实验十三 软件测试

实验目的：

1. 了解软件测试
2. 深入理解白盒测试和黑盒测试

3. 熟悉单元测试工具

实验内容：

阅读下面软件测试相关资料（或查阅其它相关资料），了解软件测试的基本概念、主要技术、重要挑战等

Software Testing-A Research Travelogue (2000–2014).pdf

Fuzzing tesing.pdf

**软件测试的流程**  
1 需求分析  
2 测试计划  
3 测试方案  
4 测试用例  
5 测试执行  
6 测试报告

软件测试的概念

1）软件测试的定义：

（1）在规定条件下运行系统或构件的过程，观察并记录结果做出评价

（2）分析软件项目的过程，检测bug并评估软件项目的特性

2）软件测试的目的：

（1）测试的目的是证伪----发现程序中各种各样的错误

（2）测试就是建立一种信心，确信程序能够按预期的设想运行---在规定的环境中运行软件的所有功能

软件测试是为了保证软件质量

一个好的测试用例在于它能发现至今未发现的错误

一个成功的测试用例是发现了至今未发现的错误的测试

3）软件测试的原则：

（1）应当把  今早的和不断的进行软件测试 作为座右铭

（2）测试用例应由输入数据和对应的预期输出数据组成

（3）程序员应避免测试自己的程序

（4）在设计测试用例时，应包括合理的和不合理的输入条件

（5）充分注意测试中的集群现象

（6）严格执行测试计划，排除测试的随意性

（7）应当对每一个测试结果做全面的检查

（8）妥善保管相关文档

软件测试质量（ISO）

软件测试质量就是 ”软甲明确的和隐含的定义与需求相符的程度“

明确的需求是：

软件符合明确叙述的功能和性能的需求、文档中明确描述的开发标准

隐含的需求是：

所有专业开发的软件都要具有的隐含特征的程度

软件测试对象

1 根据软件的定义，软件包括程序、数据、文档，所以软件测试并不仅仅是程序测试。软件测试贯穿在整个软件生命周期中

2 由于在整个软件生命周期中，各个阶段有不同的测试对象，形成了不同的开发阶段的不同类型的测试。需求分析、概要设计、详细设计以及程序编码等各阶段产生的文档，包括需求规格说明、概要设计规格说明、详细设计规格说明以及源程序，都是作为 “软件测试”的对象。

