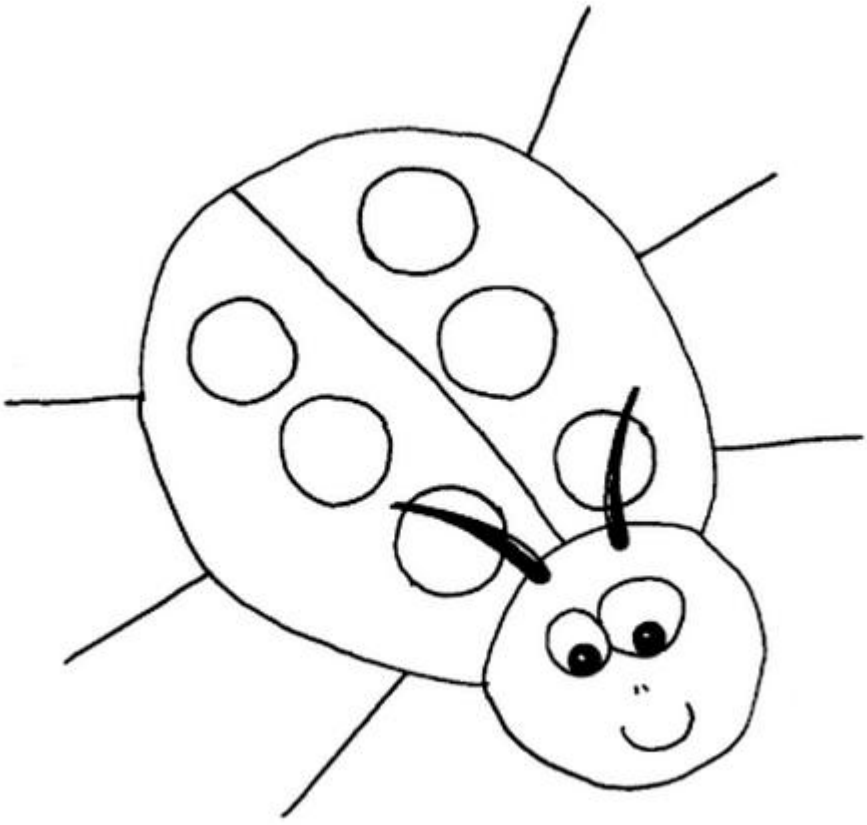
**I Am A Bug! 中文**

**作者/来源：http://www.amibug.com/iamabug/p01.html**

翻译： 妙音鸟 （jhhuawei@gmail.com）

我是一个 bug

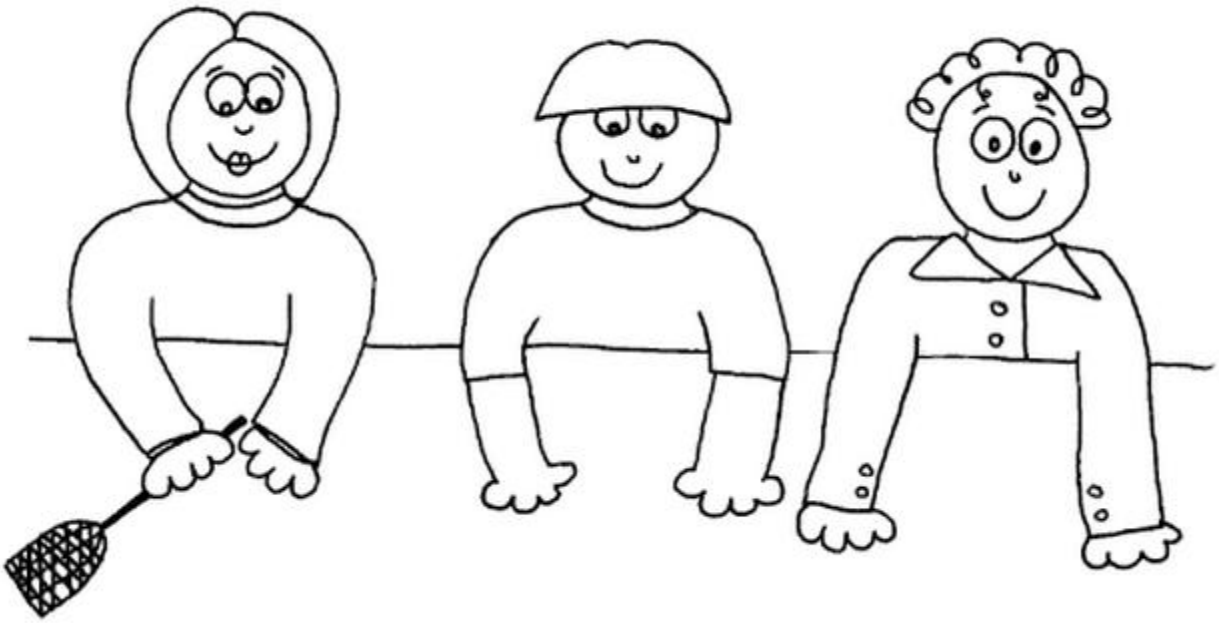


**为了使我们的工作更有趣，当我们的软件中有一个问题的时候，我们叫它 bug。**

我比较支持在软件工程中给予 BUG 以更广的定义 ,我喜欢把它定义为和项目相关所有事物。大部分人定义 bug 使用特定定义，比如 bug 是缺陷，问题，软件开发交付件（需求，设计，代码等等）中被植入的异常，或是被弄错的或者被忽略的，以软件故障为呈现。但是不一定会因为缺陷或者错误而导致

的，有时候它只是意外行为，这种狭义的定义被认为是一个 bug，因为重要的关注点是没有被定义进去的。

**这些都是关注我（bug）的人**



**Audrey 是我们的 SQA 主管--他的团队负责找到 bug！**

**Yves-Alain 是我们的开发主管，他保证软件中不引入 bug，但如果有 bug 引入了，他的团队处理他们。**

**Oliver 是我们的产品经理 ,他和客户一起确定我们程序需要做的事情。Oliver 帮助我们确定哪些 bug 需要处理，哪些不需要。**

在这种做决策的团队中三个不同的视角是非常有必要的 ,这些角色是 SQA 主管，开发主管，产品经理。

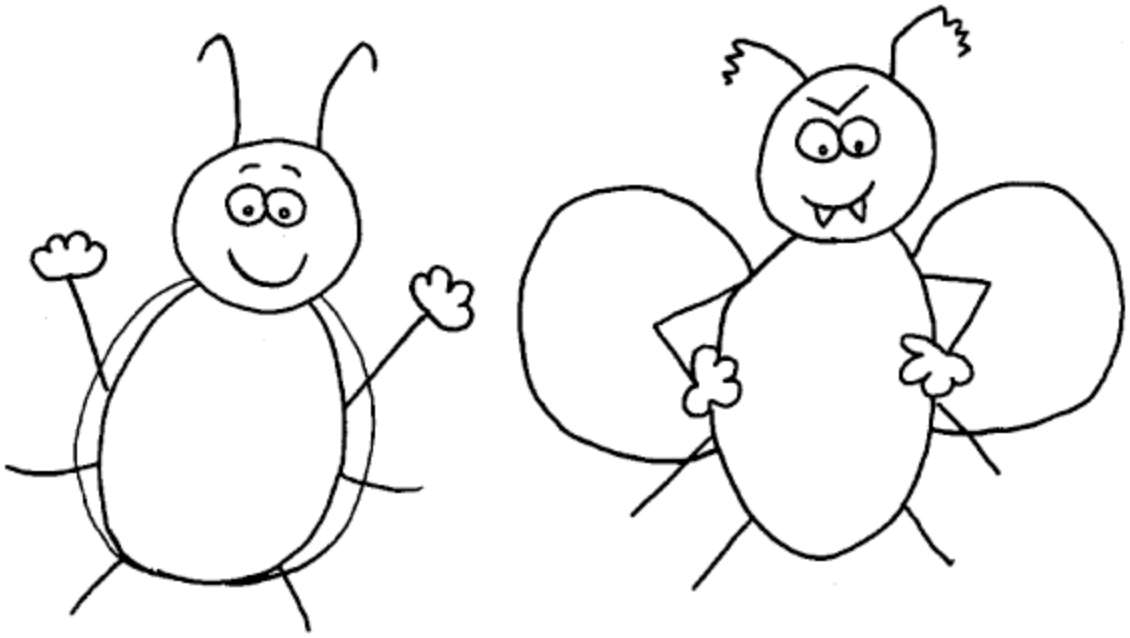
在这个决策会议中，决定 bug 是否需要修复，决定保留这个 bug 不修复（遗留问题），这种决定通常是修复这个 bug 容易对软件引发破坏，决定把这些 bug 遗留在产品中，通常是客户容易忽视和不会碰到的情况。一个决策团队应该是小型的，有效的团队，它的人员组成都是有能力做决策的人员。一个差的 bug 决策团队充满了没有能力做决策的人，这些人需要叫上更多的人来一起做决定。尽管一个好的决策团队优势也需要另外的输入，但是他们很少需要其他人介入一起才能够做决策。

SQA 主管是 QA 的代表，他提供 bug 的客观信息，同时，开发主管和产品经理讨论这些信息并作出决定，SQA 主管不应该去插手决定 bug 是否需要修改，他必须是中立的。

开发主管是开发团队的代表，他的工作是评估修改 bug 的技术风险。

产品经理代表客户，他的工作是评估修改 BUG 对客户的风险。如果开发主管和产品经理经常吵哪些 bug 应该修改，哪些不用修改，那么这个团队是低效率的。偶尔的争吵是正常的，正常情况下都应该很快达成一致。在偶尔的没能达成一致情况下，我偏重于考虑产品经理的意见，因为他是代表客户的，他才知道客户的需求。

**BUG 没有好坏**



**当我们发现 bug，我们不问这个 bug 是好是坏。我们确定它是否重要，我们考虑他对会造成多大影响。**

Bug 的重要性（优先级）和对系统造成的影响（严重性）是独立的，优先级和重要性是分别定义的。遗留 bug 的优先级和重要性评估通常是软件能否发布的条件。因此人们把他们放到了重要的位置，以至于不能够让实习生来填这些信息，我经常遇到这样的状况，测试人员提出优先级和重要性，但这不是最终解决办法。每个 bug 适用于 4 象限中的一种。

Q1 紧急，重要

Q2 紧急，不重要

Q3 不紧急，重要

Q4 不紧急，不重要

对于 BUG 数量四象限分布的正确理解有利于测试人员有效评估测试活动的有效性。如果发现 Q3，Q4 的 BUG 分布远超 Q1，2，那么测试团队应该转移战场测试其他模块或者抓换一下测试方法。

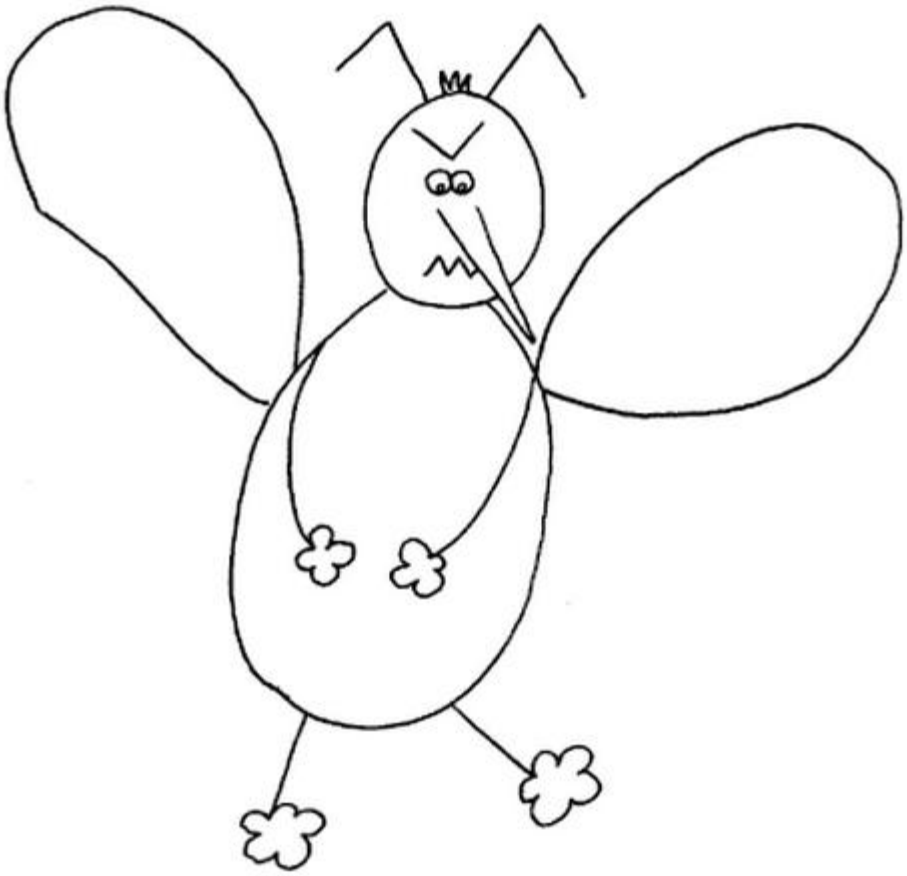
**但是一些 bug 是重要的**

**我们直接关注重要的 bug**

我们怎么来报告一个重要的 bug，答案是他系了领带。优先级的划分取决于人员，资源，时间等等的约束。一个一般严重程度的 bug 在项目中貌似是致命的，对于项目干系人来说它是高优先级的但是不严重。bug 的优先级随着商业环境或者技术的环境变化而变化。在上周觉得这个 bug 很重要但是在这周就变得不重要了。如果客户已经决定不再需要某个特性，这个特性相关的 bug 可能之前很重要但这之后就变得没有意义了。bug 的优先级和严重性要考虑商业环境。他们必须知道利益干系人是谁，客户是谁，技术限制是什么，

