“AI写的代码，我完全不看”，世界级编程大师Bob大叔警示：说软件基础不重要的人会吃苦头，不会等太久

CSDN 2026年8月21日 18:00 江苏

CSDN
关注技术世界的前沿动态，深入挖掘其背后的技术趋势与实践方法，助力技术人把握时代...
3334篇原创内容

公众号

整理 | 屠敏

出品 | CSDN（ID：CSDNnews）

“我完全不看 Agent 写出来的任何代码。”

就在一个月前，这句话从 Robert C. Martin 口中说出时，引起了不小的争议。

Robert C. Martin，这位世界级编程大师、《代码整洁之道》的作者，也是全球开发者熟知的“Bob 大叔”（Uncle Bob），已经编程超过 50 年。到了 AI 时代，他却开始尝试一件让不少程序员感到意外的事情：**让 Agent 写代码，而自己不再逐行检查。**

对于习惯了 Code Review 的开发者来说，这套方法显然很难接受。很多人不禁质疑：如果 AI 写的代码不再需要人类逐行阅读，那么程序员究竟该如何判断它是对的？

显然，Bob 大叔并不是打算“闭着眼睛”接受 AI 生成的代码。他给 Agent 加上了一层又一层的约束：单元测试、Gherkin 测试、QA 流程、质量指标、变异测试、测试覆盖率，以及大量其他自动化检查。只有当代码通过这些测试和质量检查，他才会对最终结果建立起“很高的信心”。

那么，他究竟是怎么做到的？在这个过程中，又有哪些“坑”？

近日，Bob 大叔与知名 TypeScript 教育家 Matt Pocock 进行了一场直播对谈，讨论 AI 时代的软件基础，以及自己最近几个月与 AI Agent 一起写代码的经历。

在他看来，关键并不是放弃软件工程，而是改变人类与代码之间的分工：与其让人逐行检查 Agent 写出的代码，不如建立一套确定性的规则和工具，让它们自动判断代码是否达标。

而当谈及如何识别“垃圾”代码时，Bob 大叔坦言，自己能够看出 Agent 什么时候开始“挣扎”，因为作为一名程序员，他也经历过同样的挣扎。但问题在于，一个刚入行的程序员，可能根本识别不出这种挣扎。

这也引出了他对 AI 时代软件开发者的另一个提醒：软件基础知识并没有过时。**“现在有些人认为软件基础已经不重要了。他们会吃到苦头，而且不会等太久。”**

当 Agent 写代码的速度远远超过人类，程序员究竟应该把时间花在哪里？

这场对谈，也许给出了一个值得思考的答案。

完整采访详见：https://www.youtube.com/watch?v=zcLPGC-tvgk

60 余载的编程之路，Bob 大叔的“浴袍梗”

Matt Pocock：大家下午好，今天我准备了一份特别的惊喜。我请到了一位我一直想交流的人，他在软件领域的影响力贯穿了我的整个职业生涯，甚至比我的职业生涯还要长得多。现在，他在 AI Agent 领域也开始大显身手。

我们请到了 Uncle Bob（Robert C. Martin），他正穿着浴袍，准备好开场了。Uncle Bob，你好。

Uncle Bob：你好，很高兴来到这里。

Matt Pocock：很高兴能邀请到你。对于那些不了解你“浴袍梗”的人，我们可能得解释一下。这是怎么回事？浴袍为什么成了你个人形象的一部分？

Uncle Bob：这件事大约发生在两年前，当时是早上六点，我穿着浴袍坐在前廊。

我开始思考 SQL 注入实在是一件糟糕透顶的事情。作为访问数据库的方式，让一种文本语言直接承担这个角色，本身在安全性上就很不合理。

当时我正穿着浴袍，越想越觉得不对劲，于是掏出手机发了一通牢骚。这就是后来所谓“晨间浴袍吐槽”的由来。没想到反响还不错，所以后来我又陆续做了几次。

Matt Pocock：那你今天也是带着这种心情来的吗？

Uncle Bob：我是个脾气有点古怪的老头。现在还是一大早，我连咖啡都还没喝，所以最好别惹我。

不过现在已经没事了。其实已经早上 10 点了，浴袍也脱了，咖啡也喝了，刚刚还喝完了一罐健怡可乐。现在状态非常好。

Matt Pocock：太好了。既然现在穿上了 Polo 衫，那 Uncle Bob 的故事是怎样的？我的观众大部分是开发者，但也有一些非开发者，我们该如何向他们介绍 Uncle Bob 这个“现象级

人物”？

Uncle Bob：“现象级”谈不上。我是一名程序员。我已经当了很久的程序员了，超过半个世纪。

我在 1964 年编写了我的第一个程序，当时我 12 岁。那个程序运行在我母亲为我 12 岁生日买的一台小型模拟电脑上，你需要把白色的小管子套在插桩上来编程。它本质上是一个 3 位的有限状态机，但在 12 岁的我看来，它简直让我着迷。