软件测试分类

1 按照开发阶段划分软件测试：

单元测试、集成测试、系统测试、验收测试

2 按照测试实施组织划分软件测试：

开发方测试、用户测试（Beta测试）、第三方测试

3 按照测试技术划分：

白盒测试、黑盒测试、灰盒测试

软件测试方法和技术的分类与软件开发过程相关联，它贯穿了整个软件的生命周期。

软件测试风险

1 软件测试中的软件风险分析是根据预测软件将出现的风险，制定软件测试计划并排列优先等级，风险分析是对软件中潜在的问题进行识别、评估和评价的过程。

2 风险也包括进度风险、质量风险、人员风险、变更风险、成本风险等。

软件测试工程师的职责

1 配置测试环境

2 执行软件测试

3 报告软件缺陷

4 更新缺陷报告内容

5 验证修正的缺陷

6 报告测试状态

7 完成测试相关的其它任务

测试划分标准

按阶段（等级）划分

1 单元测试

2 集成测试

3 系统测试

4 验收测试



https://blog.csdn.net/iamhuanggua/article/details/80509973

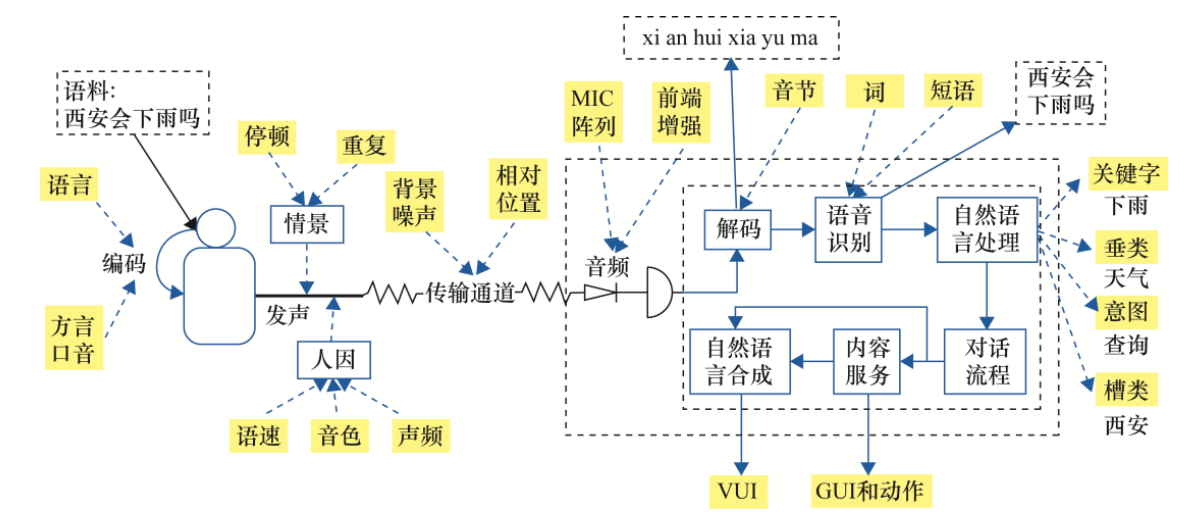
面临的挑战：

来源：

[1]朱少民.软件测试面临的挑战与发展趋势[J].测控技术,2020,39(01):1-4.

**AI 系统的测试**

对 AI 软件的测试，需要增加一个时间维度，因 为 AI 系统的能力是动态的，系统具有自身学习能 力，随着时间增长，其系统能力必须显著增强，例 如谷歌公司的 AlphaGo。针对这样的一个黑盒测试 过程，其测试代价不小，测试周期长，不容易实现 自动化测试。 像 AlphaGo 这样的系统还相对简单，而智能语 音应答系统的测试则更为复杂，需要模拟各种上下 文场景、用户的提问，如图 1 所示，很难判断其应 答能力是否有明显提升。虽然可以通过大量的应答 测试数据的统计结果确定语音识别、语义理解的正 确率，但其测试样本还是有较大的局限性。 深度神经网络学习算法，通常被认为是黑箱模型， 即其具有不可解释性，且鲁棒性差，结果不稳定。因 此，一般需要借助大量的数据或实验来进行普适性验 证，这与传统的测试相差甚远。测试（Test）从本质 上看就是“样本性的试验验证活动”，传统测试可以 借助“等价类划分”等类似方法来大幅降低样本量， 但对于 AI 软件，这类方法已经失效。AI 软件的测试 还依赖大数据，基于大数据的自动产生、分析、呈现 等技术，能有效地验证 AI 软件的合理性，但测试的 工作量较大。例如，为了测试 Numenta（工作模式更 接近人的大脑的 AI 软件），IBM 资深研究员 Winfried Wilcke 带领 100 人的团队对其算法进行测试。面对 俄罗斯的控制机器人集群的智能软件包 Unicum，又 如何测试呢？它被称为“机器人之上的机器人”， 可以独立分配集群内部的机器人角色、确定集群中 的核心、替换脱离的单位，还会自动占据有利位置、 搜寻目标，并在自动模式下向操纵员申请作战与摧 毁目标的许可。



**大数据的测试**

大数据测试，并不比 AI 软件测试简单，更准确 地说，大数据离不开 AI、AI 也离不开大数据。大数 据的体量大、多样化和高速处理所涉及的数据生成、 存储、检索和分析使得其测试也极具挑战。针对大数 据的测试，主要面临的挑战有： ① 难以确定其测试预言（Test Oracle）——测试 结果的判断准则，因为大数据的处理结果是否正确， 有时很难确定。 ② 大数据的多样性给测试数据准备、测试数据的 组合带来极大的工作量，如何高效或有效地进行测试， 也极具挑战。 ③ 数据经过采集、存储、检索和分析后，在体量 大、多样性和变化快、实时性的情况下，如何验证数 据的一致性和完整性，需要开发相应的工具。 ④ 在大数据的性能测试、安全性测试等方面，也 会遇到业务场景过多的困难。

**云服务测试**

今天的云服务包含私有云、公有云、混合云等不 同的形态，而其测试所面临复杂的系统架构、复杂的 配置、大量的计算或存储节点等方面所带来的挑战， 例如：配置参数的组合数能够达到几千万、几个亿甚 至几十亿，如何完成这巨量的组合测试？虽然可以采 用两两组合、三三组合等方法缩减组合数，但覆盖率 相对比较低，存在较大的质量风险。 其次，云服务的性能测试也很有挑战，主要体现在： ¬