设计驱动是什么等等。这时候他们需要用现有的知识来调整他们对新 bug 的看法和刷新对老 bug 的看法。

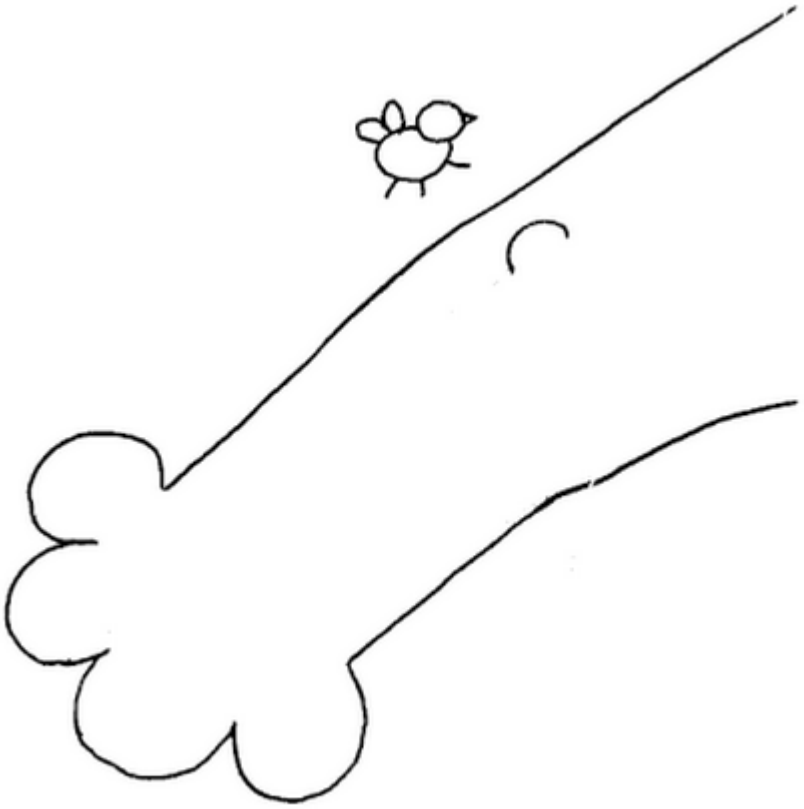
**一些 bug 容易对系统造成很大影响**



**我们经常关注这些危险的 bug**

这些危险的 bug 是严重等级高的 bug,这些 bug 会使程序崩溃，更严重的是会破坏数据。就像优先级一样，严重等级随着商业环境而改变。

**被一只蚊子咬也许没什么伤害**



**单一的小 bug 看起来没什么危险**

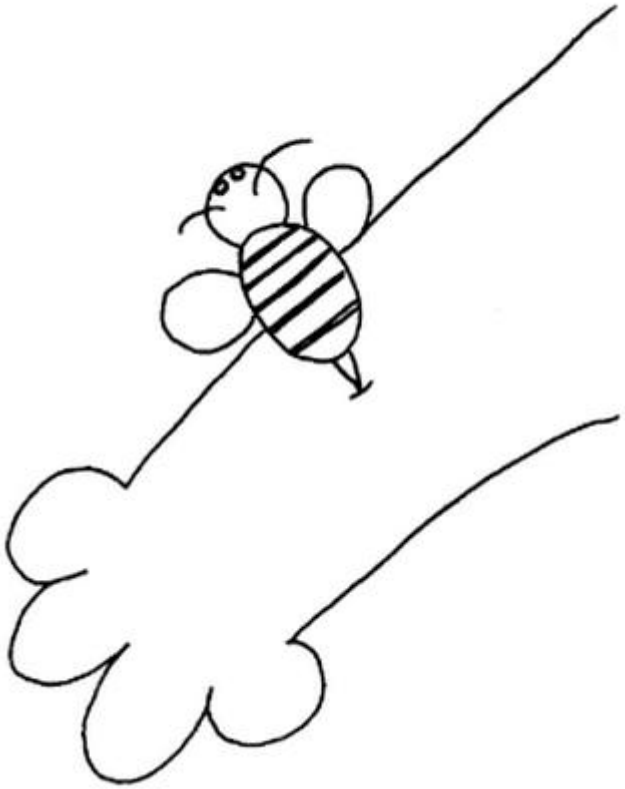
**但是成群的蚊子就很可怕了**

**成百上千的小 bug 同时出现就很可怕**

大量 bug 会激怒用户甚至使程序不可用，尽管单个 bug 来说并不严重。

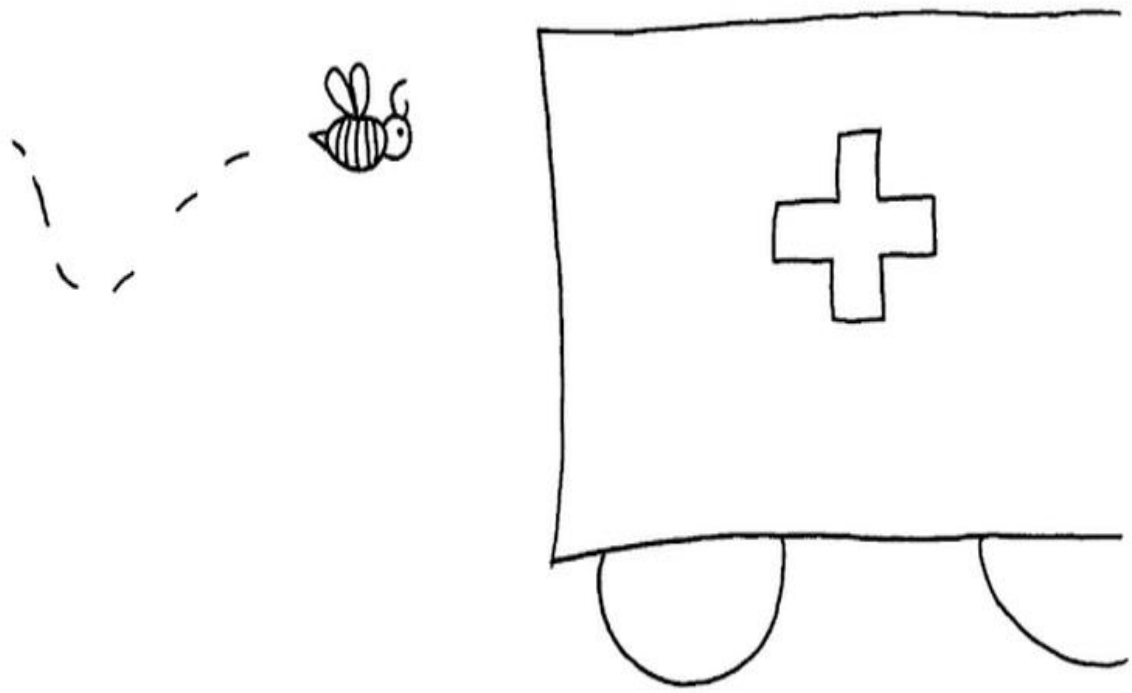
严重级别低的 bug 如果影响了大量用户就会引发大量客服投诉。

**被一个蜜蜂扎了只是一点点疼**



**同样的 bug 可以在不同计算机程序中找到。在一个程序中这个 bug 没什么破坏性。在帮助系统中语法错误不会影响使用甚至不会让人觉察。**

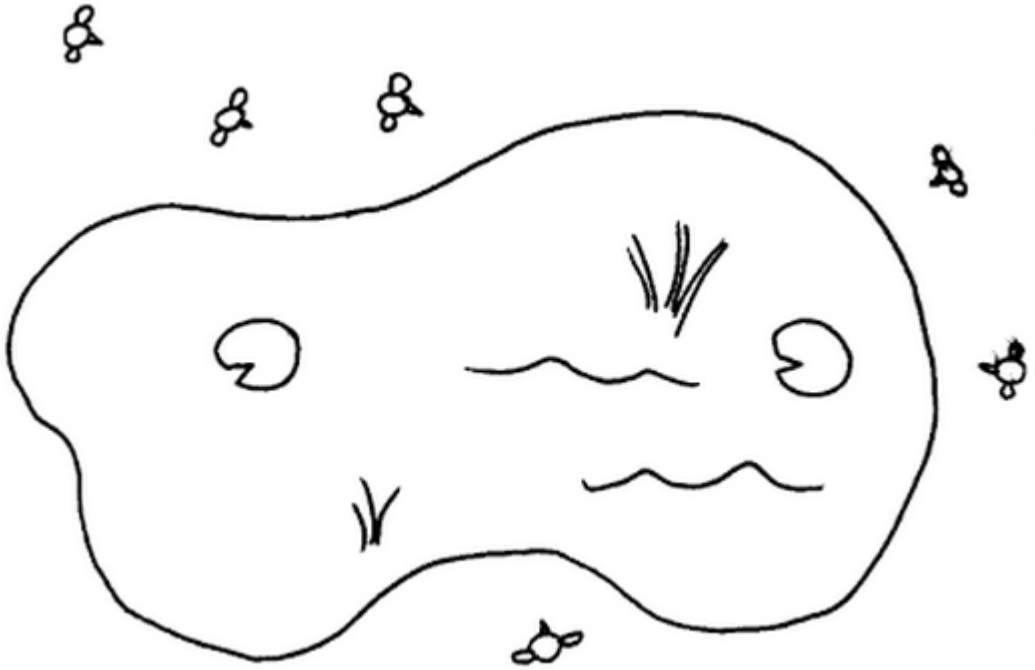
**但如果你对蜜蜂过敏，那么它将会杀死你**



**但同样的 Bug 在不同程序中就是致命的。**

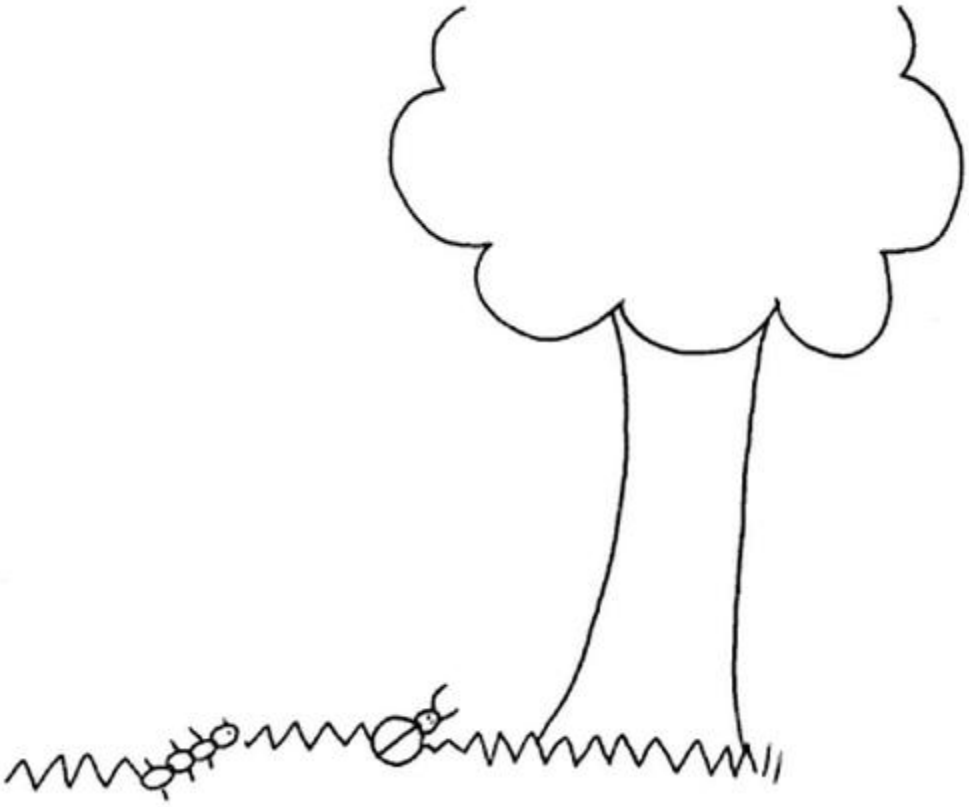
在医疗软件中，一个语法错误比一个系统死机更具有灾难性的影响。如果医疗软件死机了，医生最多不用这个软件来诊断了，没人会受伤害。但是如果是一个语法错误，比如过去式和现在式混淆了，意味着影响了患者的就诊历史记录读取，这可是关乎生死的事情。

**Bug 喜欢聚集在潮湿肮脏的环境中**



**当我们急匆匆的开始够条件新的软件时，我们的办公室和程序就会陷入混乱。我们知道这会带来很多 bug。糟糕的软件工程实践会陷入图中所示的困境，bug 满地。**

**Bug 也存在于家附近，院子附近。**



**当我们愿意花些时间做一些清洁，我们的办公室和程序就会更加干净和整洁。**

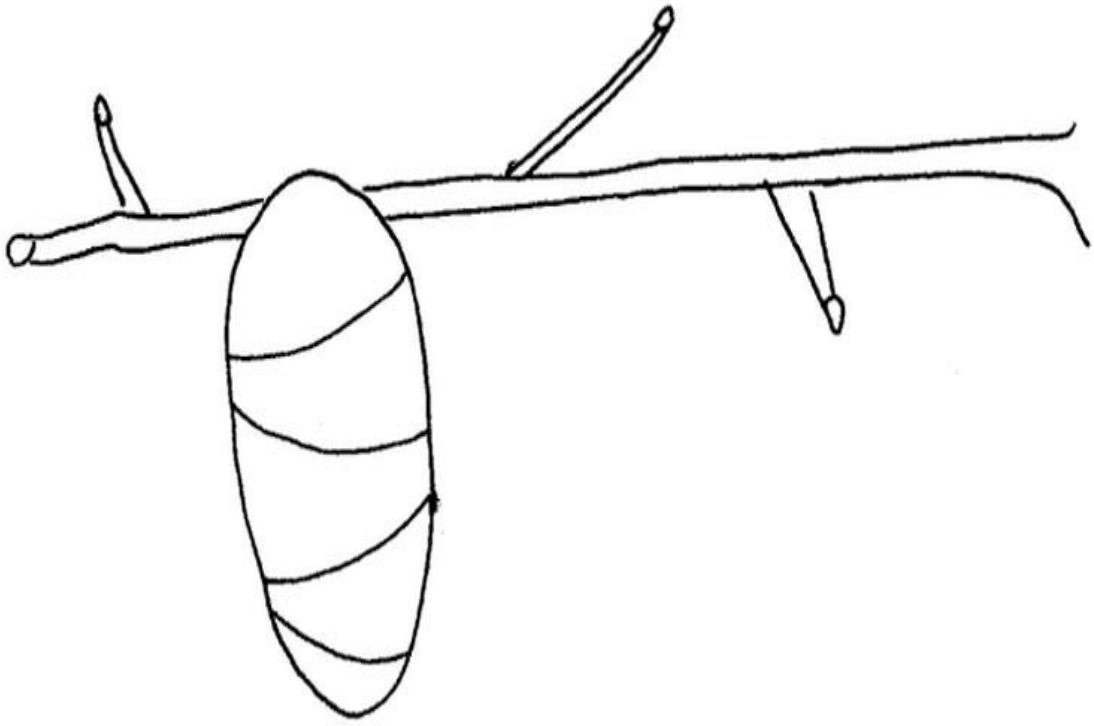
**也许还会有一些友好的 bug。**

修复 bug 是一个业务上的决定，有时候修复 bug 不见得会让你收益。

友好的 bug 是那种不会影响到用户使用的 bug。只是这些 bug 和需求不太相符合，但它并不坏。有时候客户非常熟悉的特性也许会是一个 bug 因为它不符合需求，其他一些只是程序中的细小差别并不影响客户使用，我们认为是友好的 bug。