Matt Pocock：那你是如何从 12 岁走到 50 多年后现在的状态？

Uncle Bob：实际上，已经超过 50 年了。从那以后，我开始尽可能多地学习编程。父亲给我买了一本 Fortran 的书、一本 Cobol 的书，还有一本 PL/I 的书，我把这些书全都读完了。只是当时没有机器可以运行程序，所以我只能把程序写在纸上，然后在脑子里把它运行一遍。

16 岁时，我找到了一份可以写一些代码的工作，不过那只是一份临时工作。到了 18 岁，我找到了一份真正的程序员工作，从那以后，我就一直做程序员，一直到今天。

Matt Pocock：所以你已经在“战壕”里待了很久，而且你还写了一本非常重要的书。

Uncle Bob：是的，我写过几本，其中一本确实火了。《代码整洁之道》（Clean Code），这里是第二版。

“我不用亲自去看 AI 写的代码，也可以信任它们”

Matt Pocock：当我和人们聊起好书时，在软件工程领域，这本书是被引用次数最多的。它极具影响力。这也是我今天想和你聊聊的原因，因为你在前 AI 时代拥有巨大的影响力。

那么，Uncle Bob，现在 AI 已经成为现实，你的看法发生了什么变化？

Uncle Bob：去年 12 月左右，这件事确实让我吃了一惊。

圣诞节期间，我开始尝试使用 ChatGPT、Grok 之类的工具，最初其实没觉得有多惊艳。后来我逐渐意识到，这些东西可能比我想象的更有意思。

于是，我找了一个“智能体”（Agent）。我记得最早用的应该是当时还比较早期的 Grok。我让它帮我写一些代码，结果写得确实不怎么样，但至少，它真的把代码写出来了。

当时我正忙着做一个项目，就想，也许这东西真能帮上我的忙。于是我开始让它参与我的工作。不过，那个阶段我一直在给它“擦屁股”，因为它经常把事情搞得一团糟。它的速度确实很快，但总会留下一些隐患。

所以我当时的感觉很矛盾：**它很有意思，因为确实快；但也很让人沮丧，因为它反而让我变慢了。**

后来我开始想，既然它这么快，那它其实可以做一些我自己根本做不到的事情。

早在 2000 年代初，我就有过一些自己觉得很棒、但当时完全不现实的想法。其中一个叫 CRAP，这是一个缩写。它把代码测试覆盖率和每个函数的圈复杂度结合起来，再通过一个比较复杂的公式计算出一个分数，用来衡量一个函数到底有多糟糕。

2000 年代初的时候，我觉得这是个很棒的主意。我曾经在一个大型项目上跑过一次，确实找出了大量糟糕的函数。但问题是，当时我只能一个一个地去修复这些函数，还得重新编写测试，成本实在太高了。所以最后，我只能把这个想法搁置下来。

另一个让我很感兴趣的创新叫“变异测试”（Mutation Testing）。它的原理是让一个小程序自动修改你的源代码，比如把负号改成正号、把小于号改成大于号、把等号改成不等号。每修改一次，它就会运行一遍完整的测试套件，并且预期测试应该失败，因为代码已经被故意改坏了。如果测试没有失败，就说明出现了一个“存活的变异体”，需要把这个问题处理掉。

大约在 2000 年的时候，我也在那个项目上试过变异测试。当时我得让它跑一整晚，因为完整的测试套件每次要运行 4 分钟，而我需要重复运行几百次。它确实找出了不少“存活的变异体”，但同样不现实，我根本没办法把它纳入正常的构建流程。

到了去年 12 月或者今年 1 月，我突然想到：AI 的速度很快，而且它们根本不在乎工作有多枯燥。于是我让智能体去跑 CRAP，再让它负责重构和清理代码。看着它做这些事情真的很酷。

我还让它跑变异测试。以前可能要跑一整晚的任务，**现在 30 分钟就能完成**，而且它还能把所有测试漏洞补上。

我当时就想，这也许是清理代码残留垃圾的一个好办法。AI 写代码时确实会留下很多“碎屑”和“浮毛”，但它们自己也许就是清理这些东西的好工具。

于是我继续尝试，不断加入更多工具，再让这些智能体配合着工作。到现在，它们已经做得相当不错了。

所以我现在的原则是：**让智能体去干活，让它们运行这些工具。我正在努力达到一种状态——我甚至不需要亲自去看代码，也可以信任它们。**

当然，**我还是会通过其他方式验证代码质量，比如检查 CRAP 分数、抽查代码、运行其他测试。但总体来说，这就是我现在努力实现的目标。**

既然它们处理代码的速度比我快得多，而我处理代码很慢，那就让它们负责写代码，我来处理更高层面的事情，确保一切都在正常运转。目前来看，效果还相当不错。

AI 生成的“烂代码”该怎么办？

Matt Pocock：所以你的目标是让自己脱离具体的代码编写和人工评审，构建一个环绕代码的“脚手架”，给智能体穿上“紧身衣”，让它无法犯错。

这里有一个假设：你提到了 AI 写的代码会有“隐患”或“烂代码”。既然 AI 这么快，为什么我们还要在意烂代码？为什么不能直接顶着 Bug 往前冲，直到 Bug 被冲掉为止？