* 海量云端压力的模拟将面临高昂代价、甚至无法模拟；
* 需要网络运营商、CDN 服务提供商的配合与 支持；
* 传统测试工具难以适应云计算平台的性能测试。

一般采用替代项——系统的可伸缩性（弹性）验证，以避免超大规模的性能测试。而且，云服务平台的容错性或可靠性测试也会面临挑战，例如：

* 错误类型过多；
* 错误信息表达不够充分；
* 业务错误码与 HTTP 错误码含义不匹配；
* 相同错误不同云产品表达不一致；
* 错误码超越可枚举集合。

因此诞生了一门新的工程方法（领域）——混沌工程。

**区块链测试**

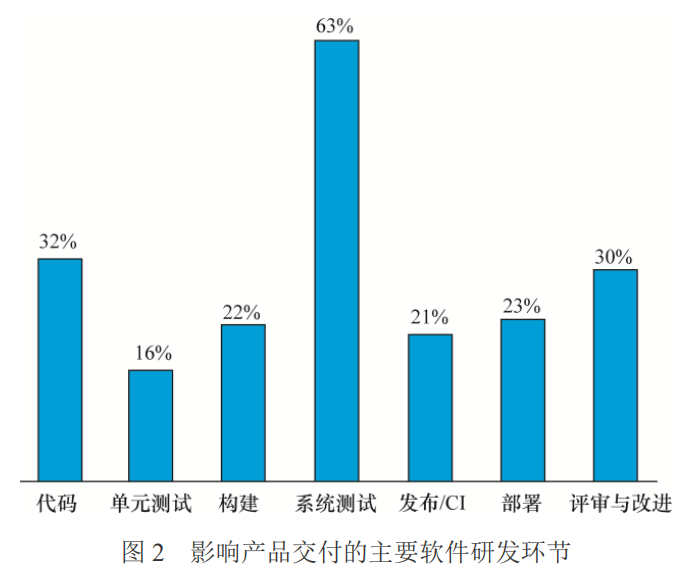
区块链一般分为公有链、私有链、联盟链等不同 类型，它们在管理、用户身份认证、节点数、共识机 制、智能合约等许多方面均有不同，其测试所面临的 挑战有所不同，但也面对一些共同的挑战。例如，安 全性测试就很具挑战性，因为安全性始终是“矛与盾” 的问题，没有绝对安全的技术、系统或平台，因此很 难把握其安全性测试的度，更何况区块链技术平台还 涉及基础设施（网络、存储设备等）安全性、加密算 法安全性、网络协议安全性、共识机制安全性（包括 共识算法漏洞）、智能合约安全性（如代码实现中的 安全漏洞）、应用安全性（包括 Web/移动客户端软件、 数字钱包等）、身份认证与鉴别等。 作为一个区块链的系统，则是一个完全去中心化 的分布式网络——P2P 网络，而且这个网络很可能跨 越多个子网、多个数据中心、多个运营商甚至多个国 家，其边界是模糊的。但区块链测试，不仅涉及前端 API 与某个区块链节点之间的测试，还涉及大量区块 链节点与节点之间的测试。因此，边界模糊、网络及 其交易的复杂性都给系统的功能测试、兼容性测试、 数据一致性验证、安全性测试和性能测试等带来挑战， 特别是在共识算法的合法性、完整性、可终止性的验证方面。另外一个挑战来自智能合约的验证，因为智 能合约执行过程中，一切均需听命于事先设定好的代 码，而且智能合约类似法律，事先设定好的代码一旦 上线就不能轻易修改。如何确保智能合约百分之百正确？测试的一项重要原则就是测试是不能穷尽的，测 试不能保证没有缺陷或百分之百正确。

**物联网测试**

在工业 4.0 和 5G 时代，万物互联变得司空见惯， Gartner预计今年将有超过200亿台物联网设备投入使 用。当这么多设备接入物联网中，需要实现异构互联 互通，也意味着物联网会支持更为广泛的工业协议， 如 MQTT、OPC UA、ModBus-TCP 等行业标准，并与现 有的互联网系统连接起来，形成规模更大的系统，这必 然给软件测试带来新的挑战，主要体现在如下方面。 ① 测试环境复杂性，如异构互联、M2M 互联、 软硬件结合紧密等，需要构建全新的仿真实验环境等。 ② 高可靠性测试，因为系统涉及生命安全，需要 确保系统的高可靠性，包括可靠性模拟、可靠性估算 和可靠性监控等。 ③ 实时性，如时间敏感网络（TSN）、实时数据 采集、实时数据同步传输等，对测试环境、测试工具 都有更高的要求。 ④ 兼容性测试，如 EMI/EMC 共存测试，以及各 种通信协议、数据采集方式等验证。 ⑤ 网络安全策略的测试、配置组合测试，这方面 测试工作量也很大，如何提高测试的有效性和高效性， 也有很大挑战。 ⑥ 另外还应对系统的可伸缩性、可扩展性、协同 感知、大数据处理性能、智能特性等进行验证。

