敏捷

批判性思考

陈硕：mf1632008
管登荣：mf1632020
顾必成：mf1632019
李隆隆：mf1632035

Contents

TDD

定义

TDD（测试驱动开发）是一种不同于传统软件开发流程的新型的开发方法。它要求在编写某个功能的代码之前先编写测试代码，然后只编写使测试通过的功能代码，通过测试来推动整个开发的进行。这有助于编写简洁可用和高质量的代码，并加速开发过程。[1]

测试是如何驱动开发过程的

测试驱动开发起源于 XP 法中提倡的测试优先实践[2]。测试优先实践重视单元测试，强调程序员除了编写代码，还应该编写单元测试代码。在开发的顺序上，它改变了以往先编写代码，再编写测试的过程，而采用先编写测试，再编写代码来满足测试的方法。在 XP1998 年出现之前，几乎没有关于让小的增量式的自动化单元测试驱动软件开发的设计过程概念。尽管缺乏公开的文档，许多开发人员可能非正式地使用了测试的第一种方法。肯特·贝克声称“他在还小的时候就在一本关于编程的书上学到了测试优先。它说，你写的程序就是获取输入然后输出结果，直到得到你期望的结果。”[3] 也许是最知名的敏捷方法，XP 常常与其他敏捷方法如 Scrum 组合。XP 建议使用 TDD 作为开发高质量的软件的一个组成部分。具有高度纪律性的 TDD 和简单的，轻量级的敏捷过程自然引起了一个有趣的冲突。潜在的 TDD 使用者经常表示关注编写和维护单元测试的时间和成本。虽然 Beck 承认没有必要为所有代码进行自动化单元测试，但他坚持认为，没有 TDD，XP 不能工作因为它使整个开发过程紧密联系在一起。[4]这种方法在实际中能够起到非常好的效果，使得测试工作不仅仅是单纯的测试，而成为设计的一部分。为什么这么说呢？

在编写程序之前，每个人都会先进行设计的工作。可能有些人的设计比较正式，绘制模型，编写文档。有些人的设计只是存在于脑海之中。且不论设计是精细还是粗糙，你都为随后的编码活动制定了一个标准。这个标准的明确程度和你的设计的细致程度有关。但应该承认，这个标准是不够细化的。因为你的设计不可能精细到代码级的程度。而标准不够明确的则会产生一些问题，例如，在编写代码的过程中，你还可能会发现原先的设计出现问题，从而中途改变代码的编写思路。这将会导致成果难以检验，进度难以度量。

既然以设计为导向的标准不够明确，不够具体。那什么样的标准才是合适的呢？只能是代码。因为代码是最明确、最具体的。所以测试优先的本质其实是目标管理。编写测试代码其实是在制定一个小目标。这个小目标非常的明确，它规定了你需要设计的类、方法。以及方法需要满足的结果。这些目标制定完成之后，你才开始编写代码来达成该目标。测试的目标要比设计的目标粒度更小，但是成本上却更为经济。其原因有四：

1. 细粒度的设计需要花费大量的成本，虽然有工具提供了代码自动生成的功能，但结果往往难以令人满意。所以，设计如果要做到和测试相同的粒度，成本不菲，如果粒度不够细，指导的意义又不够。
2. 减轻了测试的工作量。无论是否进行设计工作，测试工作都是不可避免的，先进行单元测试，可以减少后续的测试工作量。
3. 采用测试优先的过程中，设计的粒度较大。因为测试可以实现一部分的设计工作。这样，设计上可以节省一些工作量。例如，你不再需要将类图细化到每个方法。
4. 在编写测试代码上花费的成本，会在回归测试上得到回报。自动化测试的最大好处就是避免代码出现回归。两相权衡，编写测试的代价其实不高。

你也许会说，我既不进行如此精细的设计，也不事先编写测试代码，这样的成本不是最低吗？请注意，我的前提是在讨论高质量的软件设计。在一些规模较小或是开发人员能力极强的项目中，确实可以如此办理。但是对于强调质量的大项目，这种处于混沌状态的开发思路是不可取的。

测试优先是软件开发中一种细粒度的目标管理方法，通过明确的目标，推动软件开发的进行。在业界中，采用测试作为评价软件标准的做法是非常常见的。例如，sun 公司就专门设计了测试软件，对各个实现 J2EE 规范的产品进行测试。使用测试作为规范的最大好处就是明确、具体。同时，平均来说，80%的专业的开发者认为 TDD 是一个有效的方法并且

78%的人相信该方法能够提高开发者的生产率[5]。

使用测试代码建立目标，编写代码完成测试目标，再制定下一个目标，如此循环，构成了测试驱动开发的工作流程。

软件测试分类

单元测试

单元测试是典型的对代码逻辑的黑盒测试。在测试驱动方法中，不太强调白盒测试（绝大多数的白盒测试都是通过评审进行的）。这样做的好处是关注接口胜于关注实现，这是一种分析复杂软件的有效办法。

单元测试是开发人员的职责。一般来说，测试的编码最好由不同人来负责，避免出现盲点，以提高测试的有效性。但是单元测试的粒度很小，如果进行分工，沟通的成本会相当高。此外，采用测试优先的实践，对测试进行适当的培训，也能够有效的降低单个人的盲点范围。

单元测试可以加入到小组日构建中，也可以不加入。如果不加入，那么需要有一种机制来管理单元测试活动。

集成测试

集成测试的粒度和测试的范围要比单元测试大。就拿数据库测试来说，现在需要对一个业务对象进行测试，它需要用到持久化机制。在单元测试中，我们将不涉及数据库而单独对业务对象进行测试；但是在集成测试中，我们需要将数据库的数据一致性也纳入进来，所以测试包括数据库数据的建立，测试业务方法，使数据库恢复原状。

集成测试应该是日构建的重要组成部分，即日构建标准中的测试标准。最好将集成测试交给 QA 部门负责。QA 部门的精力可以放在使用或编写一些工具（Cactus 就是典型的集成测试工具），建立标准的测试数据，安排测试计划等活动上。