Uncle Bob：我很早就发现了一点：**如果我一直让 Agent 往下做，却不去清理它留下的“烂摊子”，它的速度反而会越来越慢。**

它会陷入一种困境：**改了一个地方，却无意中破坏了另一个地方；为了修复那个地方，又破坏了其他地方。最后就开始不停地绕圈子。**

我意识到，这些智能体虽然快，也确实很聪明，但它们和人类一样，也会受到烂代码的影响。也许它们能够容忍的程度和人类不同，但这个阈值依然存在。

代码烂到一定程度，Agent 就处理不了了。它们会开始原地打转，把原本的“烂摊子”越搞越大。我甚至遇到过一个智能体直接对我说：“我处理不了了。”

Matt Pocock：所以你的应对方案是什么？如何清理这些“垃圾代码”？大多数人遇到这种情况会选择给智能体增加指令，比如在 claude.md 或 agents.md 里堆满规则。每当看到坏代码，就增加一条指令。而你选择的是一种确定性的自动化检查机制。你为什么不选择这种“引导”的方式？

Uncle Bob：最开始，我确实是这么做的。最早给智能体写的 Prompt，基本都是这种思路：这是测试驱动开发（TDD）的方法，这是整洁代码的要求，你的代码应该是什么样，你应该遵循哪些规则。

最后，你可能会得到一份长达 10 页的文档，专门告诉它什么才是“好代码”。

但我很快发现，**这些模型对待规则的态度，特别像《加勒比海盗》里的“海盗法典”——听起来是规则，但对它们来说，更像是一堆“建议”。**

Matt Pocock：没错，这正是我脑子里想到的那个比喻。

Uncle Bob：它们确实会把这些规则“软化”。这背后其实有技术原因，我后来专门研究了一下模型为什么会这样，发现这与一个叫“中段丢失”（Lost in the Middle）的现象有关。

随着模型的上下文窗口越来越大，放在最前面和最后面的内容，往往比夹在中间的内容更容易被模型关注。比如你写了一份很长的 Prompt，开头的前三句话它可能记得很清楚，并把它们当作高优先级指令；但到了第 50 句、第 80 句，那些内容就可能被丢到上下文中间的某个角落。

当模型面对海量上下文时，中间的信息更容易被忽略，甚至像“消失”了一样。但确定性的工具不会这样。它们不会因为规则写在第 50 行还是第 80 行，就突然把规则当成“建议”。

我认为使用智能体的关键——虽然这确实很难做到——**就是将初始提示词精简到绝对最小值，以便尽可能多地将其保留在优先级区域，然后再在此之后使用确定性工具。**

Matt Pocock：完全同意。我把它称为上下文窗口的“聪明区”和“愚蠢区”。这不是我的说法，而是 Dex Horvath 提出来的。我觉得这是一个非常贴切的比喻。

在上下文窗口的前段，比如前 15 万个 Token，模型通常表现得相当聪明。但随着上下文不断变长，Transformer 中的注意力机制会越来越吃力，信息也会逐渐被稀释。

这就像一个越来越拥挤的房间，每个 Token 都在大声说话，而且房间里的人越来越多，最后你很难从一片噪音中分辨出真正重要的信号。这个比喻对我来说非常有共鸣。

听起来，你某种程度上是在放弃“引导”（Steering），转而重新采用一些传统的自动化检查手段。毕竟，这些检查不会像引导指令那样占用上下文窗口，所以你可以不断往上叠加。

如果使用的是一门具有强类型和完善测试体系的语言，这种模式在你看来会是什么样？自动化检查会不会也存在“太多了”的情况？

Uncle Bob：这正是我目前正在研究的问题。显然，**自动化检查一定存在一个“过多”的临界点。**最终，如果它们让智能体的速度慢到还不如人类，那你就输了。但只要它的生产力仍然高于人类，你就依然处于领先地位。

根据我目前的观察，我大概可以把这种生产力优势维持在 2 到 4 倍左右。

当然，使用确定性工具会明显拖慢智能体，因为你实际上是把它放进了一个循环里。现在很多人都在讨论这种循环：你要求智能体不断修改代码，直到工具最终告诉它“OK”为止。

于是 Agent 就会在那里不停循环——修改这个、修改那个，增加更多测试，降低圈复杂度，拆分函数……它需要花很长时间，才能达到预先设定的合规标准。

所以，本质上，你是在牺牲一部分生产力，换取更高的代码质量。

虽然目前我还没有找到这种方式的极限在哪里，但我正在尝试让多个智能体彼此协作、相互交接：一个负责写代码，下一个负责审查，再下一个负责测试和强化。

这样做确实会带来巨大的通信开销，但即便如此，它们的整体速度仍然比人类快得多。

多智能体协同，根治上下文失效难题

Matt Pocock：让我们聊聊多智能体系统（Multi-agent Systems），这非常令我着迷。我一直对那些宣称“我已经裁掉了所有员工，现在有 100 个 Agent，每个 Agent 都有不同的角色，彼此之间互相交流，甚至还有自己的邮箱账号”的人持怀疑态度。