**敏捷测试**

快速迭代、快速交付，迫使人们能高效地完成软 件测试，因此人们寄希望于自动化测试，毕竟工具的 执行效率高。但从自动化测试的调查结果来看，大多 数软件公司的自动化测试实际做得还不够好，只有 4% 的公司能做到 90%以上的测试实现自动化，所以测试 常常是敏捷开发的最大瓶颈，如图 2 所示。 自动化测试做得不够好有多方面的原因，不同的 团队有不同的原因，但根据调查，主要的原因有：需 求变更频繁、产品的可测试性差、单元测试不足、测 试人员能力不足、测试工具不够强大等。

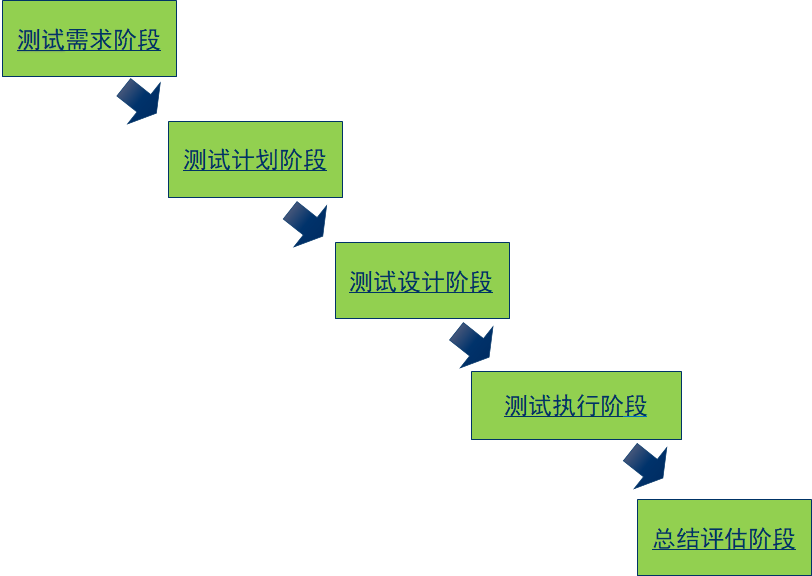


阅读下面白盒测试和黑盒测试相关资料（或查阅其它相关资料），深入理解白盒测试和黑盒测试，总结其特点 （保存到每个小组选定的协作开发平台上）

WhiteBox.pdf

BlackBox.pdf

测试项目周期图示：



**1. 黑盒测试**

黑盒测试： 又称功能测试、数据驱动测试或基于规格说明的测试，是通过使用整

个软件或某种软件功能来严格地测试,，而并没有通过检查程序的源

代码，或者很清楚地了解该软件的源代码程序具体是怎样设计的。

测试人员通过输入他们的数据看输出的结果，从而了解软件怎样工

作。应该更适合测试网页游戏。

黑盒测试基本观点： 任何程序都可看作从输入定义域到输出值域的映射

黑盒测试的两个重要的优点

（1）黑盒测试与软件具体实现方式无关。若软件实现方式如果发生了变更、修改但功能测试不变的话，仍可用原来的测试用例

（2）在进行软件开发的同时，也可进行软件黑盒测试用例的设计，这样可节省时间成本

黑盒测试常用工具： QACenter、WinRunner

**2. 白盒测试**

白盒测试： 是指实际运行被测程序，通过程序的源代码进行测试而不使用用

户界面。这种类型的测试需要从代码句法发现内部代码在算法、

溢出、路径和条件等方面的缺点或者错误，进而加以修正。

又称结构测试、透明盒测试、逻辑驱动测试或基于代码的测试。

白盒测试是一种测试用例设计方法，盒子指的是被测软件。

白盒测试方法： 逻辑驱动、基路测试等，主要用于软件验证。

（理解）白盒测试必须遵从的规则如下

* 一个模块中所有的独立路径都需至少得到一次测试。
* 所有逻辑值的真与假情况都需要被测试到。
* 为了保证程序结构的有效性，需要检查程序的内部逻辑结构。
* 在程序的上、下边界与可操作范围内都能保证循环的顺利运行。
* 白盒测试工具一般是对代码进行测试，可分为动态测试工具和静态测试工具。

