可能就越难实现（每个模型都更“沉重”了些，因此维护的负担也就大

了）。每次你要决定保留一个模型时，你就要权衡模型载有的信息对团队有多大的好处（所以才需要加强团队之间，团队和 project stakeholder 之间的沟通）。千万不要小看权衡的严重性。一个人要想过沙漠，他一定会携带地图，帽子，质地优良的鞋子，水壶。如果他带了几百加仑的水，能够想象的到的所有求生工具，一大堆有关沙漠的书籍，他还能过得去沙漠吗？同样的道理，一个开发团队决定要开发并维护一份详细的需求文档，一组详细的分析模型，再加上一组详细的架构模型，以及一组详细的设计模型，那他们很快就会发现，他们大部分的时间不是花在写源代码上，而是花在了更新文档上。

补充原则：

内容比表示更重要.一个模型有很多种的表示方法。例如，可以通过在一张纸上放置即时贴的方法来建立一个用户界面规格（基本/低精度原型）。它的表现方式可以是纸上或白板上的草图，可以是使用原型工具或编程工具建立和传统的原型，也可以是包括可视界面和文本描述的正式文档。有一点很有意思，一个模型并不一定就是文档。它们通常作为其它 artifact 的输入，例如源代码，但不必把它们处理为正式的文档，即使是使用 CASE 工具建立的复杂的图表，也是一样。要认识到一点，要利用建模的优点，而不要把精力花费在创建和维护文档上。

三人行必有我师.你不可能完全精通某项技术，你总是有机会学习新的知识，拓展知识领域。把握住这个机会，和他人一同工作，向他人学习，试试做事的新方式，思考什么该做，什么不该做。技术的变化非常的快，现有的技术（例如 Java）以难以置信的速度在改进，新的技术（例如 C#和.NET）也在有规律的产生。现存开发技术的改进相对会慢一些，但也在持续的改进中－－计算机产业属于工业，我们已经掌握了其中的试验基本原理，但我们还在不断的研究，不断的实践，不断的改进我们对它的了解。我们工作在一个变化是家常便饭的产业中，我们必须通过训练、教育、思考、阅读、以及和他人合作，抓住每一个机会学习新的处事之道。

了解你的模型.因为你要使用多种模型，你需要了解它们的优缺点，这样才能够有效的使用它们。

了解你的工具.软件（例如作图工具、建模工具）有各种各样的特点。如果你打算使用一种建模工具，你就应当了解什么时候适合用它，什么时候不适合用它。

局部调整. 你的软件开发方法要能够反映你的环境，这个环境包括组织特征，project stakeholder 的特征，项目自身的特征。有可能受其影响的问题包括：你使用的建模技术（也许你的用户坚持要看到一个细节的用户界面，而不是初始草图或基本原型）；你使用的工具（可能你没有数字照相机的预算，或是你已经拥有某个 CASE 工具的 license）；你遵循的软件过程（你的组织采用 XP 的开发过程，或是 RUP，或是自己的过程）。因此你会调整你的方法，这种调整可能是针对项目的，也可能是针对个人的。例如，有些开发人员倾向于使用某

一类工具，有些则不用。有些人在编码上花大力气，基本不做建模，有些则宁可在建模上多投入一些时间。

开放诚实的沟通.人们需要能够自由的提出建议，而且人们还应该能够感受到他们是自由的。建议可能是和模型有关的想法：也许是某些人提出某部分新的设计方法，或是某个需求的新的理解；建议还可能是一些坏消息，例如进度延误；或仅仅是简单的工作状况报告。开放诚实的沟通是人们能够更好的决策，因为作为决策基础的信息会更加准确。

利用好人的直觉.有时你会感觉到有什么地方出问题了，或是感觉什么地方有不一致的情况，或是某些东西感觉不是很对。其实，这种感觉很有可能就是事实。随着你的软件开发的经验的增加，你的直觉也会变得更敏锐，你的直觉下意识之间告诉你的，很可能就是你工作的关键之处。如果你的直觉告诉你一项需求是没有意义的，那你就不用投入大量的精力和用户讨论这方面的问题了。如果你的直觉告诉你有部分的架构不能满足你的需要，那就需要建立一个快速技术原型来验证你的理论。如果你的直觉告诉设计方法 A 要比设计方法 B 好，而且并没有强有力的理由支持你选择某一个方法，那就尽管选择方法 A。勇气的价值就已经告诉你，如果未来证实你的直觉是错的，你也有能力来挽救这种情况。你应该有这种自信，这很重要。