但对于把“实现”和“审查”分离开来，我愿意破例支持。这种做法的好处，并不在于拥有两个非常专业的智能体，而在于它有点类似“红-绿-重构（Red-Green-Refactor）”的方法。

实现者只需要让测试通过，不需要把代码写得多漂亮。然后审查者再介入。它不需要重新探索整个问题，因为它已经拿到了实现者的差异对比，也明确知道自己需要审查什么。这时候，你就可以给它加入更多的引导指令。

我很想听听你对实现者、审查者，以及你提到的“强化智能体”怎么看。

Uncle Bob：采用多 Agent 有两个优势：

- 第一，你可以并行运行多个 Agent。比如同时运行三个编码 Agent，我的笔记本电脑甚至还能支持更多。
- 第二，当你把 Agent 的任务聚焦到单一任务时，就可以更好地控制上下文窗口。“中间丢失”的问题会减轻，你还可以在顶部堆叠更多规则，让 Agent 执行得更好。

你甚至可以设置这样一套机制：让 Agent “出生、完成任务，然后消失”。这样，下一个 Agent 进来时，面对的就是一个干净的上下文窗口。缺点是启动时间会更长，一个 Agent 可能需要 10 到 15 秒才能启动并理解当前上下文。

我倾向于尽可能把任务拆得足够细、足够聚焦。

我会先运行一个“规格定义器”，它的任务是把人类编写的文档转换成 Gherkin 语言和 QA 流程。QA 流程本质上是一套系统测试，要求从人类用户的视角出发，在 UI 层面操作系统，并证明系统能够正常工作。

随后，这两份文档会交给“编码器”。编码器负责编写单元测试、实现故事逻辑，并让 Gherkin 测试跑通。

完成后，再交给“清理器”。它负责进行复杂度分析和常规代码审查，把实现者留下的“烂摊子”清理干净。

接着交给“强化器”。它负责运行变异测试。这个 Agent 非常无情，会通过修改等号、小于号之类的细节，反复验证测试是否真的有效，并确保达到 100% 的测试覆盖率。这个过程需要很长时间。

最后交给“QA Agent”。它会把书面的 QA 文档转换成可执行脚本，用脚本直接操作系统，并给出确定性的测试结果。

如果能够通过这一整套流程，最终得到的程序质量会非常高。

这种方式我自己尝试得非常成功。给单个 Agent 一个任务，它可能 5 分钟就能完成，但结果是否可靠就很难说了。而采用这套流程，可能需要一个小时，但这依然是划算的，因为如果让人类完成同样的工作，可能需要半天。

Matt Pocock：没错，这本质上是前期投入生产力，以换取后期的收益，是对代码库的长期投资。除了操作上下文窗口，你还在操纵一个会话的“轨迹”。如果你引导智能体在同一个窗口内持续做某事，它就会沿着这个轨迹走下去。

Uncle Bob：确实。这些模型存在一个众所周知的现象。即使你不是程序员，比如你正在和模型愉快地聊怎么冲咖啡，这时旁边路过一个人，开始聊他正在看的肥皂剧，而这些信息也进入了上下文窗口。从那一刻起，你再提到咖啡，模型就可能莫名其妙地把咖啡和肥皂剧联系起来。

模型很难真正区分这两类信息。所以，你提出的“轨迹”这个概念很好。只要能够让模型始终保持正确的方向，并尽可能保证上下文窗口中的内容一致，就能避免产生这些奇怪的幻觉，至少也能减少对齐偏差。

Matt Pocock：既然提到了自动化检查，我想问问你在前期的规划中，是如何考虑代码库的内部结构的？拥有好的测试套件固然重要，但如果 API 设计得很烂，或者模块划分得非常糟糕，这会如何影响这些自动化检查？

Uncle Bob：在过去大约一个月之前，我一直是手动完成这部分工作的。我会先让 Agent 构建一个东西，然后通过不断提问来“审问”它们：这里的结构是什么？这个模块和那个模块是怎么关联的？模块到底是什么？等我得到那些令人恐惧的答案后，我就会亲自设计模块结构，然后告诉 Agent：“这才是模块应该划分的方式，以及它们之间应该如何通信。”接着，再给它们一个实现计划去执行。

这个过程非常辛苦，所以我让 Agent 为我构建了一个“架构查看器”。它可以在屏幕上弹出一个类似 UML 的图表，展示整个系统的模块结构和依赖关系。我可以点击某个模块查看它内部的子模块，甚至直接查看对应的代码。这个工具对我来说非常有用。