AVT-710 病人详情记录含有很多的 bug，但是人们无法忽视它们。就算是包含有这些 bug，他也是一个零缺陷报告的可靠的产品。

**有些 bug 已经沉睡了很久**



**有时候我们是在程序用了很久之后才发现 bug 的。我们称这些 bug 是沉睡的 bug。**

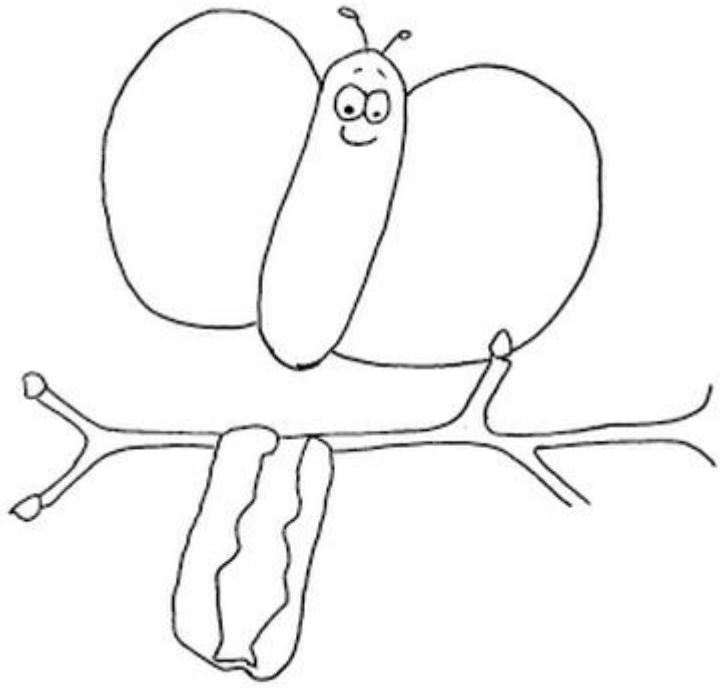
在一定的时间内软件内部含有 bug，但这些 bug 并没有暴露，当有一天软件上下文改变时 bug 就暴露了。

经过经典的例子是 Visio97， 他是在硬盘空间比当前需要的空间小的时候发生的，当在新的电脑（新电脑有很大的硬盘空间）安装 Visio97 时，安装程序会提示空间不足。为什么？因为 Visio97 采用 32 位的算法来计算剩余空间，在产品发布的时候，一点都没有问题。但是当新的有更大的硬盘空间出现的时候（此时已经硬盘空间已经不是 32 位算法），就出现问题了，数位被截断了，计算方法错误。这个 bug 的沉睡的，因为它是因为在技术限制被突破后暴露的问题。

有时候软件的使用用途也会变，比如，Windows 的打印机（LaserJet2 惠普打印机 ）驱动是设计成打印表格的，驱动中的代码并不能处理彩色照片，这些驱动在那个时代是好用的，但是现在已经不可用了。

又或者软件的用户群是会变换的，比如今天软件是专家在用，没准有一天就是一般的民众在使用，这两个群体的需求和兴趣点可以不一样的哦，因此当专家用的时候并不是 bug 而在普通大众来说可能就是一个严重 bug。

**令人惊奇的是---我们幸运的发现一个蝴蝶**



**有时沉睡的 bug 被唤醒来了。**

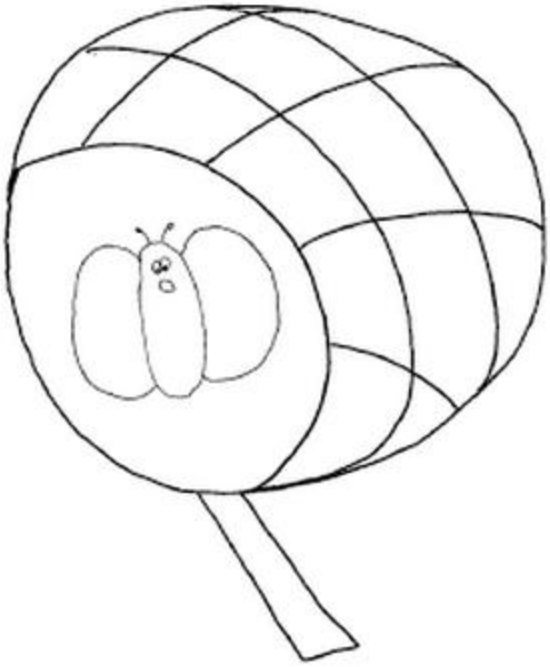
**如果我们幸运，这个 bug 不会对系统有任何影响。**

**有时当沉睡的 bug 被唤醒时，我们叫他“特性”。**

当沉睡的 bug 被唤醒时，他们成为系统的特性，不管你喜欢不喜欢。

当沉睡的 bug 被唤醒时，它会有向后兼容性的问题，特别是 bug 本身被依赖时。一个经典的例子是微软 word 的页白计算 ,在老版本中是计算错误的。当微软已经修复了这个 bug 时，所有老版本的文件页边都会有些许的偏差。

**我们的网很容易捕获这些蝴蝶。**



**当我们测试一个电脑程序时我们制定一个测试计划。**

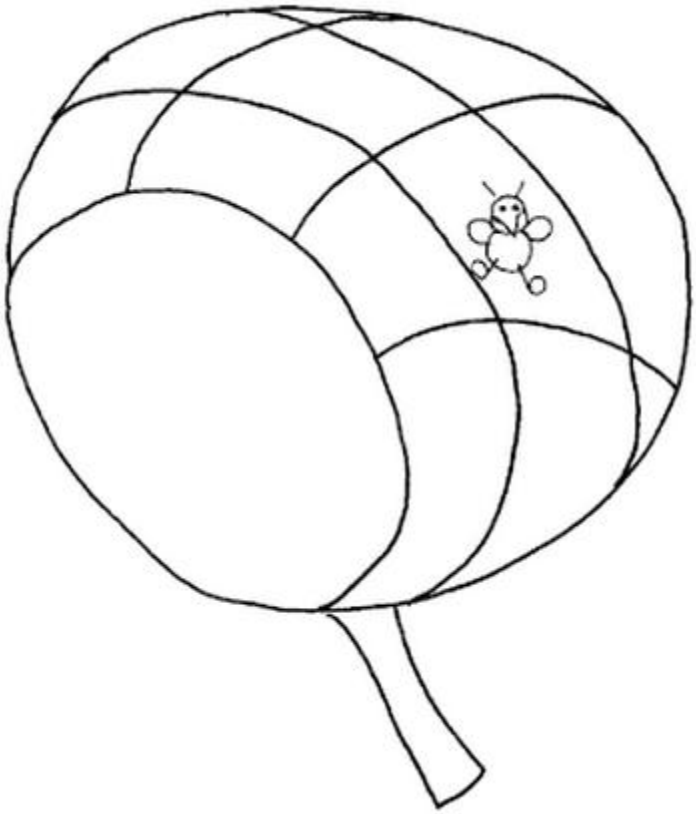
**测试计划就像捕获 bug 的网。**

**一些测试计划只能捕获明显的 bug。**

测试计划聚焦不同粒度的 bug，取决于测试时间和预算，有时候我们希望发现尽可能多的风险尽管不是很深入，而不是深入的测试每个风险。网越细，测试的深度越深。

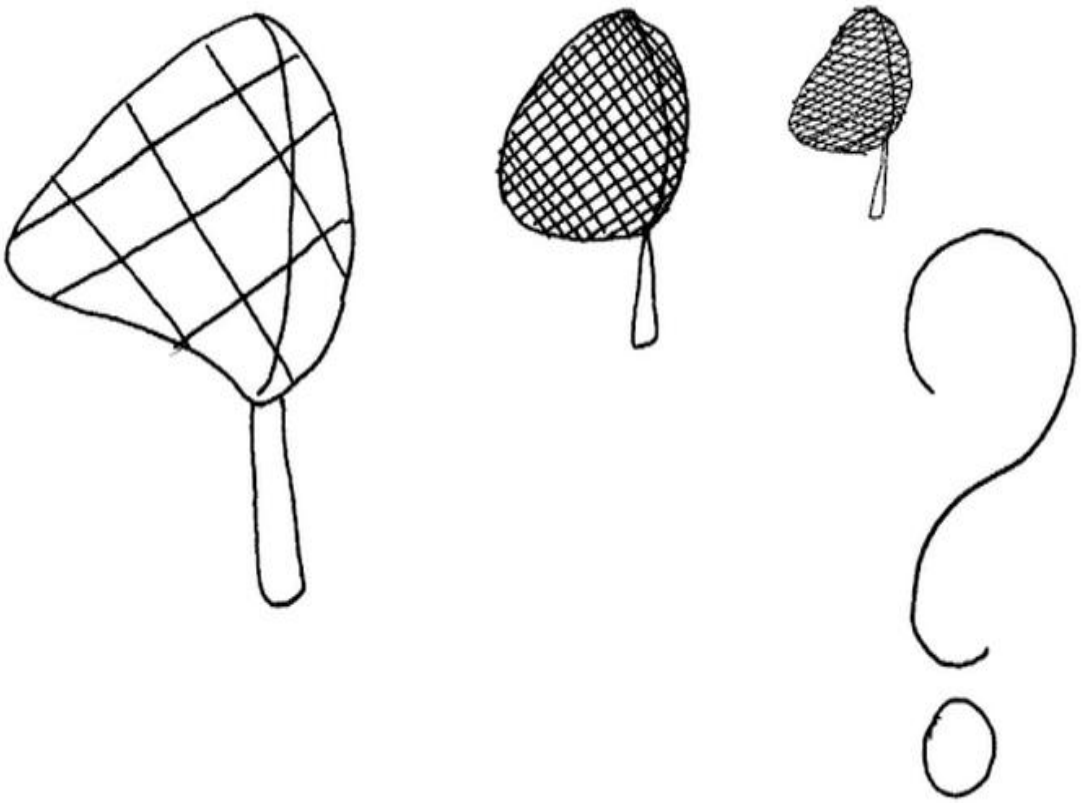
有些测试套叫做 FAST，全称是 Functional Acceptance Simple Tests。这个测试只测试系统处理正常输入数据的能力，让我们对系统对正常输入的处理有信心，但是它不保证系统能够处理其他情况。