接受测试

有时候很难区分集成测试和接受测试。接受测试的关注点是用户。用户通过将数据输入系统来观察系统的输出。所以，了解用户的需要并将用户的需要转换为接受测试是接受测试中最关键的工作。接受测试处于测试过程的最后环节，是判断软件是否满足用户需求的试金石。毫不例外。接受测试也应该是自动化的。例如，HttpUnit 就是一个自动化接受测试的工具。另外，很多的专业测试工具提供的自动化的脚本测试工具也属于这个范畴。

TDD 的优势

TDD 的基本思路就是通过测试来推动整个开发的进行。而测试驱动开发技术并不只是单纯的测试工作。

需求向来就是软件开发过程中感觉最不好明确描述、易变的东西。这里说的需求不只是指用户的需求，还包括对代码的使用需求。很多开发人员最害怕的就是后期还要修改某个类或者函数的接口进行修改或者扩展，为什么会发生这样的事情就是因为这部分代码的使用需求没有很好的描述。测试驱动开发就是通过编写测试用例，先考虑代码的使用需求（包括功能、过程、接口等），而且这个描述是无二义的，可执行验证的。

通过编写这部分代码的测试用例，对其功能的分解、使用过程、接口都进行了设计。而且这种从使用角度对代码的设计通常更符合后期开发的需求。可测试的要求，对代码的内聚性的提高和复用都非常有益。因此测试驱动开发也是一种代码设计的过程。

开发人员通常对编写文档非常厌烦，但要使用、理解别人的代码时通常又希望能有文档进行指导。而测试驱动开发过程中产生的测试用例代码就是对代码的最好的解释。

快乐工作的基础就是对自己有信心，对自己的工作成果有信心。当前很多开发人员却经常在担心：“代码是否正确？”“辛苦编写的代码还有没有严重 bug？”“修改的新代码对其他部分有没有影响？”。这种担心甚至导致某些代码应该修改却不敢修改的地步。测试驱动开发提供的测试集就可以作为你信心的来源。

当然测试驱动开发最重要的功能还在于保障代码的正确性，能够迅速发现、定位 bug。而迅速发现、定位 bug 是很多开发人员的梦想。针对关键代码的测试集，以及不断完善的测试用例，为迅速发现、定位 bug 提供了条件。

原理

测试驱动开发的基本思想就是在开发功能代码之前，先编写测试代码。也就是说在明确要开发某个功能后，首先思考如何对这个功能进行测试，并完成测试代码的编写，然后编写相关的代码满足这些测试用例。然后循环进行添加其他功能，直到完全部功能的开发。

我们这里把这个技术的应用领域从代码编写扩展到整个开发过程。应该对整个开发过程的各个阶段进行测试驱动，首先思考如何对这个阶段进行测试、验证、考核，并编写相关的测试文档，然后开始下一步工作，最后再验证相关的工作。

在开发的各个阶段，包括需求分析、概要设计、详细设计、编码过程中都应该考虑相对应的测试工作，完成相关的测试用例的设计、测试方案、测试计划的编写。这里提到的开发阶段只是举例，根据实际的开发活动进行调整。相关的测试文档也不一定是非常详细复杂的文档，或者什么形式，但应该养成测试驱动的习惯。

原则

测试隔离。不同代码的测试应该相互隔离。对一块代码的测试只考虑此代码的测试，

不要考虑其实现细节（比如它使用了其他类的边界条件）。

一顶帽子。开发人员开发过程中要做不同的工作，比如：编写测试代码、开发功能代码、对代码重构等。做不同的事，承担不同的角色。开发人员完成对应的工作时应该保持注意力集中在当前工作上，而不要过多的考虑其他方面的细节，保证头上只有一顶帽子。避免考虑无关细节过多，无谓地增加复杂度。

测试列表。需要测试的功能点很多。应该在任何阶段想添加功能需求问题时，把相关功能点加到测试列表中，然后继续手头工作。然后不断的完成对应的测试用例、功能代码、重构。一是避免疏漏，也避免干扰当前进行的工作。

测试驱动。这个比较核心。完成某个功能，某个类，首先编写测试代码，考虑其如何使用、如何测试。然后在对其进行设计、编码。

先写断言。测试代码编写时，应该首先编写对功能代码的判断用的断言语句，然后编写相应的辅助语句。

可测试性。功能代码设计、开发时应该具有较强的可测试性。其实遵循比较好的设计原则的代码都具备较好的测试性。比如比较高的内聚性，尽量依赖于接口等。

及时重构。无论是功能代码还是测试代码，对结构不合理，重复的代码等情况，在测试通过后，及时进行重构。关于重构，我会另撰文详细分析。

小步前进。软件开发是个复杂性非常高的工作，开发过程中要考虑很多东西，包括代码的正确性、可扩展性、性能等等，很多问题都是因为复杂性太大导致的。极限编程提出了一个非常好的思路就是小步前进。把所有的规模大、复杂性高的工作，分解成小的任务来完成。对于一个类来说，一个功能一个功能的完成，如果太困难就再分解。每个功能的完成就走测试代码－功能代码－测试－重构的循环。通过分解降低整个系统开发的复杂性。这样的效果非常明显。几个小的功能代码完成后，大的功能代码几乎是不用调试就可以通过。一个个类方法的实现，很快就看到整个类很快就完成啦。本来感觉很多特性需要增加，很快就会看到没有几个啦。你甚至会为这个速度感到震惊。（我理解，是大幅度减少调试、出错的时间产生的这种速度感）

建立测试文化

测试驱动方法不是一个简单的方法论，它也不会和任何的方法论进行竞争。事实上，无论你的组织采用何种方法或过程，都可以从测试驱动中获利。因为它强调的是质量文化。把测试看作一项核心工作，测试同样需要重构，以及必须的文档。固定测试的目录组织和包组织。例如，一种较好的组织测试的方法是采用和源代码同样的包名，但处于完全不同的目录中。使测试成为日创建的核心步骤。测试是所有人的事情，而不仅是 QA 的事。

结对编程

结对编程可能是近年来最为流行的编程方式。所谓结对编程，也就是两个人写一个程序，其中，一个人叫 Driver，另一个人叫 Observer，Driver 在编程代码，而 Observer 在旁边实时查看 Driver 的代码，并帮助 Driver 编程。并且，Driver 和 Observer 在一起时可以相互讨论，有效地避免了闭门造车，并可以减少后期的 code review 时间，以及代码的学习成本。

