TDD 生存手册

做TDD是为什么？

关于TDD的概念、工具、技巧等，经典的书籍材料可能介绍的更为全面细致。我所能分享的是从一个普通开发的角度怎么看待TDD的。以及我是怎么从感兴趣，到充满困惑，再到有限的尝试，直到有一天蓦然回首发现已经自然而然的用起了TDD的过程。希望能对有着类似困惑仍在探索的同学有所帮助。

遗憾的是，在开始所谓“干货”以前，首先还是要谈谈理念。因为我发现这是一个绕不过去的问题。

你为什么要使用TDD/写unit test？

不同的人也许有不同的答案：

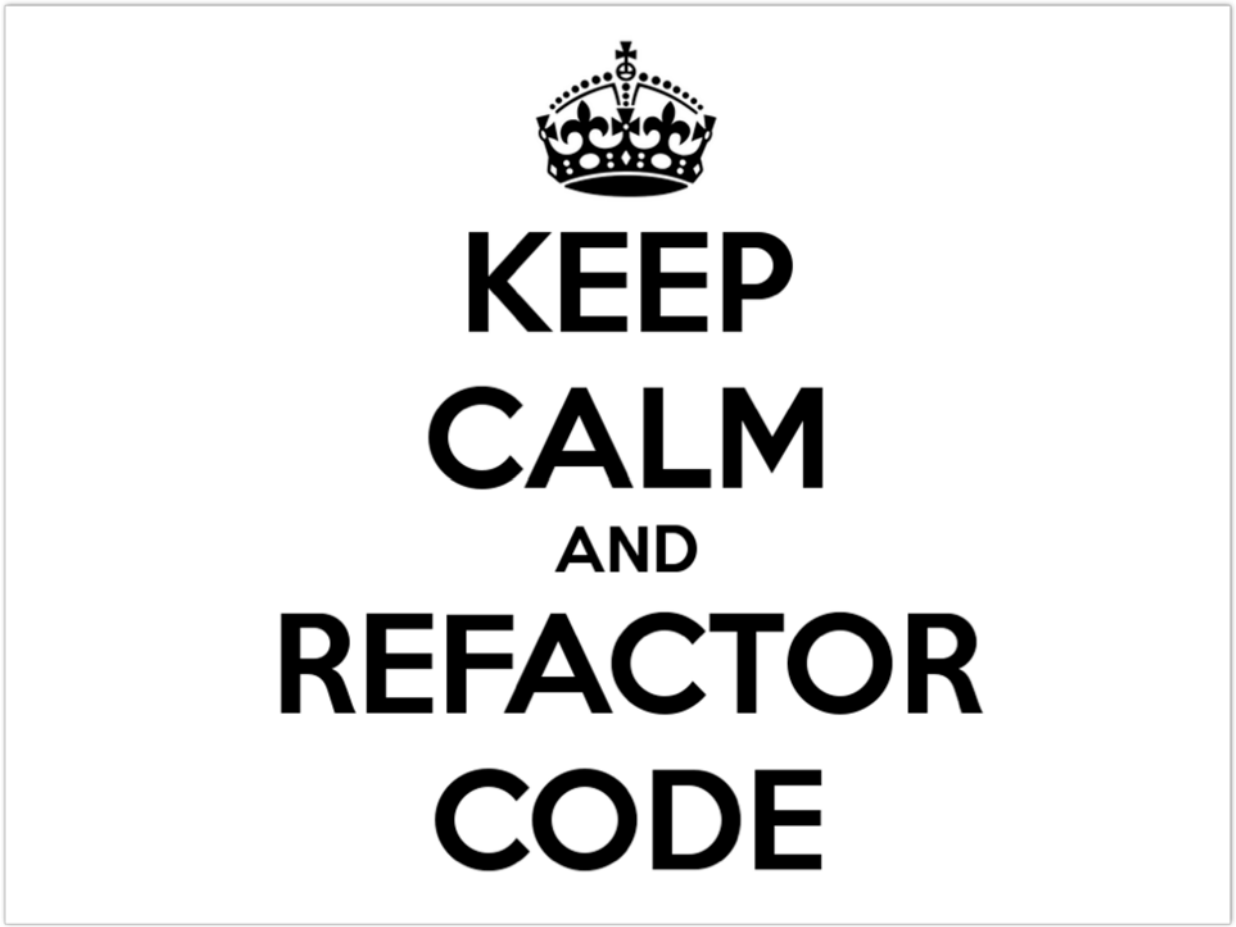
1. 因为这是现在流行的，“正确”的开发方式；
2. 因为这样写出来的代码质量更高；
3. 因为TDD和unit test能产生更好的设计；
4. 因为老板要求必须达到xx%的覆盖率；
5. ……