此外，我还构建了另一个确定性工具，让我可以定义哪些模块应该依赖哪些模块，以及哪些模块绝对不能互相依赖。这最终形成了一份 Agent 无法违反的规范文件。如果它们违反了这些规范，就必须通过反转依赖、提取接口等方式进行修复。

我现在正在尝试把整个规划过程自动化，但目前还没有取得太大进展。

Matt Pocock：我也处于完全相同的处境。通过设计良好的模块，你能获得巨大的杠杆作用。你能解释一下这种杠杆作用体现在哪里吗？

Uncle Bob：这和“烂代码 vs 好代码”的论点是一样的。

任何划分良好、接口严谨的东西，都是人类可以理解的，因为我们的大脑擅长碎片化处理。

模型和 Agent 也是如此。如果它们能聚焦于一个模块，且该模块的“轨迹”非常清晰，模型不会被该模块内部的话题所混淆。我同时使用了“模型”和“模块”这两个词，我想表达清楚。但这非常重要，这和我之前提到的“咖啡与肥皂剧”的论点是一致的。

如果你在一个模块里塞进了天底下所有的东西，可怜的 Agent 就会纳闷：“我在这里到底在干什么？我该如何在这里开展工作？”如果你能进行良好的模块化处理，它的运行效果会非常好，就像人类一样。

Matt Pocock：你可能也能从测试套件中获得更高的价值。这里我想再请教一个相关的问题：我是你作品的超级粉丝，同时也是 John Ousterhout 的拥趸。他提出的“深层模块”（Deep Modules）概念让我非常着迷。

模块大致有两种形式：一种是“浅层模块”，接口很复杂，但内部实现很少；另一种是“深层模块”，接口很简单，却在内部隐藏了大量信息。我发现，这种设计对 Agent 来说非常理想，因为它们只需要读取接口，而不必理解具体的实现。

你认同这种观点吗？你自己在处理问题时，也会采用这种方式吗？

Uncle Bob：绝对认同。模型非常关注接口的名称和结构。这意味着它们不必去阅读下层代码，这既是一种风险，也是一种优势。只要代码保持连贯一致，就不会有太大问题。

它们也会关注测试，通过阅读测试来理解系统的功能。因此，任何有助于优化代码结构的做法，都会帮助模型更好地理解代码。

顺便说一句，《代码整洁之道》（Clean Code）的附录里，记录了我和 John Ousterhout 之间的一场长篇辩论，那真的非常有意思。我不确定他是否也乐在其中，但我自己确实玩得很开心。

AI 时代，传统编程规范需要重新定义

Matt Pocock：我看过你们在 YouTube 上的那段完整讨论，非常精彩，这也是我想邀请你来的原因。那么，在你的书里，有没有什么内容是你现在想修改或更新的？尤其是像“保持函数短小”这样的建议。

我们刚才聊到，很多东西其实并没有改变。比如“Gauntlet”（严酷考验/自动化评估环）这种模式一直都很好，只是过去我们没有足够的劳动力去推动它。

但在这些原则中，有没有什么是我们现在需要摒弃的？或者说，有没有哪些东西需要换一种方式来理解或调整？我知道这个问题可能有点让你为难，但你现在有答案吗？

Uncle Bob：首先是关于“阈值”的问题。在我看来，Agent 能够处理的复杂度，与人类开发者并不一样。它们的短期记忆比人类强得多，而且非常精准。

所以，我会相应调整 CRAP 分值（CRAP score），放宽对函数规模的限制。对于人类开发者，我会要求 CRAP 分值控制在 4 以下；但对于智能代理，我目前把这个阈值放到了 6，甚至在考虑提高到 8。

我一直在寻找那个最合适的阈值，但这并不容易。

Matt Pocock：在代码行数上，4 到 8 或 4 到 12 的分值意味着什么？是 20 行还是 100 行的函数？

Uncle Bob：这实际上取决于圈复杂度，也就是一个函数内部存在多少条不同的执行路径。

如果测试覆盖率达到 100%，那么 CRAP 分值为 6，意味着这个函数有 6 条执行路径，而且每一条都经过了测试。这其实就是 CRAP 分值的核心目标：确保测试覆盖率足够高，同时限制圈复杂度。

我曾经和 Agent 就这个问题争论过很多次。当然，你不能完全相信 Agent 在辩论中的结论，但我还是会和它们讨论。它们似乎认为，6 是一个相当不错的标准。

此外，还有另一个因素。在我的书里，我谈到了很多“纪律规范”，比如测试驱动开发（TDD）。我曾经是 TDD 的坚定拥护者，但那其实是一种“人类的纪律”，是根据人类的思维方式演化出来的。