**但是这并不是我们的老朋友蚊子**



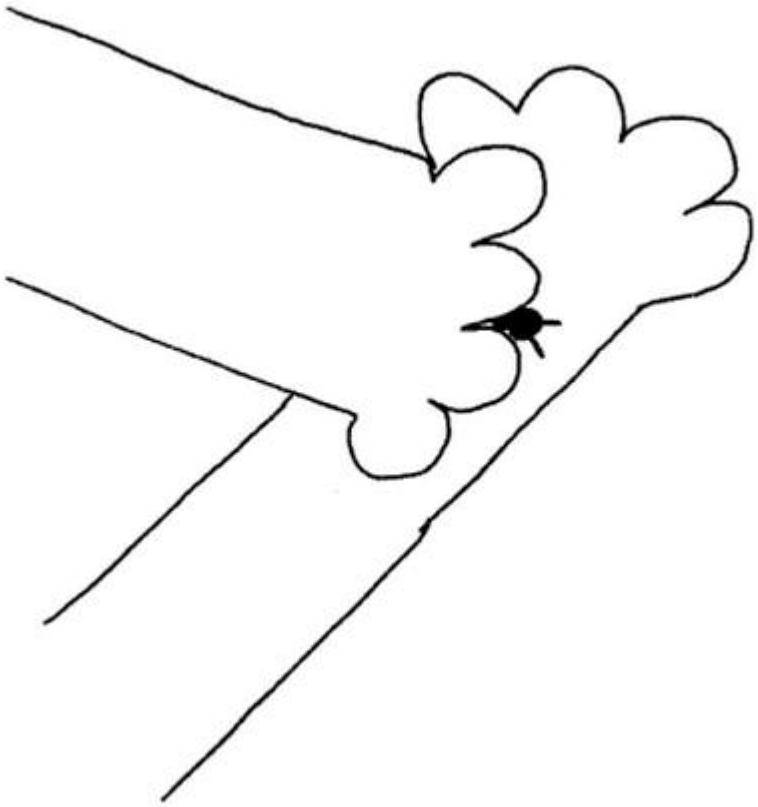
**一些测试计划 ---“网”非常好，能够捕获很小的蚊子（bug）。**

**我们应该用什么样的网呢？**



有时我们用一个测试计划来捕获大 bug，然后，如果我们还有时间，我们使用更细的网来捕获小 bug。

当我们被蚊子咬疼的时候，我们会拍打它，抓它



**当 bug 能否以简单直接的方式处理的时候，我们感觉是小小的。**

**抓一下能够减少疼痛但是也许药物可以更好解决问题。**


**bug 造成的小小影响通常可以通过改变人们的使用方法来解决。**

**bug 能够被修复，避免和忽略，如果 bug 不能被修复，应该引导用户改变使用方法来避免 bug 的发生。**

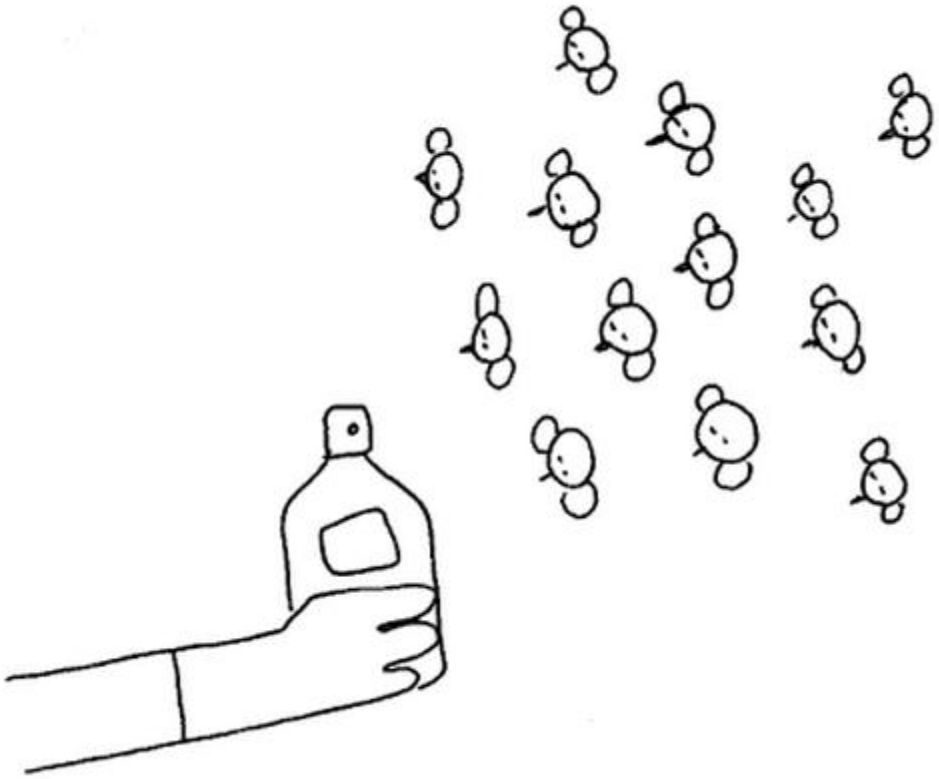
**驱蚊水能够使蚊子远离我们**

**有时我们找到了很好地办法来避免一类的 bug。**

**有很多专用的技术可以发现特种类型的 bug。**

**一种好的办法是自我数据严重机制，对于特定数据输入，理解软件后能够定义出测试 oracle。这是一种非常好的发现数据完整性 bug 的办法。**

**但是如果蚊子是成群结队的，那么驱蚊水也无济于事。**

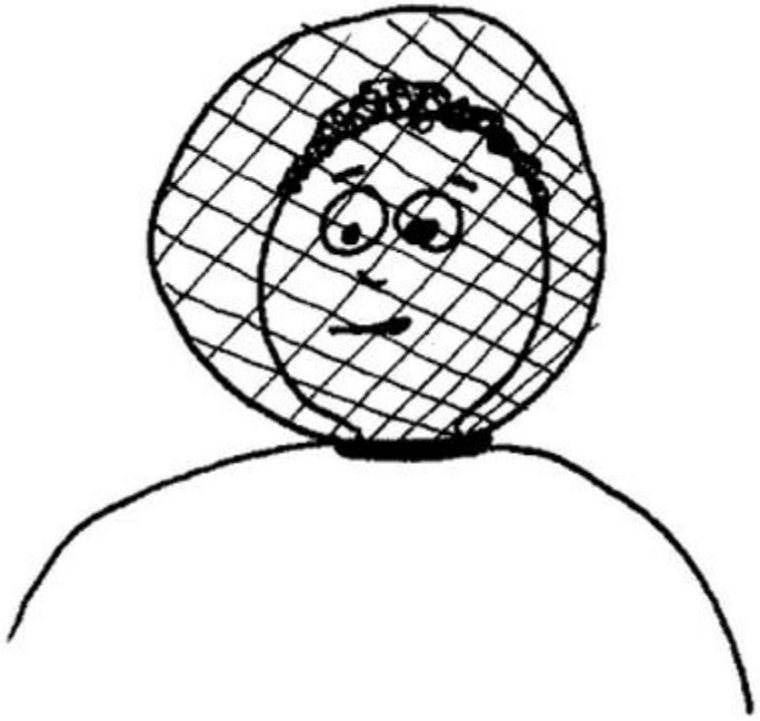


**然后我们发现这些特定类型的 bug 有很多很多兄弟姐妹。**

找到缓冲区溢出 bug 的技术是很好的一种找 bug 方式，但是他们不能找到计算错误类型的 bug。bug 的类型可远远不止缓冲区溢出一种类型。

同样的，通过自验数据完整性的校验技术，在有很多界面 bug 或者业务逻辑 bug 阻碍了数据达到数据处理模块时，不是最有效率和最有效果的。

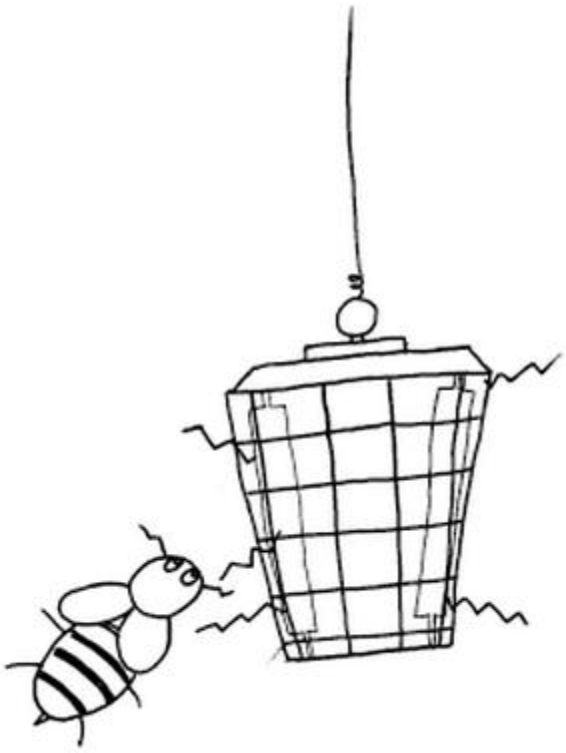
**网可以让蚊子隔离在网外。**



**我们知道很多在最开始的时候就阻止 bug 进入到我们程序的办法。**

所谓的办法包含软件工程中的三个重要工作流：配置管理，需求管理，测试管理。他们同时包含了开发规则，静态分析，代码检视，新代码的交叉检视，单元测试，好的开发实践，好的架构和一大堆其他的。

**一些人使用昂贵的微波灭虫器。**



**很久以前有个骗子销售员卖给我们一个新鲜刺激的玩具 ,叫做“bug zapper”。**

我们深信用了它所有 bug 都被消灭干净了。

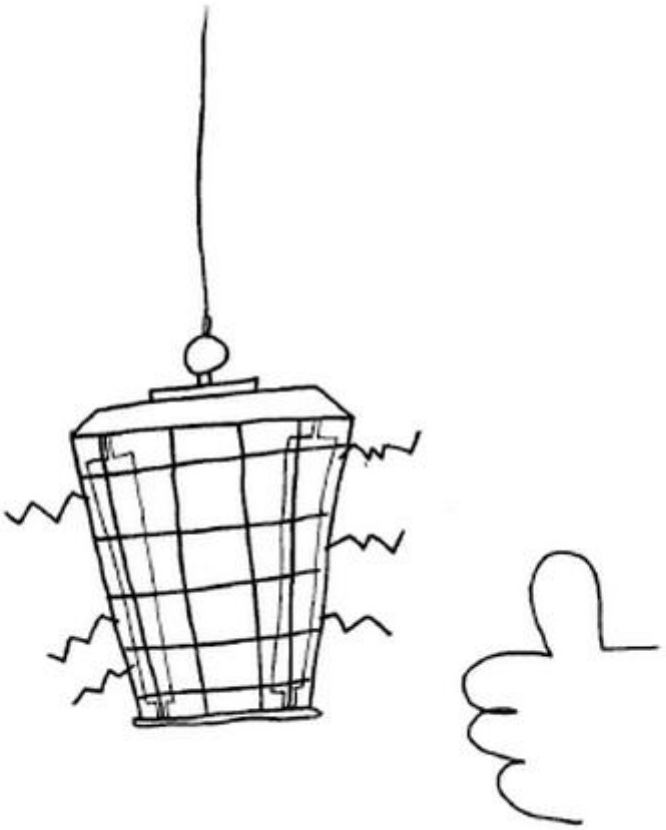
工具提供商的销售人员有时会误导我们，让我们忽视了成本和收益以及它的易用性。

比如：他们会声称测试自动化工具会捕获动作和回放这些动作，并能够解决你的所有问题。

他们会说 ,用这个工具吧 ,他们能够发现程序的所有问题。最初这会打动你，但是这些类型的工具要求大量的编程并且不能解决任何问题，甚至带来很大的维护成本，简直是一个噩梦。

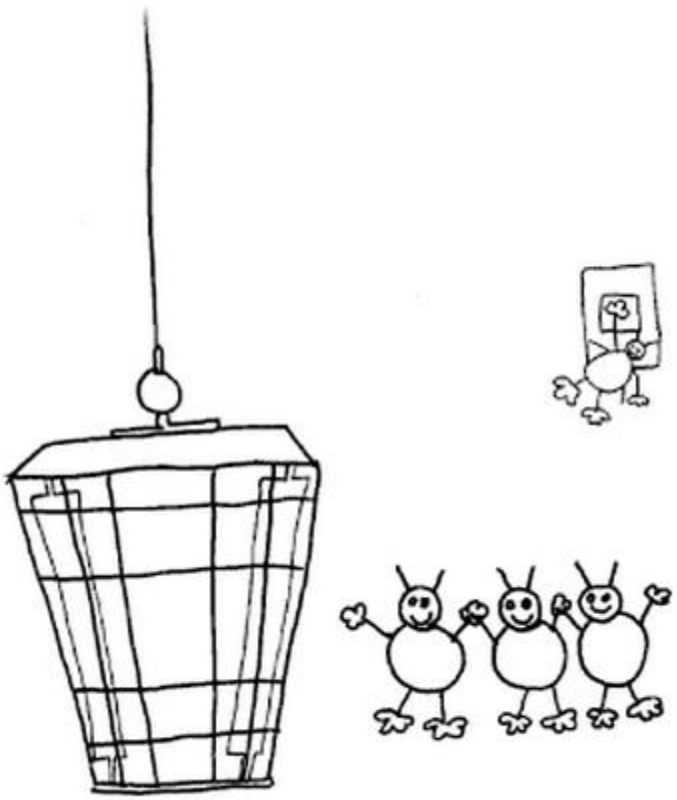
工具提供商还会声称你需要一个需求跟踪工具，它将减少成本，但是工具本身就需要钱购买。减少成本的应该是建立起需求管理的规则。

**微波灭虫器看起来很酷**



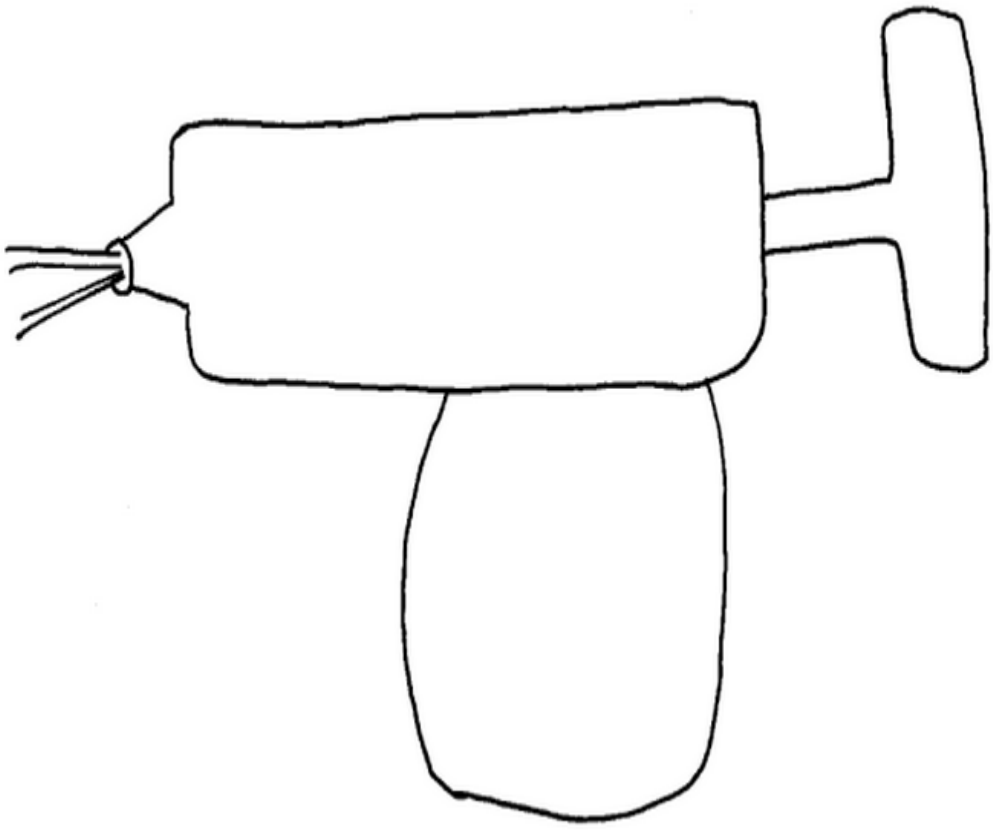
**最开始“bug zapper”貌似能够处理一些重要的 bug。**

**但是实际上并不能确保赶走蚊子。**



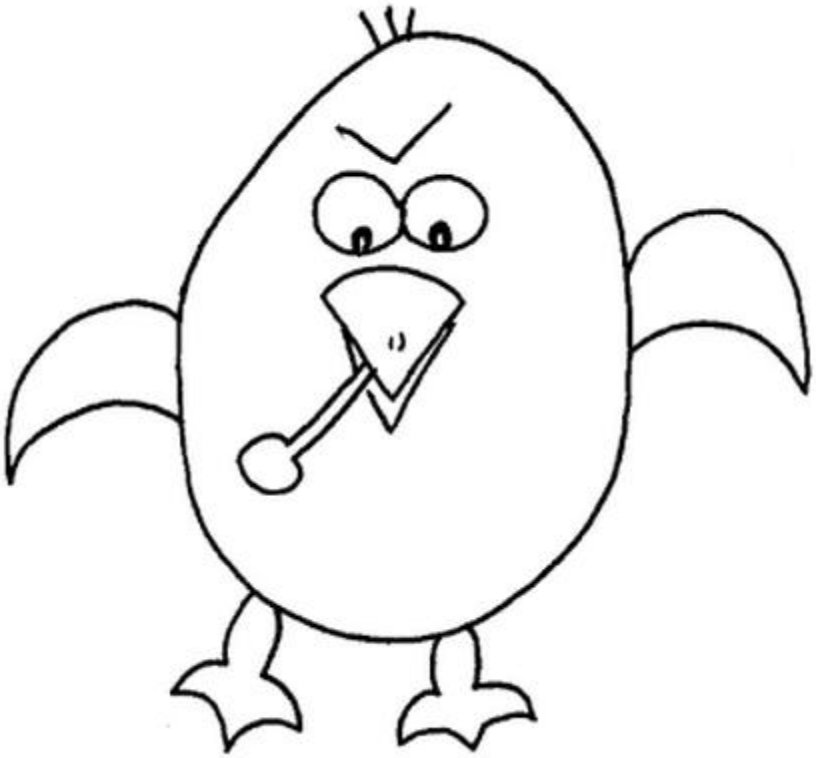
**但是 bug 很快就有免疫力了，他们避开了 bug zapper，我们连买 bug zapper 的钱收不回来。**

**喷雾杀虫剂也许是个解决办法。**



**当我们修改 bug 的时候一定要小心。**

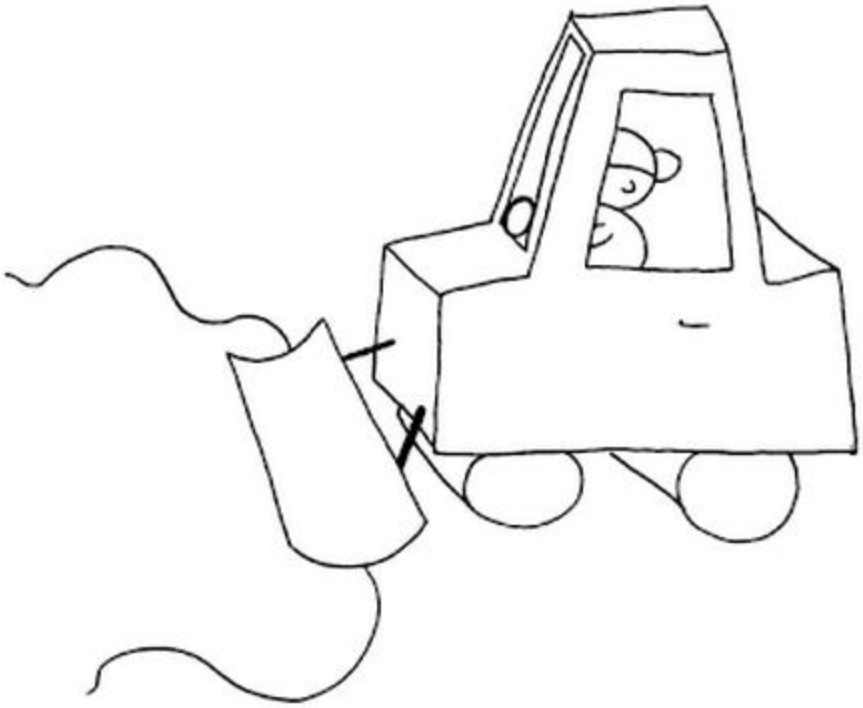
**但是这样可能不仅仅的蚊子受到伤害了**



**但我们修改 bug 的时候我们也许会引入其他 bug。**

有时候解决办法比问题还糟糕，修改一个 bug 也许会引入其他 bug，导致错误的系统行为。又或者 bug 在功能层面完美的解决掉了，但是像性能，可靠性，可用性，可维护性等等受到了影响。