三、敏捷建模的实践

敏捷建模（AM）在 AM 原则的基础上定义了一组核心实践（practice）和补充实践，其中的某些实践已经是极限编程（XP）中采用了的，并在 Extreme Programming Explained 一书中有详细的论述，和 AM 的原则一样，我们在描述这组实践时，将会注重于建模的过程，这样你可以从另外一个角度来观察这些已或 XP 采用的素材。

核心实践：

Stakeholder 的积极参与.我们对 XP 的现场客户（On-Site Customer）的概念做了一个扩充：开发人员需要和用户保持现场的接触；现场的用户要有足够的权限和能力，提供目前建构中的系统相关的信息；及时、中肯的做出和需求相关的决策；并决定它们的优先级。AM 把 XP 的“现场客户”实践扩展为“使 project stakeholder 积极参与项目”，这个 project stakeholder 的概念包括了直接用户、他们的经理、高级经理、操作人员、支持人员。这种参与包括：高级经理及时的资源安排决策，高级经理的对项目的公开和私下的支持，需求开发阶段操作人员和支持人员的积极参与，以及他们在各自领域的相关模型。

正确使用 artifact.每个 artifact 都有它们各自的适用之处。例如，一个 UML 的活动图（activity diagram）适合用于描述一个业务流程，反之，你数据库的静态结构，最好能够使用物理数据（physical data）或数据模型（persistence model）来表示。在很多时候，一张图表比源代码更能发挥作用，一图胜千言，同样，一个模型也比 1K 的源代码有用的多，前提是使用得当（这里借用了 Karl Wieger 的 Software Requirements 中的词汇）。因为你在研究设计方案时，你可和同伴们和在白板上画一些图表来讨论，也可以自己坐下来开发一些代码样

例，而前一种方法要有效的多。。这意味着什么？你需要了解每一种 artifact 的长处和短处，当你有众多的模型可供选择的时候，要做到这一点可没有那么容易。

集体所有制.只要有需要，所有人都可以使用、修改项目中的任何模型、任何 artifact。

测试性思维.当你在建立模型的时候，你就要不断的问自己，“我该如何测试它？”如果你没办法测试正在开发的软件，你根本就不应该开发它。在现代的各种软件过程中，测试和质保（quality assurance）活动都贯穿于整个项目生命周期，一些过程更是提出了“在编写软件之前先编写测试”的概念（这是 XP 的一项实践：“测试优先”）。

并行创建模型.由于每种模型都有其长处和短处，没有一个模型能够完全满足建模的需要。例如你在收集需求时，你需要开发一些基本用例或用户素材，一个基本用户界面原型，和一些业务规则。再结合实践切换到另外的 Artifact，，敏捷建模者会发现在任何时候，同时进行多个模型的开发工作，要比单纯集中于一个模型要有效率的多。

创建简单的内容.你应该尽可能的使你的模型（需求、分析、架构、设计）保持简单，但前提是能够满足你的 project stakeholder 的需要。这就意味着，除非有充分的理由，你不应该随便在模型上画蛇添足——如果你手头上没有系统认证的功能，你就不应该给你的模型增加这么一个功能。要有这样的勇气，一旦被要求添加这项功能，自己就能够马上做到。这和 XP 的实践“简单设计”的思想是一样的。

简单地建模.当你考虑所有你能够使用的图表（UML 图、用户界面图、数据模型等）时，你很快会发现，大部分时候你只需要这些图表符号的一部分。一个简单的模型能够展示你想要了解的主要功能，例如，一个类图，只要能够显示类的主要责任和类之间的关系就已经足够了。不错，编码的标准告诉你需要在模型中加入框架代码，比如所有的 get 和 set 操作，这没有错，但是这能提供多少价值呢？恐怕很少。