所以，我不会，也不打算把这种纪律强加给 Agent。我认为，**强迫 Agent 写一行测试，再写一行生产代码，然后再写下一行测试，这没有什么意义。**对人类来说，这样做的收益非常大；但对于 Agent，我更愿意允许它们采用 John Ousterhout 提倡的方式：先写出一个函数，再为这个函数编写测试。

事实上，即使我要求 Agent 严格按照 TDD 的纪律来做，它们最终也总会回到“先写代码、再写测试”的模式。

所以我的结论是：**把人类的纪律强加给智能 Agent，可能是一个错误。**我们不需要强加纪律，但需要坚持“人类的价值观”，只是具体的阈值可能需要根据智能代理的特点进行调整。

Matt Pocock：这确实和短期记忆有关。对于短期记忆容量有限的人类来说，TDD（测试驱动开发）非常有效：你只需要记住足够的信息，先写出一个测试，再记住如何让测试通过，然后就可以停下来休息一下，比如去喝杯咖啡。

既然我们聊到了“把事情做正确”以及架构设计，那么在真正把任务交给智能代理之前，你通常会做多少规划？在让 Agent 开始执行任务、测试和验证之前，你会做多少准备？

因为如果一开始交给智能代理的任务就是错的，或者需求本身没有定义清楚，那么后面无论做多少工作，都是在浪费时间。

Uncle Bob：现在最大的诱惑，是过度编写规格说明：让开发者不断完善需求和计划去写规格说明，再把这些东西一次性交给 Agent。这是一个源自 70 年代的古老诱惑，后来瀑布流开发模式就是这种思路的典型代表，而敏捷开发的兴起，很大程度上就是对这种做法的反思和回击。

过于沉重的前期规划，往往会把事情搞得一团糟，因为最终做出来的东西，几乎不可能和最初设想的一模一样。

现在面对智能代理，人们同样容易掉进这个陷阱。我这周就尝试过这种方式，结果一次次证明是一场灾难。作为人类，你很快就会发现，Agent 根本无法完全按照你制定的计划执行。因为你不可能提前考虑到所有细节，而 Agent 也没有你那么强的判断能力。于是，它们很容易朝着错误的方向一路跑偏。你只能叫停，然后回滚、重新制定计划，再从头开始。

所以我已经放弃这种做法了。我们不如回到敏捷的方式：先让 Agent 完成一两个具体任务，然后看看整体架构有没有问题；如果需要，我就手动介入，做一些调整，再让它继续完成几个任务。

我们可能永远都无法完全摆脱最后这一步“人工组织”的工作。虽然我一直在想办法解决这个问题，但目前还不确定这是否真的可行。

Matt Pocock：我把这看作一个劳动力分配的问题。

过去，构建一个东西可能需要几周甚至几个月。现在，开发过程大幅加快了，但前期规划和后期评审所需要的时间并没有相应缩短。我们希望开发者能够更快地迭代，但“把东西做错了”这件事依然可能发生。

我发现，现在很多人开始做一种“计划极大化”：他们拿着一份规格说明，让七个不同的 Agent 分别跑一遍，试图不断完善计划，然后才真正开始执行。

但这样听起来好像并不是什么好主意，对吧？

Uncle Bob：Agent 非常喜欢写计划。天哪，它们简直爱死写计划了。它们会不断修饰计划，把它写得越来越华丽、完美，细节也越来越丰富，但到了真正执行的最后阶段，往往还是会崩盘。

现在整个行业都能看到这种趋势，也就是所谓的“规格驱动开发”（Spec-Driven Development，SDD）。**但我的直觉是，这条路行不通。**

我们应该重新审视敏捷开发的核心思想：先做一点，获取反馈，再调整和重新组织，然后继续做一点。

我以前做敏捷开发演讲时，经常讲一个例子：假设改建一栋房子的成本只需要 1 美元——包括打地基、修屋顶，以及之后所有的修改，每次都只需要 1 美元。那么你会怎么盖这栋房子？

你会不会先花几千美元请建筑师设计一套完美的方案，然后再花 1 美元交给承包商，一次性把房子盖出来？

还是直接走到承包商面前说：“我想把地基打在这里，做成这个形状。噢，不对，这样不好，改一下。厨房放这边，客厅放那边。等等，还是把它们换个位置吧。”

显然后一种方式更合理。**而现在，软件修改的成本已经大幅下降，几乎接近于零。在这种情况下，我们为什么还要花大量时间和精力做昂贵的前期规划？**

为什么不直接不断尝试、调整和修改，直到它看起来正确为止？

Matt Pocock：我非常认同。我个人对“规格驱动开发”这个标签有很大意见。到底什么是SDD？提示词工程（Prompt Engineering）算吗？如果你像以前一样对同事说：“修一下页头的加载问题”，这是规格驱动开发吗？这算是一个规格说明吗？你会把这些说明持久化在代码库里吗？

Uncle Bob：不，我不会。**规格说明是转瞬即逝的，它们会消失，也会频繁变化。**规格说明并不等同于源代码。过去人们常说，“人类编写了源代码，所以代码就是最终的规格说明”，但现在这个说法已经站不住脚了。