静态测试工具： 直接对代码进行分析，不需运行

动态测试工具： 一般采用“插桩”方式，向代码生成的可执行文件中插入监测代

码，来统计程序运行时的数据

白盒测试工具： Jtest、Jcontract、C++Test、codeWizard、Insure++

白盒测试工具及其特点：

1、Jtest

Jtest是一个代码分析和动态类、组件测试工具，是一个集成的、易于使用

和自动化的Java单元测试工具。它增强代码的稳定性，防止软件错误。

2、Jcontract

Jcontract在系统级验证类/部件是否正确工作并被正确使用。Jcontract是

个独立工具，在功能上是Jtest 的补充。

3、C++Test

C++Test可以帮助开发人员防止软件错误，保证代码的健全性、可靠性、可

维护性和可移植性。

4、CodeWizard

CodeWizard是代码静态分析工具，先进的C/C++源代码分析工具，使用超过

500个编码规范，自动化地标明危险。

5、Insure++

Insure++是一个基于C/C++的自动化的内存错误、内存泄漏的精确检测工

具。Insure++能够可视化实时内存操作,准确检测出内存泄漏产生的根源。

Insure++还能执行覆盖性分析,清楚地指示哪些代码已经测试过。

**3. 黑盒测试和白盒测试的比较**

黑盒测试和白盒测试的区别：

|  |  |  |
| --- | --- | --- |
|  | 黑盒测试 | 白盒测试 |
| 从定义上 | 黑盒测试着重测试软件功能，它并不涉及程序的内部结构和内容特性。黑盒测试并不能取代白盒测试，它与白盒是互补的测试方法，它很可能发现白盒测试不易发现的其他类型错误。 | 白盒测试需要从代码句法发现内部代码在算法，溢出，路径，条件等等中的缺点或者错误，进而加以修正。 |
| 从测试目的上 | 黑盒测试的目的是检测是否有不正确或遗漏的功能；数据或者参数上，输入能否正确接收；是否有数据结构错误或外部信息访问错误；性能上是否能够满足要求；是否有初始化或终止性错误。 | 白盒测试的目的是通过在不同点检查程序的状态，确定实际的状态是否与预期的状态一致，而不顾它的功能。 |
| 测试方法上 | 黑盒测试是穷举输入测试， | 白盒测试是穷举路径测试， |
| 这两种方法是基于完全不同的观点，反应了事物的两个极端，它们各有侧重和优势，但不能彼此替代。在现代的测试理念中，这两种测试方法不是截然分开的，而是交叉使用。 | |
| 已知产品因素 | 已知程序的功能需求、设计规格 | 已知程序的内部结构 |
| 检查测试的主要内容 | 黑盒测试主要检查的内容包括但不限于：  1）功能是否都满足需求；是否有功能出现缺陷。  2）接口上是否能正确接收输入；输出结果是否正确。  3）是否有数据结构信息或者外部信息访问错误。  4）是否有初始化或终止性错误。 | 白盒测试主要检查的内容包括但不限于：  1）所有程序模块的独立路径都需要至少被测试一遍。  2）所有的逻辑判定的真值与假值都需要至少被测试一遍。  3）在运行的界限内和循环的边界上执行循环体。  4）测试内部的数据结构是否有效。 |
| 静态测试方法 | 检测产品需求文档、用户手册、帮助文件等。 | 走查、复审、评审程序源代码、数据字典、系统设计文档、环境设置、软件配置项等 |
| 动态测试方法 | 通过数据输入并运行程序来检验输出结果，如功能测试、验收测试和一些性能测试（兼容性、安全性）等 | 通过驱动程序来调用。如进行单元测试、集成测试和部分性能测试（可靠性、恢复性）等。 |

**4. 黑盒测试与白盒测试的优缺点**

|  |  |  |
| --- | --- | --- |
|  | 黑盒 | 白盒 |
| 优点 | 对于子系统甚至系统效率要比白盒测试高 | 能仔细考虑软件的实现 |
| 测试人员不需要了解实现的细节（特定编程语言） | 可检测代码中的每条分支和路径 |
| 测试人员和编程人员彼此独立 | 揭示隐藏在代码中的错误 |
| 从用户的角度进行测试很容易理解和接受 | 对代码的测试比较彻底 |
| 有助于暴露规格的不一致或有歧义的问题 |
| 测试用例可以在规格完成后马上进行 |
| 缺点 | 只有一小部分输入被测试到，要测试每个可能的输入几乎不可能 | 昂贵 |
| 没有清晰、简明的规格，测试用例很难设计 | 无法检测代码中遗漏的路径和数据敏感性错误 |
| 如果测试人员不被告知开发人员已经执行过的用例，在测试数据上会存在不必要的重复 | 不验证规格的正确性 |
| 有很多程序路径没有被测试到 |
| 不能直接针对特定程序段测试，而这些程序段可能很复杂，有可能隐藏更多的问题 |
| 大部分和研究相关的测试都是直接针对白盒测试的 |