公开展示模型.你应当公开的展示你的模型，模型的载体被称为“建模之墙”（modeling wall）或“奇迹之墙（wall of wonder）”。这种做法可以在你的团队之间、你和你的 project stakeholder 之间营造出开放诚实的沟通氛围，因为当前所有的模型对他们都是举手可得的，你没有向他们隐藏什么。你把你的模型贴到建模之墙上，所有的开发人员和 project stakeholder 都可以看建模之墙上的模型，建模之墙可能是客观存在的，也许是一块为你的架构图指定的白板，或是物理数据模型的一份打印输出，建模之墙也可能是虚拟的，例如一个存放扫描好的图片的 internet 网页。如果你想要多了解一些相关的资料，你可以看看 Ellen Gottesdiener 的 Specifying Requirements With a Wall of Wonder。

切换到另外的 Artifact.当你在开发一个 artifact（例如用例、CRC 卡片、顺序图、甚至源码），你会发现你卡壳了，这时候你应当考虑暂时切换到另一个 artifact。每一个 artifact 都有自己的长处和短处，每一个 artifact 都适合某一类型的工作。无论何时你发现你在某个 artifact 上卡壳了，没办法再继续了，这就表示你应该切换到另一个 artifact 上去。举个例子，

如果你正在制作基本用例，但是在描述业务规则时遇到了困难，你就该试着把你的注意力转移到别的 artifact 上去，可能是基本用户界面原型、CRC 模型，可能是业务规则、系统用例、或变化案例。切换到另一个 artifact 上去之后，你可能就立刻不再卡壳了，因为你能够在另一个 artifact 上继续工作。而且，通过改变你的视角，你往往会发现原先使你卡壳的原因。

小增量建模. 采用增量开发的方式，你可以把大的工作量分成能够发布的小块，每次的增量控制在几个星期或一两个月的时间内，促使你更快的把软件交付给你的用户，增加了你的敏捷性。

和他人一起建模.当你有目的建模时你会发现，你建模可能是为了了解某事，可能是为了同他人交流你的想法，或是为了在你的项目中建立起共同的愿景。这是一个团体活动，一个需要大家有效的共同工作才能完成的活动。你发现你的开发团队必须共同协作，才能建立一组核心模型，这对你的项目是至关重要的。例如，为了建立系统的映像和架构，你需要和同组成员一起建立所有人都赞同的解决方案，同时还要尽可能的保持它的简单性。大多数时候，最好的方法是和另一些人讨论这个问题。

用代码验证.模型是一种抽象，一种能够正确反映你正在构建的系统的某个方面的抽象。但它是否能运行呢？要知道结果，你就应该用代码来验证你的模型。你已经用一些 HTML 页面建立了接受付款地址信息的草图了吗？编码实现它，给你的用户展示最终的用户界面，并获取反馈。你已经做好了表示一个复杂业务规则逻辑的 UML 顺序图了吗？写出测试代码，业务代码，运行测试以保证你做的是对的。永远也别忘了用迭代的方法开发软件（这是大多数项目的标准做法），也别忘了建模只是众多任务中的一个。做一会儿建模、做一会儿编码、做一会儿测试（在其它的活动之中进行）。

使用最简单的工具.大多数的模型都可以画在白板上，纸上，甚至纸巾的背面。如果你想要保存这些图标，你可以用数码相机把它们拍下来，或只是简单的把他们转录到纸上。这样做是因为大多数的图表都是可以扔掉的，它们只有在你画出模型并思考一个问题的时候才有价值，一旦这个问题被解决了它们就不再有意义了。这样，白板和标签往往成为你建模工具的最佳选择：使用画图工具来创建图表，给你重要的 project stakeholder 看。只有建模工具能够给我们的编程工作提供价值（例如代码自动生成）时才使用建模工具。你可以这样想：如果你正在创建简单的模型，这些模型都是可以抛弃的。你建模的目的就是为了理解，一旦你理解了问题，模型就没有存在的必要了，因此模型都是可以丢弃的，这样，你根本就不必要使用一个复杂的建模工具。

补充实践：

使用建模标准.这项实践是从 XP 的编码标准改名而来，基本的概念是在一个软件项目中开发人员应该同意并遵守一套共同的建模标准。遵守共同的编码惯例能够产生价值：遵守你选择的编码指南能够写出干净的代码，易于理解，这要比不这么做产生出来的代码好得多。同样，