**我们通过设计摆脱困境**



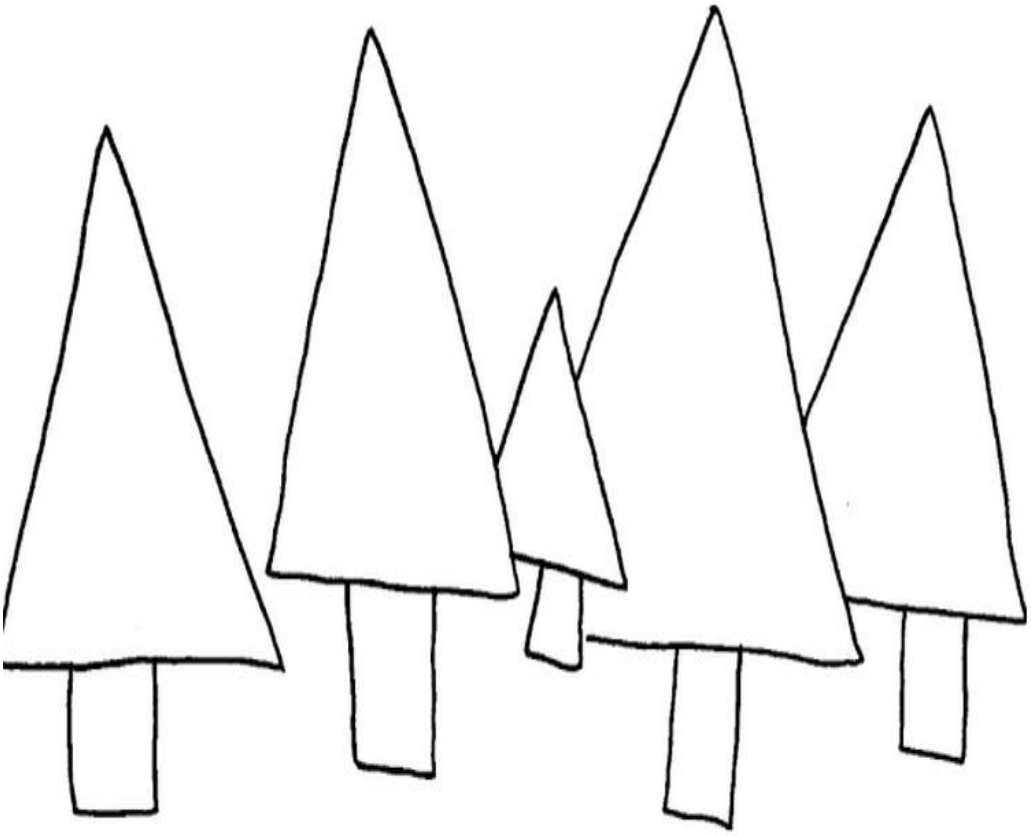
**但我们开始新的项目，我们深入思考如何避免引入 bug。**

**我们能够一开始就在一个干净整洁的办公室，开始新的想法吗？**

软件工程的三个基本流程非常重要：需求管理，配置管理，测试管理。

如果这些被深入理解并遵循执行了 那么因为流程混乱导致的 bug 将会很少，尽管因为项目风险造成的缺陷还是有，但是因为糟糕的项目管理造成的 bug 将会很少。

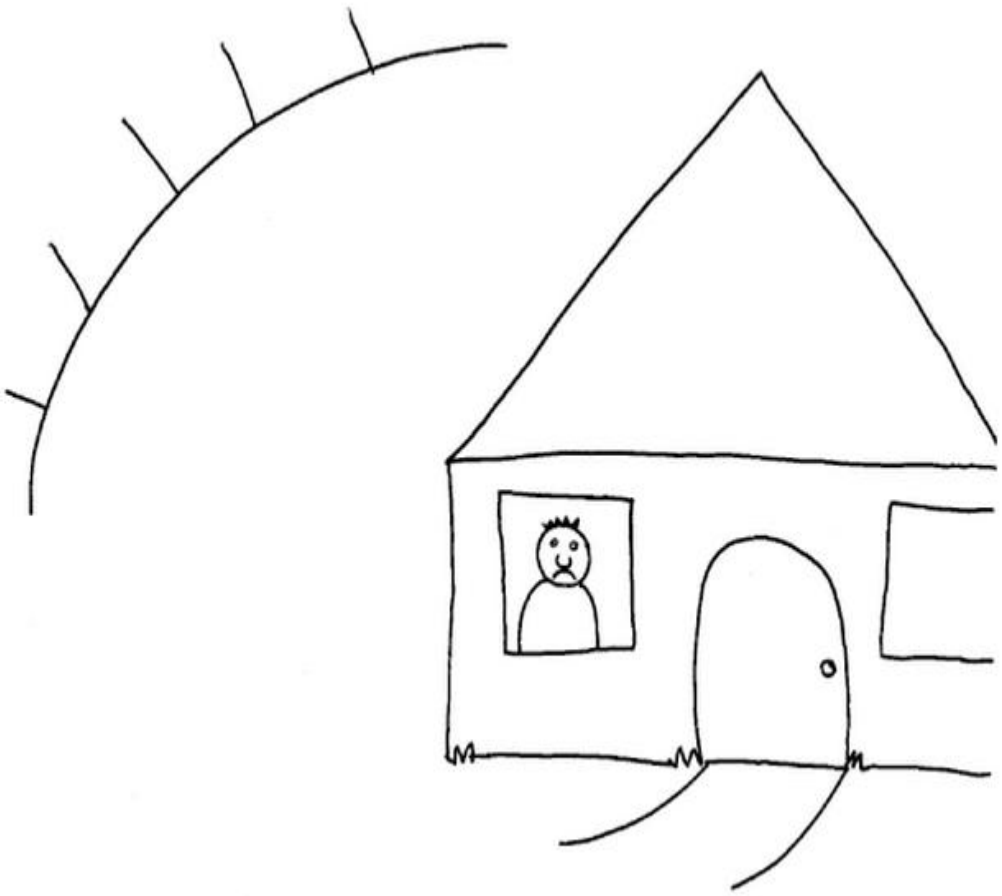
**或者搬到离森林更远的地方**



**我们用新的方式来编程。**

尝试用新的技术和方法来规避问题是一个双刃剑的做法，比如，为了规避开发一个高风险模块，有时我们使用了第三方的中间件。这个可以完全规避开发中的问题，但是结果是引入了集成其他软件带来的风险。

**当然，我们可以躲在里面，度过冬天**

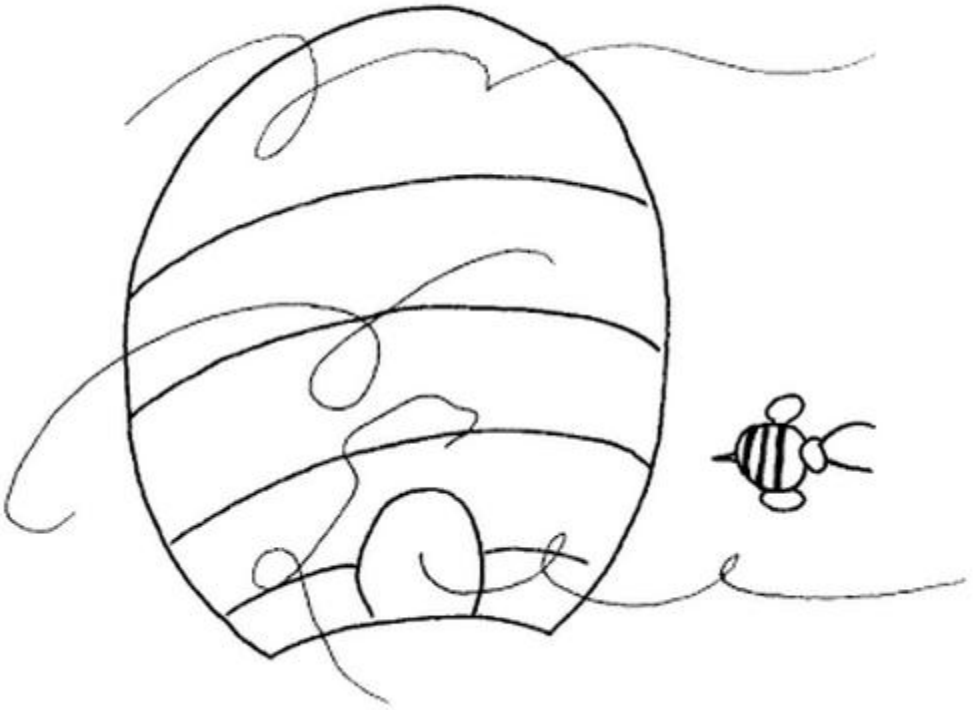


**当我们不知道该做些什么的时候，我们可以坐下来,等到一切都平息了下来**

有时候最好的办法是停下来，看看我们的全景图。取代静待和编码，还有挥之不去的担忧,我们应该花时间去理解需求和识别有用的技术,工具,技巧,和过程。否则你会解决错误的问题。

有时一个项目本身是没必要的，曾经我帮助一个开发团队理清了需求，里面有很多不符合的需求，最终项目取消了。这个项目一开始就不应该启动，如果他们在最开始就花时间了解了这些问题的话，项目就不会启动了。

**用烟熏走蜜蜂**



**我们用很多方法来处理 bug**

**我们可以把它吓走，它们并没有死，但是已经被规避了。**

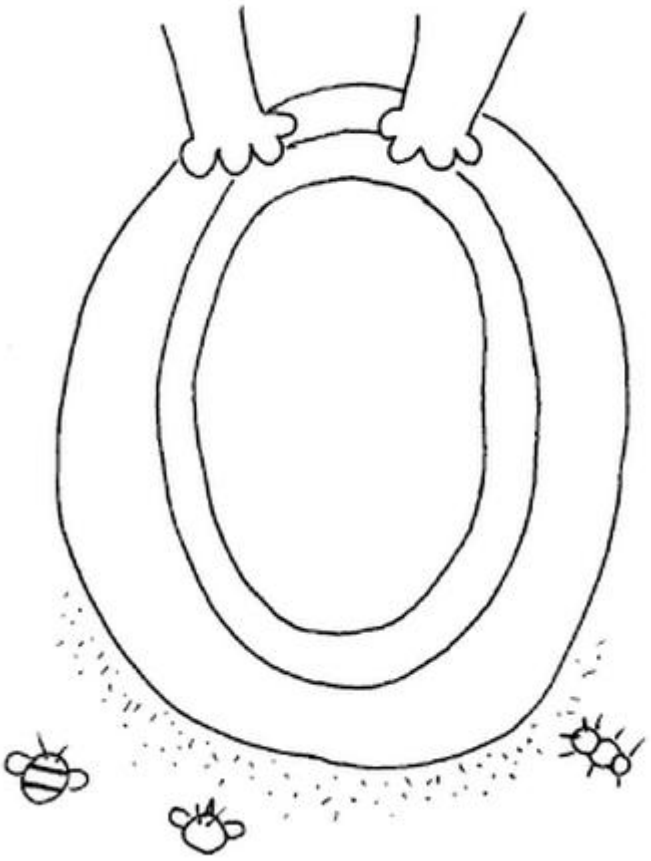
有时候暴露越多的 bug 是一个好事情因为我们能否感知他们的存在。

有时候你可能无法隔离或修复它们,并且会存在很长时间,但是你知道他们在什么地方。

比如，有些公司在产品发布之前会开展激动人心的活动叫做 bug 追击活动。员工周末加班利用所有他们能想到的办法来发现产品缺陷；这些 bug 很可能在产品发布前不会修复或者隔离开，但是公司知道这些 bug 存在。

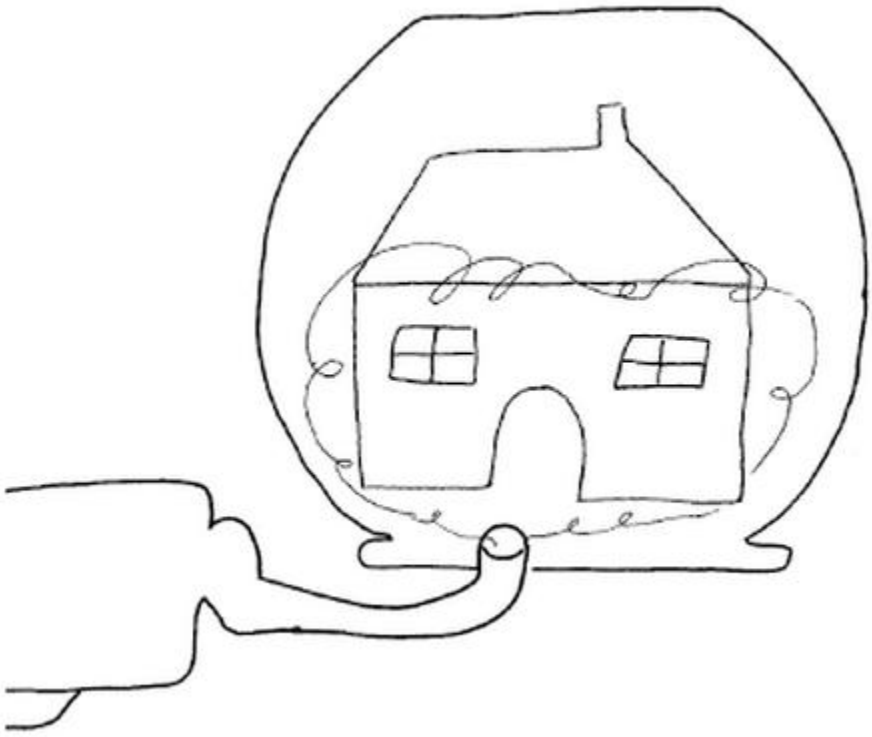
Noel Nyman 发明了一种测试叫做猴子测试，一种随机测试技术，探讨了应用程序随机输入,并用随机输入的方法找出 BUG。这种测试一般是自动化执行的，它能找到生僻的 bug,测试人员可能从来不会发现。