阅读下面单元测试相关工具的介绍（或查阅其它单元测试工具相关资料），学习单元测试工具的使用

JUnit-Tutorial.pdf

junit\_tutorial.pdf

nunit.pdf

Embunit User Guide.pdf

单元测试 Unit Test

单元测试的定义：

A unit test is a test written by the programmer to verify that a relatively small piece of code is doing what it is intended to do. They are narrow in scope, they should be easy to write and execute, and their effectiveness depends on what the programmer considers to be useful. The tests are intended for the use of the programmer, they are not directly useful to anybody else, though, if they do their job, testers and users downstream should benefit from seeing less bugs.

上面的定义强调了

1，UT应由开发人员写，给开发人员使用。

2，UT测试的应该是小段代码，目的是开发人员确定源代码做了希望它做的事。

TestNG的作者如何概括UT的：

unit testing (testing a class in isolation of the others)

首先他强调了，一个UT case只能针对一个类。不应该在一个UT case中测试一个以上的类。

其次一个重点是：一个类的UT必须是完全独立的，不应再和其他部分代码有任何交互。这也是我们要使用Mockito等mock框架的原因，这样在测试代码时，当必须和其他对象交互时，就使用mock对象来保持UT的独立性。

此外，UT 可以是自动的，也可以是手动的。

单元测试的工具：

熟知的JUnit，以及TestNG；.net专用的NUnit；VS2010开始直接提供的测试框架。

c/c++专用的CppUnit，PHP专用的PHPUnit。

Mockito，easyMock等mock框架也是重要组成部分。

另外，单元测试有一个覆盖率的问题。就是指一个UT调用了被测试类中多少代码。一个不负责任的程序员写的UT也许只测试了类中的一个方法，而负责的程序员写的UT应该覆盖掉被测试类中所有主要方法。

Cobertura是自动检测UT覆盖率并生成报告的一个开源工具。 将这个工具集成到每天的daily build中是一种很好的做法。

要求覆盖率要达到100%是无理且无意义的，通常80~90%就行了。原因有两点：

1，是否UT必须覆盖所有的代码？

JUnit官网的回答：  
No, just test everything that could reasonably break.

Be practical and maximize your testing investment. Remember that investments in testing are equal investments in design. If defects aren't being reported and your design responds well to change, then you're probably testing enough. If you're spending a lot of time fixing defects and your design is difficult to grow, you should write more tests.

If something is difficult to test, it's usually an opportunity for a design improvement. Look to improve the design so that it's easier to test, and by doing so a better design will usually emerge.

2，对于一般项目，到达到100%的覆盖率基本不可能。

尽管现在有各种测试框架，各种mock框架，但在很多可能的情况下都会遇到难于，甚至无法测试的代码。

例如private方法，唯一测试的方式就是利用反射机制。

如果测试覆盖率变成了一个目标，而不是提高代码质量，防范bug的手段，那就是本末倒置了。

但是，如果为了方便测试而修改源代码，并不是不可理喻的行为。如果发现代码难于测试，则有可能是代码存在一些问题。

例如，在另一篇关于Mockito的文章中提到了，为了将mock对象传递到被测试的对象中，需要源代码提供get/set方法，否则测试起来就很麻烦。

在实践中慢慢会发现，代码不够独立，缺乏应有的get/set方法，结构不合理，方法应该被重构等等问题，会导致难于被测试。

因此UT也是对源代码的一次review。一个有责任心的程序员，应该从UT中看到更多。永远记住一条：我们不是在为了测试而测试，我们应该思考的更多。

3, 对于每一个测试方法来说，所谓的独立到底是什么？

最近遇到了关于UT的这样一个问题：每一个测试方法到底要多么“独立”？

例如，在一个被测试类中有两个方法：

public class app{

public void method1(){

String testString = method2(); //在方法一中调用了方法二。

}

public String method2(){

}

}

于是问题是，当我们写第一个方法method1的测试方法method1Test时，我们应该mock的是什么？我们应该mock方法二？还是不需要mock？

也就是说，当我们测试方法一时，应该把方法二看做外部的东西，通过mock方法二，仅仅测试方法一。

或是不使用mock，直接调用方法一(实际同时也调用了方法二)。因此相当于同时测试了方法一和方法二的行为。

第二种做法，同时测试了方法一和二。但是在一个类中，各个方法本来就不是完全独立的。

因此有成员变量的存在。成员变量的值天生的就可以在各个方法中传递。

因此结论是：UT的独立范围，永远是一个类。测试时，当调用了其他类的代码时，就应该使用mock将被测试类和其他类隔离开。

测试时，如果调用了同一个类的其他方法时，是不需要使用mock的。因为调用的是同一个类中的代码。

集成测试 Integration Test

集成测试的定义：

An integration test is done to demonstrate that different pieces of the system work together. Integration tests cover whole applications, and they require much more effort to put together. They usually require resources like database instances and hardware to be allocated for them. The integration tests do a more convincing job of demonstrating the system works (especially to non-programmers) than a set of unit tests can, at least to the extent the integration test environment resembles production.