源代码依然存在，只是已经不再由人类编写。很多人因此感到失落，认为必须有某种由人类定义的东西放在最前面。但归根结底，即使是 Agent 产出的东西，最初也是由人类驱动的。

我现在的做法是，不再编写一份用来定义“我想要什么”或者“我拥有什么”的规格说明。我直接看最终结果，而那个结果本身就是规格说明。

我有很多工具，比如针对 Clojure、Java 和 Go 的 CRAP 工具、变异测试工具，以及 Agent Harness。我会告诉别人：不要直接下载我的工具，因为那是我为自己写的。你应该让自己的 Agent 去研究这些工具，然后让 Agent 根据你的需求量身定制一个。

我认为，这才是更好的方式：**真正定义事物的本质，并根据具体需求进行定制。**

Matt Pocock：关于 Agents，我一直觉得很奇怪的一点是：如果你发给它们一些东西，它们真的会去读。这和人类非常不同。

如果你给某人发一份详尽的技术规范，你可能只能得到 20% 的阅读率——而且 20% 可能还说多了，也许只有 5%。但如果你把规范交给智能体，它们大概率真的会读完。

Uncle Bob：反过来也一样，Agent 写出来的东西，人类反而不读了。

给新生代开发者的忠告：AI 再强，编程底层基础永不过时

Matt Pocock：确实如此。它们希望我们把它们写的所有东西都读完，但我们根本不会读。挺有意思的，这完全是一种不对等的关系。

从长远来看，我特别想知道，既然我们已经如此习惯于和智能体沟通，人类之间的关系会发生怎样的变化。我自己已经习惯了和智能体聊天，就像在和朋友聊天一样。我真的很好奇，也很着迷于看到，随着时间推移，生活会如何模仿艺术。

最后我想问你一个问题，而且这是个相当重磅的问题。我想再次引用一下 John Ousterhout 的观点，因为他对不同类型的编程有一个很好的定义：战术编程和战略编程。

战术编程就像地面作战的士官，是那个真正身处战场、负责具体战斗的人；战略编程则更像将军，负责从更高层面指挥整场战争。我想我们都同意，这是理解不同类型编程的一个很好的框架。

而现在的问题是，**智能体非常擅长战术，却非常不擅长战略。**

那么，对于那些刚刚入行的人来说，既然 AI 已经吞掉了大量战术性的工作，他们该如何学习战略编程呢？这可能是很多人都想听到你回答的问题，Bob。他们想让你把你的大脑“借”给他们，让他们理解为什么变异测试如此重要，或者应该如何设计和理解这些模块。某种意义上，这正是你现在正在做的事情——把自己的经验和思考分享给更多人。

Uncle Bob：我经常被问到这个问题。我没有一个完美的答案，因为我确实不知道，但我是这么想的。

首先，**程序员学习编程的方式（无论是在大学还是其他地方）都应该是写代码。你应该先写上一年代码（具体多久我也说不好），这样你才能真正知道 Agent 究竟在处理什么。**

下一步应该是，当你入职一家大量使用 Agent 的公司时，作为一个刚完成培训的年轻人，你应该被当作一个 Agent 来对待。那位手下运行着一堆智能体、自己负责战略决策的首席工程师，应该把你视作一个智能体。他应该给你分配和智能体一样的任务，让你接受和智能体一样的确定性工具约束。你应该在这种状态下待上几个月，虽然产出会非常低，但能学到非常多的东西。等你通过了这种严酷的考验，也许你才会被信任去亲自运行一个智能体。**你不能完全丢掉代码。**

十年前，我经常告诉人们：如果你从来没有写过汇编语言，那就应该花一个周末写写汇编，这样你才能知道后台到底发生了什么。如果你整天只写 Java，那你就是生活在一个幻觉世界里，那里仍然存在很多你不理解的“魔法”。花一个周末写写汇编，你最终就会理解这一切到底是怎么回事。

我认为这一点在今天依然成立。在这条学习路上，你必须从最基础的东西——二进制——开始，一路经过汇编语言、像 C 这样的基础编程语言、像 Python 这样的高级语言，然后再进入处理智能体级别的工作，学习使用确定性工具。最后，你才能在监督下，真正开始战略性地运行智能体。

Matt Pocock：但这很难。智能体就像是代码之上的一个抽象层。你提到的那些抽象层，比如模块查看器，就很有意思。这确实是一种很好的学习方式：在代码之上建立这些抽象，这样当你真正深入研究时，反而能获得更深的理解。

但如果一个人只做战术性的工作，作为一家公司，你可能会看着他想：“我们为什么要雇这个人？我手下有 Uncle Bob 的五人‘加固小组’——也就是五个智能体——成本只有他的一小部分，却能做得更好。”