**用地毯甩脱 bug**



**我们摆动一下，使得这些 bug 被甩到其他地方。**

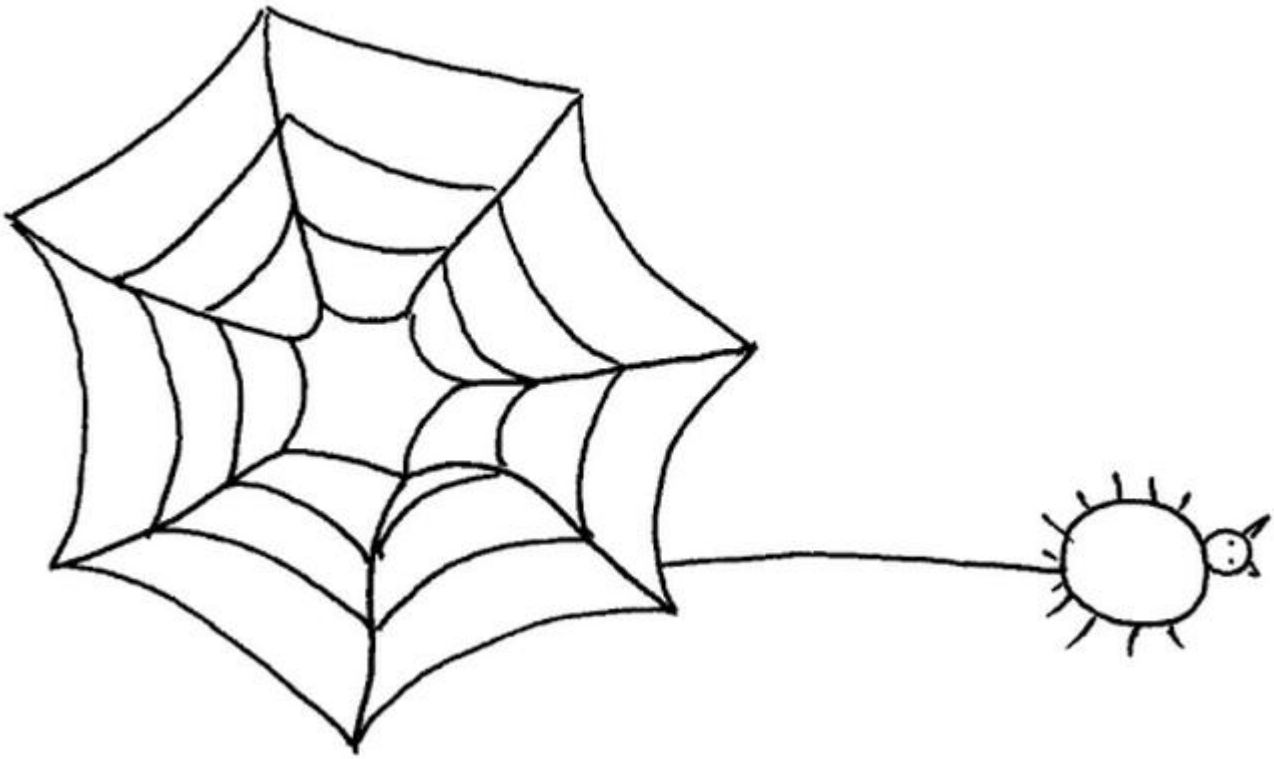
**给房子消毒**



**这个比较好的处理他们的方法。**

**我们修复它并且确保他们不会爬到其他地方。**

**蜘蛛利用网来捕获 bug**



**我朋友 James 通过设置 assertions 执行程序来捕获 bug。**

断言测试方法不管程序是在执行中处于什么状态，比如，当一个文件在被写入之前是 opened 状态，如果断言是否定的判决，那么开发人员就会认为是一个 bug。

在单元测试和早期的系统测试中，断言是快速发现 bug 的有效方法。但是，断言并不能提供业务决策需要的信息。如果断言跳出来了，程序也就停止运行了。在系统测试的后期中，这意味着直到 bug 被修复我们才能继续测试，如果这些 bug 是一般的缺陷，那么时间将耗费在这个上面，因此断言在这个阶段应该退出舞台，断言更不适合存在于要发布的一个商用版本中。

**青蛙喜欢 bug**

**人们在工作中积极的关注 bug，我们热衷于寻找 bug。**

**青蛙吃掉 bug**

**当我们真正的找到 bug 时，我们修复他们。**

是否需要修复 bug 取决于业务环境。就像青蛙的胃只能容纳一定量的 bug，我们的开发团队也没有那么多的时间 ,精力 ,钱来修复所有的 bug。因此 bug 的重要性和优先级需要定义以便做投入产出比分析以便确定哪些 bug 是值得修改的。

不修复 bug 的成本是对用户电脑的影响，大量投诉电话，公司的声誉等等。一些深藏的缺陷的修复带来的是 另一个 bug 被植入的风险或是发现更多的缺陷的机会丢失，开发时间和精力的耗费，降低可用性，性能等质量目标的机会。

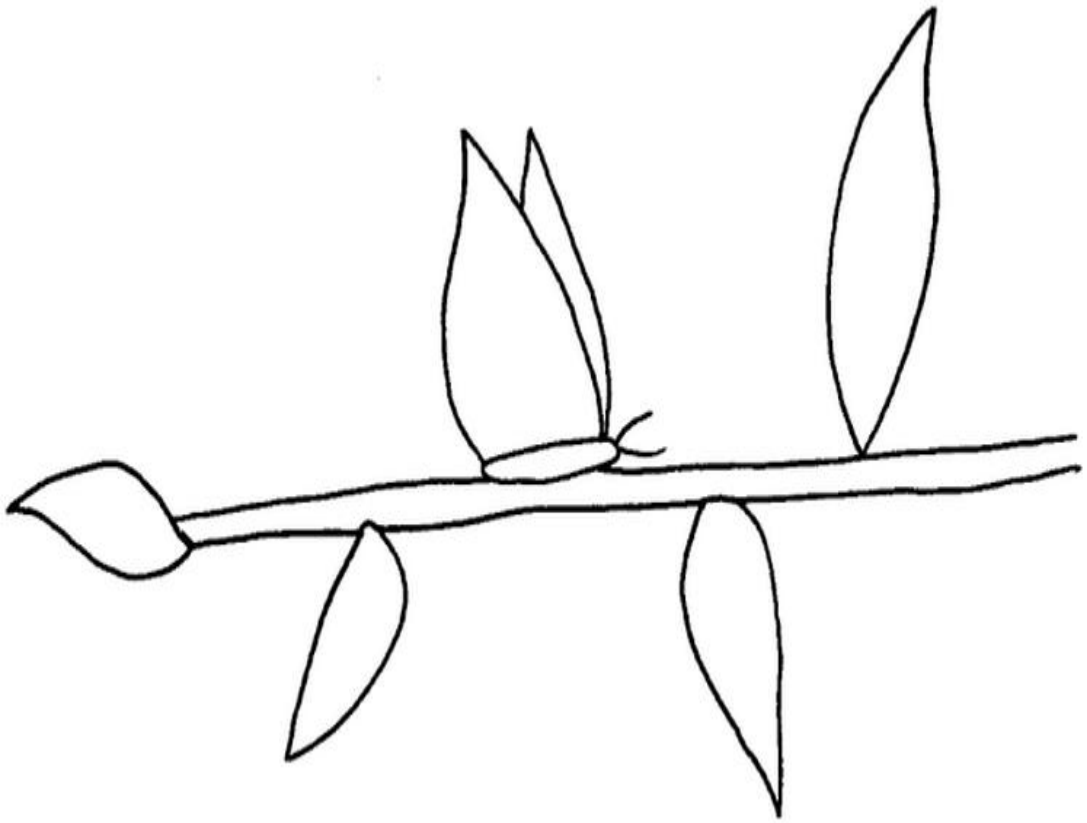
**青蛙只能吃那么点 bug**

**我们一次修复一个 bug ,当 bug 很多的时候 ,我们分发给很多人来一起修复。**

**bug 想躲开青蛙**

**显然的，当我们修复 bug 的时候，bug 尝试避开我们。**

**Bug 通过迷彩色来伪装自己**



**有时候我们就算 bug 近在眼前，却不能找到 bug 的解决方法。**

有很多种 bug 伪装的方法。

一种是普通模式，比如，在登陆之前抛出异常的 bug，在测试中，测试人员会直接登录系统而忽视这个 bug，不知不觉就漏过去了。这种情况下测试人员没有把这个当成一个 bug，长此以往，就被忽略了。

另一种是一个 bug 被另一个 bug 给掩盖了，比如，在一个数据库系统中，返回了一个错误的查询的 bug 可以被查询报告的 Bug 给掩盖了。

还有一种是小 bug（或者是众多小 bug）掩盖了大的 bug，比如一个灾难性的错误被用户接口的错误掩盖，因为程序执行中根本跑不到这段代码。这也

是为什么在 bug 修复之后需要重新测试的原因，他确保没有其他的 bug 还遗留在系统中。

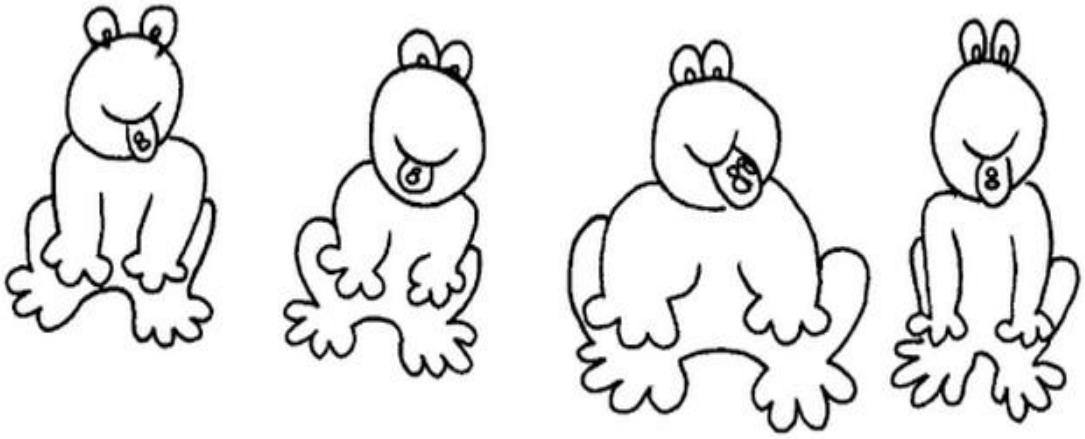
**但是青蛙也可以**

**有时候我们也需要默默地等待 Bug 现身**

一些 bug 可以很好被系统检测到，对于这些 bug，比如 BoundsChecker 安装了就可以观察到系统的行为，包括不适合的系统资源侵入使用和内存泄露等。

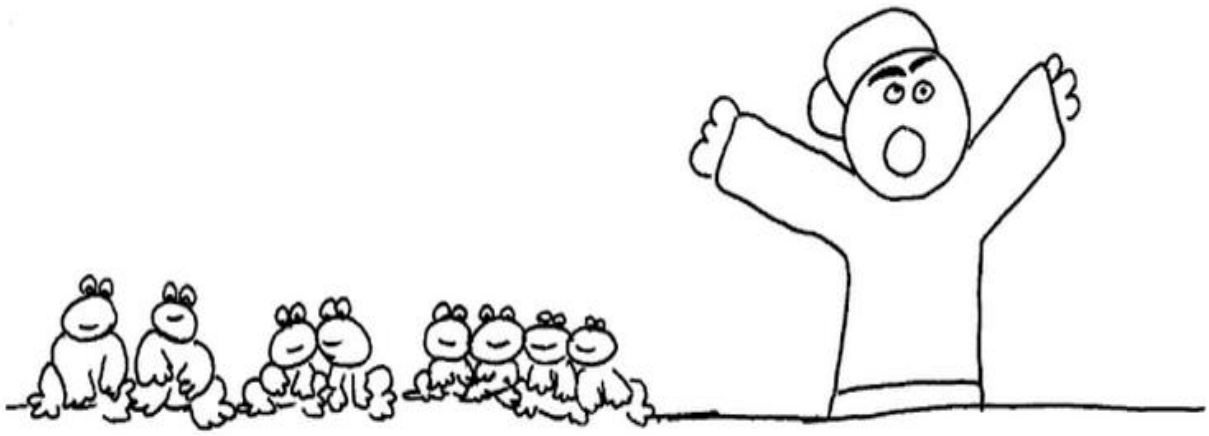
还有一些 bug 可以通过模糊测试来发现，为了找出浏览器兼容性的 bug，只要通过不同浏览器来测试就可以暴露这些 bug。