这种要求是对一个人的心智、道德修养的更高要求。结对编程中，编码不再是私人的工作，而是一种公开的“表演”。程序员的代码、工作方式、技术水平都变得公开和透明，这也许是有些同学不喜欢这一方式的原因[1]。

结对编程技术

在传统开发过程中，每个开发人员负责系统的一部分开发任务，各自分工、互不干扰，而在结对编程的情景下，是两个开发人员结为一对，来共同完成同一个开发任务。但是，人与人之间的合作不是一件简单的事情——尤其当人们都早已习惯了独自工作的时候。

实施结对编程技术将给软件项目的开发工作带来好处，只是这些好处必须经过缜密的思考和计划才能真正体现出来。而另一方面，两个有经验的人可能会发现配对编程里没有什么技能的转移，但是让他们在不同的抽象层次解决同一个问题会让他们更快地找到解决方案，而且错误更少。因为两个程序员具有相同的缺点和盲点的可能性很小，所以当我们采用结对编程的时候会获得一个强大的解决方案，而这个解决方案恰恰是其它软件工程方法学中所没有的。

在结对编程模式下，一对程序员肩并肩地、平等地、互补地进行开发工作。两个程序员并排坐在一台电脑前，面对同一个显示器，使用同一个键盘，同一个鼠标一起工作。他们一起分析，一起设计，一起写测试用例，一起编码，一起单元测试，一起集成测试，一起写文档等。

在实际实施过程中，两人在同一台计算机面前进行编程活动，一个同伴使用鼠标和键盘来编码，另一个同伴观察代码并考虑设计问题。一方发现问题时，暂停编码工作，双方讨论解决。结对者的角色根据需要来不断调整和交换，包括与别的结对组交换成员。这种结对方式可以让每个成员对项目有一个整体的认识，并且有利于团队建立起良好的合作和学习氛围。极限编程的其他几项重要原则包括：频繁地小规模发布软件，简单设计，集体拥有代码和持续集成。

结对编程开发的成果保证了一部分代码至少有两个人熟悉，编码的同时也在被复查，既改善了复查的效果也在开发团队内部传播了知识，比传统的 Desktop Review 更有意义。原始的那种让程序员互相检查程序的方式是一种相当形式化的手段，其实效果不好，因为检查者无法像开发时那样完全融入程序的逻辑之中，很难从中发现业务逻辑错误，多数情况下根本都不会仔细去看。事实上在两个人同时编写程序的过程中，每个人的注意力都会比自己一个人编程时更加集中，考虑问题更加全面，互相提醒需要注意的事项。

当两个人共同开发时，充分的交流所带来的效率是传统纸张文档所不可比拟的，也更容易深入业务逻辑，发掘问题并拓展解决方案的思路，增进团队间的友谊。而且通过大量的交流和沟通，两个人在技术水平和业务能力上的提升都远比一个人闷着头干活儿快得多。不要吝音于和自己的 partner 交流，因为就算是经验老道的程序员也能从新手身上学到有价值的东西，开阔思路，对公司整体员工质素的提升也是很好的促进[2]。

结对编程保证了任何一行程序都有两个以上的人了解，也就是说对于项目每一个模块都有 buckup 人选，这对公司抵抗人员流动风险的能力有极大的帮助。由于参与结对的成员是经常在变化的(通常半天可以重新组队一次)，所以对于同一块程序，所有相关人员全部离职的可能性相当低。对于一个需要修改的程序，只要有程序员是熟悉它的，无论过了多久，也比一个“新手”做起来快，一个对此程序完全陌生的程序员相较于 backup 人选来说必然需要更长的时间去挖清楚其中的逻辑和实现风格，一定程度上增加了公司维护程序、升级程序的成本。

结对编程的利与弊

当然，凡事有利有弊，结对编程也不例外。有人垢病结对开发的效率问题，认为传统的两个人各自码代码的速度肯定比同一时刻只有一个人打字的速度要快得多，自然结对编程的效率就相对比较低下了。但从整体开发角度，开发过程中也许表面上会慢，其实由两个人同时开发所能避免的错误和返工将是非常客观的，对于经验并不老道的程序员来说尤其如此，再算上维护的时间和成本，应当说结对开发的效率是只高不低的[3]。

每人在各自独立设计、实现软件的过程中不免要犯这样那样的错误。在结对编程中，因为有随时的复审和交流，程序各方面的质量取决于一对程序员中各方面水平较高的那一位。这样，程序中的错误就会少得多，程序的初始质量会高很多，这样会省下很多以后修改、测试的时间。具体地说，结对编程有如下的好处：

- 在开发层次，结对编程能提供更好的设计质量和代码质量，两人合作能有更强的解决问题的能力。
- 对开发人员自身来说，结对工作能带来更多的信心，高质量的产出能带来更高的满足感。
- 在心理上，当有另一个人在你身边和你紧密配合，做同样一件事情的时候，你不好意思开小差，也不好意思糊弄。
- 在企业管理层次上，结对能更有效地交流，相互学习和传递经验，能更好地处理人员流动。因为一个人的知识已经被其他人共享。

在结对编程中，任何一段代码都至少被两双眼睛看过，两个脑袋思考过。代码被不断地复审，这样可以避免牛仔式的编程。同时，结对编程避免了“我的代码”还是“他的代码”的问题，使得代码的责任不属于某个人，而是属于两个人，进而属于整个团队，这样能够帮助建立集体拥有代码的意识，在一定程度上避免了个人英雄主义。结对编程的过程也是一个互相督促的过程，每个人的一举一动都在别人的视线之内，所有的想法都要受到对方的评价。由于这种督促的压力，使得程序员更认真地工作。结对编程“迫使”程序员必须频繁地交流，而且要提高自己的技术能力以免被别人小看。

结对编程的演化

由于现在开发环境的变更发展，出现了演化变形的结对编程形式——远程结对编程和乒乓结对编程。远程结对编程，又称虚拟结对编程或分布式结对编程，是指两个程序员不在同一地点，通过协同编辑器，共享桌面，或远程结对编程的 IDE 插件进行的结对编程。远程结对编程破除了传统结对编程对地域上的限制，更加适应大环境的发展，但同时还引入了一些在面对面的结对编程中不存在的困难，例如协作的额外时延，更多的依赖“重量级”的任务跟踪工具，而不是“轻量级”的索引卡片，以及没有口头交流导致的在类似谁“控制键盘”问题上的混乱和冲突[4]。乒乓结对编程是指观察者编写失败的测试用例，驾驶者修改代码以通过该用例，观察者编写新的单元测试用例，等等。这个循环持续到观察者不能写出失败的测试用例，但是实际实施过程花费时间会比估计的计划要超出很多。