以及一个派生的问题，

如果说：测试只能用来证明bug的存在，而不能证明程序没有bug。

那么：**写Unit Test的意义是什么？程序员写出的Unit Test与软件质量有什么关系？**

一切不以重构为目标的单元测试都是耍流氓



当然，这里是指在TDD语境下的单元测试。

在与同道交流TDD经验，特别是与测试人员交流时。我们明显的发现，TDD所说的Test，与测试人员口中的Test完全不是一回事。我们甚至讨论过能不能用其它的词语替换“测试”或Test，来避免歧义。

经过朋友的启发和反思自己对TDD的执念的来源后，我发现对于我来说，TDD中写测试的真正目的，是重构。

- 我常常会在读代码或写代码时产生种种的冲动：“这是什么鬼”，“我为什么要把生命浪费在这种东西上”，“一定有更好的办法”。
- 我需要通过重构来写出更合理的代码。
- 为了安全的重构，我需要测试。

而与TDD相关的其它好处，比如文档化，将来作为回归测试集，促使开发人员从用户角度思考等等，都只是在更高效的改进代码的过程中附带产生的。

换句话说，如果你不准备在将来修改代码，无论主动（重构）还是被动（改bug，加功能），那么写单元测试对你完全是浪费时间。

但是话说回来，如果你真的确信这段代码永远无需修改，那么不要说单元测试，源代码也是没有必要的。不是么？

回到前面的另一个问题，TDD中的单元测试与代码质量之间的关系。

我的回答是：测试用例本身不能保证质量。

并不是有了更多的测试数量，更高的覆盖比例，代码就自然变好了。如果说TDD能提高质量，那一定是因为TDD给了开发者安全和快速反馈的环境进行重构，从而帮助开发者不断改进写出更好的代码。

打个比方，同一个作者，一篇文章是在交稿前半个小时赶着写完，错别字都没改就发出来的；另一篇发布以前斟酌再三，几易其稿。哪一篇的质量会更高一些呢？

答案是显而易见的吧。不过请再想一想，写代码与写文章的相似程度有多少？代码真的是越改质量越高么？

反脆弱的代码

> “反脆弱”是《反脆弱》这本书的作者生造的一个词。描述的是脆弱的反面，一种我们都知道却没有名称的性质。一般我们认为脆弱的反面是坚固，然而坚固仅仅是对外部变化不敏感。反脆弱指的是具有这种性质的东西可以从外部变化中获利，正如同脆弱的东西会被外部变化损害一样。

对大部分的程序员而言，变化是个不受欢迎的词。在我们谈论健壮的代码，合理的设计时，针对的假想敌就是未来的变化。关于将来的变化，我们能想到的最好结果不过是不要搞砸现在设计好的一切。

换句话说，我们追求的是坚固的代码，历经变化的侵蚀屹立不倒。

那么，有没有反脆弱的代码，在变化的滋养中生长壮大呢？

对待可能的变化，不外乎三种态度：

1. 这段代码不打算在将来再被利用了，所以完全不用考虑改变。这不失为一种实用的态度。但是现实中这样的情况太少。

2. 现在写出一个完美的设计，为所有可能的改变做好准备，这样将来就不会改变了。但是这是可望而不可即的目标。暂且不论需求变更等不受我们控制的，外部变化。仅仅就开发者自己而言，不论我们今天作出多少努力，随着我们在解决问题过程中的成长，往往在明天我们就是会遗憾当初没有作出更好的选择。

3. 为改变做好准备，并且主动地，时时刻刻地进行改变。这就是TDD的选择，可靠地对代码进行改变。并在这种改变中不断改善。

对于不熟悉的人而言，初看起来，TDD最大的特点是在实现代码之前写测试这个反直觉的实践。却往往忽略了藏在后面重构的那一步。事实上，前两步的红灯、绿灯，都是在为第三步的重构做准备。