**越多的青蛙可以处理越多的 bug**



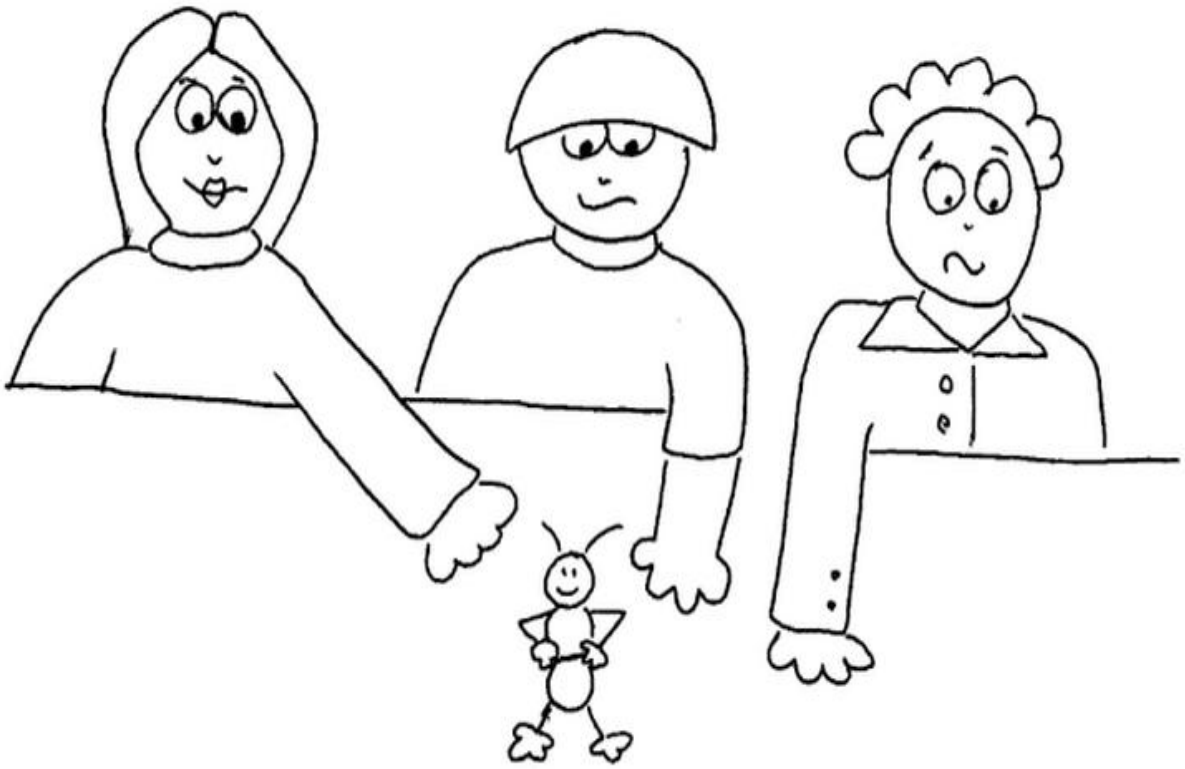
**曾经我们确信可以通过增加人手就可以找出和修复所有的 bug。**

**但是这么多的青蛙在同一个地方是一个大问题。**



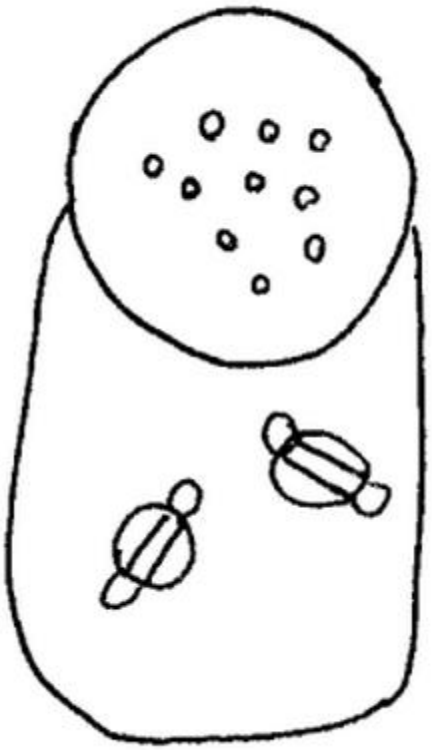
**但是这无济于事，因为这让我们陷入混乱和困惑。**

**当发现一个 bug 的时候你应该怎么办**



**当我们发现一个 bug ,或者一箩筐的 bug ,我们集合起来开个 bug 评审会。**

**我是一个 bug：把它收集到一个罐子里，顶部有孔，我们研究它。**

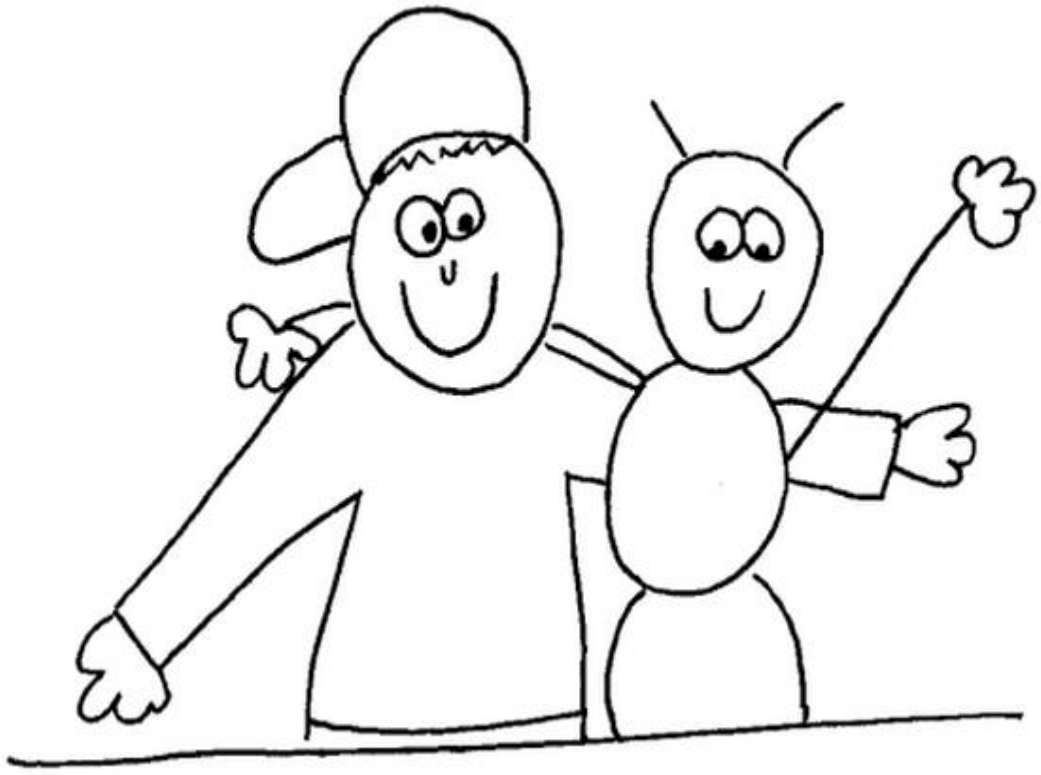


**我们仔细的研究 bug。**

**我们怎么抓住 bug，他重要吗？他的破坏性有多大？**

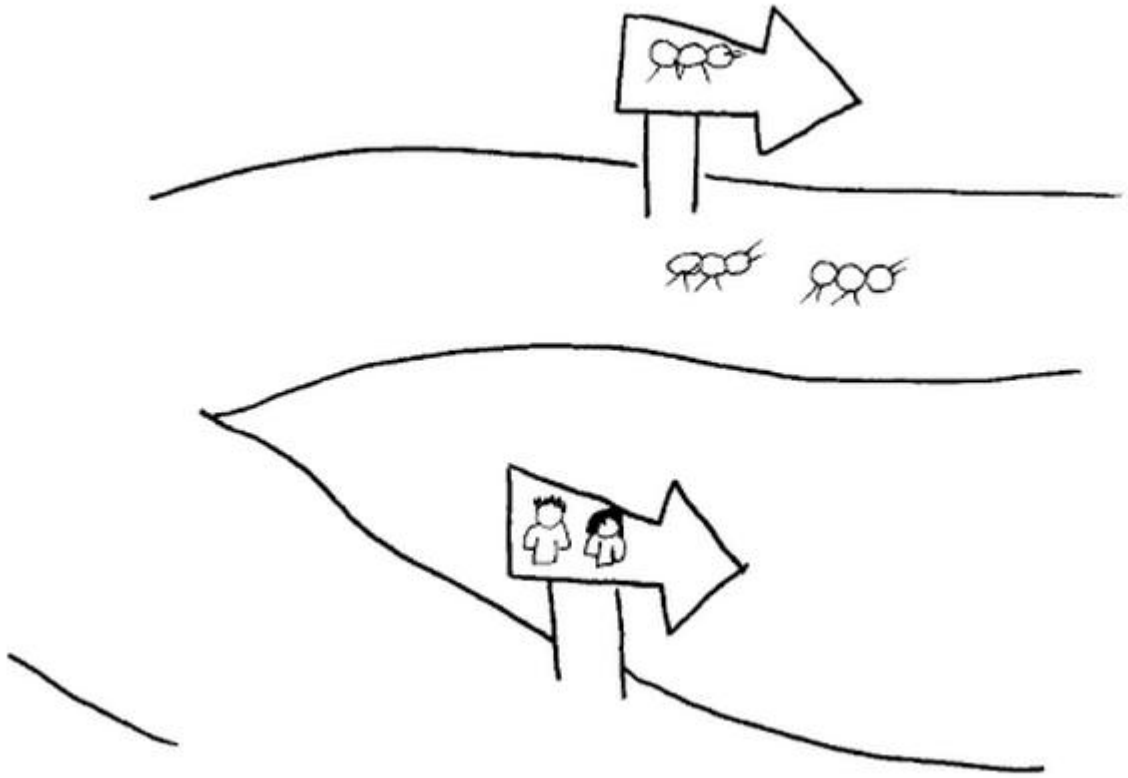
**我们观察：我们是如何发现这个 bug 的？是否需要现在就修复它？它的破坏性有多大？**

**有些 bug 是我们的朋友**



**如果我们足够幸运的话，我们可以把这些友好型的 bug 放在程序里面很长一段时间。**

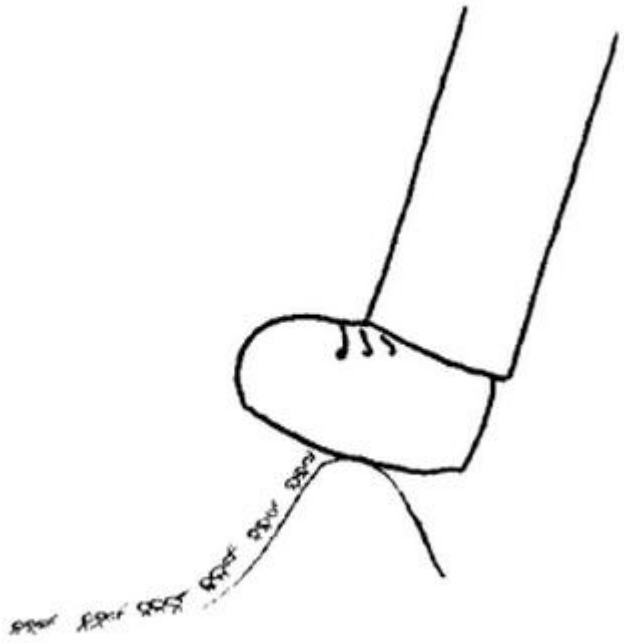
**有些 bug 是只要我们远离它就可以的。**



**如果我们用不同的方式来使用这个程序，我们可以避免这些 bug 的触发。**

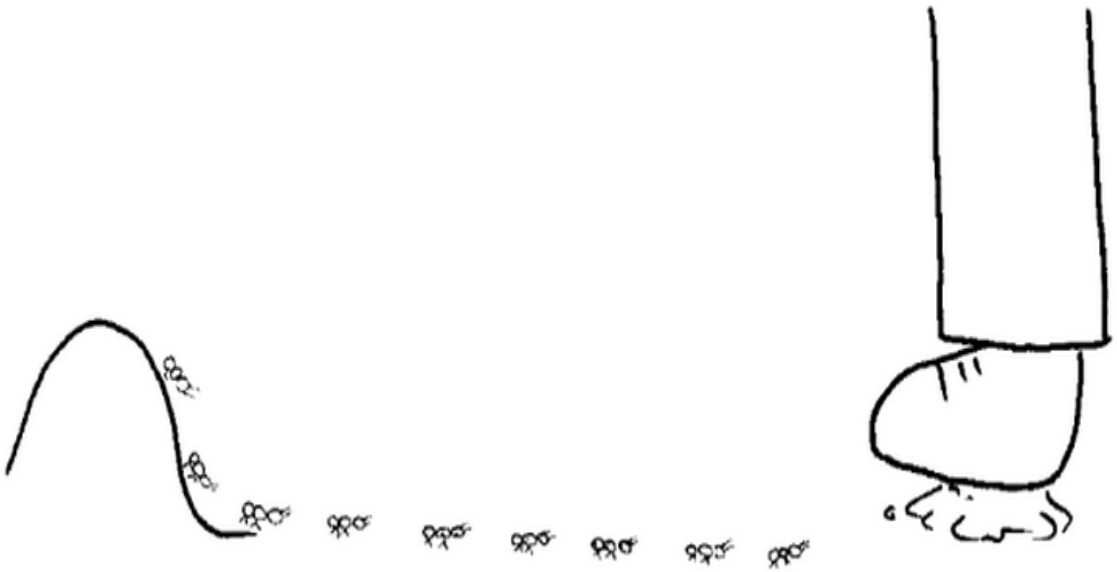
有时候我们可以提供更好的程序使用方式，而不是去修复 bug，在没有时间回归测试的情况下这一点特别对 ,因为修复一个 bug 很可能带来更多的 bug。