高适应性

适应性及其必要性

敏捷方法有时候被误认为是无计划性和纪律性的方法，实际上更确切的说法是敏捷方法强调适应性而非预见性。适应性的方法集中在快速适应现实的变化。当项目的需求起了变化，团队应该迅速适应。这个团队可能很难确切描述未来将会如何变化.[1]。敏捷软件开发方法过程是敏捷开发组织的复杂适应性的过程，所以敏捷开发组织对于系统特性具有高适应性。这里提及的适应性是指软件使不同的系统约束条件和用户需求得到满足的容易程度。它要求软件尽可能适应各种硬软件运行环境，以便软件的推广和移植[2]。敏捷软件开发方法接受用户需求的不断变化并且能迅速做出响应，以用户需求为导向，以核心价值观为指导，以原则为约束，采用灵活可靠的技术，迭代开发，从而达到最终目的。迭代开发就是将用户需求分割成细小的用户 story，规定迭代周期并阶段性循环实现[3]。下文将通过敏捷软件开发与其他传统开发方法的对比，敏捷软件开发方法的特性以及从复杂适应系统（Complex Adaptive System，CAS）理论着手分析其高适应性[4]。

敏捷软件开发对比其他的方法

对比迭代方法

相比迭代式开发两者都强调在较短的开发周期提交软件，敏捷方法的周期可能更短，并且更加强调队伍中的高度协作。

对比瀑布式开发

两者没有很多的共同点，瀑布模型式是最典型的预见性的方法，严格遵循预先计划的需求、分析、设计、编码、测试的步骤顺序进行。步骤成果作为衡量进度的方法，例如需求规格，设计文档，测试计划和代码审阅等等。

瀑布式的主要的问题是它的严格分级导致的自由度降低，项目早期即作出承诺导致对后期需求的变化难以调整，代价高昂。瀑布式方法在需求不明并且在项目进行过程中可能变化的情况下基本是不可行的。

相对来讲，敏捷方法则在几周或者几个月的时间内完成相对较小的功能，强调的是能将尽早将尽量小的可用的功能交付使用，并在整个项目周期中持续改善和增强。

有人可能在这样小规模的范围内的每次迭代中使用瀑布式方法，另外的人可能将选择各种工作并行进行，例如极限编程。[1]

敏捷软件开发方法特性

敏捷软件开发方法区别于传统方法的特性之一是拥抱快速变更、具有适应性而非预设性。传统方法如计划驱动开发方法等试图对一个软件开发项目在很长的时间跨度内做出详细的计划，然后依计划、合同、规格说明等文档进行开发，依赖于显式的文档化知识。

其实追溯传统方法的来源，其基本思路通常是从其他工程领域借鉴而来的，比如土木工程等。在这类工程实践中，通常非常强调施工前的设计规划。只要图纸设计得合理并考虑充分，施工队伍可以完全遵照图纸顺利建造，并且支持施工队之间的并行施工[5]。但是，软件开发与传统的工业工程有着巨大的差异。软件设计是一个复杂度极高的工程，而且土木工程师在设计时所使用的模型是基于多年的工程实践的，一些设计上的关键部分都是建立于坚实的数学分析之上，而在软件设计中，没有类似的基础，实践经验也积累不足。面对这样巨大的差异，仍旧生搬硬套传统工程借鉴来的开发方法自然存在许多固有缺陷，从而导致软件开发的效率低下、拒绝变更等后果。加之，软件的设计之所以难以实现，问题在于软件需求的不稳定，从而导致软件过程的不可预测、变更频繁。但是传统的控制项目的模式都是针对可预测的环境的，在不可预测的环境下，就无法使用这些方法。可是我们必须对这样的过程进行监控，以使得整个过程能向我们期望的目标前进[6]。于是需要引入敏捷软件开发这种具有适应性的方法，该方法使用反馈机制对不可预测过程进行控制。而敏捷软件开发方法本来就是为了更好地适应变化而发展出来的，甚至能允许改变自身来适应变化。

复杂适应系统理论

霍兰 J(HollandJohn)于 1994 年提出复杂适应系统 (CAS)理论，迅速引起学界关注，被尝试用于观察和研究各种不同领域的复杂系统，成为当代系统科学引人注目的一个热点。笔者于 1997 年和 2000 年两次访问圣菲研究所 (SantaFeInstitute)，现作一简要介绍。

1.CAS 理论是现代系统科学的继续和发展

CAS 理论是二十世纪几代科学家不断深入研究，对于复杂系统的日益全面理解与认识的成果之一。 20 世纪 30 年代，当贝塔朗菲重新举起系统论旗帜，向片面强调还原论、忽视系统整体性的观点挑战，为“整体大于其各部门之和”的正确思想进行申辩时，他还缺乏具有说服力的论据，给人以空泛的感觉。[7]首先给他以实际支持的是维纳。他打破了只注意分割、忽视综合的偏颇，以信息、反馈和控制的新观念研究系统行为，总结出跨越工程与生物界的一般性规律———控制论。[8]

控制论的迅速传播和在实践中的成功，使 20 世纪 50 年代成为“控制论的时代”，系统工程思想广为传播，控制论方法被用到自动控制、工程管理以至社会经济等许多领域[9，10]。二战中以及战后的一系列重大工程和重大科学进步为系统工程思想提供了有力的支持，一时间，谈系统、谈控制成为科学界的时尚，形成系统科学的第一个高潮。控制论在社会经济领域中则不那么成功，以至到了 20 世纪 70 年代，有人开始哀叹“控制论时代的终结”。这正是系统科学需要进一步深入的前兆。这一时期所说的“系统”，是以机器为背景的，部分是完全被动的、死的个体，其作用仅限于接收中央控制指令，完成指定的工作。任何其他动作或行为都被看作只起破坏作用的消极因素 (噪音)，在应当尽量排除之列。这既保证了它在工程领域的成功应用，也决定了它在生物、生态、经济、社会这类以“活的”个体为部分的系统中必然遇到困难。我们把一这阶段的观念称为第一代系统观。