第一步，写出失败的测试。是在为即将发生的重构建起保护网。

第二步，尽快的绿灯通过。是**刻意**写出需要重构的代码。

既然认为改变是有潜在破坏性的，那就尽早地、尽可能频繁地去改变代码。

测试与重构，像是硬币的两面一样，密不可分。

所以，如果你仍然认为重构像是吃完饭要洗碗一样的必要但是附属性的工作。如果你还没有感受到TDD带给你的保护与自由，让你放开对改变的恐惧，心安理得的写下将来必然会被改掉的代码。那么就算你按照三步循环去写代码，恐怕也难以从中获得益处。很快就退回尽量预先思索，想小步却慢不下来的老路上。

何谓持续

事实上，任何一个老练的程序员必然都有自己的一套方法来反复验证和调整正在开发的代码。这些方法可能包括，可控环境下的调试，添加一个临时的main方法作为实验入口，把代码片段复制到外部环境进行验证等等。TDD中的增量开发、小步快跑，用这些方法也可以做到。

我想这大概就是为什么有人会提出其实人人都在做TDD吧。尽管我不是特别认同这种说法。

如果没有那个点的话，也许做不做TDD确实无所谓。

这个点有时候叫做交付，也可能叫集成、发布；甚至有时候并没有一个清晰的事件点，不过是写完放下，过了几个星期而已。但是这个点是实实在在存在的，它就是“鲜活”代码和遗留代码的分界点。越过了这一点，你手中的代码就会摇身一变，从那个开朗敏捷的少年，变做阴郁固执喜怒无常的怪兽。

TDD的独特之处，是让测试伴随代码从生到死的整个生命周期，始终为代码变化提供保护网，让代码的“保鲜期”尽可能的长，抹平这个转变的节点。

现在持续集成、持续交付的概念已经是主流了。但是什么是持续呢？个人浅见，不是说设置了一个服务器，定时跑几个任务就是持续了。而是不再有那个代码保鲜期的拐点，可以一直平滑的发展下去。

TDD无疑是它的重要保证环节。

开发者体验

TDD的好处有哪些？关于这个问题，我原来总是尝试从客观的角度来回答。比如质量，比如可维护性，比如鼓励好的设计等等。总之，就是剔除了人的因素。

然而，当我认真探索自己这一路走来的历程。是怎么在TDD上略有心得后情不自禁的在社区分享。重新开始写博客，几乎每篇都是关于TDD的。自发的公司里组织编程道场（Dojo）推广TDD。背后的动力其实很简单，这样开发让我很爽。

这个答案听起来实在太不正式，好像也没啥说服力。但是确实是我的真实想法。

有人可能会说：工作嘛哪能那么理想化，老板给你工资就行了，谁管你开心不开心。

且不说更开心的程序员应该效率更高，而且开心本身就是公司状况良好的体现之类的客观化的理由。

从开发者个人而言，就算仅仅为了心情愉快、延年益寿，也是值得去做些努力去改进代码的。因为改善代码质量和开发流程，本身就是改善工作环境。

最近恰好读了一篇研究程序员各种不爽的论文。其中统计了上千个程序员的答卷。对工作中的不爽进行了分类。

可以看到尽管工作中有不少不受我们控制的部分，比如人的原因（416个）和公司流程（544个），
但是最大的一部分还是来源于代码相关问题（788个）

Table 1: Categories for External Causes of Unhappiness

Main category	Sub-categories
People (416)	Colleague (206) Manager (122) Customer (49)
Artifact and working with artifact (788)	Code and coding (217) Bug and bug fixing (194) Technical infrastructure (151) Requirements (99)
Process-related factors (544)	No sub-categories
Other causes (95)	No sub-categories

再来看看常见的让程序员不爽的原因。前三位中的两个是：

- 解决问题被卡住。

- 糟糕的代码质量以及代码习惯。

Table 2: Top 10 Causes of Unhappiness, Categories, and Frequency

Cause	Category	Freq.
Being stuck in problem solving	software developer's own being	186
Time pressure	external causes → process	152
Bad code quality and coding practice	external causes → artifact and working with artifact → code and coding	107
Under-performing colleague	external causes → people → colleague	71
Feel inadequate with work	software developer's own being	63
Mundane or repetitive task	external causes → process	60
Unexplained broken code	external causes → artifact and working with artifact → code and coding	57
Bad decision making	external causes → process	42
Imposed limitation on development	external causes → artifact and working with artifact → technical infrastructure	40
Personal issues – not work related	software developer's own being	39