**踩住蚂蚁山尖可以避免蚂蚁过来。**



**有时候是堵住洞迫使蚂蚁出来。**

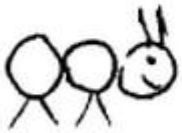
**然后一段时候后放开**

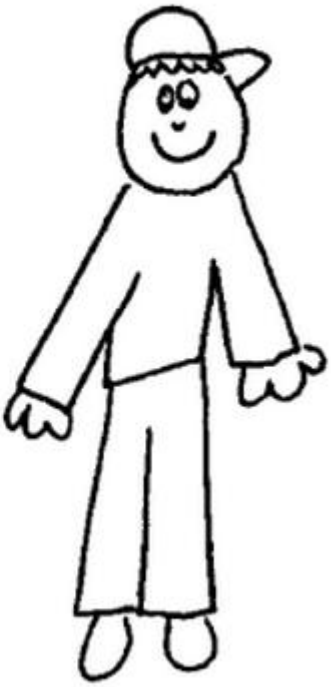


**让问题转移到另外的地方。**

热补丁并不能解决问题，第一次就应该修复它。

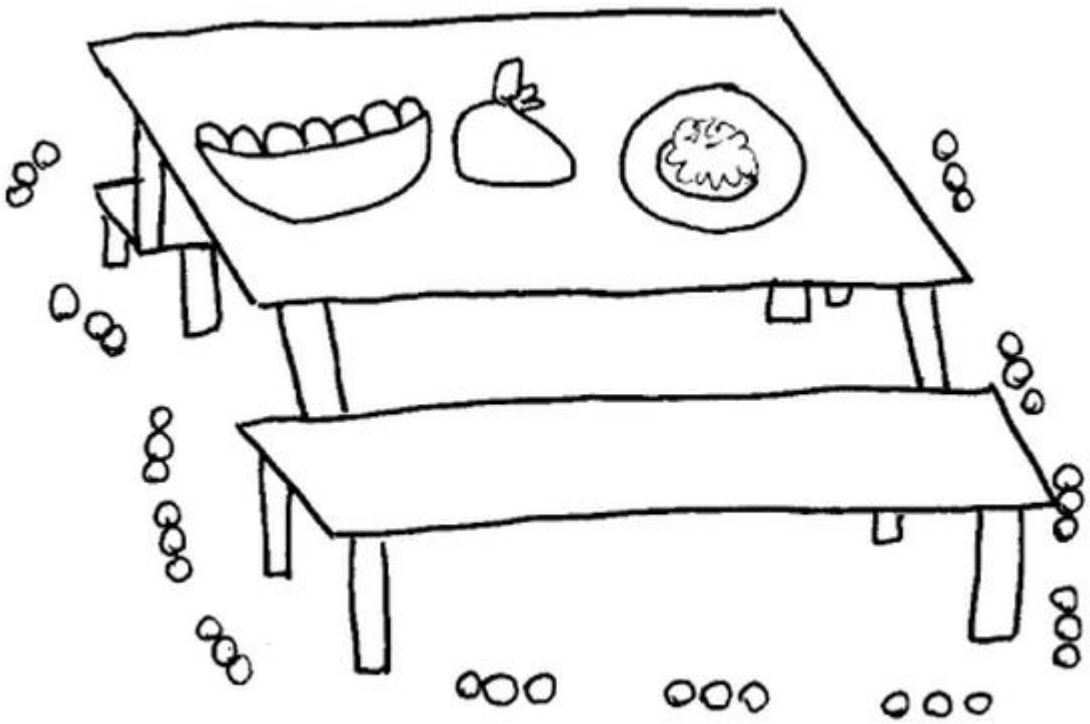
**和蚂蚁共存并不见得就是很坏的情况。**





**在我们决定把 bug 遗留的时候。**

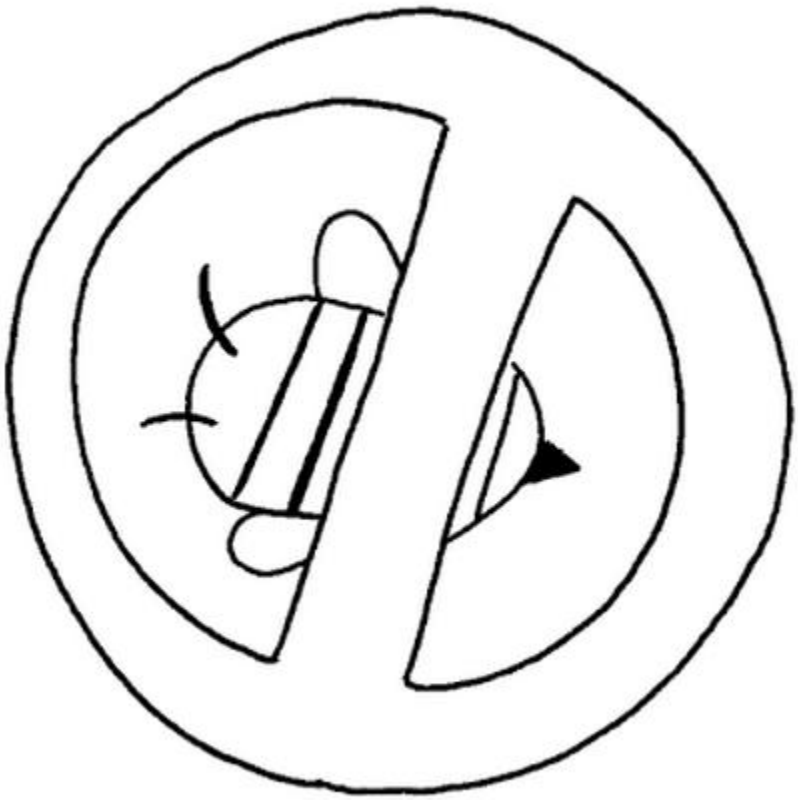
**一般我们在桌上吃晚餐，而不是在地上。**



**我们必须清楚地知道程序将会如何被使用。**

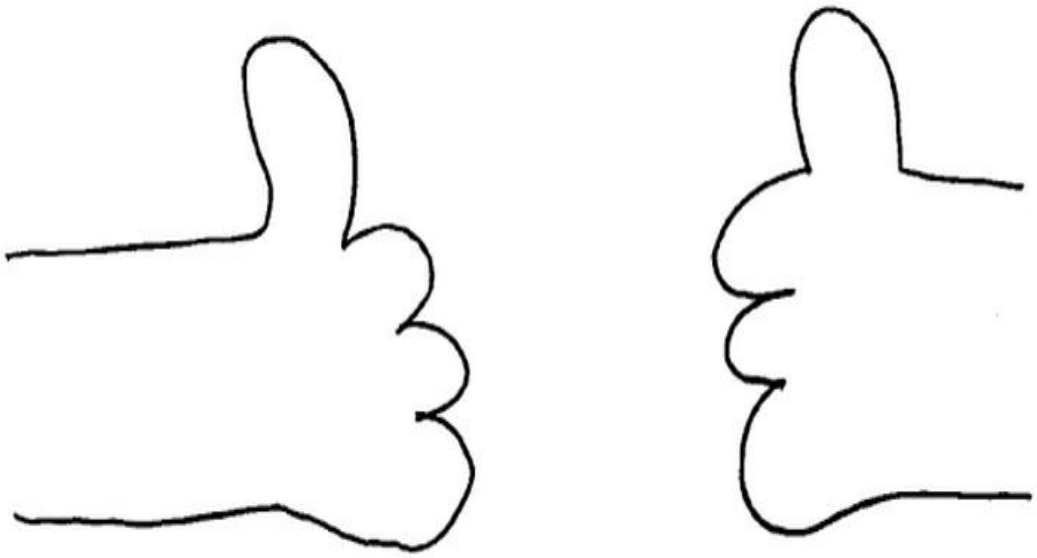
考虑程序将被如何使用，就算是同一个程序也可能被不同的人用不同的方式使用。一个 bug 对一种用户来说的致命的对于另一类用户来说是无关紧要的。

**如果你必须处理它们。**



**当我们决定处理一个 bug**

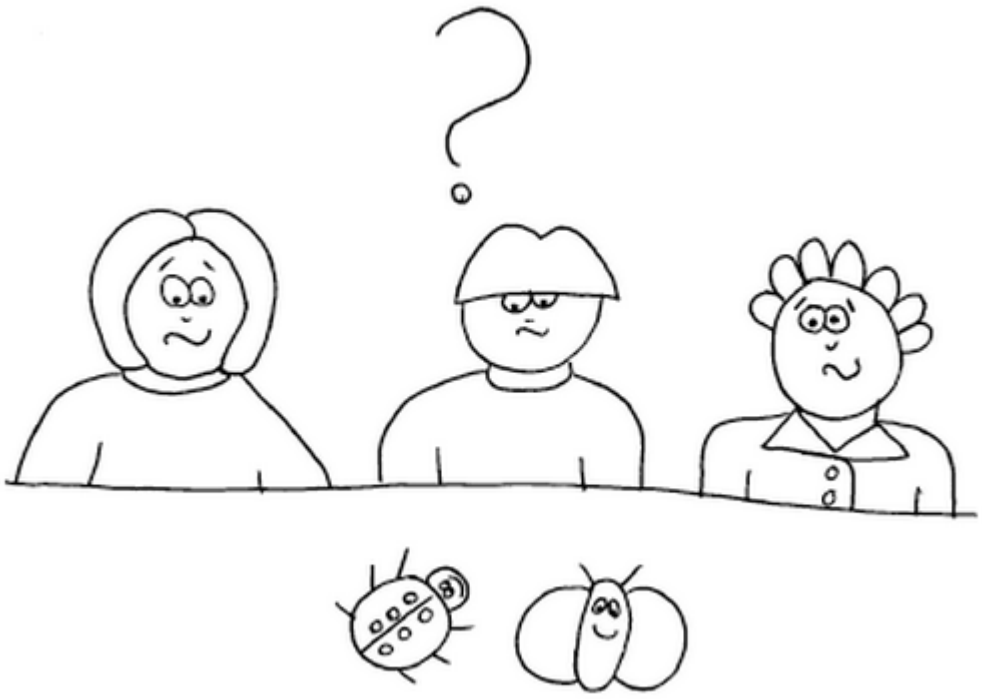
**处理好它**



**我们很自豪的修复这个 bug，我们专业的，双重的检查 bug 已经修复。**

回归测试经常用于确保 bug 已经被修复了 ,并且没有其他 bug 被引入程序。

**你想着你已经完成了。**



**最重要的问题是我们要问自己我们是不是已经完成了我们的工作。**

**是时候让某些人使用的我们的程序了吗？**

**能发布了吗？**

**我们看着我们的已知的 bug。**

我发现软件工程师的一个基本问题：如何知道你已经完成任务了。很多开发人员开始开发的时候并不知道这个问题的答案。

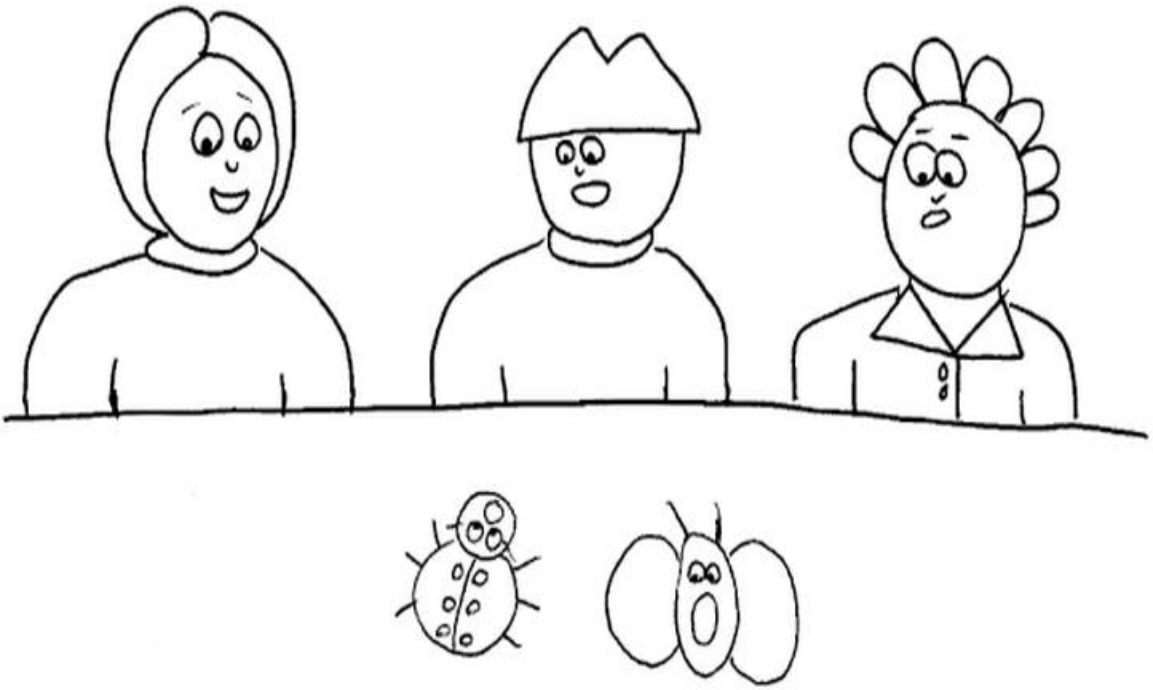
答案有好有坏，一张情况是开发人员觉得要下班回家的时候就是完成任务了，他们快速的完成任务就进入到开发的下个环节，我发现这样的结果就是糟糕的质量。如果我们获取到需求管理，配置管理，测试管理的相关信息再来回答这个问题可能会更好。

回答好这个问题也许需要获取到 某种程序的代码覆盖率，静态分析结果，质量属性情况如可用性，可靠性，可维护性等的信息。

但是为了问出这个基本的问题，准出标准必须精确定义并且达成共识。比如，用单元测试来作为准出条件，会问到“我已经实现了这个单元/模块的代码了吗？准备好移交给其他人员了吗”如果单元测试的很模糊的，它很可能（大多数情况下就是）是不可控的准出条件，因为开发人员没有正确理解这个单元代码意图。这种情况下，当开发人员使用单元测试的结果来决定这个单元的代码是否开发完成，他是基于一个错误的假定的。

这也就是我们需要在项目或者一个特定任务中要有共同的语言，有套共同遵循的规则，比如准入准出条件，通过这种方式，项目中的人都知道在项目中扮演什么角色，应该遵循哪些规则。

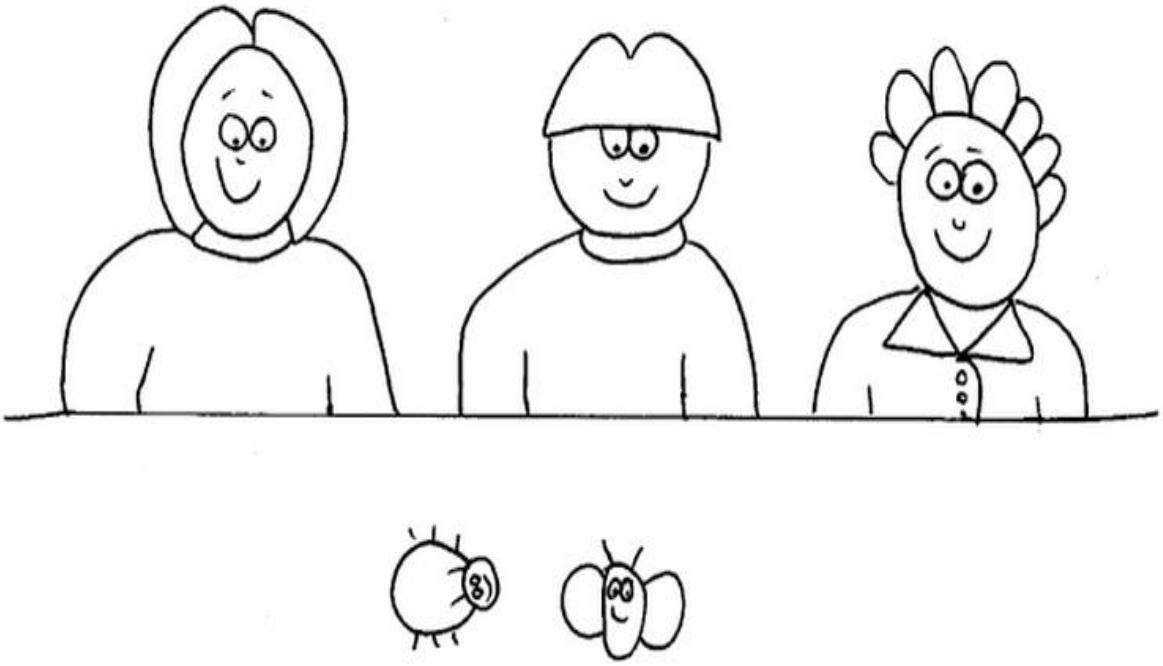
**当只有一个 bug 遗留时**



**我们问自己它的重要性如何，它对系统影响有多大。**

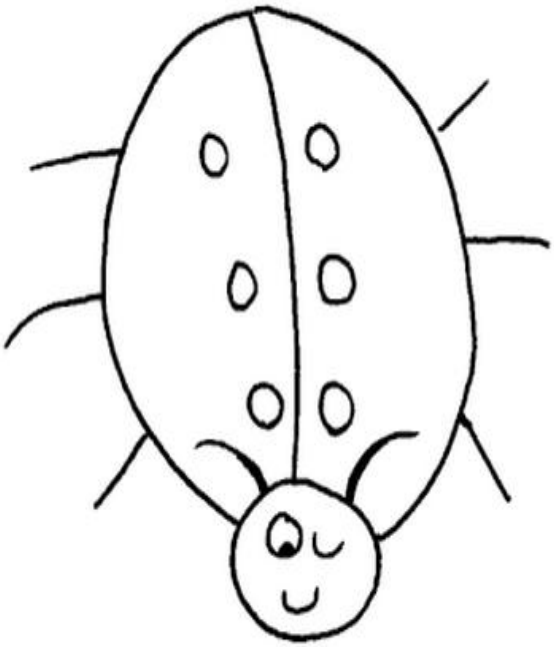
**同样 bug 也在看着我们。**

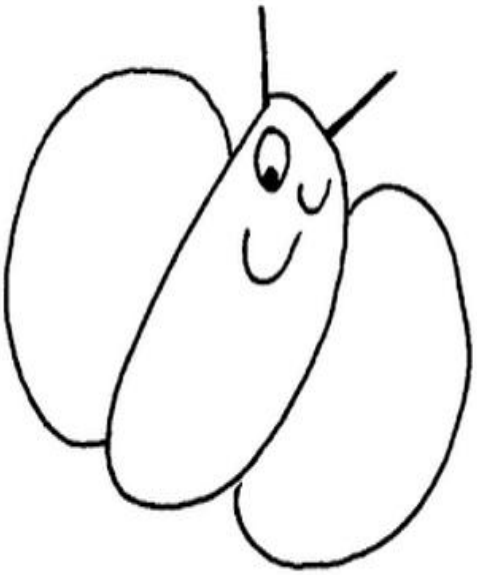
**Bug 确定是可以和我们共存的。**



**当我们确定遗留的 bug 可以和我们共存的时候，我们确定我们的工作已经完成了。**

**至少到现在为止**





**至少到现在为止**

**我们停止测试并且可以洗洗睡了。**

**晚安！**