20 世纪 70 年代兴起的耗散结构理论[11]和协同学[12]提出第二代系统观。普利高津和

哈肯所说的“系统”，具有两个新特征 :第一，元素数量极大，一般都到 1020 以上，致使“我推你动”的控制和管理方式成为不可能 ;第二，元素具有自身的、另一层次的、独立的运动，使整个系统不可避免地具有统计性和随机性。从这两点出发，第二代系统观拓宽了控制概念，引伸了随机性和确定性对立统一的思想，讨论了自组织涨落、相变等新的概念，对系统的理解深入了一大步。这里说的“系统”所隐含的背景已经不是人造机器，而是某种热力学意义下的系统。

然而，当人们试图把第二代的系统思想应用于经济、社会等系统时还是不能令人满意的。原因在于，虽然个体(或元素)可以有“自己的”运动，这种运动在一定条件下对整个系统的进化起着积极的、建设性的作用，然而这种运动仍然是盲目的、随机的，就象布朗运动那样。个体没有自己的目的、取向，不会学习和积累经验，不会改进自己的行为模式，一句话，它还是“群氓”，不是真正的“活的”主体。[13]

20 世纪 90 年代以来，中外学者不约而同地把注意力集中到个体与环境的互动作用上。我国学者提出“开放的复杂巨系统”的概念[13];澳大利亚学者通过大量例证，研究了生物界的涌现规律[14];许多学者从学习、认知等方面，对于知识的表达和获取进行探索，提出人工神经网络、基于案例的推理等许多新概念。由此形成第三代系统思想，核心是强调个体的主动性，承认个体有其自身的目标、取向，能够在与环境的交流和互动作用中，有目的、有方向地改变自己的行为方式和结构，达到适应环境的合理状态。在这方面，以圣菲研究所为中心的、关于复杂适应系统的研究，是一个十分引人注目的方向。

2.CAS 理论的基本内容与观点

2.1 具有适应能力的主体

CAS 理论最基本的概念是具有适应能力的主体(AdaptiveAgent)，简称主体。它不同于早期的系统科学用的部分、元素、子系统等概念，部分或元素完全是被动的，其存在是为了实现系统所交给的某一项任务或功能，没有自身的目标或取向，即使与环境有所交流，也只能按照某种固定方式作出固定的反应，不能在与环境交互中“成长”或“进化”。主体则随着时间而不断进化，特点是 :一能“学习”，二会“成长”，这就使得 CAS 理论与以往的系统观有了根本性差别。

2.2 主体和环境的互动作用

主体的“活”性体现在它与环境的互动关系中，理论基础是最简单的刺激———反应模型。生活在特定环境中的主体不断从环境接受刺激，并根据经验作出反应。反应的结果可以是成功的———达到预期目标，也可能失败———没有达到预期目标。CAS 理论的独特之处在于主体可以接受反馈结果，据之修正自己的“反应规则”。霍兰用他的遗传算法把反映规则表达成“染色体”———一种包括刺激与反应对应规则的字符串，通过引入“适应度(fitness)作为表达“染色体”所表示的反应规则与环境相符合的程度。主体能够根据成功还是失败的反馈信息，修改“染色体”的“适应度”。这一观点与传统的人工智能、知识

库的概念完全不同。传统的知识管理把一致性、无矛盾作为基本要求，使之成为固定的、僵化的机制，而不是“活”的、具有生长和发展前途的机制。CAS 的理论突破了这种框架，能够更真实地描述、观察、理解这一类活的复杂系统。

2.3　个体的演变过程———CGP

CGP(ConstrainedGeneratingProcedure)可以译为受限生成过程，反映在一定环境约束条件下，主体发展和进化的一般规律。霍兰以棋类游戏、数字系统、神经系统等来自不同领域的复杂系统为例，归纳了 CGP 的若干普遍性质和规律。运筹学在一定约束条件下寻找最优解，只是一种静态条件下的算法，CGP 展示的是一幅活生生的、变化中的、充满新奇和意外的进化过程。[1 0]这正是系统观从研究固定的、死的元素走向研究活生生的、成长中的主体的契机。

2.4　从个体的演化到系统的演化———ECHO 模型

基于个体演化过程，加上“资源”(Resource)和“位置”(Site)的概念，霍兰 把个体演化和整个系统演化联系起来，形成了 ECHO 模型 (可以译为“回声”模型)。这种宏观和微观统一的、有机的、内在的结合，是 CAS 理论引人入胜的又一个特点。许多学科 (例如经济学)常常为微观和宏观规律的不一致和冲突所困扰。过去习惯于用统计规律把二者联系起来，但无论是生物学还是经济学，都有许多事实表明，事情没那么简单。统计规律只是层次之间的桥梁之一，不是唯一的，甚至可以说，多半不是主要的桥梁。因此，如果说自组织在第二代系统观中是主要的主题词之一，涌现则成为今天的系统科学最引人关注的议题之一。ＥＣＨＯ模型反映了这一点。

3. CAS 理论的特点与应用

与关于复杂系统的其他理论相比，CAS 理论有三个显著特点 ：

3.1 CAS 理论恢复了古代系统思想强调的活力观

自从现代系统科学兴起以来，人们强调的主要是“整体观”，对古代系统思想关于活力的观点注意不够。“活力论”认为，物质自身具有活力，不断运动和变化，发展变化不只是由外部原因推动的。典型代表是卢克莱茨的《特性论》[1 1]虽然恩格斯一再强调物质与运动不可分，说“没有运动的物质和没有物质的运动同样不可想象”，我们在相当长的时间内还是忽视了这个重要观点。即使谈系统，也有意无意地把元素或部分看做死的对象，是整体的“齿轮和螺丝钉”。这就导致前两代系统方法在处理社会经济系统时的困惑和无力。CAS 理论在这一点上的突破，使它具有了与以前的理论根本不同的、新的洞察力。