因此，

1，通常会在指定的环境运行集成测试。

2，集成测试的目的旨在测试各个组件间是否能互相配合，正常工作。和UT一样，集成测试也是为了看代码是否按"设计或期望的方式"工作。

3，集成测试的范围比较宽泛：范围可以小到，如果在一个UT case中涉及了多个类，就可以认为这是集成测试。大到整个系统，从后台到前端所有组件。Integration tests can be performed at the unit level or the system level.

集成测试往往会涉及外部组件，如数据库，硬件，网络等等。

4，集成测试通常应由专门的测试人员完成。

HP's (mercury) QTP or Borland's Silktest 都可以做自动化的集成测试和回归测试。

在敏捷开发的今天，一个概念变得更重要了：CI (Continuous Integration)

本文并不打算探讨CI，因为这又是一个很大的话题。CI 和集成测试经常结合的很紧密。

CC(CruiseContoller), 是现在常用的CI工具之一。

Hudson 是另一个较新的CI工具。<http://hudson-ci.org/>

冒烟测试 Smoke Test

Smoke Test是从电子硬件测试来的，表示第一次对硬件原型prototype加电时，如果硬件冒烟了，就表示产品有问题。

从smoke test名字的来由我们就可以看出，smoke test是比较初级的测试(a quick test)，仅仅是为了检查各个组件是否能一起工作，而并不去深究功能上是否正确。

A simple integration test where we just check that when the system under test is invoked it returns normally and does not blow up.

注意smoke test一般是大范围的集成测试。通常可以是整个系统/端到端的测试。

例如，一个项目：客服人员在终端上输入用户的信息，最终用户可以在互联网上查询到自己的信息。

此时的smoke test就可以是在终端上输入某个用户的信息，然后查询通过web程序是否可以找到这个用户的信息，而不必关注该用户信息的正确性。

因此有人认为它是测试的第一步：first tests on which testers can conclude if they will continue testing. 但并非是必须的。

回归测试 Regression Test

A test that was written when a bug was fixed. It ensure that this specific bug will not occur again. The full name is "non-regression test".

A regression test re-runs previous tests against the changed software to ensure that the changes made in the current software do not affect the functionality of the existing software."

上面两种观点并不一致：一种认为回归测试是为了覆盖fix的bug，另一种认为回归测试是覆盖新添加的功能。

但是我们可以看到这两种说法的统一之处：

回归测试是为了覆盖系统新发生的变化而进行的测试。

回归测试可以由测试人员编写，也可以由开发人员编写。

形式上可以就是一个覆盖新功能或者bug的UT，或者测试人员写的专用测试工具。

端到端测试 End-To-End Test

validates the entire application to ensure that it satisfies previously established acceptance criteria and performs as an integrated system. The purpose of system testing is not to test all possible positive and negative conditions (reserved for functional and integration testing), but to instead execute business functions and infrastructure management (i.e.; batch processing, system security features, backup and recovery etc.) in an isolated and controlled environment to validate that a quality system is ready for production.

因此端到端测试强调的是全面的，包含硬件环境等等的测试。

最后一句话很重要：是为了上production。

因为覆盖面广，因此一般都是人工测试。通常也没有专用工具来做端到端测试。

功能测试 Functional Test

Functional tests are related to integration tests, but refer more specifically to tests that test an entire system or application with all of the code running together, almost a super integration test.

非功能测试 Non-funtional Test

非功能测试在我公司，主要指除程序主要功能以外的附属功能。如GMI，log，report，failover等，为了监控程序运行，实现程序稳定性而附加的功能。

acceptance tests

总的来说，acceptance tests 和 Functional Test是非常接近的，甚至有些地方认为二者完全一样。二者的异同在于：

functional testing: This is a **verification** activity; did we build the thing right? Does the software meet the business requirements?

For this type of testing we have test cases that cover all the possible scenarios we can think of, even if that scenario is unlikely to exist "in the real world". When doing this type of testing, we aim for maximum code coverage. We use any test environment we can grab at the time, it doesn't have to be "production" caliber, so long as it's usable.

acceptance testing: This is a **validation** activity; did we build the right thing? Is this what the customer really needs?