而这些都是可以通过开发者自己努力来改善的。我的切身感受，TDD带给了我如下变化：

- 交付代码的时候充满了信心。
- 从测试或者客户那里得到意外的错误后，不是感觉恐慌，而是回顾一遍测试，往往已经能定位到原因了。
- 几乎从不调试程序。
- 要修改遗留代码，对质量又不满意的时候，不再一边忍耐一边抱怨。因为我心里很清楚，我能可靠的改掉它，只要有必要这么做。

我想这大概就是TDD为什么给我带来这么大幸福感的原因吧。

成长路径

下面我结合个人体验写一下从初识TDD，到能够得心应手的使用的过程，希望能有所帮助。

着土

掌握最基本的，让TDD成为可能的技术。比如：什么是单元测试，如何在不同环境下运行单元测试，有哪些可选的框架等等。

在网络时代，这个阶段应该是最容易的，各种资源和教程触手可及。另外随着业界对测试越来越重视，较新的语言、框架、平台都把测试作为标配提供支持。所以这个阶段应该很容易就能度过。

出芽

尝试使用TDD做一些简单程序。体会红灯、绿灯、重构的循环过程。

本阶段往往有两个结果，一种是试了试完全摸不着头脑；另一种是试了试非常好用，然后拿去实用发现完全不是那么回事。

正像前面提到的TDD最重要的不是表面上的三步循环，而是转变编写程序的思路。如果你仍怀着对修改代码的恐惧，依赖于提前”想清楚“，那么先写测试并不会帮到你多少。这更像是学习骑自行车或游泳一样，仅仅理解并没太大用处，需要一个过程去体会和掌握。

本阶段可以说是一大难点，很多人可能就是在这里觉得TDD可望不可即，或者仅仅是看起来很美。下面是我的一些建议。

1. 一开始可以亦步亦趋根据教程示例做一遍。但是之后一定要找一个没有做过的题目尝试自己解决。

2. 不宜选择简单到你一下就可以在脑子里写出伪代码的问题，但是也不要选过于复杂的问题。练习常用的Kata是个不错的选择。详情见后面的Kata介绍。

3. 很有可能尝试了却没有成功，别担心这是正常的。如果你练习的是熟知的Kata的话，可以在网上找找别人解的过程，很多都是有视频的。看完有心得了以后再做一次。

4. “装傻”是本阶段的一个技巧。因为你已经有了一套如何解决问题的方法，在转换到新的做法的中间过程里，往往不自觉的用原有的信念来评判新的做法。这时需要靠装傻来暂时放下已有的东西。
   学习的时候不妨把它作为一项挑战，看看自己能写出多傻的代码，能用多慢的节奏达到目标。

5. “一次一个问题”是另一个需要练习才能掌握的技巧。尝试在循环的每一步**只关注**于一个问题：编写测试、实现功能、或是改善设计。
   这个建议也适用于更高层面的问题。比如，在练习的时候不要去担心诸如：“这样性能太差了”，或者“如果我每段代码都花这么长时间写测试，明天老板就会炒了我”之类的问题。

6. 如果你不把自己限定为一个“Java程序员”或“PHP程序员”，可以考虑用一种不熟悉的语言结合TDD来解决某个熟悉的问题。在重拾初学者身份后，往往会意识到一个

看似简单的问题在解决过程中有多少需要高清的地方，更容易体会到TDD的方式在这个过程中所起的作用。

7. 事实上这个阶段实在有点挑战，我建议最好找人一起练习。代码道场(Dojo）和代码静修(Code Retreat)是很好的练习活动。如果有机会可以考虑参加。关于代码道场，可以看看这位同学的笔记

当然很可能你在周边找不到这样的活动，但是又很想参加。可以考虑自己组织，没错我是认真的。从中你会获得更多意外的收获。

生根

当你在上个阶段获得了收获，对TDD方法有了足够的信心。这时就可以开始考虑在工作中玩真格的了。

如果在上个阶段学到的够多，那么用在工作中并不是很困难的一件事。但是，还是有很多的坑要注意，毕竟这不再是自己捣鼓了。

- 最好选择新增的，相对较为独立的模块开始尝试。

  一方面这是因为可以避开很多技术上的难点，更重要的是因为这种代码涉及的人比较少。相对而言更不容易受到阻力。

  可能你会觉得日常工作中更多的是修改老代码，并没有多少机会新增一块。是的，所以一定要珍惜这样的机会啊！每当我看到已经有了足够能力的程序员在写崭新的代码时，却没有为它配上足够的测试保护，任由它慢慢的变得混乱脆弱。总是无比的惋惜。