3. 2 CAS 理论对于宏观与微观之间的联系，给出了新的认识角度———涌现

涌现是在微观主体进化的基础上，宏观系统在性能和结构上的突变。这种突变在以往的观念中是难以认识和控制的，也不是用统计等传统方法所能完全说明的。CAS 理论提供了新的思路和视角，对于我们认识和解释经济、社会、生态、生物的许多现象以启发，开辟了新路。

3. 3 CAS 理论具有鲜明的可操作性，为进一步研究创造了十分有利的条件

CAS 理论的产生与遗传算法密切联系在一起，充分吸收了计算机科学与技术的成果 (特别是人工智能和计算机模拟的成果)，具有鲜明的可操作性。这一点通过 SWRAM 的开发与推广得到充分体现。ＳＷＡＲＭ是一个公开的软件平台，任何研究者都可以从 Internet 网上下载并且应用它来建模和进行模拟。从计算机技术的角度来看，使用 SWARM 建模需要掌握 OjectC 或 JAVA 语言，关键工作在于 CAS 模型的建立。２０００年 8 月，在圣菲市举行的中美复杂系统建模研讨会上，以蒋正华教授为首的中国代表团提交了六篇论文，反映了中国学术界在经济、化学、生物、计算机技术等方面研究和应用系统科学理论的成果。美国和其他国家的学者也发表了有关环境、社会学、历史学等方面的成果。

4. CAS 理论的启示和发展前景

4. 1 以涌现为特征的新的演化观

系统科学的一个根本问题是 :系统在其发展过程中是越来越简单，还是越来越复杂 ?立足于还原论的传统观点，把热力学第二定律片面夸大，描绘了一幅从复杂到简单的发展图景。几十年来，系统科学致力于从理论上驳倒这种观点。人们先是从信息控制角度去理解复杂性，考查系统的发展和运动，开始把发展看做具有主动性、可控制的过程。随后，关于微观扰动的非线性放大及由此产生的关于自组织现象的研究，开始掌握从简单到复杂，以及产生结构与差别的具体途径。但究竟自组织是怎样发生的，还只有热力学、激光等少数几个相对简单的例证，而且基本立足于随机性和统计规律的解释。与经济社会生物等真正复杂的“活”系统相比，这种理解还是非常不够的。

CAS 理论通过把系统元素理解为活的、具有主动适应能力的主体，引进宏观状态变化的“涌

现”(Emergence)概念，使从简单中能够产生复杂的观念得到有力支持。在实际的系统演化过程中，确实存在从无结构到有结构，从单一到多样，从对称到非对称的发展趋势。客观事物以及整个世界的发展趋势是多种多样的，既有从复杂到简单的“瓦解”趋势，也有从简单到复杂的“涌现”趋势，二者相反相成，此长彼消，构成丰富多彩、变化万千的大千世界。这种新的演化观正在得到越来越多的科学发现的支持。总之，CAS 理论的重要贡献在于 :为系统科学一直试图加以论证的演化观，提供了具有充分说服力的支持和证明。

4. 2 对层次概念的深入认识

层次是系统科学的重要概念。在某种意义上讲，系统科学是研究层次之间相互联系、相互转化规律的。所谓局部和整体、元素和系统、个体与群体，实际上就是上下两个层次之间的关系。迄今为止，关于层次之间的过渡与转化，主要依靠的是统计方法，即使在自组织理论 (如耗散结构理论与协同学)中，统计方法也起着决定性作用。不过单靠随机性、概率和统计，还无法解释世界的演化与宏观尺度上的涌现，需要寻找别的机制与途径。ＣＡＳ理论在这方面有所突破，通过承认个体的主动性，为系统演化找到了内在的、基本的动因，为理解层次提供了新的视角。

4.3 随机性与确定性的统一

在系统科学发展中，这一问题已经讨论过多次。耗散结构理论关于涨落的观点就是一个明显的例子。CAS 理论进一步研究了随机性和确定性如何有机地结合的机制与途径。随机性体现在以下几个方面。首先是环境刺激的随机性，主体事先不知道会接收到什么样的刺激信号，只是做好准备接受刺激。其次是反应的随机性，由于反应规则不唯一，选择哪一条规则作出反应有一定随机性。新规则的产生 (交叉与突变)也受随机因素支配。所以，CAS 理论充分考虑了随机性。但总的理论框架是十分确定的，虽然没有规定确定的演化目标 (事先谁也不知道)，但在一定环境刺激下，它向哪个方向发展是确定的。CAS 理论不是简单地靠某个随机性算法来体现，而是全面考虑了随机性的作用，尽量在演化机制的各个环节上，有规律地引入随机因素。用 GA 算法进行系统研究的实践表明，这种结合能够更好地描述客观现象，既有随机性，又有确定的发展方向，而且收敛速度比单纯的蒙特卡洛方法快得多。这也许能对于生物进化作出更符合实际的说明。

4.4　对控制与管理的新理解

最早的控制思想是机械的控制观，即我指挥，你动作;所有决策都由统一的中央处理器来进行;各个部分没有自己的目的和能动性，信息传递的速度和准确性、过渡过程的稳定性、决策算法的有效性等成了决定的因素。这种早期控制观的背景是自动控制系统。

当元素个数迅速增加时，以机械为背景的控制观就显得无能为力了。仅就信息传递的速度和准确性而言，这样的系统的崩溃和瓦解是不可避免的。故随着系统规模不断扩大，不得不引入随机因素，用各种方法对待和处理随机性，在随机环境中实现控制。最常用的是统计方法和模糊信息处理 ;最成功的例子是蒸汽机，它立足于热力学和统计规律，使大量分子无规则的热运动，通过巧妙的机制成为推动活塞运动的动力。基本思想是 :造成一定的环境 (或势)，使大量无规则运动的总效果达到某个预定目标。这是基于热力学的第二代控制观。

CAS 理论为第三代的控制观提供了背景。在这里，环境 (或势)继续发挥主要作用，但不是简单地通过统计规律，而是通过影响个体行为规则起作用。

控制不是通过“我推你动”的方式实现，而是遵从个体自身适应和变革的规律，通过环境“引导”或“诱使”个体改变自己的功能和行为规则，以达到客观控制的目标。

4.5 从平衡论到对策论

回顾一下我们以前对系统控制的基本看法及做法。以经济为例，以前总认为存在一种客观的、不变的平衡状态，是最理想的、最佳的系统状态。一旦系统偏离这种状态，就应当通过某种控制使它或迟或早回到这种状态，也许是通过政策调节 (看得见的手)，也许是通过