遵守共同的建模标准也有类似的价值。目前可供选择的建模标准有很多，包括对象管理组织（OMG）制定的统一建模语言(UML)，它给通用的面向对象模型定义了符号和语义。UML 开了一个好头，但并不充分－就像你在 Be Realistic About The UML 中看到的，UML 并没有囊括所有可能的的建模 artifact。而且，在关于建立清楚可看的图表方面，它没有提供任何建模风格指南。那么，风格指南和标准之间的差别在何处呢。对源代码来说，一项标准可能是规定属性名必须以 attributeName 的格式，而风格指南可能实说在一个单元中的一段控制结构（一个 if 语句，一段循环）的代码缩进。对模型来说，一项标准可能是使用一个长方形对类建模，一项风格指南可能是图中子类需要放在父类的下方。

逐渐应用模式.高效的建模者会学习通用的架构模式、设计模式和分析模式，并适当的把它们应用在模型之中。然而，就像 Martin Fowler 在 Is Design Dead 中指出的那样，开发人员应当轻松的使用模式，逐渐的应用模式。这反映了简单的价值观。换言之，如果你猜测一个模式可能适用，你应当以这样的方式建模：先实现目前你需要的最小的范围，但你要为日后的重构留下伏笔。这样，你就以一种可能的最简单的方式实现了一个羽翼丰满的模式了。就是说，不要超出你的模型。举一个例子，在你的设计中，你发现有个地方适合使用 GoF 的 Strategy 模式，但这时候你只有两个算法要实现。最简单的方法莫过于把算法封装为单独的类，并建立操作，能够选择相应的算法，以及为算法传递相关的输入。这是 Strategy 模式的部分实现，但你埋下了伏笔，日后如有更多的算法要实现，你就可以重构你的设计。并没有必要因为 Strategy 模式需要，就建立所有的框架。这种方法使你能够轻松的使用模式。

丢弃临时模型.你创建的大部分的模型都是临时使用的模型－－设计草图，低精度原型，索引卡片，可能架构/设计方案等等－－在它们完成了它们的目的之后就再不能提供更多的价值了。模型很快就变得无法和代码同步，这是正常的。你需要做出决定：如果“同步更新模型”的做法能够给你的项目增添价值的话，那就同步更新模型；或者，如果更新它们的投入将抵消它们能够提供的所有价值（即负收益），那就丢弃它们。

合同模型要正式.在你的系统需要的信息资源为外部组织所控制的时候，例如数据库，旧有系统和信息服务，你就需要合同模型。一个合同模型需要双方都能同意，根据时间，根据需要相互改变。合同模型的例子有 API 的细节文档，存储形式描述，XML DTD 或是描述共享数据库的物理数据模型。作为法律合同，合同模型通常都需要你投入重要资源来开发和维护，以确保它的正确、详细。你的目标是尽量使你系统的合同模型最少，这和 XP 的原则 traveling light 是一致的。注意你几乎总是需要电子工具来建立合同模型，因为这个模型是随时需要维护的。

为交流建模. 建模的次要原因是为了和团队之外的人交流或建立合同模型。因为有些模型是给团队之外的客户的，你需要投入时间，使用诸如文字处理器，画图工具包，甚至是那些“被广告吹得天花乱坠”的 CASE 工具来美化模型。

为理解建模.建模的最重要的应用就是探索问题空间，以识别和分析系统的需求，或是比较和对照可能的设计选择方法，以识别可能满足需求的、最简单的解决方案。根据这项实践，你通产需要针对软件的某个方面建立小的、简单的图表，例如类的生命周期图，或屏幕顺序，这些图表通常在你完成目的（理解）之后就被丢弃。

重用现有的资源.这是敏捷建模者能够利用的信息财富。例如，也许一些分析和设计模式适合应用到系统上去，也许你能够从现有的模型中获利，例如企业需求模型，业务过程模型，物理数据模型，甚至是描述你用户团体中的系统如何部署的模型。但是，尽管你常常搜索一些比较正确的模型，可事实是，在大多数组织中，这些模型要么就不存在，要么就已经过期了。