- 如果的确没有新模块的机会，可以把比较基础的代码，比如工具类的部分进行抽取，用单元测试围起来，然后进行重构也是不错的。

- 一个常见的困难是感觉采用了TDD后进度慢了很多，担心领导或者老板不答应。

这还真不是个简单问题：

> 首先，要区分真的进度慢了，还是感觉进度慢了。有些时候在压力之下，我们往往是自欺欺人的估算一个“理想情况”下的进度，然后假定真的能赶上。如果是这种情况，实打实的写出测试来更有利于做出现实的估算。虽然拿到任务的第一天就告诉老板延期很难说出口，我认为还是要比最后一天再说要好一些。
>
> 有可能是因为仅仅关注在“开发”的进度上，却没有考虑在调试和测试阶段省下的时间。如果有这样的压力，可以先在不引起太大抵触的范围内采用TDD，并且关注是否在后续的阶段大幅提高了效率。如果的确有效果，相信大家会越来越理解和接受；如果毫无效果，那可能要反省一下是不是哪里做的有问题了。
>
> 学习新的方法是需要一个过程的。这也是为什么在上个阶段特别提出要做专门练习的原因。如果公司和领导并不是特别给你支持，而你又真的希望通过掌握新方

> 法来提高。那可能还是需要自己在工作之外做些努力来度过这个阶段。

- 在压力之下人总是会倾向于采用熟悉的方法。哪怕明知道最后会搞得一团糟也还是这样，毕竟那一团糟是自己熟悉的一团糟。
  因此实做中发现没有练习中那么行云流水是很正常的。给自己定下实际的期望值，逐步提高。比如：
    - 写了这么多代码，至少要有一个测试。
    - 我写的每句代码在交付前至少都用测试验证过。
    - 每次我都先试试先写个测试小步前进，实在不行了再退回原来的方式。

在实际工作中发现退回老路，建议抽出专门的时间按照上个阶段的方式继续练习。我在学习TDD的过程中的最大附带收获就是养成了练习的习惯。

可能很多程序员听到练习两个字就烦。毕竟懒惰是程序员的一大美德嘛。我们是脑力工作者又不是搬砖。练那么熟、记那么多东西又有什么用呢？总还是比不过自动化的程序和搜索引擎。

的确是这样的。不过练习的目的不是超过程序和搜索引擎，而是迁就我们大脑有限的运算量。只有熟练到一定程度，大脑才可以不再疲于应对各种细节，有空去关心真正重要的问题。在改变的过程中这一点非常重要。

破土

随着越来越多的使用新方法，自然而然地会想把它推广到更大的范围。这时就要面对遗留代码这块硬骨头了。

假如你是团队中最早采用TDD的人，很可能碰到很多没有测试，而且难以测试的代码。

这里一定要隆重介绍《修改代码的艺术》(Working Effectively with Legacy Code)。在这个阶段我曾经疑惑了很久，陷入了一个无解的死循环里，多亏了这本书的点拨才得以突破。

这个无解的问题是这样的：

1. 代码好烂，想要重构。
2. 为了重构，需要写测试。
3. 代码好烂，没法测试，先要重构。
4. 为了重构，需要写测试。
5. ……

破解的方法嘛，其实说来很简单。以最少的代价迈出第一步，在没有测试保护的情况下进行重构，为后续有序的循环打开大门。

具体的手法和技巧，这本书里讲的非常好了。建议带着问题去读，一定收获满满。

需要注意的是，有些时候为了在板结的陈旧代码上敲开一条缝，必须要采用一些不是那么“最佳实践”的方式。比如放宽可见性，取消final限制等等。这些做法很有可能会遭到反对。最极端的情况下，为了方便测试修改哪怕一行代码，有些人都会觉得是荒谬的。