市场规律 (看不见的手)。这种看法与做法常常导致意想不到的结果。从思想方法上看，出现这种情况的原因之一在于没有考虑到个体是“活”的，忽略了个体的能动性和心理因素。另一方面，处于动态过程中的复杂系统往往很难确定什么是最佳的平衡状态，甚至是否有这样一种平衡状态本身就值得商榷。

CAS 理论承认个体的主体性，必然带来从平衡论到对策论的思想转变。既然主体是“活”的，当某个环境因素发生变化时，其反应和行为也必须发生改变，而不会“墨守成规”。所以，所谓“上有政策、下有对策”是必然的，不能回避的。问题在于如何因势利导，如何在考虑这种客观情况的前提下，达到预定的控制目标。

CAS 理论提示我们，平衡论的思想框架需要补充和修正，对策论的思想应当进入我们的方法库。熟悉经济学，特别是信息经济学的读者一定能联想到“委托———代理”模型，是局部利益和全局利益对立统一的例子。这里也有许多需要研究的问题，如个体和全局之间的对策应当具有怎样的数学模型。

然而，从平衡论转到对策论的思路，应当说是一个必然的发展趋势。CAS 理论提出不久，许多基本理论问题还在研究中，如建模机制与步聚，与人工神经网络等的关系，关于学习和适应的算法等等。应用研究更是刚刚开始。可喜的是，我国学术界已经开始介绍和了解 CAS 理论，在这方面和国外差距并不大。在应用方面，国内一些单位也已经开始不同角度的研究工作。例如，中国人民大学经济科学实验室已经在深入研究 SWARM 平台的基础上，利用 SWARM 和其他软件建立了一批基于我国实际的经济模型，开始为我国宏观经济决策作出贡献。

拥抱变更

依赖：实现对需求的依赖

依赖是两个元素之间的一种关系，其中一个元素变化，导致另外一个元素变化。就好比业务需求和业务实现之间的关系，业务需求发生变化，其业务实现也会发生变化。Kent Beck 和 Martin Fowler 在介绍极限编程（eXtreme Programming）时，最先提到的就是要拥抱变化。这基本上代表了敏捷阵营对于变化的一种态度，那就是不拒绝，而且还要主动求变。有一句经典的论断说得好：“在软件开发领域中，唯一的不变就是变化。”其实，不仅仅是软件开发领域，世间万事万物都处在永恒的变化之中，这是宇宙的基本规律。以来是不可避免的，重要的是如何务实的应对变化。

依赖和变化紧密联系在一起的概念。由于依赖关系的存在，当变化在某处发生时，影响会波及开去造成很多修改工作，这就是依赖的危害。变化是始作俑者,依赖是助纣为虐，我们可以不去拥抱变化吗？不可以。未来将越来越不可预测，是新经济最具挑战性的方面之一。商务和技术上的瞬息万变会产生变化。这既可以看作要防范的，也可以看作应该欢迎的机遇，既然变化不可避免，就是处理好依赖关系，影响波及范围尽量减少。

秉承敏捷宣言的精神（个体与交付重于过程和工具；可用的软件重于完备的文档；客户协作重于合同谈判；响应变化重于遵循计划），我认为，敏捷开发大致应该体现如下的思想：拥抱变化、自我组织、简单最好、客户至上、有效沟通、精益求精。

拥抱变化

传统的软件开发过程由于过分强调文档的完整，重视与客户的谈判与签订合同。所以开发团队最希望的一件事情是冻结需求规格说明书。只要双方签字，需求就确定下来，不可更改。若要更改，必须经过需求变更委员会，走非常严格的需求变更流程。如果软件开发真能如此，倒也算是我们开发人员的幸事。但现实总是残酷的，需求总是会变化。变化的原因有以下几种：

1）业务发生了变化；

2）客户对业务的理解发生了变化；

3）需求分析人员对需求的理解出现了偏差，需要修正。

对于第一点，或许我们还能够根据合同与客户讨价还价，而对于后两点，尤其是第三点，我们显然是不可拒绝的。而敏捷方法则要求团队随时响应客户的需求，针对变化给出相应的解决方案。

1.拥抱变化-技术方面

如何拥抱变化呢？我想可以通过如下方式来实现：

1）现场客户

很多开发团队并不喜欢客户对他们指手画脚，甚至认为他们不停提出的需求变化让他们疲于应对。但现场客户给团队开发带来的益处还是要远远超过他带来的坏处。无论团队中聚集了多么权威的领域专家，但真正了解客户需求的还是客户自己。也许他们很难用语言来表述自己的想法，但有了和现场客户的及时沟通，我们才能够在发生变化的初始就能够获得第一手的资讯。如果事情总要发生，早解决绝对比晚解决要好，而且要好得多。

如果在开发中，没有办法让客户成为团队的一员，那么我们也应该指定一位客户代表，退而求其次，也应该在团队中指定一位业务专家负责业务事宜，也就是 Scrum 中 Product Owner 的角色。总之，我们需要在项目开发中，能够让开发人员在对需求理解发生困惑时，能够明确地知道由谁来负责。而一旦需求发生变化，也必须有专门的角色负责向整个团队或者相关人士传达。至于业务功能的优先级和重要程度排定，也必须由这个角色指定。

2）定期迭代和小版本交付

敏捷宣言遵循的原则之“我们最重要的目标，是通过不断地及早交付有价值的软件使客户满意”。这本质上价值驱动在敏捷开发中的体现，定期迭代和小版本交付两者则是实现价值驱动的方法，面对软件领域中复杂多变的变化，变更，传统开发模式并不能有效应对这些变化，它们过分关注于需求文档的成熟和明确，企图冻结需求变化，冻结一切可能的变更。

不仅要定期迭代，而且要尽可能地让迭代周期短，并及时交付可以工作的小版本发布。敏捷方法绝对不可闭门造车，因为需求总是可能存在二义性，且需求总是处于不断的变化中。若能定期交付一个可以工作的小版本，一方面可以给与开发团队和客户以信心，另一方面也有助于我们及时获得客户的反馈。