非到万不得已不更新.你应当在你确实需要时才更新模型，就是说，当不更新模型造成的代价超出了更新模型所付出的代价的时候。使用这种方法，你会发现你更新模型的数量比以前少多了，因为事实就是，并不是那么完美的模型才能提供价值的。我家乡的街道图已经使用了5 年了，5 年来我自己街道并没有改变位置，因此这张地图对我来说还是有用的。不错，我可以买一张新地图，地图是每年出一次的，但为什么要这么麻烦呢？缺少一些街道并没有让我痛苦到不得不投资买一份新地图。简单的说，当地图还管用的时候，每年花钱买新地图是没有任何意义的。为了保持模型、文档和源代码之间的同步，已经浪费了太多太多的时间和金钱了，而同步是不太可能做到的。时间和金钱投资到新的软件上不是更好吗？

确实不错的主意：

以下的实践虽然没有包括在 AM 中，但是可以做为 AM 的一份补充：

重构.这是一项编码实践。重构，就是通过小的变化，使你的代码支持新的功能，或使你的设计尽可能的简单。从 AM 的观点来看，这项实践可以保证你在编码时，你的设计干净、清楚。重构是 XP 的一个重要部分。

测试优先设计.这是一项开发实践。在你开始编写你的业务代码之前，你要先考虑、编写你的测试案例。从 AM 的观点来看，这项实践强制要求你在写代码之前先通盘考虑你的设计，所以你不再需要细节设计建模了。测试优先设计是 XP 的一个重要部分。

四、敏捷建模是（不是）什么？

我坚信当在描述事物的范围时，你需要说明它是什么，它不是什么。不管你谈论的是系统还是案例中的 AM 都一样。以下就是我对 AM 的范围的观点：

AM 是一种态度，而不是一个说明性的过程。AM 是敏捷建模者们坚持的价值观、敏捷建模者们相信的原则、敏捷建模者们应用的实践组成的集合。 AM 描述了一种建模的风格。当它应用于敏捷的环境中时，能够提高开发的质量和速度，同时能够避免过度简化和不切实际的期望。 AM 可不是开发的“食谱”，如果你寻觅的是一些细节的指导，如建立 UML 顺序图或是画出用户界面流图，你可以看看在建模 Artifacts 中列出的许多建模书籍，我特别推荐我的书 The Object Primer 2/e(尽管这有失公允)。

AM 是对已有方法的补充，而不是一个完整的方法论。 AM 的主要焦点是在建模上，其次是文档。也就是说，AM 技术在你的团队采用敏捷方法（例如 eXtreme Programming，Dynamic Systems Development Method （DSDM），Crystal Clear）的基础上能够提高建模的效果 。AM 同样也可以用于那些传统过程（例如 Unified Process），尽管这种过程较低的敏捷性会使得 AM 不会那么成功。

AM 是一种有效的共同工作的方法，能够满足 Project Stakeholder 的需要。敏捷开发者们和 Project Stakeholder 进行团队协作，他们轮流在系统开发中扮演着直接、主动的角色。在“敏捷”的字典中没有“我”这个单词。

AM 是有效的，而且也已开始有效。当你学习到更多的 AM 知识时，有件事对你来说可能不好接受，AM 近乎无情的注重有效性。AM 告诉你：要使你的 Project Stakeholder 的投资最大化；当有清晰的目的以及需要解了受众的需要时要建立模型或文档；运用合适的工件来记录手头的情形；不论何时都尽可能创建简单的模型。

AM 是来自于实践中，而不是象牙塔般的理论。AM 的目标就是以一种有效的态度描述系统建模的技术，它有效率，足够胜任你手头的工作。我和我在 Ronin International (http://www.ronin-intl.com) 的同事将大量的 AM 技术应用于实践已经很多年了，我们琢磨的这些技术应用于非常广泛的客户，他们遍布各个不同的工业领域。而且，从 2001 年 2 月开始，就有数百位的建模专家通过“敏捷建模邮件列表”(http://www.agilemodeling.com/feedback.htm) 对这些技术进行过充分的检查和讨论。

AM 不是灵丹妙药。敏捷建模是改进众多专家软件开发成果的有效技术，充其量也就是这样了。它并不是什么了不得的灵丹妙药，能够解决你开发中的所有问题。如果你努力的工作；如果你专注其上；如果打心眼儿里接受它的价值观、它的原则、它的实践；你就可以改进你做为一个开发人员的效果。