这时候反复争论是没有太大意义的。反对者有他们正当的理由。正如前面谈到的坚固与反脆弱的代码的两种心态。他们把这种改变看作千里大堤上的一个蚁穴，还看不到在将来的改善中能带来的收益。
所以，重要的不是谁说服谁，而是做出实效。首先表明自己的做法，然后在互相可接受的限度内去做。

有一点特别特别要注意：不要用PowerMock之类的“黑魔法“去迁就代码，费尽心力只是为了避免因为加测试而修改代码。别忘了，写测试的目的是圈起一块领地来驯服遗留代码，而不是把测试当作一层粉饰去贴在代码之上。

成材

上个阶段可以说是一个分水岭，就像学游泳学会踩水，一旦掌握就“淹不死”了。到了这个阶段你应该已经很有信心的在各种场合使用TDD了。后面主要考虑的是如何更加高效的使用这种方法，怎么带动更多的人。

这个阶段我也还在路上，只能说说我观察到的大规模的推动TDD中可能会碰到的一些坑。

1. 小心Mock滥用。Mock，包括相当一部分的Stub，应该用来表达对象间的职责。而不是模拟不必要的实现细节。

2. 避免深的测试类继承结构。极端情况就是“双树结构”，测试类将生产代码的类结构依样画葫芦又做了一遍。其实我的个人看法是测试类和测试帮助类都根本不应该出现继承。

3. 不要过于执着完全的、绝对一致的方法论。

这可以说是程序员的职业病，无论什么方法听到的第一反应是找反例，即使一万个场合有用，只要一个场合不行，立即就觉得这是个无效的方法。

对于写程序这可能是很好的习惯，毕竟一个万分之一机率崩溃的软件基本上是没用的。但是人不同于机器，并不会碰到一个方法论不能解释的情况就进入死循环。80%情况下好用的方法就已经很有帮助了。

这种心态的另一面，是一旦相信了一种方法，就认定它必须100%贯彻到每个角落。

特别是在刚刚开始进入这一阶段的时候，很容易雄心勃勃的规划一个崭新的版图，一套绝对化的规则来改天换地。

为什么不要这么做？

- 往往缺乏投入产出比，为了写测试而写测试，花费大量精力在已死的代码或等死的代码上。
- 在团队和组织中对TDD有疑虑的情况下徒增反对的可能。
- 规划大，见效慢，有违小步快跑的精神。
- 将干巴巴的规则凌驾于活生生的个例之上，实际上是期待自己的道理能一劳永逸的解决所有问题的懒惰思维。更重要的是堵塞了将来进一步改进的机会。

几个启发性的问题：

1. 一个测试从写好以后就再也没有失败过，说明它非常有效还是完全无用？
2. 看看你最新写的测试，什么时候可以安全的删掉它？到了那个时候，如果是另一个程序员维护，他有没有信心删除？
3. 回顾最新一次TDD的过程，能不能用更少的测试达到同样的信心级别？

附录

一些Kata题目

- FizzBuzz：由于问题非常简单。适合用来讲解TDD的概念。这样学习者的注意力可以全部集中在流程和方法上。但也是因为问题太过简单，不适合自己拿来练习如何用TDD解决问题。
- 因数分解：来自Uncle Bob的题目和解题过程，很好的展示了TDD如何超出预期简单地解决这个问题。
- 罗马数字：有一定复杂度的题目。适合用来练习如何分解问题，以及怎么通过重构简化代码。
- 网球记分：对于不熟悉业务规则的人需要花一点时间搞清楚逻辑。问题本身较为简单但是繁琐。适合用来练习如何对付If套If的代码。
- String Calculator：练习需求不断变更的情况下如何写代码。一定要老老实实按照题目要求做一步再看下一步。
- LCD 和 Bank OCR：两个题目有类似的地方，比较适合练习如何分解单一职责。

- 生命游戏：经典的题目，对于设计测试和测试的顺序比较有挑战。
- 哈利波特：偏算法，有一定的难度。

网络资源

- 姚若舟老师的各种Kata视频
- TDD社区Kata接力
- Cyber Dojo 是一个非常好的刻意练习TDD的网站。想要进一步了解的可以看看这篇介绍，Cyber Dojo 设计者谈 Cyber Dojo——为了好玩执行代码
- Codewars 提供了很多题目，并且有由易到难的升级系统。相对于Cyber Dojo，它最大的优势是可以看到其他人的优秀解法。不足的是没有对于TDD流程的支持。