This is usually done in cooperation with the customer, or by an internal customer proxy (product owner). For this type of testing we use test cases that cover the typical scenarios under which we expect the software to be used. This test must be conducted in a "production-like" environment, on hardware that is the same as, or close to, what a customer will use. This is when we test our "ilities":

* Reliability, Availability: Validated via a stress test.
* Scalabilitiy: Validated via a load test.
* Usability: Validated via an inspection and demonstration to the customer. Is the UI configured to their liking? Did we put the customer branding in all the right places? Do we have all the fields/screens they asked for?
* Security **(aka, Securability, just to fit in)**: Validated via demonstration. Sometimes a customer will hire an outside firm to do a security audit and/or intrusion testing.
* Maintainability: Validated via demonstration of how we will deliver software updates/patches.
* Configurability: Validated via demonstration of how the customer can modify the system to suit their needs.

简单的说，functional testing倾向测试软件的所有具备的功能。倾向于是否完全了所有的需求。

acceptance testing倾向测试的是尽量真实的用户体验。测试是否完成用户的实际需求，在完全和production相同或尽量类似的环境中。

同时，二者都应该是完全的黑盒测试。

下面这句话很重要：

Acceptance Tests/Criteria (in Agile Software Development) are usually created by business customers and expressed in abusiness domain language. These are high-level tests to test the completeness of a user story or stories 'played' during any sprint/iteration.

a business domain language指非开发人员可以理解的类自然语言。现在比较流行的框架是JBehave。

Acceptance test cards are ideally created during sprint planning or iteration planning meeting, before development begins so that the developers have a clear idea of what to develop. Sometimes (due to bad planning!) acceptance tests may span multiple stories (that are not implemented in the same sprint) and there are different ways to test them out during actual sprints. One popular technique is to mock external interfaces or data to mimic other stories which might not be played out during an iteration (as those stories may have been relatively lower business priority). A user story is not considered complete until the acceptance tests have passed.

也就是说在一个sprint中，如何确定开发人员是否完成了一个功能？标准就是对应的acceptance test通过。

这也就是Acceptance-Test Driven Development的来由。

再总结一下白盒和黑盒测试：

White box testing means that you know the input, you know the inner workings of the mechanism and can inspect it and you know the output.

With black box testing you only know what the input is and what the output should be.

因此通常情况下，UT是白盒测试，但是有时候也可以是黑盒。例如现在流行的测试驱动开发，UT是根据需求先于代码被创建出来的，此时的UT只知道我们有什么，然后希望看到什么，所以就是黑盒。

而其他的测试则大部分是黑盒测试。但是集成测试和回归测试也可以是白盒。

前面提到的，测试对源代码覆盖率的问题，理论上只有对白盒测试，这个指标才有意义。对与黑盒测试，代码覆盖率根本就不make sense。

对与某些领导提的，“我们这个集成测试要100%覆盖源代码”，基本就是脑子被门挤了。

但是需要说明的是，黑盒测试一样可以计算代码覆盖率，虽然我认为没有什么意义。

使用cobertura注入后，就可以方便的得到任何一个集成测试对源代码的覆盖率。具体可以参考我写的cobertura的介绍。

最后提一下目前流行的开源mock框架(因为我是developer所以尤其关注UT相关的工具)

* 较早的时候EasyMock是最流行的工具之一。

现在来看，EasyMock由于出现的较早，因此功能也比较少。使用EasyMock会遇到很多不容易mock的情况。 

* Mocktio

EasyMock之后流行的mock工具。特点是句法结构清晰，易理解。文档比较全，由于比较流行所以遇到问题容易找到解答。

Mockito足以解决大部分UT的mock需求。总得来说是非常不错的mock工具。

* PowerMock

其实这个工具是在EasyMock和Mockito上扩展出来的，目的是为了解决EasyMock和Mockito不能解决的问题。

我在介绍PowerMock的文章中详细说明了这个工具可用于哪些特殊mock需求。

缺点是该工具文档较少，例子不全，使用时遇到问题不容易解决。同时PowerMock需要和EasyMock或Mockito配合一起使用。

因此必须先掌握EasyMock或Mockito。

* Jmokit

http://code.google.com/p/jmockit/

另外一个类似PowerMock的工具。这个工具我不准备介绍了(最近看mock工具看的审美疲劳了)。

该工具实现和PowerMock相同的功能因此如果不喜欢PowerMock可以选择这个工具。

相对PowerMock， 这个工具文档比较全，适于学习。