敏捷开发方法强调适应性而非预见性，它强调当项目需求发生变化时，团队应该快速适应。这主要靠频繁地小规模发布软件，即短迭代完成的，将尽量小的可用功能交付使用，并在整个项目周期中持续改善和增强。

在传统软件开发过程中，测试往往在后期进行，主要目的是为了发现缺陷。在敏捷的

实践中，却强调由测试驱动代买的编写，而短迭代的开发必然有利地驱动了由测试到代码的开发过程，测试被认为是一种有效的反馈，用测试反馈来监控项目的进展和知道项目的开发情况。

3）持续改进

开发过程总是会出现错误，无论是开发方法、技能，还是团队管理与团队合作。持续地改进我们的开发方法、管理方法与开发过程，才能够及时而有效地解决错误，避免重蹈覆辙。

在持续改进的过程中，要注重测试驱动的价值。测试不仅仅能够测试软件的正确性、安全性、性能，严格地进行软件测试还能体现该阶段的项目开展的情况，需求应对的有效性，该阶段工作的总结。首先软件测试结果能够帮助项目管理者监控项目的发展进度，清晰地了解到项目进行的情况，为下一迭代的项目计划设计提供基础和经验。大型项目中，由于项目的规模和复杂的庞大，项目团队的人员数量庞大和位置分布的差异严重影响了项目管理者对项目进展的掌控。再者每个迭代的测试结果不仅仅能够让顾客对自己即将接受的产品充满信任，同时也会对自己的需求进行一次明确和审核，对需求进行改进，促进顾客对软件产品的反馈，保证了顾客的参与度和软件产品的成功。最重要的是每次迭代的软件测试都能够让软件团队自我审视，矫正软件开发的方向，树立里程碑，激励团队成员改进自己的工作，保持软件开发团队敏捷开发所具有的特性。

拥抱变化-项目管理方面

在敏捷软件开发领域中，对人的主观能动性的重视远远高于传统软件开发模式。拥抱变化对团队建设和个人能力都提出了新的需求。

作为个人，拥抱变化的的方式就是与时俱进，给自己投资，学习新技术和新的开发方法很重要，同时还要摒弃陈旧过时的开发方法，懂得放弃。具体有以下几个方法可以尝试：

1. 保持一生学习的观念，软件技术变更极其迅速，增强自己的眼界，知识能力。这是自己能够适应敏捷开发的员工要求。
2. 了解最新的技术行情。随时引入合适的解决方法，将所学用到现实工作中。
3. 参加各种技术沙龙。看看别人如何思考和解决问题的，相互学习相互进步
4. 对于自己的 idea，要勇于去实践和调研，不要眼高手低，增强自己的自主开发能力。
5. 对软件领域的技术变更保持关注和学习，为以后顾客提出技术变化需求做好知识储备，个人的知识和能力储备能够使自己在敏捷开发团队中发挥自己的能力。
6. 对知识刨根问底，真正搞明白，增强敏捷团队的成员自我学习和能力挖掘。不要只停留在表面，搞清楚问题的真正根源。

对于团队上面的同样适用，但是有一些其他需要关注的。

1. 把握好分享的机会。普及陌生的技术术语和概念，从而提高沟通效率；同时将自己所学的内容，通过分享避免忘记；将新鲜的内容引入到团队中。一个学习型的团队才是好的团队。
2. 为团队提供清晰的前进方向。
3. 懂得放弃陈旧的技术，积极学习新技术。接受新的技术，才能有机会与时俱进。
4. 面对团队的问题依然要刨根问题，不是面子问题，而是团队问题。这绝对不是讲面子的时候，也绝对不是不懂装懂的时候，更不是比比水强的时候。而是找到问

题根源，千里之堤溃于蚁穴，写作过程中蛛丝马迹都要提高警惕。没有一成不变的方法，

5. 积极收集反馈：反馈需要是多方面的：用户反馈，团队成员反馈，测试代码的反馈等等。通过这些反馈才能逐渐驾驭项目。
6. 拆分任务，使它能够被尽快实现，小而可达的目标可以让人全速前进。
7. 天下武功唯快不破。

参考文献

[1] 邹欣.构建之法——现代软件工程.人民邮电出版社.2014.9.

[2] 钟扬,刘业政,马向辉.小团队结对编程实践研究和重构[J]. 计算机技术与发展. 2007(11)

[3] 杨涛.杨晓云(译者)，结对编程技术，机械工业出版社，2004.1

[4] 臧正军. XP 结对编程研究、改进和实践[D]. 东北师范大学 2008

[5] 必应学术，http://cn.bing.com/academic/profile?id=4478048&v=fos_preview

[6] David Janzen,Test-Driven Development: Concepts, Taxonomy,and Future Direction,IEEE Computer Society,1-2

[7] K. Beck, “Aim, Fire,” IEEE Software, Sept./Oct. 2001, pp. 87-89.

[8] K. Beck, Extreme Programming Explained: Embrace Change, Addison-Wesley, 1999.

[9] Boby George, Laurie Williams,An Initial Investigation of Test Driven Development in Industry,4-5

[10] 维基百科

[11] https://zh.wikipedia.org/wiki/%E6%95%8F%E6%8D%B7%E8%BD%AF%E4%BB%B6%E5%BC%80%E5%8F%91

[12] 王海鹏(译者),基于组件开发,人民邮电出版社,2003.9

[13] 崔康，Agile Developer 创始人谈敏捷适应性，2013

[14] 杨小东,徐琳. 敏捷组织的复杂适应性及其行为模式研究[J]. 微计算机信息. 2012(02)

[15] 周莹莹. 敏捷软件开发技术研究[D]. 长春理工大学 2006

[16] Alistair Cockburn，Agile Software Develpment，Addison Wesley/Pearsno，2003.11

[17] Scrum 敏捷项目管理实战;书籍作者:(美)肯·施瓦伯著，李国彪、孙媛译;清华大学出版社

[18] 崔能辉. 敏捷型需求工程实践探索,拥抱变化[J]. 计算机光盘软件与应用, 2012(23):127-128.

[19] 李晓慧. 敏捷开发,拥抱变化[J]. 中国经济和信息化, 2008(24):45-46.

[20] 敏捷开发思想之拥抱变化 http://wayfarer.blog.51cto.com/1300239/280167