所以我一直在想，人们到底该怎么获得这些能力。因为传统上，战略编程的反馈循环非常长。一个人如果干了六个月就辞职了，他可能永远都学不会战略编程，因为他犯下的错误，可能要到九个月之后才会暴露出来。他根本看不到自己的错误。

但有了智能体，情况就不一样了。因为一切都被大大加速，你实际上可以更早获得关于错误的反馈。

那么，你是怎么知道自己的智能体在犯错的？去年 12 月，当你看着它们写出来的东西，觉得“这写得像狗屎一样”的时候，你是怎么判断出来的？

Uncle Bob：早期的时候，我只是看着代码，然后发现里面那些“不好的代码”。但那其实不是最重要的部分。

更重要的是下一步：我会看着它们瞎忙。我能看出 Agent 什么时候在挣扎，因为我自己也经历过同样的挣扎。**问题在于，一个刚入行的人，可能根本识别不出这种挣扎。**

我是通过艰苦的实践学会这一点的。其实，关于这个话题，那些老书里有非常丰富的内容——只是因为年代久远，已经没人读了。但这些书真的非常棒。你可以去读 Tom DeMarco 或 Ed Yourdon 的著作。

Matt Pocock：还有《程序员修炼之道》（The Pragmatic Programmer），所有那些经典。

Uncle Bob：没错，《程序员修炼之道》就是其中一本。其实还有很多这样的老书，它们都非常出色。如果你在年轻的时候认真读过这些书，就会逐渐对更高层面的战略博弈形成感觉。

当然，你需要过滤掉其中一些已经过时的内容，因为很多书写于 20 世纪 70、80 年代。但有意思的是，很多重要的经验和教训，恰恰就是在那个时候总结出来的。

所以，这是我最初会推荐的学习方式：先通过阅读这些书建立基本的理解，然后再通过亲身实践真正掌握它。我也正因为如此，认为他们应该先“扮演”几个月的智能体，亲自经历一下智能体所处的状态，这样才能真正理解那是什么感觉。

Matt Pocock：成为智能体。让智能体把任务委派给你。你成了智能体的子智能体。我喜欢这个观点。听起来，软件基础仍然至关重要。为什么会这样？对于那些说基础不重要的人，你会说什么？

Uncle Bob：软件基础之所以重要，是因为它们一直都很重要。我想这句话是迪杰斯特拉（Dijkstra）说的：软件是人类迄今为止尝试过的最复杂的东西，比我们做过的任何其他事情都要复杂。

所以，基础其实是我们组织这种复杂性的一种方式，让复杂的东西变得可以理解——不仅是让人类能够理解，也让我们的模型能够理解。毕竟，模型终究也是模仿人类建立起来的。

基础之所以在今天依然适用，就是因为这是我们组织复杂性、理解复杂性的方式。**现在有些人认为软件基础已经不重要了。他们会吃到苦头，而且不会等太久。**虽然可能会比我想象的更久一点，因为智能体确实很厉害。但我已经见过它们撞墙了，我知道那堵墙就在那里，所以我不想再撞一次。

这其实是一个非常有意思的演变过程。你刚才提到了抽象层。我们现在已经站在编译器之上了。**过去，我们的抽象层是编译器；在那之前，是汇编语言；再往前，是二进制。现在，这个抽象层又向上提升到了模型。**

而每一次抽象层向上提升，处在更低层级的人都会抱怨。他们会说：“这会毁了一切。我们甚至都没工作可做了。编程变得这么简单，五岁小孩都能写代码了。”

每一次都是同样的故事。现在，我们又进入了下一个阶段，下面的人开始说：“这会毁了一切。”

不，它不会。

同样的规则依然适用，所有这些基础之所以存在，都是出于同样的原因。你今天扔掉的那些规则，一年之后，很可能还是会从地上把它们捡起来，掸掉上面的灰尘，然后重新想起：为什么当初需要这些东西。

GitHub宕机7小时47分钟，究竟发生了什么？官方发布复盘报告

杀进“羊圈”的AI Coding？从大厂辞职11年后，他又把代码“捡”了回来：一个人用AI干了3个月，仅花5万元给1万只羊写了套系统

曾经号称比Python快9万倍，Mojo如今彻底开源了！

📢最后，说一件事

2026 奇点智能技术大会与C++及系统软件技术大会终于要和大家见面了。

11 月 20-21 日·北京，奇点智能研究院联合 CSDN，把两场技术大会放在了同一个时空里：

- 奇点智能技术大会（始于 2016）——聊大模型、AI Native、企业级 AI 落地、多模态与世界模型；
- C++ 及系统软件技术大会（始于 2005）——聊现代 C++ 演进、AI 算力与推理优化、高性能低时延系统。

为什么要放在一起？因为我们越来越相信——上层 AI 应用的爆发，离不开底层系统软件的支撑；而底层技术的演进方向，也正在被 AI 重新定义。

这次大会汇聚 70+ 位技术专家、18 个主题、1000+ 同行到场。

如果你也在这些方向上做研究、做产品、做工程，别错过。